Olympiads in Informatics, 2008, Vol. 2, 171–180

On Using Testing-Related Tasks in the IOI

Ahto TRUU, Heno IVANOV
Estonian Olympiad in Informatics
Tähe 4-143, EE-51010 Tartu, Estonia
e-mail: ahto.truu@ut.ee, heno@siil.net

Abstract. One major area of software development that is mostly neglected by current computer science contests is software testing. The only systematic exception to this rule known to us is the Challenge round in the TopCoder contest. In this paper, we propose some patterns for designing tasks around testing theory and examine their suitability for being blended into the existing IOI competition and grading framework. We also present some case studies of using testing-related tasks on actual contestants.
Key words: programming contests, task development, software testing.

1. Introduction

Most computer science contests focus on the "programming" part of the software development process. This means the contestants are given a computational problem and are asked to find or invent an algorithm for solving it and to implement the algorithm in one of the programming languages supported at the particular contest. The programs are then graded on a predefined set of (secret) test inputs and awarded points for every input (or set of inputs) for which they produce correct output, without exceeding some predefined resource constraints; typically there are limits on the total CPU time and the maximum amount of RAM used by the program in a test run.

Several members of the IOI community have pointed out that this approach is too narrow and in particular only rewards testing as much as it has an effect on the quality of the final program submitted for grading. Due to the limited time in which the contestants have to develop their solutions (typically five hours for solving three problems), they are not able to perform systematic testing of their programs in addition to the other work. It is therefore only natural that they tend to focus on the areas where their effort is more directly rewarded.

It has been suggested by several authors (Cormack *et al.*, 2006; Opmanis, 2006) that the IOI community should consider bringing testing theory and its applications more directly into the competition material. In this paper we examine the main modes of operation of software testers and consider the suitability of each one as the basis for competition tasks, with the aim of making testing activities the central point of the task. We also provide several case studies of inflicting explicitly testing-related tasks upon actual contestants: two tasks from BOI (Baltic Olympiad in Informatics) and one from a training camp for the finalists of our national competition

2. Taxonomy

The two main attributes for classifying testing activities are black box versus white box methods and static versus dynamic testing (Patton, 2006, pp. 51–122).

In black box testing techniques, the testers do not know the internal implementation details of the software they are testing and rely exclusively on external interfaces. In white box (sometimes also called clear box or glass box) techniques, the testers have access to the implementation details of the software and actively use this knowledge to tailor the test cases.

In static testing, the testers examine the software without actually running it (one may argue that "analysis" is the proper term here instead of "testing"), whereas the dynamic approach is based on observing (and possibly influencing) the program while it executes.

2.1. *Static Black Box Testing*

Having no access to the implementation details of the software and no ability to run it either is effectively equivalent to not having the software at all.

In real-life software projects, the testing team uses the requirements documents to define the test plan to be executed when the actual software arrives. This is also the mode in which competition task developers normally operate: they must create the test data based on the task description alone, not having access to the solutions that the test data will be used on.

Since it is impossible to have an exhaustive test data set for any realistic problem, the domain of input data is usually split into a limited number of equivalence classes and the program is tested on just one instance (or at most a few instances) of each class. The main objective is to achieve as complete as possible coverage of the problem domain using limited number of test cases.

To evaluate the test data created by contestants in such a setting, the grading server could have a set of solutions, probably one of them correct and the rest somehow flawed – either producing wrong answers, violating some resource constraints or terminating abnormally under certain conditions. It may also be possible to merge some (or even all) the flawed solutions together into one that exhibits all the problems.

The grading would then consist of running the solutions on the test data provided by the contestant and checking which of the faults are triggered and which are missed. Of course, merging several flaws into one implementation brings the risk that the flaws will interfere with each other and care has to be taken to properly track which one was actually triggered by any given test case. We will discuss additional scoring details in Subsection 2.5.

2.2. *Dynamic Black Box Testing*

In dynamic black box testing, the tester has the possibility to run the software, but no knowledge of its implementation details. This is the mode in which integration testing

usually works in major software projects. Independent software verification and validation (especially in the sub-domain of security analysis) is also often done in this mode, since the vendor of the software may not be co-operating.

In a competition, this mode could be implemented by giving the contestants access to an implementation with known defects. To maintain the black box status, the implementation should obviously be given in binary form only. To prevent reverse engineering (as opposed to pure black box testing), it may be even better to expose the flawed implementation as a service on the grading server such that the students can submit inputs and receive outputs, but not interfere otherwise.

Giving out several binaries with only small differences would help the contestants focus their reverse engineering efforts on the areas where the implementations differ. Consequently, this should probably be avoided, unless the implementations have major differences in the overall structure that would render them resistant to differential analysis. Thus, the network service access would be the technically preferred way to provide several slightly different versions, each one with a distinct defect.

The scoring of the test data produced could be similar to the case of static black box testing. A grading possibility specific to this task type only is to give the contestants a binary with known defects and ask them to implement a "bug-compatible" version. This approach was used in BOI'98, as reported in Subsection 3.2.

2.3. *Static White Box Testing*

Static white box testing should involve examining the internals of the software without running it. When trying to come up with tasks based on this approach, the first concern is keeping the testing static. After all, the contestants have programming tools available and know how to use them, which means they can easily turn the static testing exercise into a dynamic one!

There are two major kinds of activities in practical software development that can be classified under this heading: code reviews and program analysis. Code review is a team effort and results in a human-readable document (most of time in the form of annotations on top of existing source code). As such, it is probably the least suited as a basis for an IOI-style task. Perhaps we could invent an annotation language for marking up existing code in some machine-readable form, but this would probably get too heavy to fit into the hour and a half that a contestant has per task.

On the other hand, asking the contestants to create a simple program analysis tool can be just the approach to keep them from rewriting any human-readable presentation of the algorithm from the task text into an executable implementation. As with the black box testing case, the key is that the actual code is not available until grading time. This approach was used in BOI'95, as reported in Subsection 3.1.

2.4. *Dynamic White Box Testing*

Dynamic white box testing means the tester is free to run the software and also has access to the implementation details. A natural way to achieve this setting in the contest environment could be to provide the software to test in source code form. To avoid bias between

contestants using different languages, either equivalent source should be provided in all languages, or the program could be given in some pseudo-code.

This approach was tried in a recent training camp, as reported in Subsection 3.3.

2.5. *Scoring*

There are some scoring considerations common to all of the task types mentioned above, involving two main aspects of test data quality: correctness and completeness.

On the correctness level, the test cases could be rated:

- well-formed when the file format meets the conditions set in the task description (for example, a line of 200 characters where the task description allowed for no more than 100 characters would be malformed);
- valid when the file is well-formed and the semantic value of its contents matches the conditions given in the task description (for example, presenting an unconnected graph in an input file would make the file invalid if the task called for connected graphs only);
- correct when both the input and output file are valid and agree with each other (a test case consisting of a valid input file and a valid output file which do not agree with each other would still be incorrect).

Obviously, no points should be awarded for any test cases other than correct ones. However, there are still several possible ways to handle the rest:

- The grading server could validate all proposed test cases as they are submitted and accept only the correct ones.
- The grading server could accept all test cases initially, but filter out the incorrect ones as the first step of the grading process. If the contestant is allowed to submit only a limited number of test cases for the task, this already is a small penalty in that the incorrect cases take up slots that could have been used for correct ones.
- The grading server could also assign explicit penalty points for the incorrect test cases submitted.

It is probably sensible to check for the format of the submitted files in any case, to weed out silly mistakes like submitting an output file in lieu of an input file or vice versa, but the decision to reject or accept invalid or incorrect cases would probably depend on the particular task.

There are also several possible ways to assign score for the correct test cases. Since the main goal besides correctness of every test data set is completeness, the following approaches come forward:

- To award some points for each test case that causes failure in at least one of the faulty programs. Of course, we need to limit the number of test cases a contestant can submit in order to have an upper limit on the value of the task. The problem with this approach is that it would grant full score to a test data set that makes the same faulty program fail in every test case. Certainly, this is not good coverage.
- To overcome the problem with the previous approach, we could instead award some points for each faulty program that fails on at least one test case submitted by the

contestant. That would put an automatic upper bound on the value of the task, as there would be a fixed number of faulty solutions that could fail. But still this approach would probably result in a lot of contestants just submitting the maximal allowed number of randomly generated large test cases in the hope that there will be something in them to trigger all the faults. It would be very hard to avoid them getting high scores with no actual design in any of the test cases.

- To further improve the grading accuracy, we could divide the faulty solutions into clusters based on the passed/failed patterns. If we then award points based on the number of clusters we get, that would encourage the students to carefully design the test cases to isolate specific faults. It would probably depend on the task details whether it would be better to collect into one cluster all the solutions that pass exactly the same set of test cases or whether we should also consider how they fail on the cases that they do not pass. The distinct failures could include abnormal termination, exceeding time or memory constraints, producing wrong answers. Sometimes perhaps even different wrong answers could be counted as different faults.

One issue that should be considered in the context of a large event like IOI is the computation power needed to grade these types of tasks. It takes $\mathrm{O}(N \cdot K)$ time to evaluate the solutions of N contestants to a batch task that has K test cases, but it will take $\mathrm{O}(N \cdot K \cdot M)$ time to evaluate the solutions to a testing task where each contestant can submit K test cases on which M different implementations have to be tested. Also the grading reports may get excessively long if their layout is not carefully considered.

3. Case Studies

3.1. *Task IFTHENELSE, BOI'95*

This is a program analysis task that was proposed for the 1995 Baltic Olympiad in Informatics by Rein Prank from Tartu University.

The task considered a simple programming language consisting of IF/THEN/ELSE statements with only comparison of two integer variables for equality as the conditions and asked the contestants to write a tool that would analyze a program written in that language, report all possible outcomes, and for each outcome also a set of initial values for the variables to produce that outcome. (The complete original text of the task is given in Appendix A.)

The grading was done on 11 independent test cases valued from 2 to 5 points, for a total of 50 points for the task. As can be seen from Fig. 1, the task resulted in quite good distribution of scores among the 14 participants of the event.

3.2. *Task RELAY, BOI'98*

This is a task that was proposed for the 1998 Baltic Olympiad in Informatics, again by Rein Prank.

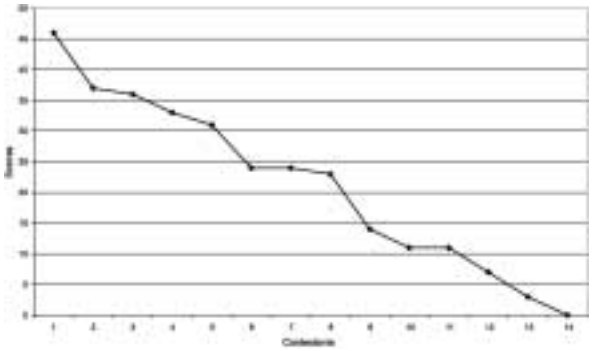

Fig. 1. Distribution of scores for the task IFTHENELSE.

The task described a sufficiently intricate data filtering and sorting problem on racing times (see Appendix B for the full text). The contestants were asked to develop a "bug-compatible" version of a binary that contained several classic mistakes:

1) using a strict inequality when a slack one should have been used in filtering;
2) sorting by the wrong key;
3) ignoring the seconds when checking that a time value does not exceed 2 hours;
4) forgetting to carry from seconds to minutes when subtracting time values.

The grading was done using 7 "positive" and 4 "negative" test cases. For each "positive" test case (that is, where the given implementation worked correctly), the contestant received 2 points if their solution produced the same (correct) answer as the given program. For each "negative" test case (that is, where the given implementation worked incorrectly), the contestant received 4 points if their solution produced the same (incorrect) answer as the given program, 2 points if their solution yielded an incorrect answer different from the one produced by the given program, and no points if their solution produced a correct answer (this indicated a fault in the given program that the contestant did not detect). Also, to prevent random output from scoring half the points for the "negative" test cases, a contestant's solution that failed in more than one "positive" test case would be denied the 2 point scores for the "negative" cases. As can be seen from Fig. 2, this task also turned out a good distribution of scores.

3.3. *Task QUICKSORT, EOI'08 Training Camp*

This task was devised by ourselves for a practice session held for the finalists of the Estonian Olympiad in Informatics.

The students were given a correct implementation of the QuickSort algorithm and several modifications where strict inequalities had been replaced by slack ones and vice versa (see Appendix C for full text of the task). They were then asked to develop test data that would enable them to tell these versions apart from each other by looking only at the test failure patterns.

In Fig. 3, the line "Tests" shows for each contestant the number of test cases that triggered at least one fault, the line "Faults" shows the number of faults triggered by at

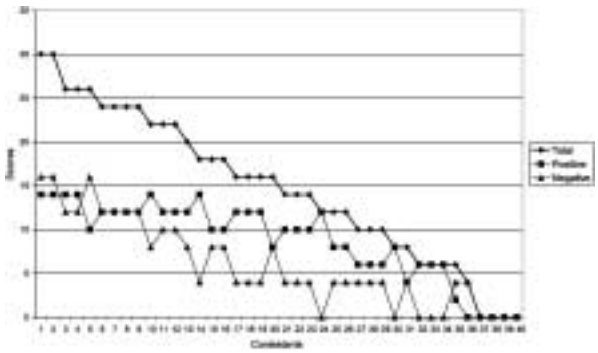

Fig. 2. Distibution of scores for the task RELAY.

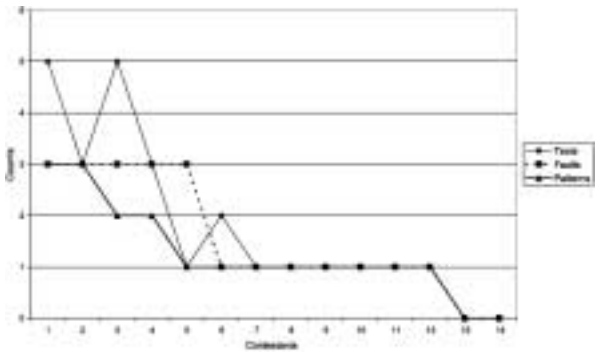

Fig. 3. Results for the task QUICKSORT.

least one test case, and the line “Patterns” shows the number of distinct passed/failed patterns yielded by the set of test cases.

The results of this experiment are probably not directly comparable to the two previous case studies. The average experience level of the participants was significantly below that of the BOI contestants, the session was the last one in the evening of a rather long day, and the students knew their work would not be graded competitively.

Additionally, for some of the participants, the session was the first time they had to perform significant part of their work outside the IDE on Linux. In fact, the main goal of the session from the viewpoint of preparing the future BOI/IOI team members was to get them comfortable setting up shell scripts for quickly running their solutions on several test cases.

4. Conclusions

As can be seen from the above case studies, it should be possible to set up good IOI-style tasks based on any of the main modes of operation observed in real-life quality assurance

teams. We have seen successful examples derived from both black-box and white-box, as well as from both static and dynamic techniques.

We have also seen a somewhat less successful example, which only confirms the obvious: even though tasks can be created based on any area of software testing, care must be taken to ensure the task matches the experience of the contestants and the time available for solving the problem posed to them.

Appendix

A. Task IFTHENELSE

Any line of the program in the programming language IFTHENELSE begins with the line number and contains only one command. The variables are of an integer type, the lowercase letters are used as their identifiers.

The input file presents a subprogram (number of lines $\leqslant 20$) that computes one integer value and contains only the lines of the following syntax:

```
<line number> IF <ident>=<ident> THEN <line number> ELSE <line number>
<line number> RETURN(<integer>)
```

where the command RETURN finishes the execution of the subprogram and returns the integer as a value.

Find all the possible results of the execution of the subprogram. Write each of them only once together with such values of all the variables of the subprogram that bring to this result.

EXAMPLE.

The file ITE.DAT contains the subprogram

```
11 IF a=b THEN 12 ELSE 15
12 IF b=c THEN 13 ELSE 15
13 IF a=c THEN 14 ELSE 16
14 RETURN(14)
15 RETURN(15)
16 RETURN(77)
```

The answer:

```
14:
a=1, b=1, c=1
15:
a=1,b=2,c=3
```

B. Task RELAY

A young programmer has written software for orienteering competitions. The file RELAY2.EXE contains his program for ranking the participants of second relay by their in-

dividual results. As input data, the program uses the files START.IN and FINISH.IN presenting some parts of Start and Finish protocols containing all the participants of second relay and possibly some others. First line of both files contains the number of competitors in the file. Each of the remaining lines consists of the number of the competitor and his (her) starting/finishing time (hours, minutes, seconds), in order of starting/finishing. The participants of first relay may have the numbers 100, 101, 102, . . ., 199; the participants of second relay the numbers 200, 201, 202, . . ., 299 etc. Maximal possible number is 499.

EXAMPLE.

```
START.IN                 FINISH.IN
 6                        5
203   13   12    7        104   13   48   59
201   13   12   10        201   13   52   40
305   13   15    8        305   13   53    1
202   13   24   31        202   13   59   47
204   13   48   59        203   15   25   21
301   13   52   40
```

The output file RELAY2.OUT must contain the numbers of the participants of second relay having received positive result (i.e., running time not more than 2 hours), ranked by their individual results. If the results are equal then the participant having finished earlier must be higher in the table. In case of our example the output must be

```
202
201
```

The program RELAY2.EXE is not completely correct. Your task is to test the program, to diagnose the mistakes in it and to write in your programming language a program MYRELAY that makes the same mistakes as the prototype. Your program will be tested with some positive tests (where RELAY2 computes correct results) and some negative tests (where the output of RELAY2 is not correct). Full points for a positive test will be given if your program gives correct output. In case of negative test you get full points if your program gives the same output as RELAY2 and half of the points if your output has correct format and is wrong but different from the output of RELAY2. You get no points for a negative test where your program computes the correct result (this indicates an error in RELAY2 that you did not detect). The half-points will be given only in the case if your program fails not more than one time with positive tests.

Your program must not incorporate the original RELAY2.EXE nor any part of it. It is also forbidden to call RELAY2.EXE from your program. If such violation of the rules will be detected by the judges, your score for the entire problem will be 0.

All the test cases contain only correct (i.e., possible in real competition) data. All participants of second relay in FINISH.IN occur in START.IN, but some participants having started can be not in Finish protocol. In all test cases the output of RELAY2.EXE has right format, i.e., contains one integer on each line. In all test cases the correct output and the output of RELAY2.EXE contain at least one and not more than 100 participants.

C. Task QUICKSORT

Consider the following implementation of the QuickSort algorithm:

```
1.    procedure qs(var a : array of integer; l, r : integer);
2.    var i, j, x, y: integer;
3.    begin
4.      i := l; j := r; x := a[(l + r) div 2];
5.      repeat
6.        while a[i] < x do i := i + 1;
7.        while x < a[j] do j := j - 1;
8.        if i <= j then begin
9.          y := a[i]; a[i] := a[j]; a[j] := y;
10.         i := i + 1; j := j - 1;
11.       end;
12.     until i > j;
13.     if l < j then qs(a, l, j);
14.     if i < r then qs(a, i, r);
15.   end;
```

Create a set of test cases that is able to distinguish between the following variations:
1) the above (correct) implementation;
2) the '<' on line 6 is replaced by a '<=';
3) the '<' on line 7 is replaced by a '<=';
4) the '<=' on line 8 is replaced by a '<';
5) the '>' on line 12 is replaced by a '>='.

References

Cormack, G., Munro, I., Vasiga, T. and Kemkes, G. (2006). Structure, scoring and purpose of computing competitions. *Informatics in Education*, **5**(1), 15–36.
Opmanis, M. (2006). Some ways to improve olympiads in informatics. *Informatics in Education*, **5**(1), 113–124.
Patton, R. (2006). *Software Testing.* Sams Publishing.

A. Truu is a software architect with GuardTime AS. He has been involved in programming competitions since 1988, first as a contestant and later as a member of the jury of the Estonian Olympiad in Informatics as well as a team leader to the Baltic, Central European and International olympiads, and the coach of Tartu University's team to the ACM ICPC.

H. Ivanov is a software developer with AS Logica Eesti. He has been to several different international competitions (BOI, CEOI, IOI, ACM ICPC) both as a contestant and as a team leader. He headed the team that created the grading system for the BOI'03 in Tartu, and has since maintained it for use in the Estonian Olympiad in Informatics.