SOUTHERN MAINE COMMUNITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSCI 290 DATA STRUCTURES

Manual

NOAA Weather App

Advisor: Dr. Anne Applin

SOUTH PORTLAND, MAINE, APRIL 2023

Contents

1 How to Use Application

This section will briefly introduce you to the application, it's use cases, and most importantly, how to prepare your data for the applications built in CSV scanner.

1.1 Preparing a Comma Separated Values File for Use With Application

First, begin by navigating to the NOAA website `https://www.ncdc.noaa.gov/` and click on > Home > Climate Data Online > Data Tools > Select a Location. Or, alternatively, you can click this link to be brought there.

Next, from **Select a Dataset** you want to choose **Daily Summaries** and search for your desired location in whichever way is easiest for you. I recommend searching by **Zip Code** and using airport data as they usually collect the widest range of stats.

Finally, once you have found your desired station, under **Data Types** you're going to want to ensure that the data comes with the following attributes: Station, Name, Date, ACSH, FRTH, SNOW, SNWD, TMAX, TMIN, WSFG, WT... Moreover, it's critical that under **Custom Flag(s)** you enable the Station Name, Geographic Location, and Include Data Flags attributes. For a more detailed overview of what these attributes accomplish take a look at NOAA's documentation.

1.2 Using Dataset with the Application

When you open the application, you will begin in the **Home** tab. To begin loading the CSV navigate to the **Load File** tab. Once there, click **Open File** and navigate to the desired CSV. Following this step, you will recieve a database notification based on the success or failure of dataset parsing.

1.3 Navigating *View Data* and *Sorts and Algorithms* Tabs

Once the data base is initialized, you can navigate to the **View Data** tab to see the sorted data in a table that contains parameters about a given dataset. Moreover, if you enable **Display All Fields** and click the **Refresh** button, you will see all of the dataset parameters for a given `Record` object (see section 3.1 for more information about Record objects).

Under the **Sorts and Algorithms** tab you will have access to a plethora of functionality. First, and most importantly, you will have three main specifiers to choose from before you sort: the **Sort By...** parameter, **Algorithm** parameter, and, if desired, the **Trim Data Length** parameter. The **Sort By...** parameter allows a user to specify which data parameter they'd like to sort the objects by, the **Algorithm** parameter chooses a given algorithm to sort the data set with, and the **Trim Data Length** field can be used to trim a given data set down to a specific length.

Finally, when it comes time to sort the data, you can choose to sort using the selected algorithm and data specifier using the **Sort!** button, or alternatively, you can run all the sorts available in the application using the **Run All Sorts** button (*NOTE:* ***Run All Sorts*** *will sort by the selected* ***Sort By...*** *parameter just like* ***Sort!*** *would*). Once the sort(s) is/are complete, the run statistics will be shown in the window on the right-hand side of the application. If you are gathering mass statistics, you must use the **Reload Dataset** button to refresh the arrays before you use **Run All Sorts** again.

2 System Error Events

This section covers all system exit codes that are baked into the program. These exit points are visible only when running the application in development mode or by running the GUI version through an IDE where the console is still active.

- **Exit location 1**: this occurs only if program arguments, in development mode, were not set (I.E., no file path was set).
- **Exit location 2**: this occurs when the program cannot find specified file path. When using the development version, this will occur if the file path argument is non-existent. When using the GUI version, this will occur if user enters a non-existent directory.
- **Exit location 3**: this occurs when the dependency `CSVReader` failed to initialize. Normally, this should only ever happen if Maven fails to find dependencies. This would never occur in the shipped version of the application.
- **Exit location 4**: this occurs when `CSVReader` initializes properly, reads through data, but a `datatype` object was not properly constructed, or, if `CSVReader` went out of bounds and pointed to a null cell.

3 Analysis of Sorts

In the world of software engineering, we regularly make predictions using Big O to project the worst-case, average case, and best case performance of our algorithms. In this section we will cover the actual performance of the algorithms used in the project, the predicted performance, and most importantly, the algorithms themselves.

3.1 Collection and Creation of Results

The application begins by populating a `ArrayList` of `Record` objects using a CSV from the National Oceanic and Atmospheric Administration. Once the CSV has been fully scanned, the `ArrayList` is moved into a standard array of records. `Record` objects contain parameters `TMAX` (maximum temperature for that day), `TMIN` (minimum temperature for that day), `PeakWindSpeed` (fastest recorded wind speed for that day), etc... When it comes time to sort, the program begins by making a duplicate of the pre-sorted array. After that, the program calls its `Timer` class and logs the start time, runs the sort, and then the `Timer` class logs the end time. From here, the program handles reverting the sorted array back to the pre-sorted array and the next sort in the `Stack` is queued and timed in the same fashion.

Although a user can sort by any `Record` object parameter, all of these test results were compiled using the `TMAX` parameter to ensure the lowest overhead (TMAX is just an int).

3.2 Analysis of Predicted vs Actual Number of Comparisons

To begin, let us consider Figure 1. The graphs x-axis is measured in N (number of items in array), and it's y-axis is the number of comparisons made. The blue bar highlights **actual** results and the yellow bar highlights pre-calculated **projections**. As we can see in each series, the sorts consistently out performed our pre-calculated projections by 50% at best (dual-pivot quick sort and merge-sort) and 10% at worst (insertion sort). These facts illustrate that it's highly improbable that any given data set will ever be entirely unsorted, rather, data sets are likely to be at least partially naturally sorted.

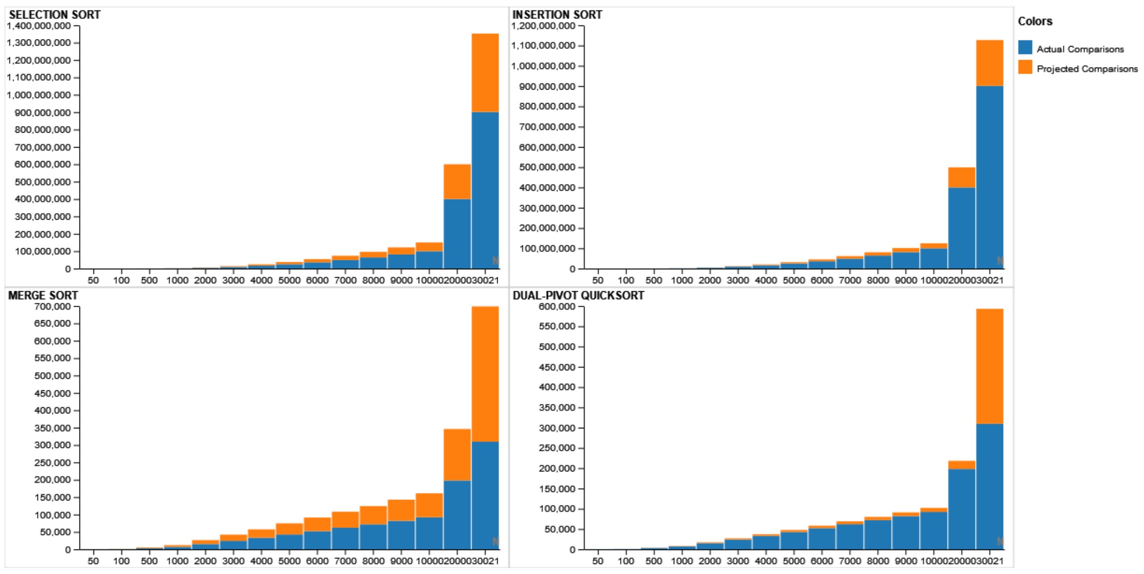

Figure 1: Projected vs Actual Comparisons

3.3 Analysis of Sort Performance

As we can see in Figure 2, all four algorithms show comparable run times (less than 0ms) under $N \approx 500$. However, performance for selection and insertion sorts begin to degrade heavily above $N \approx 500$ and both insertion and selection algorithms share identical run times from $N = 100$ through $N \approx 1000$. Thus, we can conclude that either of these four algorithms would prove to be adequate to sort lists under size $N = 500$ if run time constraints illustrate a need for sub 0ms sorts.

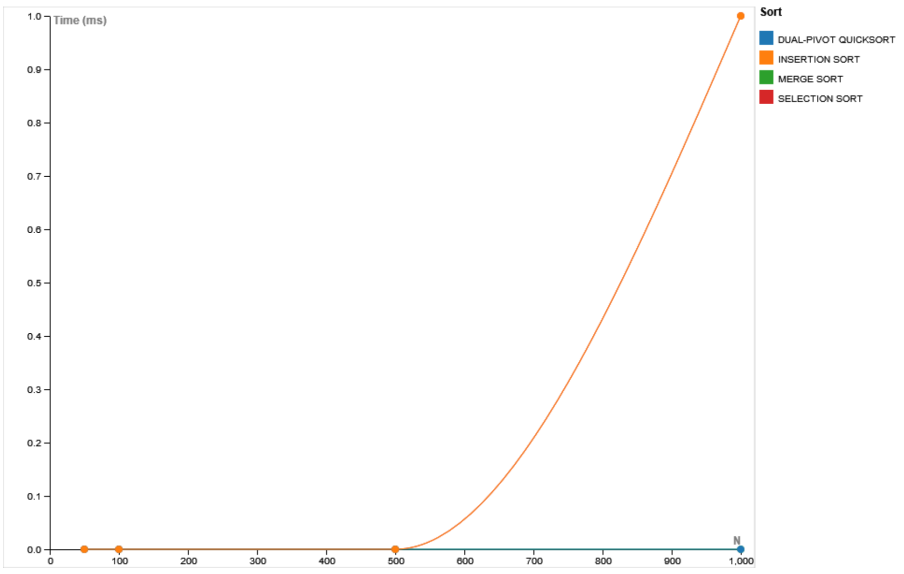

Figure 2: Time vs N

The next algorithm to reach run times over 0ms is merge sort. As we can see in Figure 3, merge sort was consistently returning a sorted array under 0ms until $N \approx 4,500$. Comparatively, we can see that merge-sort and dual-pivot quick sort were completing their sorts under 0ms at $N = 4000$ while both insertion and selection sorts were completing in ≈ 28ms. From this information alone, we can clearly see logarithmic (merge sort and dual pivot quick sort) growth vs exponential (insertion sort and selection sort) growth.

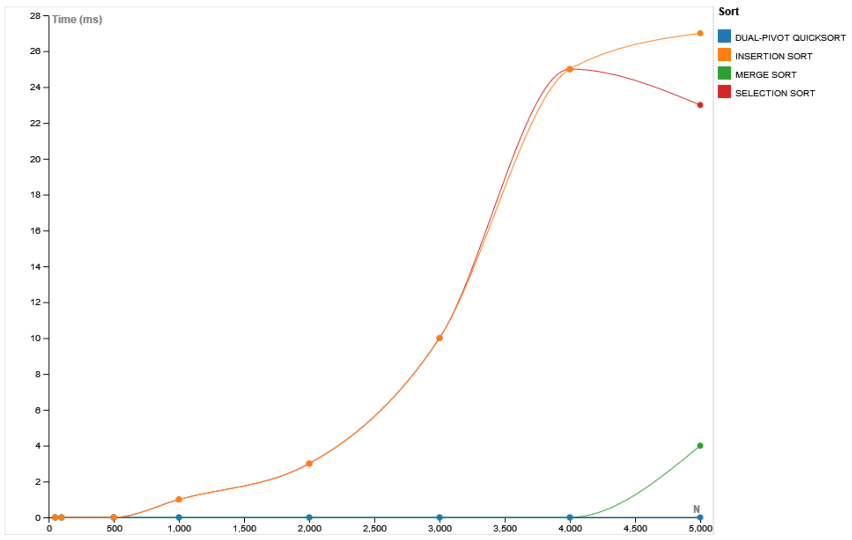

Figure 3: Time vs N