Lecture 1:

- Course Overview

- Algorithm Analysis (Intro/Review)

Basic Logistics

1. Course webpage: www.cs.gonzaga.edu/bowers/courses/cpsc450
2. Piazza: for Q&A, announcements (see invite)
3. GitHub: for homework
4. Blackboard: for tracking points
5. Office Hours: Tu/Th 12:30–1:30, Wed 1–3

See webpage for syllabus, weekly schedule, homework assignments, etc.
Course Overview

Course Goals:

• Study CS algorithm “greatest hits” (the classics)
• Graph data structures and algorithms
• More algorithmic problem solving approaches (e.g., greedy, dynamic prog.)
• Algorithm analysis (correctness, efficiency, complexity, including P vs NP)

Course Overview (cont)

Course work: 700 points

• C++ assignments (7) 280 points
• Individual project 60 points
• Problem sets (8) 80 points
• Quizzes (6) 60 points
• Exams (3) 160 points
• Attendance / participation (30) 60 points

Must score at least 60% on homework (assignments, final project, problem sets) and 60% on tests (quizzes and exams)

Please read the syllabus for additional information
Course Overview (cont)

**Expectations:**
- Engage and participate in class (including doing your own work!)
- Start assignments early, give yourself enough time to succeed
- Assume you have everything you need (ask when in doubt)

(Some) collaboration is encouraged ... e.g., discussing assignments
- but avoid plagiarism & other related issues — e.g., no code sharing

**Hints:**
- carefully read and follow instructions
- do not use youtube, google, stackoverflow, etc., to "learn" material
- study for quizzes, exams, etc
- come to office hours and frequently check piazza

---

Schedule Overview

1. Algorithm analysis  
   $\approx$ 2 weeks
2. Graph data structures  
   $\approx$ 1 week
3. Basic graph searching (BFS, DFS) and applications  
   $\approx$ 4 week
4. “Harder” graph problems and weighted graphs  
   $\approx$ 3 weeks
5. Greedy algorithms  
   $\approx$ 1 week
6. Dynamic programming  
   $\approx$ 2 weeks
Algorithm Analysis Basics

Algorithms in class as (pseudocode) **functions** with **input** and **output**
- focus on basic ideas without language details
- usually no error handling, etc. (which you’ll need to include in HW)

For algorithm analysis, interested in **input size**
- performance cost grows with "larger" input sizes
- want to show algorithms work for all input sizes
- implies input data structures (collections) such as lists, trees, graphs, etc.

Various forms of output, for example:
- yes/no (true/false)
- computed value (e.g., tree height)
- permutation of input data collection (e.g., sorted list)

---

Algorithm Analysis Basics (cont)

**Example:** Checking if array contains a given value

**Algorithm:** LinearSearch

**Input:** An integer array $A[]$ of size $n$ and an integer $v$.
**Result:** True or false depending on if $v$ is in $A$.

```plaintext
begin
  for $i \leftarrow 0$ to $n - 1$ do
    if $A[i] = v$ then
      return true
  return false
```

- Input is the array $A$ and value $v$
- Input size is the array size $n$
- Output is yes (list contains value) or no (doesn’t contain value)
Algorithm Correctness

Loop invariants help to understand why an algorithm is correct

*Example:* At the start of each iteration, \( v \) is not in the subarray \( A[0 .. i - 1] \)

Must show three things about a loop invariant:

1. **Initialization:** True prior to first loop iteration (similar to a base case)
2. **Maintenance:** Holds from iteration to iteration (similar to inductive step)
3. **Termination:** After termination, invariant gives a useful property to help show correctness

Check-In: Show the loop invariant “works” for our linear search algorithm

Invariant: At the start of each iteration, \( v \) is not in the subarray \( A[0 .. i - 1] \)

The three things we need to show about the invariant:

1. **Initialization:** Holds since \( A[0 .. -1] \) is an empty subarray
2. **Maintenance:** For any \( 0 \leq i < n \), we assume the invariant holds prior to the iteration (\( v \) not in \( A[0 .. i - 1] \)). If \( A[i] \neq v \), the subarray \( A[0 .. i] \) also does not contain \( v \), implying the invariant holds for \( i + 1 \) as well.
3. **Termination:** On the last iteration \( i = n - 1 \). If \( A[n - 1] \neq v \), then on termination we have that \( v \) is not in \( A[0 .. n - 1] \) (the entire array).

(*) Note that the invariant helps prove correctness. E.g., we’d still need to show linear search finds \( v \).