Guidelines for Producing a Programming-Contest Problem Set

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

Abstract

In this note we give some guidelines to produce a problem set for "ACM-style" programming contests, in particular for the European Regional Programming Contest. 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 Head 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 Creators.

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 a number of people. The guidelines presented in this note should enable you to do a good and enjoyable job.

1 Problem set requirements

The Head Judge is responsible for the timely production of a suitable problem set. In the contest, each team should be provided with two (2) copies of the problem set.

The problem set consists of at least six (6) problems. There should be sufficient variation in both the difficulty and the computational aspects of the problems. Two (2) easy problems and one (1) more difficult problem should be included. 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 can be new.

The reason for having at least six 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 the Finals (a six-hour contest with 24 teams[1]) typically eight problems were posed with satisfactory results.

The reasons to include 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), which acts as an incentive for themselves and others (including judges etc.).

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 one or two 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 degrade to mere 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. Hence, it unlikely that a problem is solved within 15 minutes.) Moreover a problem can readily be made more difficult; making it easier is often painful. 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. This is not worthwhile the trouble of including such a problem.

The reason to cover various computational aspects are the following.

[1] Five hours since 1990.

1. If one aspect (say, juggling with dates 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 problem they like most (only strong teams will have little choice: they should solve all problems). On the other hand, a nice thing about two problems with a common aspect is that a team recognizing this fact may benefit by writing a combined solution.

The reasons for having original problems are

1. to make the contest interesting for the contestants and the organizers,

2. to avoid giving 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. This identifier will also be used on the Run Envelopes, Clarification Requests, and Standings. The cover page of the problem set mentions the date, the number of problems, their identifiers, and the number of pages. For example,

This is the problem set for the
1990–91 European Regional Programming Contest
held on Saturday November 3, 1990.
It contains 7 problems labeled A through G
printed on 10 numbered pages.

2 Production schedule

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

1. Generate ideas for problems, making explicit what the crucial computational aspect is of each problem.

2. Select initial set, taking into account the need for originality and variation in difficulty and computational aspects, and eliminating too difficult problems.

3. Specify problems in detail.

4. Write complete programs for problems (including alternative solutions where necessary).

5. Write input validator and output verifier.

6. Produce test input data files (and test output data files where necessary).

7. Validate examples, validate test data, check spelling.

8. Print a sufficient number of copies of the problem set.

Each phase should take no more than one week to complete. We suggest that you have a review meeting at the end of each phase.

The Head Judge needs to take into account that only afterwards a problem may turn out to be unsuitable. Initially, aim at eight or even nine problems. In the 1989–90 European Regional we posed all nine problems that we had prepared (none had to be abandoned) and, contrary to our expectations, one team was able to solve all nine of them!

See to it that for each problem there is one person taking full responsibility for all aspects (this would usually include judging as well). Of course, it is also a good idea to have Problem Creators read each other's specifications and possibly even attempt solutions.

3 Problem selection

Here are some facts to keep in mind when creating a problem.

Each team consists of up to four (4) programmers (usually computing science students at undergraduate or graduate level). They have one (1) personal computer with Turbo Pascal 5.5 at their disposal and they may consult non-computer-readable literature and non-programmable calculators. The contest lasts six (6) hours. However, for the Finals that will now be five hours. By the way, they reduced the length of the Finals so that

1. there would be more excitement at the end of the contest.
2. it would be less likely that the winners be known prior to the Banquet.
3. teams not making much progress would feel less frustrated.
4. judges would not feel compelled to increase the number of problems (compensating for the increased power of modern software development environments).
5. judges would be less tired and less likely to err.
6. five hours fits nicely between 1pm and 6pm.

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 minutes' penalty for the problem and one of the following reasons for rejection is indicated:

1. Syntax Error
2. Run-Time Error
3. Time-Limit Exceeded

4. Wrong Answer
5. Failed Test Case
6. Inaccurate Answer
7. Too Little/Much Output
8. Bad Output Format
9. Check Clarifications

The run-time limit is three (3) minutes unless otherwise indicated in the problem specification. There is still some controversy over these messages. We will come back to them in the section on testing.

A problem may be interesting for several reasons. Important aspects can be

- clever algorithm
- efficiency (time and/or space)
- preprocessing
- complicated case analysis
- irregularities
- internal data representation
- input parsing
- output format
- tricky issues (going beyond maxint, end-of-file detection)
- common subproblems

4 Problem specification

Problems will be posed in English. Each problem prescribes the names of the program source file and the data input and output files. Since judges recompile the program there is no need to mention the name of an executable file. A problem may require more than one input and/or output file. An execution time-limit is mentioned if it deviates from the standard amount (see above).

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, an example input set with corresponding output is given.

Of course, the aim is to specify a problem completely and unambiguously. Although teams may make clarification requests, it is better to obviate these by a careful problem specification. Keep in mind that English is a second language for most of the teams. Also keep in mind that long specifications tend to be more controversial than concise ones.

The following example was adapted from the 1989–90 Finals held in Washington D.C., USA.

Problem A	**Rare Order**
Source	`ORDER.PAS`
Input	`ORDER.DAT`
Output	`ORDER.OUT`

A rare book collector recently discovered a book written in an unfamiliar language that used the same characters as the English language. The book contained a short index, but the ordering of the items in the index was different from what one would expect if the characters were ordered the same way as in the English alphabet. The collector tried to use the index to determine the ordering of characters (i.e., the collating sequence) of the strange alphabet, then gave up with frustration at the tedium of the task. You are to write a program to complete the collector's work. In particular, your program will take a set of strings that has been sorted according to a particular collating sequence and determine what that sequence is.

Input The input consists of an ordered list of strings of uppercase letters, one string per line. Each string contains at most 20 characters. The end of the list is signalled by a line that is the single character `'#'`. Not all letters are necessarily used, but the list will imply a complete ordering among those letters that are used. A sample input file is shown below.

```
XWY
ZX
ZXY
ZXW
YWWX
#
```

Output Your output should be a single line containing uppercase letters in the order that specifies the collating sequence used to produce the input data file. Correct output for the input data file above is shown below.

```
XZYW
```

(END OF PROBLEM A)

This is a nice and inviting problem with a concise specification. However, it might not be completely clear to the contestants whether the input file may contain empty strings (at the very beginning only, since it is sorted) and whether the file may consist of the end-of-list marker `'#'` only. As a Problem Creator you should be aware of these extremes.

The Problem Creator should summarize the crucial computational aspects of the problem for the Head Judge. 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. For the above sample problem the following summary might be produced.

> **Problem A** (Rare Order): Summary of computational aspects.
>
> Let R be the total order induced by the collating sequence that was used to sort the input file. It is R that has to be reconstructed from the input.
>
> **Algorithm** For two strings s and t, the following holds: either (a) one string is a prefix of the other, or (b) for some index k we have $s[i] = t[i]$ for all $i < k$ and $s[k] \neq t[k]$. Which of (a) or (b) holds and, if (b) holds, what the index k is, can be determined independent of R. The required procedure is a variant to determining the lexicographic order of two strings.
>
> Therefore, each pair of strings s and t, where s precedes t in the input file and s is not a prefix of t, contributes to our knowledge about R, viz. $s[k]\ R\ t[k]$ (because precedence in the input file is based on the lexicographic order). One needs to consider only neighboring strings in the input file, since other pairs will not provide new information about R.
>
> The input can be scanned once sequentially to collect all information about R, for example, in a $N \times N$-matrix, where N is the number of letters in the alphabet ($N = 26$ always works). For the sample input provided in the specification this matrix might look like
>
> | R | W | X | Y | Z |
> |---|---|---|---|---|
> | W | | | | |
> | X | | | | 1 |
> | Y | 3 | | | |
> | Z | | | 4 | |
>
> where the number n at entry (a, b) indicates that $a\ R\ b$ holds on account of input lines n and $n + 1$. This matrix must completely determine R (as promised in the specification). The collating sequence can now be extracted in several ways (e.g. repeatedly find minimal element and remove it; the example matrix above is atypical in that each column has at most one non-blank entry). Of course, one should

consider only letters that occur in the input. Keeping track of some additional information (e.g. the number of non-blank entries per column) may speed things up.

Special cases Duplicate strings and empty strings may be present in the input file. Their presence may slightly complicate the determination of index *k* mentioned above, but otherwise has no consequences. The input file may consists of the 'end-of-list' marker only, in which case the output should be an empty collating sequence.

Input and output format are straightforward. No special efficiency considerations are required (other than noticing that to try out all collating sequences is hopelessly inefficient).

5 Problem solution

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 Creator, who is intimately familiar with the problem after specifying it, no more than a couple of hours to implement a working solution. That solution should be more or less along the lines of an "intended" solution and should not make use of implementation specific extensions to Pascal. It is not necessary to come up with a very neat and fully annotated solution (in case you still think that would be a waste of time). 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 next section).

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, possibly using some implementation specific extensions to Pascal! You may sometimes be surprised how "good" unintended solutions turn out to be.

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 may assume that input supplied to their programs is in agreement with the specification. The Problem Creator, however, has to see to it that examples and test input data are valid. It is strongly recommended that input validation is done by a program. The input validator may be implemented as a separate program or may be incorporated in an "embellished" solution. N.B. The embellished solution need no longer be suitable for run-time and memory utilization assessment, so keep a copy of the original "bare" solution!

For Problem A (Rare Order) above, the input validator must check that each line but the last contains a string of at most 20 upper case letters and that the last line consists of a single `'#'`. Moreover it must determine that the input file is *sorted* according to a *unique* collating sequence. That is, leaving out the first string `XWY` should be detected as invalid (collating sequence not unique), as is the insertion of `ZXZ` after the first string (input not sorted). For Problem A it is convenient to incorporate the input validator in a solution.

6 Testing

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

If there is not much variety in the required output of a problem (e.g. only yes/no), then a program might "unrightfully" be accepted when it correctly "guesses" the output. Try to avoid this.

The test cases that are applied to exercise a program must allow a good assessment of the program's correctness. It may therefore be necessary to run more than one test case. Often it is convenient to specify the problem such that one input file actually allows a sequence of tests to be applied (this is *not* the case for Problem A above). Don't forget, however, that an execution time-limit must be set for the entire run (default unless otherwise stated).

Include both "normal" and special (boundary) cases in the test data. Determine the typical run-time of each test and mention it in your report to the Head Judge. For Problem A above, one would include the following input files among the tests: containing no strings (i.e., only the end-of-list marker `'#'`); in which all letters are the same; in which all 26 upper case letters appear; in which duplicate strings appear; in which strings that are prefixes of other strings appear; in which the directly deducible ordering relation is not a linear graph (e.g., by prepending the string `XZ` to the sample input: this directly implies the ordering `Z` *R* `W`, which is also obtained by transitivity from the other strings); containing strings of minimal and/or maximal length, differing only in certain positions; where both `X` *R* `Z` and `Y` *R* `Z` can be deduced before `X` and `Y` can be ordered; etc.

In order to generate test cases it is desirable to have a programmed solution for your problem. In order to judge a program it is convenient to have 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. It may be convenient to specify the problem such that each input file produces a unique output file, in which case the evaluator can simply compare two files character by character (this *is* the case for Problem A above). Note, however, that in this case the output must be *completely* specified, taking into account such oddities as trailing blanks, leading zeroes, empty lines, etc. "Normalization" of output text files before doing the output verification partly solves this problem (also see my note *Additional Contest Information*).

Of course, a simple file comparator cannot distinguish between such errors as Wrong Answer and Bad Output Format (but for the time being we do not mind). Currently, the error messages issued by the judges in the Finals when they reject a program are not sufficiently well-defined to be useful for automated judging. In particular, the messages Failed Test Case, Inaccurate Answer, Too Little/Much Output, Bad Output Format, and Check Clarifications are controversial. My suggestion is to ignore these possibilities. The teams will be told that these messages will not be issued. Hence, Wrong Answer covers any error not covered by Syntax Error, Run-Time Error, and Time-Limit Exceeded.

To get a good insight into the severity of the test cases it may be necessary to write yet another program that collects some statistics (to determine which situations occur how often). Like the input validator, this program could be a separate program or it could be incorporated in an "embellished" solution. It is sometimes also advisable to write a program that generates (possibly random) test cases with certain predetermined characteristics.

7 Summary

Aim at eight or nine original problems. Include two easy problems and one or two more difficult problems. Avoid too difficult problems. Cover various computational aspects. The best team should be able to solve (almost) all problems in the available time.

Each problem should fall under the responsibility of one person. For each problem the Head Judge needs to receive the following items from its Problem Creator:

1. a specification,
2. a short description of the crucial computational aspects,
3. a programmed solution (possibly also alternative solutions),
4. an input validation program,
5. an output verification program (the verifier could be a simple file comparator for deterministic problems, in which case a test output data file must be provided), and
6. a set of test input data files (preferably just one), including their typical runtime and some motivation for their discriminating power.

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

It would be nice if the Head Judge prepares solution suggestions to hand out after the contest. These could, for instance, be based on the "description of crucial computational aspects" as provided by the Problem Creators.

In case you want to organize contests more often, it is advisable to do an evaluation afterwards, in which you attempt to find out how well the objectives have been met. For instance, how difficult did the problems turn out to be, how many and which Clarification Requests were made, how well were the test cases, were any programs accepted unrightfully, were there any programs that got rejected because they failed on a single test case only, etc.