Training ICPC Teams: A Technical Guide

Rujia Liu

Department of Computer Science and Technology
Tsinghua University
Beijing 100084, China

Abstract. ACM/ICPC is a world-wide annual programming contest over thirty years. As in other competitions, the result of a team largely depends on how they are trained. Methods and guidelines introduced in this article are extracted from various successful teams, including regional champions and world finals medalists. The long-term training is divided into three aspects. Principles and practical suggestions are given for each, providing both theoretical and practical information for instructors, coaches and competitors.

Key words: programming contest, ACM/ICPC, online judge, team training, algorithm, data structure, team collaboration

1 Introduction

Programming contests have grown steadily in recent years. Programming lovers have a plenty of choices on what contests to attend. Nevertheless, ICPC is still one of the most influencing world-wide programming contests, attracting a huge number of people representing thousands of schools in over sixty countries.

This article discusses some technical aspects on ICPC team training, which can hopefully help some teams improving their training process and hence then results. Note that there are also quite a few non-technical issues that greatly affect a team's performance in a contest, but they're beyond the scope of this article.

2 Getting started

Programming is fun. This is the main reason for most beginners to stay in ICPC. Problem solving is even more fun, exciting and challenging. My advice is simple: keeping these good feelings in mind, do whatever you love, until you find it necessary to think about something serious.

2.1 Solving Problems Online

When getting started, practice is much more important than theory. Everyone's encouraged to program as much as he can, as long as enthusiasm is perfectly kept.

But there is one thing you need to know first: ICPC concentrates on problem solving rather than software engineering, so practicing real-world programming for software development (networking, GUI, concurrent programming etc) does not help much in ICPC.

Online Judges. It's recommended that you try problems especially designed for ICPC, which can be easily found in many world-famous 24-hour Online Judges [18], [19] [20] [21] [22] [23] [24]. There are also online contests held in these sites, which are essential in team training, as we'll see in this paper soon.

Websites for UVa Online Judge Users. Be warned, though. Solving many problems found in the online-judges requires more than programming skills. That's why some problems are solved by a lot less people than other. An interesting website closely related to UVa Online-Judge is [29], which gives you advice of what problems to solve next. It also takes care of your training progress, which is great if you're designing your own training schedule based on UVa. There are also other amazing websites which helped a lot of competitors from novice to advanced [28] [30] [31]. Interestingly, a large part of [28] is written and published in a book [2], which could be downloaded freely from [28].

TopCoder. Besides ICPC-style Online Judges, many programmers find it helpful to participate in algorithm contests by TopCoder Inc. [25] There are a lot of problems in different categories and difficulties available in the practice room, as well as frequently held Single Round Matches (SRM) and tournaments. What's more, there are tutorial articles about commonly-used algorithms and problem analysis for every contest it held. One of the most useful and unique features, for both novices and advanced programmers, is that you have chances to read codes from everyone there, including some of the most talented programmers in the world. This could be extremely helpful to your coding ability, if you spend a lot time to analyze merits and defects of others' code, just as fiction writers and movie directors do.

2.2 Tackling Easy Problems

Most people got started by solving easy problems. Here by easy problems we mean the problems in which you only need to do what you're asked to do, i.e a direct implementation of the problem description. For example, do some statistics, string processing or simulation. These problems mainly require some coding ability but not any sophisticated algorithm or deeper mathematics.

Elementary data structures. They should be studied together with the programming language. The concept of algorithm complexity should be established at the same time, as told in every book on data structure and algorithms. Both C++ and Java programmers can use stacks, queues, deques[1] directly, but new programmers are recommended to implement their own, as to ensure solid coding ability. Many beginners tend to write extremely long and complex implementations of basic data structures, which are both error-prone and hard to debug. They're encouraged to read elegant codes (e.g from [25]), until they could

[1] and even priority queues, sets and maps, both hash-based or comparison-based

easily write (or at least mimic) their own simple and correct codes, with little effort.

String manipulation. Strings are involved in most programming contests due to its popularity and difficulty. Some of them are easy in algorithm, but complex in implementation. In most string problems (including problems that need special I/O formatting), familiarity with library functions and clever tricks play a more important role than algorithms. Some people find it relatively easier in Java to manipulate strings[2], others prefer to write snippets in the contest material, illustrating commonly-used techniques. As for basic data structures, it is recommended for beginners to learn from others' code.

Sorting and Searching. Although there are deep theories behind sorting and searching, beginners usually concentrate on the usage of algorithms rather than implementation details or philosophy behind them, since many classic sorting and searching algorithms are available in standard library of most popular programming languages. However, please keep in mind that major theories behind need to be studies once you've mastered their usage.

2.3 Improving Your Coding Ability

By "programming ability", we mean coding, debugging and testing. Though being individual skills, these abilities greatly affect cooperation too. It's better for the team members to use the same language and similar coding conventions, if one cannot find his bug and asks another person to read his code. Though programming is the very first skill, it needs improving all the time. As an example, coding complex algorithm can only be trained after studying these algorithms.

Language Considerations. Most contestants use C/C++ for competition, but sometimes it is more convenient to use Java instead. For example, Java provides a handy BigInteger class[3], which saves some typing work (C++ programmers needs to type the prepared code from standard code library into the computer) and reduce the probability of silly bugs (there might be some typos). Java also has a collection of regular expression functions, which is great for some string-manipulation problems[4]. Also, some people find it easier to debug Java programs, so it's always worth trying it, if you haven't done so.

Refining Your Own Code. Coding is extremely flexible, especially for C++ programs. Two programs implementing the same algorithm can look very different. As stated before, it is always a good idea to look at others' programs, especially ones that are generally believed clean and elegant, but don't forget a more basic but important way to practice: rewrite your code for the same problem again and again, until you find it satisfying, just like fiction writers do (again!). Note that rewriting does not mean to recite your code then enter exactly the same one again and again. The crucial part is the rethinking of the

[2] even if you don't use regular expressions

[3] There is also a BigDecimal class, but is not required for most of the time

[4] Though it's better to have language-neutral problems in contests, it isn't always the case

structure and style of your code before each rewriting, possibly after reading others' programs. As a by-product, you can also gradually accumulate your own snippets, standard code library, and even coding reminders discussed later.

Testing and Debugging. Testing and debugging are best described as "art" rather than "technical". There are a lot of general-purpose discussions on this topic. There are also tips specially for competitors, as in Section 2.7 of [3]. In order to improve, never separate coding from testing and debugging. Clean and elegant codes are usually easier to test and debug than long and confusing codes, so you should always take into account testing and debugging difficulty in your code training. An important feature in ICPC is that three people have only one computer, so it's often risky to find an unknown bug of a complex program by tracing (i.e. stepping or running with breakpoints, inspect watches etc), since it usually takes a lot of time. A good way to avoid tracing is to force yourself to write programs in text editors like EditPlus, with shortcut keys only to compile and run your program. Adding some debugging outputs in addition to what is asked in the problem description usually helps, but it is again an art. It's also helpful to learn some language tricks (macros, assertions etc), but the philosophy behind your testing and debugging method is the most crucial.

Know Your Defects. Even after heavy training, it's likely that you're still making small mistakes that you've already made a lot of times. But the more you practice, the better you know yourself, which is very important. Imagine you have just coded a program but failed the first sample in the problem description. You cannot find your bug in 10 minutes, but someone else claimed that he found another easy problem, hence want to use the computer. Then it's usually better to leave the computer, sit down by the table and examine the printed code with eyes and pencils. Now you can't apply any debugging techniques because you're not on keyboard, whether and how fast you can find the bugs largely depends on your carefulness, and how well you know yourself. An easy way to know yourself better, is recording your common mistakes, especially the ones that cost you a lot of time to find. If all these records are well organized as a reminder (discussed later), they'll help you greatly in real contests.

3 Enhancing Your Theoretical Background

Trying to solve more problems is good, but the quantity is not the most important thing. When you managed to solve 50 easier problems somewhere, it's better to seek for more challenges.

3.1 Mathematics

In real contests and online judges, there are a large number of problems that require less programming skills but more maths. Most mathematics involved can be divided in these categories:

Arithmetics. Arithmetics in computer is not exactly what we learnt in high school mathematics. Actually this is something related to computer architecture.

Programmers should know the way integers and real numbers are represented internally, be familiar with binaries (and some tricks with bits), and be able to code high-precision numbers (Java programmers can use BigInteger class, which deals with an arbitrary numeric bases), fractions and complex numbers.

Combinatorics. being an important branch of mathematics on itself, Combinatorics is essential for competitors to estimate asymptotic complexity of algorithms, as well as to solve problems related to counting and probability. Besides basic principles, formulae and theorems, case studies (e.g famous integer sequences, different proofs of various equations involving binomial coefficients) are necessary to acquire adequate understanding of combinatorics, especially problems involving recurrence. Sometimes Pólya theorem and generating functions are required to solve a combinatorics problem.

Number theory. Number theory is popular in programming contests[5] partially because of its simplicity in problem statement. Number theory is extremely deep, but most problems in ICPC requires a rather small set of background knowledge such as prime numbers (prime table generation and primality testing), greatest common divisor, modular arithmetic and congruence, e.g. solving linear congruences, and Euler's $\phi(n)$. By the way, the BigInteger class in Java comes with quite a few number-theoretic functions like greatest common divisor, modular exponentiation, modular inverse and primality testing, which can be extremely helpful in some problems.

Games. By the word "games" we mean combinational games. In most cases, it's sufficient to understand how to brute force the answer (possibly by minimax search with $\alpha - \beta$ pruning), how to solve the Nim-type games with the Sprague-Grundy theorem[9]. Ad-hoc game problems sometimes appear too, some of them could also be solved with the SG theory, but others need some math work or are solvable via other methods like dynamic programming or graph algorithms. Though more comprehensive books exist [7] [8], I feel it's unnecessary for ICPC contestants to dive into it too deep.

Other mathematical concepts and algorithm are helpful, too. For example, matrix multiplication can be used to solve linear recurrence more quickly, Gauss elimination is required in many problems related to Markov chains, permutation group usually helps in card shuffling and sorting by swapping, numerical analysis can do a great job in some computation-oriented problems. All in all, it is usually a good idea to collect formulae, problems, algorithms and codes that are not too widely-used but could help a lot in certain circumstances.

Readers are encouraged to read this excellent book on mathematics [10], for combinatorics, probability, number theory and other topics not covered here.

[5] For example, most recent regional contests in Japan has a pure number-theoretic problem

3.2 Algorithms Designing Techniques

It's great if a few mathematical conclusions are enough to solve the problem, but in most cases this isn't true (after all, ICPC is not a math contest). You have to design an algorithm, carefully code it[6] and make it work.

Brute Force. Brute force is one of the most important problem-solving techniques applicable to a large range of problems, when the problem instances are small enough, or efficiency is not particularly important. It's also a good glue for different algorithms. Apart of simple brute force algorithms that only involve a few nested loops, people have to get familiar with backtracking. Although theoretically easy, backtracking is a challenge in algorithm design, coding and debugging for most beginners. The first thing to learn is generate-and-testing different structures of solutions like subsets and permutations, then pruning in the search tree, and estimating the worst-case performance. Fine tunings in code-level usually help a lot in problems involving backtracking, so it's a good practice for every programmer to try different ways to implement the same algorithm, and pick the one with maximal satisfaction. As a rule of thumbs, people can hardly master backtracking without a sufficient amount of practice on various types of problems.

Dynamic Programming. Frankly, dynamic programming is not easy to understand at first, but once you understand it, everything's natural and beautiful. It's a pattern to analyze problems and design algorithms[7], and the ability of solving dynamic programming problems increases as the number of problems you tried. Don't miss the classic problems that are taught in almost every algorithm textbooks like Longest Common Subsequence (LCS), Longest Increasing Subsequence (LIS), Optimal Binary Search Tree (OBST), 0-1 knapsack, edit distance, Matrix Chain Product etc. But as the difficulty and complexity increasing, other theoretical knowledge and skills are required, as we'll see shortly.

Data structures. Data structures play an important role in time-critical problems. Priority queues, union-find sets, (augmented) interval trees, (augmented) balanced BSTs and binary indexed trees often help to reduce time complexities on other algorithms, frequently appeared in tough contests in Europe and Asia. There are also some pure data structure problems that involve statistics or simulation. Compared to ICPC problems, data structures are more emphasized in IOI-style contests[27].

Combining Algorithm Designing Techniques. There are other techniques like divide and conquer, greedy... they are all very important and frequently appeared in ICPC contests. Harder problems often require a combination of several algorithm designing techniques (e.g binary search is easily combined with other algorithms). Readers are recommended to read classic algorithm books [14], [15] and [12] for these algorithm designing techniques with applica-

[6] though some mathematical modelings and deductions could be done in prior to this

[7] Though some harder problems involves non-trivial optimizations involving convexity or concavity of some function, which requires insight of the problem as well as some math work

tions. But again, the most effective way to gain experience is to try a lot of different problems.

3.3 Graph Theory

Graph theory is a big source of ICPC problems. Very few contests do not have a problem involving problems. Recent World Finals often have graph theory problems that distinguish outstanding teams from ordinary ones. However, understanding graphs is not always easy for beginners. If you're new to graph theory, make sure you're familiar with popular concepts and terms that are extensively used. Examples of graphs in different areas help a lot here. Readers are encouraged to read Chapter 9 of [3], which explained the most widely used terms, graph representations, BFS and DFS, and topological sorting, with a few problems left as exercises. Before further study, it's a must to practice problems that needs mathematical modeling in terms of graph theory, and try out different representations of the same graph.

Classic Problems. Some problems in graph theory are so classic that everyone should be able to solve them. For example, problems related to connectivity (including strongly connectivity and bi-connectivity), shortest paths (Dijkstra, Bellman-Ford or Floyd-Warshall etc), spanning trees (Prim or Kruskal), eulerian paths and circuits, matchings in bipartite graphs (Hungary, Hopcoft-Karp and Kuhn-Munkres) and network flows (maximum s-t flows and mincost flows). Readers are encouraged to read a comprehensive but easy-to-understand book on graph theory[5]. However, many graph problems demand good mathematical modeling ability, which needs practicing. Some algorithms, though generally believed easy, need further investigating. For example, Dijkstra's algorithm can also handle minimal bottleneck path problems, maximal flow algorithms can also be used to calculate minimal cut. To my experience, people can be easily confused by some concepts, theorems and algorithms in network flow. The best thing to make it clear is through experiments on different contest problems that can force the programmer to correctly consider every detail.

More Classic Problems. Here is another list of problems that appeared in some contests (though very infrequently): k-th shortest path between two nodes, k-th smallest spanning tree, minimum degree-constraint spanning tree, minimum arborescence, minimum mean-weight cycle, minimum path cover in both unweighted and weighted directed graph, stable marriage problem, eulerian circuits in mixed graph, Chinese postman problem, 2-SAT problem, maximal weight closure, maximal density subgraph, graph recognition (e.g chordal graph, interval graph, co-graph, line graph...), and more. For in-depth references, you may want to take a look at [13], though it looks too difficult to be practical for ICPC.

Interestingly, many concepts from other algorithms closely relates to graph theory. For example, many dynamic programming turns into shortest-path problems when losing the topology, and single-agent search algorithms for path-finding problems like A* and IDA* could be stated in pure graph theory terminology. There are also some "ad-hoc" graph-theoretic problems that mainly requires hard thinking instead of mathematical modeling (although some of the algorithms discussed above are also required). You can find many of these problems in European regionals.

3.4 Geometry

Problems involving geometry are usually difficult, complex or very easy to make mistakes. There are a lot of geometry algorithms that are almost impossible to code during a contest, but there are still a lot problems that can be tackled. Elementary geometry, analytic geometry and trigonometry would be sufficient to solve these problems in an easy and elegant way, so people may want to print and bring related math formulae and/or pre-written routines. In fact, geometry routines composed a large part of printed materials for many experienced teams. For some more algorithm-oriented geometry problems, fundamental concepts and algorithms from computational geometry are required.

people should at least know how to find the convex hull of points on a plane, how to compute the area of a simple polygon, and decide if a point is inside a given polygon, outside of it, or on its boundary. But when you're starting to seek for more challenges, there are just too many classic problems to solve[8]. Some of them even involve 3D geometry, which itself is much more complex compared to 2D geometry.

Some classic contents that are involved in contest problems include: convex hulls in higher dimension, voronoi diagram, arrangment of lines, data structures for geometry, path planning, geometry optimization and classic algorithm design techniques like scanning, divide-and-conquer, deterministic and randomized incremental etc.

People are encouraged to read [4] and [6], but be warned that inexperienced programmers will soon find most of the algorithms in it extremely hard to implement correctly. Actually, only very few super-tough problems requires complex classic geometry algorithms. Here is another book I would like to recommend: [16], which is a good source of handy snippets. Of course, they should be rewritten by yourself to maximize its usefulness, as I explained above.

3.5 Last Words

We've already mentioned quite a lot of classic problems, mainly in graph theory and geometry, but there are more.

Probably the most famous ones that are not mentioned are Range Minimal Query (RMQ) and Lowest Common Ancestor (LCA). They are extremely frequently used in problems about sequences and trees. Another big class of problems not mentioned is string-theoretical algorithms. They are probably a lot more difficult than many people thought. A serious training should cover pattern-matching algorithms like Knuth-Morris-Pratt algorithm (KMP) for single-pattern

[8] In fact, there are still quite a lot of non-modified version of classic problems appeared in real contests

case and Aho-Corasick for multi-pattern case. Many string-theoretical problems requires deep understanding of these two algorithms in terms of string-matching automata. Classic divide-and-conquer algorithms for finding maximal palindromic substring or repetitions are also recommended to study. Suffix tree, or its good substituent, suffix array, both concept and related algorithms (e.g. construction and height array computation) are essential for many problems of this kind. People are encouraged to read[11].

Though there are still something missing[9], but mastering everything mentioned is already a very challenging task. Fortunately, there are some good competition-oriented books for beginners [2] and [3]. It's also a good idea to try USACO Training Gateway[26], which is a step-by-step training system. Early chapters take you through most of the basic but practical stuffs, while later chapters involve more complex algorithms and problems. Though the problems are in IOI-style, the algorithm nature is the same as ICPC problems.

For advanced topics, though, people need to spend more time and take more practice. Though I listed quite a few references for these topics, all of them are not for programming contests. ICPC contestants may find it more practical to read [1][10], as it's specially written for programming contests[11]. It is especially suitable for in-depth training, as there are hundreds of real problems of a more difficult level.

Some books [1] [2] [3] contain a list of problems in category, which is especially useful when training a particular algorithm like dynamic programming or network flows. Websites mentioned above also contained similar lists, which can also be utilized.

4 Planning Your Training

Always try to train together, once you have a team. Try to know better about each other and become good friends[12]. There are non-technical aspects that have great influence on a team's performance, but they're beyond the scope of this paper, hence will not be discussed further here.

4.1 Know Each Other Better

When team members are not so familiar with each other, they may try to solve the same set of problems individually, in a fixed time. After comparing their results, then some discussions, they'll gradually know the merits and defects of everyone. There are more types of specialization than many teams are aware of, which are listed below (still incomplete!). Some of them are hard to train, but it's helpful to know who is good at what, according to the list.

[9] like linear programming (LP), but it is rarely required in real contests
[10] Currently only Chinese edition available
[11] Though its main audience include potential IOI competitors, not only ICPC competitors
[12] Have meals together, read each others' blogs etc

Problem Reading. Any mistake in problem reading can be hazardous. Some people can quickly grasp the main idea of a problem, skipping long and useless background information or stories; some other people can find hidden hints, potential ambiguities, traps or possible misunderstandings in the problem description more easily.

Algorithm Designing. In theory, algorithm designing is the heart of ICPC. For simple problems, the algorithm might be straightforward, but complex problems usually require an intensive discussion. Some people like to do mathematical modeling or problem transformation, before actually designing the algorithm; some people like to propose conjectures that greatly simplifies the problem, once proved; some people like to provide ideas or directions without knowing whether they could actually work; some people like to follow a specific idea and think in more depth and detail (e.g deriving math formulae); some people like to simplify an existing algorithm and make it easier to implement; some people like to optimize an existing algorithm (possibly with low-level tricks) to make it pass the jury's tests; some people are even experts of designing randomized algorithms that are virtually incorrect but very hard to beat, at least by the jury. Having people especially trained in most (or even all) of these aspects mentioned, the team could become very strong.

Coding and Bug-finding. In most cases, a program should always be written by a single person, but it does not mean no one needs to read others' code. At times, it is even easier to find silly bugs written by another person. Thus, the ability to read others' code should be taken into account. [25] is a great place to practice writing and reading algorithmic codes.

Test-setting. Unlike many teams do, preparing test cases should often by done in parallel to coding or even algorithm designing. When reading the problem, some people create simple but typical test data (with answers evaluated by hands) for all-purpose use (discussing algorithms, testing and debugging etc), as well as tricky data that is very likely present in the judge data, independent of the algorithm or implementation. When discussing algorithms with other people, some programmers can think of counterexamples for incorrect algorithms more easily. What's more, even after another person already started coding, it's usually a good idea to have a teammate to analyze the error-prone parts of the algorithm being used, and create simple but special-purpose white-box test data for testing and debugging. In my opinion, a good collaboration in coding, debugging and testing greatly enhances a team's performance.

4.2 Individual Training

Roughly speaking, there are two types of training: individual training and team training. The aim of individual training is to improve individual skills, while team training concentrates on collaboration and contest strategies.

We've already discussed a lot about code practicing and background learning, why we mention it again? The answer is: though everyone in a team should try to learn more algorithms and solve more problems, there should be some specialization. It could be according to abilities defined above, or more frequently, be according to a problem category. For example, many teams have a graph theory expert, every graph-theoretic problems go to him. The problem is: do other people need to know something about graph theory? If yes, how much and how deep?

My suggestions is: they should at least have a solid theoretical background in graph theory: the terminology, theorems, classic algorithms and problems. The main difference between them and the expert is familiarity, experience, sense and creativity. If the other two are novices in graph theory, no discussion is possible for graph theoretic problems, which is very risky. What's more, if every one can only solve one or two kinds of problems, it's almost impossible to solve complex problems that require a combination of several different kinds of knowledge.

Keeping in mind the specialization in your team, the individual training will be more effective.

4.3 Team Training

The best way to improve a team's collaboration is to participate in online contests, especially the ones with some of the top teams in the world.

Rule-based Strategy. Most teams would like to use a relatively fixed strategy in every contest. [32] defines three example strategies: Simple, Terminal Man(TM) and Think Tank(TT). In recent years, people tend to use a mixed strategy: there is no terminal man, nor a thinking tank. Instead, a team should have set of simple rules with priorities for themselves. For example, a typical rule is to request a printed copy of code after each submission, and have another person to read the code if it's soon rejected. Another possible rule is that no problem can be left unread after the first hour. Try to discuss the rules after each team training, possibly adding some new ones that you feel necessary during this training, and removing ones that you feel unnecessary, or conflict to the ones you've just added. It's also worth trying some experimental rules to see if they're good, but again, participate in online contests as many times as you can, if order to find a really fine-tuned strategy for your team.

Situation Judging. Judging the current situation is arguably one of the most difficult but crucial task in collaboration during real contests. The judgement relies on the problem analysis(ideas and algorithms discussed so far, rejected codes etc), the balloons on site, and possibly the board if available. It's usually helpful to draw a table containing the current status for each problem, and update it each time a person reads a problem, finds an algorithm, starts writing the code or creates test data. An important part of situation judgement is setting current short-term and long-term goals according to every team's performance, since they greatly affect decision making.

Decision Making. When the situation is judged, the team leader can make decisions[13]. The decision should be in accord to the current goal, considering possible risks. If necessary, the team leader can give a whole list of candidate

[13] It's usually a good idea for a team to have a team leader who is good at making decisions

actions to take next, and have a short discussion with other two members. But in most circumstances, the team leader evaluates pros and cons of each possible action and makes the final decision himself. The decision is often about three factors: time, terminal and team members. That is, the rough schedule for the rest of the contest, who should use the computer for what problem next, and what other two people are expected to do at the same time. Be warned that the situation changes constantly during the contest, so the team leader should be sensitive enough. Also, it's very important to have all three members know the current situation and the team leader's decision clearly, otherwise the decision makes no sense.

One last word: after the contest is over, don't forget to spend some time to sit down and discuss what have we learnt from the contest, about team strategy, situation judgement and decision making. This is a lot more important than your final ranking in the contest.

4.4 Preparing your Contest Materials

In real ICPC contests, people are allowed to bring a limited amount of printed materials. All three people should be familiar with these materials, and it's recommended that it is prepared by all three people rather than a single person.

Snippets. When solving more and more problems, many people realized that they're writing similar codes quite frequently. It's a good idea to collect some snippets (short codes, not as long as the codes in the standard code library, discussed next). The tasks that the snippets do should be easy, and the snippets themselves should be very short, handy to use, easy to understand, self-explanatory and, of course, bug-free. Many people use snippets to illustrate the use of some programming features and standard libraries (e.g STL, JCF). It's usually impossible to find your favorite code tricks in any books, manuals... etc if you happen to forget them. Experienced programmers may find it unnecessary, but new programmers usually benefit from it quite well.

Customized Handbook. By handbook we mean concepts, formulae, facts, constants... It does not have to be complete, but should be helpful. This is required because sometimes it's clearer to print original formulae instead of their corresponding codes, in order to do mathematical derivations. Most frequently used ones are geometry, linear algebra, trigonometry, numeric analysis and other important formulae published in most mathematical handbooks. I personally prefer to place short but hard-to-understand or math-oriented codes just after their relevant mathematical concepts and formulae, as we need to know the preconditions, arguments and return values of the routines. Short comments in the code may be inadequate.

Standard Code Library. Codes in the library, unlike snippets, are algorithmic. The main purpose is to provide correct, short and fast implementation of some classical algorithms. Here the rule of thumbs is: everything should be written by yourself, and thoroughly tested. If you borrow codes from someone else, you're probably abusing them without actually understanding them. Some harder problems requires slight modifications on the classical algorithms, but even smallest changes can be difficult to make if the codes are not yours. I personally suggest that brief introductions to the problem, algorithm with examples (even with test data) should be given together with the codes. In practice, these words might also provide some hints to algorithm design. Though standard code library may be useful at times, don't rely on it too much. Try to think of them as code examples to remind you of algorithm outlines and details, not black boxes that you use without looking at.

Reminders. This is one of my favorite kind of material, but it is probably prepared by the least number of people. For example, examining a short list of most common mistakes [33] might be helpful in many situations. The famous "How to Solve it"[17] might also be helpful if you're facing hard problems. Other stuffs that you believe is important but easy to forget can also go here. If you love to write a short summary after each training session, it's easy to collect useful reminders from these summaries.

5 Summary

In this paper, we discussed several aspects of training an ICPC team, for both beginners and experts. Many techniques and suggestions can apply to other programming contests, too. However, this paper is by no means a complete guide. As stated above, some non-technical factors can be equally important, and, this paper does not give any concrete example of training schedule or a list of good problems and contests for training, due to the limited space. Nevertheless, students and coaches should still be able to benefit from the principles and methods discussed above, especially when they're adapted to the reality.

6 Biography

Rujia Liu (rujia.liu@gmail.com) participated in the 2001-2002 ACM/ICPC, winning the champion of Shanghai regional contest in 2001, and then a silver medal (the 4th place) in the world finals, Hawaii in 2002. He is also a coach of IOI[27] China national training team (a team consisting of 20 students from which the final national team is selected) since 2002. All the world finals contestants in Tsinghua University, who achieved one gold medal (the 2nd place in 2007) and two silver medals (the 4th and 5th places in 2002 and 2003) have been in the national training team when in high school. He has also arranged two sets of regional contest problems for 2006 Xi'an and 2007 Beijing, authoring 18 out of all 20 problems in them[14]. He also creates problems for online contests in UVa Online Judge[18] and other programming contests.

7 Acknowledgements

Rujia Liu is grateful to Prof. Miguel A. Revilla and Prof. Bill Booth for providing the chance to write a paper for CII. He would also thank Gelin Zhou (Tsinghua

[14] Other two problems are from Shahrair Manzoor, one for each problemset

University), Bin Wu (Wuhan University), Naiyan Wang (Zhejiang University), for suggestions and discussions on this paper.

References

1. Rujia Liu, Liang Huang: The Art of Algorithms and Programming Contests. Tsinghua Press, China, 2004
2. Ahmed Shamsul Arefin: Art of Programming Contest (Special Edition for UVa), second edition. Gyankosh Prokashoni, Bangladesh, 2006
3. Steven S. Skiena, Miguel A. Revilla: Programming Challenges - The Programming Contest Training Manual. Springer-Verlag, 2003
4. J.O'Rourke: Computational Geometry in C. Cambridge University Press, New York, second edition, 2000.
5. R.Sedgewick: Algorithms in C++: Graph Algorithms. Addison-Wesley, third edition, 2001.
6. M de Berg, M.van Kreveld, M.Overmars, and O.schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag, Berlin, second edition, 2000.
7. Elwyn Berlekamp, John H. Conway, and Richard Guy: Winning Ways for Your Mathematical Plays, Massachusetts: AK Peters, second edition, 2001.
8. John Conway: On Numbers and Games, second edition, Massachusets: AK Peters, 2000.
9. Thomas Ferguson: Impartial Combinatorial Games Notes. `http://www.math.ucla.edu/tom/Game_Theory/comb.pdf`
10. R.Graham, D.Knuth, and O.Patashnik: Concrete Mathematics. Addison-Wesley, Reading MA, 1989.
11. D.Gusfield: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997
12. S.Skiena: The Algorithm Design Manual. Springer-Verlag, New York, 1997
13. R.K.Ahuja, T.L.Magnanti and J.B.Orlin: Network Flows: Theory, Algorithms and Applications. New Jersey: Prentice-Hall, 1993
14. Thomas H.Cormen, Charles E.Leiseison, Ronald L.Rivest and Clifford Stein. Introduction to Algorithm, second edition, MIT Press and McGraw-Hill, 2001.
15. Jon Kleinberg, Eva Tardos: Algorithm Design, Addison-Wesley, 2005.
16. Philip J.Schneider, David H.Eberly: Geometric tools for computer graphics, The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, Morgan Kaufmann Publishers, San Francisco, 2003
17. G.Polya: How to Solve It. http://www.math.utah.edu/ pa/math/polya.html
18. UVa Online Judge. `http://acm.uva.es/problems`
19. The 2000's ACM-ICPC Live Archive Around the World. `http://acmicpc-live-archive.uva.es/nuevoportal/`
20. POJ Online Judge. `http://acm.pku.edu.cn/JudgeOnline`
21. ZOJ Online Judge. `http://acm.zju.edu.cn`
22. Timus Online Judge. `http://acm.timus.ru`
23. SGU Online Contester. `http://acm.sgu.ru`
24. Sphere Online Judge. `http://www.spoj.pl`
25. TopCoder Competitions. `http://www.topcoder.com/tc`.
26. Rob Kolstad, USACO Training Gateway. `http://ace.delos.com`
27. International Olympiad in Informatics official website. `http://www.ioinformatics.org/`

28. “World of Seven”, METHODS TO SOLVE VALLADOLID ONLINE JUDGE PROBLEMS, Steven Halim, National University of Singapore. `http://www.comp.nus.edu.sg/~stevenha/`
29. Felix Halim .NET - Hunting UVA Problems!, Felix Halim. `http://felix-halim.net/uva/hunting/`
30. ACMSolver.org, ACM/ICPC Programming Contest Tutorial Website for Valladolid OJ by Ahmed Shamsul Arefin. `http://www.acmsolver.org`
31. ACMBeginner.tk, ACM Valladolid Online Judge (OJ) Tools, Tips and Tutorial by M H Rasel. `http://www.acmbeginner.tk`
32. Ernst,F., J.Moelands, and S.Pieterse: Teamwork in Programming Contests: 3 * 1 = 4, ACM Crossroads Student Magazine, `http://www.acm.org/crossroads/xrds3-2/progcon.html`
33. Shahriar Manzoor: Common Mistakes in Online and Real-time Contests. ACM Crossroads Student Magazine. `http://www.acm.org/crossroads/xrds7-5/contests.html`