جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

College of Computer and Information Sciences (CCIS)

CS210: Data Structures and Algorithms

Developing an Intelligent System for Students' Course Registration Analysis

Course Instructor: Dr. Najla Althuniyan

Section #: 951

Project Contributors:

LAYAN HESHAM RASHED BINDAYEL 223410104

SHOUG FAWAZ ABDULLAH ALOMRAN 223410392

Table of Contents

Abstract

This project designed an intelligent system for managing students' course registration records at Prince Sultan University, predicting course demand through data structures such as linked lists and arrays as well as sorting algorithms. The system stores records in a self-developed linked list and has four algorithms for sorting—Selection Sort, Insertion Sort, Merge Sort, and Quick Sort - to compare their time complexity performance. The outcomes were according to theoretical time complexities in the sense that Merge Sort and Quick Sort O(n log n) had a huge margin over Selection and Insertion Sorts O(n^2) when given a database with more than 3,000 records. The project amply justifies the importance of algorithm choice on performance by confirming that computationally intensive algorithms like Merge and Quick Sort are best suited for efficient management of large data.

Introduction

At the very foundation of programming is a simple but powerful necessity: sorting things out. Sorting is one of the most useful tools for a programmer, taking jumbled data and turning it into organized information that is easy to locate, examine, and maximize. For any code writer, having knowledge of the different ways to sort isn't abstract—it's practical. Knowing which technique to employ can be the difference between a speedy application and a slow one, particularly as data volume increases.

This project answers one central question of this field: in practice, how do different sorting algorithms work? We're bridging theory and practice by comparing how long it takes to run each one. The goal is to demonstrate that an algorithm's inherent complexity has a direct impact on real-world speed, and how our choices as programmers affect the behavior of a program.

To achieve this, we designed a special list to manage the enrollments of the students into courses. We took the data and sorted it four times through regular sorts: the inefficient but simple Selection and Insertion Sorts, and the more sophisticated divide-and-conquer Merge Sort and Quick Sort. We also timed each one in order to see how their theoretical performance stacked up against their actual speed.Ultimately, this project serves as a practical example.

High-Level Solution / System Design Overview

This application is meant to develop a Prince Sultan University-specific system that takes the record of the student's course registration and computes the demand for each course accordingly. Additionally, this system undergoes three major phases, which are:

- **Data Collection Phase:** The system begins this phase by gathering the records of student course registration retrieved from Prince Sultan University's database, which has approximately 3000 entries. Each record has the student's basic information like their ID number, course enrolling, identified by its course ID, study level at the moment, and the enrollment time. In addition, the system carefully checks and reviews each record to ensure the information is correct and complete, then starts sorting and formatting it in a readable format. This provides the system with a strong foundation in its capacity to effectively understand and analyze students' requests in order to ascertain course demand and trends, critical to inform future-stage decision-making. Processing Phase (Demand Score Calculation): The demandScorer calculates a definite score for each student on the basis of 4 criterions for each node:

- **Processing Phase (Demand Score Calculation):** For each node the demandScorer calculates a certain score for each student based on 4 criterions:

1. **Base score: 50**, which represents the initial score each student starts with.
2. **Class multiplier:** the base score is multiplied by a factor depending on the current grade level of the students.
    - **Freshman:**If the student is a freshman, his or her base score will be multiplied by 0.75 to indicate that it has been decreased by 25%
    - **Sophomore:** If the student is a sophomore, his or her base score will be multiplied by 0.90 to indicate that it has been decreased by 10%
    - **Junior:** When the student is a junior, their base score is multiplied by 1.1 to indicate that it has been raised by 10%
    - **Senior:** When the student is a senior, their base score is multiplied by 1.25 to indicate that it has been raised by 25%
    - **Graduate:** When the student is a graduate, their base score is multiplied by 1.5 to indicate that it has been raised by 50%

Therefore, the system ensures that high priority is assigned to nearly graduated students or graduates and low priority to fresh students, who are fortunate enough to have additional time to register courses in future semesters than students who don't..

3. **Time of Registration(s):** The demand score of students is highly determined by when they register for a given course. That is, students who choose to register early will be given bonus points, while students who register late will be penalized in relative sequence. The rules governing the imposition of bonuses and penalties are as follows:

- ➔ Bonus of 5 points is awarded to the total demand score, if the student registers for the course between 6:00-8:00AM
- ➔ Bonus of 10 points is awarded to the total demand score, if the student registers for the course between 8:00-10:00AM (peak time)
- ➔ No penalty or bonuses are given to students registering for courses between 12:00-2:00PM
- ➔ There is a penalty of 5 points subtracted from the total demand score, if the student registers for the course between the period 2:00-4:00PM
- ➔ There is a penalty of 10 points subtracted from the total demand score, if the student registers for the course between the period 4:00-8:00PM
- ➔ There is a penalty of 15 points or more subtracted from the total demand score, if the student registers for the course after 8:00PM

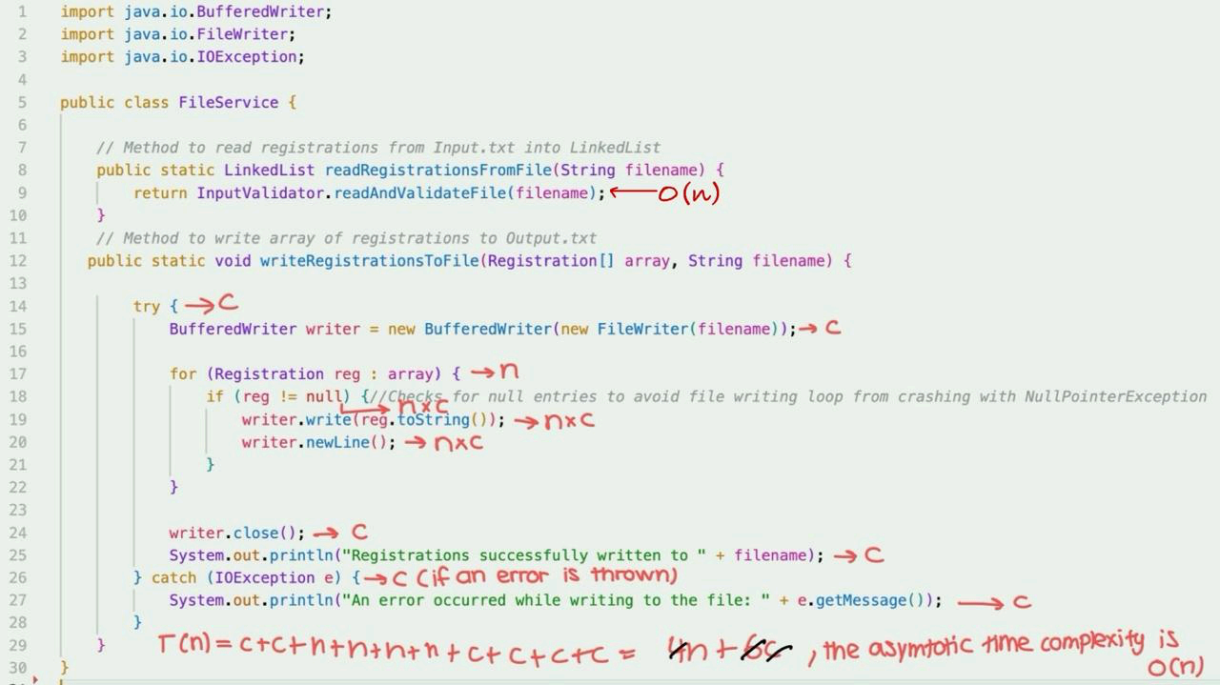
4. **Course ID Priorities:** As the very name indicates, this measure assigns greater weight to certain courses over others, and is categorized under two headings:
    - **High-priority courses:** High priority level courses possess a course code which contains an odd number at the end e.g. CS311. This will add a bonus of 20 points to the overall demand score.
    - **Low-priority courses:** The course with a lower priority level will have a course code which will end with an even number e.g. CS210. This will deduct a penalty of 10 points to the overall demand score.

- **Documentation (Writing) Phase:** Once all the data has been fully analyzed, examined, and processed based on the above criterions, the system writes the final results in a new file**.** It then proceeds to update each student's record with his or her own demand score, which is a measure of how highly demanded or popular a given course happens to be. It then, after storing the updated student records, notifies students that their registration requests have been processed and results can be viewed. It leaves the data unsorted at this time, allowing for the possibility of future performance testing using sorting algorithms. Hence, these phases emphasize how the prototype builds records data and transforms it into beneficial information by way of analyzing demand that enables smart decision making on the registration system. These comprise enhanced course capacity and reduced overcrowding, timetabling properly, and effective future planning with resources being assigned and utilized based on the demand of the students.

Overall, it results in a more efficient way of course registration management while enhancing the student experience with a data-driven and intuitive registration system.

Asymptotic Performance Analysis

- **The Writing and Reading Running Time**



```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileService {

    // Method to read registrations from Input.txt into LinkedList
    public static LinkedList readRegistrationsFromFile(String filename) {
        return InputValidator.readAndValidateFile(filename);
    }
    // Method to write array of registrations to Output.txt
   public static void writeRegistrationsToFile(Registration[] array, String filename) {

        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter(filename));

            for (Registration reg : array) {
                if (reg != null) {//Checks for null entries to avoid file writing loop from crashing with NullPointerException
                    writer.write(reg.toString());
                    writer.newLine();
                }
            }

            writer.close();
            System.out.println("Registrations successfully written to " + filename);
        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file: " + e.getMessage());
        }
    }
}
```

O(n)

c (if an error is thrown)

T(n) = c+c+n+n+n+n+c+c+c+c = 4n + 6c, the asymptotic time complexity is O(n)

- **The Processing (Demand score computation) Run Time**

```
1   public class Benchmarker {
2       // Helper method to make a copy of the array
3       private static Registration[] makeCopy(Registration[] original) {
4   O(1)→Registration[] copy = new Registration[original.length]; // Create new empty array
5   O(1)→for (int i = 0; i < original.length; i++) { // Loop through each element
6     O(n)→copy[i] = original[i]; // Copy each element
7           }
8   O(1)→return copy; // Return the copy
9       }
10
11      // Runs all 4 sorting algorithms and times them.
12      public static void runAllSorts(Registration[] originalArray) {
13          System.out.println(x:"--- RUNNING ALL SORTS ---");←O(1)
14
15          Timer timer = new Timer();←O(1)
16
17          // 1. SELECTION SORT
18  O(n)→Registration[] copy1 = makeCopy(originalArray); // Make fresh copy.
19    O(1)→timer.start(); // Start timer.
20 O(n²)→new SelectionSort().sort(copy1); // Do the sorting + create an object.
21    O(1)→timer.stop(); // Stop timer
22          // File writing moved OUTSIDE timed section to measure only sorting time
23  O(n)→FileService.writeRegistrationsToFile(copy1, filename:"Sorted_Output_SS.txt"); // Save result.
24          System.out.println("Selection Sort: " + timer.getElapsedTimeMillis() + " ms");←O(1)
25
26          // 2. INSERTION SORT
27  O(n)→Registration[] copy2 = makeCopy(originalArray); // Fresh copy again.
28          timer.start();←O(1)
29  O(n²) →new InsertionSort().sort(copy2); // Do the sorting + create an object.
30          timer.stop();←O(1)
31          // File writing moved OUTSIDE timed section to measure only sorting time
32  O(n)→FileService.writeRegistrationsToFile(copy2, filename:"Sorted_Output_IS.txt"); // Save result.
33          System.out.println("Insertion Sort: " + timer.getElapsedTimeMillis() + " ms");←O(1)
34
35          // 3. MERGE SORT
36  O(n)→Registration[] copy3 = makeCopy(originalArray); // Fresh copy again.
37          timer.start();←O(1)  ↙O(n log n)
38          new MergeSort().sort(copy3); // Do the sorting.
39          timer.stop();←O(1)
40          // File writing moved OUTSIDE timed section to measure only sorting time
41  O(n)→FileService.writeRegistrationsToFile(copy3, filename:"Sorted_Output_MS.txt"); // Save result.
42          System.out.println("Merge Sort: " + timer.getElapsedTimeMillis() + " ms");←O(1)
43
44          // 4. QUICK SORT
45  O(n)→Registration[] copy4 = makeCopy(originalArray); // Fresh copy again.
46          timer.start();←O(1)  ↙O(n log n)
47          new QuickSort().sort(copy4); // Do the sorting + create an object.
48          timer.stop();←O(1)
49          // File writing moved OUTSIDE timed section to measure only sorting time
50  O(n)→FileService.writeRegistrationsToFile(copy4, filename:"Sorted_Output_QS.txt"); // Save result.
51          System.out.println("Quick Sort: " + timer.getElapsedTimeMillis() + " ms");←O(1)
52      }
53  }
```

- **The Sorting Running Time**

  - **Selection Sort**

```java
public class SelectionSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        // Sort in DESCENDING order (highest demand scores first).
        for (int i = 0; i < array.length - 1; i++) {
            int maxIndex = i; // Assume the current position has the highest score.
            for (int j = i + 1; j < array.length; j++) { // Look at all remaining elements to find the actual highest.
                if (array[j].getStudDemandScore() > array[maxIndex].getStudDemandScore()) {
                    maxIndex = j; // Remember this as the new highest position.
                }
            }
            // Swap
            Registration temp = array[i];
            array[i] = array[maxIndex];
            array[maxIndex] = temp;
        }
    }

    @Override
    public String name() {
        return "Selection Sort";
    }
}
```

  - **Insertion Sort**

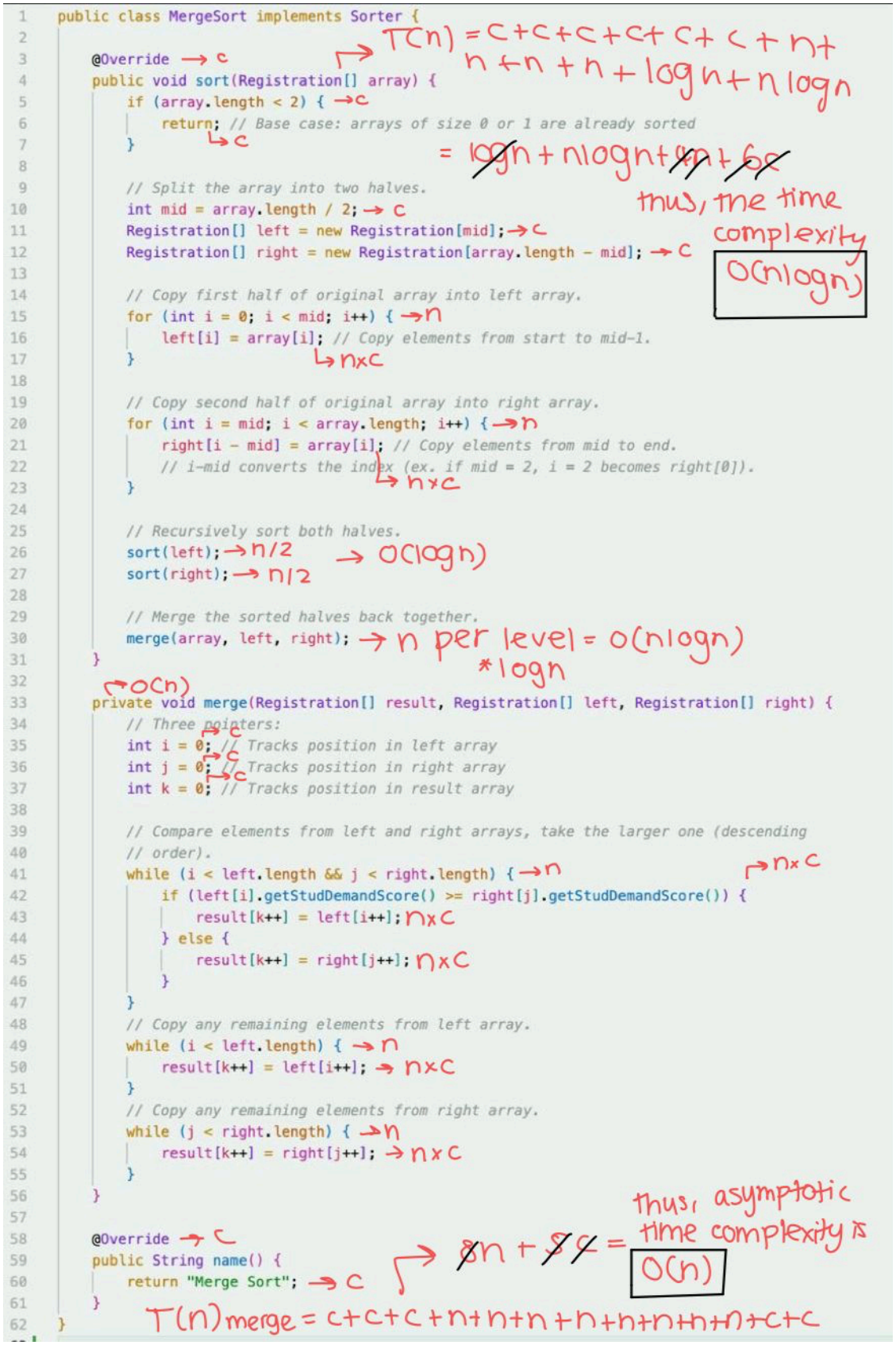
```java
public class InsertionSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        // Start from the second element (index 1).
        for (int i = 1; i < array.length; i++) {
            Registration current = array[i];
            double currentScore = current.getStudDemandScore(); // Get the demand score of the current element for
                                                                // comparison.
            int j = i - 1; // Start comparing with the element immediately to the left.

            // Shift elements that are smaller than current to the right.
            while (j >= 0 && array[j].getStudDemandScore() < currentScore) {
                array[j + 1] = array[j]; // Shift element to the right.
                j--;
            }
            // Insert current element in correct position.
            array[j + 1] = current;
        }
    }

    @Override
    public String name() {
        return "Insertion Sort";
    }
}
```

- **Merge Sort**



```java
public class MergeSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        if (array.length < 2) {
            return; // Base case: arrays of size 0 or 1 are already sorted
        }

        // Split the array into two halves.
        int mid = array.length / 2;
        Registration[] left = new Registration[mid];
        Registration[] right = new Registration[array.length - mid];

        // Copy first half of original array into left array.
        for (int i = 0; i < mid; i++) {
            left[i] = array[i]; // Copy elements from start to mid-1.
        }

        // Copy second half of original array into right array.
        for (int i = mid; i < array.length; i++) {
            right[i - mid] = array[i]; // Copy elements from mid to end.
            // i-mid converts the index (ex. if mid = 2, i = 2 becomes right[0]).
        }

        // Recursively sort both halves.
        sort(left);
        sort(right);

        // Merge the sorted halves back together.
        merge(array, left, right);
    }

    private void merge(Registration[] result, Registration[] left, Registration[] right) {
        // Three pointers:
        int i = 0; // Tracks position in left array
        int j = 0; // Tracks position in right array
        int k = 0; // Tracks position in result array

        // Compare elements from left and right arrays, take the larger one (descending
        // order).
        while (i < left.length && j < right.length) {
            if (left[i].getStudDemandScore() >= right[j].getStudDemandScore()) {
                result[k++] = left[i++];
            } else {
                result[k++] = right[j++];
            }
        }
        // Copy any remaining elements from left array.
        while (i < left.length) {
            result[k++] = left[i++];
        }
        // Copy any remaining elements from right array.
        while (j < right.length) {
            result[k++] = right[j++];
        }
    }

    @Override
    public String name() {
        return "Merge Sort";
    }
}
```

- **Quick Sort**

```java
public class QuickSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        quickSort(array, low:0, array.length - 1);O(n log n)
    }

    private void quickSort(Registration[] array, int low, int high) {
O(1)    if (low < high) {
            // Partition the array and get the pivot index.
            int pivotIndex = partition(array, low, high); O(n)

            // Recursively sort elements before and after pivot.
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);  O(log n)
        }
    }

    private int partition(Registration[] array, int low, int high) {
        // Choose the rightmost element as pivot.
        double pivot = array[high].getStudDemandScore();<— O(1)
O(1)—>int i = low - 1; // Index of smaller element.

O(n)—>for (int j = low; j < high; j++) {
            // If current element is greater than or equal to pivot (descending order).
 O(n)—>if (array[j].getStudDemandScore() >= pivot) {
                i++;<— O(n)
                // Swap array[i] and array[j]
                Registration temp = array[i];
   O(n)—>array[i] = array[j];
                array[j] = temp;
            }
        }

        // Swap array[i+1] and array[high] (put pivot in correct position).
        Registration temp = array[i + 1];<— O(1)
        array[i + 1] = array[high];<— O(1)
        array[high] = temp;<— O(1)

        return i + 1;<— O(1)
    }

    @Override
    public String name() {
        return "Quick Sort";<— O(1)
    }
}
```

Runtime Measurement

Operation	Theoretical Complexity	Actual Runtime (ms)
File Reading	O(n)	**106**
Demand Score Calculation (Processing)	O(n)	**98**
File Writing (Unsorted)	O(n)	**48**
Selection Sort	$O(n^2)$	**45**
Insertion Sort	$O(n^2)$	**26**
Merge Sort	O(n log n)	**17**
Quick Sort	O(n log n)	**22**

```
File reading completed in: 106 ms
Successfully read 3432 registrations.

Starting demand score calculations...
Demand score calculations completed in: 98 ms

Writing unsorted data to file...
Registrations successfully written to Output.txt
File writing completed in: 48 ms
Saved unsorted data to Output.txt

Starting sorting algorithms...
--- RUNNING ALL SORTS ---
Registrations successfully written to Sorted_Output_SS.txt
Selection Sort: 45 ms
Registrations successfully written to Sorted_Output_IS.txt
Insertion Sort: 26 ms
Registrations successfully written to Sorted_Output_MS.txt
Merge Sort: 17 ms
Registrations successfully written to Sorted_Output_QS.txt
Quick Sort: 22 ms
All sorting completed and results saved.
```

Interpretation of results

The results in the runtime table were obtained using a dataset of 3432 registrations. Stating the dataset is significant because it places the measured runtimes in context and enables the performance results to be fairly interpreted. Since algorithmic efficiency depends on

input size, reporting that all timings were measured with 3432 records shows that analysis was conducted under realistic and standard conditions. This also allows fair comparison between the sorting algorithms and makes it easier to reproduce in the event that the same dataset is used again.

Teamwork Contribution and Coordination

This project was completed in teamwork by Shoug Alomran and Layan Bindayel under the supervision of Dr. Najla Althuniyan. The two members worked together during the development, keeping each other posted and sharing the responsibilities to ensure smooth flow and integration.

Layan Bindayel coordinated and organized the project. She created the report template, drew up the document outline, and divided the work evenly between the two members. Layan did the implementation of the lower-level system components, including Node.java, LinkedList.java, FileService.java, and Comparators.java. Layan wrote the High-Level Solution and Asymptotic Performance Analysis section of the report, explaining the overall design and theoretical performance of the algorithms.

Shoug Alomran completed the analysis and runtime components of the project. She implemented Input Validator Class , Demand Scorer Class, sorting algorithms (SelectionSort, InsertionSort, MergeSort, and QuickSort), and Bench Marker Class and Timer Class. Shoug also completed the Runtime Measurement section of the report, including measuring actual runtimes, generating tables, and including screenshots to illustrate the performance of the system.

Both partners collaborated on Main Class, bringing all components together in one program. Layan did data input and list integration, while Shoug did processing, sorting, and benchmarking

phases. Together, they tested, debugged, and validated the entire system, demonstrating overall teamwork, coordination, and fair contribution.

Conclusion

In essence, the results of this project confirmed that the algorithms used behaved as theoretical and empirical expectations. Runtime was utilized as a metric to determine that Merge Sort and Quick Sort were the fastest in terms of runtime because of their sound O(n log n) complexity, with Selection Sort and Insertion Sort being much slower, as would be expected from their $O(n^2)$ complexity. That theoretical consideration would match actual results in such a manner ensures system implementation and design correctness and optimality. Along the way, we were taught a valuable lesson in the trade-off between simplicity and efficiency.

The more straightforward algorithms that we have such as Selection Sort and Insertion Sort are easier to comprehend and implement but thus adequate for small data or teaching purposes. Their performance is traded off, however, with larger input size, to the more complicated divide-and-conquer approach used by Merge Sort and Quick Sort. This project expressed extremely well the direct relation of algorithmic efficiency to program performance in real-world applications. The system can be adapted to take actual registration data directly from university databases to analyze live course demand patterns in the future.

Any other features, such as the graphical user interface to display the demand scores, machine verification of the input data, or inclusion of yet another complicated algorithm like Heap Sort or Counting Sort, would also make it all the more performance-driven and user-friendly. Overall, the project accomplished what it intended to do—coming up with an

effective system that demonstrates how an algorithm's design is incorporated into data structures and run-time behavior in the real world for a learning context.

References and Resources

Alomran, S., & Bindayel, L. (2025). *CS210 Project — Student Course Registration Analysis* [Computer software]. GitHub. https://github.com/Shoug-Alomran/CS210-Project-Linked-List-Implementation-and-Runtime-Analysis

GeeksforGeeks. (n.d.). *Sorting Algorithms – Complete Guide.* Retrieved October 2025. https://www.geeksforgeeks.org/sorting-algorithms/

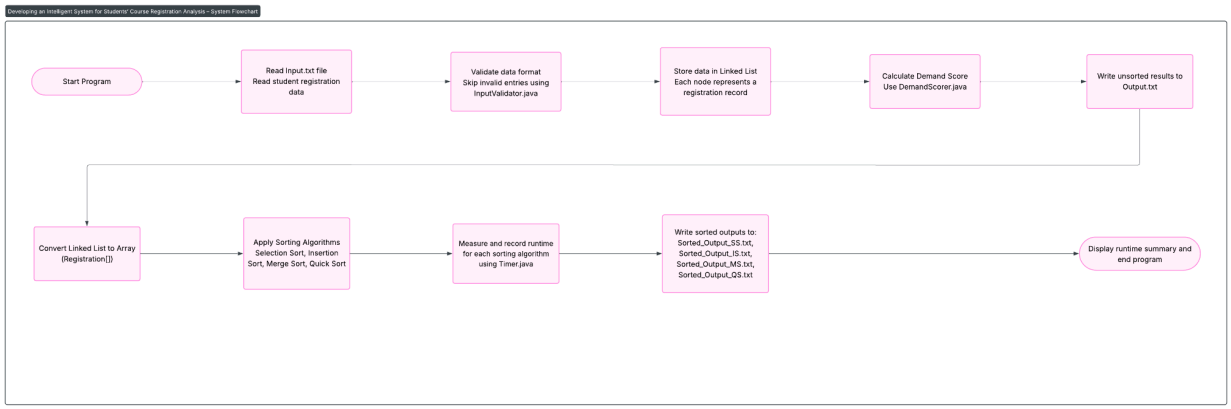
W3Schools. (n.d.). *Java BufferedWriter*. Retrieved October 2025. https://www.w3schools.com/java/java_bufferedwriter.asp

W3Schools. (n.d.). *Java BufferedReader*. Retrieved October 2025. https://www.w3schools.com/java/java_bufferedreader.asp (w3schools.com)

W3Schools. (n.d.). *Java Files*. Retrieved October 2025, from https://www.w3schools.com/java/java_files.asp (w3schools.com)

Alsomali, Y., & Alsubaie, H. (n.d.). *Prince Sultan University Parking Intelligent System Report*. CS210: Data Structures and Algorithms, Dr. Najla Althuniyan, Prince Sultan University.

Appendix A: Flow Chart Illustration



Appendix B: Full Java Code Implementation

Main Class

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();

        // Time reading operation
        System.out.println("Starting file reading...");
        timer.start();
                                                LinkedList      registrations      =
InputValidator.readAndValidateFile("Input.txt");
        timer.stop(); // This timer times the file reading only
        System.out.println();
                        System.out.println("File   reading   completed   in:   "   +
timer.getElapsedTimeMillis() + " ms");
              System.out.println("Successfully  read  "  +  registrations.size  +  "
registrations.");
        System.out.println();

        // Time processing operation
        System.out.println("Starting demand score calculations...");
        timer.start();
        calculateAllDemandScores(registrations);
        timer.stop(); // This timer times the demand score calculations only
              System.out.println("Demand  score  calculations  completed  in:  "  +
```

```
timer.getElapsedTimeMillis() + " ms");
            System.out.println();

            // Convert linked list to array
            Registration[] array = registrations.convToArray();

            // Time writing operation
            System.out.println("Writing unsorted data to file...");
            timer.start();
            FileService.writeRegistrationsToFile(array, "Output.txt");
            timer.stop(); // This timer times the file writing only
                          System.out.println("File   writing   completed   in:   "   +
timer.getElapsedTimeMillis() + " ms");
            System.out.println("Saved unsorted data to Output.txt");
            System.out.println();

            // Run and time all sorting algorithms
            System.out.println("Starting sorting algorithms...");
            Benchmarker.runAllSorts(array);
            System.out.println("All sorting completed and results saved.");
            System.out.println();

            // Additional sorting using comparators (Not required but useful)
            System.out.println("\n--- ADDITIONAL COMPARATOR SORTING ---");

            // Sort by Student ID
            Registration[] idSorted = registrations.convToArray();
            Arrays.sort(idSorted, Comparators.by_ID);
            FileService.writeRegistrationsToFile(idSorted, "Sorted_By_ID.txt");
            System.out.println("Saved student ID sorted data to Sorted_By_ID.txt");

            // Sort by Course Code
            Registration[] courseSorted = registrations.convToArray();
            Arrays.sort(courseSorted, Comparators.by_CourseCode);
                                    FileService.writeRegistrationsToFile(courseSorted,
"Sorted_By_Course.txt");
                          System.out.println("Saved   course   code   sorted   data   to
Sorted_By_Course.txt");

            // Sort by Academic Level
            Registration[] levelSorted = registrations.convToArray();
```

```
            Arrays.sort(levelSorted, Comparators.by_Level);
            FileService.writeRegistrationsToFile(levelSorted, "Sorted_By_Level.txt");
                        System.out.println("Saved  academic  level  sorted  data  to
Sorted_By_Level.txt");

            // Sort by Registration Time
            Registration[] timeSorted = registrations.convToArray();
            Arrays.sort(timeSorted, Comparators.by_RegTime);
            FileService.writeRegistrationsToFile(timeSorted, "Sorted_By_Time.txt");
                      System.out.println("Saved  registration  time  sorted  data  to
Sorted_By_Time.txt");
        }

        // Calculate scores for all registrations
        public static void calculateAllDemandScores(LinkedList registrations) {
            Node current = registrations.head;
            while (current != null) {
                DemandScorer.computeDemandScore(current.studData);
                current = current.next;
            }
        }
    }
```

Sorter Interface

```
    public interface Sorter {
        void sort(Registration[] a);
        String name();
    }
```

Node Class

```
    public class Node {
        Registration studData; // Field to store Registration data
        Node next; // Field to store next Node reference
```

```
    // Constructor to initialize node with Registration
    public Node(Registration studData) {
        this.studData = studData;
        this.next = null;
    }
}
```

Timer Class

```
public class Timer {
    // Private variables for safety, no other class can access these directly and affect the elapsed time.
    private long startTime;
    private long endTime;

    // Method to start timer
    public void start() {
        startTime = System.currentTimeMillis(); // Simpler than nanoTime
    }

    // Method to stop timer
    public void stop() {
        endTime = System.currentTimeMillis();
    }

    // Method to get elapsed time in milliseconds
    public long getElapsedTimeMillis() {
        return endTime - startTime;
    }
}
```

LinkedList Class

```
public class LinkedList {
```

```
// Field to store head node
   Node head;
// Field to store size of list
   int size;
// Field to store tail node for faster inserts
   Node tail;

// Method to insert node at head
   public void insertNodeAtHead(Registration data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
            size++;
            return;
        }
        newNode.next = head;
        head = newNode;
        size++;
   }

// Method to insert node at tail
   public void insertNodeAtTail(Registration data) {

        Node newNode = new Node(data);
        //Case 1: handle special case (if node is empty)
        if (head == null) {

            head = newNode;
            tail = newNode;
            size++;
            return;
        }
        tail.next = newNode;
        tail = newNode;
        size++;
   }
// Method to delete node by value

public void deleteNodeByValue(Registration data) {
        //Case 1: empty list
```

```
        if (head == null) {
            return;
        }

        //Case 2: if head node is to be deleted
        if (head.studData.equals(data)) {
            head = head.next;
            size--;
            return;
        }

        //Case 3: if any other node is to be deleted
        Node prev = head;
        Node curr = head.next;

        while (curr != null) {
            if (curr.studData.equals(data)) {
                prev.next = curr.next;
                size--;
                return;
            }
            prev = curr;
            curr = curr.next;
        }
    }

// Method to search for a node
    public boolean searchKey(Registration data) {
        Node temp = head;
        while (temp.next != null) {
            if (temp.studData.equals(data)) {
                return true;
            }
            temp = temp.next;
        }
        return false;
    }

// Method to display/traverse all nodes
    public void displayLinkedList() {
        Node temp = head;
```

```
            while (temp != null) {
                System.out.println(temp.studData);
                temp = temp.next;
            }
        }
    // Method to convert list to array (Registration[])
        public Registration[] convToArray() {
            // Count number of nodes
            int size = 0;
            Node temp = head;
            while (temp != null) {
                size++;
                temp = temp.next;
            }

            // Create an array of that size
            Registration[] dataArray = new Registration[size];

            // Traverse through node to store into array
            temp = head;
            int index = 0;
            while (temp != null) {
                dataArray[index++] = temp.studData;
                temp = temp.next;
            }
            return dataArray;
        }
    }
```

Benchmarker Class

```
    public class Benchmarker {
        // Helper method to make a copy of the array
        private static Registration[] makeCopy(Registration[] original) {
              Registration[] copy = new Registration[original.length]; // Create new
empty array
            for (int i = 0; i < original.length; i++) { // Loop through each element
                copy[i] = original[i]; // Copy each element
```

```
            }
            return copy; // Return the copy
        }

        // Runs all 4 sorting algorithms and times them.
        public static void runAllSorts(Registration[] originalArray) {
            System.out.println("--- RUNNING ALL SORTS ---");

            Timer timer = new Timer();

            // 1. SELECTION SORT
            Registration[] copy1 = makeCopy(originalArray); // Make fresh copy.
            timer.start(); // Start timer.
            new SelectionSort().sort(copy1); // Do the sorting + create an object.
            timer.stop(); // Stop timer
            // File writing moved OUTSIDE timed section to measure only sorting time
               FileService.writeRegistrationsToFile(copy1, "Sorted_Output_SS.txt"); //
Save result.
              System.out.println("Selection Sort: " + timer.getElapsedTimeMillis() + "
ms");
            // 2. INSERTION SORT
            Registration[] copy2 = makeCopy(originalArray); // Fresh copy again.
            timer.start();
            new InsertionSort().sort(copy2); // Do the sorting + create an object.
            timer.stop();
            // File writing moved OUTSIDE timed section to measure only sorting time
               FileService.writeRegistrationsToFile(copy2, "Sorted_Output_IS.txt"); //
Save result.
              System.out.println("Insertion Sort: " + timer.getElapsedTimeMillis() + "
ms");
            // 3. MERGE SORT
            Registration[] copy3 = makeCopy(originalArray); // Fresh copy again.
            timer.start();
            new MergeSort().sort(copy3); // Do the sorting.
            timer.stop();
            // File writing moved OUTSIDE timed section to measure only sorting time
               FileService.writeRegistrationsToFile(copy3, "Sorted_Output_MS.txt"); //
Save result.
            System.out.println("Merge Sort: " + timer.getElapsedTimeMillis() + " ms");

            // 4. QUICK SORT
```

```
            Registration[] copy4 = makeCopy(originalArray); // Fresh copy again.
            timer.start();
            new QuickSort().sort(copy4); // Do the sorting + create an object.
            timer.stop();
            // File writing moved OUTSIDE timed section to measure only sorting time
               FileService.writeRegistrationsToFile(copy4, "Sorted_Output_QS.txt"); //
Save result.
            System.out.println("Quick Sort: " + timer.getElapsedTimeMillis() + " ms");
        }
    }
```

Comparator Class

```
    import java.util.Comparator;

    public final class Comparators {
        // Private constructor to prevent instantiation (utility class)
        private Comparators() {
        }

        // Comparator to sort Registration by studentId (ascending)
                public    static    final    Comparator<Registration>    by_ID    =    new
Comparator<Registration>() {
            @Override
            public int compare(Registration r1, Registration r2) {
                return r1.getStudentID().compareTo(r2.getStudentID());
            }
        };

        // Comparator to sort Registration by courseCode (A-Z)
             public    static    final    Comparator<Registration>    by_CourseCode    =    new
Comparator<Registration>() {
            @Override
            public int compare(Registration r1, Registration r2) {
                return r1.getCourseID().compareTo(r2.getCourseID());
            }
        };
        // Comparator to sort Registration by academicLevel (ascending)
```

```
            public static final Comparator<Registration> by_Level = new
Comparator<Registration>() {
          @Override
          public int compare(Registration r1, Registration r2) {
              return Integer.compare(r1.getAcademicLevel(), r2.getAcademicLevel());
          }
      };

      // Comparator to sort Registration by regTime (ascending)
            public static final Comparator<Registration> by_RegTime = new
Comparator<Registration>() {
          @Override
          public int compare(Registration r1, Registration r2) {
              return Integer.compare(r1.getRegTime(), r2.getRegTime());
          }
      };
        // Static helper to get comparator by user choice (e.g., "id", "name",
"course")
      public static Comparator<Registration> getComparator(String criterion) {
          switch (criterion.toLowerCase()) { // Case insensitive matching
              case "id":
                  return by_ID;
              case "course":
                  return by_CourseCode;
              case "level":
                  return by_Level;
              case "time":
                  return by_RegTime;
              default:
                  throw new IllegalArgumentException("Unknown sorting criterion: " +
criterion);
          }
      }
  }
```

Registration Class

```
  public class Registration {
```

```
    private String studentID;
    private String courseID;
    private int academicLevel; // Field to store class/group
    private int regTime; // Field to store time
    private double studDemandScore;

    // Constructor to initialize fields
     public Registration(String studentID, String courseID, int academicLevel, int
regTime, double studDemandScore) {
        this.studentID = studentID;
        this.courseID = courseID;
        this.academicLevel = academicLevel;
        this.regTime = regTime;
          this.studDemandScore = 0; // temporary value prior to processing demand
score (score hasn't been recorded yet)
    }

    // Getters and setters for all fields
    public String getStudentID() {
        return studentID;
    }

    public void setStudentID(String studentID) {
        this.studentID = studentID;
    }

    public String getCourseID() {
        return courseID;
    }

    public void setCourseID(String courseID) {
        this.courseID = courseID;
    }

    public int getAcademicLevel() {
        return academicLevel;
    }

    public void setAcademicLevel(int academicLevel) {
        this.academicLevel = academicLevel;
    }
```

```
    public int getRegTime() {
        return regTime;
    }

    public void setRegTime(int regTime) {
        this.regTime = regTime;
    }

    public double getStudDemandScore() {
        return studDemandScore;
    }

    public void setStudDemandScore(double studDemandScore) {
        this.studDemandScore = studDemandScore;
    }

    // toString() method to format output line
    @Override
    public String toString() {
         return studentID + ";" + courseID + ";" + academicLevel + ";" + regTime +
";" + Math.round(studDemandScore)
                + ";";
     } // Math.round function used to round the demand score to the nearest whole
number
  }
```

DemandScorer Class

```
  public class DemandScorer {
    public static void computeDemandScore(Registration reg) {
        double score = 50; // Base score

        score = applyStudentClassMultiplier(score, reg.getAcademicLevel());
        score = applyTimePreferenceMultiplier(score, reg.getRegTime());
        score = applyCoursePriority(score, reg.getCourseID());

        score = Math.round(score); // Round to nearest whole number.
```

```
            score = Math.max(0, Math.min(100, score)); // Stop between 0-100.

            reg.setStudDemandScore(score); // Save the final score to the Registration
object.
        }

            private  static  double  applyStudentClassMultiplier(double  score,  int
studentClass) {
            if (studentClass == 1) {
                score = score * 0.75; // Freshman bonus
            } else if (studentClass == 2) {
                score = score * 0.90; // Sophomore bonus
            } else if (studentClass == 3) {
                score = score * 1.10; // Junior bonus
            } else if (studentClass == 4) {
                score = score * 1.25; // Senior bonus
            } else if (studentClass == 5) {
                score = score * 1.5; // Graduate bonus
            }
            return score;
        }

        public static double applyTimePreferenceMultiplier(double score, int time) {
            if (time >= 6 && time < 8) {
                score += 5;
            } else if (time >= 8 && time < 10) {
                score += 10; // 8:00 - 10:00: +10
            } else if (time >= 10 && time < 12) {
                score += 5; // 10:00 - 12:00: +5
            } else if (time >= 12 && time < 14) {
                score += 0; // 12:00 - 14:00: No change.
            } else if (time >= 14 && time < 16) {
                score -= 5; // 14:00 - 16:00: -5
            } else if (time >= 16 && time < 20) {
                score -= 10; // 16:00 - 20:00: -10
            } else {
                score -= 15; // After 20:00: -15 (also covers 0-6 AM).
            }
            return score;
        }
```

```
    private static double applyCoursePriority(double score, String courseID) {
        // Get the last character of the course code.
        char lastChar = courseID.charAt(courseID.length() - 1);

        // Check if it's a digit and whether it's odd/even.
        if (Character.isDigit(lastChar)) {
            int lastDigit = Integer.parseInt(String.valueOf(lastChar)); // Convert
char to int
            if (lastDigit % 2 == 1) { // Odd number.
                score += 20;
            } else { // Even number.
                score -= 10;
            }
        }
        return score;
    }
}
```

InputValidator Class

```
import java.io.*;
import java.util.*;

public class InputValidator {

   private static Set<String> validCourses = new HashSet<>(); // Empty list to store
valid course codes extracted from
                                                              // text file

   // Method to validate file path exists and is readable.
   public static boolean isValidFilePath(String filePath) {
       File file = new File(filePath);
       return file.exists() && file.canRead();
   }

   // Method to read and validate input file.
   public static LinkedList readAndValidateFile(String input) {
       LinkedList registrations = new LinkedList(); // Create instance of Layan's
```

```
list (LinkedList class).

        // Check if the file exists and is readable to avoid FileNotFoundException.
        if (!isValidFilePath(input)) {
            System.out.println(
                    "ERROR: File '" + input + "' not found or not readable. Please
check that the file exists.");
            return registrations; // Stop the entire method
        }
        // If file is found, proceed to the BufferedReader section to actually read
the
        // file.
        try (BufferedReader list = new BufferedReader(new FileReader(input))) { //
BufferedReader to read the input file
                                                                                //
more efficiently. It reads larger
                                                                                //
chunks of data at a time, reducing
                                                                                //
the number of I/O operations.
            String line;
            while ((line = list.readLine()) != null) { // readLine is like nextLine
but for BufferedReader.
                String[] parts = line.split(";"); // Split line into parts based on
semicolon.
                if (parts.length != 4) { // We are expecting exactly 4 fields per
line.
                    System.out.println("Invalid entry (wrong number of fields): " +
line);
                    continue; // Skip invalid entry
                }
                // Example line: 2021555;CS222;2;10;
                String studentID = parts[0];
                String courseID = parts[1];

                // We need to parse integers to convert from String.
                try {
                    int academicLevel = Integer.parseInt(parts[2]);
                    int studyTime = Integer.parseInt(parts[3]);

                    // Validate non-empty string (for names, course titles)
```

```
                    if (studentID.isEmpty() || courseID.isEmpty()) {
                        System.out.println("Invalid string field in entry: " + line);
                        continue; // Skip invalid entry
                    }
                    // Validate studentId.
                    if (studentID.length() != 7 && studentID.length() != 5 ) { //
Assuming student ID should be exactly 7 or 5 characters.
                        System.out.println("Invalid student ID in entry: " + line);
                        continue; // Skip invalid entry
                    }
                    // Validate credits/section as positive integers within range.
                    if (academicLevel < 1 || academicLevel > 5) { // Student class
should be 1-5.
                        System.out.println("Invalid academic level in entry: " +
line);
                        continue; // Skip invalid entry
                    }

                    if (studyTime < 0 || studyTime > 23) { // Assuming time is in
24-hour format
                        System.out.println("Invalid study time in entry: " + line);
                        continue; // Skip invalid entry
                    }

                    // If all validations pass, add courseID to validCourses Hashset.
(Previously it
                    // was empty).
                    validCourses.add(courseID);

                    // Create a Registration object and add it to the linked list.
                    Registration reg = new Registration(studentID, courseID,
academicLevel, studyTime, 0);
                    registrations.insertNodeAtTail(reg);
                    System.out.println("VALID: " + line);
                } catch (NumberFormatException e) {
                    System.out.println("Invalid number format in entry: " + line);
                    continue; // Skip invalid entry.
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
```

```
        }
        return registrations;
    }
}
```

FileService Class

```
import java.io.*;

public class FileService {

    // Method to read registrations from Input.txt into LinkedList
    public static LinkedList readRegistrationsFromFile(String filename) {
        return InputValidator.readAndValidateFile(filename);
    }

    // Method to write array of registrations to Output.txt
      public static void writeRegistrationsToFile(Registration[] array, String
filename) {

        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter(filename));

            for (Registration reg : array) {
                if (reg != null) {// Checks for null entries to avoid file writing
loop from crashing with NullPointerException
                    writer.write(reg.toString());
                    writer.newLine();
                }
            }
            writer.close();
                  System.out.println("Registrations successfully written to " +
filename);
        } catch (IOException e) {
             System.out.println("An error occurred while writing to the file: " +
e.getMessage());
        }
    }
```

```
    }
```

Insertion Sort Class

```
public class InsertionSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        // Start from the second element (index 1).
        for (int i = 1; i < array.length; i++) {
            Registration current = array[i];
             double currentScore = current.getStudDemandScore(); // Get the demand
score of the current element for comparison.
             int j = i - 1; // Start comparing with the element immediately to the
left.
            // Shift elements that are smaller than current to the right.
            while (j >= 0 && array[j].getStudDemandScore() < currentScore) {
                array[j + 1] = array[j]; // Shift element to the right.
                j--;
            }
            // Insert current element in correct position.
            array[j + 1] = current;
        }
    }

    @Override
    public String name() {
        return "Insertion Sort";
    }
}
```

Merge Sort Class

```
public class MergeSort implements Sorter {
```

```
    @Override
    public void sort(Registration[] array) {
        if (array.length < 2) {
            return; // Base case: arrays of size 0 or 1 are already sorted
        }

        // Split the array into two halves.
        int mid = array.length / 2;
        Registration[] left = new Registration[mid];
        Registration[] right = new Registration[array.length - mid];

        // Copy the first half of the original array into the left array.
        for (int i = 0; i < mid; i++) {
            left[i] = array[i]; // Copy elements from start to mid-1.
        }

        // Copy the second half of the original array into the right array.
        for (int i = mid; i < array.length; i++) {
            right[i - mid] = array[i]; // Copy elements from mid to end.
            // i-mid converts the index (ex. if mid = 2, i = 2 becomes right[0]).
        }

        // Recursively sort both halves.
        sort(left);
        sort(right);

        // Merge the sorted halves back together.
        merge(array, left, right);
    }

    private void merge(Registration[] result, Registration[] left, Registration[]
right) {
        // Three pointers:
        int i = 0; // Tracks position in left array
        int j = 0; // Tracks position in right array
        int k = 0; // Tracks position in result array

           // Compare elements from left and right arrays, take the larger one
(descending
        // order).
        while (i < left.length && j < right.length) {
```

```
            if (left[i].getStudDemandScore() >= right[j].getStudDemandScore()) {
                result[k++] = left[i++];
            } else {
                result[k++] = right[j++];
            }
        }
        // Copy any remaining elements from the left array.
        while (i < left.length) {
            result[k++] = left[i++];
        }
        // Copy any remaining elements from the right array.
        while (j < right.length) {
            result[k++] = right[j++];
        }
    }

    @Override
    public String name() {
        return "Merge Sort";
    }
}
```

Quick Sort Class

```
public class QuickSort implements Sorter {

    @Override
    public void sort(Registration[] array) {
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(Registration[] array, int low, int high) {
        if (low < high) {
            // Partition the array and get the pivot index.
            int pivotIndex = partition(array, low, high);

            // Recursively sort elements before and after pivot.
            quickSort(array, low, pivotIndex - 1);
```

```
            quickSort(array, pivotIndex + 1, high);
        }
    }

    private int partition(Registration[] array, int low, int high) {
        // Choose the rightmost element as pivot.
        double pivot = array[high].getStudDemandScore();
        int i = low - 1; // Index of smaller element.

        for (int j = low; j < high; j++) {
                // If the current element is greater than or equal to pivot
(descending order).
            if (array[j].getStudDemandScore() >= pivot) {
                i++;
                // Swap array[i] and array[j]
                Registration temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        // Swap array[i+1] and array[high] (put pivot in correct position).
        Registration temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }

    @Override
    public String name() {
        return "Quick Sort";
    }
}
```

Selection Sort Class

```
public class SelectionSort implements Sorter {
```

```
    @Override
    public void sort(Registration[] array) {
        // Sort in DESCENDING order (highest demand scores first).
        for (int i = 0; i < array.length - 1; i++) {
            int maxIndex = i; // Assume the current position has the highest score.

            for (int j = i + 1; j < array.length; j++) { // Look at all remaining elements to find the actual highest.
                if (array[j].getStudDemandScore() > array[maxIndex].getStudDemandScore()) {
                    maxIndex = j; // Remember this as the new highest position.
                }
            }
            // Swap
            Registration temp = array[i];
            array[i] = array[maxIndex];
            array[maxIndex] = temp;
        }
    }

    @Override
    public String name() {
        return "Selection Sort";
    }
}
```

Appendix C: Full Run Pictures

```
VALID: 2023111;EE300;3;15;
VALID: 2018222;MATH400;4;20;
VALID: 2017333;COE100;1;7;
VALID: 2020444;ARAB400;4;13;
VALID: 2021555;CS200;2;10;
VALID: 2016666;ENGL100;1;15;
VALID: 2019777;PHYS400;4;20;
VALID: 22888;ACCT100;1;7;
VALID: 2023999;EE200;2;13;
VALID: 2018000;MATH500;5;10;
VALID: 2017111;COE300;3;15;
VALID: 2020222;ARAB200;2;20;
VALID: 2021333;CS400;4;7;
VALID: 2016444;ENGL300;3;13;
VALID: 2019555;PHYS100;1;10;
VALID: 2022666;ACCT300;3;15;
VALID: 23777;EE400;4;20;
VALID: 2018888;MATH200;2;7;
VALID: 2017999;COE100;1;13;
VALID: 2020000;ARAB300;3;10;
VALID: 2021111;CS500;5;15;
```

```
File reading completed in: 106 ms
Successfully read 3432 registrations.

Starting demand score calculations...
Demand score calculations completed in: 98 ms

Writing unsorted data to file...
Registrations successfully written to Output.txt
File writing completed in: 48 ms
Saved unsorted data to Output.txt

Starting sorting algorithms...
--- RUNNING ALL SORTS ---
Registrations successfully written to Sorted_Output_SS.txt
Selection Sort: 45 ms
Registrations successfully written to Sorted_Output_IS.txt
Insertion Sort: 26 ms
Registrations successfully written to Sorted_Output_MS.txt
Merge Sort: 17 ms
Registrations successfully written to Sorted_Output_QS.txt
Quick Sort: 22 ms
All sorting completed and results saved.


--- ADDITIONAL COMPARATOR SORTING ---
Registrations successfully written to Sorted_By_ID.txt
Saved student ID sorted data to Sorted_By_ID.txt
Registrations successfully written to Sorted_By_Course.txt
Saved course code sorted data to Sorted_By_Course.txt
Registrations successfully written to Sorted_By_Level.txt
Saved academic level sorted data to Sorted_By_Level.txt
Registrations successfully written to Sorted_By_Time.txt
Saved registration time sorted data to Sorted_By_Time.txt
```