A Programming Contest Strategy Guide

Aaron Bloomfield
University of Virginia
aaron@virginia.edu

Borja Sotomayor
University of Chicago
borja@cs.uchicago.edu

ABSTRACT

The ACM's International Collegiate Programming Contest (ICPC) is the world's oldest and largest programming contest. Although students can benefit both pedagogically and professionally from participating in this contest, participation in North America is far smaller than in the rest of the world, which we partially attribute to the perceived low payoff of participating in ICPC. We discuss the pedagogical benefits of participation in ICPC, which include higher student enthusiasm for computer science studies and better career prospects post-graduation, and present a set of cohesive strategies aimed at increasing involvement and success within the ICPC. For aspiring coaches and contestants, we also provide links and references for further study.

Categories and Subject Descriptors

K.3.m [**Computers and Education**]: Miscellaneous

General Terms

Algorithms, Measurement

Keywords

Programming contest, ICPC, Strategy

1. INTRODUCTION

There are many collegiate programming contests, but arguably the most prestigious is the ACM's International Collegiate Programming Contest[1] (ICPC), the largest and oldest programming contest in the world. The contest is open to undergraduate students and first year graduate students. Students form teams of three, coached by a faculty member – or someone designated by a faculty member – and compete in regional contests ("regionals") in the fall. In the 2014 regionals, the latest for which data is available as of publication time, there were 38,160 students from 2,534 institutions in 101 different countries. The top 128 teams from the regionals advance to the world finals, held most recently in Marrakesh, Morocco, in May 2015.

Although participating in the ICPC has the potential for winning large cash prizes and "bragging rights", it also offers students a number of pedagogical advantages, such as solidifying their understanding of core computer science concepts, often going beyond what is covered in a typical computer science major, as well as building many cross-cutting skills, such as effective teamwork.

However, despite the number of pedagogical and professional benefits, ICPC is not as popular or well known within ICPC's North American "super-region"[2] as in the rest of the world, where it is regarded as a highly prestigious competition. In fact, participation in North America lags behind that of the other ICPC super-regions, particularly Europe and Asia, and no North American team has won the world finals since 1999. In fact, since 2000, only teams from Russia, China, and Poland teams have won the world finals.

In this paper, we describe the pedagogical benefits of participating in ICPC and present a number of training and contest day strategies that we have used successfully with our own teams[3], and which we have seen consistently used by schools that regularly qualify for the world finals.

We are certainly not the first to present an article on programming contest strategy. Indeed, articles on this topic were appearing as early as the 1980's by Myers and Null [7]. Two valuable strategy guides are by Trotman and Handley [10], and by Khera, et. al. [4]. Manzoor provides a description focused on the common mistakes made during contests [5]. We have synthesized the wisdom of these articles, combined it with our successful experience in the last decade of competitions, and updated the strategies with recent advancements.

Our contributions to the literature are two-fold. Firstly, we have synthesized much of the extant literature on contest strategy, and presented it in a single, cohesive whole. Secondly, we present strategies for the contest itself, including training activities; much of the extant literature focuses on describing how to solve certain problems, rather than training and contest day strategies.

[1] `http://www.acmicpc.org/`

SIGCSE '16 March 02-05, 2016, Memphis, TN, USA

http://dx.doi.org/10.1145/2839509.2844632

[2] ICPC defines this super-region as encompassing all the regional contests in the US and Canada, but not Mexico nor the Caribbean countries

[3] The authors have had considerable success over the last decade with coaching our respective teams. In the last eight years, the two authors have led their respective teams to the world finals in 14 of the 16 possible berths.

2. CONTEST STRUCTURE

An ICPC-style programming contest is a five hour event in which teams of three students attempt to solve up to 12 provided programming problems using one of several allowed languages. Since the allowed languages at the world finals are Java, C, and C++, most of the North American regions allow only those three languages, although a few allow additional languages[4]. Each team has only a single computer with no Internet access. During the contest, teams submit their solutions using a contest submission system, and are then ranked according to how many problems they can solve and how quickly they can solve them (the ranking system is further explained below). During the contest, the solutions submitted by the teams are evaluated by several judges, with the help of an automated contest control system.

2.1 Contest Environment

The exact software environment on a team's computer varies by region, but is always known in advance. In general, contestants will have access to the compilers and interpreters for the allowed languages, and a number of text editors and debuggers, including IDEs such as Eclipse.

Since contestants have no access to the Internet, a selected number of electronic references are loaded onto the computers, typically language references (Java SDK API, a C++ STL reference, etc.), references for using editors, and system-specific references (how to print, etc.). On UNIX systems, contestants also have access to system documentation, including `man` pages.

Students are allowed to bring printed reference materials to the contest, with each region having its own rules on how much and what type of reference materials are admissible. Nearly all North American regions allow effectively unlimited amounts of reference material, while at the world finals each team is limited to a 25 page reference binder.

2.2 Problems

Problems are designed to test not just the team's programming abilities, but also their command of various computer science topics. A typical contest will present between 8 and 12 problems of varying difficulty.

The simpler problems usually require using, but not implementing, basic data structures covered in most introductory courses. The medium difficulty problems often cover the material taught in an upper division Algorithms course (dynamic programming, divide-and-conquer, greedy algorithms, graph algorithms, etc.). The most difficult problems will often combine multiple concepts from an Algorithms course – such as a max flow problem that requires dynamic programming and search space culling – or will require topics covered in advanced mathematics courses.

Steven Skiena's *Programming Challenges* book [9] provides a very detailed description of the typical types of problems that are often encountered in a regional contest, and describes how to solve these problems. Similarly, Steven Halim's *Competitive Programming* book [3] is similar in structure, but also provides more details on some topics and numerous solutions.

[4]Currently, the Greater NY region allows Ada, and the Southeast region allows Ada, C#, Fortran >= 90, Haskell, Lua, Pascal, Python 2&3, and Scala. In addition, many regions, not enumerated here, are now allowing Python.

2.3 Judgement of Solutions

Problems in programming contests are designed to accept input as plain text from standard input in a specific format, and similarly for its output (via standard output). For example, a simple problem could involve reading the input one line at a time, with each line containing a list of integers, and outputting the sum of the integers:

Input	Output
10 20 30 40	100
5 10 5	20
10 0 7	17
0	

A line of just '0' indicates the end of the input.

For each problem, teams are typically provided with simple test cases like the one above, mainly to provide an example the input/output format of the problem. Judges will use a separate collections of test cases (unknown to contestants), covering corner cases, largest and smallest inputs, etc., to test the solutions submitted by teams. The process of submitting solutions and judging them with the judges' test cases is handled by a "contest control system" like PC^2[5], KATTIS[6], or DOMjudge[7].

A solution is considered correct if and only if, given the judges' test cases, it produces the correct output within a given time limit. This time limit often precludes grossly inefficient solutions, such as brute force solutions. Any other outcome (such as producing incorrect output, raising an exception or a segmentation fault, etc.) results in the solution being marked as incorrect.

2.4 Scoring

During a contest, teams are ranked on a scoreboard based on a scoring metric that starts by looking at the number of problems solved: if team A solves more problems than team B, then team A will place higher than team B. When two teams solve the same number of problems, the tie-breaker is the total time taken to solve the problems. This is the sum, in whole minutes, of the time elapsed from the beginning of the contest until when each problem was solved successfully. So if a team solves problems at times 35, 90, and 140 (all in minutes; seconds are not counted), then their total time is the sum of those values, or 265. Any incorrect submission would additionally result in a penalty of 20 minutes. Note that the time (including penalties) is *only* counted if a problem is solved correctly; any problem not solved by the end of the contest does not contribute to the total time. ICPC defines an additional (and rarely invoked) tie-breaking rule if two teams have the same number of problems solved *and* have the same total time.

Due to how the scoring works, if a team was going to successfully solve n problems, then it will be strictly better to solve the easier ones (i.e., those that can be solved in less time) first, as that will lower the second metric of time.

Contest scoreboards can reveal trends and statistics that can be used to further analyze a team's performance. Manzoor provides a highly detailed description of analyses of both online submission systems and programming contest results [6].

[5]`http://www.ecs.csus.edu/pc2/`
[6]`https://kth.kattis.scrool.se/`
[7]`http://domjudge.sourceforge.net/`

3. PEDAGOGICAL BENEFITS

Although both the regionals and the world finals award prizes to the top teams (typically no cash prize for a regional, but with a grand prize of over $10,000 for the top team in the world finals), we have found that the vast majority of coaches and students are not motivated by these prizes. Instead, there are a number of pedagogical and professional benefits to participating in ICPC.

One important benefit is that the contest not only reinforces many concepts and skills that students see in computer science courses, but also builds a number of valuable cross-cutting skills. In preparation for a contest, ICPC contestants must hone their programming skills and have a solid grasp of data structures and algorithms, sometimes at a graduate level. Students are also exposed to many topics, such as advanced and esoteric algorithms, that are not typically covered in a classroom setting, requiring them to be independent learners who can research new topics and solve problems without the guidance of an instructor.

Furthermore, training for ICPC builds teamwork skills both inside the individual teams and between the teams. Inside a team, the three contestants must be able to collaborate on solving challenging problems. As we will explain later, team dynamic is as important as raw talent in the success of a team. Between the teams, students typically approach training as a cooperative competition: even though they must compete with each other during a contest (or a practice contest), they cooperate in all other aspects of training. For example, after a practice contest, a team that solves a problem that no one else solves is usually happy to explain their solution to the other teams. We have never witnessed ICPC teams hoarding resources or problem solutions to gain a competitive advantage over other teams in the same institution.

Participation in ICPC can also benefit the students' job prospects. Many current and former ICPC participants have told us that the skills they acquired through ICPC have been invaluable when applying for technical positions, specially those involving a "technical interview", which often requires candidates to solve complex programming and algorithmic problems on the spot.

4. TRAINING ACTIVITIES

Not surprisingly, the biggest factor that will help a team succeed is training. However, newcomers to ICPC tend to assume that working through programming problems is sufficient to prepare for the competition. Although that will certainly help contestants familiarize themselves with the style of problems found in ICPC, other forms of individual training are also important, along with training in teams.

There are many facets to selecting students who will work together well on a team; we discuss those strategies in Section 5.2, and focus on training strategies in this section.

4.1 Individual Training

Working through problems.

The best way of working through problems individually is to replicate, as much as is possible, the conditions of a programming contest, especially the automatic judging of solutions with test cases that the student is not privy to.

Fortunately, there are a number of online sites that provide exactly such a service. These include the UVa Online Judge[8] (from the University of Valladolid, Spain), with hundreds of regional problems; the Live Archive[9], a revamp of the previous site with many new problems; Open Kattis[10], the public version of the Kattis contest management system, with problems from contest all over the world; the companion site to the Programming Challenges[11] book (but note that this site cannot properly judge some problems); and TopCoder[12] and CodeForces[13], which have become popular lately as a contest training method.

As problems are solved, copies should be kept of the correct solutions, as this can help future students who are stuck on that problem. This is ideally done by the coach.

College courses.

Students participating in programming competitions need a solid understanding of many core computer science concepts, many of which are often easier to learn through collegiate courses than through self-study.

Contestants should plan to take some theory-based courses in computer science curricula, such as Algorithms and Theory of Computation. While both are of use, the former is somewhat more useful than the latter, largely because the material in a typical Algorithms course (dynamic programming, searches, etc.) is more viable to be made into a contest problem than the material in a typical Theory of Computation course (parsing, formal languages, etc.). If the opportunity to take both arises, then that is certainly the ideal. Computational geometry, combinatorics, and graph theory courses – sometimes taught by the mathematics department – are of use as well, as some problems will focus on those topics.

Many instructors use the ICPC-style of writing programs successfully in both Algorithms and Theory courses. One of the authors has done so in an Algorithms course at the University of Virginia, and similar concepts have been applied to Theory of Computation classes [8].

4.2 Team Sessions

Running practice contests.

Essential training includes running contests that replicate the conditions of real contests. We describe below the ways in which these conditions can be replicated.

Using real problem sets: When running a practice contest, there is no need to prepare problems specifically for the practice. It is much more useful to use problems from actual regional contests, especially those from regions where the difficulty of the problems is similar to one's home region. Most North American regional problem sets are good when practicing for a North American regional contest, while problem sets from other regions in the world should be used with care, as their difficulty tends to be closer to that of the world finals problems.

Most regionals provide their problem set and judging data online. With this data, a contest control system can be con-

[8] http://uva.onlinejudge.org
[9] http://livearchive.onlinejudge.org
[10] https://open.kattis.com
[11] http://programming-challenges.com
[12] http://topcoder.com
[13] http://codeforces.com

figured to judge the responses just as was done on the original contest. Some also provide a detailed scoreboard (i.e., not just the ranking of teams, but the exact problems solved and their times and penalties) that can be useful for post-contest analyses, and can show the team how they would have placed if they were competing in that regional.

Running five hour contests: Although five hours for a practice may seem excessive, it is important to avoid the temptation to run shorter practices. Five hour practices allow students to experience the stamina necessary to focus for that length of time. They also allow students to work through the four phases of a programming contest, described below.

Not all practices will be five hour practices. Indeed, two hour practices in the evenings are a popular option due to busy schedules. However, having as many five hour practices as possible is essential to becoming a competitive team.

Replicating the contest computer environment: As described earlier, a real ICPC contest will provide each team with a single computer with no Internet access and limited amounts of documentation. It is important to enforce, as much as possible, the same constraints they would encounter at a real contest, since students typically find it hard to "unplug" during a practice contest, or to limit themselves to using just a single computer. Using the same operating system, compilers, tools, submission system, and reference material that will be provided during the contest is also helpful.

Run contests with multiple teams: In our experience, practice contests with only one team are much less effective. When only one team is "competing", the temptation to get distracted or simply give up on the practice is much higher. It also means the team does not have access to a realistic scoreboard during the practice, which prevents them from making strategic decisions based on the state of the scoreboard. For example, if there is a problem that most of the other teams have solved, then it is likely an easier problem, and one that should be worked on earlier.

On the other hand, when multiple teams participate in a practice contest, there will be some friendly competition amongst them to see which team can solve more problems, and it provides a more realistic scoreboard. It also allows those teams to have a more productive discussion after the contest (see below), and share the insights that each team had on the various problems.

Problem discussion sessions.

Once a contest is done, the teams will have likely all solved different sets of problems. Some teams will have been unable to complete a problem that others were able to solve. If a team never understands how a problem is solved – either because they were unable to solve it or because it was not explained to them – then much pedagogical potential is lost.

Thus, having a discussion session after the contest is essential. There is often resistance, as students feel mentally tired after spending five hours working on the problems. Nonetheless, our students tend to feel that practices are not as useful unless they are followed by a problem discussion session.

During this discussion session, teams that solved a problem should explain their solution to the teams that did not solve it. This allows the teams who did not solve it to know if it was because of an incorrect theoretical approach, if they ran into implementation issues, or if they neglected to consider corner cases and boundary cases. These discussions also allow teams to understand how one would solve the problems that they were unable to attempt in the practice contest.

Reviewing official solutions.

Besides discussing problems in the context of a problem discussion session, teams should also review the official solutions provided by the authors of the problem set used in the practice contest. Most contests provide such solutions, and we tend to stay away from problem sets with no official solutions: if no team was able to solve a problem in a practice contest, it makes it harder to discuss its solution (or places the onus on the coach to come up with a solution).

4.3 Preparing the Reference Documentation

Before the actual contest, the team will need to assemble the reference documentation that they will bring with them to that contest. In regions with no limits on the amount of documentation, many teams invariably show up with backpacks filled to the brim with Algorithms books, programming books, and printed solutions to problems. However, quality is more important than quantity when preparing the reference material. In particular, Algorithms books are only useful if the students are *already* familiar with their contents (e.g., by having taken a course with that textbook). They should be regarded as a reference for algorithms that are too complex to memorize, and not as a source of new algorithms to be learned (and applied) during the contest itself.

Another useful strategy is for the team to write implementations of common algorithms (Dijkstra's shortest path, convex hull, etc.) in the language they are most familiar with, and bring printouts to the contest. This will allow them to quickly code those algorithms with a lower risk of making mistakes when reading them from pseudo-code in an Algorithms book or from notes. Similarly, solved problems from their practice sessions can also be valuable, as it is not uncommon to encounter problems that are very similar, or that take an existing problem and apply different constraints on it.

4.4 Continuity and Time Investment

While both individual and team training allow the students to gain experience and knowledge about the contest, that institutional knowledge is lost as soon as the students on that team graduate. Thus, it is crucial for there to be somebody who is actively involved in training the teams, allowing that accumulated knowledge to be retained across "generations" of teams. This is often a faculty member who acts as the coach, although this role is sometimes filled by community volunteers or graduate students, many of whom were involved with ICPC as undergraduates. Of all the competitive teams we are familiar with, and certainly all the teams we are familiar with that regularly qualify for the world finals, there is always someone to provide this form of continuity and to help coordinate and train the students.

Regardless of who fills the coach role, it will be a significant time investment. The most obvious time sink is the planning and running of the practice contests. In addition, retaining the knowledge in whatever format is desired (answer database, notes, etc.) will take time. We have found that an hour a week, in *addition* to the time coordinating the practices, is a reasonable estimate. Coordinating a practice usually takes one or two hours of work beyond the time

of the practice itself (selecting the questions, preparing the submission system, etc.). Five hour practices are ideal, although it is also possible to have shorter practices on a regular basis, and hold five hour practices occasionally. Participating in online contests, often set up by experienced ICPC coaches, can help lower the time commitment by an individual coach. Likewise, involving graduate students, past ICPC contestants, or co-coaches can dramatically decrease the time required by the coach.

5. CONTEST STRATEGIES

In addition to proper preparation, a team should be familiar with how to approach the contest itself.

5.1 The Phases of the Contest

During the contest, contestants must make the most out of the five hours they are allotted, which can be challenging given the constraints under which they will be solving the problems – most notably having just one computer for the entire team. We hold that a programming contest, whether it be a local practice, a regional contest, or the world finals, is split into four distinct phases. Our terminology for the phases is loosely based on the terminology for chess games.

The opening: As soon as the contest starts, each team must determine which are the easiest problems – or, more specifically, the problems that can be solved the quickest. Since the time that each solved problem accrues is counted from the start of the contest, it is imperative to solve these short and easy problems first since it will result in a lower "total time taken" in the team's score, which may end up being the tie-breaker if the team solves the same number of problems as another team. This phase should not take more than a few minutes, and lasts until just *one* easy problem is identified, as the rest of the easy problems can be identified in the next phase. Many teams will split the problem set into thirds, and each contestant will look through one third of the problems.

The early game: After identifying at least one of the easy problems, one or two contestants should focus their attention on this problem (one if the problem is trivial to implement, two if it requires some discussion), while the other team member(s) simultaneously work on identifying the rest of the easy problems. Once all the easy problems have been identified, this phase is also used to identify the difficulty and the category of the remaining problems, ideally with a rough order in which the remaining problems will be attempted.

In this phase, the teams will be solving problems fairly rapidly. Checking the scoreboard is particularly useful in this phase, as it will help the team determine whether they correctly identified the easy problems (if they see other teams are submitting them), or if they chose a deceptively easy problem (if no other teams are attempting them, or there are many attempts but no correct solutions).

This phase lasts until all the easy problems have been attempted and ideally solved. The length of this phase will vary depending on the contest and the experience level of the team, but tends to last half an hour in regionals with an experienced team and a few easy problems.

The mid game: Once the easy problems have been attempted, it is time to return to those that were not solved successfully or start on the medium difficulty problems. Like the previous phase, the length of this phase will vary by contest and team experience. Due to the increased difficulty of the problems, submitted runs do not occur as often. This phase typically takes the most time in the contest.

The end game: The last phase is *not* necessarily to work on the hard problems. Any problems that were too hard to solve prior to this point are not likely to be solved in the little time remaining. Instead, this phase, typically lasting the last hour or half hour, is to focus on the problems that have been attempted in the previous phases, but not solved successfully. That being said, if all the easy and medium problems have been solved, then of course the team should move on to the hard problems.

The difficulty of the regional contests will vary considerably, and thus the length of the phases will vary considerably as well. Some particularly telling examples were in the 2009 regionals. In the Mid-Atlantic regional contest, the problems were difficult enough that only 48 of the 161 teams (30%) solved any problems, and only two teams solved more than three problems. Conversely, the Greater New York regional contest that same year was the opposite; this region in general tends to have great teams and easier problems. Thus, also in 2009, 9 of the 53 teams (17%) solved all 9 problems, and the top team solved them all in 1 hour and 41 minutes! Thus, in that contest, who qualified for the world finals was not a matter of how many problems were solved, but how fast they were all solved.

5.2 Team Dynamics

The reason that the contests are team-based, and not individual, is because teamwork is essential to solving the harder problems. This is perhaps the most overlooked part, as most college students are accustomed to working on programming problems individually. Yet forming a good team dynamic, and being able to work together constructively on a problem, is essential to success in a programming contest.

This is not to say the individual prowess of each team member is not important; in fact, individual work is very effective in the early game, and many experienced teams can solve several problems based on individual work alone. However, this strategy breaks down once the team has to deal with more difficult problems. In fact, many teams incorrectly assume that, as they learn more about programming and computer science, they will simply be able to individually solve more problems faster, and neglect working on the team dynamics necessary to solve the harder problems.

There are several team dynamics that teams can find useful. One common strategy is to have one person thinking about the next problem (working out the algorithm, sketching out pseudo-code, etc.), while the other two team members pair program the current problem. Once the current problem is submitted, the team members rotate – the "thinker" for the next problem starts pair programming with one other member, and the remaining member starts contemplating the problem to work on next.

However, pair programming is a bit of a mixed bag. On one hand, it will make it easier to catch simple mistakes, which can help prevent incorrect submissions, but it also ties up two people with the programming aspect, and thus only allows two problems to be worked on at a given time. If there were no paired programming, then the team could be working on three problems in parallel. We feel that less experienced teams will benefit from a pair programming strategy. More experienced teams will often not need it for the

easier problems, although they should be able to tell when they do need to use it.

Intra-team relationships are critical as well. While the team need not all be best friends, they need to be friendly enough that they can (constructively) criticize each other's work without getting defensive. This exact issue has doomed some of our world finals teams.

5.3 Team Roles

There are many different proposals for how to split up the work that the team members do during a contest.

As mentioned above, some people propose that team members work on problems individually. This allows for maximum parallelization, as three problems can be worked on concurrently [1, 5]. However, this poses challenges for difficult problems, at which point the team members will have to work together.

Skiena and Revilla [9] define three roles: the coder, who is the primary programmer; the algorist, who is the one with the most theoretical knowledge, and the debugger, who helps determine why a program is not working correctly.

Lately, the most common strategy seems to be for teams to have members that take on *specialist* roles, where each member has experience with a particular set of problem types. When such a problem type comes up, that person will be the lead on solving that problem. Ernst discusses the specialist strategy, among other strategies [2].

We have adopted a hybrid of these different roles. In the early game, when the problems are easy, writing solutions in parallel is feasible, especially for experienced teams. Even then, however, pair programming can still help to reduce the number of errors in a program. As the problem difficulty increases, the teams often move more towards a specialist role. Training for the specialist is feasible, as each member can focus on older ICPC problems of those type(s). The specialist role setup effectively makes everybody an algorist, as defined by Skiena, but for a subset of the problem coverage. The dynamics of an individual team will determine who is the coder and debugger – it may well be the case that two or all three team members are equally proficient at those tasks.

5.4 Avoiding Tunnel Vision

One of the biggest problems that students face in the contest is the problem of tunnel vision. After several incorrect submissions for the same problem, students will often feel that they are really close to a correct answer, and that just a few more minutes of work and one more submission will achieve a correct answer. At some point, students need to move on to another problem, and come back to the one that is causing issues at a later point, hopefully with a fresher mind. Teams are typically allowed to print their code during a contest, and should do this to perform "offline debugging" of programs without utilizing the computer, so other team members can make progress on other problems.

We often tell our less experienced teams that after five unsuccessful submissions, then they must move on to a different problem. We do not feel that this rule is necessary for the more experienced teams, nor for any team in the world finals, as they should be experienced enough to avoid the tunnel vision trap on their own. However, for less experienced teams, this rule can help them avoid this trap, which has been the bane of many teams.

6. CONCLUSIONS

Our goal in this paper was to address the barriers limiting increased participation in ICPC, particularly barriers at individual institutions, who may struggle with how to start – and properly train – an ICPC program. We addressed the pedagogical benefits of an active ICPC program and presented concrete training activities and contest day strategies, in the hopes that this will help convince people to become more involved.

In our experience, participating in ICPC is a great experience for the students (and the coach!), as well as an enthusiasm generator, a way to encourage learning outside the classroom, and good preparation for going into the job market. Regardless of whether a team advances to the regionals or the world finals, ICPC can be a rewarding experience for all involved.

Acknowledgments

The authors would like to gratefully acknowledge the dedication and commitment of the ICPC staff and volunteers at the regional and world finals level. Participating in ICPC is a wonderful experience thanks to them.

We would also like to thank David Van Brackle and Godmar Back, who provided feedback on drafts of this article.

7. REFERENCES

[1] D. Chavey, T. L. Monrey, D. Van Brackle, and J. Werth. Preparing a team for the ACM scholastic programming contest (panel session). In *Proceedings of the 19th Annual Conference on Computer Science*, CSC '91, pages 701–, New York, NY, USA, 1991. ACM. Chairman-Bagert, Donald.

[2] F. Ernst, J. Moelands, and S. Pieterse. Programming contest strategies. *Crossroads*, 3(2):17–19, Nov. 1996.

[3] S. Halim and F. Halim. *Competitive Programming, 3rd Edition*. Lulu, 2013.

[4] V. Khera, O. Astrachan, and D. Kotz. The internet programming contest: a report and philosophy. In *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, SIGCSE '93, pages 48–52, New York, NY, USA, 1993. ACM.

[5] S. Manzoor. Common mistakes in online and real-time contests. *Crossroads*, 7(5):4–4, July 2001.

[6] S. Manzoor. Analyzing programming contest statistics. *Southeast University, Dhaka, Bangladesh. http://acm. uva. es/p/13_Manzoor_rev. pdf*, 2006.

[7] D. Myers and L. Null. Design and implementation of a programming contest for high school students. In *Proceedings of the seventeenth SIGCSE technical symposium on Computer science education*, SIGCSE '86, pages 307–312, New York, NY, USA, 1986. ACM.

[8] N. V. Shilov and K. Yi. Engaging students with theory through ACM collegiate programming contest. *Commun. ACM*, 45(9):98–101, Sept. 2002.

[9] S. Skiena and M. Revilla. *Programming Challenges: The Programming Contest Training Manual*. Texts in Computer Science. Springer, 2003.

[10] A. Trotman and C. Handley. Programming contest strategy. *Computers & Education*, 50(3):821 – 837, 2008.