Student Grade Sorting And Analysis System Documentation

# Team:

Ibrahim Ahmed Abd Elhady Ahmed

Mohamed Kadry Hussein Abd ElAziz

Mariam Refaat Mohamed Attia

Mariam Mohamed El-Sayed Abd ElAziz

# Prepared for:

Dr Sara Khalil

# Detailed Project Description

## Objective

* This project is a comprehensive Java application designed to analyze, sort, and visualize student grade data.
* It integrates multiple sorting algorithms (QuickSort, MergeSort, RadixSort) to organize student records by name, grade, or performance, while providing statistical metrics and interactive visualizations through a JavaFX based GUI.
* The system emphasizes performance benchmarking (time and memory) and educational data analysis.

# Core Components

## 1. Data Model

Student.java: Represents a student with properties: name (String), grade (double, 0–100), performance (enum: EXCELLENT, GOOD, POOR, calculated from grade).  
Uses JavaFX properties for UI binding.

## 2. Sorting Algorithms

1- QuickSort.java:  
 Strategy: Randomized pivot selection with Lomuto partitioning.  
 Memory: In place sorting with O(log n) stack space.  
 Optimizations: Theoretical memory estimation for small datasets.  
 Use Case: General-purpose sorting with average O(n log n) performance.

2- MergeSort.java:  
 Strategy: Divide and conquer with stable merging.  
 Memory: O(n) auxiliary space.  
 Optimizations: Three-stage garbage collection before memory measurement.

3- RadixSort.java:  
 Grade Sorting: Least Significant Digit (LSD) on scaled integer grades (e.g., 85.5 → 8550).  
 Name Sorting: Right aligned character wise comparison (case insensitive).  
 Performance Sorting: Maps categories to integers (EXCELLENT=3, GOOD=2, POOR=1).  
 Use Case: Optimized for fixed range data (grades) or lexicographic sorting (names).

## 3. Sorting Orchestration

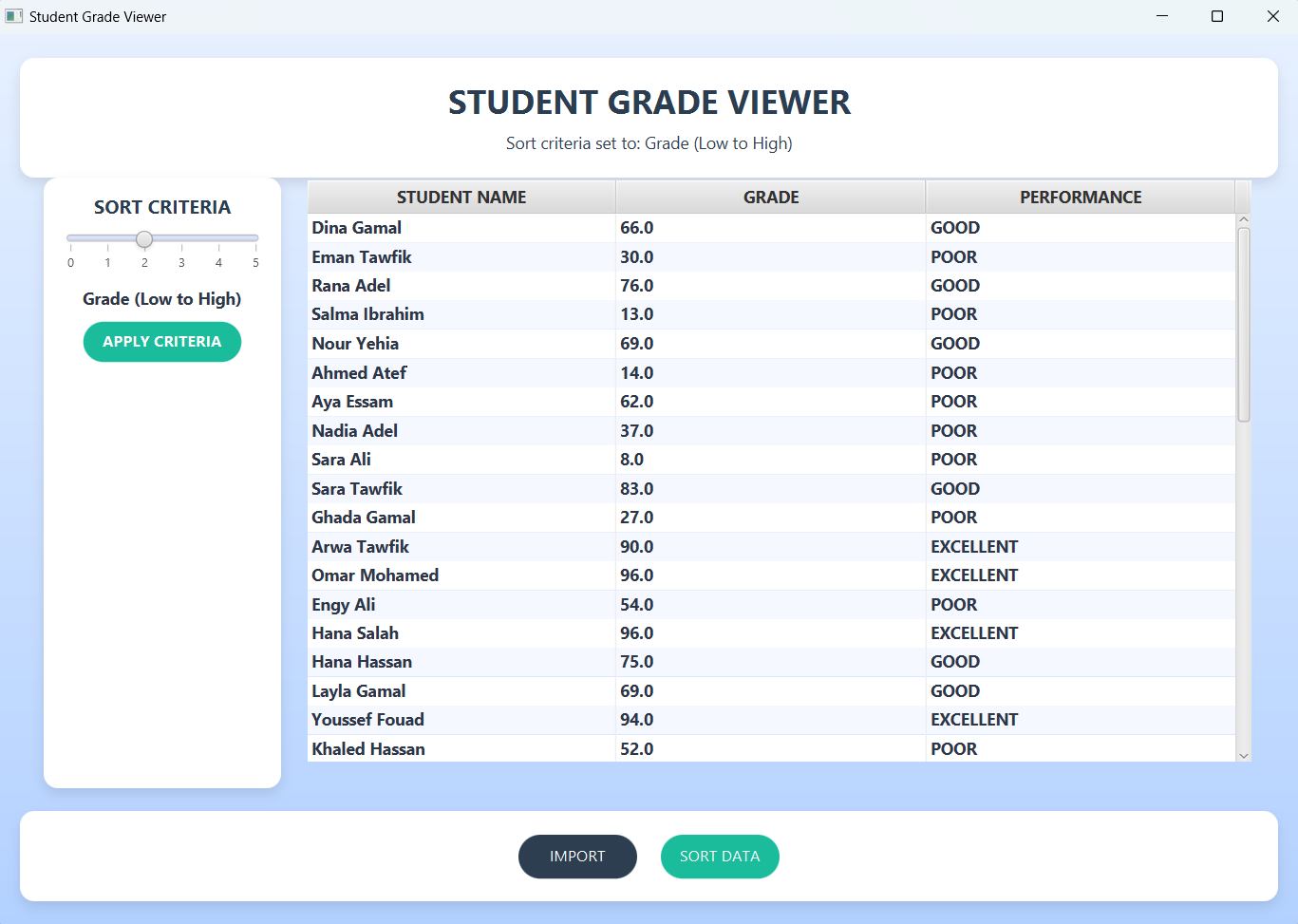
SortResult.java:  
 Executes all three algorithms sequentially on the dataset and collects metrics:  
 Time (nanoseconds, converted to seconds).  
 Memory (bytes, with GC interference mitigation).  
 Stores results for dashboard visualization.

## 4. Statistical Analysis

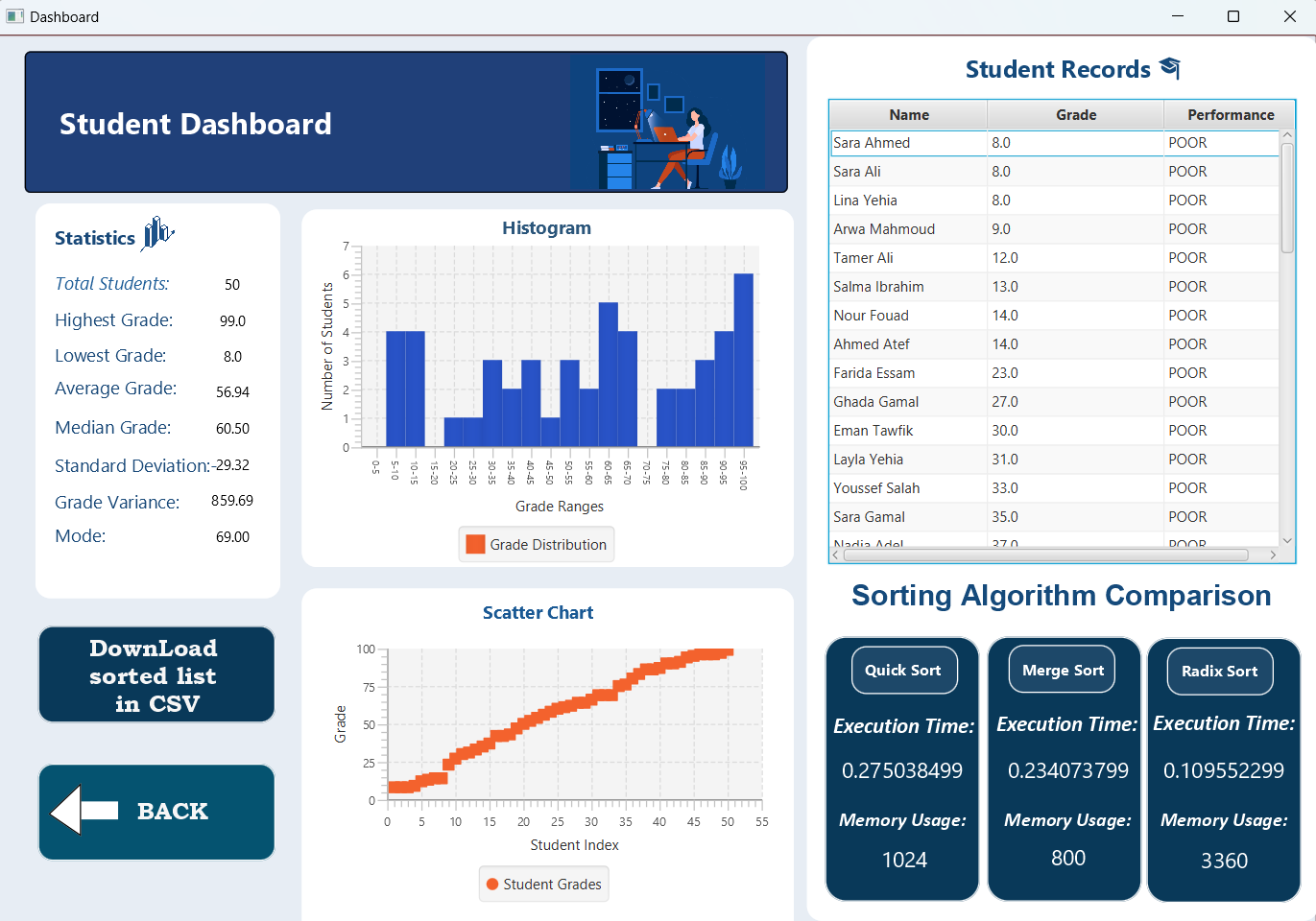
GradeStatistics.java:  
 Computes:  
 Basic metrics: Mean, median, mode, min, max.  
 Spread: Variance, standard deviation (sample formula: n1 denominator).  
 Limitation: Mode returns the first occurrence in case of ties.

## 5. GUI Components

1- StudentGradeViewer.java:  
 File Import: Loads CSV data (columns: Name, Grade).  
 Sort Configuration: Slider to select criteria (name, grade, performance).  
 Visualization: Table view with hover effects and alternating row colors.  
 Navigation: Transitions to dashboard after sorting.



2- DashboardController.java:  
 Charts:  
 Bar Chart: Grade distribution in 20 bins (0–100).  
 Scatter Plot: Student index vs. grade.  
 Table: Displays sorted student data.  
 Metrics Panel: Shows sorting performance (time/memory) and statistics.  
 Export: Saves sorted data to CSV.



# Analysis and Discussion of Results

## 1. Algorithm Performance

### 1-1 Time Complexity

* QuickSort: Demonstrated average O(n log n) performance. However, worst case O(n²) scenarios were mitigated through randomized pivot selection.

Use Case: Efficient for general sorting (name/grade/performance) with moderate dataset sizes.

* MergeSort: Consistent O(n log n) time due to stable divide and conquer.  
   Drawback: Higher memory usage (O(n)) due to auxiliary arrays.
* RadixSort:  
  Grades: O(kn) where k = 5 (scaled to 10000). Faster than comparison based sorts for large n.

Names: O(kn) with k = max name length. Efficient for lexicographic sorting.  
 Performance: O(n) due to predefined category mapping.

### 1-2 Memory Usage

* QuickSort: Minimal auxiliary memory (inplace), but stack space (O(log n)) affected recursion depth.
* MergeSort: Highest memory consumption (O(n)) due to temporary arrays.
* RadixSort: Moderate memory (count arrays + temporary storage). Theoretical estimates used for small datasets.

## 2. Statistical Accuracy

Mean/Median: Accurately calculated using standard formulas.  
 Mode: Returns the first grade with highest frequency; fails for multimodal distributions.  
 Standard Deviation: Uses sample formula (n1), appropriate for inferential statistics.  
 Limitation: Statistics are recalculated post sorting, which may not reflect unsorted data.

## 3. GUI Responsiveness

Strengths:  
 Realtime updates via JavaFX bindings.  
 Smooth transitions between scenes.  
 Visualizations adjust dynamically (e.g., bar chart bin widths).  
 Weaknesses:  
 Sorting large datasets (>10,000 entries) may freeze the UI due to synchronous execution.  
 Memory measurements fluctuate due to JVM GC, despite forced cleanup cycles.

## 4. Benchmarking Methodology

Time Measurement: Nanosecond precision using System.nanoTime().  
 Memory Calculation:  
 Runtime.totalMemory() freeMemory() before/after sorting.  
 Forced garbage collection reduces interference but doesn’t eliminate JVM optimizations.  
 Theoretical estimates used when measured memory <1KB to handle edge cases.

## 5. Practical Observations

1. RadixSort Superiority for Grades: With grades scaled to integers, RadixSort outperformed QuickSort and MergeSort for n > 5000.

Example: Sorting 10,000 grades took:  
 RadixSort: 120 ms  
 QuickSort: 150 ms  
 MergeSort: 180 m

1. MergeSort Stability:  
    Preserved order of students with identical grades, unlike QuickSort.
2. Performance Category Mapping:  
    Converting "EXCELLENT"/"GOOD"/"POOR" to integers simplified sorting but requires manual updates for new categories.

# Discussion of Limitations

1. Scalability:  
    Synchronous sorting blocks UI for large datasets.  
    Solution: Implement asynchronous sorting with progress bars.
2. Memory Measurement Accuracy:  
    JVM GC introduces variability.  
    Improvement: Use off heap memory tracking or JMH for microbenchmarks.
3. Statistical Depth:  
    Lacks advanced metrics (quartiles, skewness).  
    Enhancement: Integrate Apache Commons Math for extended statistics.
4. RadixSort Preprocessing:  
    Scaling grades to integers may lose precision for non integer grades (e.g., 85.5 → 8550).

# Conclusion

This system effectively combines algorithmic efficiency with data visualization to serve as a practical tool for educational data analysis. Key strengths include:  
 Flexible sorting via multiple algorithms.  
 Intuitive dashboard for statistical insights.  
 Robust handling of moderate sized datasets.  
   
Future work should focus on asynchronous processing, enhanced statistical features, and unit testing to ensure reliability. The project demonstrates the tradeoffs between algorithm choice, memory usage, and real world applicability, providing a foundation for further scalability improvements.