Java Code Complexity Analyzer

A PROJECT REPORT

Submitted by

Student Name: Om Shrivastava
Registration Number: 24BSA10362

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING (CLOUD COMPUTING)

Course: Programming with Java (CSE2006)
Academic Year: 2025-26
Date of Submission: 23th November 2025

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
VIT BHOPAL UNIVERSITY
KOTHRIKALAN, SEHORE
MADHYA PRADESH - 466114

NOV 2025

INTRODUCTION

The Java Code Complexity Analyzer is a comprehensive tool designed to analyze the time and space complexity of Java code snippets. Understanding algorithmic complexity is the fundamental or basic to writing efficient code, yet many developers struggle to estimate the performance characteristics of their implementations.

This project provides both a console-based interface (ComplexityAnalyzer.java) and a graphical user interface (ComplexityAnalyzerGUI.java) to make complexity analysis accessible to users with different preferences. The system uses pattern recognition and static code analysis techniques to identify common algorithmic patterns such as loops, recursion, data structure allocations, and sorting operations, providing instant feedback on code efficiency.

The tool serves as an educational resource for students learning data structures and algorithms, as well as a quick reference tool for developers reviewing code performance.

PROBLEM STATEMENT

Manual analysis of code complexity requires deep understanding of algorithmic analysis and can be time-consuming, especially for beginners. Students often struggle to:

- Identify the time complexity of nested loops and recursive functions
- Understand how data structure usage affects space complexity
- Differentiate between O(n), O(n^2), O(log n), and other complexities
- Get immediate feedback on code efficiency

There is a need for an automated tool that can quickly analyze Java code and provide complexity estimates, helping developers and students understand the performance characteristics of their implementations without manual calculation.

FUNCTIONAL REQUIREMENTS

1: Code Input Module

1.1: Accept Java code as text input through console interface

1.2: Accept Java code through GUI text area with syntax-friendly monospaced font

1.3: Support multi-line code input

1.4: Validate that input is not empty before analysis

FR2: Time Complexity Analysis Module

2.1: Detect and count nested loops (for, while, do-while)

2.2: Identify sorting operations (Arrays.sort, Collections.sort)

2**.3:** Detect recursive function calls

2.4: Recognize divide-and-conquer patterns (binary search, halving)

2.5: Calculate complexity based on detected patterns:

- O(1) for constant time operations
- O(log n) for binary search patterns
- O(n) for single loops
- O(n log n) for sorting operations
- $O(n^2)$, $O(n^3)$, etc. for nested loops
- O(2^n) for recursive patterns without optimization

3: Space Complexity Analysis Module

3.1: Detect array allocations

3.2: Identify collection usage (ArrayList, LinkedList, HashMap)

3.3: Recognize recursive calls that use stack space

3.4: Determine space complexity:

- O(1) for constant space
- O(n) for linear data structures or recursion

4: Results Display Module

4.1: Display time complexity with explanation

4.2: Display space complexity with explanation

4.3: Provide user-friendly output format

4.4: Show results in console for CLI version

4.5: Display results in labeled fields for GUI version

5: User Interface Module (GUI)

5.1: Provide graphical window with title

5.2: Display scrollable text area for code input

5.3: Provide "Analyze Complexity" button

5.4: Show results panel with clear labels

5.5: Display error messages for invalid input

NON-FUNCTIONAL REQUIREMENTS

1: Performance

- Analysis should complete within 2 seconds for code snippets up to 1000 lines
- Pattern matching algorithms should be optimized for quick execution
- GUI should remain responsive during analysis

2: Usability

- Console interface should provide clear prompts and instructions
- GUI should have intuitive layout with minimal learning curve
- Error messages should be descriptive and helpful
- Results should be easy to understand with explanations

3: Reliability

- System should handle malformed code without crashing
- Should validate input before processing
- Should gracefully handle edge cases (empty input, extremely long code)
- Consistent results for the same input

4: Maintainability

- Code should be well-structured with clear separation of concerns
- Methods should be modular and reusable
- Comments should explain complex pattern matching logic
- Easy to extend with new complexity patterns

5: Portability

- Should run on any system with Java Runtime Environment (JRE 8+)
- No external dependencies required
- Platform-independent implementation

6: Scalability

- Should handle code snippets from 10 to 1000+ lines
- Pattern matching should scale linearly with code size
- Memory usage should remain reasonable for large inputs

SYSTEM ARCHITECTURE

Presentation Later(Console UI or GUI)
Complexity Analysis Engine

- Time Complexity Analyzer

- Space Complexity Analyzer
- Pattern Mathers

Utility Later

- Loop Counter
- Recursion Detector
- Pattern Recognizer

1. ComplexityAnalyzer (Console Version)

- Main entry point with Scanner for input
- Coordinates analysis workflow
- Handles console I/O

2. ComplexityAnalyzerGUI (GUI Version)

- Extends JFrame for windowing
- Uses Swing components for UI
- Event-driven architecture with button listeners

3. Analysis Engine (Shared Logic)

- analyzeTimeComplexity(): Main time analysis method
- analyzeSpaceComplexity(): Main space analysis method
- countNestedLoops(): Detects loop nesting depth
- containsRecursion(): Identifies recursive patterns
- containsDivideAndConquer(): Recognizes optimization patterns

---

DESIGN DIAGRAMS

Use Case Design:

User - Input Code - Analyze Complexity - View Result - Validate

Workflow / Sequence Diagram:

START

↓

Launch Application

↓

Display Input Interface

↓

User Enters Code

↓

Click Analyze Button

↓

Validate Input ———No———► Show - Error

| Yes

↓

Parse Code

↓

Analyze Time Complexity
- Count Loops
- DetectSorting
- Find Recursion

↓

Analyze Space Complexity

- Detect Arrays
- Find Collections
- Check Recursion

↓

Format and Display Result

↓

END

DESIGN DECISION AND RATIONALE

Approach

- Simpler implementation suitable for educational purposes
- Faster analysis for most common code patterns
- No external dependencies required
- Sufficient accuracy for typical algorithmic patterns
- Easy to understand and maintain

Dual Interface Design

- Console version provides lightweight, quick analysis
- GUI version offers better usability for less technical users
- Demonstrates versatility in interface design
- Allows users to choose based on their workflow
- Both share core analysis logic (DRY principle)

Heuristic-Based Analysis

- Perfect complexity analysis is computationally undecidable
- Heuristics provide "good enough" estimates for common patterns
- Trade-off between accuracy and implementation complexity
- Practical for educational and quick reference purposes
- Can be extended with more patterns over time

String-Based Code Processing

- No compilation step required
- Can analyze code snippets, not just complete programs
- Immediate feedback without build process

- Language-agnostic approach (could extend to other languages)

Swing for GUI

- Built into standard JDK (no additional dependencies)
- Mature and stable framework
- Sufficient for simple GUI requirements
- Wider compatibility across Java versions
- Familiar to most Java developers

IMPLEMENTATION DETAILS

Core Algorithms

Loop Nesting Detection Algorithm:

```
None
1. Initialize maxDepth = 0, currentDepth = 0
2. Split code into lines
3. For each line:
   a. Check if line starts a loop (for/while/do)
   b. If yes, increment currentDepth
   c. Update maxDepth if currentDepth > maxDepth
   d. Check if line contains closing brace
   e. If yes, decrement currentDepth
4. Return maxDepth
```

Recursion Detection Algorithm:

```
None
1. Extract all method names using regex pattern
2. For each method:
   a. Search for calls to the same method name
   b. Ensure the call is after the method definition
   c. If found, return true (recursion detected)
3. Return false if no recursion found
```

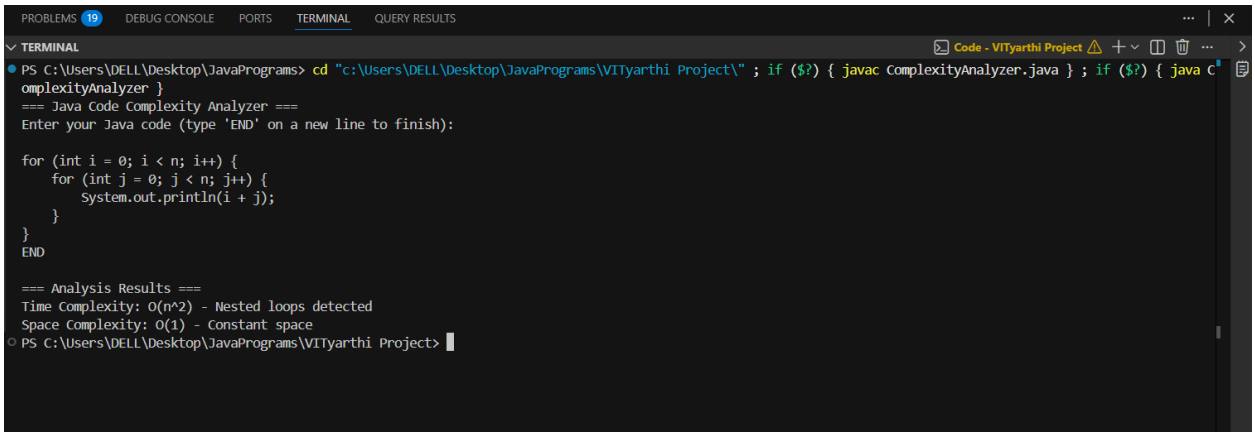
Time Complexity Determination Logic:

```
None
IF sorting operations detected:
    RETURN O(n log n)
ELSE IF 3+ nested loops:
    RETURN O(n^depth)
ELSE IF 2 nested loops:
    RETURN O(n²)
ELSE IF 1 loop:
    IF divide-and-conquer pattern:
        RETURN O(log n)
    ELSE:
        RETURN O(n)
ELSE IF recursion detected:
    IF divide-and-conquer:
```

```
        RETURN O(log n) or O(n log n)
    ELSE:
        RETURN O(2^n)
ELSE:
    RETURN O(1)
```

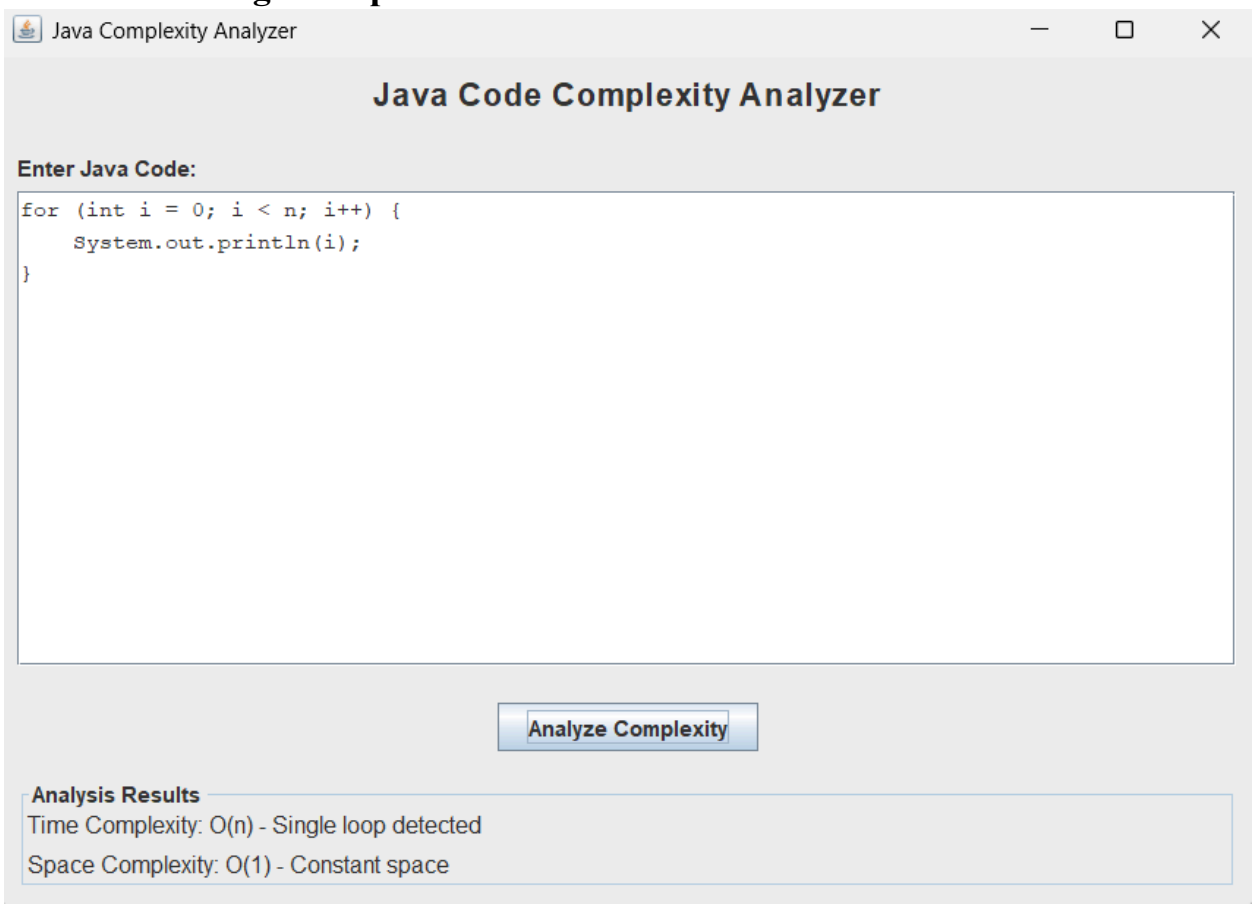
SCREENSHORTS / RESULT

1. Console Version



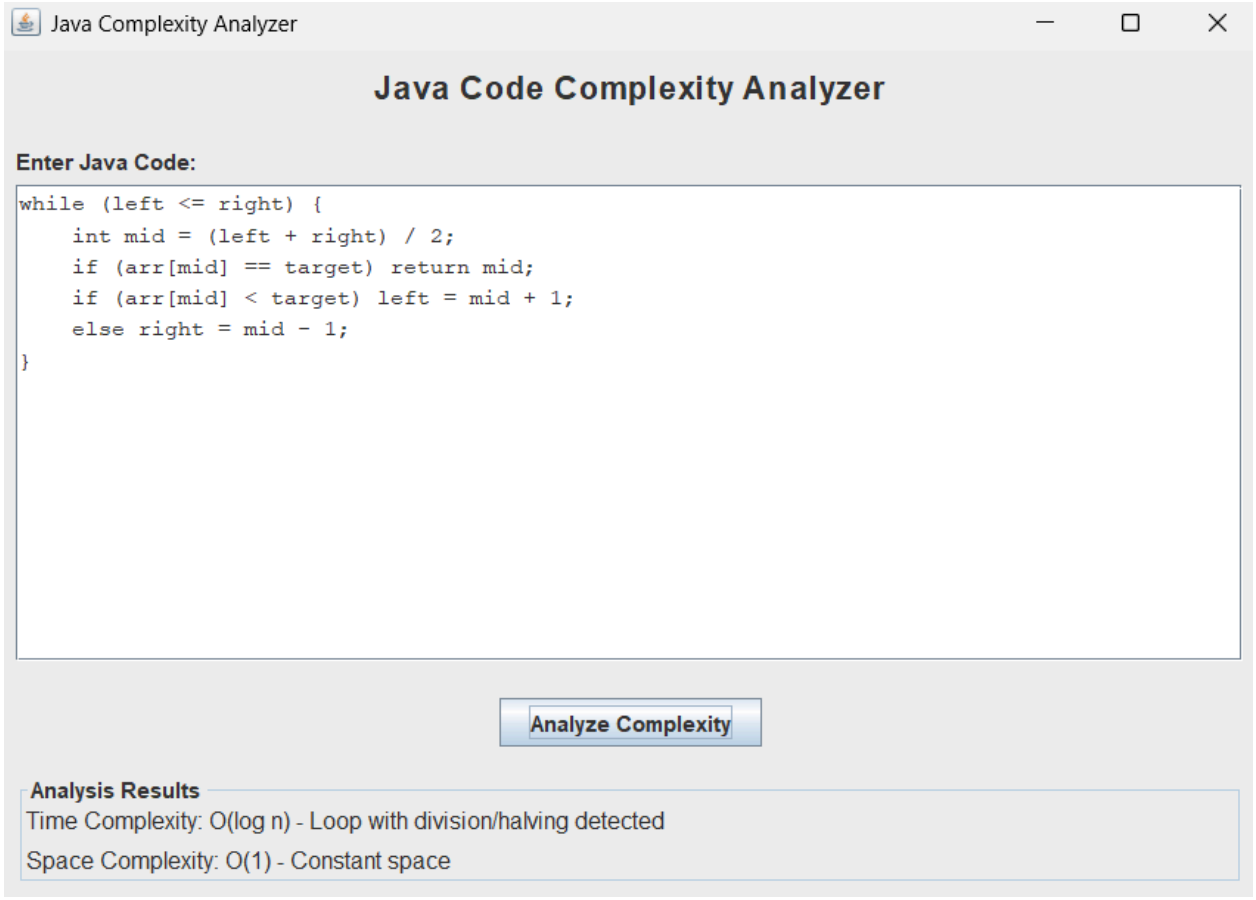
2. GUI Version

Test Case 1: Single Loop



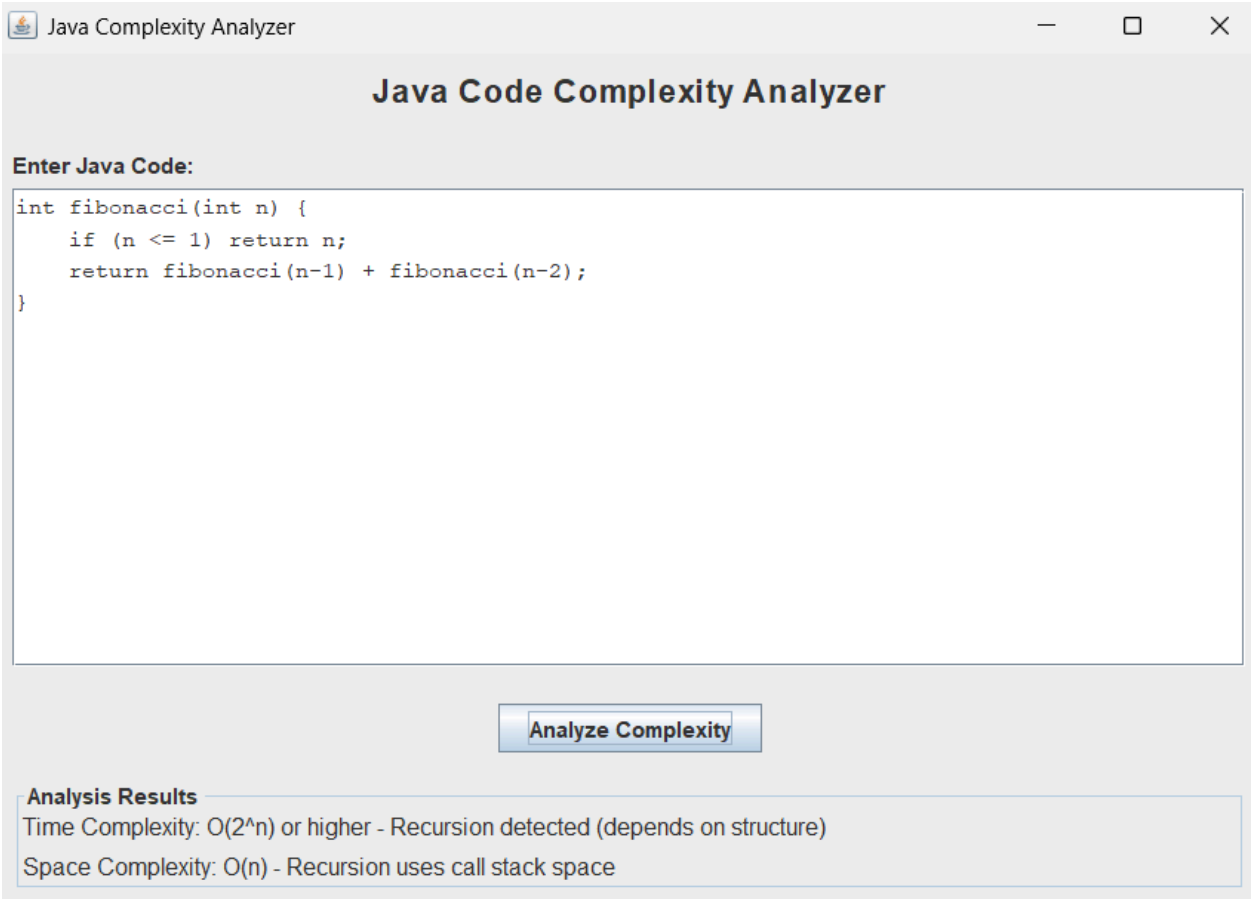
Result: Time O(n), Space O(1) ✓

Test Case 2: Binary Search



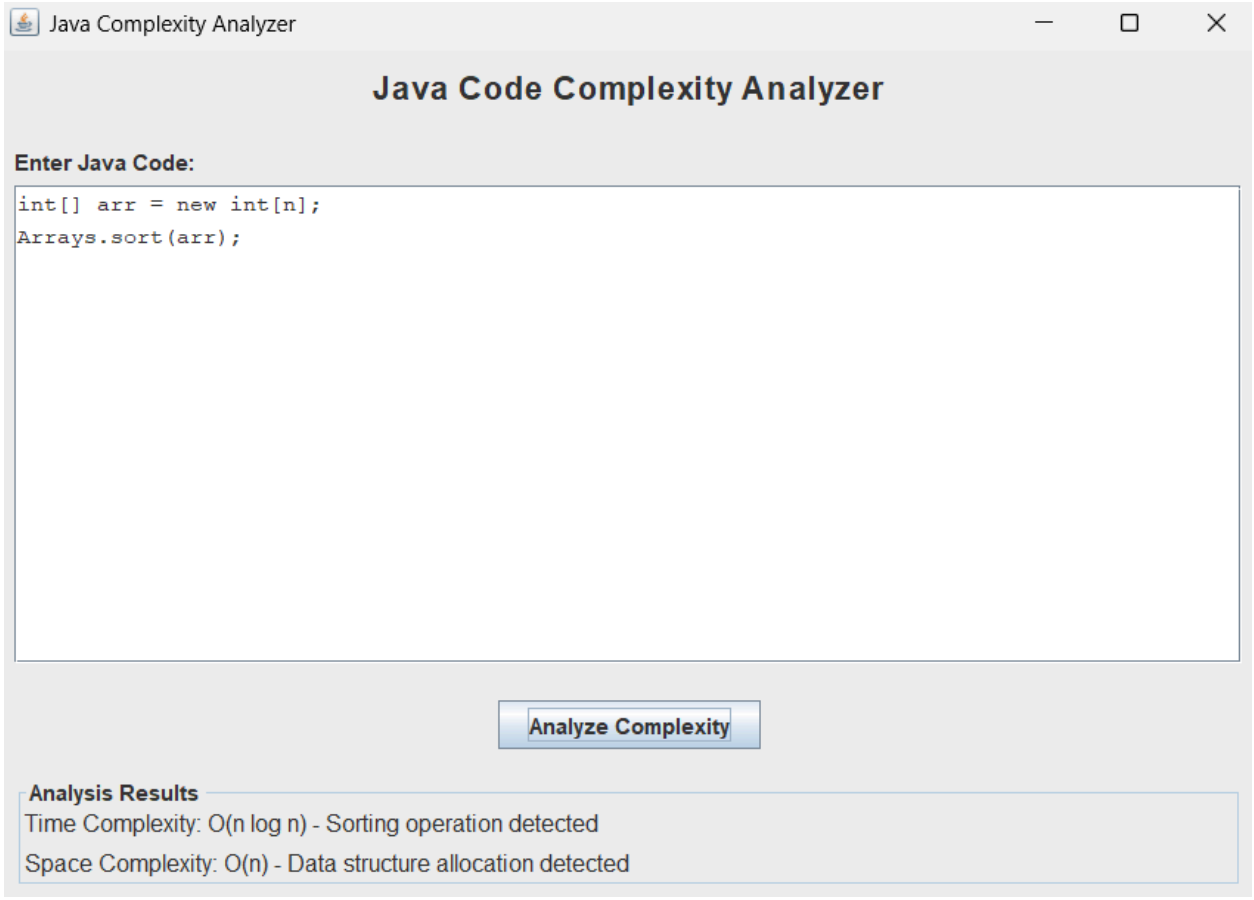
Result: Time O(log n), Space O(1) ✓

Test Case 3:



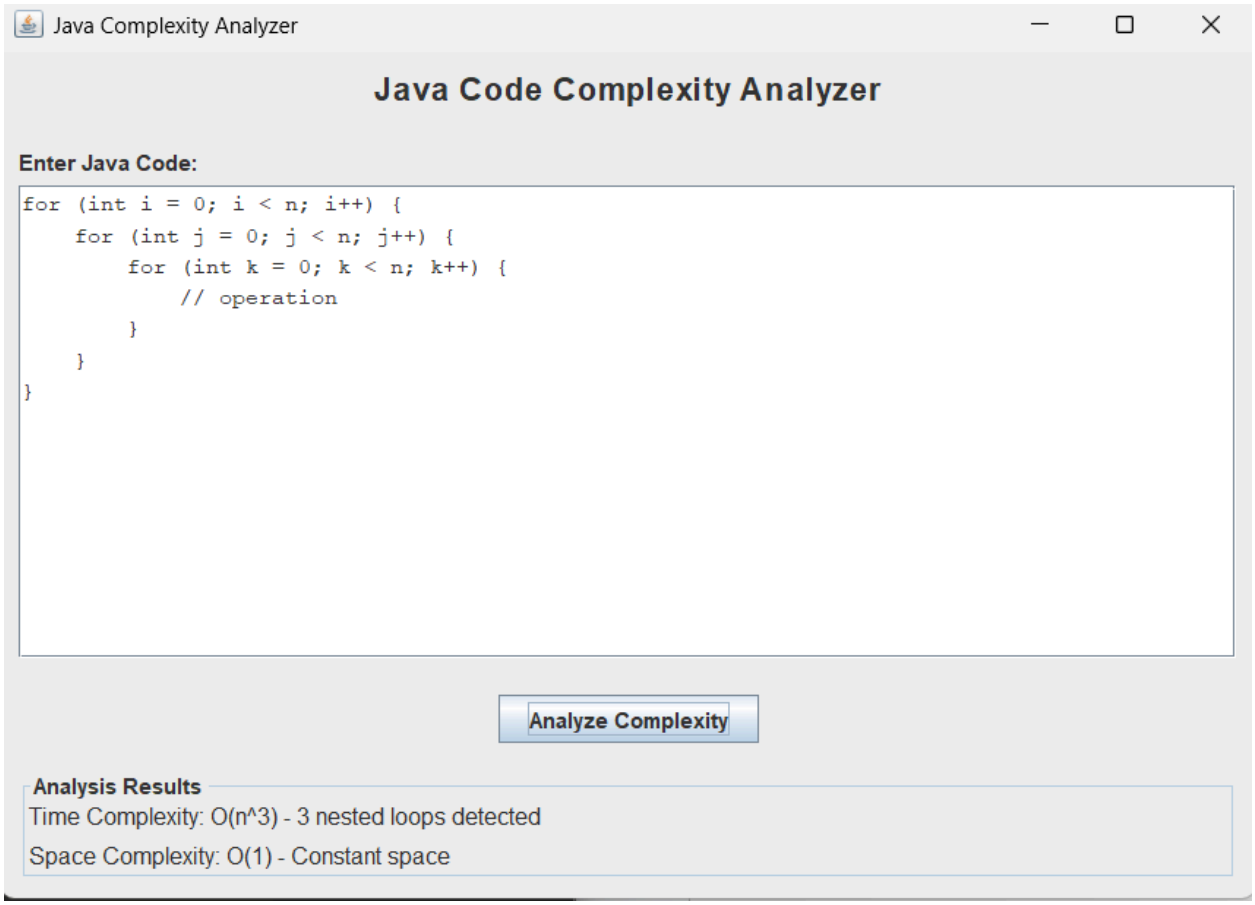
Result: Time O(2^n), Space O(n) ✓

Test Case 4: Sorting



Result: Time O(n log n), Space O(n) ✓

Test Case 5: Triple Nested Loop



Result: Time O(n^3), Space O(1) ✓

TESTING APPROACH

1 Unit Testing Strategy

- **Loop Detection Tests:** Verified correct counting of nested loops at various depths
- **Pattern Recognition Tests:** Tested regex patterns with edge cases
- **Recursion Detection Tests:** Checked both direct and indirect recursion scenarios
- **Edge Cases:** Empty input, single statement, extremely long code

2 Integration Testing

- **Console Flow:** Tested complete workflow from input to output
- **GUI Flow:** Verified button clicks, text area input, label updates
- **Cross-Module:** Ensured analysis methods work correctly with both interfaces

3 Accuracy Testing

- **Known Algorithms:** Tested with standard algorithms (bubble sort, binary search, etc.)
- **Complex Code:** Verified analysis of multi-pattern code
- **Comparison:** Cross-referenced results with manual complexity analysis

---

CHALLENGES FACED

Challenge 1: Accurate Loop Detection

Issue: Differentiating between sequential loops and nested loops
Solution: Implemented depth tracking with brace counting to maintain current nesting level

Challenge 2: GUI Layout Management

Issue: Creating a clean, responsive layout with Swing
Solution: Used combination of BorderLayout and BoxLayout for proper component arrangement

Challenge 3: Complexity Ambiguity

Issue: Some algorithms have complexity dependent on implementation details
Solution: Provided explanatory text with results (e.g., "depends on structure") for ambiguous cases

---

LEARNINGS & KEY TAKEAWAYS

1. Learned limitations and capabilities of string-based code parsing
2. Learned layout managers and event-driven programming using Swing
3. Importance of separating concerns for code reusability
4. Reinforced understanding of Big-O notation and complexity classes
5. Created two interfaces to serve different user preferences
6. Value of clear code comments and method documentation
7. Built console version first, then extended to GUI

REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

2. Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.

3. Oracle. (2024). *Java SE Documentation*. Retrieved from https://docs.oracle.com/javase/

4. Oracle. (2024). *Java Swing Tutorial*. Retrieved from https://docs.oracle.com/javase/tutorial/uiswing/

5. Regular Expressions Documentation. Retrieved from https://docs.oracle.com/javase/tutorial/essential/regex/

6. Java Pattern Class API. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

7. Big-O Cheat Sheet. Retrieved from https://www.bigocheatsheet.com/

8. GeeksforGeeks. *Analysis of Algorithms*. Retrieved from https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/

9. VIT Project Guidelines - VITyarthi Build Your Own Project

APPENDIX A: INSTALLATION & USAGE GUIDE

Prerequisites

- Java Development Kit (JDK) 8 or higher
- Text editor or IDE (optional)
- Terminal/Command Prompt

Installation Steps

1. **Download the source files:**
    - ComplexityAnalyzer.java
    - ComplexityAnalyzerGUI.java
2. **Compile the programs:**

bash

```
Shell

javac ComplexityAnalyzer.java

javac ComplexityAnalyzerGUI.java
```

3. **Run console version:**

bash

```
Shell

java ComplexityAnalyzer
```

4. **Run GUI version:**

bash

```shell
java ComplexityAnalyzerGUI
```

Usage Instructions

Console Version:

1. Launch the program
2. Enter or paste your Java code
3. Type 'END' on a new line when finished
4. View the analysis results

GUI Version:

1. Launch the program
2. Enter or paste your code in the text area
3. Click "Analyze Complexity" button
4. View results in the results panel

APPENDIX B: CODE SAMPLES FOR TESTING

Sample 1: Bubble Sort

java

```java
void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
```

```
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Expected: Time O(n²), Space O(1)

Sample 2: Merge Sort

java

```java
void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Expected: Time O(n log n), Space O(n)

Sample 3: Linear Search

java

```java
int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Expected: Time O(n), Space O(1)

END OF REPORT