CSCI-UA.0480-004
Algorithmic Problem Solving

Brett Bernstein and Sean McIntyre

Class 1: Introduction

Course Administration

- Class website
  - http://cs.nyu.edu/courses/spring14/CSCI-UA.0480-004/
- Contact
  - Sean: sm4266@nyu.edu
  - Brett: brett.bernstein@nyu.edu
- Office hours
  - Not set yet

Course Administration

- Grading
  - Homework (70%)
  - Midterm (10%)
  - Final exam (20%)

Course Administration

- Homework (70%)
  - Released Friday afternoon at 5 PM at practice
  - Due the following Friday at 2 AM
  - Work with your classmates, but submit your own code
    - We will be checking

Course Administration

- Midterm (10%)
  - 1 week to solve problems from the topics learned from class up to the midterm

Course Administration

- Final (20%)
  - 2 hour competition exam

Course Administration

- Friday practice / optional homework help session
  - Homework assignments will be worked on together every Friday 5-7 PM
  - WWH 102

Course Administration

- Textbook
  - Steven and Felix Halim's “Competitive Programming”, **Third Edition**
  - https://sites.google.com/site/stevenhalim/

Course Administration

- Syllabus
  - Data Structures
  - Complete Search, Greedy, D&C/Binary Search
  - DP
  - Graph Algorithms
  - Math
  - String Processing
  - Geometry
  - Advanced Topics

Course Introduction

How this course fits in

- Introduction to programming class
  - How to write programs
  - Basic software engineering and design
- Data structure class
  - Learn about sorting algorithms
  - Basic data structures and how they differ
  - Basic graph algorithms

How this course fits in

- Theory of algorithms class
  - An overview of a number of “classical” algorithms
  - Dive into the algorithms: how they work, and why they work
  - Proof of correctness
  - Some discussion on how to apply the algorithms

How this course fits in

- CSCI-UA.0480-004 (this class)
  - Combines a lot of these classes
  - Problem-based learning
  - No formal emphasis on proof of correctness
  - Uses narratives with either contrived or practical scenarios challenge the learner
  - Brings in more challenging problems than the previous classes

What you will learn

- Reading comprehension
- Problem evaluation
- Parsing and formatting text
- Tricks to reduce code and bugs
  - bitmasks, traversing 2D spaces
- Generating test cases for your code

What you will learn

- Algorithms that will be practiced
  - Dynamic Programming (DP)
    - state-space search, games
  - Data structures
    - binary indexed tree, union-find
  - Computational geometry
    - convex hull
  - Graph algorithms
    - flow

Why compete

- Makes you a better programmer and thinker in many situations
- Intangible skill that will set you apart in the workforce
- You're all invited to join NYU progteam
- It's fun :)

Programming Contests

- ACM International Collegiate Programming Contest (ICPC)
- TopCoder
  - Weekly online individual competitions
- Google Code Jam
- Internet Problem Solving Competition
  - Annual, fun, different style of problems
- IOI, USACO

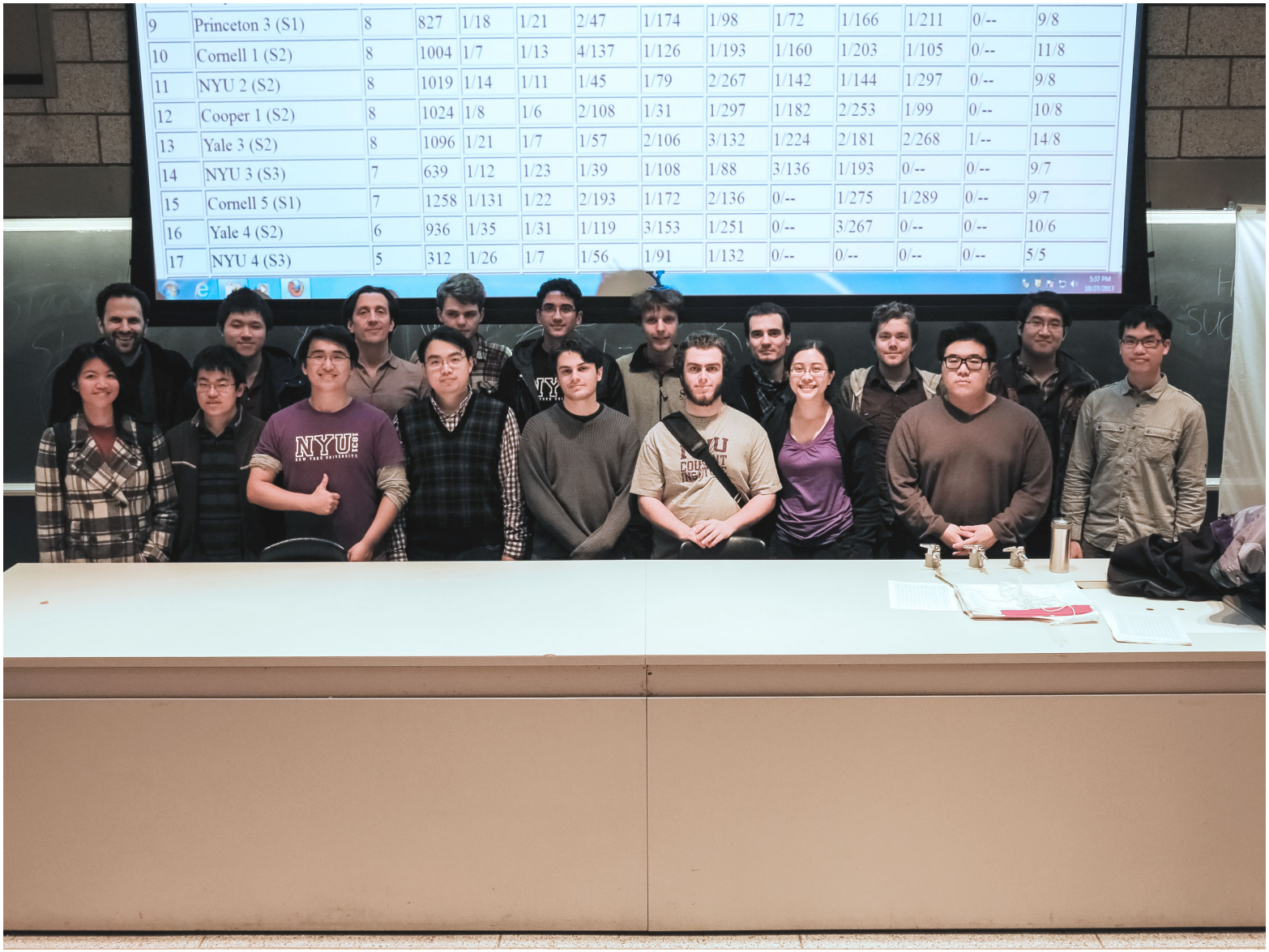
ACM ICPC

- Format:
  - 3 people
  - 1 computer
  - 5 hours
  - 10 problems

NYU competition history

- 2011 Greater New York Regional (GNYR)
  - 8th, 11th, 20th, 21st
- 2012 GNYR
  - 3rd, 10th, 11th
- 2013 GNYR
  - **1st**, 11th, 14th, 17th, 25th

NYU 2013 team



Join progteam

- Join the progteam mailing list!
  - http://www.cs.nyu.edu/mailman/listinfo/progteam
  - (link on course website)

Course Material

Competitive Programming

- Given *well-known* computer science problems, solve them *as fast as possible*
  - Find a solution that reduces down to a well-known problems, not research problems
  - Pass all the judge data correctly
  - Solution should run fast enough
  - Do not over-engineer the solution

First problem

- There are 2*N houses scattered on a 2D plane
- Find N pairs of houses such that the sum of the distances between the houses is minimized
- e.g.
  - Houses: (1, 1), (8, 6), (6, 8), (1, 3)
  - Sum of distances: 4.83

First problem

- **Competitive programmer** style (20+ practices)
  - Realizes all of this and can solve the problem in 30 minutes or less without mistakes

First problem

- **Beginner** style of solving this problem (0-3 practices)
  - Never seen this kind of problem
  - Takes awhile to comprehend problem statement
  - Starts coding without knowing a solution
  - Tries either a greedy solution (“pick the closest two”) or a complete search (backtracking)

First problem

- **Inexperienced** style (3-9 practices)
  - Recognizes the problem from a prior practice
  - Realizes that greedy and backtracking doesn't work
  - Thinks there is a DP solution
  - Gives up and moves on

First problem

- **Non-competitive programmer** style (10+ practices)
  - Realizes the solution is matching on a graph
  - Realizes the input size is small so can solve it with DP
  - Realizes that bitmasks help solve the problem
  - Makes a mistake in implementation, has to debug
  - Gets accepted answer after a couple of hours

How to be a competitive programmer

1) Type fast and correctly

- Know your IDE (in ICPC competition, Eclipse!)
- In competition, you may bring in a limited set of notes which contain code that you can type from the paper

How to be a competitive programmer

2) Quickly identify problem types

- Ad hoc
- Complete search
- Divide and conquer
- Greedy
- Dynamic programming
- Graph
- Mathematics
- String processing
- Computational geom.
- HARD

How to be a competitive programmer

- Moreover, identify whether or not you can solve the problem type
  - **Solved** before / can solve again quickly
  - **Solved** before / will take time to solve again
  - **Seen** before / will solve this if all easier ones are solved
  - Not sure

How to be a competitive programmer

- Exercise: What kind of problem is this?

- Given an *M***N* integer matrix *Q*, check if there exists a sub-matrix of *Q* of size *A***B* where mean(*Q*) = 7?
  - 1 <= M, N <= 50
  - 1 <= A <= M
  - 1 <= B <= N

How to be a competitive programmer

3) Algorithm analysis

- After discovering a solution, convince yourself that it runs in time and memory
- Look at the constraints of the problem
- Worst-case analysis *before* starting to code

How to be a competitive programmer

4) Master a programming language

- After thinking of a solution, convey the solution in code as quickly as possible
- Use libraries, shortcuts, and write simple code
- Know the C++ STL or Java API without having to look at the reference
- Like being a painter, photographer, or musician

How to be a competitive programmer

- Java:
  - Scanner, BigInteger, String static functions, Collections, different data types
  - Integer.parseInt()
  - String.substring()
  - etc
- C++
  - next_permutation()

How to be a competitive programmer

5) Test your code. There are many ways to fail:

- Presentation Error (PE)
- Wrong Answer (WA)
- Time Limit Exceeded (TLE)
- Memory Limit Exceeded (MLE)
- Runtime Error (RTE)

How to be a competitive programmer

- Submit correctly
  - Competitions only care about correct code
  - Is it worth the 20 minute penalty to submit without test cases?
  - The best teams write test cases before submitting their solutions

How to be a competitive programmer

6) Practice

- Talking about programming contests only get you so far
- UVa Online Judge
  - http://uva.onlinejudge.org
- TopCoder
  - http://topcoder.com
- Project Euler
  - http://projecteuler.net/

How to be a competitive programmer

7) Teamwork

- Knowing your teammates
  - Delegating problems to each other
- Sharing the computer time effectively
- Creating test cases for each other
- Being able to convey ideas
- Pair programming

Problem recipe

- Problem narrative
  - Can be unnecessarily long or misleading
- Input and output description
  - Usually very precise
  - You may assume that all input will be formatted like this
- Sample input and output
  - One or more inputs and expected outputs

Problem recipe

- Example problem, “Automatic Answer”

Steps to solving a problem

- Read the problem
- Decide whether or not you know how to solve it
- If you think you can solve it:
  - Parse the input
  - Write the algorithmic code
  - Check that the program works on the sample input/output
  - Submit!

Steps to solving a problem

- If you're not sure, move onto the next problem

Solving Automatic Answer

```
import java.io.*;

public class Main {
	public static void main(String[] args) throws Exception {
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

		int nCases = Integer.parseInt(in.readLine());

		for (int caseNum = 0; caseNum < nCases; caseNum++) {
			// Parse the input number
			int n = Integer.parseInt(in.readLine());

			// Calculate the answer
			n *= 567;
			n /= 9;
			n += 7492;
			n *= 235;
			n /= 47;
			n -= 498;

			// Digit in the tens column
			int tens = (n / 10) % 10;

			// Print it out!
			System.out.println(tens);
		}
	}
}
```

Solving Automatic Answer

```
import java.io.*;

public class Main {
	public static void main(String[] args) throws Exception {
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

		int nCases = Integer.parseInt(in.readLine());

		for (int caseNum = 0; caseNum < nCases; caseNum++) {
			// Parse the input number
			int n = Integer.parseInt(in.readLine());
```

Parse

```
			// Calculate the answer
			n *= 567;
			n /= 9;
			n += 7492;
			n *= 235;
			n /= 47;
			n -= 498;

			// Digit in the tens column
			int tens = (n / 10) % 10;

			// Print it out!
			System.out.println(tens);
		}
	}
}
```

Solving Automatic Answer

```
import java.io.*;

public class Main {
	public static void main(String[] args) throws Exception {
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

		int nCases = Integer.parseInt(in.readLine());

		for (int caseNum = 0; caseNum < nCases; caseNum++) {
			// Parse the input number
			int n = Integer.parseInt(in.readLine());

			// Calculate the answer
			n *= 567;
			n /= 9;
			n += 7492;
			n *= 235;
			n /= 47;
			n -= 498;

			// Digit in the tens column
			int tens = (n / 10) % 10;

			// Print it out!
			System.out.println(tens);
		}
	}
}
```

Algorithm

Parsing test cases

- Most problems will have numerous test cases
- Different problems ask for different ways of parsing test cases
  - e.g., Automatic Answer tells you how many test cases there are
  - Some problems say “parse until a termination line of all zeros”
  - Others will have you read until end of file

Automatic Answer with termination test case

```
import java.io.*;

public class Main {
	public static void main(String[] args) throws Exception {
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

		while (true) {
			// Parse the input number
			int n = Integer.parseInt(in.readLine());

			// Quit if the input is -99999
			if (n == -99999) {
				break;
			}

			// Calculate the answer
			n *= 567;
			n /= 9;
			n += 7492;
			n *= 235;
			n /= 47;
			n -= 498;

			// Digit in the tens column
			int tens = (n / 10) % 10;

			// Print it out!
			System.out.println(tens);
		}
	}

}
```

Automatic Answer reading until end of file

```
import java.io.*;

public class Main {
	public static void main(String[] args) throws Exception {
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

		String line;

		while ((line = in.readLine()) != null) {
			// Parse the input number
			int n = Integer.parseInt(line);

			// Calculate the answer
			n *= 567;
			n /= 9;
			n += 7492;
			n *= 235;
			n /= 47;
			n -= 498;

			// Digit in the tens column
			int tens = (n / 10) % 10;

			// Print it out!
			System.out.println(tens);
		}
	}

}
```

Readings

- Referenced in this class
  - Chapter 1.1, 1.2, 1.3