Problem

You want to become a better coder. You know you have to practice but you are not sure how.

Solution

Your practice routine may become more fruitful if you put things into order.

A. Learn new algorithms and techniques. You can pick up theory from various websites and books [1]. Make sure you understand each new concept thoroughly and play around with it until you master it.

B. Focus on specific groups of problems. Instead of picking random problems, devote one week to a certain topic, then the next to another one, then another one etc. [2]. Put higher priority on recently learned theory and on your weak spots.

C. Vary your sources for problems [3].

D. Know your language. Be familiar with its variable types and its syntax. Experiment, read documentations and others' source codes.

E. When an SRM approaches, run complete timed simulations of the competition. Do not distract yourself with new algorithms or other types of contests. Take it easy the day before the SRM: no mind teasers, just finger exercises.

F. Solve all practice problems the same way as you would in an SRM [4]. Think carefully and test exhaustively. No peeking at the editorial or the others' solutions unless you are completely out of ideas.

G. Track your progress. Keep record of all solved problems, along with dates, brief solutions and a list of challenges you faced in the solving process. Use this to periodically summarize common mistakes you make. Log problems you couldn't solve and come back to them later. Note unfamiliar theory you stumble upon and learn it.

H. Practice hard, practice regularly. One hour per day is the absolute minimum.

Discussion

A. Almost all Top Coder problems require some knowledge of algorithms. While most frequent topics are covered in this book, you will certainly find plenty of additional theory that is applied in many situations. When choosing what to study next from the vast sea of computer science, the simplest strategy is to review the analysis of problems you couldn't solve. If the solution of a problem involves some standard technique, algorithm or data structure, the editorial would surely name it and perhaps mention where you could get additional information about it. Another self-tutoring approach is to follow some book [1] or an organized training program [3]. And, of course, you could always Google something you have seen mentioned somewhere, or have known to be useful in competitions, but have never actually gotten down to delving deeper into what it is. Thus it is helpful to have a list with unfamiliar concepts you have encountered so that you know what to study when you have time. But no matter how you pick your next subject, it is crucial that you fully understand how and why it works. Standard algorithms are often disguised and sometimes even modified when used in competitions, so simply memorizing some steps (or worse, some piece of code) does not guarantee you will know when and how to use them. Try to get to the heart of the algorithm, to feel its logic and make it become natural as if it is something you have invented by yourself. A few tricks that help to achieve this include working out several examples by hand, writing at least two different implementations and tracing how they work on various examples, modifying the code to see how changes affect its behavior, reinventing and proving the algorithm/technique/data structure after you have forgotten it, thinking of different scenarios in which it might be applied, and solving a lot of problems involving it. [TODO: examples: dijkstra max edge, tricky dijkstra, kmp table]

B. Practicing on problems that gravitate around a common topic, as opposed to selecting tasks at random, enables you to see connections between different problems with somehow similar solutions [5]. It provides you with further insight into the concept behind them and makes it much easier for you to spot such a problem in a competition, to quickly find a solution and code it correctly. So if you have learned something new recently, practice on it. For example, each article in the Top Coder Algorithm Tutorials [1] or the USACO Training Program [3] ends with a list of problems to practice on, and if you find this list insufficient, you can always use the Top Coder Problem Archive [2] or simply Google to look for more problems related to it. The Problem Archive is extremely helpful if you want to strengthen your skills in a given type of problems that you have difficulties with. In any case, you should do this when there is enough time until the next competition, for this type of training has its drawbacks [4]. As much as it shows you how to solve some problems, it may also blind you from seeing the solutions of others. It often happens that a coder learns some powerful new algorithm or data

structure and suddenly every new problem tempts them to use it, when in reality the right approach is much different, and usually much simpler. So have an open mind when narrowing your choice of problems, and leave some time for other problems before you start competing.

C. Top Coder has numerous problems that cover a wide variety of topics and it probably takes a lifetime to solve them all. However, it is a good idea to sometimes take a look at other online judges [3]. For one thing, distracting yourself from the Problem Archive after browsing it for several nights can be quite refreshing, especially if you feel you are getting stuck more and more often. Different contests have different rules and, more or less, different problems, and while they all require and train an algorithmic type of thinking, they tickle your brain in different ways. Just as bodybuilders periodically switch exercises for some groups of muscles so that they never get used to doing the same thing and continue to grow, you should not let your mind get used to the same type of competition. Muscles have to be shocked if you want them to develop, and the same thing goes for your brain. Eventually this variety will make you see things from a broader perspective and when you go back to Top Coder after doing, for example, Google Code Jam or USACO, you will handle the same problems with greater ease.

D. Get comfortable with the programming language you are using. It is annoying to spend precious minutes figuring out why you code is doing some weird stuff in the middle of an SRM. Therefore you should first become aware of what your language does "under the hood" - how values are stored and passed, how memory is allocated, which situations cause an error and which produce unpredictable results, how some data structures work and how much time and memory they consume, etc. To achieve this, you should read the documentation and experiment with various portions of code. For example, you should know when an integer overflows, when a floating point variable loses precision, that red-black trees are sometimes slow, that changes in return values after removing cout-statements in C++ are usually caused by accessing the wrong memory (i.e. small arrays or indexing out of bounds), and other tricky details that might mess up your solution. Check out what the "scriptures" say [6], then play around: write simple programs and see what happens if you do this or that.
It also frequently happens that one coder writes two screens of source code while another solves the same problem in five lines using some fancy functions. So you should know what you can do with your language. For example, it is redundant to code, or even copy-paste a heap-like data structure when C++ has its set and priority_queue. Take maximum advantage of the functionality you have at hand. Something great about Top Coder is that you can view the other coders' solutions, so make use of it and learn from them. You will see new functions and types, new elegant ways to handle certain situations that spare you time and effort. Read their documentation

and experiment with these unfamiliar features as you do when learning some new theory. Reading other source codes also exposes you to fruitful ideas that you might otherwise never come up with. Have an open mind and be willing to try out the others' styles of coding.

E. As an SRM approaches, you should prepare your mind for doing an SRM. Allocate two hours each day when no one will disturb you, pick a past SRM with problems you haven't seen, set the timer to 1h 15m and start competing, at least figuratively. When the time runs out, test all problems with the system tests and compare your score to the others' in the match results to see how you have done. Participating in a Top Coder Single Round Match is much different than any other competition, and has nothing to do with using the Problem Archive as a source for programming exercise. Here you have no idea of what type the problems will be, the clock is ticking mercilessly, and your rating (a symbol of your reputation as a coder) is at stake. It takes a lot of effort to turn the stressful atmosphere of an SRM into something natural and while the best way to fight the stress is participating in actual SRM's, you will definitely help yourself by doing timed simulations. This teaches you time management, tests your approach tactics, distracts you from the last things you have learned and gets you ready to employ all of your programming knowledge and coding skills for doing your best in the SRM. That way you can also observe your behavior during a timed competition and try to be calmer. Although it is much harder to "be cool" while doing the real thing, practicing like this before it will surely make it easier. But most importantly, it will help you build your competing strategy. Make sure you feel comfortable with this strategy and use it when the SRM starts; it is usually harmful to experiment with different tactics during the match. Finally, if the day of the event is today or tomorrow, do not push yourself to the limit. This is the time to relax and gain confidence by exercising with easy problems. It is also a great time to review the list of common mistakes (see G.) so you will know what to avoid.

F. The benefit you receive from practicing depends greatly on how you solve practice problems. An easy and popular approach is to find an idea for a solution and start coding immediately, figuring out the details on-the-go, then debug until your code works on the example tests and submit. If it fails, you debug some more. If it works, you are proud and move along to the next task; if you get stuck, you look at the editorial and again move along. Overall, you don't put too much effort since this is no real competition. Although better than nothing, this is not quite the most efficient strategy. First of all, you should approach practice problems the same way you approach problems in a real SRM. Think carefully and plan your approach [4]: explore different potential paths to a solution, choose one and make sure your idea is correct. Clarify all implementation details in your head before you start coding. Use pen and paper, and do not touch the keyboard unless you know exactly what you will code and why it will work. Five minutes spent planning and thinking might spare you thirty minutes of debugging, or coding a long solution when a shorter one exists, or worse, coding the wrong solution. Then

write the source code carefully and think how each piece of code may break. Before submitting, test your code, and test it a lot. Your goal should be to pass all system tests upon your first submission. Each successive resubmit is a minor failure, equivalent to zero points in an SRM. So there is no need to rush, especially when you are not competing. Remember, a correct submission for 267 points is worth much more than a wrong one for 456. Quickness will eventually come by itself; it is your technique that should get better first. And don't be tempted to see how other people have solved it before you do - if you don't pass the system tests, go back to your code and find out what's wrong.

Secondly, if nothing works, it is natural to admit defeat and read the editorial, follow the steps, code the author's solution and debug until it passes all tests. Yet this is another thing many coders underestimate - the importance of a problem that made them give up. Merely understanding and implementing the write-up does not guarantee you will be able to recognize it in another problem, not to mention inventing a different solution based on the same idea. It is important to read the problem analysis carefully and actively [4]. It helps to ask yourself questions like "Have I seen such a solution before? If yes, why didn't I think of it? If not, could I have thought of it? Is the idea similar to something I've seen before? What hint would have probably lead me to it? Will I bear it in mind next time I see a problem like that one? What new tricks does it use? In which other cases will this approach work? What if the problem statement was slightly different - will it work again, or will I have to change it a bit, or will it require a totally different algorithm?". If there is some unfamiliar theory involved, you should put it in the list of topics-to-be-studied and reveal its mysteries as soon as possible (see A.). The main goal is to understand - even feel - the concept behind the solution, way beyond the context of the particular problem; to extract the essence of the proposed approach and be able to apply it in different circumstances. And to make sure you have truly comprehended the idea, do not start coding immediately. Have a to-do list of problems and put that one in it, along with the date, then come back to it after a couple of weeks. By then there should be a few problems on top of it (assuming you have practiced conscientiously) and you would have probably forgotten the editorial. In any case, do not try to remember the solution - approach it with a clear mind and try to make it up by yourself. If failure strikes again, repeat the whole procedure. It may seem irritating, but it is far more efficient than the other way round.

Third, after you successfully solve the problem, see how other people have solved it. The editorial presents an approach, sometimes outlining one or two alternatives, but usually the implementation details are left for the reader and these can make the difference between a fast correct submission and a wrong or slow one. Besides fancy programming tricks (see D.) this will present you with other interesting ways to approach the problem, and definitely a lot more ways to code it. As a side benefit, you will learn to read and understand code quickly, something that is extremely useful during

the challenge phase.

You should also figure out why your attempt(s) was/were unsuccessful. Ask yourself things like “Was it some small bug? Was it a series of bugs, resulting from hasty and messy coding? Does it simply miss some trivial case? Or perhaps it only solves a trivial case of the general problem? Did I misunderstand the problem statement?”. Have those things in mind and try not to repeat the same mistakes (see G.).

G. Something that will significantly enhance your performance is keeping a log file with all problems you have solved (or tried to solve). For each problem record the current date, its name and location, a very short problem statement, its type and level of difficulty, a brief description of its solution(s), and most importantly, a list of all difficulties you encountered while struggling with it. If you know the time you solved each problem, you become aware of the frequency and intensity of your practice routine (i.e. you will see how lazy you are and hopefully become more motivated), and you will easily track your gradual improvement. Summarizing each solution helps to clarify it and reemphasize its basic idea so it will be easier to think of it in the future. And by noting all challenges and bugs you faced during the solving process (including the most insignificant ones), you will be able to learn from your own mistakes [7]. It will surprise you how often the same mishaps will reoccur in your log file - minor pitfalls that you often ignore and immediately forget once they are overcome. But programming becomes much faster and much easier if they do not occur in the first place, and this is something that cannot happen by itself. Therefore you should scan your log file from time to time and compile a separate list with all common mistakes you make. Read that list occasionally, especially before an SRM, and do your best to avoid these mistakes. Think about them before you start coding, while you code and while you debug. And do you best not make them ever again.

By keeping this sort of programming diary, you also create your own problem archive that can serve many useful purposes. You could review all tasks of a certain type, or find several problems that require the same approach, or see which problems appear most difficult to you. Of course, it is tedious to log every single task, so after a while you will most probably ignore some of them. However, if you choose not to log one, this means you quickly came up with the solution, you coded it with no difficulties, it had no bugs and it worked correctly the first time you ran it. As this is an extremely rare scenario, it is best to write down everything.

Here is an excerpt from an example “problems log.txt”:

And an example summary of common mistakes:
- misread/misinterpreted problem statement - read carefully and see example test cases
- missed corner cases

- type overflow
- wrongly sized arrays
- stack overflow
- double arithmetic
- mistyped/wrongly named variables

H. Succeeding in anything requires dedication and consistent effort. The time you spend practicing (the right way) is proportional to your improvement, so it is a good idea to know what you expect and how much you would sacrifice for it. They say that “Patience, persistence and perspiration make an unbeatable combination for success” (Napoleon Hill), yet it is also true that “If you love what you are doing, you will be successful” (Albert Schweitzer). Practicing should not feel like a burden or an obligation; it should rather be like pleasant mind teasing, with each new problem like a provocative puzzle that teaches you something. You should spend these hours in the arena by desire, not just by will. And they will be a lot, for having profound theoretical knowledge is far from enough if you want to become a great coder. Finding the right solution often requires creativity and intuition which only come with rich experience.
It is also important to realize that failure will be your constant companion all the way to the top. As clichéd as it may sound, everyone has had his bad moments, and each victory is preceded by numerous disasters. Failing should not discourage you; on the contrary, it means you still have a lot to learn and should motivate you to practice more and practice harder. If you practice properly, your results will steadily start getting better. Of course, a high score in a match does not mean you know everything and should stop practicing - you should never stop practicing, not to mention that programming is a discipline in which you always have to be up-to-date or otherwise you fall behind.

And last, but not least - rest. If it is late at night, your eyelids are growing heavy, the monitor is getting blur and each fixed bug seems to cause another, just go to bed. It is much better to practice when you are fresh - you think faster, you are able to consider more details while making decisions and your mind is clear from any distracting thoughts and emotions you have accumulated during the day. Besides, just as muscle cells grow while the body is resting between workouts, your brain needs to have a break and subconsciously digest all the information it has collected while you were practicing. This is when the magic happens that gives you the ability to solve a task and have no idea how exactly you came up with the solution.
As a conclusion, remember that solving programming problems is a long-term investment: as much as it polishes your coding skills, it also immensely improves your reasoning abilities in general.

See Also

1. Algorithm sources:
a. Wikipedia http://en.wikipedia.org/wiki/List_of_algorithms
b. Top Coder Algorithm Tutorials http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index
c. Introduction to Algorithms http://books.google.com/books?id=NLngYyWFl_YC&lpg=PP1&ots=BwTuHF2hJ8&dq=algorithms&pg=PP1#v=onepage&q=&f=false [TODO: different link?]
d. Algorithm wiki http://www.algorithm-code.com/
e. [TODO: add more?]
2. Top Coder Problem Archive http://www.topcoder.com/tc?module=ProblemArchive
3. Online judges and problem archives
a. USACO Training http://train.usaco.org/usacogate
b. USACO Contests http://ace.delos.com/ioigate
c. Google Code Jam http://code.google.com/codejam/
d. Sphere Online Judge http://www.spoj.pl/
e. Timus Online Judge http://acm.timus.ru/
f. International Olympiad in Informatics http://ioinformatics.org/history.shtml
g. Baltic Olympiad in Informatics http://www.csc.kth.se/contest/boi/about.php
h. Central European Olympiad in Informatics http://en.wikipedia.org/wiki/Central_European_Olympiad_in_Informatics
i. [TODO: add more/ remove some?]
4. “Planning an Approach to the Problem” [TODO: chapter link]
5. “Recognizing Problem Types” [TODO: chapter link]
6. [TODO: link to documentations for different languages]
7. “Finding and Avoiding Mistakes” [TODO: chapter link]

Comments

syg96

Great thing! Read it all... Very interesting!

"One hour per day is the absolute minimum."
I had more when I participated in ICPC contests. Now I'm retired and hour per day of pure practicing sounds like hell=) Seems that I won't be red for a long time=)

About C. It is a good idea to vary the things you do. Then you will become good at everything without getting bored. For example, preparing problems for contests (like TopCoder SRM=)) may add something to your skills.

40% of Part F looks like short retelling of recipe "Planning an Approach to

the Problem" which I just wrote. The same ideas, pen and paper, code planning, problem similarity and etc. It is cool that we came to same thoughts independently!