Olympiads in Informatics, 2008, Vol. 2, 64–74
© 2008 *Institute of Mathematics and Informatics, Vilnius*

A Proposal for a Task Preparation Process

Krzysztof DIKS, Marcin KUBICA, Jakub RADOSZEWSKI,
Krzysztof STENCEL

Institute of informatics, University of Warsaw
Banacha 2, 02-097 Warszawa, Poland
e-mail: {diks,kubica,jrad,stencel}@mimuw.edu.pl

Abstract. This paper presents in details the task preparation process in the Polish Olympiad in Informatics. It is a result of over 15 years of experience in organization of programming contests for high-school students. It is also a proposal of best practices that should be applied in a task preparation process for any programming contest. Although many elements of the described process are widely known, the rigorous implementation of the whole process is the key to high task quality.

Key words: algorithmic problem solving, programming contest, informatics olympiads.

1. Introduction

In this paper we present in details the task preparation process used in the Polish Olympiad in Informatics (POI). It represents over 15 years of experience and evolution. Although the general schema of the task preparation process is widely known, its details are not obvious. We believe that rigorous implementation of such a process is the key to assure good quality of tasks. This paper is based on the POI Handbook (The Polish Olympiad in Informatics Handbook, 2008) – an internal document describing task preparation and contest organization (in Polish). It is the result of lessons we have learned during the last 15 years. We hope that the process described here can help avoid mistakes we have once made. Other aspects of the contest organization in the POI are described in (Diks *et al.*, 2007).

Why is the task preparation so important? The answer is: time constraints. A typical programming contest takes 2–5 days. It is a very busy period and many things have to be prepared in advance, including tasks. However, any mistakes done during task preparation are usually discovered during the contest itself, when it is already too late to correct them. Therefore, quality assurance is not a question of saving work, but organizing or not a successful contest.

The structure of the paper is as follows. In Section 2 we give an overview of a task life-cycle. Then, in the following sections we describe consecutive phases of task preparation. Finally, we draw conclusions. Checklists for all the phases of the task preparation can be found in the Appendix.

2. Task Life-Cycle

The task preparation process consists of several phases. The life-cycle of a task is shown in the Fig. 1. Initially, the author formulates an idea of a task together with suggestions how it can be solved. Such ideas are then reviewed, usually by a person supervising the whole process. If the task is suitable, it is formulated. (Task suitability is discussed in Section 3.) The next phase is analysis of possible solutions together with their implementation and preparation of tests. During this phase, after a more thorough analysis, it may also turn out that the task is unsuitable. However, this happens rarely. The last stage prior to the contest is verification. The last check-up should be done shortly before the competition.

In different phases of preparation different people modify the tasks. Storing the tasks in a revision control system is therefore necessary. Otherwise, one can easily collide with changes done by someone else or a wrong version of the task can be used. In the POI a dedicated, web-based system is used to control phases of task preparation and their assignment to people. Each task has a form of a tarball with a specified directory structure and file naming convention. The tasks are checked in and out after every phase or assignment. The system stores all versions of the tasks and detects any collisions in their modifications.

Although we find such a system very useful, we do not argue that development of a task-control system is necessary. What is necessary is to use some revision control system to take care of different versions of the tasks.

3. Task Review

The form in which a task is provided varies a lot depending on the author. Sometimes it has a form of a couple of paragraphs describing a problem and sometimes it is a fully formulated task. In fact, what is required at this stage is a short definition of the algorithmic problem and description of expected solutions. Everything else can be done during the formulation phase.

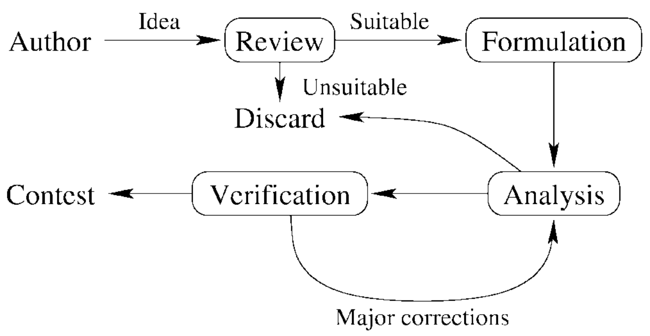

Fig. 1. Task life-cycle.

Task review requires expertise and algorithmic knowledge. The reviewer has to foresee possible solutions that are within contestants' capabilities and their technical complexity. When judging the appropriateness of the task, the following aspects should be taken into account:

- One must be able to formulate the task statement in terms understandable for a secondary-school pupil. Moreover, such a formulation must be clear, comprehensive and not too long. If it is too complicated or requires explanation of many terms, it is not suitable for a competition.
- Is the task a 'handbook' one, i.e., can it be solved by a straightforward application of an algorithm/technique known from handbooks? If so, it would test the knowledge of a particular algorithm/technique rather than creativity. In such a case it is not appropriate.
- The task should be unique to the best knowledge of the reviewer.
- Can the task be solved in a polynomial time? If not, it is very difficult to evaluate it in a reasonable time. However, exceptions are possible.
- The task should neither be too easy nor too difficult. Otherwise, the task will not significantly influence the results. There are no universal guidelines. Our requirements are a little bit higher than those defined in (Verhoeff *et al.*, 2006). The expected knowledge is covered by most general handbooks on algorithms, e.g., (Cormen *et al.*, 1989) (skipping more advanced chapters) covers it all. But in case of doubts, it is better to have a task that is too easy than too hard.
- There should exist many ways of solving a task at different difficulty levels; moreover, it should be possible to distinguish these solutions by testing. Only then the task will differentiate contestants' results.

4. Task Formulation

During task formulation all elements missing in the task idea should be added. In particular: a short story can be added to make the task more attractive. The language should be simple. One should avoid complex sentences. All new notions should be introduced before they are used. Greek symbols should be avoided. If a coordinate system is needed, then the standard one should be used. Other detailed guidelines are consistent with those that can be found in (Verhoeff *et al.*, 2006).

Input and output specifications must be precise – only limits on the data sizes can be left undefined (until analysis). Preferably, the output should be short and hard to guess (e.g., not a yes/no answer, but rather some integer) and unequivocally determined by the input. However, this last requirement is not crucial. If the output is indefinite, a grader program checking correctness of outputs will have to be implemented.

Task formulation should contain an example, preferably with a picture. The task should fit on one or two pages. Three pages are the absolute limit. (However it can happen for interactive tasks, see Section 5.6.). The formulation should be also accompanied by a short description (one or two paragraphs) of author's solutions – it will be taken into account during the analysis. It can be either the original description provided by the author or its edited version.

5. Task Analysis

Task analysis is the most costly phase. Its goal is to provide all task elements necessary at the competition site. These elements include an analysis document, programs and test data. All of them are prepared by one expert from the Jury, who is called the task analyst or simply the analyst. Even if the author of the task provides a description of the solution or even a ready-to-go model solution, the task analyst is obliged to prepare all task artefacts from scratch. The role of the analyst is to invent and code model solutions in all competition languages, slower solutions and expected wrong solutions. Furthermore, he/she writes the analysis report and prepares test data as well as a program which checks the consistency of test cases. Task description can also be altered by the analyst. At least he/she firmly sets the limits on the size of test data, on memory and on the running time. If answers to be produced by solution programs are indefinite for each test data, the analyst should prepare a grader program which checks the correctness of outputs.

To sum up, the task analysis report should contain the following items:

1. An analysis document with description of model, slower and wrong algorithms, test data, and a discussion and settlement of limits.
2. A set of source codes of solutions (model, suboptimal and wrong; in all competition languages).
3. A set of test data (some files plus possibly a generator program).
   a) A program which verifies correctness of test data.
4. A grader program used when outputs for some test inputs are indefinite.
5. Updated task description.

5.1. *The Analysis Document*

The analysis document is an internal document of the Jury so it can be written in a professional language. However, as it is later used as the basis for the publicized solution presentation, it is wise to use a language that is as friendly as it is reasonable within the time frame allotted for the task analysis. The document is to discuss the widest possible spectrum of expected algorithms presented by contestants. This includes the model solution, other possibly slower solutions and a number of incorrect solutions that could be produced by contestants. It should discuss all the solutions proposed by the author of the task, but must not be limited to them. It should be possible for model solutions to be created by contestants. We do not include algorithms which are not in the scope of pupils. Such algorithms can be described in the final post-competition presentation of the task, but they should not influence the grading. The analysis document also sets limits on the size of test data, memory and the running time. These limits are later reflected in the task description but the analysis should contain a study on their choice and the information which discussed solutions meet each particular limit. The latest information is to be accompanied with solution-test case matrix, which for each pair of a solution and a test case tells whether this solution is intended to pass the given test case. The analysis document must also include a list of modifications in the task description in order to preserve the traceability from the author's proposal, through the task formulation to the result of analysis.

5.2. *Solutions*

The model algorithm should be implemented in all competition programming languages, i.e., C/C++, Pascal and Java (the latter was introduced in the POI in 2007). Since in many cases the usage of STL is relevant, a separate C++ solution using STL must be presented during the analysis. It is recommended not to use any programming tricks, especially those which blur the readability of solutions. The readability and maintainability of the presented solutions are crucial issues since the model solutions are used to produce and verify correct outputs and last but not least they are presented to contestants as an example of best programming practices. For example, this means that model solutions must compile in the pedantic mode without even a tiny warning.

Suboptimal solutions are slower than model solutions. They represent possibly the largest set of such solutions which are imaginable to be presented by pupils. They contain some more obvious yet slower algorithms or those which are certainly simpler to code. The same remarks concern expected incorrect solutions. These include incorrect heuristics, solutions which do not cover all cases to be considered or greedy algorithms (if this is a faulty approach for a given task). This category also consists of solutions consuming too much memory. If they are run within the memory limit, they result in a run-time error.

5.3. *Test Data*

The task analyst prepares a set of test data to be used during the competition. Typical test sets consist of 10 to 20 test cases. About additional 5 simple test cases are set up to be electronically available during the contest. The purpose of these tests is to provide real examples so that students can check the basic correctness of their programs using data prepared by the Jury.

The objective of tests is to distinguish correct and incorrect solutions. They should also distinguish all efficiency classes of correct solutions. Optimally, test cases should reflect the conceptual and programming effort needed to produce solutions. In case of doubt, the intended distribution of points should be linear among more and more effective algorithms. Tests should put stress on the asymptotic time-cost rather than absolute running time. In the POI we assume that solutions up to twice as slow as the model solution score the full points. Moreover, the result of testing should not depend on the choice of programming language or usage of STL. 30%–60% of points should be allotted to correctness tests, i.e., correct but inefficient solutions (however running in a reasonable time) should score 30%–60% of points. The particular choice of this limit (between 30%

and 60%) depends on the task – on how the correctness vs. efficiency is important. In this "correctness" part of the test data, the efficiency should count as low as possible – even very slow but correct programs are to score all points below the "correctness" limit. The rest of points should be granted for efficiency. If necessary, tests can be grouped – a solution is granted points for a group of tests only if it passes all the tests from the group. Grouping should be used when the correct result could be 'guessed' with high probability or more than one test is needed to distinguish correct from incorrect solutions.

Test data must conform **exactly** to the specification stated in the task description. It concerns also white spaces and the kind of new line characters (all of them must be ASCII code 10, since we use Linux while grading). Tests can be prepared in the form of files or a generating program can be provided. A hybrid set (test files plus a generator of the rest) can also be used. Generating is especially useful in case of huge efficiency tests. If such a program uses random number generator, it should also set the random seed, so that it always generates exactly the same set of tests. The set of test cases always comes together with a program verifying its correctness. Such a program should verify all conditions defined in the task. It is recommended, however not required, that the verifying program concludes its successful run printing some simple statistics like the size of the input data in terms defined in the task description (e.g., `OK n=15 k=67`).

5.4. *The Grader*

The last element of the result of the task analysis is required only for tasks where the output for a given test is not fixed, i.e., where for a given input data there can be a number of correct outputs. For such a task, a so called *grader* is prepared. A grader is a program with specific interface suited for communication with the grading system. It accepts three parameters which are names of files: input file, contestant's output file and "*hint file*". Hint files usually contain the most important and definite part of the solution, e.g., length of the expected output sequence. Particular sequences may vary, while all correct sequences are of this given length. Given these three arguments, the grader checks the correctness of contestant's output data and produces the grading report possibly with an error message if the encountered output is wrong.

5.5. *Output-Only Tasks*

If the task is an output-only one, the set of tests should be prepared in a similar way as for batch tasks, however we cannot control the running time. Contestants can even use separate programs to produce outputs for different test cases. We cannot measure efficiency of contestants' solutions. However, in this type of tasks the running time is not so crucial or the competition time is a sufficient limit. So, the implementation of all correct solutions in all contest programming languages is not necessary.

5.6. *Interactive Tasks*

The most common kind of task is the batch task, where contestants' programs are to read input data and produce appropriate output data. However, there are also tasks having form of games, on-line problems or optimisation problems which consist in minimizing the number of calls issued by a solution program. Such tasks are called *interactive*.

In the POI interactive tasks are formulated so that a solution is supposed to interact with the grading library it is compiled with. A contestant's program calls grading library functions. Such a communication mode is the most convenient from the point of view of contestants. However, it requires caution from the task analyst.

The description of an interactive task must contain very precise specification of grading library functions and the exact guidelines how to compile a solution with the library and a list of statements which must be included in the solution (e.g., `#include` directives). This information should be included for all contest programming languages. The task description must also include example interactions for all languages.

The task analyst implements the grading library for all languages (usually for C and C++ it suffices to produce one library). Its interface should be as simple as possible, e.g., it must not require an initialisation call. Grading libraries should be immune to communication patterns which do not conform to the task description. Furthermore, to avoid any naming clashes, the grading library should not export entities other than those specified in the task description. If there are many playing strategies, the grading library should be able to use the smartest one as well as some less optimal. At the beginning of the interaction the grading library reads the information about the test case from a file. For example, it can be the description of the playing board and the strategy to be used by the library. Then, the grading library interacts with the contestant's solution and eventually prints the report on the results of the interaction. If something goes wrong (incorrect or malicious behaviour of the solution), the library can break the interaction during execution.

The task analyst also provides a toy version (in all languages) of the grading library for contestants. They can download it and test their programs with it during the competition. This will ease the construction of formally correct solutions and assist in avoiding foolish errors. The task description must explicitly warn that these are only toy libraries.

It must be taken into account that contestants might try to reverse-engineer the toy library to uncover its internal data structure. Therefore, the real grading library should use different data structures. It can also hide its internals during run-time. For example, if it is obvious from the task description that the number of moves is counted, the grading library can start the counting from a bigger number and count the number of moves multiplied by a constant (or some other more sophisticated encryption can be used). This way, even a smart contestant's memory sniffer is unable to find the location of the variable which ticks on each move.

Solutions provided by contestants are not allowed to output anything. The standard output is dedicated to the grader library and its final message on the result. However, the library should be protected against illegal contestant output. All its messages are simply surrounded by some magic code unknown to contestants (of course it is not produced by the toy version of the library).

Above we collected a number of guidelines which make solving and grading interactive tasks easier. Contestants are provided with precise specifications, examples and downloadable code. This minimizes the number of mistakes in their solutions. On the other hand, the grading process is protected against most of imaginable attacks and blameless misuses. This aids fair grading. Interactive tasks are still rare. However, our experience proves that properly prepared (e.g., according to above mentioned best practices) interactive tasks can be successfully used in programming contests.

6. Task Verification

All actions performed during the verification process are focused on checking correctness of artefacts which were prepared during previous phases. The inspection should cover: task formulation, the analysis document, model solutions, programs for test generation and verification of the test cases and the test cases themselves.

Task description should be investigated thoroughly to ensure that there are no unclear statements, the limits for input data and the memory limit are stated (it should be checked whether the output can be clearly determined for the border cases of the input), and that the sample test conforms to the figure (if it is present). Additionally the description should be spell-checked.

The main goal of the remaining checks is to verify correctness and accuracy of the test cases, since any error in test data revealed once the competition has started is really disastrous. Thus, another program for input verification should be implemented from scratch. To verify the output files, an independent model (but not necessarily optimal) solution should be prepared. All model and incorrect solutions should be evaluated on all the test cases to ensure that classes of solutions of interest are distinguished properly. It should be also checked whether there exists either a simple solution which performs better than the most efficient model solution from the task analysis or a solution with significantly worse asymptotic time which scores too well on the test cases. If necessary, some test cases may be added or changed to correct all mistakes found.

A separate document describing the verification process should also be prepared. It should contain: a list of all performed activities, a short description of the alternative solution implemented during the verification, a list of all mistakes that were found in the task analysis, and a list of all modifications performed in the task description and the analysis document.

7. Preparation of the Task for the Competition

This phase is performed after the task is qualified for the actual competition, therefore in some cases it can be a part of the task verification process. It includes some minor changes in the task formulation. A more representative header is set – it includes the logo of the contest and some information about the date, stage and day of the competition. Also some last-moment simple checks should be performed, like verification of input data limits and sample test data correctness.

8. Conclusions

In this paper we have described in details task preparation process as it is applied in the POI, with focus on less obvious details and quality assurance. There are so many details that should be taken into account that we found it necessary to write (The Polish Olympiad in Informatics Handbook, 2008) for members of the POI Jury team. The issue of quality of tasks in IOI is raised from time to time. Since the task preparation criteria seem to be universal, the guidelines stated here should also be applicable to other programming contests. Therefore, we hope that this paper can be of use for organizers of various programming contests.

9. Appendix. Checklists

9.1. *Task Formulation:*

- Is the task description prepared according to the template style?
- Is the source file of the document formatted appropriately (indentation etc.)?
- Is the input and output format specified clearly?
- Is a sample test case prepared?
- Is a figure depicting the sample test case prepared (if applicable)?
- Is the author's solution description present?
- Does the task description (excluding the author's solution description) fit on at most two pages?
- Has the document been spell-checked?

9.2. *Task Analysis:*

- Is the model solution implemented?
- Does the model solution use STL? (Yes/No)
- Is the model solution implemented in all required programming languages?
- Is at least one less effective solution implemented?
- Are the less effective solutions implemented in all required programming languages?
- Is at least one incorrect solution implemented?
- Is the program verifying tests created?
- Is the task output uniquely determined for every possible input? (Yes/No)
- Is the grader created (if the task output may not be unique for some input)?
- Are the tests created?
- Is the test-generating program implemented (if the total size of tests exceeds 1 MB)?
- Are the example test cases for contestants created?
- Are the limits on input data size stated in the task description?
- Are the memory (and time) limits stated in the task description?

- Is the analysis document prepared?
- Is the model solution described in the analysis document?
- Are the less efficient solutions described in the analysis document?
- Are the incorrect solutions described in the analysis document?
- Does the analysis document contain rationale for the chosen input data and memory limits?
- Are the test cases described in the analysis document?
- Does the analysis document contain a table listing which solutions should pass which test cases?
- Are the changes in the task description listed in the analysis document (if applicable)?
- Have the task description and the analysis document been spell-checked?

9.3. *Task Verification:*

- Is the verification document prepared?
- Is an alternative model solution implemented?
- Are all conditions from the "Analysis" checklist fulfilled?
- Are the example input and output correct?
- Does the figure correspond to the example input and output (if applicable)?
- Are the limits for the input data specified correctly and clearly?

9.4. *Preparation for the Competition:*

- Is the task description header properly prepared (including competition stage, day and dates)?
- Does the time limit correspond to the speed of computers actually used for evaluation?
- Have all the auxiliary comments been removed from the task description?

References

Cormen, T.H., Leiserson, C.E. and Rivest, R.L. (1989). *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company.

Diks, K., Kubica, M. and Stencel, K. (2007). Polish Olympiad in informatics – 14 years of experience. *Olympiads in Informatics,* **1**.

POI Handbook (2008). *The Polish Olympiad in Informatics Handbook*, v.0.10.0. POI internal document (in Polish).

Verhoeff, T., Horváth, G. , Diks, K. and Cormack, G. (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science,* **IV**(I).

K. Diks (1956), professor of computer science at University of Warsaw, chairman of the Polish Olympiad in Informatics, chairman of the IOI'2005.

M. Kubica (1971), PhD in computer science, assistant professor at Institute of Informatics, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, scientific secretary of Polish Olympiad in Informatics, IOI-ISC member and former chairman of Scientific Committees of BOI'2008 in Gdynia, IOI'2005 in Nowy Sacz, CEOI'2004 in Rzeszów and BOI'2001 in Sopot, Poland. His research interests focus on combinatorial algorithms and computational biology.

J. Radoszewski (1984), 5-th year computer science student at Faculty of Mathematics, Informatics and Mechanics of Warsaw University, member of the Main Committee of Polish Olympiad in Informatics, former member of Host Scientific Committee of IOI'2005 in Nowy Sacz.

K. Stencel (1971), PhD hab. in computer science, at the moment works at the Faculty of Mathematics, Informatics and Mechanics of Warsaw University. His research interests are connected with non-relational databases. From 1995 he has been the chairman of the jury of Polish Olympiad in Informatics. He was also the chair of jury at CEOI'97, BOI'2001, CEOI'2004, IOI'2005 and BOI'2008.