Course Introduction

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti

Spring 2026

Spelman College

Welcome to SCIS 313!

What is this course about?

We'll study how to analyze and design efficient algorithms:

- Algorithm complexity analysis (Big-O notation)
- Advanced data structures (trees, graphs, heaps)
- Algorithm design techniques (recursion, greedy, dynamic programming)
- Theoretical limits of computation (P vs NP)

Why does this matter?

- Write faster, more efficient code
- Ace technical interviews at top companies
- Understand what's computationally possible (and impossible)
- Think like a computer scientist

Meme


Kylie Jenner
@ikyliejenner
Can you guys please recommend books that made you cry?
Saransh Garg @saranshgarg
Replying to @ikyliejenner
Data Structures and Algorithms in Java (2nd Edition) 2nd E
by Robert Lafore (Author)

Course Topics

Weeks 1-7: Foundations

- Empirical analysis
- Big-O notation
- Analyzing loops & code
- Mathematical foundations
- Recursion & recurrences
- Searching & sorting
- Data structures review

Weeks 8-13: Advanced Topics

- Amortized analysis
- AVL trees
- Graph algorithms
- Greedy algorithms
- Dynamic programming
- Complexity theory

By the end: Analyze any algorithm and understand fundamental CS theory

Grading Breakdown

Component	Weight
Exams (3-4 total)	**65%**
Exam 1 (Week 6)	15%
Exam 2 (Week 9)	15%
Exam 3 (Week 12)	15%
Final Exam (Week 15)	20%
Projects (3-4 presentations)	**25%**
Homework (weekly)	**10%**

Homework: Completion-based (submit screenshots + work)

Projects: Group presentations on advanced topics (2-3 students)

Weekly Schedule

Tuesday:

- New content lecture
- Work through examples
- Theory and applications

Thursday:

- Finish content (first 30 minutes)
- **Homework review** (last 30-45 minutes)
  - Students present solutions (popcorn style)
  - Mandatory participation - everyone presents $\sim$once per month
  - Build your presentation skills!

Homework due: Thursday before class

How to Succeed in This Class

How To Do Well:

- **Attend class!**
- **Ask questions**
- Start homework early
- Come to office hours
- Practice coding
- Form study groups

How to Fail:

- **Skip class!**
- **Never communicate**
- Procrastinate
- Try to memorize
- Work alone when stuck

Reality: This material is **abstract**. You won't get everything immediately. That's normal! Keep working, ask for help, and it will click.

Homework: Programming + Written Problems

Weekly homework (∼8-10 problems):

- Mix of written analysis and coding
- **Data Structures and Algorithms (LeetCode) problems** (3-5 per week starting Week 5)
- Completion-based grading

Why LeetCode?

- Industry standard for technical interviews
- Auto-graded (instant feedback!)
- Build your problem-solving portfolio
- Practice implementation, not just theory

Submission:

- LeetCode: Screenshot showing green "Accepted"
- Written: Photos or typed solutions

Projects

3-4 group projects throughout semester

Each project:

- 15-20 minute presentation
- 3-5 page written report
- Research advanced topics not covered in depth

Example topics:

- Red-Black trees vs AVL trees
- Dijkstra's shortest path
- String matching (KMP)
- Stern-Brocot tree
- Approximation algorithms

What Makes This Course Different?

First course: Implement data structures

This course: *Analyze* algorithms and understand *why* they work

We focus on:

- **Efficiency:** Is this $O(n)$ or $O(n^2)$? Why?
- **Correctness:** Does this always work?
- **Limits:** What *can't* be solved efficiently?
- **DSA Questions:** Companies ask these questions!

This is **theoretical computer science** - more math and proofs than typical programming.

But it's also **practical** - these concepts appear in real interviews.

The Red Thread of This Course

Core themes we'll explore:

1. **Programming requires algorithmic design**
   - How do we solve problems systematically?
2. **Computers are founded on universal computation**
   - What can and cannot be computed?
3. **We will study fundamental data structures and algorithms**
   - The building blocks of all software
4. **Some problems are undecidable**
   - Even simple-looking problems may have no algorithmic solution
5. **Some problems are computationally hard (NP-complete)**
   - Traveling salesman, graph coloring - probably cannot be solved efficiently
   - Critical question: Does P = NP?

What This Course Is (Not) About

This is NOT:

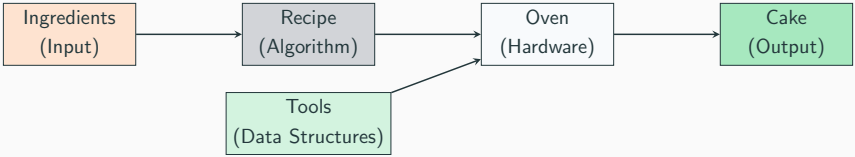
- A programming course
  - No computer labs
- A pure theory course
  - Proofs sketched, not rigorous

This IS:

- Crucial for becoming a good programmer
  - Algorithms and data structures
- Foundations of computer science
  - Computability and complexity
- Exploration of computation models
  - Classical (and possibly quantum) computing

Balance: Practical programming skills + theoretical understanding

Algorithms in Real Life: Baking a Cake



Key insight:

- **Algorithm** = step-by-step instructions (recipe)
- **Data structures** = how we organize ingredients (tools)
- **Hardware** = physical computer (oven)
- **Software** = recipe implemented in programming language

Classic Example: Euclid's Algorithm

Greatest Common Divisor (circa 300 BC)

Problem: Find the greatest common divisor (GCD) of two numbers $n \geq m$

Euclid's Algorithm:

- If $m = 0$, return n
- Otherwise, compute $\gcd(m, n \bmod m)$

Example: $\gcd(129, 57)$

$$\begin{aligned}\gcd(129, 57) &= \gcd(57, 129 \bmod 57) = \gcd(57, 15) \\ &= \gcd(15, 57 \bmod 15) = \gcd(15, 12) \\ &= \gcd(12, 15 \bmod 12) = \gcd(12, 3) \\ &= \gcd(3, 12 \bmod 3) = \gcd(3, 0) \\ &= 3\end{aligned}$$

Note: This algorithm uses recursion - a function calling itself!

Euclid's Algorithm in Pseudocode

Algorithm 1 Euclid(m, n)

```
1: if m = 0 then
2:     return n
3: end if
4: if n = 0 then
5:     return m
6: end if
7: if m ≤ n then
8:     return Euclid(m, n mod m)
9: else
10:    return Euclid(m mod n, n)
11: end if
```

Pseudocode is algorithm description that:

- Resembles programming language syntax
- Is more abstract and language-independent
- Focuses on logic, not implementation details

What is an Algorithm?

Definition: A list of precise instructions that captures the essence of solving a problem

Three critical aspects:

1. **Termination:** Does the algorithm always finish?
   - Must produce output in finite time
2. **Correctness:** Does it solve the problem?
   - Must meet its specifications
   - Produces correct output for all valid inputs
3. **Efficiency:** How fast is it?
   - Time complexity: How many steps?
   - Space complexity: How much memory?

This course focuses heavily on analyzing efficiency!

Our Model of Computation

We will use these abstractions throughout the course:

Computer: Random Access Machine (RAM)

- Theoretical model of a computer
- Unlimited memory cells (registers)
- Basic operations take constant time

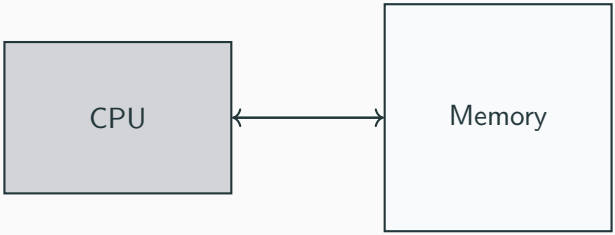
Data Structure: Abstract Data Type (ADT)

- Specification of operations, not implementation
- Example: Stack with push() and pop()

Algorithm: Pseudocode

- Language-independent description
- Focus on logic and structure

Random Access Machine (RAM)



Assumptions:

- Unlimited number of memory cells (registers)
- Primitive operations (read, write, arithmetic) take constant time
- Simplifies analysis - focus on algorithm, not hardware details

What is a Data Structure?

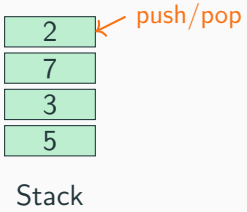
Definition: A particular way of organizing data in a computer for efficient use

Abstract Data Type (ADT): Specification by operations, not implementation

Example: Stack ADT

Operations:

- `push(d)`: Add item d to top
- `pop()`: Remove and return top item
- Exception: pop() on empty stack



Different data structures suit different applications!

Pseudocode: Our Language for Algorithms

Pseudocode is more abstract than real code, includes:

Basic elements:

- Input and output
- Variable declarations
- Parameters
- Abstract data types
- Memory operations (read/write)

Control flow:

- `if ... then ... else ...`
- `while ... do ...`
- `for ... to ... do ...`
- Recursive calls

Key features:

- Uses indentation instead of explicit block delimiters
- Language-independent (not Python, not Java, not C++)
- Focuses on algorithm logic, not syntax details

Topics We'll Cover This Semester

Weeks 1-5: Foundations

- Running time analysis
- Asymptotic notation (Big-O)
- Recurrence relations
- Mathematical induction

Weeks 6-7: Sorting

- Comparison sorts
- Quicksort, Mergesort, Heapsort
- Lower bounds

Weeks 8-10: Data Structures

- Stacks, queues, lists
- Trees (BST, AVL, heaps)
- Hash tables
- Graphs

Weeks 11-13: Advanced Topics

- Greedy algorithms
- Dynamic programming
- NP-completeness

Questions?

Next: Lesson 1 - Empirical Analysis

Learn how to measure algorithm performance.