SOUTHERN MAINE COMMUNITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSCI 290 Algorithms and Data Structures

Manual

NOAA Weather Analysis Application

Advisor: Dr. Anne Applin

SOUTH PORTLAND, MAINE, MAY 2023

Contents

1 Application Use

This section will briefly introduce you to the application, its use cases, and how to prepare 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. Alternatively, you can click this link to be brought there.

Next, from **Select a Dataset**, 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 statistics.

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 land 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 has been 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. 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; 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 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 integrated 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, while running in the development version, 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 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 graph's x-axis is measured in N (number of items in array), and its 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 outperformed our pre-calculated projections by 50% at best (dual-pivot quick sort and merge-sort) and 10% at worst (insertion sort). These visualizations 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.

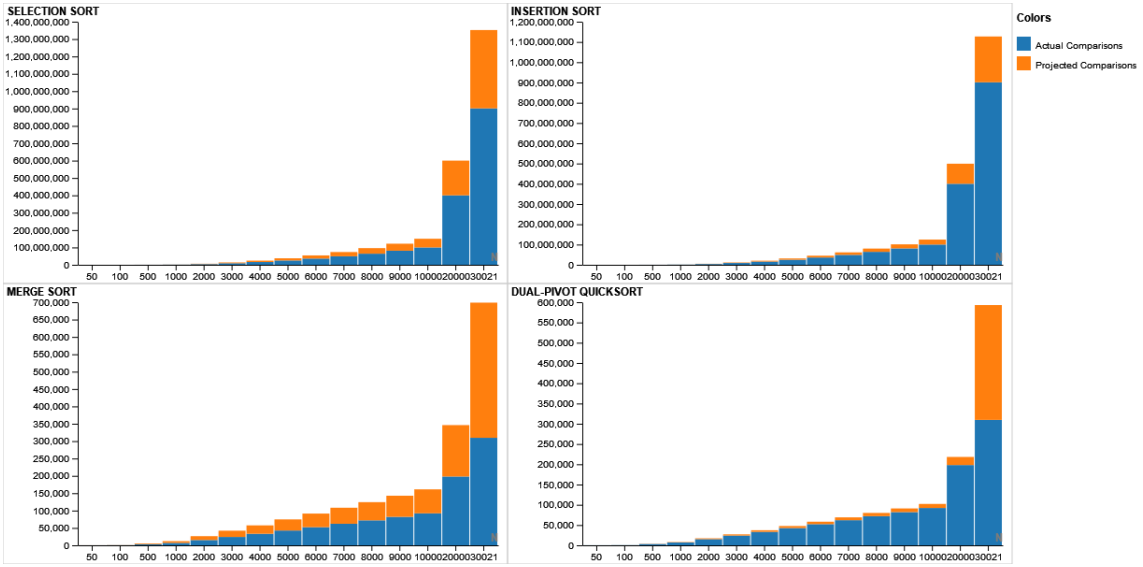

Figure 1: Projected vs Actual Comparisons

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.

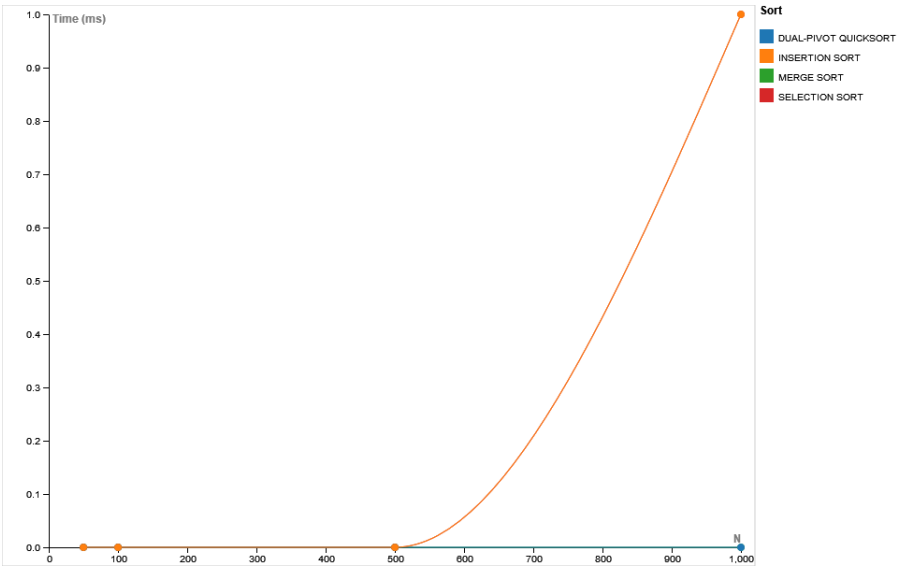

Figure 2: Time vs N

The next algorithm to reach run times over 0ms is merge sort. As we can observe 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 $\approx$ 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.

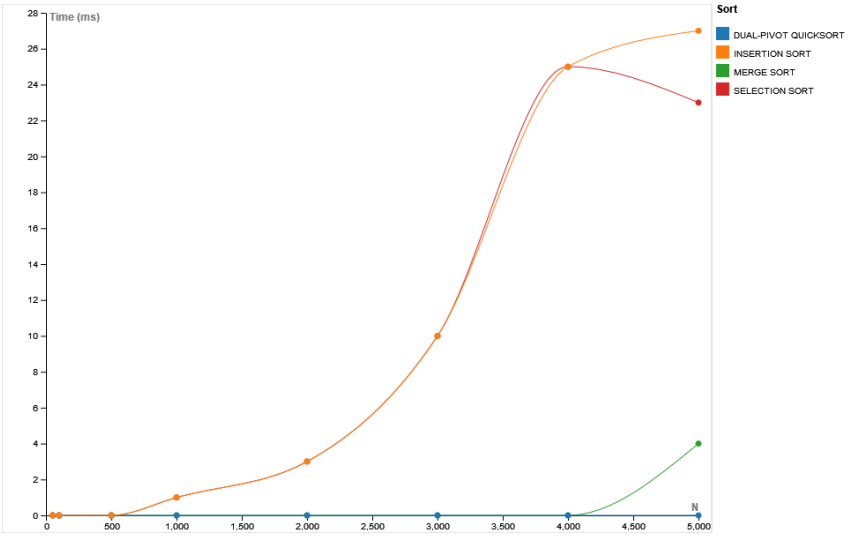

Figure 3: Time vs N

The last algorithm to reach run times greater than 0ms was dual pivot quick-sort. Once *dpqs* (dual pivot quick-sort) reached a run time of ≈ 1ms, insertion sort was nearing ≈ 2000ms, selection sort was nearing ≈ 3000ms, and merge sort was nearing ≈ 8ms. Thus, we can gather that out of these four algorithms – even though both merge sort and *dpqs* scale logarithmically – the speed and scalable nature of *dpqs* is simply unmatched when comparing it against its counterparts.

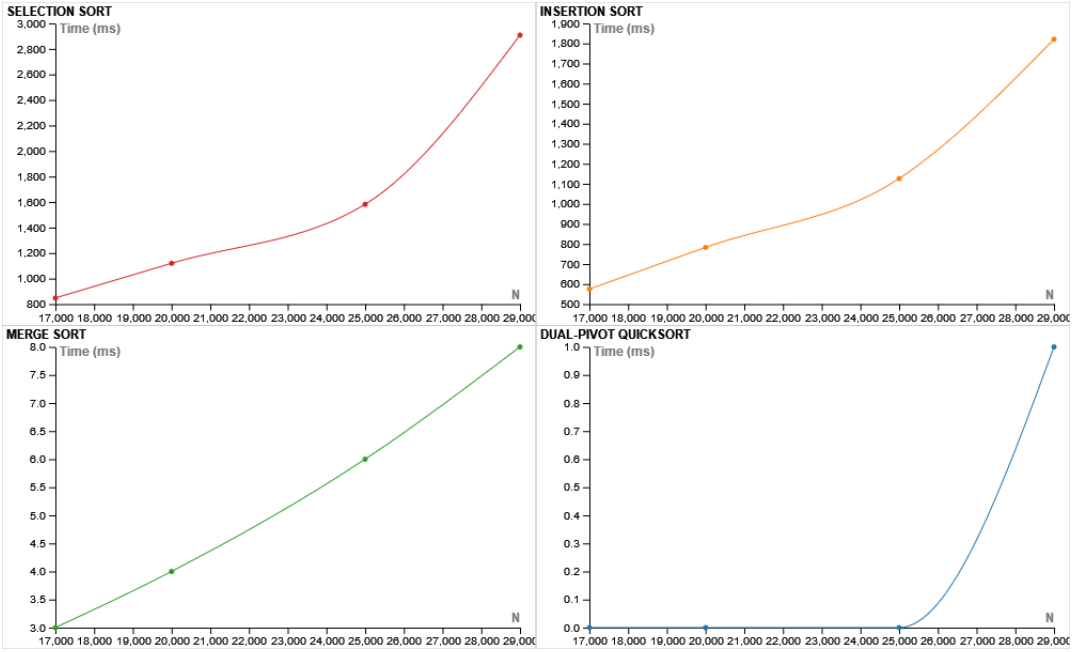

Figure 4: Independent Scale Series, *N vs Time*

3.3.1 Sort Stability

Continuing forward about the issue of stability, we can see that in Figure 5, selection sort had the largest instability resulting in a shift of 200ms, insertion sort comes next with a max shift of $\approx$ 100ms, then comes merge sort with a max shift of 8ms. *dpqs* illustrated no signs of instability as its run times were consistently under $\approx$ 1ms until $N \approx 29,000$. It's important to remember that these graphs **do not** share a shared scale, thus, the actual stability of merge sort is a lot better than illustrated.

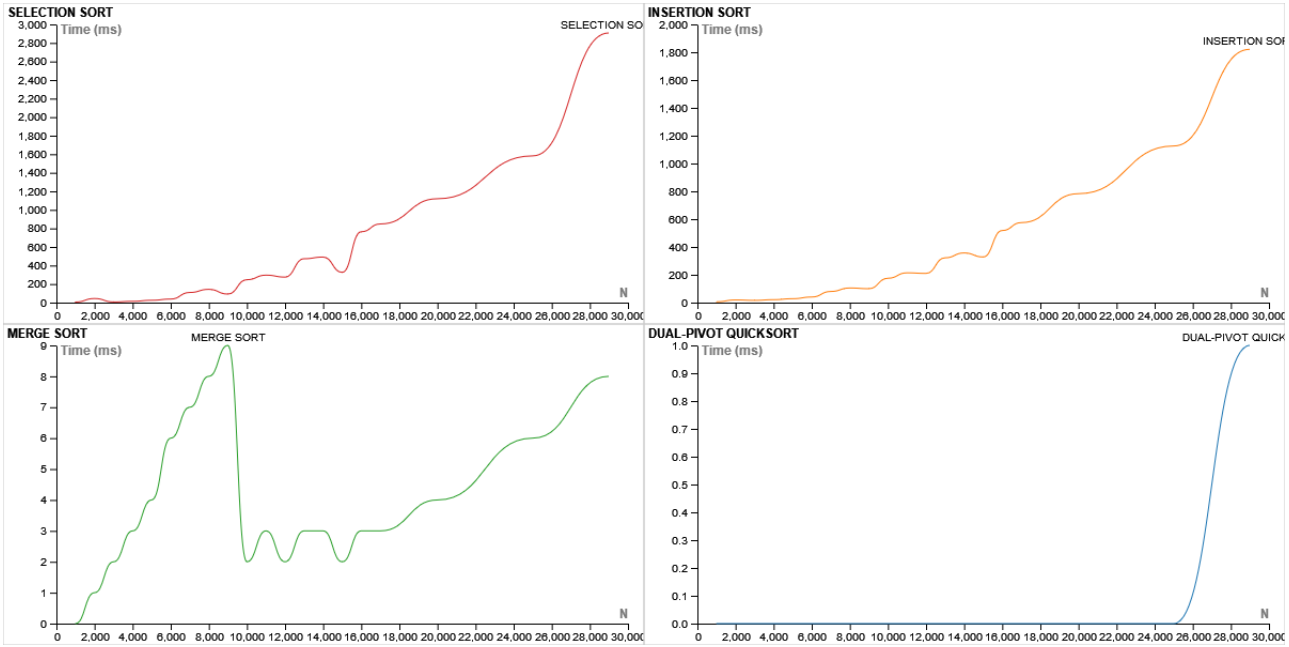

Figure 5: Independent Scale Series, All Data, *N vs Time*

3.3.2 Disclaimer on Stability Results

It's also important to note that these test results were gathered in a standard computing environment apposed to sterile VM environment. Thus, some of the variations shown – specifically the ones around the $N = 15,000$ mark – may be due to a background process running. If this project was allotted more time, a sterile environment would've been used to record the statistics. However, given this limitation, we can still gather concretely that merge and *dpqs* both outperformed their quadratic counterparts.

4 Summary of Findings

This section will highlight the core findings I made when working through the data that has been generated from the given algorithms.

4.1 Sort Performance

We can conclude that all of our sorts except *dpqs* and merge-sort would fail to produce acceptable run times when deployed upon any data set that isn't minuscule in size $N > 500$.

4.2 Accuracy of Projections

We can conclude that our mathematical projections using Big O effectively illustrate the worst-case performance of any given piece of code. Thus, moving forward, when engineering large systems (and algorithms that make up these systems) we should be well equipped to provide logical and accurate advice when it comes to design or reworking under performing algorithms and data structures.

4.3 The Need for Sterile Environments

It has been discovered that to effectively time the execution of any type of sort, a sterile computing environment would be crucial in compiling accurate, **comparable**, results – especially when measuring small changes in run times (sub 1ms). Because of this limitation, the general stability of any given algorithm is quite hard to measure. However, we can still use the aforementioned results to form a better understanding of performance relative to other sorts. In other words, we can accurately compare all sorts ran in the same pass i.e., $N = 1000$ because if there was any 3rd party load placed on the processor it would affect all sorts, thus, distorting all results in a comparable way. We lose this granular comparability when observing sorts from $N = 1000$ against sorts from $N = 2000$ (when moving along the x-axis) because of the varied load placed upon the CPU – although, we still get a very clear picture of growth and *overall* performance.

The most scientifically sound method for approaching such issues with timing would be to setup a sterile VM and run passes thousands of times at the given N values, at which point we could average out all runs and come to a solid conclusion about sort stability. However, as stated earlier, this data still gives us a good idea of what we should expect when considering the growth of **any** algorithm and testing to the degree as mentioned earlier was unfortunately outside of the scope of this project.

4.4 Where the Inefficient Algorithms Prove Useful

Although we will rarely see such algorithms in the real world of computing, selection sort and insertion sort still remain as two of the first *real algorithms* you learn in CS2. These algorithms work efficiently to guide students while beginning to understanding the broad concept of algorithms. More importantly, we can see that these algorithms could be learned without severe performance penalties as long as we are using small data sets. Thus, for CS2/CS1 selection and insertion sorts prove to be a great stepping stone into the broad world of sorting algorithms.