Olympiads in Informatics, 2009, Vol. 3, 132–143

Improving the Automatic Evaluation of Problem Solutions in Programming Contests

Pedro RIBEIRO

Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
e-mail: pribeiro@dcc.fc.up.pt

Pedro GUERREIRO

Universidade do Algarve
8005-139 Faro, Portugal
e-mail: pjguerreiro@ualg.pt

Abstract. Automatically evaluating source program files is a crucial part of programming contests. The evaluation aims at discriminating programs according to their correctness and efficiency. Given the performance of today's computers, in order to be able to distinguish the complexity of solutions, it is often necessary to use very large data sets. This is awkward, because it is against the nature of the stated problem and puts an unintended burden on the input operations. Besides, by advertizing a limit for the size of the input, the problem description gives away information with which the contestants may guess the algorithmic complexity that their solutions must attain. It would be more realistic to omit that information and let the contestants discover the limits by analyzing the problem, using a scientific approach. The complexity of the solution can then be estimated automatically by measuring the execution time of the function that solves the problem in incremental test cases, and plotting it against the size of the input. By calling the function multiple times and taking the overall time, we may use only data files the size of which is related to the nature of the problem being solved.

Key words: programming contests, computer science education, automatic evaluation, asymptotic complexity, IOI, International Olympiads in Informatics.

1. Introduction

Serious programming contests, such as the International Olympiads in Informatics, share the need for an efficient and fair way of evaluating and distinguishing the solutions proposed by the contestants. Nowadays, this is typically done using an automatic method by which the submitted code is compiled and run against a set of pre-defined test cases. This black-box approach is very practical but has several drawbacks, some of which have been identified in previous work (Cormack, 2006; Forisek, 2006; Verhoeff, 2006), and others that are discussed in this paper. In any case, for the participants, the main challenge of programming contests is to develop correct and efficient algorithms for the problems that

are presented, and, therefore, the evaluation procedure must be capable of reliably ascertaining correctness and gauging efficiency.

Using the black-box approach, a program will be deemed correct, or, more appropriately, a program will be *accepted*, if it passes all the tests. If these tests are also set up to be sensitive to efficiency, it may happen that a correct program, i.e., a program that given unlimited time would computed correct outputs for the given inputs, will not be accepted.

The ACM ICPC contest takes the crude, yet practical, all or nothing approach: a program is accepted if and only if it passes all the tests and the tests are designed to exclude solutions with a complexity beyond a certain degree. The IOI, on the other hand, sets up its tests so that correct solutions on high complexity (and low efficiency) may get some points, but necessarily less points than more efficient solutions.

In the former case, a certain level of complexity is required; in the latter, complexity has to be discriminated. In both cases, some tweaking with the tests is necessary, using official solutions provided by the judges. Typically, all test cases must run within a certain amount of time, and that amount of time is known to contestants. Thus, the toughest inputs must be designed so that the judge's solution runs in, say, half the allotted time. The setup gets more elaborate if different languages are allowed in the competition, in order to accommodate the intrinsic computational overhead of each of them. For example, a perfectly fine solution written in Prolog might not measure up in efficiency with an algorithmically equivalent one written in C. Sadly, this extra setup often deters organizers from adding new languages to the set of accepted languages in a contest.

Therefore, we should look for more flexible ways of distinguishing the complexity of solutions. Additionally, we should consider the possibility of omitting from the problem description the information about the maximum size of the input data, in those cases where this limit is not inherent to the problem. Indeed, in most cases, the limit is rather arbitrarily chosen, only so that the available judge solution passes within a margin of safety. Besides, given the speed of modern computers, those limits sometimes are unreasonably large. This disfigures the problem statement, discloses the complexity required for the solution and overemphasizes the runtime importance of reading the input data.

In this paper, we report on a technique we are experimenting, which consists of having the automatic judge run the submitted program on a set of test cases with linearly increasing size and plot the execution time versus the size of the input. The test cases are of a moderate size, and are run many times, to overcome the lack of accuracy of the system clock.

This paper is organized as follows. In Section 2, we give a detailed description of the current kind automatic evaluation used in programming contest, pointing what we consider to be the main drawbacks. In Section 3, we propose several ideas that could lead to an improved evaluation of submitted code, along the lines discussed above. In Section 4, we present some preliminary experimental results that seem to confirm that the ideas have some merit. A conclusion follows, in Section 5.

2. Current Automatic Evaluation

At present almost all programming contests share a very strict model of automatic evaluation. We will focus on the IOI, but the other major programming contests follow a very similar line of thought and our considerations are valid for them.

Current IOI contest regulations (IOI'2008 Contest Rules) allow three kinds of tasks:

- **Batch tasks:** the most traditional, where the program must read data from *standard input*, process it and then produce a result on the *standard output* within some time and memory constraints – contestants must submit the source code of the standalone program;
- **Reactive Tasks:** the program must interact with a provided library and all input output is made within the context of the library, in a way that the next input depends on the previous output – contestants must submit the source code of the program, to be linked with the library;
- **Output Only Tasks:** contestants are given a set of input files for the problem and must only deliver the set of corresponding output files – no program code is submitted. In theory, contestants could compute the output files by hand, but typically they must write programs to compute them. These programs, however, need only to handle the given files, may adjust to them, and have no time limits other than the duration of the contest, and no memory limits, other than the memory available in the contest computer.

Looking at Table 1, we can observe that in the past five editions of the International Olympiads of Informatics, batch tasks prevail largely, allotting for more than 80% of the tasks. The extreme case was IOI 2008, where all tasks were of this type. There are many reasons for this, the main one being that it appears to be difficult to design good problems of other types, but what remains is that batch tasks do constitute the core of IOI. In other programming contests, such as the ACM ICPC, batch tasks are the only used form.

Having acknowledged this state of affairs, we will now focus our attention on batch tasks. With these, in order to score automatically the programs submitted by the contestants, a set of pre-determined input files is prepared beforehand. These input files are aggregated in several test groups (some of which may contain a single file) and a predetermined number of points are allocated to each test group. The submitted code is then

Table 1
Types of tasks in the last five IOIs

IOI Edition	Batch Tasks	Reactive Tasks	Output Only Tasks
2008	6	0	0
2007	5	1	0
2006	4	0	2
2005	5	1	0
2004	5	0	1

compiled and run against each of those test groups, receiving for each one the allocated number of points when it solves it correctly, that is, when it produces a correct output within the specified time and memory constraints for all its constituent input files. The final score is just the sum of points obtained in each test group.

Therefore, the input files, their grouping and the assigned number of points are crucial to the results of the contest. They must be designed and then created very carefully. When doing so, we must take into consideration two main aspects:

- **Correctness**: the program should provide a correct answer in the sense that it should solve all instances of the problem;
- **Efficiency**: the program should be evaluated having in account its asymptotic time and memory complexity.

Evaluating the correction is tricky and impossible to do with perfection in this kind of black-box testing. As Dijkstra famously said *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"* (Dijkstra, 1972). Indeed, we cannot check all possible ways of making an error, and even when we do find that the program is not able to solve a specific test case, the same score (zero) is awarded whether it is a mere implementation bug or a fundamental reasoning error in the algorithm itself. And how can we really compare two different incorrect implementations? How should we distinguish them in terms of score? These shortcomings regarding program correctness have already been analyzed by other authors (Cormack, 2006; Forisek, 2006; Verhoeff, 2006).

Regarding efficiency, typically the judges create a set of correct model solutions of different complexities and the tests are designed in such a way that the model solutions achieve the preplanned number of points. A considerable degree of manual tuning regarding the actual system used is normally needed and there is no guarantee that the submitted programs will score as expected, even if the algorithm matches the required complexity. In general, one cannot predict every possible approach and students always find surprising ways of solving the problem at hand. And when the submitted program goes in a different direction than the ones that modeled the input, the number of points may not be what the problem setter had in mind for that type of solution. For example, how can we compare the efficiency of two programs that solve the exact same set of test groups if the test groups of this set do not cover all the expected complexity range?

Another important factor is input-output. Besides constituting a distracting aspect by itself (Vasiga *et al.*, 2008), it may represent a considerable part of the execution time. This fact complicates the evaluation of the complexity of algorithm used by looking at the overall runtime only. In addition, the language and the type of input-output that is used matters. For example, *cin* and *cout* in C++ are considerably slower than *printf* and *scanf*.

One other important issue is the constant increase in computer performance. When this kind of automatic evaluation started, we were able to use relatively small and self-contained limits for the inputs. These small limits would allow testing even linear time complexities. This is however not true for today's computers. In order to force small asymptotic complexities one must use huge limits that detach the problems from reality. Although realistic statements are nice, we are forced to have things like a sailing ship

with 100,000 masts (Problem Sails, IOI 2007), just so naïve solutions can be rejected. More than that, the poor CPU clock resolution may make it unfeasible to distinguish between small complexities like `O(N)` or `O(N log N)`. The minimum time limit ever used was 0.3 seconds (Problem Training, IOI 2007). Constructing an input that would pass with an `O(N)` algorithm, but not with `O(N log N)` would need a really large `N`. To separate between `O(log N)` and `O(N)` we would need an even larger `N`. But this would mean that the program would probably spend even more time reading the input (linearly) than on the actual solving phase. Besides, the value `N` would probably be too big to fit in memory.

A final important point is that by advertizing in the problem description the maximum size of the data used in the evaluation, we really give a big hint to the contestants. We do avoid the need for dynamic memory allocation, and that is a good thing. However, experienced contestants can more or less easily infer the expected time complexity of the best algorithm for the problem from those limits. And they will immediately try to just create a solution that matches that, without any other concerns. This is in a sense a very different situation from the scientific way of approaching a problem in the spirit of making our best possible effort, without knowing beforehand what that best is (at least without thinking about the problem).

3. Proposed Improvements

As we have observed above, a great deal of effort is put on devising the test cases. Test cases are typically divided in two families: one formed by small test cases (most of them handcrafted) devised to test correction, and the other made up of increasingly larger test cases, usually generated by other programs, that try to evaluate performance. We will focus on ways to improve this second family of tests. Our proposal is to include one or more of the following ideas, which we will then describe in more detail:

1. **Developing a specific function** (together with other auxiliary functions, if convenient) as opposed to developing a full program.
2. **Abstraction of input-output**.
3. **Repeat the same function call** several times to increase the clock precision.
4. **Do not give hints to contestant** about the best expected asymptotic complexity.
5. **Estimate asymptotic complexity** by looking at time-spent behavior on several tests.

For supporting the discussion of these ideas, we will use a toy problem: finding the smallest difference between two numbers of a set of integers. Let's call this problem `SmallDiff`.

Our first proposal is that the task of the contestant in solving the problem might be to develop a specific function rather than a complete program. Actually, this is already done in the TopCoder contests and also in introductory programming courses (Ribeiro and Guerreiro, 2008). This allows the judges to have greater control on how to manage the submitted code and creates the opportunity for new ways of testing which

would be infeasible for full programs. In the context of the problem `SmallDiff`, instead of presenting the traditional detailed input and output specifications, the problem description would specify the signature of the function to be developed, for example `int FindSmallDiff(n,s)` which would receive a set `s` of `n` integers and would return the smallest difference found. This would not in any way make it more difficult for the student to program the solution. In fact, it would do the opposite, by allowing the contestants to concentrate on the core algorithm that solves the task and by minimizing the impact of some of the distractors described in (Vasiga *et al.*, 2008). Indeed, in pedagogical environments, students seem to be caught by these distractors, wasting their effort on subsidiary issues, and then only addressing the true problem when there is not enough time.

In particular, submitting functions instead of full programs would allow minimizing the information processing details of input-output, thus accomplishing the input-output abstraction that we propose. All languages would have a much more balanced and equal performance in this field. Besides, we could now measure the time that the program spent computing the solution, and not the time spent doing input and output. One could argue that by giving the data already in memory to the program, the contestant does not have to think about how to store it and what data structure to use, but we can always give it on the most simplest of forms (a simple chunk of values), unprepared for efficient processing, which would require the contestant to migrate the data to the desired data structure. And we could even provide access to the data in a way to the similar traditional one by providing functions that get the next integer, the next string or whatever other kind of data, from the basic data structure that was used for the function argument. Thus, by abstracting the data we minimize implementation details and language differences, and at the same time we are able assess the core of the solution, without almost any input-output overhead.

Another concept we advocate is that instead of having really large input sizes to test efficiency, we could use smaller ones, whose size is compatible with the nature of the problem. In reality, what deters one from doing that in the present situation is that the clock resolution is too poor and small input sizes will lead to "instant" responses, all with zero time spent and no way of distinguishing among them. If an arbitrary precision clock was available, we could concentrate more in the quality of the test cases and not have to resort to randomization techniques in order to produce huge tests that are virtually impossible to verify manually. This can be done by calling several times the function that solves the problem, with the same arguments or with different arguments of the same size. It is true that an `O(N^4)` algorithm will probably be instantaneous when we run it twice with `N` = 10. But if we run it a few hundred times, what will it happen? We gain the accuracy that we need precisely by running it multiple times! This is the same technique that we might use to measure the thickness of a sheet of paper: if a stack of 100 sheets measures 5 cm, we conclude that the thickness of each sheet is 0.1 mm, even if we do not have an instrument that reaches that accuracy. The main point is that since we have the solution isolated in a function, we can call the function as many times as we want and then just compute the average time spent inside each call. It is true that

one must be careful not to let memory persistence between successive function calls be exploited by dishonest contestants in order to give quicker responses. This may be dealt with, for example, by always using different equal-sized instances of the problem. In the context of problem `SmallDiff`, we could call `FindSmallDiff` in the evaluation script the required number of times, providing arguments that are hardwired in the script or generated randomly, if the overhead of the randomizer is acceptable. Notwithstanding, in other problems, there may be other opportunities for persistence, such as storing pre-calculated values in static memory, and serious consideration has to be given to this issue.

Another idea that we propose is not to give hints to the contestants about the intended program complexity on the problem description. Indeed, just by looking at the maximum input size and allowed runtime, contestants may perceive whether there is a polynomial solution to the problem. This must be one of the first techniques in which IOI competitors are coached, in preparation for the competition. One can get very precise in these matters, and really identify the specific complexity needed, for the problem at hand, for example `O(N^2)`. This has a great impact on the problem solving aspect and the student approaches the problem with a mindset different from the one he would bring, should he have no idea of the targeted complexity. For example: what strategy should a contestant adopt when he is given an optimization problem and he has no clues as to the size of the input? If he did not know in advance whether there is a polynomial solution, would he try to find one or settle with an exponential approach? In the *real world*, this is what happens. We have open problems that we do now know how to solve optimally. In some cases, the problem has polynomial solutions, in other case, it does not. Sometimes we create efficient solutions only to verify afterwards that they fail on some particular test cases. We never know if we are achieving the optimal solution, unless we prove it ourselves. This contrasts sharply with knowing in advance not only the needed complexity but also being satisfied with our solution, not because it is the best possible, but because it solves the particular instances that will be tested within the given constraints. In our opinion, we should favor the more open ended approach, making students think how to really solve the problem and not on how to write a program that passes the test cases of a given size. Note that in what concerns implementations, limits can still be given (thus avoiding the need for dynamic memory allocation), while making clear that those limits are not related to the unknown efficiency that is sought.

In the example problem, `SmallDiff`, consider that it is said that the maximum size of the set is 10. Since the number of problems instances that will be tested remains secret, the contestant cannot know beforehand if the brute-force `O(N^2)` approach would suffice even if it run instantaneously. Neither will he know if simply ordering the numbers in `O(N log N)` would suffice. He would have to think if an `O(N)` is possible. As a side note, one of the authors of this paper, himself a participant in several programming contests, was once exposed to an ACM IPCP type contest where no limits were given. While in some cases this complicated the implementation by requiring dynamic memory allocation (what we avoid, as discussed), the fact was that the contest was very interesting and refreshing and also more challenging, because it was impossible to discover beforehand whether the problem was NP-hard or if it had a greedy or dynamic programming solution.

With no hints on the expected best running time, we needed to analyze the problem more thoroughly and try to figure out whether there were more effective approaches than the ones we were taking so far.

Our final idea regards measuring the time efficiency of the submitted programs. Typically one creates the test cases in such a way that our model solutions of different complexities will have the intended number of points. However, we cannot predict all the different solutions contestants will create and sometimes programs with better complexities than others will score fewer points because of other factors (Forisek, 2006). Passing a specific test case does not also really advertise that a specific time complexity was achieved. Or rather, not passing does not imply that the solution is wrong or that the required complexity was not achieved. In reality, it just shows that that particular test case is solved within the constraints. Pen-and-paper evaluation could be a solution to really scoring the programs with regard to their actual complexity, but that is unfeasible in practice for a competition of the size of an IOI and also moves away from the required objectivity in the evaluation. What one could do is to try to automatically estimate the complexity of the submitted solution by augmenting the data and plotting the time spent computing the result from those data. A curve fitting analysis could then be done in order to estimate the complexity and possibly even extrapolate other time data points. We are aware that this is impossible to do for all possible problems but the fact remains that many contest problems (or program assignments) have solutions with simple complexities than can be approximated automatically. In any case, even a trivial (although imperfect) curve can provide more information than just checking which test cases the program passes, because it give us a real feel for the asymptotic behavior of the function being evaluated. The main point is that by globally analyzing the runtime behavior of the solution, we get more information than by just timing it on each separate test case. We may not be able to infer the exact worst case asymptotic complexity of a program, but we may discover that in practice it spends half the time on a test case with half the data, while another solution for the same problem spends only a quarter of the time and still another one solves the problem in constant time. This is meaningful information that can be used for grading the program.

4. Experimental Results

In order to perform a first evaluation of the practical application of the proposed improvements, we implemented a preliminary, experimental version. As a proof of concept we will be using a simple problem which was presented as one of the easy tasks in a Portuguese IOI training campus. Stripping the problem to its core, we are given an array of integers (positive or negative), and we want to compute the sequence of consecutive elements which has a maximum sum.

We did this example in C but it can easily be ported to other languages. Instead of a complete program, we require a function `int calculate(int n, int v[])` that returns the maximal sequence as defined before, given an array `v` of `n` integers. The

contestant may concentrate only on solving the algorithmic task. We will now focus on how to test programs efficiency, particularly in estimating asymptotic time complexity. We experimented by repeating the exact same function call several times until the elapsed runtime was measurable, at a human scale, say more than one second. Then, the expected time for a single run of the function call is equal to the total time spent divided by the number of times the function was called. With this in mind we went further ahead and calculated the time it takes for a linearly increasing size `n`. We can then look at the data and do statistical analysis in order to discover a good *curve fitting* that can explain the data obtained.

Putting this into practice, we experimented with random test data and we programmed three model solutions of different complexities:

- **Solution A** is a brute force approach with two nested cycles choosing all possible upper and lower limits of the intervals and then running another cycle for each of these pairs in order to calculate its corresponding sum – this approach is `O(N^3)`.
- **Solution B** is a more refined approach, again with two nested cycles for the sequence followed then by a `O(1)` calculation of the corresponding sum using pre-calculated partially sums, which in turn was made in `O(N)` – the global complexity of this approach is `O(N^2)`.
- **Solution C** is even more optimized and simply maintains the current candidate sum on an auxiliary variable, linearly going trough all the numbers and summing while the candidate sum is positive and therefore contributes to a possible best – this approach is `O(N)`.

We then run these solutions within our improved evaluation framework for a series of increasingly bigger `N` (we used {1,4,8,12,16, . . ., 64}) and measured the time the solution takes for that `N` compared to the time the same solution takes for `N = 1`. Fig. 1 is a plot of the results obtained. Note the very nice curves that already visually suggest the corresponding complexities.

We can now use several statistical approaches to discover which class of complexity the observed behavior of the solution best matches. For the sake of simplicity we will just use one simple measure to show how much information this data provides. We calculated the correlation coefficient between the obtained times and the typical and classical complexity functions. The results are illustrated in Table 2. In gray we can see the maximum

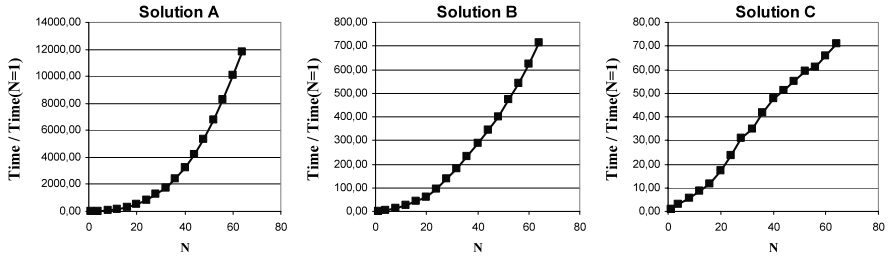

Fig. 1. Comparative time spent for running the three implement solutoins as `N` linearly grows.

Table 2
Correlation coefficient of the time spent by the solutions

Solution	Log N	N	NlogN	N^2	N^3	N^4	2^N	N!
A	0.6848	0.9264	0.9515	0.9912	**0.9993**	0.9869	0.6033	0.5722
B	0.7524	0.9666	0.9835	**0.9998**	0.9848	0.9564	0.5469	0.5183
C	0.8624	**0.9952**	0.9927	0.9586	0.8985	0.8417	0.4136	0.3906

coefficient obtained for each solution. This coefficient corresponds precisely to the class of time complexity to which the solutions belong.

Note than we can not only identify the most probable complexity, but we can also "approximate" the strength of our conviction in the result. Thus, we can use the identified complexity to automatically give more or less points. We did experiments with this method on the solutions submitted by our students and the results that we obtained confirmed our manual analysis of the code and were always strongly correlated to the respective program complexity.

It is out of the scope of this paper to give a more detailed mathematical analysis of the statistical significance or which kind of statistical measure should be used to better fit the data. What stands is that this preliminary yet elegant approach shows that this is indeed a path that can lead to meaningful results. The same approach could also be used when more variables are involved by extending the calculation to higher dimensions.

We are aware that the statistics do not prove that the programs have the corresponding asymptotical complexities. They only mean that, in practice, for the group of selected test cases, the runtime is somehow consistent and correlated with a certain function and therefore appears to grow following a pattern that we were able to identify. For example, when we detect a correlation to a linear function, we recognize explicitly that the program appears to take twice the runtime when the test case doubles the amount of data.

5. Conclusion

The IOI has been running for 20 years but the problems of the early 1990's are similar to those of the late 2000's. The structure of the event has not changed much either. Even the languages used have remained essentially the same. There has been the continuous increase in the speed of computers and on the memory available but, from the point of view of the competition, this is both a blessing and a curse, as we have discussed. In our opinion, the single most significant change occurred when manual evaluation was replaced by automatic evaluation. Automatic evaluation has been made ever more efficient, and lately the results come out minutes after the competition ends. It has also been made more accurate, when the system of points per test was replaced by the system of points by sets of tests targeted at certain complexity. Setting up such sets of tests is a delicate task that may be done in "serious" contests with a scientific committee devoted to it, but may be too time-consuming for the purpose of more informal contests, such as those carried out within programming courses, for pedagogical purposes.

The evaluation system that we envision simplifies the task of setting up a contest in several ways: first, it is not necessary to handle large data sets; second, the problem statements can be more natural, by avoiding the distraction of displaying huge limits for the size of data; third, the problems themselves become more interesting, since there is no clue on the intended complexity, giving contestants the illusion that they may target at something better that what the problem setters may already have achieved, when designing the problem; forth, the human judge does not have to program a solution for each level of complexity, in order to be able to design the test cases for that level of complexity; fifth, different languages can be added at will (provided the automatic judge and the operating system is capable of handling them, but these are other issues), and still be comparable, thus enriching the contest itself and opening it up to larger audiences. And, as a side-effect, by releasing the contestants from the drudgery of input-output, it allows them to concentrate on problem-solving proper. A further benefit is that the system can be used for pedagogical purposes from the early stages, even before students learn how to read and write data files (Ribeiro and Guerreiro, 2008).

More work is needed in order to obtain a robust system, but the preliminary results are promising and seem to indicate that this line of approach has some merit and deserves further consideration. If successful, it would significantly improve the methods of automatic evaluation used in programming contests.

References

Cormack, G. (2006). Random factors in IOI 2005 test case scoring. *Informatics in Education*, **5**, 5–14.

Dijkstra, E.W. (1972). The humble programmer, 1972 Turing award lecture. *Communications of the ACM*, **15**(10), 859–866.

Forisek, M. (2006). On the suitability of programming tasks for automated evaluation. *Informatics in Education*, **5**, 63–76.

IOI'2008 Competition Rules. `http://www.ioi2008.org`

Ribeiro, P. and Guerreiro, P. (2008). Early introduction of competitive programming. *Olympiads in Informatics*, **2**, 149–162.

Vasiga, T., Cormack, G. and Kemkes, G. (2008). What do olympiad tasks measure? *Olympiads in Informatics*, **2**, 181–191.

Verhoeff, T. (2006). The IOI is (not) a science olympiad. 2006. *Informatics in Education*, **5**, 147–159.

P. Ribeiro is currently a PhD student at Universidade do Porto, where he completed his computer science degree with top marks. He has been involved in programming contests since a very young age. From 1995 to 1998 he represented Portugal at IOI-level and from 1999 to 2003 he represented his university at ACM-IPC national and international contests. During those years he also helped to create new programing contests in Portugal. He now belongs to the Scientific Committee of several contests, including the National Olympiad in Informatics, actively contributing new problems. He is also co-responsible for the training campus of the Portuguese IOI contestants and since 2005 he has been deputy leader for the Portuguese team. His research interests, besides contests, are now focused on parallel algorithms for pattern mining in complex networks.

P. Guerreiro is a full professor of informatics at Universidade do Algarve, in Faro. He has been teaching programming to successive generations of students, using various languages and paradigms for over 30 years. He has been involved with IOI since 1993. He was director of the Southwestern Europe Regional Contest, within ACM-ICPC, International Collegiate Programming Contest (2006, 2007), and chief judge of the worldwide IEEExtreme Programming Competition in 2008 and again in 2009. He is the author of three popular books on programming, in Portuguese. His research interests are programming, programming languages, software engineering and e-learning. He currently spends too much time in tasks related to university administration.