Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 1 - Introduction to Competitive Programming

Syllabus First, I've handed out along with the syllabus a short pre-questionnaire intended to give us some idea of where you're coming from and what you're hoping to get out of this class. Please fill it out and turn it in at the end of the class.

Let me briefly cover the material in the syllabus. This class is CSE 232, Programming Skills Workshop, and we're going to focus on practical programming skills and preparation for programming competitions and similar events. We'll particularly be paying attention to the 2014 Midwest Regional ICPC, which will be November 1st, and I encourage anyone interested to participate. I've been working closely with the local chapter of the ACM, and you should talk to them if you're interesting in joining an ICPC team.

My name is Dr. Brett Olsen, and I'm a computational biologist on the Medical School campus. I was invited to teach this course after being contacted by the ACM due to my performance in the Google Code Jam, where I've been consistently performing in the top 5-10%, so I do have some practical experience in these types of contests that I hope can help you improve your performance as well. Assisting me will be TA Joey Woodson, who is also affiliated with the ACM, and would be a good person to contact about the ICPC this fall.

Our weekly hour and a half meetings will be split into roughly half lecture and half practical lab work, where I'll provide some sample problems to work on under our guidance. We'll also be handing out homework every Friday, to be submitted by email to me by Wednesday. We'll also be scheduling some practice competitions out-of-class to try your skills in a more formal environment. Grading is pass/fail, so please don't stress about getting everything "right"; just try your best at the problems we give you. A passing grade is guaranteed if you complete 80% of the homeworks and participate in at least one of the practice competitions.

We haven't chosen a set textbook for the course. However, I have recommended several books in the syllabus that will be useful to you. Competitive Programming 3 is most useful for particular strategies for the ICPC, and so is strongly recommended if you're planning on participating. CLRS and the Algorithm Design Manual are excellent reference books on algorithms with different focuses, and are also recommended.

Introduction So, the name of the course is "Programming Skills Workshop". What we're specifically going to cover is programming competitions: how they work and how to do well in them. During each class period, we'll use the first half of the time for a brief lecture and do some practical lab work in the second half of the period.

Why participate in programming competitions at all?

- Learn how to do fast problem and input analysis.
- Pick up lots of useful algorithms, data structures, and mathematical techniques in a practical setting.
- Develop skills in rapid testing and debugging of code.
- (For some competitions) learn how to work effectively in a team.
- Great practical skills for classes, job interviews, projects.
- It's fun!

What won't programming competitions teach you?

- Limited to small problems that you can solve in a few hours at most.
- No large programs, and limited established codebase.
- Encourages coding "tricks" that aren't advised in a real-world context.

Languages I expect you as a student in this class to have a good working knowledge of at least one modern programming language. That means that if I give you a description of an algorithm, you should be able to quickly and correctly implement that algorithm in your chosen language. When choosing which language to use in a given competition, you should usually use the language that

1. is allowed in the competition (some competitions, such as the ICPC, limit possible languages),
2. you're most fluent in,
3. (in team competitions) that the whole team knows.

Less important considerations are intrinsic execution speed (compiled languages like `C++` are usually 5-20 times faster running an arbitrary algorithm than interpreted languages like `Python`), coding speed (`Python` code will often be 2-3 times shorter than equivalent `C++` code), and library availability (what can you do with, *e.g.*, the `C++ STL` or `Python numpy` and `networkx` packages).

Whatever language you choose you should be fluent in it, meaning: you should know how it works, how to do all the basic functions (like reading/writing from `stdin` and `stdout` as well as files, looping, arrays, *etc.*), and what library functions are available (like complex data structures, implemented algorithms, etc.). We won't cover much in terms of these specifics during the course, but if you have particular questions on `Python` or `Java`, I'm a `Python` expert and Joey Woodson, the TA, is a `Java` expert, so those are the people to ask. I'll usually be using `Python` code for example code, as it's fairly easy to read. Let me know if you have any problems reading the sample code and I'll explain it for you.

Competition Formats There's two competitions we're going to focus on. The first is the ICPC, which is oriented specifically towards college students. The second is the Google Code Jam, which is much broader and open to basically anyone. While the kinds of problems they ask are quite similar, the overall competition format is different and can affect your strategy during the competition.

ICPC The ICPC is a team competition, with teams of three students provided with a single computer. You're given a large number (8-12) of problems to complete in 5 hours. You then have to submit code (in C, C++, or Java) that correctly solves these problems. The automated judge will then tell you whether your submitted solution is correct (Accepted, or AC), gives the wrong answer (Wrong Answer, or WA), or has other problems (Presentation Error, Time Limit Exceeded, Memory Limit Exceeded, etc.) Correct solutions give your team one point. The competition winner is the team with the most points, with ties broken based on when correct solutions were submitted.

Google Code Jam The Google Code Jam is an individual competition. A fewer number of problems, usually 3-4, are provided to complete in 2 1/2 hours. Each problem usually has two different input limits, one where inputs are smaller (the Small problem) and one where inputs can be very large (the Large problem). Each combination of problem and size is given a point value based on how hard the competition organizers think the problem is. Rather than submitting code to be run by Google, you download a small or large input data set, run your code on your own computer and must submit the answers within a time limit. Correct solutions give you the points associated with that problem. Small problems can be resubmitted multiple times if you get the answer incorrect, but you only get one submission for the Large problem. Competitors are ranked on total points, with ties broken based on when correct solutions were submitted.

Strategy

Competition Competition strategy will vary between contests due to differences in contest rules. Usually the key step here is to choose the correct problems to work on. You'll usually want to do easier problems first, and get credit for them quickly before moving on to harder problems. Problem difficulty is going to depend on your own background, but often you can get an idea based on how many other people in the competition have solved that problem; if 98% of contestants have solved it, there's probably a simple solution you've skipped. Similarly, if you're stuck on a problem and you're not sure how to proceed, it's often worthwhile to move on to another problem that you think you can solve. In the Google Code Jam, which has inputs of varying sizes for the same problem, it's important to know that you can get points by

solving Small problems separately from Large problems using "bad" algorithms with poor scaling. In the ICPC, dividing tasks between the team members is essential because there's only one computer, so only one person can be writing code at a time.

Individual Problems Once you've chosen a problem to work on, your goal is to come up with a solution that:

1. Gets the correct answers, and
2. Runs quickly enough.

Ideally, you also want to write clear and concise code, because that will make it easier for you to debug your code if there's a problem.

Here's a general strategy for attacking each problem:

1. Read the problem statement
2. Design an algorithm to solve the problem.
3. Implement and debug an algorithm.
4. Submit your solution and get a result
5. If not correct, go back to an earlier step and try again.

Step 1: Make sure you read the problem statement thoroughly and accurately. Misreading a problem statement is a good way to waste time solving a problem different from the one that was asked and get the right answer to the wrong problem. Make sure you carefully read the input and output specifications - decimal precision of output, line spacing, and so forth. Often competitions will have a consistent input/output formatting that you can prepare for in advance with your own library of code. Check the input limits carefully. Trying to solve a problem with a maximum input size of 10 is very different from trying to solve the same problem with a maximum input size of 10^8.

Step 2:
Design an algorithm. This is the key step. Remember that for programming competitions, unlike for many real life problems, there *is* a known solution - you just have to figure out what it is. Usually you can apply a known algorithm or approach or data structure to the problem to solve it. Very rarely will you be presented with a instance of a problem in the canonical form that you'll see in a textbook; instead you'll have to figure out how you can fit the problem into a known framework. Teaching you these algorithms and approaches and how to identify when a problem can be attacked using a particular method is going to be a major focus of this course. Sometimes, however, you'll see *ad hoc* problems that really are singular and require an obscure or novel approach. These can either be very simple if the approach is straightforward or some of the most difficult problems.

Step 3: Once you think you have an algorithm that solves the problem, you need to implement and debug it. You need to make sure that the output format is correct as specified, that tricky edge cases are handled correctly, and that it will run within the time limit. Often it's useful to come up with a set of small test data that you can solve by hand to check your solution.

Steps 4-5: Ideally when you submit your solution, you'll find that your solution correctly solves all the input cases. If it doesn't, then you'll need to identify the source of the problem. If you ran out of time, then was there a bug in your program or did you choose an algorithm that was too slow on large data sets? If you got a presentation error, double-check your output formatting. If you get wrong answers, then try and find a test case that shows why, and use that test case to rethink your algorithm.

Competitor "Levels" We can roughly divide up competitors into four categories.

Novice The novice is completely new to competitive programming, and may have problems with even basic tasks. When presented with a problem, they'll try to code up input and output but aren't sure how to solve the problem. If they come up with an approach, they'll most likely consistently get the wrong answer.

Beginner Beginners will read the problem and recognize the kind of problem it is but aren't sure the best way to solve it. They may come up with a brute force solution only to find that it runs too slowly before they give up.

Intermediate Intermediates will recognize the problem as one they've seen before and after some thought realize they can solve it with a known algorithm. After implementation and some debugging, they'll be able to correctly solve the problem within the time limits - but only after an hour or two.

Advanced Advanced competitors have enough experience that they'll be able to solve the problem within 20 minutes or less - fast enough to solve several problems during the competition.

Our goal for this class is to get everyone in the class to the intermediate stage - able to solve most of the problems you'll see in competitions.

Implementation Problems The first kind of problems we'll talk about are purely implementation problems: they're not testing your ability to find the right algorithm because the right algorithm is trivial to find. Instead, they're testing your ability to correctly implement that algorithm. These are usually fairly easy, but can be more difficult when you're asked to implement a more complex system.

Here's an example from the Timus Online Judge, #1000. This problem provides two integers as input and asks you to provide their sum as output. No algorithmic challenge at all, you just need to know how to handle input and output.

Here's a sample solution, in Python:

```
In []: import sys
       for line in sys.stdin:
           print '%i' % sum([int(i) for i in line.split()])
```

Brute Force Algorithms The second kind we'll talk about today are problems that can be solved using brute force or complete search algorithms. This simply means that we go through all possible solutions to a problem and stop when we find the correct answer. While this approach is guaranteed to be correct - as long as you're careful to ensure you cover the whole solution space - the solution space is usually quite large and so sometimes this approach is simply infeasible. So before you try a brute force approach, you need to ensure that there aren't too many solutions to examine. Let's look at an example.

> Find all pairs of 5-digit numbers that collectively use all digits from 0 to 9 once each, such that the first number divided by the second is equal to an integer N, where 2 <= N <= 79. The first digit of one of the numbers is allowed to be zero. That is, abcde / fghij = N, where each letter represents a different digit.

The smallest number fghij could be is 01234, while it could only range as high as 98765. In fact, because fghij is the smaller of the two numbers, it can only be as large as 98765 / N. For N=2, this limits us to only about 50K possibilities. This is small enough to do a complete search by choosing each value for fghij in this range, multiplying it by N to get a putative abcde, and then checking to see if each digit is found in the two numbers.

Lab Work

Before beginning, please set up an account on the Sphere Online Judge so you can submit your solution, and the Timus Online Judge for later. Don't forget to write down your account names on the pretest to hand in at the end of class.

Implement solutions to the following problems.

1. Iterated Sums, Sphere Online Judge #15710
2. Tic-Tac-Toe-Tomek, Google Code Jam Qualification Round 2013 (Small & Large)
3. Bullseye, Google Code Jam Round 1A (Small only!)