Homework 2
Sorting Algorithm Performance Analysis

COMS 2280 Fall 2025

Due at **11:59 PM, Oct. 22nd**

Contents

1 Homework Overview

The "Sorting Algorithm Performance Analysis" homework is designed to provide hands-on experience with implementing and comparing the efficiency of fundamental sorting algorithms. You will implement four sorting algorithms: **Selection Sort**, **Insertion Sort**, **Merge Sort**, and **Quick Sort**.

The goal is to analyze their performance on a dataset of `Student` objects. The performance is measured by timing how long it takes each algorithm to perform two sorting passes on the data to identify a "median student" based on median Grade Point Average(GPA) and median credits. The application will support generating random student data or reading data from a file, present a performance comparison table, and optionally export the performance as a Comma Seperated Values(CSV) file.

2 Core Concepts

The analysis uses an array of `Student` objects. Each sorting algorithm will be timed, and the results will be displayed to compare their efficiency.

2.1 Student Data

Each student has two attributes:

- **GPA:** A `double` value between 0.0 and 4.0.
- **Credits Taken:** A non-negative `int` value representing the total credits.

2.2 Sorting Criteria

The sorting of students can be configured using two different criteria managed by a `Comparator`:

1. **Order 0 (Sort by GPA):** Students are sorted primarily by **GPA in descending order** (highest GPA first). If two students have the same GPA, the tie is broken by sorting by **credits taken in descending order** (higher credits first).
2. **Order 1 (Sort by Credits):** Students are sorted primarily by **credits taken in ascending order** (lowest credits first). If two students have the same number of credits, the tie is broken by sorting by **GPA in descending order** (higher GPA first).

The `Student` class also provides a different natural order using a `Comparable` interface.

2.3 The "Median Student"

The primary task of the performance analysis is to find a "median student." This is an imaginary `Student` object created from the median values of the dataset. The process is as follows:

1. The array of students is sorted using **Order 0 (by GPA)**. The student at the median index of this sorted array provides the **median GPA**.
2. The same array is sorted again using **Order 1 (by Credits)**. The student at the median index of this second sorted array provides the **median credits**.
3. A new `Student` object is constructed using the median GPA from step 1 and the median credits from step 2.

2.4 Performance Analysis

The main goal is to compare the execution time of the four sorting algorithms. The application uses `System.nanoTime()` to measure the time taken by each `sort()` method. This timing data is then presented in a summary table for comparison.

3 Assignment Tasks

3.1 Importing the Skeletal Code

You are given skeletal code containing the following files:

- `AbstractSorter.java`
- `Algorithm.java`
- `CompareSorters.java`

- `Student.java`
- `StudentScanner.java`

Import this homework into your IDE of choice. Refer to the steps from Homework 1 if you have questions.

3.2 To Do

- Review the provided skeletal code, and implement any incomplete functions or methods.
- Implement four concrete sorter classes by extending `AbstractSorter`:
  1. `SelectionSorter.java`
  2. `InsertionSorter.java`
  3. `MergeSorter.java`
  4. `QuickSorter.java`
- For each of these classes, you must:
  - Provide a constructor that calls the parent constructor for managing the `Students` objects and set the `algorithm` name.
  - Implement the `public abstract void sort()` method with the logic for that specific algorithm.
  - Use the `studentComparator` field (from `AbstractSorter`) for all element comparisons to support both sorting orders.
- `QuickSorter.java` is implementing a **median-of-three** pivot strategy and has additional methods that need to be implemented.
- Complete the main application logic in `CompareSorters.java` to handle user interaction, file I/O, and orchestrate the sorting and reporting process.
- Complete the `StudentScanner.java` class to perform the two-pass sort required to find the median student.
- Refer to the class descriptions in Section 4 and UML diagram in Figure 4to understand the method signatures and their intended use.
- Every method and field described there needs to be present in your implementation to score full points.

4 Class Hierarchy Overview

The project uses a class hierarchy to model the different sorters. The `AbstractSorter` class serves as the base for all four sorting implementations. It contains common logic, such as storing the `students` array, managing the `studentComparator`, and providing a `swap` utility method.

Each concrete sorter (`SelectionSorter`, `InsertionSorter`, etc.) extends `AbstractSorter` and provides a specific implementation for the `sort()` method.

The `StudentScanner` class uses an `AbstractSorter` polymorphically. It creates an instance of one of the concrete sorters based on the `Algorithm` enum and uses it to perform the sorting tasks without needing to know which specific algorithm is being executed.

The `CompareSorters` class acts as the main driver for the application, interacting with the user and utilizing `StudentScanner` objects to run and time the sorters.

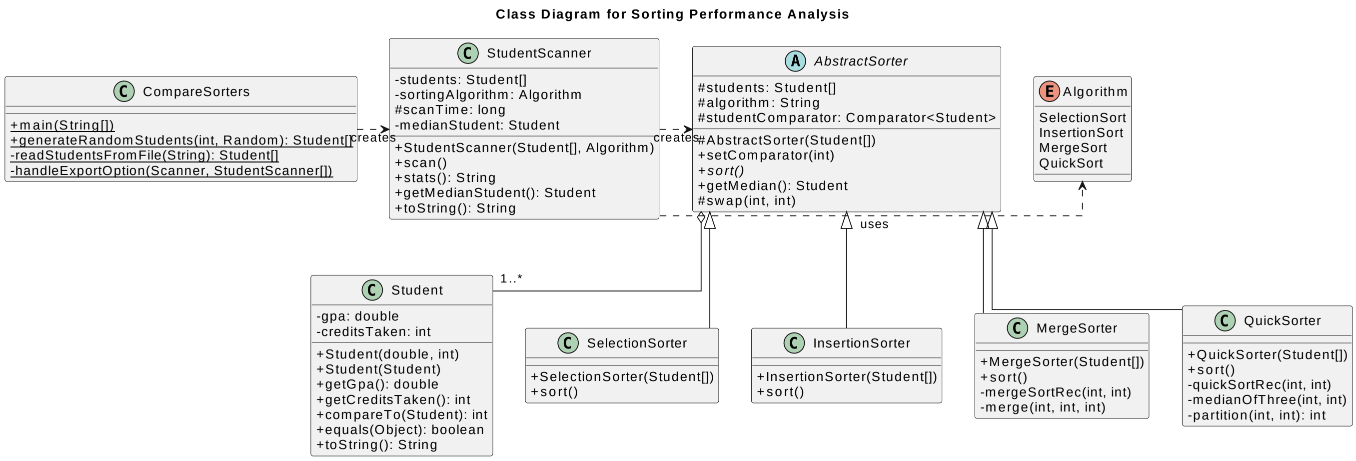
A Class Hierarchy diagram is given in Figure 4.

5 Algorithm Descriptions

The performance of each algorithm is determined by its implementation of the `sort()` method.

5.1 Selection Sort

This algorithm sorts the array by repeatedly finding the minimum element, according to the given `studentComparator`, from the unsorted part and putting it at the beginning.



5.2 Insertion Sort

This algorithm builds the final sorted array one item at a time. It iterates through the input elements and inserts each element into its correct position in the sorted part of the array.

5.3 Merge Sort

A divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves to produce the final sorted array.

5.4 Quick Sort

Another divide-and-conquer algorithm. It picks an element as a pivot and partitions the array around the pivot. The implementation uses a **median-of-three strategy** to select a better pivot and avoid worst-case performance.

6 Input/Output Format

The program interacts with the user via the console and presents an interactive menu.

- **Main Menu:** The user is presented with three choices:
  1. Generate a list of random students. The user will then be asked for the number of students to generate.
  2. Read student data from a file. The user will be asked to provide a filename.
  3. Exit the program.

  ```
  keys:  1 (random student data)  2 (file input)  3 (exit)
  ```

- **File Input:** If reading from a file, the file should contain one student per line where each line contains two values separated by whitespace. Example:

  ```
  3.85 95
  2.50 40
  4.00 120
  ```

- **Console Output:** After processing the data, the program will print a formatted table showing the performance of each of the four sorting algorithms, including the algorithm name, the size of the dataset, and the total time taken in nanoseconds. It will also display the profile of the calculated median student. Example:

  ```
  algorithm        size  time (ns)
  ------------------------------------
  SelectionSort    100   1433576
  InsertionSort    100   434557
  MergeSort        100   173697
  ```

```
QuickSort       100   160077
------------------------------------

Median Student Profile: (GPA: 2.13, Credits: 71)
```

- **CSV Export:** The user has the option to export the results to a CSV file. Example:
  `Export results to CSV? (y/n):`

7 Class Descriptions

7.1 `CompareSorters` Class

This is the main driver class. Its `main` method contains the user interaction loop. It also includes static helper methods like `generateRandomStudents` and `readStudentsFromFile`.

7.2 `StudentScanner` Class

This class orchestrates the sorting process for a single algorithm. Its `scan()` method performs two sorts on the data (one by GPA, one by credits) to find the median values and records the total time taken.

- `StudentScanner(Student[] students, Algorithm algo)`: Constructor that takes the student data and the algorithm to use.
- `scan()`: Creates an instance of the appropriate sorter, performs the two sorting passes (by GPA, then by credits), records the total time, and constructs the `medianStudent`.
- `stats()`: Returns a formatted string containing the performance results for the table.
- `getMedianStudent()`: Returns the calculated median student.

7.3 `AbstractSorter` Abstract Class

The base class for all sorters. It holds the `students` array (as a deep copy), the algorithm name, and the `Comparator<Student>`. It provides the `setComparator(int order)` method to switch between sorting criteria.

- `students`: A protected array of `Student` objects. A deep copy is made in the constructor to avoid modifying the original data.
- `studentComparator`: A protected `Comparator` used for all comparisons.
- `setComparator(int order)`: Sets the comparator to sort by GPA (order 0) or by credits (order 1).
- `sort()`: An abstract method that must be implemented by subclasses to perform the sorting.
- `getMedian()`: Returns the student at the median index of the sorted array.
- `swap(int i, int j)`: A utility method to swap two elements in the `students` array.

7.4 `Student` Class

A simple data class representing a student. It implements `Comparable<Student>` to define a natural order (by GPA, then credits), though this natural order is not used by the sorters, which rely on the dynamic comparator.

- `Student(double gpa, int creditsTaken)`: Constructor to initialize a student.
- `compareTo(Student other)`: Implements the natural ordering (by GPA descending, then credits descending).

7.5 `Algorithm` Enum

An enumeration that defines the four sorting algorithms.

- Values: `SelectionSort`, `InsertionSort`, `MergeSort`, `QuickSort`.

8 Testing

The homework must include a comprehensive suite of unit tests using JUnit 5. Each sorter class must have a test that verifies the correctness of the `sort()` method for both ordering criteria (order 0 and 1) and under various conditions, including empty arrays, arrays with one element, sorted arrays, and reverse-sorted arrays.

9 Submission

Write your classes in the `edu.iastate.cs2280.hw2` package.

9.1 Deliverables

- A zip file containing all the Java source files. Please do not turn in any generated `.class` files.
- The zip should include the `src` folder and `pom.xml` at its root.
- If you have any files associated with your JUnit tests, include them.
- The zip file should be named `FirstName_LastName.zip`.
- Make sure the zip file you are uploading is the correct version and not empty.

9.2 Important Instructions

1. Write Javadoc for documenting your homework and provide additional comments where necessary.
2. Do not change any function signatures or class names from the skeleton. Any violation will be penalized by up to a 50% reduction in score.
3. For this homework, all the private method we have specified cannot be changed and will be considered contract violation.
4. You are welcome to add additional helper methods and fields.
5. In the output table, the columns should be aligned.
6. Make sure to match any error message, input options or outputs given in the working sample.
7. If the project is not passing the Gradescope Autograder, then recheck your submission to make sure there are no compilation errors or contract violations. Contract violations can include wrong package name, folder structures on top of modifying function signatures.

9.3 Significant Dates

The homework is due on **Oct. 22nd at midnight**. You can utilize a 3-day extension without any penalty or permission. The homework may be marked as late but will not be penalized up to Oct. 25th midnight. The final day to submit the homework is Oct. 26th, which will incur a 10% penalty. No further extensions will be allowed.

You can also accrue 10% bonus points by submitting early. For each day you submit early, limited to five days (from Oct. 17th), you get a 2% bonus point. These points are calculated based on your homework's graded score, not the total possible points.

10 Working Example

Below is a sample simulation scenario. In the first trial, the user chooses option 1 to randomly generate 1000 students and choses not to export the data. In the second trial, the user reads data from a file named `students.txt` and writes the data to `perf.csv`. In the third trial, the user exits.

```
Sorting Algorithms Performance Analysis using Student Data

keys:  1 (random student data)  2 (file input)  3 (exit)
Trial 1: 1
Enter number of random students: 1000

algorithm       size  time (ns)
------------------------------------
SelectionSort   1000  23416845
InsertionSort   1000  6281591
MergeSort       1000  901032
QuickSort       1000  1847391
------------------------------------

Median Student Profile: (GPA: 1.91, Credits: 65)

Export results to CSV? (y/n): n
Trial 2: 2
File name: students.txt

algorithm       size  time (ns)
------------------------------------
SelectionSort   500   2551678
InsertionSort   500   6037005
MergeSort       500   445382
QuickSort       500   223635
------------------------------------

Median Student Profile: (GPA: 2.50, Credits: 73)

Export results to CSV? (y/n): y
Enter filename for export (e.g., results.csv): perf.csv
Data exported successfully to perf.csv

Trial 3: 1
Enter number of random students: 0
Number of students must be at least 1.
Trial 4: 1
Enter number of random students: ten
Invalid number. Please enter an integer.
Trial 5: 2
File name: t.txt
Error: File not found: t.txt
Trial 6: 2
File name: perf.csv
Error: Input file format is incorrect. File format error: Invalid GPA format. Expected a double.
Trial 7: 2
File name: empty.txt
Error: Input file format is incorrect. File is empty or contains no valid student data.
Trial 8: 2
File name: error.txt
Error: Input file format is incorrect. File format error: Invalid credits format. Expected an integer.
Trial 9: 4
Invalid choice. Please enter 1, 2, or 3.
Trial 10: 3
Exiting program.
```