Regional Contest Cookbook-Judging-guidelines

Guidelines for Producing a Programming Contest Problem Set

First Edition:
Tom Verhoeff
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
E-mail: Tom.Verhoeff@acm.org
October 1988, Revised and expanded July 1990 and August 2020

Revised Edition:
June 2026
Marc Furon, David Van Brackle, Etienne Vouga
(with contributions from documents by Nick Wu and Donna Furon)

Abstract

In this document we give some guidelines to produce a problem set for "ICPC-style" programming contests. The reader is assumed to be familiar with the operation of such programming contests. Where possible, we also explain the motivation behind the guidelines.

The first two sections deal with aspects related to the problem set as a whole, viz. requirements and production schedule. They are especially intended for the Chief Judge. The remaining sections cover the following aspects concerning individual problems in the problem set: selection, specification, solution, and testing. These sections are more directed towards the problem authors.

A good problem set is the key to a successful programming contest. The production of such a problem set should be taken very seriously and requires a major effort by several people. The guidelines presented in this note should enable you to do a good and enjoyable job.

Problem Set Requirements

The Chief Judge is responsible for the timely production of a suitable problem set.

Most contest problem sets consist of at least ten problems. There should be sufficient variation in both the difficulty and the topic areas of the problems. The problems should span a range of difficulty; at a regional contest, at least two should be readily solvable by first-time contestants in the first year or two of study. Ideally, the problems then escalate in difficulty. The most challenging problems should determine which teams are ready to advance to compete at a championship level.

All problems should be original in some respect. For example, a problem can be new as such, or its embedding into a context can be new, or its boundary conditions may be changed such that a new approach is required.

The reason for having at least ten problems is that we prefer to measure the strength of teams by number of problems solved and not by amount of time spent on solving them. (Recall that ranking is done first on account of number of problems solved and, in case of a tie, on account of time spent.) To provide sufficient resolution, the problem set should be so large that the strongest team can solve (almost) all problems just in time. Of course, the number of competing teams, the difficulty of the problems, and the duration of the contest are also important factors. At World Finals 49 (a five-hour contest, with over 130 teams) 12 problems were posed with satisfactory results.

The reasons for including at least two easy problems are the following.

1. Easy problems can serve as starters and allow early usage of the PC (this requires that the teams make the right judgment about difficulty; hence, assessment of problem difficulty is an important aspect of the contest).
2. Weak teams should have some fun too; therefore, they should be enabled to solve---not just work on---some problems.
3. Strong teams should be able to solve easy problems quickly (say, in half an hour or less), which can validate that the contest systems are working as expected and gives other teams assurance that these problems are solvable.
4. Teams solving no problems cannot be differentiated in the final ranking, whereas teams that solved at least one problem can be ranked according to time spent.

The presence of several more difficult problems serves to differentiate the strongest teams on account of the number of problems solved and not just on account of time spent. Furthermore, difficult problems make the contest more interesting: we would not want the contest to consist solely of implementation jobs.

We know from experience that it is unlikely that an interesting problem is too easy. (Keep in mind that teams usually read all problems first and need to decide on an appropriate order to solve them, before they start programming. Problems should not be sorted by difficulty in ICPC. Ordering the problems is the chief judge's responsibility. There are some considerations that go into choosing the first problem; it should not be a hard problem that looks easy, as that will lead to contestants effectively DDOSing the Contest Control system.) Also note that it hardly makes sense to include problems that are too difficult. The only thing that the presence of a too difficult problem can measure is the ability of the teams to recognize and then ignore the problem, which is not an objective of the contest.

The reasons to cover various topic areas are the following:

1. If one aspect (for example, juggling with dates, graph theory, computational geometry, or numerical approximation) appears in many of the problems, then "specialists" in that field have an advantage.
2. Variation allows teams to work on those types of problems they like most (only strong teams will have little choice: they should attempt to solve most, or even all, problems).

However, a nice thing about two problems with a common aspect is that a team recognizing this may benefit by sharing code between the solutions.

The reasons for having original problems are:

1. to make the contest interesting for the contestants and the organizers,
2. to avoid giving an unfair advantage to teams familiar with particular problems, and
3. to attract some attention from the Computing Science community.

Each problem must have a short and unambiguous identifier, for instance, a letter or number. This identifier will also be used on the Clarification Requests and Standings.

Overall Production Schedule

The following phases can be distinguished in the production of a problem set.

1. Call for Problems:

   Generate ideas for problems, understanding what the crucial computational aspect is of each problem. Gather ideas from the judges and potentially from outside sources that are permitted by the rules of the contest. (For example, are coaches allowed to submit problem ideas?)

2. Problem Evaluation:

   Judges review proposed problems, considering the need for originality and variation in difficulty and topic areas, and eliminating problems that are too easy or too difficult. At a regional level, even very easy problems may be appropriate. A regional contest is likely to be the first competitive programming experience for many contestants, and the nature of the ICPC--working in a team environment with only one computer in a timed setting--is already demanding. New—and sometimes not-so-new—judges frequently

underestimate the difficulty of a problem. Problems found unsuitable are eliminated. Remaining problems are ranked and prioritized for development.

3. Problem Selection, Development, and Quality Control:
    1. Specify problems in detail.
    2. Write complete programs for problems (including alternative solutions).
    3. Write input validator and output verifier. The Contest Control Systems provide default output verifiers which are often suitable. Tools for input validation are described below.
    4. Produce test sample (public) and judges' (secret) input data files with corresponding output files.
    5. Validate examples, validate test data, proofread for spelling, grammar, and clarity.
    6. Prepare post-contest editorial or slides to discuss intended problem solution(s) and potential pitfalls.

4. The Chief Judge should coordinate with the contest and site director(s) to arrange for the problem set to be printed. Each team should be provided with a copy of the problem set for each contestant.

Set a planned schedule for each phase. We suggest that you have a checkpoint at the end of each phase—this could be an on-line discussion or meeting.

The Chief Judge needs to consider that a problem may turn out to be unsuitable after it has been developed. Initially, aim for several more problems than the judges intend to pose. In the 1989-90 European Regional the judges posed all nine problems that had been prepared (none had to be abandoned) and, contrary to expectations, one team was able to solve all nine of them! The top teams are even stronger now, and the computers and development tools used in 2026 are much more capable than those used in 1990.

Ensure that for each problem there is one person taking full responsibility for all aspects of a problem's development. This person is sometimes referred to as a problem's "boss". It is also expected that judges will read each other's specifications and attempt solutions. Every problem

should have multiple reference solutions developed independently. These solutions should be written in multiple contest language "families" (as of this writing, C/C++, Java/Kotlin, and Python 3). As of 2026, every problem should have reference solutions in at least two of those language families within the given time and memory limits. There is no guarantee that every problem can be solved within the given time and memory limits in every language.

Problem Creation Overview

Keep the following in mind when developing problems:

Each team consists of up to three programmers (usually computing science students at undergraduate or first-year graduate level, depending on the nature of the computer science programs in the region). They have one (1) personal computer with compilers and development tools for the supported contest languages at their disposal and they may consult their printed Team Reference Document. (At many regional contests, teams may bring their choice of non-computer-readable literature.) The contest lasts five hours.

Judging is done by running the submitted program for some test cases and subsequently checking its output. A program is either accepted or rejected. In the latter case, the team is given a 20 minute time penalty for the problem should the team eventually solve it and one of the following reasons for rejection is indicated:

1. Compilation Error *(no rejection time penalty assessed)*
2. Run-Time Error
3. Time-Limit Exceeded
4. Wrong Answer

The run-time limit is specified for each problem, as is the memory limit. Time limits are set specifically for each problem; memory limits are frequently defaulted to a common value (2 GB as of 2026).

A problem may be interesting for several reasons. Various aspects can include

- clever algorithm
- efficiency (time and/or space)
- preprocessing
- complicated case analysis
- irregularities
- internal data representation
- input parsing
- understanding limits
- common subproblems

Problem Specification

Problems will be posed in English.

The aim is to specify a problem completely and unambiguously. Although teams may make clarification requests, it is better to obviate these with careful problem specification. English is a second language for many contestants, even in countries where English is the predominant language. Ensure that the problem text is clearly written and uses proper "book" English grammar. Minimize the use of idioms and colloquial language. A problem specification should provide sufficient context to understand what is being asked, along with the needed information about the input for the program and the output to be produced. Ideally, no clarification request should need to be answered.

The problem specification will usually start with a story and the introduction of some concepts, possibly accompanied by instructive examples. This is followed by a short informal description of the programming task and then a more precise specification of input and output format and their desired relation. Finally, one or more example input sets are provided, along with corresponding output. Most problems provide the specified input on "Standard Input" and require the output to be judged to be produced on "Standard Output". It is permitted, although not common, to require more than one input and/or output file for a problem.

The following example is from the 2024/2025 North America Championship held in Orlando, Florida.

Problem D: Geometry Rush

You are playing the summer's hottest rhythm-based action platformer—Geometry Rush! The game is played on a 2D plane. Your character begins at (0, 0) and every second must move at a 45-degree angle either up-right or down-right, which takes your character from position (x, y) to ($x + 1$, $y + 1$) or ($x + 1$, $y - 1$) respectively. You can change which direction you move every second, but not in between moves. There are obstacles protruding from the floor and ceiling that you must dodge. You win the game if, after w seconds, you reach the line $x = w$ without having touched any obstacles on the way.

The play area extends vertically from $y = -h$ to $y = h$. Obstacles are two polygonal curves: one curve starts at (0, h) and ends at (w, h) and represents a ceiling of varying height. The x values of the vertices of this curve are non-decreasing, and the y values lie between $-h$ and h inclusive. A second polygonal curve starts at (0, $-h$) and ends at (w, $-h$) and represents the floor. Its vertices satisfy similar constraints.

Your character is a point of negligible extent: you can move from position (x, y) to ($x + 1$, $y - 1$) so long as the line segment between your start and end position does not intersect either obstacle. (Exactly touching either polygonal curve counts as intersecting an obstacle, and loses the game.)

You have played *a lot* of games of Geometry Rush. To keep the game interesting, you have started to set challenges for yourself. For example: you win the game no matter where you cross the $x = w$ goal line. But for what maximum value of y can you win the game by crossing at (w, y) without touching any obstacles on the way? For what minimum value? Compute these numbers.

Input

The first line of the input contains four space-separated integers n, m, w, and h. The first two integers ($3 \le n, m \le 10^5$) are the number of vertices in the ceiling and floor polygonal curves,

respectively. The second two integers ($3 \le w, h \le 10^5$) are the width and height of the play area, as described above.

The next n lines each contain two space-separated integers x and y ($0 \le x \le w$; $-h \le y \le h$): the coordinates of the vertices of the ceiling polygonal curve, in order from left to right. It is guaranteed that the first vertex is at $(0, h)$ and the last vertex is at (w, h).

The next m lines each contain two space-separated integers x and y ($0 \le x \le w$; $-h \le y \le h$): the coordinates of the vertices of the floor polygonal curve, in order from left to right. It is guaranteed that the first vertex is at $(0, -h)$ and the last vertex is at $(w, -h)$.

For both polygonal curves: the x coordinates are non-decreasing, all vertices are distinct, and the curve does not self-intersect. Neither curve intersects $(0, 0)$. (The floor and ceiling curves might intersect each other, in which case the game is unwinnable.)

Output

If it is impossible to win the game, print `impossible`. Otherwise, print two space-separated integers: the minimum and maximum y values that the player could reach at $x = w$ without losing the game by touching an obstacle along the way.

Sample Input 1

```
4 4 5 5
0 5
0 2
5 2
5 5
0 -5
0 -2
5 -2
5 -5
```

Sample Output 1

```
-1 1
```

Sample Input 2

```
4 4 6 5
0 5
```

Sample Output 2

```
0 0
```

0 2 6 2 6 5 0 -5 0 -2 6 -2 6 -5	

Sample Input 3 *Sample Output 3*

3 3 7 5 0 5 5 -1 7 5 0 -5 2 1 7 -5	impossible

Sample Input 4 *Sample Output 4*

4 3 5 5 0 5 0 2 5 2 5 5 0 -5 3 -1 5 -5	-1 1

The specification clearly indicates the possible counts of input values and the allowed ranges of those values. As a problem author you should be aware of and specify these limits in the text as needed. One set of sample input and corresponding sample output should be provided; more sets may be added if the problem author feels that doing so is warranted. For this problem, four sets of sample input and output are provided.

The problem author should be prepared to summarize the crucial computational aspects of the problem. In this summary, tricky issues, boundary cases, and required efficiency considerations

may be mentioned. Of course, the summary should not be revealed to the contestants until after the contest. To keep reference solutions independent, the problem author should not initially share this with the other judges. Many judging teams set up a private platform to discuss problems during the development process, such as Discord or Slack.

At some time after the contest, this summary can be edited and released as either a set of presentation slides or as an "editorial" document. Doing so is encouraged as the problem can more easily be studied and used for team training and other instructional purposes.

For the above problem the following summary was produced:

Problem

Starting at (0, 0), you can move diagonally (1, −1) or (1, 1). You are given two polygonal

curves, one guaranteed to be above you, and the other below you. For both curves, the points of each curve have non-decreasing x values, and each curve does not self-intersect. You lose if you collide with either curve along the way. The goal is to reach $x = w$ without losing. Output the minimum and maximum y values such that you win, or print impossible if you cannot win.

Solution

Since the curves have non-decreasing x values, we can use a two-pointer solution. We will keep track of the minimum and maximum y values as we move along x, denoted y_{min} and y_{max} respectively. As we increment x, we check every line that includes x and find the minimum/maximum y values that do not collide with the curves, denoted y_{low} and y_{high} respectively. Since you can only move (1, −1) or (1, 1), the parity of x and y must be the same, so we tighten the bounds y_{low} and y_{high} accordingly. Finally, y_{min} can be updated as the maximum of $y_{min} - 1$ and y_{low} , and y_{max} can be updated as the minimum of $y_{max} + 1$ and y_{high}. If at any point, $y_{min} > y_{max}$ it is impossible to win. This gives us an $O(n + m + w)$ solution.

The problem can also be solved via a linear sweep.

Testing

Although we are interested in formally correct programs for the problems, judgment is based on testing. Hence, testability is an important issue.

The test cases that are applied to exercise a program must allow a good assessment of the program's correctness. There must be enough test cases to thoroughly test all the relevant aspects of the problem and its input space. It may be convenient to specify the problem such that one input file allows a sequence of tests to be applied (this is not the case for the above example). Remember that a per input test file execution time-limit must be set.

If there is not much variety in the required output of a problem (e.g. only yes/no), then a program might be accepted when it correctly "guesses" the output. Produce sufficient separate test case input files to test the submitted program thoroughly.

Include both "normal" and special (boundary) cases in the test data. For the above problem, cases would include the minimum and maximum number of vertices on each curve, the minimum and maximum play area, curves that intersect or touch each other, curves such that the final minimum and maximum y values are equal, and so forth.

To judge a submission the Contest Control System uses an output verification program. This verifier takes the output of the submitted program and checks it, e.g., against the output of a known correct program or/and with the aid of the input file. Most problems generate an output stream on standard output; this is compared with a judges' expected output file for each input test case. A default comparison tool is normally included with the Contest Control System. The standard for this tool includes comparison of whitespace-separated tokens, also allowing for comparison of floating-point values within a given tolerance ("epsilon"), and specifying that letter case and spacing is or is not significant. The output must be specified as much as necessary, considering such oddities as trailing blanks, leading zeroes, empty lines, etc.

If the standard comparison tool is insufficient to evaluate a submission, a custom output validator can be written. The specifications for this are included in the ICPC Problem Package Format documentation. Programs can be written in C/C++ or Python 3. The output validator

reads the output from the contestant submission and ultimately exits with a return code indicating that the submission should be accepted or rejected.

ICPC Problem Package Format Specifications:

https://icpc.io/problem-package-format/spec/legacy-icpc.html

It is also possible to write interactive problems—these are problems where the contestant submission reads from and writes to another program supplied by the judges. The judges' program can adapt its responses based on the data written by the contestant submission. This makes it possible to ask problems that play games, solve puzzles, etc. The judges' program can act in a way that will attempt to expose errors in a contestant submission.

Problem Reference Solutions

Once a problem has been specified it is necessary to obtain a better estimate of its suitability for the contest. This can be done by writing a complete program for the problem. That way one may encounter some trouble spots that otherwise would remain unnoticed.

It should take the problem author, who is intimately familiar with the problem after specifying it, no more than a few hours to implement a working solution. That solution should be more or less along the lines of an "intended" solution. It is not necessary to come up with a very neat and fully annotated solution (although that may be desirable). From this solution one can also determine some limits on run-time and memory utilization, which are useful when putting together test cases (discussed in the previous section). Other judges should contribute their independent solutions as well, in multiple language families as described earlier.

If it is the intention to rule out some straightforward solutions on account of their inefficiency (e.g. $O(N^3)$ instead of $O(N)$), then it is necessary to implement "unintended" alternative solutions as well. The alternative solutions must fail the tests (see below) even if they were otherwise cleverly coded! You may sometimes be surprised how "good" unintended solutions turn out to be. The time limits for a problem are set in conjunction with the technical team; the technical team runs the reference solutions in the contest environment and reports the

time taken for the slowest test case by the slowest reference solution that is intended to be accepted, and by the fastest reference solution that should be considered too slow. The time limit is set to a value in between these times. The same is true for memory in the (infrequent) event a non-standard memory limit is to be set. The unintended reference solutions become part of the problem package. In addition to helping set time limits, these slow, often exhaustive, solutions can also help validate the correct behavior of the intended solutions with at least some of the test data.

Apart from the bare "intended" solution it is also necessary to write another program. Input validation is not part of the contestants' programming task. That is, teams are to assume that input supplied to their programs agrees with the specification. The problem author, however, must see to it that examples and test input data are valid. There are tools developed specifically to assist with this; VIVA and CheckTestData are two. These tools check certain aspects of the input data files, but may not check everything that should be verified. A problem author can write a custom program to perform input validation—Python 3 is commonly used for this. Another approach is to modify a reference solution to make the input validation checks.

VIVA can be found here:

http://viva.vanb.org/

Checktestdata can be found here:

https://github.com/DOMjudge/checktestdata

For the above sample problem, the input validator must check that the first line contains four integer values, that the specified number of vertices in the ceiling and floor curves are in range, and that the height and width of the play area is also in range. It also must check that there are the proper number of lines specifying coordinates for each curve, that each coordinate is in the specified range, and that the coordinates are in order from left to right (increasing x values).

Summary

Aim for at least ten original problems. Include two easy problems and have the problem difficulty increase from there. Avoid too difficult problems. Cover various topic areas. The best team should be able to solve (almost) all problems in the available time. The goal is that each problem is solved by at least one team, every team solves at least one problem, and no team solves all the problems in the allotted time (informally known as an "All Kill" or "Sweep" outcome).

Each problem should fall under the responsibility of one person. For each problem the Chief Judge needs to receive the following items from the problem's "boss":

1. a problem statement specification,
2. a set of public sample and secret test input data files, along with corresponding expected output,
3. an input validation program, or input validation tool specification files,
4. an output verification program (the verifier is often the one provided by the Contest Control System for deterministic problems, in which case test output data files must be provided),
5. multiple independent solutions in different language families,
6. (frequently) alternate solutions that are too slow, or incorrect, and
7. a short description of the crucial computational aspects and recommended solution technique(s), suitable for release as presentation slides or an editorial document. Post-contest, this material can also cover behavior of both accepted and rejected submissions during the contest.

Items 1 and 7 are discussed in the section on Problem Specification, items 2, 3, and 4 in section Testing, items 5 and 6 in section Problem Reference Solutions.

The judges are advised to do an evaluation after the contest to review how well the objectives were met. For instance, how difficult did the problems turn out to be, how many and which Clarification Requests were made, how thorough were the test cases, were any programs accepted that should not have been, etc.