Jonathan Julian

CSC – 510

Dr. Rahul Singh

Project documentation

Sorting Card Transactions Using Multiple Sorting Algorithms

When facing the manipulation of very large data sets there are numerous different techniques available to sort offering different advantages and pitfalls. The purpose of this project was to implement various sorting algorithms and variations of those algorithms to observe their effectiveness and their accuracy. When utilizing sorting algorithms there are multiple dimensions which categorize different algorithms. One of the most important aspects of sorting is the time efficiency to correctly and accurately sort the dataset. Another key facet of sorting is the space complexity required to sort. The final aspect observed in this project is the effectiveness of randomization in particular scenarios.

The idea for this project came from the idea that many large companies which issue credit cards to executives, purchasers and drivers need a way to effectively sort through many transactions. My personal experience in this matter comes from the time I spent in the Air Force. I was in charge of auditing the purchases for every government issued credit card on our base. This lent itself to hundreds, if not thousands, of paper monthly statements. In which case more often then not I was only looking for certain information. Traditionally credit card companies provide the statement sorted by ascending date. The project itself can be easily handled by the applications readily available, but my time in the corporate world has shown that not every one is as computer literate as we’d hope. One particular solution is to use the sort function in excel when you export the transaction list. Even most credit card companies give the ability to sort based on merchant and purchase/credit amount on their sites. A quick search online alludes to the fact that most of these sorts are in fact done by quick sort.

For this project I implemented the Quick Sort and Merge Sort algorithms. For the implementation of each of these methods I used a “standard” use of pivots selecting the middle most element. These two procedures offer similar time complexities, being O (n log n) in the best case scenarios. The worst case scenario for quicksort is O(n2), in contrast to merge sort maintaining its O (n log n) efficiency even in its worst case. Although these algorithms offer similar time complexities one of the biggest down fall is that merge sort is not a sort in place algorithm.

The project structure was loosely based on a project I had previously completed in terms of class hierarchy. The “main” method is located in the TransactionSorter class. This class immediately instantiates a “TransactionSorter” object which enters the project. The TransactionSorter is instantiated by storing a Seperator object which essentially is just reading in and storing the statement into a buffered reader for use in the TransactionManager class. After setting up the run environment the TransactionSorter class calls the UserInterface class. This class is the bridge between the program and the TransactionManager which has every transaction read in from the program arguments parsed into an ArrayList of Transaction objects. The UI consists of a few nested loops, mainly to verify input from the user, these loops acquire the information needed by the TransactionManager to correctly call the desired sorting algorithm. Each call to a sort creates an array copy of the main ArrayList contained in the TransactionManager to ensure that the order does not change between successive sorts while in the program.

The dataset used consists of 151038 transactions, with mostly randomly generated transaction amounts and dates. The dataset is a tab separated value set stored in a .txt file passed into the program via program arguments. The data is read into transaction objects and stored in an arraylist to allow for the generation of a deep copy array trivial with the use of the arrayList.toArray() method. The user is then prompted to enter the criteria for sorting of the items. The user has the ability to select the data field, sorting algorithm, and pivot used when using the Quick Sort method. Once completed the program returns the time taken to properly sort the dataset. The program utilizes the flow described in figure 1.1.

In the 2 sorting classes there is a shared function which utilizes reflection to select the proper comparison from the data object based upon user input for field to sort by. Each of these classes also utilize generics to allow for a vast range of portability and future use. See figure 1 for a full UML/Class diagram.

After running the test in numerous configurations, it is noticed that despite their similarities in time complexity, the quicksort is hand-over-fist quicker in sorting the dataset. This is attributed to the fact that quicksort utilizes the sort in place algorithm, where in Merge Sort, every call has more steps to create a new array of size n/2, until n = 1. At this time Merge sort has to take additional steps to assess the comparison and place the 2 merged arrays into the correct order. This accounts for almost a 100% increase in step taken to sort the dataset. Quick Sort accomplished the sort in 4,742,958 steps taking an average of 692 milliseconds on 100 trials ran. Merge Sort accomplished its task in 8,217,278 steps and took an average of 26925 milliseconds (26.925 seconds).

To ensure my results I forced the worst case scenario on to both of the algorithms, surprisingly this didn’t add much time to either of the algorithms. For the case of quicksort it raised the average time to just over 1054 ms. To accomplish this I renamed all the merchants to the same name, this would ensure that the order would not change, as quicksort is not a stable algorithm. For merge sort, the average time was just over 27887 ms.

In conclusion it seems that that most effective sorting algorithm is the quicksort when using random pivots. This helps prevent the worst case scenario, but as seen the worst case scenario does not add much time onto this extremely efficient algorithm. The main disparity between the time required for these two algorithms to complete their sort is attributed to the number of steps required in the merge sort being substantially higher. The other factor which attributes to this increased time is the fact that merge sort is not a sort in place algorithm. This requires the CPU to access memory in order to complete the process. It is relatively apparent that even in a worst case scenario, even when focusing on time complexity, that the real world application of the quick sort is a superior choice to merge sort even in a worse-case scenario, at least on datasets which are sufficiently large and not required to be stored in (main) memory.

