Teamwork in Programming Contests: 3 * 1 = 4

by Fabian Ernst
Jeroen Moelands, and Seppo Pieterse

Introduction

Every year since 1977, the ACM has organized the ACM International Collegiate Programming Contest. This contest, which consists of a regional qualifying contest and the Finals, provides college students with the opportunity to demonstrate and sharpen their programming skills. During this contest, teams consisting of three students and one computer are to solve as many of the given problems as possible within 5 hours. The team with the most problems solved wins, where ``solved'' means producing the right outputs for a set of (secret) test inputs. Though the individual skills of the team members are important, in order to be a top team it is necessary to make use of synergy within the team. As participants in the 1995 Contest Finals (two of us also participated in the 1994 Finals), we have given a lot of thought to strategy and teamwork.

In this article we summarize our observations from various contests, and we hope that if you ever participate in this contest (or any other) that this information will be valuable to you.

The Basics: Practice, Practice, Practice!

Because of the fact that only one computer is available for three people, good teamwork is essential. However, to make full use of a strategy, it is also important that your individual skills are as honed as possible. You do not have to be a genius as practicing can take you quite far. In our philosophy, there are three factors crucial for being a good programming team:

- Knowledge of standard algorithms and the ability to find an appropriate algorithm for every problem in the set;
- Ability to code an algorithm into a working program; and
- Having a strategy of cooperation with your teammates.

  Team strategy will be the core discussion of this article. Nevertheless, there are some important considerations for improving your individual skills.

  After analyzing previous contest programming problems, we noticed that the same kind of problems occurred over and over again. They can be classified into five main categories:

  1. Search problems. These involve checking a large number of situations in order to find the best way or the number of ways in which something can be done. The difficulty is often the imposed execution time limit, so you should pay attention to the complexity of your algorithm.
  2. Graph problems. The problems have a special structure so they can be represented as a graph-theoretical problem for which standard algorithms are available.
  3. Geometrical problems. These involve geometrical shapes, lines, and angles.
  4. Trivial problems. The choice of appropriate algorithm is clear, but these usually take quite a long time to program carefully.
  5. Non-standard problems.

  For the first three categories, standard algorithms are well documented in the literature, and you should

program these algorithms beforehand and take the listings with you to the contest. In this way, you can avoid making the same (small) mistakes repeatedly and you can concentrate on the difficult aspects of the problem.

Another angle of practice is efficient programming. This does *not* mean type as fast as you can and subsequently spend a lot of time debugging. Instead, think carefully about the problem and all the cases which might occur. Then program your algorithm, and take the time to ensure that you get it right the first time with a minimum amount of debugging, since debugging usually takes a lot of valuable time.

To become a team, it is important that you play a lot of training contests under circumstances which are as close to the real contest as possible: Five hours, one computer, a new set of problems each time, and a jury to judge your programs.

Team Strategy: The Theory

When your algorithmic and programming skills have reached a level which you cannot improve any further, refining your team strategy will give you that extra edge you need to reach the top. We practiced programming contests with different team members and strategies for many years, and saw a lot of other teams do so too. From this we developed a theory about how an optimal team should behave during a contest. However, a refined strategy is not a must: The World Champions of 1995, Freiburg University, were a rookie team, and the winners of the 1994 Northwestern European Contest, Warsaw University, met only two weeks before that contest.

Why is team strategy important? There is only one computer, so it has to be shared. The problems have to be distributed in some way. Why not use the synergy that is always present within a team?

``Specialization'' is a good way of using the inherent synergy. If each team member is an expert for a certain category of problem, they will program this problem more robustly, and maybe more quickly than the other two team members. Specialization in another sense is also possible. Maybe one of the team is a great programmer but has poor analytical skills, while another member can choose and create algorithms but cannot write bug-free programs. Combining these skills will lead to bug-free solutions for difficult problems!

Another way to use synergy is to have two people analyze the problem set. Four eyes see more than two, so it is harder for a single person to misjudge the difficulty of a problem. Forming a think-tank in the early stages of a contest might help to choose the best problems from the set and find correct algorithms for them. However, once the algorithm is clear, more than one member working on a single program should be avoided.

It is our experience that the most efficient way to write a program is to write it alone. In that way you avoid communication overhead and the confusion caused by differing programming styles. These differences are unavoidable, though you should try to use the same style standards for function and variable names. In this way you can really make 3*1 equal to four!

Other Considerations

Since the contest final standings are based on the number of problems correctly solved, and (in the case of ties) on the sum of elapsed time for each problem, a team should adopt a strategy that maximizes the number of solved problems at the end of the five hours, and view the total elapsed time as a secondary objective. In every contest there are some teams in the ``top six'' after three hours, that are not even in

the ``top ten'' after the total five hours. The reverse also occurs. A long term strategy is therefore important: Try to optimize the 15 man hours and 5 hours of computer time, and do not worry about your total time or how quickly you solve the first two problems.

To optimize this scarce time, try to finish all problems that you start. A 99% solved problem gives you no points. Analyze the problem set carefully at the beginning (for example, by using a ``think-tank'' approach) to avoid spending more time than absolutely necessary on a problem that you will not finish anyway, and to avoid misjudging an easy problem as being too difficult. You need a good notion about the true difficulty of the various problems as this is the only way to be sure that you pick exactly those which you can finish within five hours.

Since you never have perfect information, you have to take risks. If you follow a risky strategy by choosing to tackle a large number of problems, you might end up in the bottom half of the score list when each is only 90% solved, or you might be the winner in the end. On the other hand, choosing a smaller number of problems has the risk that you have solved them correctly after four and a half hours, but the remaining time is too short to start and finish a new problem, thus wasting ten percent of the valuable contest time.

Time management should play a role in your strategy. If you are going to work on a large problem, start with it immediately or you will not finish it. Although this sounds trivial, there are a lot of teams which start out with the small problems, finish them quickly, and end up with only three problems solved because they did not finish the larger ones. In our opinion, debugging should have the highest priority at the terminal after 3.5 hours. When you start with a new problem that late in a contest, the terminal will become a bottleneck for the rest of the contest.

Of course terminal management is crucial. Though most programs are quite small (usually not exceeding one hundred lines of code), the terminal is often a bottleneck: Everyone wants to use it at the same time. How can this be avoided? The first thing to remember is: Use the chair in front of the terminal only for typing, not for thinking. Write your program on paper, down to the last semicolon. In this way you usually have a much better overview, and you have the time to consider all possible exceptions without someone breathing down your neck, longing for the terminal. Once you have finished writing, typing will take no more than 15 minutes. Though you should avoid debugging (this IS possible if you plan the program carefully on paper), when you really have to do it you should do it in a similar way: Collect as much data as possible from your program, print it out and analyze it on paper together with your code listing. Real- time tracing is *THE ULTIMATE SIN*.

Some Example Strategies

1. The Simple Strategy

This strategy is quite useful for novice teams, or those who do not want to get into a lot of practice and strategy tuning, and, therefore, is in no way optimal. The basic idea is to work as individually as possible to try to minimize overhead. Everyone reads a couple of problems, takes the one he likes most and starts working on it. When a problem is finished, a new one is picked in the same way and so on.

Advantages are that little practice is needed. Total elapsed time will be minimal, since the easiest problems are solved first. However, there are also severe disadvantages: Since the easiest problems usually have the same level of difficulty, everyone will finish their problem at about the same time. Thus the terminal will not be used for the first hour, since everyone is working on a problem on paper, and remains a bottleneck thereafter. Furthermore, only the easy problems will be solved, because no

time will be left for the hard ones. The conclusion is that, provided your programming skills are adequate, you will solve about three or four problems as a team. This will bring you, with a good total elapsed time, into the top ten, but probably not into the top three.

2. Terminal Man

In the terminal man (TM) strategy, only one of the team members, the T, uses the computer. The other two team members analyze the problem set, write down the algorithms and the (key parts) of the code, while the T makes the necessary I/O-routines. After an algorithm is finished, the T starts typing and, if necessary, does some simple debugging. If the bug is difficult to find, the original author of the algorithm helps the T to find it.

Advantages of this strategy are that the terminal is not a bottleneck anymore, and the task of solving a problem is split over people who specialized in the different parts of the problem solving process. A disadvantage is that no optimal use is made of the capacities of the T, who is mainly a kind of secretary. If you only one of you is familiar with the programming environment, this might be a good strategy. You can write a lot of programs in the first part of the contest when your brain is still fresh, since the typing and debugging is done by someone else. It depends strongly on the composition of your team if this strategy is suitable for you.

3. Think Tank

The strategy we followed during the preparation and playing of the Contest Finals of 1995 made use of the above-mentioned ``think tank'' (TT). We felt that choosing and analyzing the problems was such a crucial task in the early stages of a contest that it should not be left to a single person. The two team members who are the best problem analyzers form the TT and start reading the problems. Meanwhile the third member, the ``programmer'', will type in some useful standard subroutines and all the test data, which are checked carefully. After 15 minutes, the TT discusses the problems briefly and picks the one most suitable for the third team member. After explaining the key idea to the programmer, they can start working on it. Then the TT discusses all problems thoroughly, and puts the main ideas of the algorithm down on paper. We found out that two people examining hard problems often lead to creative solutions. After one hour the TT had a good overview over the problem set, and all algorithms were found. The next decision is how many problems you want to solve. The easiest or shortest problems are handled by the programmer, while the TT divides the other ones among themselves.

The terminal is only used for typing in code from paper or gathering information from a buggy program. If a program is rejected by the jury and no bug can be found, it is put aside until the last hour of the contest. In principle, after three and a half hours no more new code is typed. The team will briefly discuss the situation, and a plan is made for how to solve the problems which have yet to be debugged.

Some advantages of this approach are that you will almost always tackle the programs which have a reasonable chance of being solved correctly, and the hard problems can be solved because the TT will start working on them in an early stage of the contest. A clear disadvantage is that you will have a relatively slow start and your total time is not optimal. So to win, you need to solve one problem more than the other teams. We feel that for a team consisting of partners with about equal skills, this strategy will help you solve as many problems as possible.

Some Other Tips

You can practice a lot for a programming contest, and your strategy can be very good. However, luck always has its part in the contest and you have to live with that. Do not be disturbed by it (or the lack of it). Play your own contest. Never look at other team's standing, except to see if some teams solved a problem rather quickly that you thought to be too hard. If a program gets rejected by the jury, don't panic. Try to find the bug, there always is one. Consider especially the limits of your program, and ask yourself under which circumstances these limits will be exceeded. You do not have to submit a correct program. It only has to produce the right output for the jury input. Therefore you should program robustly and cleanly, and not write the shortest or fastest code possible. And always remember: Programming contests are great fun!

Concluding Remarks

In this article we have recorded some of our experiences with programming contest strategies. Though these strategies are very dependent on the individual qualities of the team members, the concepts apply equally to all teams. We hope that the information contained in this article will be useful to you, should you ever want to participate in the ACM Programming Contest (we definitely recommend it!). More information about the contest can be found on http://www.acm.org/~contest. A report of our experiences at the Contest Finals, including more considerations on team strategy, can be found at http://www.cs.vu.nl/~acmteam/ .

About the authors:

Fabian Ernst is a 24-year old PhD-student at Delft University of Technology, the Netherlands, in the field of Mathematical Physics, and also studies Administration Science in Delft. He competed once in the Contest Finals (for VU Amsterdam) in 1995. He can be reached at `ernst@math.tudelft.nl`

Jeroen Moelands is a 22-year old computer science and business computer science student at Vrije Universiteit Amsterdam, the Netherlands. He competed in the Contest Finals both in 1994 and 1995. His email-address is: `jeroenm@cs.vu.nl`

Seppo Pieterse is a 23-year old computer science and econometrics student at Vrije Universiteit Amsterdam, the Netherlands. He competed in the Contest Finals in 1994 and 1995, finishing 5th in 1994. He can be reached at `spieters@cs.vu.nl`