CCT College Dublin

Assessment Cover Page

Module Title:	Algorithms & Constructs
Assessment Title:	CA 2
Lecturer Name:	Taufique Ahmed
Student Full Name:	Rafael Valentim Ribeiro
Student Number:	2025129
Assessment Due Date:	29/11/2025
Date of Submission:	29/11/2025

Algorithm Selection Justification Report

Sorting Algorithm: Merge Sort

I decided to follow with Merge Sort as the main sorting algorithm for our school management system after thinking through a good range of options like Quick Sort, Heap Sort, and Binary Search. The main reason that made me proceed with Merge Sort was its consistent O(n log n) time complexity, no matter the situation, whether if it's best, average, or worst-case scenario. Not like Quick Sort, which can slow down to $O(n^2)$ if the pivot is not well designed, Merge Sort always performs with certain reliability, which is very important for a school system that might handle sorted data coming from other databases.

Another plus is that Merge Sort is stable, meaning it keeps the original order of employees with the same names. This is important when showing staff records, as there might be multiple people with identical names, and we want to keep the order consistent across different data views. Beyond that, Merge Sort is used in Java's standard library for sorting objects in code (Collections.sort()), which shows its practical reliability. This user-experience thinking added to real-world challenges was important for creating a quality school system.

Search Algorithm: Linear Search with Partial Matching

Even though Binary Search offers O(log n) performance compared to Linear Search's O(n), I picked Linear Search with partial matching for a better user experience.

The main reason was usability. People often don't remember exact names—they might search "summ" to find "Summers" or "buff" to find "Buffy." According to Gong. Y 2022, Binary Search needs exact matches and sorted data, which is not very practical for real-world cases. Our implementation searches first names, last names, and full names all at once with case-insensitive matching, which is amazing for day-to-day tasks.

For typical school staff sizes (up to 500 employees), Linear Search runs in a second. The O(n) complexity isn't a big deal with this dataset size. I focused on the 90% use case (partial search) rather than just optimizing the theoretical performance of a linear search—again, always thinking on the end user.

Linear Search works on unsorted data, so we don't have to sort before every search. While we do sort for display purposes, search operations don't rely on sorted data, making the system more flexible. Moreover, our implementation returns all matching employees, not just the first one. When searching "Summers," users see Buffy, Dawn, and Joyce together. Binary Search would need extra complexity to do this, since it requires an exact match to perform (Gong. Y, 2022). The trade-off was obvious at this point, we gave up O(log n) theoretical speed for O(n) practical usability.

GitHub Commits

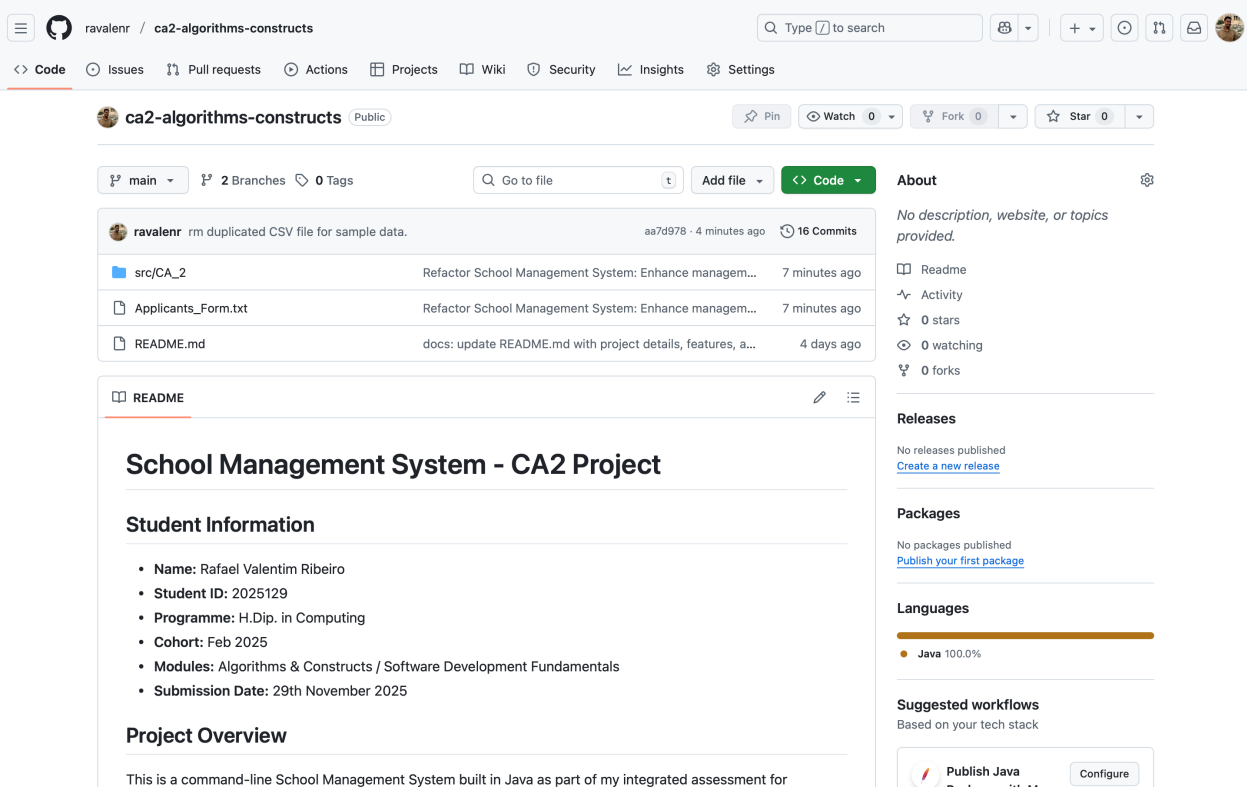

Figure 1 - Showcases comprehensive commiting while developing the school management system.

This GitHub repository can be accessed at: https://github.com/ravalenr/ca2-algorithms-constructs.

References

Oracle (2025) *Class Collections – Java Platform Standard Edition 8 Documentation*. Available at: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html (Accessed: 25 November 2025).

Gong, Y. (n.d.) *Master All Different Types of Binary Search and Their Sample Code*. Medium. Available at: https://gongybable.medium.com/master-all-different-types-of-binary-search-and-their-sample-code-ee3496b871d3(Accessed: 25 November 2025).

Appendix A – Use of Artificial Intelligence

Tools Used

• Claude (Anthropic, 2025)

In completing this assignment, I made limited use of Generative AI (Claude) as an assistive tool. The purpose of using AI was solely to support my understanding of concepts, improve the quality of my written documentation, and validate the correctness of my implementation. At no point did AI generate complete solutions, code bases, or system designs on my behalf; all final design, coding, testing, and UML modelling decisions were produced, validated, and implemented by me.

Purpose of AI Use

The roles of AI in this assignment were restricted to:

1. **Project Organization**
    - AI was used to provide a schedule of task that should be prioritize weekly so I could tackle every aspect of the requirements provided.
2. **Grammar and Formatting Improvements**
    - AI was used to proofread my documentation, improving grammar, consistency, and academic tone.
    - It did not create or alter any requirements, diagrams, test cases, or functional descriptions unless they originated from my own work.

What AI *Was Not* Used For

- AI did **not** generate full code files or project structure.
- AI did **not** design the system architecture, class relationships, UML diagrams, or use cases.
- AI did **not** create the final implementation of sorting, searching, input validation, random data generation, or menu handling.
- AI did **not** write any portion of the final report autonomously.

Student Responsibility

All Java code, UML diagrams, system design decisions, algorithm selections, test cases, and final documentation were produced by me. AI was used only as a consultative and proofreading tool, comparable to seeking clarification from an instructor or tutor, and did not replace my own reasoning, problem-solving, or implementation work.

Prompts and Responses

Prompt:
Can you go through the files and provide a plan to tackle all the tasks by 29 of November? Make sure all the requirements are cover for both modules. To get started I already initiated a github repository and shared with the professor.

Response:

Project Plan: Integrated Assignment (Due Nov 29, 2025)

Great job getting the GitHub repo started! Let me break down this assignment into a structured plan. This is a substantial project with **two modules** to cover, so organization is key.

📊 Assignment Overview

Two Modules:

- **Software Development Fundamentals (SDF)** - 55% weight: Design & Documentation
- **Algorithms & Constructs** - 60% weight: Implementation

Deliverables:

1. GitHub Repo with NetBeans Java project (10-15 commits minimum)
2. PDF document with all SDF diagrams/documentation
3. Report explaining algorithm choices (max 500 words)

🎯 Phase 1: Planning & Design (Week 1 - Nov 12-17)

Step 1.1: Choose Your Organization (Day 1)

Pick ONE: **School**, **Bank**, or **Department Store**

My Recommendation: Choose **School** - it's intuitive and easier to model:

- **Manager Types**: Principal, Vice Principal, Department Head, etc.
- **Departments**: Mathematics, Science, English, Sports, Administration
- **Employees**: Teachers, Administrative Staff, Support Staff

Step 1.2: Define Your Classes (Day 1-2)

Create at least **3 different types of each**:

Example for School:

Managers: Principal, VicePrincipal, DepartmentHead

Departments: AcademicDept, AdministrativeDept, SupportDept

Employees: Teacher, AdminStaff, SupportStaff

Step 1.3: SDF - Requirements (Day 2-3)

[20 marks]

Create requirements list:

- **Functional Requirements (minimum 5):**
    - System shall allow sorting employee records alphabetically

    - System shall enable searching employees by name
    - System shall validate manager type on input
    - System shall maintain employee hierarchy
    - System shall allow adding new employee records
- **Non-Functional Requirements (minimum 5):**
    - System shall respond to search queries within 2 seconds
    - System shall handle minimum 100 employee records
    - Code shall be maintainable with clear comments
    - System shall validate all user inputs
    - System shall display user-friendly error messages

Step 1.4: SDF - Use Cases (Day 3-5)

[20 marks - FIVE required]

Create 5 use cases with diagrams:

1. **Sort Employee List** - User sorts applicants alphabetically
2. **Search Employee** - User searches for specific employee
3. **Add New Employee** - User adds new record with validation
4. **Display Hierarchy** - User views binary tree structure
5. **View Employee Details** - User retrieves employee information

Each needs:

- Use Case diagram
- Use Case specification (Actor, Preconditions, Main Flow, Postconditions)
- Reference to code location

Step 1.5: SDF - UML Diagrams (Day 5-7)

[30 marks - TWO different types]

UML Model 1 (15 marks): Class Diagram

- Show Manager, Department, Employee parent classes
- Show 3 subtypes of each
- Include attributes and methods
- Show relationships (inheritance, composition, aggregation)
- **Add future features** (e.g., Payroll, Attendance)

UML Model 2 (15 marks): Sequence Diagram OR Activity Diagram

- **Option A:** Sequence Diagram showing "Add New Employee" interaction
- **Option B:** Activity Diagram showing main menu flow

Step 1.6: SDF - User Stories (Day 7)

[15 marks - FIVE required]

Format: "As a [role], I want [feature], so that [benefit]"

Example:

User Story 1: Sort Employees

As an HR administrator, I want to sort employee records alphabetically,

so that I can quickly locate specific employees.

Acceptance Criteria:

- System displays sorted list of first 20 employees

- Sorting completes within 3 seconds for 100 records

- Original data file remains unchanged

Step 1.7: SDF - Unit Tests (Day 7)

[15 marks - THREE required]

Use standard unit test template (NO CODE needed):

Test ID: UT-001

Test Name: testSortEmployeeList

Description: Verify sorting algorithm correctly orders employee names

Preconditions: List of 20 unsorted employee names loaded

Test Steps:

1. Load employee list from file
2. Call sorting method
3. Verify first employee name is "Adams, John"
4. Verify last employee name is "Zimmerman, Sarah"

Expected Result: List sorted in alphabetical order

Actual Result: [To be filled during execution]

Pass/Fail: [To be determined]

💻 Phase 2: Implementation (Week 2-3 - Nov 18-28)

Step 2.1: Project Setup (Day 8)

- Create NetBeans project with package name **CA_2**
- Create Applicants_Form.txt with 50+ random names
- Set up basic class structure
- **Commit #1-2**

Step 2.2: Core Classes (Day 8-9)

[Included in 20 marks for Binary Tree]

Create class hierarchy:

java

```
// Parent classes
abstract class Manager { name, id, type }
abstract class Department { name, id, managerList }
abstract class Employee { name, managerId, deptId }

// Minimum 3 subtypes each
class Principal extends Manager
class VicePrincipal extends Manager
class DepartmentHead extends Manager
```

Commit #3-4

Step 2.3: Menu with Enums (Day 10)

[10 marks]

java

```
enum MenuOption {
   SORT("Sort Employee List"),
   SEARCH("Search Employee"),
   ADD_RECORD("Add New Record"),
   BINARY_TREE("Display Hierarchy"),
   EXIT("Exit Program");

   private String description;
   // Constructor and methods
```

}

Commit #5

Step 2.4: Recursive Sorting Algorithm (Day 11-12)

[25 marks]

Recommended: Merge Sort (recursive)

Implementation:

- Read Applicants_Form.txt
- Sort alphabetically using Merge Sort
- Display first 20 names

Rationale (for your report - max 500 words):

- **Why Merge Sort?**
    - Guaranteed O(n log n) performance
    - Stable sort (maintains relative order)
    - Recursive nature required by assignment
    - Efficient for large datasets
    - Predictable performance vs Quick Sort

Alternative considered: Quick Sort (explain why NOT chosen - worst case $O(n^2)$)

Commit #6-7

Step 2.5: Searching Algorithm (Day 13-14)

[25 marks]

Recommended: Binary Search (requires sorted list)

Implementation:

- Search for employee by name
- Return Manager Type and Department
- Handle not found cases

Rationale:

- **Why Binary Search?**
    - O(log n) efficiency on sorted data
    - List already sorted from previous step
    - Significantly faster than linear search for large datasets
    - Simple implementation

Commit #8-9

Step 2.6: User Input & Validation (Day 15-16)

[20 marks]

Requirements:

- Accept: Name, Manager Choice, Department
- Validate manager type exists
- Validate department exists
- Add to list
- Display all newly added records

java

```
// Validation example
if (!validManagerTypes.contains(inputManager)) {
   System.out.println("Invalid manager type!");
   // Show valid options
}
```

Commit #10-11

Step 2.7: Binary Tree Implementation (Day 17-20)

[20 marks]

Key Requirements:

- Minimum 20 employee records
- Level-order (breadth-first) insertion
- Each node: Name, Manager Type, Department
- Display with level-order traversal
- Show tree height
- Show total node count

java

```
class TreeNode {
   String name;
   String managerType;
   String department;
   TreeNode left, right;
```

}

Implementation approach:

1. Use Queue for level-order insertion
2. Fill left child first, then right
3. Implement level-order traversal for display
4. Calculate height recursively
5. Count nodes recursively

Commit #12-15

Step 2.8: Testing & Debugging (Day 21-22)

- Test all menu options
- Verify validation works
- Test edge cases
- Ensure clean output
- Final commits

📝 Phase 3: Documentation (Day 23-24 - Nov 27-28)

Step 3.1: Algorithm Rationale Report

Max 500 words explaining:

1. Why Merge Sort was chosen (recursive requirement, stability, performance)
2. Why Binary Search was chosen (efficiency on sorted data)
3. Comparison with alternatives
4. **Don't explain HOW they work - focus on WHY you chose them**

Step 3.2: Complete SDF PDF

Compile all SDF deliverables:

- ✅ 5 Use Cases with diagrams
- ✅ 2 UML models (NOT use case diagrams)
- ✅ Requirements list
- ✅ 5 User Stories with acceptance criteria
- ✅ 3 Unit tests
- Add CCT cover page

Step 3.3: Final GitHub Prep

- Ensure 10-15 commits visible
- Add GitHub repo link to report
- Verify professor is added as collaborator
- Check all .java files have package **CA_2**

📤 Phase 4: Submission (Nov 29)

Moodle Submissions:

1. **Algorithms & Constructs:**
    - Individual .java files (all with package CA_2)
    - Report with algorithm rationale + GitHub link
2. **Software Development Fundamentals:**
    - PDF with all diagrams and documentation

⚠️ Common Pitfalls to Avoid

1. ❌ **Don't use GUI** - Command line only!
2. ❌ **Don't use Maven/Gradle** - Plain NetBeans project
3. ❌ **Don't use built-in sort()** - Implement your own recursive sort
4. ❌ **Don't forget validation** - Critical for 20 marks
5. ❌ **Don't skip commits** - Need 10-15 throughout development
6. ❌ **Don't use Use Case diagrams for UML models** - They specifically exclude these!

🎯 Success Checklist

Before Submission:

- All files have package name CA_2
- 10-15 GitHub commits done
- Professor added as collaborator
- GitHub link in report
- 5 Use Cases complete
- 2 UML diagrams (NOT use case)
- Requirements list (5 functional + 5 non-functional)
- 5 User Stories with acceptance criteria

- 3 Unit test templates
- Recursive sorting implemented
- Searching implemented
- Input validation working
- Binary tree with 20+ nodes
- Menu uses Enums
- Algorithm rationale written (max 500 words)
- Code well-commented
- Ready for Q&A on Dec 2nd

💡 My Recommendations

1. **Start with SDF documentation THIS WEEK** - It's 55% and helps you think through the design
2. **Use School as your organization** - Easiest to conceptualize
3. **Implement Merge Sort** - Perfect recursive algorithm for this task
4. **Use Binary Search** - Complements sorted data perfectly
5. **Commit regularly** - Don't wait until the end

Would you like me to help you with any specific phase? I can provide code examples, help create UML diagrams, or review your work as you progress!