Computing Contest Strategy Lecture

Introduction

- USACO and IOI are less about programming than they seem.
- Plenty of very good, fast, reliable programmers are terrible at USACO.
- USACO is about Algorithms, Algorithms, Algorithms, Data Structures
- USACO is about Testing & Debugging
- **Parts of a Contest**
  1. Reading *all* the Problems
  2. Thinking about the Problems
  3. Coding
  4. Testing
  5. Debugging
- **Think about strategy during the contest.**

Reading and Thinking about the Problems

- **Why**
  - Since USACO is so much about algorithms, you should potentially spend plenty of time here.
  - You *really really* don't want to spend a good 45-minutes coding an algorithm which won't work, either because it's wrong or it's way too slow. So get it right, before coding anything.
  - Read *all* of the problems first. That way, you can get some idea of how hard the contest is, how long it will take, and, above all, in what order to solve the problems.
  - Think about — and, if possible, solve — *all* of the problems before you start coding.
- **How**
  - On a five-hour contest, try to spend *at least* the first hour thinking about the problems. On a three-hour contest, try to spend at least the first 45-minutes thinking on problems. (One year at camp this was actually enforced in one contest, as a strategy exercise.)
  - If you can, get away from the computer. (Exception: occasionally a calculator is useful.)
  - Once you have a solution, try to think of more. Then pick the simplest one that works.

- Do the Math! Check whether it should run in time. (You can also work backwards: for a clue about the kind of algorithm you're looking for, you can often reverse-engineer the problem limits.)
- Make sure that your algorithm is correct.
  1. It should at least work on the sample input.
  2. If it's easy to do by hand, devise new inputs and try your algorithm on them. (These test cases will be helpful later.)
  3. Consider weird & boundary cases. First, they might break your algorithm. Second, they're often easier to try (by hand) than random cases.
  4. If there's a really simple proof that your algorithm is right, that's great.
     If not, make sure that you're convinced (and could convince someone else). But be careful... try to find a counter-example. (Simple Greedy algorithms are particularly seductive.)
  5. If you earlier thought of a dumb algorithm, compare answers.
- Sketch quick notes so that you won't forget what you were thinking.

Coding

- Your goal during coding is not to produce a program as quickly as possible. Your goal is to code in such a way that you can have a correct program as quickly as possible. This includes time spent debugging. More time now can save you time later.
- Use sensible variable names. A variable named `c` or `gj` could be anything. All variables should have names that you can recognize and understand.
- Code defensively. Use `assert`. It's wonderful. You can take it out later with `#define NDEBUG`. Get used to putting a standard set of assertions in your code. (e.g. `assert(dist[x][y] >= 0)`, `assert(best != -1)`, etc.)
- Compiling with `-Wall` helps, although it won't catch semantic bugs.
- Test different parts of your code as you write them. Independent data structures are easy to test, and thus should always be tested. If your solution has two parts, quickly check the intermediate output of the first part before coding the second part.
- Write simple comments for the tricky bits of your code. This makes sure that you understand it and helps when you go back for testing & debugging.

Computing Contest Strategy Lecture

Testing

- **Test, test, test. Then test some more.** A slow program can easily get half the points; a buggy program may well not.
- Start with the sample input, then go from there.
- If you wrote some small test cases earlier, test your program with them now. If not, do that now.
- If you came up with (or can quickly come up with) a really simple, really easy, really slow algorithm, you're golden. Spend 5-minutes to write the simple program (which will almost certainly be correct, or at least differently incorrect), spend 5-minutes to write a random test case generator, and then run them and compare answers. Do this with 10 or 20 random test cases, and you're pretty much sure that your program works (except possibly for weird edge cases).
- Speaking of which, make sure to always test weird edge cases.
- Similar to above, keep old versions of your program around when optimizing for speed, and make sure that they give the same answers.
- Have your program print extra information (intermediate results, etc.) and check that these are also right.
- Read your code. Especially make sure that you understand the important/tricky bits and that they're correct.

Debugging

This is black magic. You need to find your own black magic. There are, however, a few suggestions about how debugging fits into your overall contest strategy.

- Debugging is hard: you'd much rather have not made the bug in the first place.
- Debugging is hard: at some point, you shall have to just cut your losses.

Breaks & Strategy

Coding is an immersive process, and it needs to be. Still, you do need to step back from time to time and say "I have x hours left in the contest. How can I best use that time?"

- Submit something for every problem! No program gives you zero points. A bad program will give you some points. Thus, make sure your time doesn't slip away without writing something.
- Try to solve problems completely. With USACO scoring, you're often better off with 1 perfect program and 2 quick hacks than with 3 mediocre programs. This means spending time getting the algorithm right, and testing your program to be sure it works.
- **Keep a log during each contest.**
  - Maintaining it will force you to take a quick look at where you stand time-wise.
  - If your contest doesn't go so well, you can refer back to the time log and see how you could have better spent your time (e.g. "I should have spent more than 2 minutes testing that", "I should have given up that program before spending 90 minutes debugging it", etc.)
- Take a short break in the middle! Five minutes stretching your legs, getting a drink, can really help you pace yourself, and gives you a moment to get perspective on the rest of the contest. Five minutes really isn't that much, especially on a five-hour contest (and five-hours is too long to be sitting in one chair anyway). You'll have a fresh perspective on the problems when you return.