GitHub Copilot in a Data Structures and Algorithms Course

## Downloads

* [Download as Word Document (DOCX)](/downloads/teaching/data-structures-github-copilot.docx)

# Case Study: Integrating GitHub Copilot in a Data Structures and Algorithms Course

## Course Context

**Course:** Data Structures and Algorithms (CSE 2321)
**Level:** Sophomore
**Enrollment:** 75 students
**Prior Format:** Lecture with weekly programming assignments
**Tools:** Java programming environment, version control systems
**Faculty:** Dr. L, Assistant Professor of Computer Science and Engineering

## Implementation Challenge

Dr. L identified several challenges in teaching data structures and algorithms that AI-assisted coding could potentially address:

1. **Implementation Focus vs. Conceptual Understanding:** Students often get bogged down in syntax and implementation details, missing the bigger conceptual picture
2. **Different Programming Experience Levels:** Wide variance in coding proficiency among students
3. **Debugging Frustration:** Significant time spent on debugging rather than algorithm design
4. **Limited Examples:** Difficulty providing enough varied examples of algorithm implementations
5. **Real-World Context:** Students struggling to connect theoretical algorithms with practical applications

## Implementation Goals

The integration of GitHub Copilot aimed to:

1. Shift focus from syntax details to algorithmic thinking and design
2. Provide scaffolding for students with less programming experience
3. Accelerate the implementation process to enable more iterations
4. Expose students to various implementation approaches
5. Prepare students for real-world coding environments where AI tools are increasingly common

## Implementation Process

### Phase 1: Faculty Preparation (Month before semester)

1. **Tool Exploration:**
	* Extensively tested Copilot’s capabilities with data structures implementations
	* Identified algorithm types where Copilot excelled vs. struggled
	* Developed effective prompting strategies for different data structures
	* Created test cases to validate Copilot-generated implementations
	* Documented common misconceptions in Copilot’s code
2. **Assignment Redesign:**
	* Restructured programming assignments to focus on design decisions
	* Created AI-appropriate and AI-restricted components for assignments
	* Developed exercises comparing different implementation approaches
	* Designed new code analysis and critique requirements
	* Created reflection prompts for AI collaboration
3. **Assessment Adaptation:**
	* Shifted evaluation criteria from implementation to analysis
	* Developed rubrics for code quality and design rationale
	* Created requirements for documenting AI contributions
	* Designed new testing and edge case analysis components
	* Established guidelines for appropriate AI use in assessments

### Phase 2: Student Onboarding (First two weeks)

1. **Introduction to AI-Assisted Coding:**
	* Conducted hands-on workshop introducing GitHub Copilot
	* Demonstrated effective comment-based prompting techniques
	* Discussed ethical considerations of AI-generated code
	* Practiced code review and validation of AI suggestions
	* Established expectations for citation and attribution
2. **Guided Exploration:**
	* Assigned simple data structure implementations with Copilot
	* Required documentation of Copilot’s strengths and limitations
	* Facilitated group discussions comparing different Copilot-generated approaches
	* Practiced critically evaluating and refining AI-generated code
	* Provided feedback on effective prompting and directing Copilot
3. **Skill Development Framework:**
	* Established progressive proficiency guidelines for AI-assisted coding
	* Created scaffolded exercises with decreasing guidance
	* Developed peer review protocols for AI-generated code
	* Practiced explaining algorithmic choices in AI-generated implementations
	* Introduced testing frameworks to validate correctness

### Phase 3: Progressive Integration (Throughout semester)

1. **Algorithmic Thinking Focus:**
	* Shifted assignment focus to design decisions and algorithm selection
	* Required students to plan before implementing with Copilot
	* Emphasized theoretical analysis alongside implementation
	* Encouraged experimentation with different algorithmic approaches
	* Required comparative analysis of time/space complexity
2. **Comparative Implementation Approach:**
	* Students implemented key algorithms both manually and with Copilot
	* Conducted analysis of differences in approaches and efficiency
	* Evaluated edge cases and potential failures in both implementations
	* Documented where Copilot made conceptual algorithmic errors
	* Practiced refining Copilot’s initial implementations
3. **Professional Practice Development:**
	* Introduced industry-standard documentation practices
	* Implemented code review sessions for AI-assisted implementations
	* Required comprehensive test coverage for all implementations
	* Practiced responsible AI use and appropriate citation
	* Developed skills in directing AI tools effectively

## Implementation Examples

### Example 1: Guided Binary Search Tree Implementation

**Traditional Challenge:** Students typically spent most of their time on implementation syntax and debugging rather than understanding tree properties and operations.

**Copilot-Enhanced Approach:** Students first designed BST operations conceptually, then used directed prompts to implement with Copilot, followed by analysis and optimization.

**Assignment Components:** > **Part 1: Pre-implementation Design (AI-restricted)** > - Draw the conceptual structure of a binary search tree with 7 elements > - List the invariants that must be maintained for valid BST operations > - Outline pseudocode for insert, delete, and search operations > - Identify at least three edge cases for each operation > > **Part 2: Implementation with Copilot** > - Use specific, detailed comments to guide Copilot in implementing your design > - Document where Copilot’s suggestions aligned with or differed from your pseudocode > - Implement test cases for your identified edge cases > - Fix any logical errors in Copilot’s implementation > > **Part 3: Analysis and Reflection** > - Compare the time complexity of your implementation with the theoretical optimum > - Identify any optimizations you applied to Copilot’s initial code > - Discuss one alternative implementation approach Copilot suggested > - Reflect on how using Copilot affected your understanding of BST operations

**Student Prompt Example:**

/\*\*
 \* Implementation of a Binary Search Tree with the following operations:
 \* - insert: adds a new value to the tree while maintaining BST properties
 \* - delete: removes a value while preserving tree structure
 \* - search: finds a value and returns the node containing it
 \* - inOrderTraversal: visits all nodes in ascending order
 \*
 \* Edge cases to handle:
 \* - Insertion when tree is empty
 \* - Deletion of root node
 \* - Deletion of node with two children
 \* - Searching for non-existent value
 \*/

public class BinarySearchTree<T extends Comparable<T>> {
 // Please implement the Node class with generic type T

 // Add instance variables for the BST

 // Implement insert method that maintains BST properties

 // Implement delete method that handles all edge cases

 // Implement search method that returns the node or null

 // Implement in-order traversal that prints values in ascending order
}

### Example 2: Algorithm Comparison Analysis

**Traditional Challenge:** Students often implemented only one approach to solving algorithmic problems, missing insights from alternative methods.

**Copilot-Enhanced Approach:** Students used Copilot to quickly implement multiple algorithmic approaches to the same problem, then analyzed tradeoffs.

**Sorting Algorithm Comparison Assignment:** > In this assignment, you will implement and compare four different sorting algorithms: Merge Sort, Quick Sort, Heap Sort, and Insertion Sort. > > **Part 1: Multiple Implementations** > - Use Copilot to help implement each sorting algorithm > - For each algorithm, write detailed comments describing the approach before generating code > - Verify correctness with provided test cases > - Instrument each implementation to count operations (comparisons and swaps) > > **Part 2: Empirical Analysis** > - Run each algorithm on the provided datasets (sorted, reverse sorted, random) > - Record execution time and operation counts for each algorithm and dataset > - Create visualization comparing performance across different input sizes > - Identify crossover points where algorithm preference changes > > **Part 3: Critical Analysis** > - Compare the actual performance with theoretical complexity > - Analyze how Copilot implemented each algorithm and any optimizations it included > - Identify one case where Copilot’s implementation could be improved and implement your optimization > - Discuss how your understanding of sorting algorithms was enhanced by implementing and comparing multiple approaches

**Prompt Strategy Example:**

/\*\*
 \* Implementation of the Merge Sort algorithm with the following characteristics:
 \* - Divides array into halves recursively until single elements
 \* - Merges subarrays in sorted order
 \* - Stable sort that preserves relative order of equal elements
 \* - O(n log n) time complexity regardless of input arrangement
 \* - O(n) auxiliary space complexity
 \*
 \* Please implement an optimized version that:
 \* - Uses insertion sort for small subarrays (size < 10)
 \* - Avoids unnecessary array copies
 \* - Includes operation counting for comparisons and array accesses
 \*/
public class MergeSort {
 // Implement merge sort with optimizations
}

### Example 3: Data Structure Selection and Application

**Traditional Challenge:** Students often struggled to select appropriate data structures for real-world problems.

**Copilot-Enhanced Approach:** Students designed solutions to complex problems by selecting and integrating multiple data structures, using Copilot to accelerate implementation.

**Real-World Application Assignment:** > **Text Analysis Tool Project** > > Design and implement a text analysis tool that processes large text files and provides the following functionality: > - Word frequency counting and ranking > - Finding k-most common word sequences (n-grams) > - Efficient spell-checking against a dictionary > - Identifying semantically similar paragraphs > > **Part 1: Design Phase (AI-restricted)** > - Select appropriate data structures for each functionality > - Justify your data structure choices with complexity analysis > - Design the interfaces between components > - Create a system architecture diagram > > **Part 2: Implementation with Copilot** > - Implement each component using your selected data structures > - Use clear, directive comments to guide Copilot > - Ensure appropriate error handling and edge cases > - Optimize performance bottlenecks > > **Part 3: Analysis and Extension** > - Test with provided large text corpus (1M+ words) > - Profile performance and identify bottlenecks > - Compare actual performance with theoretical predictions > - Propose and implement one algorithmic improvement > - Discuss how your data structure selections affected overall performance

**Implementation Guidance Strategy:** Students created detailed specification comments for each component before using Copilot, focusing on data structure selection rationales:

/\*\*
 \* WordFrequencyAnalyzer: Counts and ranks word frequencies in text.
 \*
 \* Data structure choice: HashMap<String, Integer> for counting occurrences with O(1) lookups,
 \* then transferred to a PriorityQueue for efficient retrieval of top-K words in O(n log k) time.
 \*
 \* Requirements:
 \* - Case-insensitive word counting
 \* - Filtering punctuation and special characters
 \* - Efficient retrieval of top-K most frequent words
 \* - Handling very large texts without memory issues
 \* - Support for updating counts with additional text
 \*/
public class WordFrequencyAnalyzer {
 // Implement the data structures and methods based on the requirements
}

## Assessment Strategy

### Evolved Assessment Approach

#### Before Copilot Integration: Traditional Assessment

* Algorithm implementation correctness (40%)
* Code quality and style (25%)
* Testing and edge cases (20%)
* Documentation (15%)

#### After Copilot Integration: Design-Focused Assessment

* Algorithm design and selection justification (30%)
* Implementation analysis and optimization (25%)
* Testing strategy and edge case handling (20%)
* Comparison of approaches and trade-offs (15%)
* AI collaboration and direction skills (10%)

### New Assessment Components

1. **Design Documentation Requirements:**
	* Pre-implementation design documents outlining approach
	* Justification of algorithm and data structure selection
	* Identification of edge cases and handling strategy
	* Complexity analysis before implementation
2. **AI Collaboration Skills:**
	* Quality and specificity of prompts/comments
	* Critical evaluation of AI-generated code
	* Appropriate refinement of initial implementations
	* Documentation of AI contribution and human modifications
3. **Comparative Analysis:**
	* Implementation of multiple approaches to the same problem
	* Empirical testing across different scenarios
	* Identification of performance trade-offs
	* Selection criteria for real-world applications

## Potential Outcomes and Considerations

### Expected Benefits

* Increased focus on algorithmic thinking rather than syntax debugging
* More time for exploring multiple implementations and approaches
* Greater exposure to industry-standard coding practices
* Reduced frustration with implementation details for novice programmers
* Better preparation for professional environments using AI coding tools
* Development of critical code review skills
* More ambitious projects completed within course timeframe

### Potential Challenges

* Risk of over-reliance on AI for implementation without understanding
* Copilot occasionally suggesting incorrect algorithmic approaches
* Variable student proficiency in effectively directing the AI
* Difficulty assessing individual understanding in group projects
* Potential for shallow engagement with fundamental concepts
* Need for clear boundaries on appropriate use in assessments

## Faculty Implementation Considerations

### Key Implementation Strategies

1. **Design-first approach** requiring planning before implementation
2. **Comparative implementations** to understand algorithmic trade-offs
3. **Critical evaluation** of AI-generated code as a core skill
4. **Progressive responsibility** framework with decreasing scaffolding
5. **Real-world application focus** leveraging AI for implementation speed

### Important Considerations

1. **Fundamentals must come first** before AI acceleration
2. **Code review skills** become increasingly important
3. **Clear attribution guidelines** prevent academic integrity issues
4. **Strategic assignment design** with AI-appropriate components
5. **Balance between manual and AI-assisted implementation** maintains core skills

### Future Refinement Directions

If implementing such an approach, consider: 1. Creating language-specific prompt libraries for different data structures 2. Developing better methods for assessing algorithm design understanding 3. Incorporating more comparative analysis between AI and human approaches 4. Adding industry partners to provide real-world coding contexts 5. Exploring pair programming approaches with AI as a third participant

## Resources Developed

1. **Prompting Guide:** Structure for effective GitHub Copilot direction
2. **Design Documentation Templates:** Frameworks for pre-implementation planning
3. **Code Review Checklists:** Focused on AI-generated code evaluation
4. **Algorithm Selection Decision Trees:** Guides for choosing appropriate algorithms
5. **Implementation Comparison Rubrics:** Evaluation criteria for approach trade-offs

## Implementation Advice

### For Faculty Considering Similar Integration:

1. **Start with design documentation** before implementation
2. **Create clear boundaries** for appropriate AI use in different contexts
3. **Emphasize critical evaluation** of AI-generated code
4. **Focus assignments on comparative analysis** rather than implementation alone
5. **Develop strong test cases** to validate all implementations

### Technical Considerations:

1. **Copilot performance varies** across different algorithm types
2. **Detailed comments produce better results** than vague directions
3. **Test-driven approaches work well** with AI implementation
4. **Version control integration** is essential for tracking contribution
5. **Regular updates to guidance** may be needed as Copilot evolves

*This case study was developed as part of the “Strategies for Integrating Generative AI in Engineering Education” workshop materials in collaboration with Claude-3.7 Sonnet.*