

# On suitability of programming competition tasks for automated testing\*

Michal Forišek

Department of Informatics

Faculty of Mathematics, Physics and Informatics

Comenius University, Bratislava, Slovakia

## Abstract

Nowadays, most of the programming competition use some kind of automatic evaluation of contestants' solutions. While this approach is clearly highly efficient, it also has some drawbacks. Often it is the case that the test data isn't able to "break" all flawed solutions. In this article we show that this situation may occur due to two different reasons, show how to recognize these "dangerous situations" and discuss other possibilities of doing the evaluation.

## 1 Introduction

Competitions similar to the International Olympiad in Informatics (IOI) and ACM International Collegiate Programming Contest (ACM ICPC) have been going on for many years. In the ACM ICPC contest model the contestants are given feedback on the correctness of their solutions during the contest. Due to a vast amount of submitted solutions this has to be done automatically. Also the IOI adopted the model of automatically testing the contestants' solutions, with a small twist: solutions get partial score for solving each of the test inputs prepared before the competition.

We will now describe this canonical IOI scoring model in more detail. Each of the tasks presented to the contestants is worth 100 points. The author of the task prepares his own solution and a set of test inputs and correct outputs for these inputs. The 100 points are distributed among the test cases. After the contest ends, each of the contestants' programs is compiled and run on each test case. For each test case the program solves correctly (and without exceeding some enforced limits) the contestant is awarded points associated with that test case. This testing model is commonly known under the name *black-box testing*.

---

\*This work has been partially funded by the Slovak Informatics Society (SISp) and it received a partial financial support from the IOI budget.

We now answered the question “How?”. However, an even more important question is “Why?”. What are the goals this automated testing procedure tries to accomplish? After many discussions with other members of the IOI community our understanding is that the main goals are:

1. Contestants are supposed to find and implement a **correct algorithm**. I.e., their algorithm is supposed to correctly solve any instance of the given problem. A contestant that found and implemented any reasonably efficient correct algorithm should score more than a contestant that found an incorrect algorithm.
2. The number of points a contestant receives for a correct algorithm should be proportional to **its asymptotic time complexity**. Several points should be deducted for small mistakes (e.g., not handling border cases properly).

(As a side note, this is just a general rule. There may be different task types, e.g., open data tasks or optimization tasks, where a different scoring schema has to be used. However, the points mentioned above apply to a vast majority of competition tasks.)

The canonical way to reach the above goals is a careful preparation of the test inputs. The inputs are prepared in such a way that any incorrect solution should fail to solve most, if not all of them. Different sizes of test inputs are used to distinguish between fast and slow correct solutions.

Well, at least that’s the theory. In practice, sometimes an incorrect solution scores far too many points, sometimes an asymptotically better solution scores less points than a worse one, and sometimes a correct solution with a minor mistake (e.g., a typo) scores zero points.

In this article we document that these situations do indeed occur, try to identify the various reasons that can cause them and suggest steps to be taken in future to solve these problems.

In the next section we present some problematic IOI-level tasks from the recent years.

## 2 Investigating recent IOI-level tasks

To obtain some insight into the automated testing process we investigated all competition tasks from IOIs 2003, 2004, and 2005 (USA, Greece, and Poland). Our goal was to check whether there is an **incorrect** solution that’s *easy to find, easy to implement* (in particular, easier than a correct solution) and *scores an inappropriate amount of points* on the test data used in the competition.

For many of these tasks we were able to find such solution(s). The results of this survey are presented in Table 1, some more details are in Appendix A.

(In addition, both the table below and the Appendix A contain a task “safe” that was already used on several IOI-like and ACM-like contests. The main reason for including this task in the survey results was to illustrate that sometimes an unintended solution can easily achieve a full score.)

In most of the cases we are aware that solutions similar and/or identical to the ones we found were indeed submitted by the contestants. Sometimes, we are also aware of correct solutions that scored much less.

<b>task</b>	<b>algorithm type</b>	<b>points</b>
IOI 2004: artemis	brute force, terminate on time	80
IOI 2004: farmer	greedy	90+
IOI 2004: hermes	“almost” greedy	45
IOI 2005: rectangle	symmetrical strategy	70
ACM ICPC: safe	pruned backtracking	100

Table 1: Task survey results.

As we show in Appendix A, there are two different reasons behind these unpleasant facts. For some of these tasks the test data wasn’t designed carefully, but the rest of these tasks wasn’t suitable for black-box testing at all! In other words, it was impossible to design good test data for these tasks.

After writing most of this article we became aware of the fact that [7] carried out a more in-depth analysis of one of the tasks we investigated (Phidias, IOI 2004) and managed to show that also the test data for this task allowed many incorrect solutions to score well, some of them even got a full score.

Together, these results show that this issue is quite significant and we have to take steps to prevent similar issues from happening in the future.

### 3 Tasks unsuitable for black-box testing

We would like to note that it can be formally shown that some interesting algorithmical tasks are not suitable for automatic IOI-style testing. In this section we present two such tasks and discuss why they aren’t suitable.

As a consequence, this means that while the current model of evaluation is used at the IOI, there will be some algorithmical tasks that can’t be used as IOI problems. In order to broaden the set of possible tasks a different testing procedure would have to be employed.

#### Planar graph coloring

Given is a planar graph, find the smallest number of colors sufficient to color its vertices in such a way that no two neighbors share a color.

Regardless of how the test data is chosen, there is a simple and wrong solution that’s expected to solve at least half of the possible inputs: Check the trivial cases (1 color: discrete graph, 2 colors: bipartite graph). In the general case, flip a coin and output 3 or 4.

The Four-color theorem [1] guarantees that the answer is always at most 4. Thus when the answer is 3 or 4, our solution is correct with a 50% probability. Thus, this algorithm is expected to solve at least 50% of any set of test inputs correctly.

## Substring search

Given are two strings, *haystack* and *needle*, the task is to count the number of times *needle* occurs in *haystack* as a substring.

There are lots of known linear-time algorithms solving this task, with KMP [6] being probably the most famous one. If we use  $n$  and  $h$  to denote the length of *needle* and *haystack*, the time complexity of these types of algorithms is  $O(n + h)$ . These algorithms are usually quite complicated and error-prone.

The problem is that there are simple but incorrect algorithms which are able to solve almost all possible inputs.

As an example, consider the Rabin-Karp algorithm [5].

In its correct implementation, we process all substrings of *haystack* of length  $n$ . For each of them we compute some hash value. (The common implementation uses a “running hash” that can be updated in  $O(1)$  whenever we move to examine the next substring.) Each time the hash value matches the hash of *needle*, both strings are compared for equality.

The worst-case time complexity of this algorithm is  $O(n(h - n))$ , but its expected time complexity is  $O(h + n)$ .

The algorithm as stated above is correct. However, there is a “relaxation” of this algorithm that is even easier to implement, and guaranteed to be  $O(h + n)$ : Each time the hash value matches, count it as a match.

Note that while this algorithm is incorrect, it is very fast, and it is very highly improbable that it will fail (even once!) when doing automated testing on a small set of test inputs. Moreover, it is impossible to devise good test data beforehand, as the goodness of the test data depends on the exact hash function used.

We would like to stress that in practice programs that misbehave only once in a large while are often one of the worst nightmares, as testing a program for such bugs is almost impossible and the bugs may have a critical impact when they occur after the program is released. By allowing such solution to achieve a full score in a programming contest we are encouraging students to write such solutions. This may prove to be fatal in their future career.

## 4 Heuristic methods for recognizing tasks unsuitable for black-box testing

Here we give a short summary of the guidelines informally published after IOI 2004 in [3].

The tasks that are unsuitable for black-box testing usually exhibit one or more of the following properties:

1. The set of correct solutions is large.
2. The value of a random solution is close to the optimal value.
3. There is a (theoretically incorrect) heuristic algorithm scoring well on random data.

The rationale behind this statement follows.

It is often possible to write a randomized brute force search that examines a subset of possible solutions and outputs the best one found. If the set of correct solutions is large, it is quite common that this approach will be successful and the found solution will often be optimal. We are in a similar situation if the condition 2 holds for the problem. Examples of such problems are IOI 2004 tasks Artemis and Farmer, see Appendix A.

There are some problems that exhibit only the third property. Here it may be possible to devise test data that breaks one known heuristic algorithm, but practice shows that many contestants will write a variation on the known heuristic algorithm(s) that solves all (or almost all) test data. An example is the substring search problem presented in the previous section. While we can break one fixed incorrect implementation, it is impossible to create test data that breaks all of them.

More detailed arguments can be found in [3].

## 5 Theoretical results on testing

The general theoretical formulation of the testing problem (given are two Turing machines, do they accept the same language?) is not decidable – see [4]. While this result does not exactly apply to programming competitions (the set of legal test data is finite), it does give us insight about the hardness of our goal.

It can be shown that a more exact formulation of our testing problem is an NP-hard problem. For the lack of space we present just the idea behind this claim: We will make a reduction to the boolean formula satisfiability problem (SAT). Suppose we have an instance of SAT, i.e., a boolean formula with  $n$  variables. We will encode it as follows: Take the reference solution and in the beginning of the program add a piece of code that splits the input into a sequence of bits, considers the first  $n$  bits to be the values of variables and evaluates the formula for these values. If the formula turns out to be true, the modified program gives an incorrect answer and terminates.

Thus, deciding whether the contestant's program is correct is at least as hard as deciding SAT. This implies that there is no (known) way to do the automated testing efficiently, and at the same time to guarantee 100% accuracy.

## 6 Other evaluation methods

In the previous sections we have shown that the currently used automated black-box testing is not suitable for some tasks the computer science offers. To be able to have contestants solve other task types it may be worth looking at other evaluation methods, discuss their advantages, disadvantages and suitability for the IOI.

Moreover, a huge disadvantage of black-box testing is that an almost correct solution with a small mistake may score zero points. In our opinion the most

important part of solving a task is the thought process, when the contestant discovers the idea of the solution and convinces himself that the resulting algorithm is correct and reasonably fast. We shall try to find such ways of evaluating the solutions that the contestant's score will correspond to the quality of the algorithm he found. The implementation is important, too, but the punishment for minor mistakes may be too great when using the current evaluation model. This is another reason why discussing new evaluation methods is important.

Below we will present a set of alternate evaluation methods along with our comments.

## **Pen-and-paper evaluation**

In Slovak Olympiad in Informatic, in the first two rounds contestants have to solve theoretical problems. Their goal is to devise a correct algorithm that's as fast as possible, and to give rationale why their algorithm works. Their solutions are then reviewed and scored by some experienced informatics teacher.

While we believe that this form of a contest has got many benefits (as it forces the contestants to find correct algorithms, to be able to formulate and formally denote their ideas), we don't see it as suitable for the IOI. The main problems here are the time necessary to evaluate all the solutions, the language barrier, and objectivity. Note that the model currently used at the IMO is subjective. In spite of the best effort of the delegation leaders and problem coordinators it is sometimes the case that almost identical solutions written by contestants from different countries score differently. Also, the process is more error-prone.

## **Supplying a proof**

For the sake of completeness, we want to state that some programming languages allow the programmer to write not only the program itself, but also its formal proof of correctness.

While this theoretically solves all the difficulties with testing the programs for correctness, we don't see this option to be suitable for high school students. Giving a detailed formal proof is far beyond the current scope of the IOI.

## **Code review – white-box testing**

Often a much easier task than designing a universal set of test cases is proving a given implementation wrong, i.e., finding a single test case that breaks it.

This model is currently implemented and used in the TopCoder Algorithm competitions. In each competition there is some allocated time when the contestants may view the solutions other contestants submitted. During this phase, whenever a contestant thinks that he found a bug in a solution, he may try to construct a test case that breaks the solution. Solutions broken in this way score zero points. (Moreover, the contestant that found the test case is awarded some bonus points for this.)

This procedure is accompanied by black-box testing on a relatively large set of test inputs. Only solutions that successfully pass through both testing phases score points.

We are aware that also in some online contests organized by the USA Computing Olympiad (USACO) the contestants are encouraged to send in difficult and/or tricky sets of test data. While this isn't exactly white-box testing, this approach is similar in that the contestants get to design the test data.

Some variation of this type of testing could be implemented on the IOI. During the phase that's currently only used to check the results of testing and to make appeals the contestants could be able to read the submitted solutions and suggest new test cases. This is an interesting idea and we feel that it should be discussed in the IOI community.

## Open-data tasks

An open-data task is a fairly new notion, made famous by the Internet Problem Solving Contest (IPSC, <http://ipsc.ksp.sk/>) and later adopted by other contests, including the IOI. The main point is that the contestants are only required to produce correct output for a given set of inputs. The only evaluated thing are the output files, the method the contestant used is not important.

There is a wide spectrum of tasks that are suitable to be used as open-data tasks at the IOI. For example, tasks where different approximation algorithms exist can be used as open-data tasks and evaluated based on the value of the solution the contestant found. (See the task XOR from IOI 2002.) Large input sizes can be used to force the contestants to write effective solutions. There may be tasks where some processing of the data "by hand" can be necessary or at least useful.

In our opinion the IOI didn't use the full potential of open-data tasks yet and there are many interesting problems that can be formulated as open-data IOI-level competition tasks.

## More extensive black-box testing

In contests other than the IOI the correctness of a submitted program has a larger importance. Usually if the program fails just one of the test data it is considered incorrect and it scores zero points.

We don't think that this would be a good model for the IOI, as the secondary school students are just beginners in programming and they often tend to make minor mistakes. However, there are interesting variations on the canonical testing process that make correctness of the implementation more important.

One particular model worth mentioning is the model commonly used in the Polish Olympiad in Informatics. (This model was also used for testing several tasks on IOI 2005 in Poland.) The test cases are divided into sets and each set is assigned some points. To gain the points for a set, a solution has to solve all inputs from the set correctly.

Clearly this approach makes it easier to distinguish between correct and incorrect solutions (e.g., large random test cases can be bundled with small tricky hand-crafted inputs that will ensure that most of the heuristic algorithms fail). On the other hand, its disadvantage is that the result of a minor bug in implementation may have an even worse impact.

## Requiring correctness, aiming for speed

In our opinion, the goal we want to reach is to guide the students to write correct programs only. If the current evaluation scheme shall be changed, the new scheme should be better in pushing the students towards writing correct programs.

We would like to suggest the following scheme: The author of the problem creates two sets of test inputs. The first set, called the *correctness set*, will consist of 10 relatively small inputs. Any reasonably fast correct solution should be able to solve each of these inputs well within the imposed limits. The second set, called the *efficiency set*, will consist of 20+ inputs of various sizes.

Solutions will be **required** to solve all inputs in the correctness set and they will be scored according to their performance on the efficiency set **only**.

During the actual competition the contestant will be able to submit his solution at any time. Upon submitting his solution he will receive a report (pass/fail, runtime) for each of the (at that time unknown) inputs from the correctness set.

This proposal is still open for suggestions. (For example, we are considering to award a small amount of points to solutions that fail to solve one or two inputs from the correctness set, or alternatively the contestants could be allowed to see some of these inputs for a penalty.)

## 7 Plans for the future

Clearly, the short-term solution of the problems discussed above is awareness that these problems exist. In past, it was sometimes the case that the ISC was aware of some problems connected with a proposed task, but they didn't find these problems important enough to reject the task.

This article attempted to show that not all tasks are suitable for automated testing. We described some of the reasons behind this fact and some possible ways of recognizing such tasks. In our opinion it is important that both the host SC and the ISC are aware that these issues do exist and that they'll take them into consideration when selecting future competition tasks.

The mid-term to long-term solution includes discussion in the IOI community. The currently used evaluation model has got many known disadvantages and if we are able to come up with a better way to do the evaluation, the whole IOI could benefit from that. To reach this goal it is imperative that members of the IOI community actively take part in discussing the alternatives, some of which were presented in this article.

## A Details on tasks investigation.

Here we present a short summary of our investigation for each of the IOI tasks where we were able to find a incorrect solution that scored too many points. As we already stated, a detailed description of problems with the IOI 2004 tasks Artemis and Farmer was informally published after IOI 2004 in [3].

### IOI 2004: Artemis

**Task summary:** Find an axes-parallel rectangle containing exactly  $T$  out of  $N$  given points.

**Problem statement:** <http://olympiads.win.tue.nl/ioi/ioi2004/contest/day1/artemis.pdf>

**Correct solution:**  $O(N^2)$  dynamic programming

**Flawed solution:**  $O(N^3)$  brute-force search, terminate before the time limit expires and output the best solution. This solution requires only a few lines of code and scores 80 points on the test data used in the competition.

**Reason of the problems:** A bad task. It is hard to construct a test case where no valid rectangle contains exactly  $T$  points. The set of possible solutions is usually very large, it is easy to find one, thus also the brute force algorithm scores well. The sad thing is that the ISC was aware of the above facts(!) and still decided to use this task in the competition.

**Notes:** We are aware of the fact that several contestants submitted a correct  $O(N^2 \log N)$  algorithm, which timed out on the larger test cases, thus scoring approximately 40 points. This is the immediate consequence of a “solution” the host SC and ISC applied – raising the limit for  $N$  so that **one known brute-force solution** doesn’t score well.

This problem could probably be saved by requesting that the contestants output the exact number of minimal rectangles satisfying the given criteria. This forces all brute-force solutions to timeout on larger test cases.

### IOI 2004: Farmer

**Task summary:** Given a graph with  $N$  vertices containing only circles and lines, find the largest possible number of edges in a subgraph with  $Q$  vertices.

**Problem statement:** <http://olympiads.win.tue.nl/ioi/ioi2004/contest/day2/farmer.pdf>

**Correct solution:** Knapsack-style dynamic programming.

**Flawed solution:** Greedy. Consider all the cycles. If they contain more than  $Q$  vertices, flip a coin and output  $Q$  or  $Q - 1$ . Otherwise, take all the cycles and several strips (in sorted order, starting from the longest ones).

This solution is expected to score at least 50 points for any test data. For the actual test data used in the competition this solution was expected to score 95 points.

**Reason of the problems:** The host SC and ISC were aware that some heuristic algorithms can score well. The test data was chosen in such a way that the algorithms known to them didn't score too many points. Sadly, this wasn't taken to be a reason to reject the task, as none of the problemsetters had the insight presented in [3]. It is impossible to create good test data for this task.

## IOI 2004: Hermes

**Task summary:** Given a vector of  $N$  lattice points, find a lattice path starting at  $(0, 0)$  such that a postman walking the path "can see" (horizontally or vertically) each of the given points in the given order.

**Problem statement:** <http://olympiads.win.tue.nl/ioi/ioi2004/contest/day1/hermes.pdf>

**Correct solution:** Somewhat complicated  $O(N^2)$  dynamic programming.

**Flawed solutions:** A simple  $O(N)$  greedy solution: Out of the two possibilities for the next step choose the shorter one. In case of a tie flip a coin. This solution was apparently known to the SC and scores only 10 points.

A  $O(N^2)$  greedy solution: Run the first greedy solution  $N$  times, in  $k$ -th run do the opposite choice in the  $k$ -th step. (I.e., make exactly 1 choice that's not locally optimal.) This solution is clearly incorrect and still it scores 45 points.

**Reason of the problems:** Bad test data. In the problem specification it was stated that 50% of the test cases will have  $N \leq 80$ . The truth was that in 50% of test cases  $N$  didn't exceed 20. This number of lattice points wasn't enough to make sufficiently complicated inputs.

**Notes:** In this problem the dimension  $D$  of the grid was smaller than  $N$ . There was a correct  $O(DN)$  solution (which is faster than the presented  $O(N^2)$ ). No additional points were awarded for finding and implementing this algorithm.

Moreover, the size of test cases allowed a simple backtracking solution to score more points than the problem statement promised.

## IOI 2005: Rectangle

**Task summary:** Find an optimal strategy for a 2-heap NIM where an allowed move is to choose one of the heaps and remove at most half of the tokens. The player not able to make a move loses.

**Problem statement:** <http://olympiads.win.tue.nl/ioi/ioi2005/contest/day2/rec/rec.pdf>

**Correct solution:** An optimal move may be found by a clever observation after encoding both heap sizes as base-2 numbers. Also, the more general Sprague-Grundy theory can be applied.

**Flawed solution:** Whenever possible, try to make two equal heaps. This solution is incorrect and yet it scored 70 points.

**Reason of the problems:** Bad test data. In almost all test inputs the heap sizes were similar and thus the player was able to enforce this strategy from the first move on. Actually, this solution easily solves all of the large test cases and fails only on several small hand-crafted cases.

## ACM ICPC: safe/ouroboros

We are aware that this problem was presented on different contests, including ACM ICPC Mid-Central European Regionals 2000, Slovak IOI Selection Camp 2003, and ACM ICPC Murcia Local Contest 2003.

**Task summary:** Given a set of  $D$  digits and  $N$ , find the shortest string of digits such that it contains each of the  $D^N$  possible  $N$ -digit strings as a substring.

**Problem statement:** [http://www.acm.inf.ethz.ch/ProblemSetArchive/BEU\\_MCRC/2000/problems.pdf](http://www.acm.inf.ethz.ch/ProblemSetArchive/BEU_MCRC/2000/problems.pdf), problem E

**Correct solution:** These strings are known as de Bruijn strings/sequences (see [2]), they correspond to an Eulerian path in a wisely constructed oriented graph.

**Flawed solution:** Pruned backtracking is able to solve all reasonably large inputs quickly.

**Reason of the problems:** A bad problem. The set of all  $(D, N)$  de Bruijn sequences is large and the backtracking algorithm can find one quickly.

## References

- [1] Appel, K.; Haken, W. *Solution of the Four Color Map Problem*. 1977. Scientific American 237(4):108-121.

- [2] de Bruijn, N.G. *A Combinatorial Problem*. 1946. Koninklijke Nederlandse Akademie v. Wetenschappen 49:758-764.
- [3] Forišek, M. *On suitability of tasks for the IOI competition*. 2004. <http://ksp.sk/~misof/ioi/tasks.html>
- [4] Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation* 1979. Addison-Wesley.
- [5] Karp, R.M.; Rabin, M.O. *Efficient randomized pattern-matching algorithms*. 1987. IBM Journal of Research and Development 31(2):249-260.
- [6] Knuth, D.E.; Morris (Jr), J.H.; Pratt, V.R. *Fast pattern matching in strings*. 1977. SIAM Journal on Computing 6(1):323-350.
- [7] van Leeuwen, W.T. *A Critical Analysis of the IOI Grading Process with an Application of Algorithm Taxonomies* 2005. Master Thesis at TU Eindhoven. <http://www.win.tue.nl/~wstomv/misc/ioi-analysis/thesis-final.pdf>