Peyton Kelly, Gary Chen, and William Connelly

Progress Report

In this technical report, we will document the progress made by our team on our project to visualize various sorting algorithms using Python and PyQt5. Our goal is to create a user-friendly interface that allows users to visualize and compare the performance of different sorting algorithms on various datasets. We have made significant progress toward implementing our project. We have created a graphical user interface (GUI) using PyQt5, which allows users to select from a list of available sorting algorithms and datasets. We have also implemented the functionality to read in CSV files and extract the data to be sorted. We have also created a function for each sorting algorithm and successfully tested them on small datasets. We can visualize the sorting process using a bar chart that updates in real-time as the algorithm progresses. In addition, we have implemented the functionality to write the sorted data into a JSON file. We faced some unexpected challenges during the development process. One of the main challenges was dealing with large datasets. Initially, we tried to sort a dataset with over 1 million records, but our code was not optimized for such large datasets, and the program took a long time to sort the data. We had to restructure our code to use a more efficient algorithm for larger datasets. Another challenge we faced was designing the GUI. We had to carefully consider the layout of the interface and how to make it intuitive for users to navigate. We also had to ensure that the GUI was responsive and provided timely feedback to the user during sorting. As we progressed with the project, we made several changes to our initial plan. We initially planned to use a different visualization library, but we found that it was not suitable for our needs, so we switched to using

Matplotlib. We also had to make changes to our code to improve the performance of the sorting algorithms when dealing with large datasets.

Sorting Algorithms:

Bubble Sort is a simple comparison-based algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It has a time complexity of O(n^2) and is inefficient for large datasets. However, it is easy to implement and is efficient for small datasets.

Selection Sort is another simple comparison-based algorithm that sorts an array by repeatedly finding the minimum element from the unsorted part and placing it at the beginning. It also has a time complexity of O(n^2) and is not efficient for large datasets, but is efficient for small datasets.

Insertion Sort is a simple comparison-based algorithm that builds the final sorted array one item at a time. It iterates through an array, comparing each element to the sorted part of the array and placing it in the correct position. It has a time complexity of O(n^2) but performs better than Bubble Sort and Selection Sort on small datasets.

Merge Sort is a divide-and-conquer algorithm that divides an array into two halves, sorts each half, and then merges the two sorted halves. It has a time complexity of O(n log n) and is efficient for large datasets.

Quick Sort is also a divide-and-conquer algorithm that divides an array into two sub-arrays, sorts each sub-array, and then combines them. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. It has a time complexity of O(n log n) on average but can have a worst-case time complexity of O(n^2) if the pivot is not chosen correctly.

Counting Sort is an efficient sorting algorithm that works by counting the number of occurrences of each distinct element in the input array and then sorting them based on the count. It has a time complexity of O(n+k), where k is the range of the non-negative key values. It is efficient for small datasets with a small range of key values.

Radix Sort is another non-comparison-based algorithm that sorts elements based on their digit values. It works by grouping the elements by digit position, starting from the least significant digit, and sorting them based on that digit. It has a time complexity of O(d*(n+k)), where d is the number of digits in the maximum element and k is the range of the digit values. It is efficient for large datasets with small key values.

In conclusion, the team has made significant progress in implementing a sorting algorithm visualization tool. We started by researching and selecting several common sorting algorithms, namely bubble sort, selection sort, insertion sort, merge sort, quick sort, counting sort, and radix sort. We then implemented the algorithms in Python and integrated them with a PyQt5 GUI to enable the visualization of the sorting process in real time. While we encountered some unexpected challenges during the development process, such as figuring out how to update the plot in real time, we were able to overcome them through collaboration and research. We also had to change our initial model/framework to better suit our needs, but we believe the current implementation is more robust and efficient. Through our testing, we have determined that the sorting techniques used in our code are highly efficient and perform well on large datasets. However, it is important to note that the speed at which the algorithms sort is much faster than how the plot is being shown. This is because we intentionally slowed down the sorting process to enable the visualization in real time. Therefore, the sorting algorithms may appear to take longer than they do. In addition to the sorting visualization, we plan to include a feature that displays the runtimes of each sorting function as the plotting is being done. This will give users a better understanding of how each sorting algorithm performs in terms of speed and efficiency. However, it is important to note that displaying the full runtimes of large datasets may require more computing power and resources. We suggest that users have their machines plugged in and have sufficient GPU to handle the display of large datasets. By incorporating this feature, we hope to provide users with a more comprehensive understanding of the sorting algorithms and their performance. Overall, we believe that the sorting algorithm

visualization tool we have developed has the potential to be a valuable resource for developers and educators who are interested in learning more about sorting algorithms and their performance. We plan to continue improving the tool and expanding its functionality in the future.