CCDSALG MCO1

Submitted by: MP Group 35

BANIQUED, Lourenz Jhay G.

GUTIERREZ, Allen Andreii D.

REYES, Ma. Julianna Re-an Dg.

UNISAN, Ryu Kisen P.

BARFEH, Davood Paur

June 26,2023

I. Introduction

The project consists of sorting seven data sets with varying sizes using four sorting algorithms three of which are already given and one algorithm of the group's choosing. This report contains the resources used as well as their findings after accomplishing the tasks. The Sections include information about the various algorithms implemented in this project, The process of verifying the algorithms, whether they were able to sort every element of the list, and providing information on the algorithm's average time to sort and their frequency counts, a comparative analysis of the results of the frequency counts of the different algorithms, and lastly The group's insights, realizations as they accomplished the project.

II. Algorithms Implemented

The four algorithms used to sort the data sets are Insertion Sort, Selection Sort, Merge Sort and lastly Quick Sort.

Insertion Sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted list one element at a time. It works by dividing the input list into two parts: the sorted portion and the unsorted portion. The algorithm iterates through the unsorted portion, selecting each element and inserting it into its correct position within the sorted portion. The algorithm builds the sorted portion of the list gradually by inserting each element into its appropriate position. At any given iteration, the elements to the left of the current position are always sorted.Insertion Sort has a time complexity of O(n^2), where n is the number of elements in the list. It performs well on small lists or lists that are nearly sorted, but it becomes inefficient for large lists. However, it has some advantages over other sorting algorithms.

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm that divides the input list into two parts: the sorted portion and the unsorted portion. The

algorithm works by repeatedly selecting the smallest (or largest, depending on the sorting order) element from the unsorted portion and swapping it with the element at the beginning of the unsorted portion. This process continues until the entire list is sorted.The algorithm works in-place, meaning it doesn't require additional memory beyond the input list. It repeatedly finds the smallest (or largest) element from the unsorted portion and swaps it with the element at the current position. This process gradually builds up the sorted portion of the list from left to right. Selection Sort has a time complexity of O(n^2), where n is the number of elements in the list. This makes it inefficient for large lists. However, it performs well on small lists or lists that are almost sorted.

Merge Sort

Merge Sort is a popular and efficient comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by dividing the input list into smaller sublists, sorting them independently, and then merging them back together to obtain the final sorted list. The core idea of Merge Sort is the merging step, where two sorted sublists are combined into a larger sorted sublist. During the merging process, the algorithm compares the elements from the two sublists and places them in the merged sublist in the correct order. This process is repeated recursively until all sublists have been merged into a single sorted list. Merge Sort has a time complexity of O(n log n), where n is the number of elements in the list. It guarantees a consistent and optimal performance, making it suitable for large lists. One of the key advantages of Merge Sort is its ability to handle very large data sets efficiently due to its divide-and-conquer approach. However, it requires additional memory space proportional to the size of the input list, which can be a drawback for limited-memory environments.

Quick Sort

Quick Sort is a widely used comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a pivot element from the list, partitioning the other elements into two subarrays based on their relationship to the pivot, and recursively sorting the subarrays. The key step in Quick Sort is the partitioning process, where the elements are rearranged around

the pivot. This process ensures that the pivot ends up in its final sorted position, with all elements less than the pivot on one side and all elements greater than the pivot on the other side. The partitioning is typically done using two pointers that traverse the subarray and swap elements when necessary. Quick Sort has an average time complexity of O(n log n), where n is the number of elements in the list. In the best-case scenario, where the pivot is chosen optimally, the time complexity can be reduced to O(n). However, in the worst-case scenario, where the pivot is consistently chosen poorly, the time complexity can increase to O(n^2). Quick Sort is generally efficient and performs well in practice due to its average-case time complexity. Quick Sort is an in-place sorting algorithm, which means it operates directly on the input list without requiring additional memory. It is not a stable sorting algorithm, as the relative order of elements with equal values may change during the partitioning process.

III. Process of running algorithms

The group has a class called "FileReader" where it has a method that reads information from a file and stores it into an array. That method, when called on in main, returns an array containing the ID numbers and names from the file. FileReader comes from the given code provided to us by the professor.

```
public Record[] readFile(String path) {
    try {
        File f = new File(path);
        Scanner scanner = new Scanner(f);
        int n = scanner.nextInt(); // Get the number of records
        Record[] result = new Record[n]; // Initialize array of records
        for (int i = 0; i < n; i++) { // Loop through each record
            int idNumber = scanner.nextInt(); // get the ID number
            String name = scanner.nextLine(); // get the name
            Record record = new Record(name, idNumber); // instantiate a new Record object
            result[i] = record; // Store the record in the array
        }
        scanner.close();
        return result;
    } catch (FileNotFoundException e) {
        System.err.println("File not found.");
        e.printStackTrace();
        return null;
    }
}
```

Image 3.a

To verify the results, the group utilized a simple for loop and an if statement in the main class. The for loop cycles every element of the recently sorted array and checks if the element next to it is a higher or lower number. If everything has been sorted correctly, the loop prints true. If there has been a mistake in sorting then it prints false.

```
System.out.print("Verifying List: ");
boolean arranged = true;
for(int i = 0; i < records.length - 1;i++){
    if(records[i].getIdNumber() > records[i+1].getIdNumber()){
        arranged = false;
    }
}
System.out.println(arranged);
```

Image 3.b

To benchmark the execution times, the group used a built-in function of java "System.currentTimeMillis()". What the group did was to get the current time of the system at the start of running the algorithm and get the current time of the system again as soon as the algorithm has finished running. To get the final execution time, we simply subtracted the ending time and the starting time to get the execution time in milliseconds

```
long startTime = System.currentTimeMillis();
switch (choice) {
    case 1:
        sortingAlgorithms.insertionSort(records, records.length); //Insertion Sort
        break;
    case 2:
        sortingAlgorithms.selectionSort(records, records.length); //Selection Sort
        break;
    case 3:
        sortingAlgorithms.mergeSort(records, l: 0, r: records.length - 1); //Merge Sort
        break;
    case 4:
        sortingAlgorithms.quickSort(records, lowerIndex: 0, higherIndex: records.length - 1); //Quick Sort
        break;
    default:
        System.out.println("Invalid choice. Sorting aborted.");
        return;
}
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

Image 3.c

IV. Execution Times and Frequency Counts

Algorithms	**Data Set**	**Execution Times (average) in ms**	**Frequency Counts**
Insertion Sort	almostsorted.txt	2716, 4726, 3960, 3985, 3922 **= 4094.6**	**1145344654**
	random100.txt	**0**	**5816**
	random25000.txt	382, 418, 375, 390, 394 **= 391.8**	**312043184**
	random50000.txt	1777, 1758, 1787, 1817, 1935 **= 1814.8**	**1251639466**
	random75000.txt	4597, 4526, 4183, 4351, 4243 **= 4380**	**2.823814228E9**

	random100000.txt	20419, 18197, 19235, 19024, 18936 **= 19162.2**	**4.97875292E9**
	totallyreversed.txt	36270, 42606, 34668, 39049, 38906 **= 38299.8**	**1.0000200002E10**
Selection Sort	almostsorted.txt	11829, 11730, 12803, 11724, 12449 = **12107**	**5.00057131E9**
	random100.txt	**0**	**5852**
	random25000.txt	493, 506, 562, 502, 451 **= 502.8**	**3.12855256E8**
	random50000.txt	2434, 2362, 2309, 2346, 2302 **= 2350.6**	**1.250742318E9**
	random75000.txt	5893, 5914, 5835, 6055, 6150 **= 5969.4**	**2.813644549E9**
	random100000.txt	13241, 13239, 14311, 13431, 13365 = **13517.4**	**5.001559584E9**
	totallyreversed.txt	9755, 10737, 10095, 9803, 10832 = **10244.4**	**7.500549995E9**
Merge Sort	almostsorted.txt	25, 31, 28, 28, 25 = **27.4**	**600007**
	random100.txt	**0**	**602**
	random25000.txt	14, 15, 13, 13, 13 = **13.6**	**150007**
	random50000.txt	20, 18, 19, 21, 17 = **19**	**299997**
	random75000.txt	24, 25, 21, 25, 24 = **23.8**	**450010**
	random100000.txt	38, 31, 34, 31, 29 = **32.6**	**599977**
	totallyreversed.txt	29, 28, 22, 22, 21 = **24.4**	**550012**
Quick Sort	almostsorted.txt	36, 36, 40, 43, 39 = **38.8**	**8513358**
	random100.txt	**0**	**2486**
	random25000.txt	9, 10, 10, 10, 8 = **9.4**	**1509689**

	random50000.txt	22, 16, 14, 18, 14 = **16.8**	**3549565**
	random75000.txt	25, 24, 25, 20, 18 = **22.4**	**4913147**
	random100000.txt	38, 37, 36, 27, 37 = **35**	**7013865**
	totallyreversed.txt	24, 24, 24, 25, 24 = **24.2**	**5403985**

V. Comparative analysis

Among all the Data Set, it seems that Totallyreversed.txt and almostsorted.txt are both the ones with the highest execution times and frequency count. From the Insertion sort we could conclude that this sorting is really bad when the data set is Totallyreversed.txt you could see that it had **38299.8** of execution time and also the highest frequency count among the 4 sorting algorithms (**1.0000200002E10**). According to an online article by Khan Academy (2015), the worst case for insertion sort is when the data or array is in a reverse order. This statement solidifies that the frequency count and execution time is the worst for that totallyreversed.txt. Another bad sorting for the totallyreversed.txt data set would be selection sort, as proven by the average execution time and also the frequency count. However, the next two algorithms; Merge sort and Quick sort have no problems in arranging the data set even if it is in a reversed order.

Next data set is almost sorted, the same as the last data set, the algorithms Selection sort and Insertion sort performed the worst in Execution times and Frequency Count and the other two algorithms Merge sort and Quick sort didn't have any problems with this data set.

Next data set has similar categories but differs in the amount of data. Randomly sorted data starting from 100 all the way up to 100000, the algorithm that performed the best was Merge Sort out of the 4 but Quick sort is just below by a few milliseconds. The time difference as the amount of data goes up by a little in both Merge Sort and Quick Sort but this isn't the case for Selection sort and Insertion sort wherein the more data needed to sort the longer the execution time and also larger frequency count.

Overall, Merge sort is the fastest among the 4 with Quick sort just behind a couple of milliseconds and after those two we have Insertion sort and Selection sort for having the slowest execution time among the data set.

VI. Summary of Findings

The project revolves around the implementation of the various sorting algorithms from CCDSALG into code. The primary objective of this project is to assess the knowledge of the group in terms of code implementation from the topic of algorithms. The members were given a choice on which programming language to utilize, to which the group used JAVA for the code implementation. All sorting algorithms were inspired from GeeksForGeeks, a portal consisting of computer science articles. The group was also tasked to determine the average execution time (in milliseconds) and the frequency count for each sorting algorithm for multiple test files. In conclusion, out of all the algorithms used, the most convenient sorting algorithm to use is merge sort. Merge sort is the fastest sorting algorithm out of all the algorithms used because of the "divide and conquer" strategy utilized to dissect the whole array into subarrays making it easier for the program to compare to smaller sizes, hence compiling "faster".

There were some instances where the group encountered challenges. For instance, the Quick Sort algorithm when called (sorting the *totallyreversed.txt* file) resulted in a stack overflow error. The way how quick sort chooses a pivot element is the higher index at first which will result in a stack overflow. The group realized that and used the middle index instead as its pivot index. For the insights of the group, we realized that in this project where Merge and Quick Sort have lower execution times compared to Insertion and Selection Sort. Dividing a whole array of elements into multiple smaller branches of subarrays then comparing the values after is better and much more stable than comparing a value to every element in sorting a huge size.

VII. Contributions

MEMBER NAME	CONTRIBUTIONS

Baniqued, LourenzJhay G.	- Code of Merge Sort - Execution times and Frequency count testing - Comparative Analysis
Gutierrez, Allen Andrei D.	- Code of Merge Sort - Section 1 of the Report - Section 2 of the Report
Reyes, Ma. Julianna Re-an Dg.	- Code of the Insertion Sort - Code of the Selection Sort - Code of Quick Sort - Section 3 of the Report(Process of Running Algorithms) - Section 1 of the Report(Introduction)
Unisan, Ryu Kisen Unisan	- Code of Merge Sort - Summary of Findings - Frequency Testing of all Files and Algorithms - Execution Time Average Testing of all Files and Algorithms

VIII. References

Analysis of insertion sort (article) | Khan Academy. (2015). from https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

GeeksforGeeks. (2023). Insertion Sort Data Structure and Algorithm Tutorials. *GeeksforGeeks*. https://www.geeksforgeeks.org/insertion-sort/

GeeksforGeeks. (2023). Merge Sort – Data Structure and Algorithms Tutorials. *GeeksforGeeks*. https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (2023). QuickSort – Data Structure and Algorithm Tutorials *GeeksforGeeks*. https://www.geeksforgeeks.org/quick-sort/

GeeksforGeeks. (2023). Selection Sort – Data Structure and Algorithm Tutorials *GeeksforGeeks*. https://www.geeksforgeeks.org/selection-sort/

Merge Sort (With Code in Python/C++/Java/C). (n.d.-b). https://www.programiz.com/dsa/merge-sort

Patel, H. (2022, March 14). An Overview of QuickSort Algorithm - Towards Data Science. *Medium*. https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72#:~:text=Quicksort%20is%20a%20fast%20sorting,them%2C%20and%20then%20recombining%20them.

Selection Sort (With Code in Python/C++/Java/C). (n.d.). https://www.programiz.com/dsa/selection-sort

IX. Images

Image 3.a

```java
public Record[] readFile(String path) {
    try {
        File f = new File(path);
        Scanner scanner = new Scanner(f);
        int n = scanner.nextInt(); // Get the number of records
        Record[] result = new Record[n]; // Initialize array of records
        for (int i = 0; i < n; i++) { // Loop through each record
            int idNumber = scanner.nextInt(); // get the ID number
            String name = scanner.nextLine(); // get the name
            Record record = new Record(name, idNumber); // instantiate a new Record object
            result[i] = record; // Store the record in the array
        }
        scanner.close();
        return result;
    } catch (FileNotFoundException e) {
        System.err.println("File not found.");
        e.printStackTrace();
        return null;
    }
}
```

Image 3.b

```java
System.out.print("Verifying List: ");
boolean arranged = true;
for(int i = 0; i < records.length - 1;i++){
    if(records[i].getIdNumber() > records[i+1].getIdNumber()){
        arranged = false;
    }
}
System.out.println(arranged);
```

Image 3.c

```
long startTime = System.currentTimeMillis();
switch (choice) {
    case 1:
        sortingAlgorithms.insertionSort(records, records.length); //Insertion Sort
        break;
    case 2:
        sortingAlgorithms.selectionSort(records, records.length); //Selection Sort
        break;
    case 3:
        sortingAlgorithms.mergeSort(records, l: 0, r: records.length - 1); //Merge Sort
        break;
    case 4:
        sortingAlgorithms.quickSort(records, lowerIndex: 0, higherIndex: records.length - 1); //Quick Sort
        break;
    default:
        System.out.println("Invalid choice. Sorting aborted.");
        return;
}
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```