Competitive Programming Contest Training

SI does its best to expose students to fun and helpful topics in Competitive Programming, but success in competitive programming relies mostly on individual preparation. We've discussed methods and resources that may be helpful in your personal training.

In particular, this year we had more students with less prior experience than past years, so it's possible that after this introduction to competitive programming, some of you may not enjoy it as much as you thought you might. But, if you do, this document is for you.

New Student Training

For many students in this camp, this was their first exposure to competitive programming and some students didn't have a full command of the tools their language has to offer and didn't have as much background in general problem solving. For these students, it makes sense to find pedagogically sound learning materials (book, online modules) and spend time learning carefully how to use the methods/functions that reside in an API. A good way to do this is to start with the language documentation, open up a main function, and start creating objects and calling functions/methods to see what they do. Brainstorm some sort of objective that a tool will be able accomplish and then use the tool to achieve that task. Spend some time in this format until feeling comfortable with all the language specific skills discussed in the camp (built in data structures, sorting, lists, functional layout of code, etc.)

Once you feel somewhat confident in your language skills, it makes sense to go back and try to fully understand at the algorithms presented in camp, coding up a generic version of them (maybe with some help looking at the notes or sample code), but in your own style so you understand what every line is doing. From there, try to solve some directed practice problems like the ones from the camp or from any other contest site that has problems sorted by category.

With some algorithmic knowledge, it's worth trying live contests! (In fact, it makes sense to do live contests while you are in the process of completing the steps in the first two paragraphs.)

From there, to continue improving it makes sense to continue to learn new methods and algorithms in a systematic way at your own pace while simultaneously competing when opportunities arise. Make sure to try to get your school involved as many local areas host competitions, like UCF does. But, even without your school involvement, there's both USACO and Codeforces.

Live Contests or Virtual and Simulated Competitions

Codeforces (http://codeforces.com/, Div 1/2/3, upsolve with editorials)

USACO Monthlies (http://www.usaco.org/)

Organized Problems by Category, Sorted by Difficulty

CSES (https://cses.fi/problemset/)

Large Collegiate Problem Database

Kattis (https://open.kattis.com/)

Beginning Resources

UCF High School Contest Archive
https://hspt.ucfprogrammingteam.org/index.php/past-contests/past-problem-sets
https://hspt.ucfprogrammingteam.org/index.php/past-contests/past-problem-sets-online

UCF Local Programming Contest Archive
https://lpc.ucfprogrammingteam.org/index.php/past-contests/past-problem-sets

Arup's Learning Modules/Kattis List
https://www.cs.ucf.edu/~dmarino/progcontests/modules/
https://www.cs.ucf.edu/~dmarino/progcontests/kattisproblist/

CP Methods to Solve Problem List (for Kattis)
https://cpbook.net/methodstosolve

Advanced Resources

Algorithms Live (https://www.youtube.com/channel/UCBLr7ISa_YDy5qeATupf26w)

Codeforces Problem Topics (http://codeforces.com/blog/entry/55274)

USACO Guide (https://usaco.guide/)

Ben Qi's github (https://github.com/bqi343/USACO)

Components to Success in Programming Contests

Algorithmic and Data Structure Knowledge

Some problems are near impossible to solve without some pre-requisite knowledge. For these problems, that knowledge is necessary (but not always sufficient) to solve the problem in contest. Thus, a component of success in competitive programming is your knowledge of algorithms and data structures.

Problem Solving Skills

The solution to a problem might involve a data structure that you know, but you might not solve the problem because you didn't see how to utilize the data structure in a solution to the problem or because you couldn't devise the solution idea on paper. Ultimately, this turns out to be the most difficult aspect of programming contests to master. It's more of an art than a science. Problem authors intentionally and unintentionally try to deceive contestants in many different ways, and contestants get better seeing through the deception via practice. In addition, there are simply many clever problems out there with non-intuitive solutions. The more exposure to ad-hoc problems you have, the better you will get at solving problems.

Implementation Skills

Once you know how to solve a problem on paper, there is the task of implementing those sketched out steps in the computer. This component of a programming contest comes under implementation skills. Naturally, for programming contests, the key is to come up with both correct implementations that run "fast" enough in the shortest amount of time possible. The most typical way to get better at implementation skills is simply to implement problems where you've already worked out the solution on paper, or have read an editorial and have a decent understanding of the solution path. Another way to get better at implementation is to read other contestants' code (Codeforces, Google Code Jam, Blogs/Github of top competitors, etc.) and pick up some of their implementation tricks.

Debugging Skills

Some might put this in the category "Implementation Skills", but based on its importance, we've placed it in its own category. After the initial implementation, there might be bugs, either caused by implementing an idea you understand incorrectly or by having some initial misunderstanding of the problem or solution. Debugging is the process by which we uncover ***and fix*** these errors. Debugging requires great patience and the best competitors are excellent at finding their bugs quickly. The best way to improve is to debug on your own without getting help from others. Once you find errors, catalog them (write them on a notecard or a log). Over time, you might get to see patterns of the types of errors you tend to make. It might be useful to try to categorize those errors and write them out on a checklist of reminders that you keep with you when practicing. In addition, practicing making test cases and practicing writing brute force solutions to generate small data for testing can be incredibly important tools for debugging.

Methods to Practice and Improve

Live/Simulated Contest

The benefit of competing in a real online contest or a simulated one (either virtual or where you set a timer and do problems from a single contest under a contest setting) is that you practice what it feels like to problem solve and implement under time pressure as well as practice making decisions where time matters. The drawback is that many online contests are at inconvenient times or the length of some contests is too long to fit into a regular schedule, so waiting until you have 4 or 5 free hours in a row before you train at all may not give you much training.

On the positive side, generally speaking, live contests are fun and you can improve your rating on various sites, which gives you a relative idea of where you stand. Everyone has to decide for themselves how often they run live/virtual contests, taking into account both the benefits and drawbacks.

This type of training helps you work on your problem-solving skills because in an arbitrary contest, no one will tell you the method to solve a problem, nor will there be any labels on most problems. Naturally, all four components come into play, but this is prime practice for both problem-solving skills and decision making.

After contests, it's important to analyze the decisions you made during contest and see if there were any actions you could have taken to improve how you performed based on your knowledge base. Often times, for collegiate contests, problem selection tends to be an issue that many individuals and teams can improve.

Upsolving Problems

Solving a problem on your own time, typically after the contest it was in has completed, is called "upsolving." It makes sense to upsolve hard problems. In many real contests, you simply don't even have time to get to the harder problems because you were busy working on easier problems. But, when there is no clock, it's a perfect time to read and work on these harder problems. Some might take days or even weeks to conquer, but almost always, you learn a great deal from the process. One can either upsolve a problem by not seeking out any information about it, or they can read an editorial/talk to someone about the solution idea. If you do the former, you get to practice the problem solving phase without the time pressure. If you do the latter, you don't practice problem solving and only get to practice implementation. There is certainly a time and place for both, you have to decide how often and how quickly you will seek help when attempting to upsolve a problem. If it's a problem you care about, the recommendation is to at least devote several hours (5 or more) before giving up and reading an editorial or talking to someone. You can't do this with all problems of course, but there is some value to working on problem solving in this sort of setting. Also, it might be worthwhile to peek at a scoreboard to see just how difficult the problem was. If it was way out of your league, it might be best just to read the editorial. But, if the scoreboard indicates that the problem is within your upper range of ability, it's generally more valuable to spend some time with the problem before consulting resources.

Reading Books or Notes/Watching Videos

When you want to learn new material on a specified topic (say Fast Fourier Transform, Convex Hull, etc.) one way to learn about it is to read lecture notes or a book section about the specific algorithm. Most of these sources won't be for competitive programming, so it's likely that they'll just explain the steps of the algorithm and why it works, with any attention to implementation details. Competitive Programming 3 is well-suited for competitive programming and should be consulted early on. But occasionally, it might be best to use a traditionally academic source. Alternatively, you can learn new material by watching videos and the YouTube show "Algorithms Live" has some good episode with explanations of various algorithms and data structures. In addition, if you scour the internet, you can find quite a few videos on various topics that come up in competitive programming.

Reading a book or watching a video is not enough to solidify relevant information. It's very important to code the idea for yourself in a generic setting without the pressure of a problem. The goal of this is to truly understand the ins and outs of the new topic you are learning. More than likely, after you get your idea to work, it'll make sense to refactor your code and write it more cleanly and concisely after you understand the idea well. (In addition, for many algorithms, it's inevitable that you'll have to modify the code to be able to use it to solve a problem in a contest, so it's important to understand how the code works.)

This technique specifically addresses learning about algorithms and data structures. How do you choose what to learn? You can either follow one of the aforementioned sources as a guideline, skipping over things you know and going over things you don't. Alternatively, whenever you do a contest, afterwards, if someone mentions that a particular technique is necessary to solve a problem, you can go and learn about that technique and then read that problem again.

Reimplementation

On a few occasions, it's worth reimplementing a particular algorithm or idea after you understand it well to see if you can speed up your implementation or make the code more concise. For some patterns this is unnecessary simply because they come up so often in contests and upsolving. For other patterns, this might be worthwhile. Also, looking at code after the fact and analyzing it might lead you to improving your coding style.

Hard Problem List

Keep a list of hard problems you would like to solve. Order it based on your priority of what you want to learn. When you have free time, chip away at the list.

Last but on least, here are Spencer Compton's (2018 IOI Gold Medalist) Words of Wisdom:

Struggling is important (both in terms of problem-solving, and debugging your own code), as this is the only way you grow.