Chris's ACM Programming Contest Tips

Written by Christopher Thomas. Last updated 16 Oct. 2006.

This page attempts to provide a few useful guidelines, pointers, and tips for writing solutions to problems given in York University's training sessions for the ACM programming contest.

What is the ACM programming contest, and what happens at these training sessions?

The contest is a competition where teams of students from different universities try to solve as many programming problems as they can from a list within a fixed time limit. The practice sessions are a lot like this, but you're working individually with fewer problems and a shorter time limit.

How do I become good at these contests?

A few things help:

- Have a favourite programming language that you know very well. C++ and Java are both used at the practice sessions.
- Be familiar with commonly used algorithms and data structures (these are typically covered in second- and third-year courses).
- Enjoy programming. If you're doing this for fun, it'll motivate you to practice enough that performance will improve with time.
- Practice a lot. Debugging, in particular, is something that is usually only improved through experience.

Any specific suggestions?

Yes, plenty:

- Know about the following concepts, as you'll end up using them:
  - arrays
  - pointers ("references" in Java)
  - linked lists
  - binary trees
  - hashes
  - graphs
  - recursion
  - induction

    - sorting algorithms (mostly so you know what ones _not_ to use)
    - search algorithms
    - dynamic programming

- Be able to quickly estimate algorithmic complexity.

  This is critical, because problems are usually set up so that there's a simple solution that would take far too long to run to be practical. You need to know how long a given solution will take, and how to modify a solution to change its algorithmic complexity.

- Know the language resources available to you, and use them to the fullest. In particular, you should never have to implement a sorting algorithm by hand, or anything resembling a tree-balancing algorithm. Specific tools include:

    - The `qsort()` function in C will sort a list of elements for you in O(n log n) time (usually).

    - The `std::set` and `std::multiset` templates in C++ will let you sort lists of values efficiently in C++, and `std::map` and `std::multimap` will let you associate key/value pairs and sort on keys efficiently.

    - The `std::set` template is also very handy for making a checklist without having to worry about properly sizing and bounds-checking an array of boolean values.

    - The `std::list` template is handy for queues and other unsorted lists, though it's **extremely bad** for searching for random elements. Use a set or map for that instead (these can also be used as lists).

    - When possible, avoid writing parsing routines. Instead, use `scanf()` (in C) or `cin` (in C++) to extract formatted input for you. Unlike real life, you'll always be given well-formatted input in the contests.

- If you have a solution algorithm that works well except for a few specific cases, add checks to flag these special cases and divert them to different routines. It's usually much easier to write several simple solutions than one ultra-flexible general solution.

- If a lookup table would help, use one. This happens where you have only a few specific special cases, which you already know the answers to; just emit their answers instead of writing additional algorithms. This also happens when you need the first few prime numbers, or something similar. List the values rather than computing them, as long as there are fewer than about a dozen, and you have high confidence in the correctness of your hardcoded values.

Can I see an example?

Sure. Here's one:

- problem
- solution

Noteworthy features of this solution:

- This is a recursive solution. An iterative solution could also be written, but was more complicated. Simpler is usually better.
- I used `std::set` for the checklist of candidates I've already used.
- I tested all candidates in the helper function, instead of the list of ones available, because that's simpler and doesn't change the algorithmic complexity. If it had changed the complexity, it would have been a bad idea.
- I used `std::sstream` instead of mucking around with `sprintf()`. This is slightly slower, but avoids any risk of array overflow problems.
- The list of prime numbers is hard-coded. I can do this because I know the maximum value of **n** is 16, so the biggest pair sum is small enough that there are only a few primes. If there had been a hundred primes, I'd have filled the prime table algorithmically.
- I didn't bother error-checking scanning the input. That would have taken a while to implement, and time is usually pretty short in these contests. Just make sure you can debug adequately if errors do come up (a little error checking is usually worth it to make bugs easier to isolate). I usually sprinkle tattletale statements through the code to track what the program is doing, for this purpose. These are removed for the final version.

Enjoy!