ASSIGNMENT 2: ALGORITHMIC ANALYSIS AND PEER CODE REVIEW

Learning Goals

- Implement fundamental sorting and array algorithms with proper asymptotic analysis
- Apply rigorous complexity analysis using Big-O, Big-Theta, and Big-Omega notations for best/worst/average cases
- Conduct professional peer code review focusing on algorithmic efficiency and optimization opportunities
- Validate theoretical analysis through empirical measurements and identify performance bottlenecks
- Communicate findings via comprehensive analysis reports and maintain clean Git workflow

1) ALGORITHM PAIRS (REQUIRED - WORK IN PAIRS)

Each student in a pair implements **ONE** algorithm from their assigned pair. Students then swap implementations for analysis.

Pair 1: Basic Quadratic Sorts

- **Student A:** Insertion Sort (with optimizations for nearly-sorted data)
- **Student B:** Selection Sort (with early termination optimizations)

Pair 2: Advanced Sorting Algorithms

- **Student A:** Shell Sort (implement multiple gap sequences: Shell's, Knuth's, Sedgewick's)
- **Student B:** Heap Sort (in-place implementation with bottom-up heapify)

Pair 3: Linear Array Algorithms

- **Student A:** Boyer-Moore Majority Vote (single-pass majority element detection)
- **Student B:** Kadane's Algorithm (maximum subarray sum with position tracking)

Pair 4: Heap Data Structures

- **Student A:** Min-Heap Implementation (with decrease-key and merge operations)
- **Student B:** Max-Heap Implementation (with increase-key and extract-max operations)

2) IMPLEMENTATION REQUIREMENTS (PART 1 - INDIVIDUAL)

Code Quality Standards

- Clean, readable Java code with proper documentation
- Comprehensive unit tests covering edge cases (empty arrays, single elements, duplicates)
- Input validation and error handling
- Metrics collection (comparisons, swaps, array accesses, memory allocations)
- CLI interface for testing with different input sizes

Performance Considerations

- Implement optimizations specific to your algorithm
- Track key operations (comparisons, swaps, recursive calls)
- Memory-efficient implementations where possible
- Handle edge cases gracefully

3) PEER ANALYSIS (PART 2 - CROSS-REVIEW)

Each student analyzes their partner's implementation and produces a detailed report covering:

Asymptotic Complexity Analysis

- **Time Complexity:** Derive and justify Θ, O, Ω for best, worst, and average cases
- **Space Complexity:** Analyze auxiliary space usage and in-place optimizations
- **Recurrence Relations:** Where applicable, solve using appropriate methods

Code Review & Optimization

- **Inefficiency Detection:** Identify performance bottlenecks and suboptimal code patterns
- **Time Complexity Improvements:** Suggest algorithmic optimizations to reduce time complexity
- **Space Complexity Improvements:** Propose memory usage optimizations
- **Code Quality:** Comment on style, readability, and maintainability

Empirical Validation

- **Performance Measurements:** Run benchmarks on various input sizes ($n = 100, 1000, 10000, 100000$)
- **Complexity Verification:** Plot time vs n to confirm theoretical analysis
- **Comparison Analysis:** Compare measured performance with theoretical predictions
- **Optimization Impact:** Measure and report the effect of suggested improvements

4) REPORT REQUIREMENTS

Individual Analysis Report (PDF)

Each student submits a report analyzing their partner's algorithm:

- **Algorithm Overview** (1 page): Brief description and theoretical background
- **Complexity Analysis** (2 pages):
    - Detailed derivation of time/space complexity for all cases
    - Mathematical justification using Big-O, Θ, Ω notations
    - Comparison with partner's algorithm complexity
- **Code Review** (2 pages):
    - Identification of inefficient code sections
    - Specific optimization suggestions with rationale
    - Proposed improvements for time/space complexity
- **Empirical Results** (2 pages):
    - Performance plots (time vs input size)
    - Validation of theoretical complexity
    - Analysis of constant factors and practical performance
- **Conclusion** (1 page): Summary of findings and optimization recommendations

5) GITHUB WORKFLOW

Repository Structure

```
assignment2-[algorithm-name]/
├── src/main/java/
│   ├── algorithms/[AlgorithmName].java
│   ├── metrics/PerformanceTracker.java
│   └── cli/BenchmarkRunner.java
├── src/test/java/
│   └── algorithms/[AlgorithmName]Test.java
├── docs/
│   ├── analysis-report.pdf
│   └── performance-plots/
├── README.md
└── pom.xml
```

Branch Strategy

- **main** — only working releases (tag v0.1, v1.0)
- **feature/algorithm** — main implementation
- **feature/metrics** — performance tracking
- **feature/testing** — unit tests and validation
- **feature/cli** — command-line interface
- **feature/optimization** — performance improvements

Commit Storyline Example

- init: maven project structure, junit5, ci setup

- feat(metrics): performance counters and CSV export

- feat(algorithm): baseline [algorithm-name] implementation

- test(algorithm): comprehensive test suite with edge cases

- feat(cli): benchmark runner with configurable input sizes

- feat(optimization): [specific optimization description]

- docs(readme): usage instructions and complexity analysis

- perf(benchmark): JMH harness for accurate measurements

- fix(edge-cases): handle empty and single-element arrays

- release: v1.0 with complete implementation

6) TESTING REQUIREMENTS

Correctness Validation

- **Unit Tests:** Cover all edge cases (empty, single element, duplicates, sorted/reverse-sorted)
- **Property-Based Testing:** Verify sorting correctness across random inputs
- **Cross-Validation:** Compare results with Java's built-in implementations where applicable

Performance Testing

- **Scalability Tests:** Measure performance across input sizes 10^2 to 10^5
- **Input Distribution Tests:** Test on random, sorted, reverse-sorted, and nearly-sorted data
- **Memory Profiling:** Track memory usage patterns and garbage collection impact

Peer Testing

- **Integration Testing:** Ensure partner's code compiles and runs correctly
- **Benchmark Reproduction:** Verify reported performance measurements
- **Optimization Validation:** Test suggested improvements for correctness and performance gain

7) DELIVERABLES

Individual Submission (via GitHub)

1. **Implementation Repository:** Complete working code with clean Git history
2. **Analysis Report:** PDF analyzing partner's algorithm (8 pages max)
3. **Performance Data:** CSV files with benchmark results and plots

Pair Submission

1. **Cross-Review Summary:** Joint document comparing both algorithms
2. **Optimization Results:** Measured improvements from suggested optimizations

8) GRADING CRITERIA

- **Implementation Quality (40%):** Code correctness, testing, optimization, Git workflow
- **Analysis Depth (35%):** Theoretical complexity analysis, bottleneck identification, optimization suggestions
- **Empirical Validation (15%):** Benchmark accuracy, plot quality, theory-practice alignment
- **Communication (10%):** Report clarity, professional presentation, actionable recommendations