Setting problems for programming competitions

Emil Indzhev

October 24, 2019

Introduction

In this essay I will give an overview of the process of setting problems for programming competitions in the style of the IOI (International Olympiad in Informatics), though a lot of the information does not depend on the exact format of the competition. It may serve as a guide to others wanting to do the same (perhaps in a different field) or it might just be of interest to the reader.

I will first explain the style of the competitions, then the different approaches toward coming up with an idea for a problem and finally how to actually implement it into a suitable task for a competition.

1 Competition description

1.1 Format

Each competition consists of one or more (usually two) competition days that are independent from each other. In each day the contestants individually solve three problems, each of which is scored from 0 to 100 points. The competitors are given a working space with a PC and other materials (such as pens, paper and so on) and have five hours to attempt to solve the problems. While they program on their PCs, they submit their solutions to a grading system, which tests their solution against a set of tests prepared by the problem author. Usually this means checking whether they get the same answers as the author, but for some problems the process may be more complicated. At the end of this process, the contestants (usually) see how their program performed and what score it got. They can submit solutions a limited number of times (usually fifty per task or fifty for the whole day). The final score of each competitor is the sum of the number of points they got in each problem.

1.2 Contestants

This format is most popular for high school level competitions. This is because this is the format of the IOI and each participating country holds national competitions to perform the selection for the national team, the four people it sends to compete at the IOI. So, in order to select the most capable students, it is in the country's best interest to make the national competitions as similar to the IOI as possible. Online competitions or ones for university students usually have different formats (though the style of problems is similar, there are some differences there as well).

1.3 Problems

Each problem has a statement which describes what task the contestant's program must solve. For example it might describe that there is a directed graph and we are looking for the length of the shortest path between two nodes. A typical feature of informatics problems at basically all competitions is an old tradition for the problems to have some story (so it isn't a graph, but rather a country with towns and roads, and we aren't just looking for the shortest path, rather our protagonist must get from town A to town B as fast as possible because...). The statement also includes other specifications, which will be discussed later.

The focus of these competitions is on the logic behind the solution and not any of the fluff around it. This logic might be based on some well known algorithm or some mathematical idea, or just some clever construction, but the most important thing for a good problem is that the solution should be interesting. The main part should never be implementing this one obscure algorithm or just using some theorem. Rather, it should be spotting how to

transform the problem to use some algorithm/technique or deriving some maths that helps us solve it. Furthermore, it might not include anything too standard and be entirely based around original ideas. Important features of the solution, which the contestant should keep in mind, are its memory and time complexity. These are what actually provide difficulty in most cases, as almost any problem can be solved in some absurd amount of time through brute force for example.

Because of this, the statement also specifies some constraints. These are the constraints of all variables from the input (e.g. N is a whole natural number that is less than ...), but also the memory limit (how much memory the program can allocate) and the time limit (how long the program can take to terminate). If the contestant's program breaks the memory limit or time limit when running on a particular test, it will fail that test. This is basically equivalent to setting a limit on the complexity, but since the goal is to write a program, we have to actually measure how fast it is. The contestant is expected to roughly calculate how long a program with some complexity will take to run by approximating its constant and knowing the specifications of the grading system.

Another thing the statement formally describes are the input and output formats or, more generally, the way the contestant's program interacts with the world (it might be through function calls for example). Something which is definitely not the focus of the problems is parsing input or getting data from a database, or formatting the output, or anything else of that fashion. Any well made problem will have a clean IO format. A while ago the most popular way for communication used to be through standard input and output, while function calls were used only for some problems where they are needed. However, in recent times all problems at the IOI use function communication, even when all the input is given at once and then all the output is received at once (as opposed to taking turns). This is because a more experienced programmer might know of a way to speed up the input (e.g. not using the built-in read function of the language, but instead writing a custom one that is faster for the specified input format). However, as mentioned earlier, this is not the focus of the competitions.

The last important part of each problem are its subtasks. However, the easiest way to explain what those are is to first explain what they aren't. As has been noted, each problem is scored from 0 to 100. Obviously, a completely correct solution will get a hundred points and a completely incorrect one will get zero points, but the in-between remains. Like before, a while ago the most common thing to do was to prepare a number of tests of ascending difficulty (for example by gradually increasing the constraints and other techniques) and score each separately as either pass or fail (or maybe group them in small batches of 2-3 tests, to prevent the program from randomly guessing). The way these tests are arranged and exactly how the difficulty varies among them would remain a mystery for the contestants. This poses two main problems. Firstly, when a contestant comes up with some solution, he or she can't know or even have a vague idea of how many points it will get without first implementing it, which introduces a factor of luck into the competition. Secondly, the nature of gradually increasing difficulty encourages the contestants to repeatedly perform small optimizations on their solutions, which isn't the intention behind most

problems and takes the focus away from the important parts.

This is where the idea of subtasks comes in. They are partitions of the problem with increasing difficulty. Each subtask is fully described in the problem statement: how many points it is worth, what are the constraints of the input variables, what are the additional constraints that affect the difficulty (if any), and anything else that is relevant. Then each subtask is tested with a number of tests and the contestant either gets all the points for it or none of them. This, of course, is the most standard way for scoring, though others exist as well; however, the general idea remains the same.

Some problems don't fit into the pattern described above. They are what is commonly called optimizational problems. There your goal isn't always just to pass some tests, but rather get as good a result as possible. Think of the traveling salesman problem or facial recognition. Both are problems in which you wouldn't expect optimal solutions. Such problems would usually be partially scored on each test and have the tests be independent (or group them by subtasks and for each subtask take the minimum result of any test in it as the result for the subtask). A common thing to do when trying to optimize for some metric is to compare the result the contestant got to the result the author got and receive a fraction of the points for the test.

2 Creating problems

2.1 Ideas

The first step of setting a task is coming up with an idea for a problem and of course a solution, which solves the problem. Very often this is also the hardest step. Here I will describe several possible approaches, but there will always be some element of inspiration needed.

2.1.1 Problem first

The cleanest approach (which usually also leads to the cleanest problems) is to come up with some problem and then try to solve it. A possible source of inspiration are real-world problems with slight simplifications to make them more suitable. Another approach is thinking about some more abstract setup and then having to find the value of some mathematical property that it has (such as the number of combinations which satisfy some criteria). If your solution is not too obvious (but also doesn't take way too long), is better than the trivial solutions and is interesting (not just some obscure algorithm or data structure) you might have a good candidate for a problem. Here I will give as an example one of my favorite problems that I set.

A rover on Mars wants to send N bits of data to Earth. It will send some M bit long message, where it chooses M. It knows that D of those bits will be randomly deleted. On Earth we receive the corrupted message and want to reconstruct the original data as best as possible. The goal is to optimize the accuracy of the reconstruction divided by the length of the received message.

The contestant has to write two programs: one for the transmitter (generates the message when given the data as well as N and D) and one for the receiver (reconstructs the date when given the corrupted message and again N and D).

The statement is simple enough and seems like a very practical problem. The other nice thing is that as it happened there are several different approaches for solving it, each of which has a different difficulty (to come up with and implement), but also no matter which one you go with you can also try doing optimizations to it to further increase your score (which is a desirable property for this class of problems). The main idea is to realize that the most significant danger in the reconstruction is misalignment and not single bit errors. The simplest idea for a solution is to repeat each bit several times. This approach (with some other improvements) can get between 20 and 50 points. Another idea is to group the data in chunks and then extend each chunk with some extra bits (such as a hash or just some marker in the end), after that we will need some way to detect the chunks and their markers. Such approaches can get upwards of 70 points if really well done. The best idea was to randomly pad the message (or rather pseudo-randomly generate a skeleton with N empty slots and insert the data there). Then in the receiver we just do a simplified version of the minimum edit distance algorithm between the skeleton and the corrupted messages (empty slots in the skeleton are always marked as a match). After that we only need to fine tune the formula for the length of the skeleton depending on N and D, but for simplicity of the task, N is always set to 10 thousand.

Another simple sounding problem I invented in this way is the following: Given a set of N line segments with different integer lengths, how many different valid triangles can we construct? The gist of the solution is: first, a step which is equivalent to raising a polynomial to the second power and second, a linear step with a two pointers approach. The first step can be done quadratically for 40-50 points (depending on the constant), with Karatsuba's algorithm for 75 points and with Fast Fourier Transform for 100 points.

2.1.2 Solution first

As we can see, the approach above can lead to some very clean problems with nice solutions. However, the issue is that very few problems one comes up with will have interesting solutions in polynomial time that aren't too difficult and aren't too hard. So, one idea is to first come up with a solution and then backwards interpolate what the problem should be. In general, this leads to very bad problems. Since we start with the solution, we usually think of standard algorithms, structures and approaches (and not of novel ideas, since we don't have anything to base them on). Then what usually happens is we just chain them together one after another. Thus, the solution ends up being very ugly. Furthermore, the problem becomes way too artificial, to the point where it is extremely obvious it had no actual motivation.

2.1.3 Compromise

The most common thing to do, which still yields nice problems is a compromise between these two approaches. First you come up with some problem and try to solve it. Then based on that repeatedly tweak the problem and/or solution while keeping in mind how this affects the other. These tweaks may be small, such as changing some detail to make the solution cleaner, or add some difficulty by introducing an additional case. However, they may sometimes be quite significant, leaving the original problem unrecognizable and even forcing us to invent an entirely new problem. This may leave the reader wondering what the point of the original problem was, why not just start with a solution. As said above, the main problem with this approach is that there is no outside factor (the problem we should be trying to solve) to force us to invent new ideas. By first starting with some problem, we may end up coming up with really interesting original techniques to solve it. Even if those don't result in a solution for that particular problem, we may then invent a new problem where these techniques can be applied. We should still be careful though, as the issue of ending up with an overly artificial problem remains. This is where the true compromise is made.

2.2 Scoring and subtasks

After having invented the main idea, we need to come up with the scoring system, which in most cases will be creating subtasks. The main thing to keep in mind is that each subtask should have an associated solution. Or rather, you should first think of suboptimal solutions for your problem that have worse complexity or solutions, which don't cover the entire problem (only some cases for example). Then you should arrange all those solutions in some progression (not necessarily linear). Then for each node in this directed acyclic graph decide how many points it is worth. Those are your subtasks. What is left is to create tests for them.

In the rare case when subtasks are not appropriate the same thought process applies. You should again first consider different possible solutions and only then decide on the scoring (or at least tweak it after considering them).

2.3 Tests

Then you have to create the actual tests. This may sound simple enough (and sometimes it really is), but if neglected might result in the task being effectively worthless. If for example we decide to randomly generate all our tests without much thought, there is a very high likelihood that some greedy solution, which isn't even correct in the general case, will pass most, if not all, of the tests. And even if it happens to fail a few, don't forget that the contestants have 50 submissions to try their hardest to pass those final few tests. Alternatively, maybe they have a correct solution with worse complexity, but after adding a few "break" statements and other constant optimizations, they might get it

under the time limit. This collection of techniques for getting points you don't really deserve is colloquially called "cheating".

Because of this, the author must take measures to make their tests strong and "cheat"-proof. This generally requires thinking of many possible "cheat" techniques for the problem and adding tests against each of them, for example by having the tests follow different structures.

In some tasks the contestant might need to write a program which takes turns interacting with the author's program. This may be because he is trying to find something hidden by asking questions of a specific format and receiving answers from the jury (and maybe there is a limit on the number of questions he is allowed to ask). A "cheater"'s solution might make assumptions and guesses it isn't sure of based on a probabilistic strategy. A common counter-strategy the author can employ is to have his program change the answer depending on what the contestant's program asked to the worse possible one, while still keeping all previously given answers valid. If this was the real world and the contestant and the author were playing a real game, we would say that the author is cheating, but when setting problems this is completely legitimate.

2.4 Finalizing

To finish a problem, it is very important to have people test it, attempt to solve it and implement it. Ask them for feedback and possibly adjust the problem or at least the scores you assigned for the different subtasks. Also, make sure the problem statement is clear to everyone and, if it isn't, rewrite it. The best selection for these beta-testers are other ex-competitors and problem setters as they will both have the skills to actually solve the problem and also know what feels right in a problem and what doesn't. After that you just need to come up with a nice story for the statement. Experience has shown that a common choice for a protagonist is a current girlfriend (or, as inevitably happens, an ex-girlfriend).

Conclusion

This essay has covered the nature of informatics competitions and the problems in them. Furthermore, the current conventions in problem setting were explained, as was some of the motivation behind them. Finally, a guideline for actually setting problems was described. Of course, just reading this essay doesn't mean the reader is fully prepared to instantly set perfect problems, but should have given them some helpful advice.

References

[1] *https://ioinformatics.org/*
Official website of the IOI, used for referencing when different approaches were used.

[2] *http://www.math.bas.bg/infos*
Official website of the Bulgarian national competitions, all referenced problems and solutions can be found here.

[3] *https://github.com/indjev99/Competitive-Programming-Problems*
GitHub repository with all programming problems I have given.