ISSN 1822-7732

INTERNATIONAL OLYMPIAD IN INFORMATICS
INSTITUTE OF MATHEMATICS AND INFORMATICS
INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING

OLYMPIADS IN INFORMATICS

Tasks and Training

Volume 2 2008

Selected papers of
the International Conference joint with
the XX International Olympiad in Informatics
Cairo, Egypt, August 16–23, 2008

OLYMPIADS IN INFORMATICS

Country Experiences and Developments

ISSN 1822-7732

Editor

Valentina Dagienė

Institute of Mathematics and Informatics, Lithuania, dagiene@ktl.mii.lt

Co-editors

Arturo Cepeda, the Mexican Committee for Informatics, Mexico, acepeda@auronix.com

Richard Forster, British Informatics Olympiad, UK, forster@olympiad.org.uk

Krassimir Manev, Sofia University, Bulgaria, manev@fmi.uni-sofia.bg

Troy Vasiga, University of Waterloo, Canada, tmjvasiga@cs.uwaterloo.ca

http://www.mii.lt/olympiads_in_informatics

Olympiads in Informatics, Volume 2, 2008.

2008.06.25. 13 leidyb. apsk. l.
Tiražas 160 egz. Užsakymas Nr. 1715.

Printed by Printing house “Mokslo aidai”, Goštauto 12, 01108 Vilnius, Lithuania
Cover design: Multimedia Center, Arab Academy for Science and Technology, Alexandria, Egypt

Olympiads in Informatics, 2008, Vol. 2, 3–4

Foreword

OLYMPIADS IN INFORMATICS is a refereed scholarly journal that provides an international forum for presenting research and development in the specific area of teaching and learning informatics through competition. The journal is focused on the research and practice of professionals who are working in this field. OLYMPIADS IN INFORMATICS is published annually (in summer). Only original high quality research papers are accepted. All submitted papers are peer reviewed.

The journal is substantially connected with the conference organized during International Olympiads in Informatics (IOI). Papers are requested to be presented during the conference. The main goals of the conference (and journal) are:

– to improve the quality of olympiads,

– to improve the skills of contestants and educators, and

– to discuss developing tasks and topics for olympiads.

The first OLYMPIADS IN INFORMATICS conference was organized in Zagreb (2007) and put attention on organizing olympiads at the national level. The papers in this volume are being presented during the IOI in Cairo (2008).

The national olympiads do not exist in isolation, and the papers in the inaugural conference showed how similar problems arise in different environments, and illustrated some of the solutions that both identify and distinguish the community. This volume concentrates on training and task types, and many of the ideas and experiences are drawn from the national olympiads. There are common trends here and, we hope, useful discussions for both the national and international level.

Tasks are perennial issue for contests, their most visible aspect and, for many contestants, the primary reason for participation. We strive for quality, variety and suitability. We endeavour to make tasks interesting, understandable and accessible. They are used to test contestants and to train them, and perhaps even to capture the imagination of those outside the contest, be they family, sponsors or the media.

If tasks seem the main purpose of an Olympiad to the contestants often, from an educator's perspective, there is equal interest in training the contestants. This is not only a question of how we choose the best, or enable the to show their true ability. We seek to enthuse them with a passion for the subject. Some of the best students in the world participate in our olympiads, nationally and internationally; their interest in the subject is win for everyone.

That the national olympiads exist in a wider community is also true of the international olympiads. We are delighted, for this second conference, to have an invited paper by Prof. M. Revilla discussing an international topic outside of the IOI. This is intended to be the start of a trend, both for invited papers and for a widening of the contributors and audience for the conference.

Thanks are due to everyone who has contributed to the IOI conference. In particularly, we would like to thank the organizers of this year's IOI in Egypt – the Ministry of Communications and Information Technology, the Ministry of Education, and the Arab Academy for Science, Technology and Maritime Transport. Many thanks to Prof. Dr. Ekram F. Abdel Gawad, Prof. Dr. Mohamed Aly Youssef, Dr. Salah A. Elewa – without their assistance it would not have been possible to hold this event.

Editorial Board

Valentina Dagienė
Arturo Cepeda
Richard Forster
Krassimir Manev
Troy Vasiga

Olympiads in Informatics, 2008, Vol. 2, 5–15

Breaking the Routine: Events to Complement Informatics Olympiad Training

Benjamin A. BURTON
Department of Mathematics, SMGS, RMIT University
GPO Box 2476V, Melbourne, VIC 3001, Australia
e-mail: bab@debian.org

Abstract. Like many other countries, Australia holds live-in schools to train and select students for the International Olympiad in Informatics. Here we discuss some of the more interesting events at these schools that complement the usual routine of lectures, problems and exams. In particular we focus on three events: (i) codebreakers, aimed at designing tests and finding counterexamples; (ii) "B-sessions", aimed at implementation and rigour; and (iii) team events, which use challenges in cryptography and information security to encourage teamwork in a competitive setting. Practical issues are also discussed.
Key words: programming contests, training, testing, implementation, teamwork.

1. Introduction

Since 1999, Australia has hosted live-in schools to train and select secondary school students for the International Olympiad in Informatics (IOI). Many countries follow a similar pattern; examples are discussed by Anido and Menderico (2007) and Forišek (2007), amongst others.

The Australian training schools run for approximately ten days. A typical day consists of lectures and laboratories in the morning (or possibly a programming contest instead), more lectures and laboratories in the afternoon, and then an evening session at a whiteboard where students present and analyse their solutions to problems.

In order to keep the students interested and to help them learn a broader range of skills, a variety of different events are slotted into the programme over the ten days. Some of these events focus on particular skills that are useful in the IOI, and others extend students beyond traditional IOI material.

The aim of this paper is to describe some of these additional events that complement the usual routine of lectures, laboratories, contests and problem sessions. In particular, we focus on:

- *codebreakers*, in which students create input files to break incorrect solutions to contest problems;
- *B-sessions*, in which students focus on the implementation of a difficult problem that they already understand in theory;

- *team events*, in which students work in groups to solve puzzles in cryptography and information security.

These three events are described in Sections 2, 3 and 4 respectively. Section 5 discusses practical requirements for running such events, including preparation and supporting software, and Section 6 briefly outlines some other events not included in the list above.

For further information on the Australian training programme, including written contests, programming contests and joint events with other delegations, the reader is referred to (Burton, 2008). The author is grateful to Bernard Blackham for his valuable comments on this paper.

2. Codebreakers

Much of the regular teaching programme in Australia focuses on *correct* solutions to problems. This is largely through necessity; the training schools are short and there is much material to cover.

However, it is critical that students be able to identify *incorrect* algorithms – in a contest, one cannot afford to spend hours coding up a solution only to discover during testing that it gives the wrong answers. This is particularly important for students with weaker mathematical backgrounds, who find it difficult to prove algorithms correct and instead rely heavily on intuition, examples and testing.

In order to identify incorrect algorithms, students must be willing to spend time searching for *counterexamples*, i.e., specific test cases for which an algorithm gives an incorrect answer. This often requires patience and creativity, since examples that are simple or natural are often not rich enough to show why an algorithm is wrong.

For these reasons, the codebreaker was created as a training school event in April 2005, and has been a fixture on the programme ever since. The codebreaker aims to help students develop the following skills:

- finding counterexamples to algorithms, which is important during the pen-and-paper stage of designing an algorithm;
- generating pathological test cases as real input files, which is important during the final stage of testing a solution.

2.1. *Codebreaker Structure*

The codebreaker runs as a live (and sometimes noisy!) competition, with a running scoreboard at the front of the room and a contest website that evaluates submissions on the fly (illustrated in Fig. 1). The competition is relatively informal, and is structured as follows:

- Students are given three or four problems. These are all problems that the students have seen and thought about before (typically exact copies of problems from the most recent national contest).

Codebreaker Submissions for Bernard Blackham

Log out

Below is a list of all problems and source files for this codebreaker.

Problem	Source	Score out of 10	Attempts	Actions
1. Wetlands	wetlands1.java	10 (success!)	1	View submissions
	wetlands2.c	10 (success!)	1	View submissions
	wetlands3.cpp	8 (success!)	2	View submissions
2. Mansion	mansion1.cpp			Break!
	mansion2.cpp	8 (success!)	2	View submissions
	mansion3.cpp			Break!
	mansion4.c	10 (success!)	1	View submissions
3. Invasion	invasion1.cpp	8 (success!)	2	View submissions
	invasion2.cpp	-1	1	Break!
	invasion3.cpp			Break!
	invasion4.pas			Break!
	invasion5.cpp			Break!
4. Restaurants	restaurant1.cc	9 (success!)	2	View submissions
	restaurant2.py	-3	3	Break!
	restaurant3.cpp	10 (success!)	1	View submissions
	restaurant4.cc	10 (success!)	1	View submissions

Your total score for the codebreaker so far is 79.

Fig. 1. A screenshot of the codebreaker website during the contest.

- Students are given several "solutions" to each of these problems. Each solution is given as source code (e.g., C++ or Pascal), and each contains an error, possibly in the implementation of the algorithm or possibly in the algorithm itself.
- While the event is running, students attempt to break these solutions. To break a solution, a student must submit an input file to the contest website. This input file is scored as follows:
  - if the input file is valid (conforms to the problem specification) and the solution does not solve it correctly within the time and memory limits, the solution is considered broken and the student gains ten points;
  - if the input file is invalid (does not conform to the problem specification), the student loses two points;
  - if the input file is valid but the solution solves it correctly within the time and memory limits, the student loses one point.
- Students may make as many attempts as they wish to break each solution. At the end of the competition, the student with the most points is declared the winner!

A typical codebreaker contains around 15 solutions to break, and runs for about two hours. Ideally a few (but not too many) students will have successfully broken every solution by the end of the contest, and so the winner is determined not only by the ten point rewards but also by the 1–2 point penalties. Indeed, the duration of the contest is often changed on the fly, and at least once it was declared to be "until three people have broken every solution".

A nice feature of the codebreaker is that it often gives weaker students a chance to shine. Students with a strong sense of thoroughness and rigour can do very well in the

codebreaker, even if they find it difficult to design or implement their own algorithms. Furthermore, students who are slow but careful can beat students who are fast but sloppy (since sloppy competitors tend to accrue several 1–2 point penalties). For these reasons, the winners of the codebreaker are often not the students who typically come first in regular programming contests.

2.2. *Selecting Problems and Solutions*

Some effort needs to go into choosing the problems and their "solutions" that students are required to break.

As discussed earlier, students should already be familiar with the problems. The focus should be on breaking incorrect solutions, not working out what a *correct* algorithm might look like. Indeed, some students like to code up correct algorithms during the event so they can verify where the incorrect solutions fail. Problems from the latest national contest work well, since students will all have taken this contest, and because the contest problems are often discussed earlier in the training school.

The bulk of the effort in preparing a codebreaker lies in creating the incorrect solutions. General guidelines are as follows:

- The solutions are written by staff members (in particular, we do not use real erroneous submissions from the national contest). This avoids privacy issues, as well as allowing the staff to ensure a good spread of coding styles and types of error.
- The different solutions for each problem should be written by a variety of people, and should be in a variety of languages. This is because, if one person writes several similar solutions, it may be easy to see where the solutions differ and thus spot the errors in each. For the same reason, if one person does write several solutions, they should go to some effort to use different layouts, variables, control structures and so on.
- The coding styles should be clear and readable. The aim is not to obfuscate the code so that students cannot work out what it does; instead the aim is to allow students to analyse the code and find where the algorithm or implementation is broken. Often the solutions even include comments explaining what they are doing or why they are "correct".

  This of course does not mean that the *algorithms* cannot be unwieldy or complicated. For instance, some past solutions have read integers from the input file digit by digit and pasted them together into a string (much like some inexperienced students do in the national contest). We merely require that students should be able to understand precisely what each algorithm is trying to do.
- Occasionally solutions are written in languages that are not part of the official set (such as Java, Haskell or Python). Although many students cannot *write* in these languages, the aim is for them to realise that they can still *reason* about programs written in these languages. Of course these solutions are clear, commented and avoid obscure language features.
- Every solution should give the correct answer for the sample input in the problem statement. Moreover, different solutions should break on different types of input

files, so that students cannot just reuse the same input files again and again. This is sometimes difficult to achieve, but it is a nice ideal to keep in mind when designing the various solutions.

There are several types of errors that can be introduced into these "solutions", including:

- *implementation errors*, where the algorithm is correct but the code is buggy, such as using $<$ instead of $\leqslant$ or missing a boundary case;
- *algorithmic errors*, where the implementation is correct but the underlying algorithm is wrong, such as a greedy solution to a shortest path problem;
- *comprehension errors*, where the program solves the wrong problem, such as a program that omits one of the conditions in the problem statement.

Some students like to use black-box methods during the codebreaker and some like to use white-box methods. *Black-box methods* involve creating a range of test cases with known answers and running them through a solution one at a time. *White-box methods* involve manually reading through the code, verifying each step and identifying mathematically where the solution breaks down.

Whilst many solutions can be broken using either method, it is always nice to include one or two solutions that require white-box methods. Such solutions might only break under very special circumstances that are difficult to obtain through random number generators, but that are easy to identify once the mathematics of the incorrect algorithm is understood.

As a related event, it is worth noting the TopCoder contests (`www.topcoder.com`), which incorporate codebreaking into their regular programming competitions. After writing their code, TopCoder contestants enter a brief 15-minute "challenge phase" in which they attempt to break each others' solutions. This allows for a codebreaking environment that is more realistic and competitive, but where the difficulty, variety and readability of solutions is not controlled. See (Cormack *et al.*, 2006) for a more detailed comparison between the TopCoder contests and the IOI.

3. B-Sessions

Although the focus of most lectures is on algorithms, it is critical that students be able to implement these algorithms correctly in the tight constraints of a real contest. The B-session aims to address the following problems:

- Many students are sloppy with implementations – even when they understand an algorithm in theory, they often rush the coding and end up with buggy code. If they test their programs well, they might observe the bugs and spend valuable time trying to find and fix them. On the other hand, if their testing is sloppy also then they might never notice the bugs at all.

  In a real contest this can have disastrous consequences. Depending on the official test data, small bugs can sometimes reduce an otherwise correct solution to score as low as 5–10%. This is particularly true of tasks whose test data is grouped into all-or-nothing batches. Opmanis (2006) discusses these difficulties in greater detail.

- Students may be reluctant to code up complicated algorithms during a contest – even if they can see how to solve a problem in theory, they might decide the risk is too great that (i) they might need to spend hours debugging their code, or (ii) they might not have time to finish their implementation at all. This is particularly true of problems with 50% or 30% constraints, where students can guarantee a healthy partial score by coding an inefficient algorithm that is much simpler and safer to implement.
- When training, many students like to focus on problems that they know they can solve. Whilst this is good for cementing what they have already learned, it does not necessarily help them solve more difficult problems. In order to improve, students must invest time working on problems that are hard for them, so that such problems can in time become easier for them. To use a cliché: no pain, no gain.

A typical B-session runs as follows:

- A single difficult problem is chosen by the organisers. This problem is handed out to students the day before the event. Students are encouraged to think about the problem, either individually or in groups, but they are instructed not to write any code.

  The chosen problem should be too hard to give in a regular contest, but the stronger students should be able to solve it given enough time.
- The B-session itself runs for an entire morning or an entire afternoon (usually $3\frac{1}{2}$ hours). The session begins in the lecture room, where the students discuss their ideas as a group. A staff member chairs the discussion, but ideally the students should be doing most of the talking.

  As the discussion progresses, the group works towards a correct algorithm – students with promising ideas present them at the whiteboard, and the group analyses and refines them until (i) the group has produced a solution that should score 100%, and (ii) all of the students understand this solution in theory. Typically this takes about an hour.

  The staff member plays an important role in this discussion. They must keep the group moving towards a correct solution, dropping hints where necessary but not appearing to reveal too much. They must also ensure that every student understands the algorithm well enough to begin coding it, and they should prevent the discussion from going into too much detail about the implementation.
- Once the discussion has finished, students move into the laboratory and enter exam conditions. The remainder of the B-session runs as a programming contest with just one problem (which is the problem they have been working on). For the remaining $2\frac{1}{2}$ hours, the students must implement the solution that has been discussed and test it thoroughly, with an aim to score 100%.

The B-session therefore has a very clear separation of algorithms and implementation. During the discussion and the day beforehand, the focus is on algorithms only – this allows students to attack a problem that would usually be too hard for them, without the time pressure of a contest or the difficulties of implementation. During the exam period, students focus single-mindedly on implementation – this allows them to spend time de-

signing their code, writing it carefully and testing it thoroughly, without the distraction of other problems that need to be solved.

The eventual hope is that students gain experience at coding difficult algorithms correctly, and also that they learn to become less intimidated by difficult tasks.

It is important for the organisers to select the B-session problem carefully. The problem must be difficult but approachable, and it must require a reasonably complicated implementation. An example is *Walls* from IOI 2000, which was given to the senior students in 2005 at their first training school. The algorithm is challenging for students new to graph theory, but is approachable because it is essentially a breadth-first search with some non-trivial complications. The implementation is messy because of these complications, and also because the input data is presented in an inconvenient format.

In general it is nice to use IOI problems for B-sessions, especially with younger students who might have never solved an IOI problem before. In 2004 the junior group was given the reactive task *Median Strength*, also from IOI 2000. The reason was that younger students are sometimes intimidated by reactive tasks, which are outside their usual experience, and also to show them that IOI problems are sometimes simpler than they first look. The task *Median Strength* is analysed in detail by Horváth and Verhoeff (2002).

For the curious, the name "B-session" is taken from a similar event introduced into the Australian mathematics olympiad programme in the early 1990s, where students discuss a difficult problem as a group and then individually write up proofs in exam conditions.

4. Team Events

The team event is a deliberate break from IOI-style problem solving. The goals of the team event are:

- to encourage teamwork, which traditional informatics olympiads for secondary school students do not do;
- to encourage students to look outside the narrow focus of informatics olympiads and explore the larger world of computer science;
- to spend an afternoon doing something noisy and fun!

Each team event is based around a topic that is accessible, new to many students, and that can support short team-based puzzles. The following example uses classical cryptography and cryptanalysis, which was the topic of the most recent team event in December 2007.

A typical team event runs for a single afternoon (usually $3\frac{1}{2}$ hours), and is structured as follows:

- The organisers choose a topic as described above.
- The students are given a one hour lecture on this topic, with a focus on solving real problems. For the cryptography event, students were shown how to encrypt messages using a number of classical ciphers, and also how to crack messages written using these ciphers.

- After the lecture, students move into the laboratory and are organised into small teams. They are told that a prize has been hidden somewhere in the building, and they are given their first puzzle to solve.
  In the cryptography event, this first puzzle was a short message coded using a shift cipher (rotate the alphabet by a fixed amount). The puzzle was simple to solve because there were only 26 possible "keys" to try (i.e., 26 possible rotations).
- When a team has solved a puzzle, they bring their solution to a staff member; if it is correct then the team is given a new puzzle. For the cryptography puzzles, each solution was the English plaintext obtained from a coded message.
  The puzzles essentially work their way through the topics in the lecture. For instance, a subsequent puzzle in the cryptography event contained a message encrypted using a general substitution cipher, where students needed to use frequency analysis and human guesswork to break the code. Later puzzles involved a Vigenère cipher (requiring more sophisticated frequency analysis and human techniques) and a linear stream cipher (requiring some mathematics to solve).
- Once a team has solved every puzzle, they must work out where the prize has been hidden. The secret is usually contained in a "master puzzle", which involves information that teams have collected throughout the event.
  For the cryptography event, the master puzzle was a one-time pad whose key was not random, but was instead a combination of names. Since the event took place just after a federal election, all of the previous messages involved quotes from former prime ministers; the key to the final puzzle was formed from the names of these prime ministers. Students were not told the structure of the key, and had to find it through experimentation, guesswork and of course the occasional hint.
- Once a team has cracked the master puzzle, they run away and return with the hidden prize!

Teams are allowed several computers (one per person), and the puzzles are usually chosen in a way that encourages teamwork. For instance, when cracking a substitution cipher, one part of the team may be coding up a frequency analysis program while another part is coding up a real-time substitution tool to help with the guesswork. Eventually the entire team will be crowded around a screen looking at pieces of the message and guessing at English words. For other puzzles, such as the linear stream cipher, some team members might be working on cracking the code with a program while others might be working on pen and paper.

Since the team event is meant to be informal and fun, staff members are usually quite liberal with hints, particularly for teams who are behind the others.

All of the topics to date have been taken from the field of information security. This is partly because information security lends itself well to fun team-based puzzles, and partly because the author spent some years working in the field. Over the three years that the team events have run, topics have included:

- *Classical cryptography and cryptanalysis.* This essentially follows the first chapter of (Stinson, 2002), which runs through a number of classical ciphers and describes how to crack them. Some of the ciphers require a known plaintext attack (where

students know in advance a small piece of the message); in these cases students are given strong hints that allow them to guess what the first word of the plaintext might be.

Cryptography and cryptanalysis is extremely popular, and was used in both 2005 and 2007 (though with a different set of puzzles). Specific topics include shift ciphers, affine ciphers, substitution ciphers, Vigenère ciphers, and linear stream ciphers.

- *Secret sharing schemes.* This runs through some of the different ways in which a group of people can share a secret key amongst themselves, in a way that requires "enough" participants to join forces before the key can be recovered. The puzzles involve (i) retrieving keys based on the information given by several participants, and (ii) *cheating*, where one participant provides false information but is then able to find the correct key without the knowledge of other participants.

  Donovan (1994) provides a good description of several well-known schemes and how to cheat using these schemes. Specific schemes used in the team event include the basic modular sum-of-numbers scheme, Shamir's polynomial interpolation scheme, and Blakley's 2-dimensional geometric scheme.

5. Practical Issues

It is worth pausing to consider the difficulty of running each of the aforementioned events, including preparation time and technical requirements.

5.1. *Running a Codebreaker*

The codebreaker is the most demanding of the events described here (though also one of the most useful). To begin, the codebreaker cannot be run using a traditional contest website – instead it needs its own special web software. With proper use of configuration files this software only needs to be written once – the Australian codebreaker software was written in April 2005, tidied up in December 2005 and has remained more or less the same ever since. If the organisers already have contest software that is well-modularised, writing the codebreaker software should be a tedious but straightforward task.

Beyond the one-off effort in creating the codebreaker software, each individual event requires significant preparation:

- A set of incorrect solutions must be written for each task. As mentioned in Section 2, this process should ideally involve many different people.
- A *sanity checker* must be written for each task; this is a short program that checks whether an input file conforms to the task specifications. Sanity checkers must be written carefully, since (unlike in an ordinary contest) they are not just verifying the organisers' input files, but they are also verifying students' submissions. A weak sanity checker may allow students to gain points by merely submitting invalid test data, instead of finding real cases for which a solution breaks.

- A *universal evaluator* must be written for each task; this is a program that reads an input file (submitted by a student), reads an output file (created by an organiser's incorrect solution), and determines whether the output correctly solves the given input file. In most cases this universal evaluator must solve the task itself before it can check the given output. Often the universal evaluator is just an official solution that has been adapted to talk to the codebreaker software.

As a final note, the organisers must decide what to do about whitespace in the input files that students submit. Allowing too much whitespace may give students more room to break the solutions in unintended ways (such as creating massive input files that cause a program to time out, or by causing buffer overflows in solutions that read strings). On the other hand, being rigorous and insisting on no extra whitespace can make students grumpy when they submit trailing spaces or newlines by accident and lose points as a result. A simple workaround is for the contest website to strip unnecessary whitespace from any input files before they are processed.

5.2. *Running a B-Session*

A B-session is extremely simple to run. The organisers must choose a single problem, and one staff member must study the problem in detail so that she or he can chair the discussion. The exam component can be run using whatever system the organisers generally use for running contests.

5.3. *Running a Team Event*

Since the team event is fun and informal, there are few technical requirements. Typically some of the larger data files are hosted on a web server (such as long messages to be decrypted), and everything else is done on pen and paper – puzzles are printed and distributed by hand, and answers are checked manually by a staff member.

However, a team event does require a significant amount of preparation. The lecture must be written and a cohesive series of puzzles must be devised. More importantly, the solutions to these puzzles must be verified by a third party. This is of course true of any contest, but it is a particular concern for the team event – if one of the cryptography puzzles contains an error, teams could spend a long time getting nowhere before anybody suspects that something might be wrong.

6. Other Events

The Australian training schools feature a number of other events not described here. These include:

- *Mystery lectures*, where a guest lecturer talks for an hour on a pet topic from the larger world of computer science;
- *Proof and disproof sessions*, where students work as a group, alternating between formally proving good algorithms correct and finding counterexamples that show bad algorithms to be incorrect;

- *Game-playing events*, where students write bots to play a simple game and then play these bots against each other in a tournament;
- *Team crossnumbers*, fun events where teams of students work to solve crossnumber puzzles in which half the team has the "across" clues, half the team has the "down" clues, and all of the clues depend on one another (Clark, 2004).

It is hoped that the Australian training programme can remain dynamic and fresh, and the author looks forward to learning how other delegations work with their students in new and interesting ways.

References

Anido, R.O. and Menderico, R.M. (2007). Brazilian olympiad in informatics. *Olympiads in Informatics*, **1**, 5–14.

Burton, B.A. (2008). Informatics olympiads: Challenges in programming and algorithm design. In G. Dobbie and B. Mans (Eds.), *Thirty-First Australasian Computer Science Conference* (*ACSC 2008*). Wollongong, NSW, Australia. *CRPIT*, vol. 74. ACS, pp. 9–13.

Clark, D. (2004). Putting secondary mathematics into crossnumber puzzles. *Math. in School*, **33**(1), 27–29.

Cormack, G., Munro, I., Vasiga, T., Kemkes, G. (2006). Structure, scoring and purpose of computing competitions. *Informatics in Education*, **5**(1), 15–36.

Donovan, D. (1994). Some interesting constructions for secret sharing schemes. *Australas. J. Combin.*, **9**, 37–65.

Forišek, M. (2007). Slovak IOI 2007 team selection and preparation. *Olympiads in Informatics*, **1**, 57–65.

Horváth, G. and Verhoeff, T. (2002). Finding the median under IOI conditions. *Informatics in Education*, **1**, 73–92.

Opmanis, M. (2006). Some ways to improve olympiads in informatics. *Informatics in Education*, **5**(1), 113–124.

Stinson, D.R. (2002). *Cryptography*: *Theory and Practice*. Chapman & Hall/CRC, 2nd edition.

B.A. Burton has been the director of training for the Australian informatics olympiad programme since 1999, and before this was a trainer for the mathematics olympiad programme. His research interests include computational topology, combinatorics and information security, and he currently works in the murky world of finance.

Olympiads in Informatics, 2008, Vol. 2, 16–36

Creating Informatics Olympiad Tasks: Exploring the Black Art

Benjamin A. BURTON
Department of Mathematics, SMGS, RMIT University
GPO Box 2476V, Melbourne, VIC 3001, Australia
e-mail: bab@debian.org

Mathias HIRON
France-IOI 5, Villa Deloder, 75013 Paris, France
e-mail: mathias.hiron@gmail.com

Abstract. Each year a wealth of informatics olympiads are held worldwide at national, regional and international levels, all of which require engaging and challenging tasks that have not been seen before. Nevertheless, creating high quality tasks can be a difficult and time-consuming process. In this paper we explore some of the different techniques that problem setters can use to find new ideas for tasks and refine these ideas into problems suitable for an informatics olympiad. These techniques are illustrated through concrete examples from a variety of contests.

Key words: programming contests, task creation.

1. Introduction

Like writing music or proving theorems, making new informatics olympiad tasks is a highly creative process. Furthermore, like many creative processes, it is difficult to create tasks on a tight schedule. With a rich yearly programme of national and international contests however, this is a problem that many contest organisers face. How can organisers come up with fresh tasks, year after year, that are not simple variations of problems that students have already seen?

We do not hope to solve this problem here – this paper does not give an "algorithm" for creating good tasks. What we do offer is a range of techniques for creating new tasks, drawn from the collective experience of the scientific committees of two countries. Of course each contest organiser has their own methods for setting tasks; the hope is that readers can blend some of these techniques with their own, and that this paper can encourage a dialogue between contest organisers to discuss the many different methods that they use.

We begin in Section 2 by describing the qualities of a "good" informatics olympiad task, and in Section 3 we run through different characteristics that make one problem different from another. Section 4 covers the difficult first step of finding an original idea; the methods offered span a range of creative, ad-hoc and methodical techniques. In Section 5

we follow up with ways in which an original idea can be massaged and modified into an idea suitable for a real contest. Finally Section 6 reformulates the ideas of the earlier sections in a more abstract, "algorithmic" setting.

Creating new tasks is only part of the work required to organise a contest. Diks *et al.* (2007) discuss the work that follows on from this, such as task write-ups, analysis, documentation, solutions, test data, and final verification.

2. What Makes a Good Task?

Before we set out to create new tasks, it is important to understand what we are aiming for. In an informatics olympiad, the target audience (talented high school students), the tight time frame (typically at most five hours for a contest) and the need to fairly rank competitors all place restrictions on what tasks can be used.

The following list identifies some features of a good informatics olympiad task. This list is aimed at contests modelled on the International Olympiad in Informatics (IOI) – other programming contests have different goals, and so their tasks might have different needs. Additional notes on the suitability of tasks can be found in Diks *et al.* (2007) and Verhoeff *et al.* (2006).

- It should be possible to express the task using a problem statement that is (relatively) short and easy to understand. The focus of the task should be on problem solving, not comprehending the problem statement. It is also nice to include some tasks that relate to real-world problems, though this is by no means necessary.
- Ideally the algorithm that solves the task should not directly resemble a classic algorithm or a known problem. It might be a completely original algorithm, it might be a modified version of a classic algorithm, or it might resemble a classic algorithm after some transformation of the input data.
  An example of transforming a classic algorithm is *Walls* from IOI 2000. Competitors essentially read a planar graph from the input file. If they convert this graph into its dual, the problem becomes the well-known breadth-first search, with some minor complications involving multiple starting points.
- The task should support several solutions of varying difficulty and efficiency. This allows weaker students to gain partial marks by submitting simple but inefficient solutions, while stronger students can aim for a more difficult algorithm that scores 100%.
- The official solution should allow a reasonably concise implementation (at most a few hundred lines of code). It should also be possible to estimate during the pen-and-paper design stage whether this solution can score 100% for the given time and memory constraints.
- Ideally the official solution should also be the best known solution for the task, though whether this is desirable may depend on the intended difficulty of the task.
- For contests aimed at experienced students, it is nice to have tasks where it is not obvious in advance what category (such as dynamic programming or graph theory)

the desired algorithm belongs to. It is also nice to have solutions in which two or more different algorithms are welded together.

3. Characteristics of Problems

In this section we run through the different types of characteristics that a task can have. This is useful for categorising problems, and it also highlights the different ways in which a task can be changed.

Although this paper is not concerned with categorisation per se, understanding the different characteristics of old tasks can help problem setters create new tasks that are different and fresh. This categorisation is particularly important for the techniques of Subsection 4.4, and it plays a key role in the algorithmic view of task creation described in Section 6.

Examining characteristics of tasks also helps us to experiment with the many different ways in which a task can be changed. Changing tasks is a critical part of the creation process; for example, tasks can be changed to make them more interesting or original, or to move them into a desired difficulty range. The process of changing tasks is discussed in detail in Section 5.

Each task has many different characteristics; these include characteristics of the abstract core of the task, of the story that surrounds the task, and of the expected solution. These different types of characteristics are discussed in Subsections 3.1, 3.2 and 3.3 respectively. This list does not claim to be exhaustive; readers are referred in particular to Verhoeff *et al.* (2006) for additional ideas.

3.1. *Characteristics of the Abstract Core*

Here we reduce a task to its abstract, synthesised form, ignoring the elements of the story that surrounds it. Different characteristics of this abstract form include the following:

- *What kinds of objects are being manipulated*?
  Examples of objects include items, numbers, intervals, geometric objects, and letters. Some tasks involve sets or sequences of these basic objects; these sets or sequences can be arbitrary (such as a sequence of numbers read from the input file) or described through rules (such as the set of all lattice points whose distance from the origin is at most d). A task may involve more than one kind of object.
- *What kinds of attributes do these objects have*?
  Attributes can be numerical, such as the weight of an edge in a graph, or the score for a move in a game. They can also be geometric, such as the coordinates of a point or the diameter of a circle; they can even be other objects. Attributes can have static or dynamic values, and can be given as part of the problem or requested as part of the solution.
  What are the relationships between these objects?
  Relationships can be arbitrary (such as the edges of a given graph forming a relationship on the vertices), or they can be described by rules (such as intersections,

inclusions, adjacencies, or order relations). They can relate objects of the same kind or of different kinds, and they can even be relations between relations (such as relationships between edges of a graph). Relationships can be constraints that one object places upon another (such as a maze, which constrains the coordinates of objects inside it), or they can constrain the successive values of an attribute (such as a graph in which some object can only move to adjacent vertices).

- *What kind of question is asked about these objects?*
  The question might involve finding a specific object (or set of objects) that has some property, or that maximises or minimises some property. It might involve aggregating some attribute over a set of objects, or setting the attributes of each object to reach a particular condition.
- *Does the task ask just one question, or does it ask many questions over time?*
  The same question can often be asked in two ways: (i) asking a question just once, or (ii) asking it many times, where the objects or their attributes change between successive questions. An example of the latter type is *Trail Maintenance* from IOI 2003, which repeatedly asks for a minimal spanning tree in a graph whilst adding new edges between queries. These changes might be given as part of the input or they might depend on the previous answers, and questions of this type can lead to both interactive and non-interactive tasks.
- *What general constraints are given on the structures in the task?*
  Examples for graph problems might include general properties of the graph, such as whether it is directed, sparse, acyclic or bipartite. Constraints for geometric problems might include the fact that some polygon is convex, or that no two lines in some set are parallel.

Some of these characteristics – in particular, the objects, attributes and relationships – are reminiscent of other fields of computer science, such as object-oriented programming or database design. However, in the more flexible domain of informatics olympiad tasks, these characteristics can cover a much broader scope.

3.2. *Characteristics of the Story*

The same abstract task can be presented in many different ways, by wrapping it inside different stories. Selecting a good story can have a great influence on the quality and accessibility of a task. As discussed in Section 5, it can also affect the difficulty of the task through the way it hides elements of the core problem. Characteristics of the story include the following:

- *What real world item do we use to represent each object?*
  Typical items include roads, buildings, cows, farmers, strings; the list goes on. Rectangles might become cardboard boxes; a grid of numbers might become heights in a landscape. The possibilities are immense.
- *What kind of measures do we use for each attribute?*
  Numerical attributes might represent time, weight, age, quantity or price. Symbolic attributes could represent colours or countries; geometric attributes such as coordinates or diameter could describe a location on a map or the range of a radio tower.

- *How do we justify the relationships between objects?*
  This typically depends upon how the objects themselves are represented. For instance, if the objects are people then relationships could be described as friendships or parent/child relationships. Often the descriptions of relationships follow naturally from the descriptions of the objects themselves.
- *How do we explain why the question needs to be solved?*
  This explains the motivation for the task, and often provides an overall storyline for the problem.

3.3. *Characteristics of the Solution*

The solution of a problem is its most important aspect – a large part of what makes a task interesting is how interesting and original the solution is. Unfortunately it can be difficult to change the solution directly without having to make significant changes to the entire task. Characteristics of the solution include:

- *What domains of algorithmics are involved?*
  Examples include graph theory, dynamic programming and computational geometry. Verhoeff *et al.* (2006) provide an excellent reference of domains that are relevant for the International Olympiad in Informatics (IOI).
- *What mathematical observations need to be made?*
  Often a solution relies on some property of the task that is not immediately obvious. For instance, students might need to realise that a graph contains no cycles, that some attribute can never be negative, or that a simple transformation can always convert a "sub-optimal" answer into a "better" answer.
- *What is the main idea of the algorithm?*
  This is the core of the solution, and describes the key steps that are eventually fleshed out into a detailed algorithm and implementation. Is it a greedy algorithm? Does it build successive approximations to a solution? Does it solve a problem by iteratively building on solutions to smaller problems? What are the important loop invariants?
- *What data structures can be used?*
  In some cases the data structures form an integral part of the algorithm. In other cases, clever data structures can be used to improve an algorithm, such as implementing Dijkstra's algorithm using a priority queue. For some algorithms there are several different choices of data structure that can lead to an efficient solution.

4. Finding a Starting Point

One of the most difficult steps in creating a new task is finding a starting point – that elusive new idea for a task that is like nothing students have seen before. In this section we explore some of the different techniques that can be used to find an original new idea.

The techniques presented here are based on personal experience. This list was compiled by approaching several problem setters from the authors' respective countries, and

asking them to reflect on past contests and analyse the different processes that they used to generate ideas.

Many of the examples in this section are taken from the French-Australian Regional Informatics Olympiad (FARIO), the France-IOI training site, and the Australian Informatics Olympiad (AIO). Full texts for all of these problems are available on the Internet: the FARIO problems from `www.fario.org`, the France-IOI problems from `www.france-ioi.org` (in French), and the AIO problems from the Australian training site at `orac.amt.edu.au`. The last site requires a login, which can be obtained through a simple (and free) registration.

Finding a new idea is of course only the beginning of the task creation process, and is usually followed by a round of modifications to improve this initial idea. These modifications tend to follow a similar pattern regardless of how the initial idea was formed, and so they are discussed separately in Section 5.

4.1. *Looking Around*

One of the most common techniques, though also one of the least predictable, is to "look around" and take inspiration from things that you see in real life. An alternative is to find somebody unfamiliar with computer science or programming contests, and to ask them for ideas.

An example is *Guards* (FARIO 2006), which describes a circular area with roads entering at different points on the edge of the circle (Fig. 1). The task is then to place guards at some of these roads so that every road is "close enough" to a guard, and to do this using as few guards as possible in linear time. This turned out to be a relatively tricky sliding window problem, and has proven useful for training in the years since. The inspiration for this problem was the 2006 Commonwealth Games in Melbourne, where one author had a large part of his suburb barricaded off for the netball and hockey matches.

Another example is *Banderole* (France-IOI), which gives a line of vertical sticks and asks for the number of integer positions in which a rectangle can be balanced (Fig. 2). The idea for this problem came from a France-IOI team dinner, where people were balancing the objects in front of them (plates, cider bottles and so on).

Looking around for inspiration has some disadvantages. It requires a great deal of effort trying to solve the problems that you create, since you do not have a solution in

Fig. 1. The circle of roads for the task *Guards* (FARIO 2006).

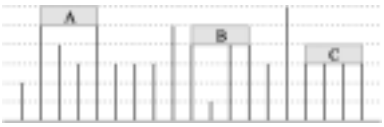


Fig. 2. Examples of balanced rectangles for the task *Banderole* (France-IOI).

mind in advance. Furthermore (as is frequently found in the real world) these ideas can often lead to problems that are NP-hard, though the techniques of Section 5 can be used to convert them into more tractable tasks.

However, the same cause of these difficulties – that you do not have a solution ready in advance – means that this method can sometimes lead to highly original problems that do not rely on simple modifications of standard algorithms. One of the first author's favourite problems is *Citizenship*, from the Australian team selection exam in 2006; this gives rules for gaining and losing citizenship of different countries, and asks for the largest number of countries for which you can hold citizenship simultaneously. The solution involves an interesting representation of the input data and unusual ad-hoc techniques on directed graphs. The motivation for this problem was a dinner conversation at the 2006 Australian Linux conference (which was held in Dunedin, New Zealand).

There are many places to "look around" for inspiration. One is your immediate environment – focus on a nearby object, or an action that somebody performed. Another is to recall events in your life or a friend's, or even in the news. Plots from books or movies can also be used as a starting point.

Once you have picked a specific object or event, you can focus on that object or event and work to create a task around it; this is usually more efficient than looking in lots of different places until inspiration comes. Restricting your attention to a given branch of algorithmics can also help; for instance, you might focus on creating a graph problem involving light bulbs. Working with unusual objects, actions or events can help to create original ideas.

4.2. *Drawing on the Day Job*

Another technique employed by many problem setters is to use tasks that arise in their daily work. For instance, you might be working on a large and complex research problem but find that a small piece of this problem is just the right difficulty for an informatics olympiad. Alternatively, you might be reading through a research paper and find an interesting algorithm that can be adapted for a programming contest.

One problem of this type is *Giza* (FARIO 2007), which was inspired by real-world work in tomography. This task is a discrete 2-dimensional version of the general tomography problem. Specifically, it asks students to recover a secret image formed from grid cells, given only the cell counts in each row, column and diagonal (Fig. 3).

Another example is *Frog Stomp*, from the 2007 Australian team selection exam. This is an output-only task that asks students to find "sufficiently short" integer sequences with particular properties. This problem arose as part of a larger research project in eliminating

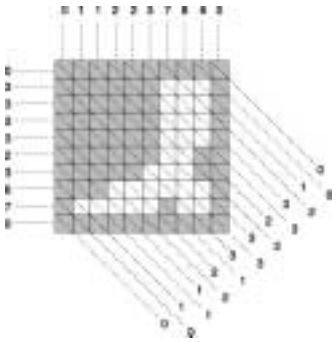


Fig. 3. A secret image and cross-section totals for the task *Giza* (FARIO 2007).

statistical biases in financial data, and was easily extracted from its financial setting into a standalone mathematical task.

In contrast to "looking around", with this technique you already begin with a solution – either you have already solved it yourself, or you have read the algorithm in a research paper. While this is convenient, it also risks underestimating the difficulty of such tasks. Personal research projects typically involve ideas that you have worked on for months, and research papers rarely give an indication of *how* they arrived at an algorithm or how difficult this was. It is usually best to give the problem to other people to solve, in order to gain a better estimate of its true difficulty.

4.3. *Modifying a Known Algorithm*

One of the simpler techniques for producing new tasks is to begin with a standard algorithm (such as quicksort or a breadth-first search) and set a task that modifies it in some way.

An example is *Chariot Race* (FARIO 2004), which requires a modification to Dijkstra's algorithm for the shortest path through a graph. In this task the problem setters began with Dijkstra's algorithm and set out to remove the "no negative edge weights" constraint.

If we simply allow negative edge weights, this creates difficulties with infinite negative cycles. To avoid this, the task was changed so that, instead of edges with fixed negative weights, we allow *wormholes* – edges that "travel back in time" by dividing the current total distance by two (Fig. 4). By making this division round to an integer, the problem setters were able to avoid infinite negative cycles entirely.

The solution to Chariot Race is essentially Dijkstra's algorithm with adaptations to deal with (i) the fact that the total distance can decrease after following an edge, and (ii) the fact that the precise *change* in distance when following a wormhole is not a constant, but instead depends upon the total distance so far. Good students should be able to deal with both adaptations easily enough, and indeed this turned out to be the easiest problem on the FARIO contest paper.

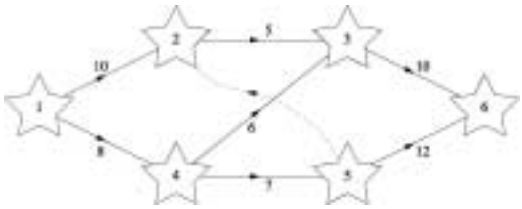


Fig. 4. A graph for *Chariot Race* (FARIO 2004), with the wormhole as a dotted arrow.

The advantages of beginning with a standard algorithm are that it is easy to create new tasks (there are plenty of standard algorithms, and plenty of modifications to make), and that you typically have a good idea of what the solution will look like in advance. The main disadvantages are that the resulting tasks will often be similar to well-known problems, and that good students should see through this easily. It is therefore difficult to create highly original problems using this technique.

It should be noted that the algorithm you begin with does not need to be one of the fundamental algorithms that appear regularly in olympiad problems. There are plenty of standard algorithms that are less commonly taught; examples include graph vertex connectivity, permutation generation, and union-find data structures. As a possible starting point, Skiena (1998) offers an excellent catalogue of standard algorithms and their variants, and Verhoeff *et al.* (2006) propose a detailed syllabus of topics that relate to the IOI.

4.4. *Filling the Holes in a Domain*

Here we describe a technique that focuses on a particular domain, such as shortest path problems or binary trees, where you already have a pool of problems in stock that you cannot reuse. This technique is useful when preparing training sessions for experienced students who have seen many problems before; it can also help create new tasks for programming contests.

The technique essentially begins with a large pool of tasks, examines the characteristics of these tasks, and then forms new combinations of these characteristics. In detail:

- The first step is to compile a thorough list of "old" tasks in the domain of interest. This can be time consuming, but it is important to include as many tasks as possible – not only does this list remind you of tasks that you cannot reuse, but it also serves as a basis for creating new tasks in the domain.

  To work with this list efficiently, it helps to have each task clearly in mind. A suggestion is to synthesise each task and its solution in a few sentences; in many cases a quick drawing can also help. As a side-effect, compiling this list helps give a clear view of the types of problems that can be created in this domain, and new ideas might come naturally as a result.
- The next step is to examine the different characteristics of the tasks in this list; examples of such characteristics can be found in Section 3. The aim is to find char-

acteristics that tasks have in common, and also characteristics in which tasks differ. In this way, each task can be described as a combination of different characteristics.
- Creating new tasks is then a matter of finding combinations of characteristics that have not yet been used. Blending the characteristics of two or more different problems in the domain can often lead to a meaningful (and interesting) new task.

To illustrate, we can apply this technique to the domain of sweep-line algorithms and sliding windows. A list of 30 old problems was compiled by the second author; although the individual problems and solutions are too numerous to list here, the main characteristics are as follows:

(i) The objective may be to find a point, a set of points, an interval, or a set of intervals satisfying some given properties. Alternatively, it may be to aggregate some value over all points or intervals satisfying some given properties.
(ii) The points or intervals may need to be chosen from a given list, or they may be arbitrary points or intervals on either a discrete or continuous axis.
(iii) The properties used to select these points or intervals may involve relationships amongst these objects, or they may involve relationships between these objects and some entirely different set of objects. In the latter case, this second set of objects may itself involve either points or intervals.
(iv) The properties used to select these points or intervals may be purely related to their location on the axis, or they may involve some additional attributes.
(v) When intervals are manipulated, intervals that overlap or contain one another may or may not be allowed.
(vi) In the story, the objects may appear directly as points or intervals on a given axis, or they may be projections of some higher-dimensional objects onto this axis. The axis may represent location, time, or some other measure.
(vii) Solutions may involve sweep-line algorithms, sliding windows, or possibly both.

Having listed the most important characteristics within this domain, we can now choose a new combination of values for these characteristics. We will ask for (i) an interval, chosen on (ii) a continuous axis. We will require this interval to satisfy some relationship involving (iii) a different set of intervals, given in the input file. Intervals will (iv) be assigned a colour from a given list of possible colours, and (v) no rules will prevent intervals from intersecting or containing one another. The story will use (vi) circles as its primary objects, where the intervals describe the angles that these circles span when viewed from the origin.

Now that a choice has been made for each characteristic, most of the elements of the problem are set. Fleshing this out into an abstract task, we might obtain the following:

> *A set of circles is given on a two dimensional grid. Each circle has a colour identified by an integer. The task is to choose two half-lines starting from the origin, so that at least one circle of each colour lies entirely in the region between these lines, and so that the angle between these lines is as small as possible.*

This task is illustrated in Fig. 5. A story still needs to be built around this idea, but we have a good start for a new task that is different from the 30 problems originally listed.

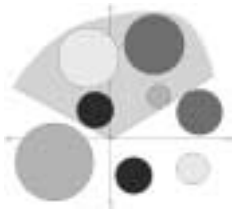


Fig. 5. Creating a new task by combining the different characteristics of old tasks.

This technique gives an efficient way of creating many new problems in a short time within a specific domain. However, it tends to produce tasks that are not truly original, since they always share the characteristics of existing problems.

One way to obtain more original problems with this technique is to choose values for some characteristics that do not appear in *any* of the listed problems. For instance, in the example above, the objects in the original list of 30 problems include points, intervals, segments, squares, rectangles, right-angled triangles, and circles. We might therefore try parallelograms or ellipses for something new.

4.5. *Building from Abstract Ideas*

If you don't have a day job or other external sources of inspiration, another way to generate new ideas is to draw random things on a blank piece of paper. Here we describe a technique that begins with vague sketchings and gradually refines them into a usable problem.

In Subsection 3.1 we describe several kinds of objects that can appear in a task. To search for ideas, you can pick one kind of object at random and draw some instances of it on a piece of paper. You might also choose a second kind of object and do the same. You can then search for a task that involves these objects; this might require you to find some attributes, relationships, and a question about these objects that you can ask. Drawing diagrams to represent the different elements of the task can help ideas to come.

During the initial stages of this process, what you manipulate is not always a precise problem with a solution. As you add objects, attributes and relationships one after another, you might not have specific algorithms in mind but rather vague ideas of where you are heading. Each time you define another characteristic of the task, you obtain a more precise idea of the problem, and a better idea of what kind of solution you might end up with. If at some stage you are unhappy with the direction the problem is taking, you can go back and alter one of the characteristics that you defined earlier. Little by little, you get closer to a usable problem.

To illustrate this process we give a concrete example; some of the intermediate sketchings are shown in Fig. 6. We might begin by drawing circles on a piece of paper, with some intersections and inclusions. Perhaps it reminds us of a graph problem we have seen before; to avoid this we decide to add another kind of object. We choose lines, and think about ideas such as finding the line that intersect the most circles.

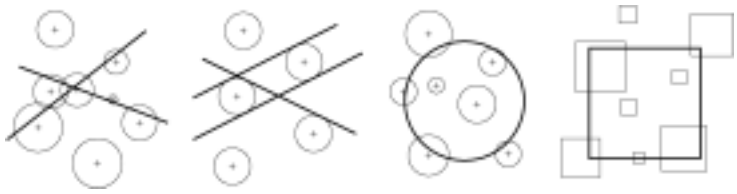


Fig. 6. Successive diagrams used whilst creating a task from abstract ideas.

However, this also reminds us of problems we have seen before; we therefore try to replace lines with circles, and look for the position of a new circle that intersects the most circles from a given set. This might become a relative scoring task, which is not what we are looking for today. Replacing circles with squares yields an interesting problem for which we can find several solutions of varying complexities. We massage it a little further by changing squares into rectangles and giving them a "value" attribute, and obtain the following task:

> *You are given N rectangles ($N \leqslant 500$) whose sides are parallel to the x and y axes. Each rectangle is described by four integer coordinates between 0 and 1 000 000 and a value between 1 and 1 000. Given a width W and height H ($0 \leqslant W, H \leqslant 100\,000$), your task is to find a new rectangle of size $W \times H$ for which the total value of all rectangles it intersects or contains is as large as possible.*

This task is not a great one but can be good enough for some contests. One solution involves a sweep-line algorithm with an interval tree.

4.6. *Borrowing from Mathematics*

Although ideas for tasks can come from many different disciplines, some branches of mathematics are particularly useful for creating informatics olympiad problems. Here we focus on tasks derived from the mathematical field of *combinatorics*.

To summarise, combinatorics problems typically involve counting things or arranging things according to certain rules. Examples of combinatorics problems are (i) counting different colourings of an $n \times n$ chessboard, (ii) counting the number of different triangulations of an n-sided polygon, and (iii) creating latin squares, which are $n \times n$ grids filled with the numbers $1, 2, \ldots, n$ so that each row and each column contains every number once.

Combinatorics problems can often lead to interesting or unusual dynamic programming tasks, because they both share a core dependency on *recurrence relations*: formulae that solve a larger problem in terms of one or more smaller problems.

For instance, consider example (ii) above. Let T_n be the number of ways of triangulating an n-gon; in Fig. 7 it can be seen that $T_6 = 14$. In general, T_n is the famous sequence of Catalan numbers; see (van Lint and Wilson, 1992) for some of the many, many other things that it counts.

The sequence $T_1, T_2, \ldots$ obeys a recurrence relation, which we can see as follows. Choose the bottom edge of the hexagon, and consider the possible triangles that this edge

Fig. 7. The 14 different ways of triangulating a hexagon.

Fig. 8. The four possible triangles that might include the bottom edge.

could belong to. There are four possibilities, illustrated in Fig. 8. In the first and last diagrams we must still triangulate the white 5-gon, whereas in the middle two diagrams we must still triangulate the white 3-gon and the white 4-gon. Thus $T_6 = T_5 + T_3T_4 + T_4T_3 + T_5$. In general, a similar argument shows that

$$T_n = T_{n-1} + T_3T_{n-2} + T_4T_{n-3} + \cdots + T_{n-3}T_4 + T_{n-2}T_3 + T_{n-1}.$$

It is easy to turn this mathematical recurrence into a dynamic programming problem. One way is to ask precisely the same question – count the number of triangulations of an n-gon (possibly writing the answer mod 1000 or similar to avoid the inevitable integer overflows).

Another way is to turn it into an optimisation problem – give each triangulation a score, and ask for the largest score possible. For example, we could place a number at each vertex, and score each edge of the triangulation according to the product of the two numbers that it joins. The recurrence relation would change a little (for instance, the sum would become a maximum), but its overall structure would remain the same. This is precisely the task *Polygon Game* from the 2002 Australian team selection exam, and it was created in precisely this way. See (Burton, 2007) for a more detailed analysis of this problem.

Combinatorial recurrence relations are plentiful, and can be found from several sources:

- They frequently appear in the study of generating functions (Wilf, 1994). The problem *Architecture* (FARIO 2007) was based on a problem described by Wilf that counts the number of n-square polyominoes with a certain convexity property (where polyominoes are "generalised dominoes" formed by joining several squares together, similar to Tetris blocks). The final problem *Architecture* asks students to maximise a "prettiness" score for n-block buildings with this same convexity property.
- The online encyclopedia of integer sequences (Sloane, 2008) is full of interesting combinatorial sequences and recurrence relations; simply browsing through the encyclopedia can yield interesting results.

- Finally, it is useful to look through mathematics competitions, where combinatorics is a popular topic. There are many compilations of mathematics competition problems available; see (Kedlaya *et al.*, 2002) and (Larson, 1983) for examples.

Combinatorics is of course not the only branch of mathematics that can yield interesting problems. We focus on it here because many combinatorial problems are easily accessible to students with no formal mathematics background. For different examples the reader is referred to (Skiena and Revilla, 2003), who work through problems from several different branches of mathematics.

4.7. *Games and Puzzles*

Another concrete source of ideas is games and puzzles. Many obscure games can be found on the Internet, and puzzles are readily available from online sources such as the `rec.puzzles` newsgroup, and from friends' coffee tables.

Games and puzzles are useful in two ways. On the one hand, they provide ready-made tasks such as playing a game optimally or solving a puzzle using the smallest number of moves. On the other hand, they can supply interesting sets of objects and rules that can act as starting points for other tasks, using techniques such as those described in Subsection 4.5.

An example of a game-based problem is *Psychological Jujitsu* (AIO 2006), which involves a card game where players bid for prizes. In the real game players cannot see each others' bids. To make the problem solvable, the AIO task assumes that a player can see their opponent's bids in advance, and then asks for a sequence of bids that allows the player to beat their opponent with the largest possible margin. The solution is an original (though relatively simple) ad-hoc algorithm.

A problem that began as a game but grew into something different is *Collier de Bonbons* (France-IOI). The original source was *Bubble Breaker*, an electronic game involving stacks of bubbles of different colours, in which clicking on a group of two or more bubbles of the same colour makes those bubbles disappear. The game was changed to use a single stack, which then became a single cycle. The task was then changed so that students must remove all of the bubbles in the cycle; bubbles can be removed individually if required, but this must be done as few times as possible. The cycle of bubbles was changed to a candy necklace, and the final solution is a nice example of dynamic programming.

One difficulty of using games and puzzles is that solving them is often NP-hard (otherwise they are not interesting for humans to play!). For this reason, tasks derived from games or puzzles often need to be simplified or constrained before they can be used in a programming contest.

4.8. *Final Notes*

We close this section with some final suggestions when searching for an initial idea:

- *Restrict your search to a single theme or domain.*
  To generate ideas faster, it is often useful to restrict the search space. For instance, you might search for a task relating to trains (a restriction on the story), a task

involving hash tables (a restriction on the solution), or a task involving geometry (a restriction on the underlying objects). One benefit of reducing the search space is that you can quickly enumerate the classic problems in that space, which in turn makes it easier to find ideas outside these classic areas.

- *Get other people involved in the search process.*
  If you explain an idea to someone, they can often suggest improvements that you might not have considered yourself. For instance, they could suggest a different way to present the task, or add originality by modifying the task in a new direction. Moreover, explaining a task to a seasoned problem solver can occasionally give an unexpected result: by misunderstanding your explanation, they might solve the wrong task and end up with a different, more interesting version of your initial idea!
- *Be proactive!*
  Instead of only creating tasks when you need them, it helps to always be on the lookout for new ideas. It is worth keeping a file or notebook of ideas that you have had, or interesting papers you have read. Even if you just write one or two lines for each idea, it is far easier to browse through your notebook when you need new tasks than try to remember everything you thought of over the past months.

5. Improving the Task

In the experiences of the authors and most of the problem setters approached for this paper, the initial idea is typically not the final version of a task. Instead tasks are repeatedly modified, passing through several intermediate versions, until problem setters are happy with both the originality and the difficulty of the problems and their solutions. In this section we explore this modification process in more detail.

There are several reasons a task might need modification. It might be missing one of the qualities of a good problem as outlined in Section 2. On the other hand, it might have all of these qualities but not fall within the required difficulty range. Sometimes individual tasks are fine but the contest as a whole needs rebalancing.

In Subsection 5.1 we examine the ways in which tasks are changed one step at a time. Subsection 5.2 discusses modifications with a particular focus on changing the difficulty, and Subsection 5.3 looks at the related issue of adding and removing subproblems. In Subsection 5.4 we discuss the process of modifying a task to fit an intended solution, and offer some general advice when searching for solutions to candidate tasks.

5.1. *Elementary Changes*

If a task is not perfect, it does not make sense to throw it away completely. Initial ideas with nice properties are hard to find, and problem setters like to preserve these properties where possible by keeping their changes small. In this way, the modification process is often a succession of small changes, where good changes are kept and bad changes are undone, until eventually all of the concerns with a task have been addressed.

It is frequently the case that a single "small change" is a result of modifying a single characteristic of the task; for instance, the principal objects of the task might change from rectangles to circles. Sometimes other characteristics must change as a result; for instance, the old attributes of position, height and width might need to become centre and radius instead.

A more interesting example is *Chariot Race* (FARIO 2004), discussed earlier in Subsection 4.3. The initial idea for this task was Dijkstra's algorithm; the first change was to alter the properties of the graph to allow negative edges. In order to preserve some properties of the original idea (in this case the general structure of Dijkstra's algorithm), some other changes were needed – in particular, it was necessary to replace negative edges with edges that "divide by two and round to an integer". See Subsection 4.3 for details.

These observations suggest a general technique for modifying a task. Examine the various characteristics of the task, as outlined in Section 3. Then try changing these characteristics one at a time, beginning with those characteristics that take the most common or uninteresting values, and keep the changes that work best.

5.2. *Changing the Difficulty*

It is often the case that a task is original and interesting but simply pitched at the wrong audience. Here we examine some concrete ways in which the difficulty of a task can be changed.

If a task is too difficult, simplification techniques such as those described by Ginat (2002) and Charguéraud and Hiron (2008) can be applied. Although these techniques are designed to aid problem solving, they have the side-effect of generating easier versions of a task. The general idea behind these techniques is to remove or simplify one or more dimensions of a task.

For example, consider the problem *Giza* (FARIO 2007), discussed earlier in Subsection 4.2. The original idea for *Giza* was the general tomography problem (recovering the shape and density of a 3-dimensional object from several 2-dimensional x-rays). The problem was simplified by removing one dimension (making the object 2-dimensional) and simplifying others (allowing only three x-rays, and restricting the densities of individual grid cells to the values 0 or 1).

On the other hand, suppose we wish to increase the difficulty of a task. One obvious method is to add dimensions to the problem. For instance, consider the task *Speed Limits* from the 2002 Baltic Olympiad in Informatics. Here students are asked to find the fastest route for a car travelling through a network of intersections and roads, where some roads have an additional attribute (a "speed limit") that changes the speed of the car. The solution is a classical Dijkstra's algorithm, but on an unusual graph where each (intersection, current speed) pair defines a vertex. In this way the extra dimension of "speed limit" increases the difficulty of the task, since students must find the hidden graph on which the classical algorithm must be applied.

Some other ways of increasing the difficulty include:

- Changing how some of the dimensions are described in the story. For example, a problem involving x and y coordinates could be reformulated so that x and y

become time and brightness. Although the abstract task stays the same, the unusual presentation may make it harder to solve.
- Transforming the task into a dynamic task. Instead of asking just one question, the task might ask a similar question many times, changing the input data between successive queries.
- Asking for a full backtrace or list of steps instead of just a minimum or maximum value. For instance, a task that asks for the length of a shortest path might be changed to ask for the path itself. In some cases this can make a solution significantly harder to implement.
- Reducing the memory limit, which can often bring new challenges to an otherwise straightforward algorithm.

5.3. *Adding or Removing Subproblems*

One way to make a task more difficult and sometimes more original is to add a second problem or subproblem. This can be done in several ways:

- By transforming the input of the task, so that additional work is needed before the data can be used in the main algorithm. For instance, an input graph might be presented as a set of circles in the coordinate plane, where individual circles correspond to the vertices of the graph and intersections between these circles correspond to edges.
- By using the output of the original task as input for another task. For example, suppose the original task asks for the convex hull of a set of points. A new subproblem might then ask for the specific point on that hull from which the other points span the smallest range of angles.
- By embedding a new subtask into each iteration of the original algorithm, or by embedding the original task into each iteration of a new algorithm. For example, suppose the original task asks whether a graph with some given attribute x is connected. A new task might ask for the *smallest* value of x for which the graph is connected. The new algorithm then becomes a binary search on x, where the original connectivity algorithm is used at each step.

Even beyond the extra work involved, adding a subproblem can make a task more difficult by hiding its real purpose, or by hiding the domain of algorithmics that it belongs to. For instance, in the first example above (graphs and circles) the extra subproblem might make a graph theory task look more like geometry.

Conversely, a task can be made easier by removing subproblems where they exist. For instance, the input data could be transformed in a way that requires less implementation work but still preserves the algorithm at the core of the task.

5.4. *Aiming for a Solution*

A significant part of the task creation process involves searching for solutions. This search can sometimes present interesting opportunities for changing a task.

While trying to solve a task, an interesting or unusual algorithm might come to mind that cannot be applied to the task at hand. Such ideas should not be ignored; instead it can be useful to study why an interesting algorithm does not work, and to consider changing the task so that it *does* work.

An example of this technique can be seen in the problem *Belts* (FARIO 2006), which began its life as a simple dynamic programming problem. Whilst trying to solve it the author found a different dynamic programming solution over an unusual set of subproblems, and eventually changed the task so that this unusual algorithm was more efficient than the simple approach.

As an aside, when working on a task it is important to invest serious time in searching for a solution. In particular, one should not give up too easily and simply change the problem if a solution cannot be found. If problem setters only consider tasks for which they can see a solution immediately, they risk only finding tasks that are close to classic or well-known algorithms.

Even once a solution has been found, it is worth investing extra time into searching for a more efficient solution. This proved worthwhile for the problem *Architecture* (FARIO 2007) – in its original form the task required an $\mathrm{O}(n^5)$ solution, but after much additional thought the organisers were able to make this $\mathrm{O}(n^4)$ using a clever implementation. This gave the task more power to distinguish between the good students and the very good students.

6. Algorithmic Task Creation

As explained earlier, the task creation techniques described in this paper are largely post-hoc – they have been developed naturally by problem setters and improved over the years through experience. As an attempt to further understand and refine these techniques, we now revisit them in a more abstract setting. In Subsection 6.1 we introduce the idea of an abstract "problem space", and in the subsequent Subsections 6.2–6.4 we reformulate our task creation techniques as an "algorithmic" search through this problem space.

6.1. *Defining the Problem Space*

We began this paper by outlining our objectives, as seen in Section 2 which described the features of a good task. Essentially these features give us different criteria by which we can "measure" the quality of a task. Although these criteria are imprecise and subjective, they essentially correspond to an *evaluation function* in algorithmics; given a task, they evaluate the quality or interest of that task.

Following this, we outlined the many different characteristics that can define a problem, as seen in Section 3. Although once more imprecise, we can think of these characteristics as different dimensions of a large multi-dimensional *problem space*. Each task corresponds to a point in this space, and the characteristics of the task give the "coordinates" of this point.

With this picture in mind, creating a good task can be seen as a global optimisation problem – we are searching for a point in the problem space for which our evaluation function gives a maximal value (a problem of "maximal interest"). In the following sections we attempt to understand the particular optimisation techniques that we are using.

6.2. *Finding a Starting Point*

In Section 4, we devoted much attention to the many different techniques used by problem setters to find an initial idea for a new task. This essentially corresponds to choosing a starting point in our problem space from which we can begin our search.

It should be noted that different techniques leave different amounts of work up to the problem setter. Most techniques give a reasonably precise task, though sometimes this task comes without a story (for instance, when modifying a known algorithm, or borrowing from mathematics), and sometimes it comes without an interesting solution (such as when drawing on games and puzzles).

On the other hand, some techniques do not provide a specific task so much as a vague idea; this is particularly true of looking around (Subsection 4.1) and building from abstract ideas (Subsection 4.5). Although these techniques do not give a precise starting point in our problem space, they do specify a region in this problem space in which we hope to find interesting tasks.

6.3. *Moving About the Problem Space*

We push our analogy onwards to Section 5 of this paper, where we discussed how problem setters modify their initial ideas until they arrive at a task that they like. This modification process typically involves a succession of small changes to different characteristics of the task.

In our abstract setting, changing a single characteristic of a task corresponds to changing a single coordinate in our problem space. In this way the modification process involves a series of small steps through the problem space to nearby points, until we reach a point whose evaluation function is sufficiently high (i.e., a good task).

As noted earlier, sometimes a problem setter has a vague idea in mind rather than a precise task; in this case a small change may involve pinning down an additional characteristic to make the task more precise. In our abstract setting, this corresponds to reducing the dimension of a region in the problem space, thus making the region smaller (with the aim of eventually reducing it to a single point).

Problems setters typically do not change a task just once, but instead try many different changes, keeping changes that improve the task and discarding changes that do not. Gradually these changes become smaller as the task becomes better, until the problem setter is fully satisfied.

From an algorithmic point of view, this is reminiscent of *simulated annealing*, one of the most well-known optimisation algorithms. Since simulated annealing is an efficient method of optimisation, we can hope that the techniques described in this paper are likewise an efficient way of creating new and interesting tasks.

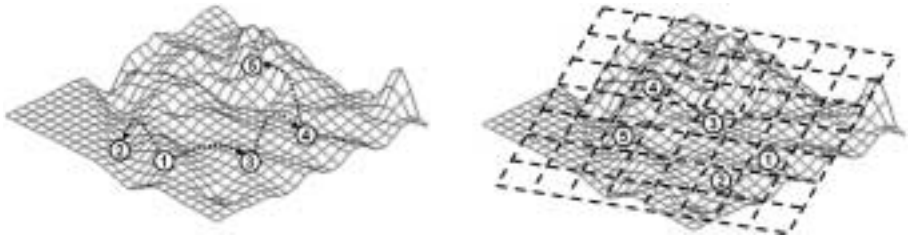


Fig. 9. Depictions of the "simulated annealing" and grid exploration methods

This overall process is illustrated in the left hand diagram of Fig. 9. Here the problem space is drawn as landscape, where the height shows the value of the evaluation function. The arrows show the progression of small changes that convert an initial idea (1) into a final task (5).

It is worth noting that simulated annealing does not guarantee a global maximum, though thankfully that is not the goal of the problem setter. The search space has many local maxima, and problem setters can look through these for suitable tasks that meet their criteria.

6.4. *The Grid Exploration*

The technique described earlier in Subsection 4.4 (filling the holes in a domain) is worth a special mention here. With this technique we make a list of existing tasks in a domain, examine their different characteristics, and then forge a new combination of characteristics that will eventually become our new task.

This technique fits naturally into our abstract setting. The domain of interest can be seen as a subspace of the larger problem space, and our list of existing tasks forms a collection of points in this subspace. By examining the different characteristics of these tasks we effectively list the "significant dimensions" of this subspace.

As a result we essentially impose a multi-dimensional grid over our subspace, where the different dimensions of this grid represent the characteristics that we are focusing on. By listing the characteristics of each task, we effectively place these tasks in individual grid cells. This is illustrated in the right hand diagram of Fig. 9.

To create a new task, we then search through empty grid cells, which correspond to new combinations of characteristics that we have not seen before. To make our tasks as original as possible, we aim for empty cells that have few coordinates in common with our existing tasks.

6.5. *Final Notes*

We have seen in Subsections 6.3 and 6.4 how different task creation techniques correspond to different ways of exploring and optimising the problem space. In particular, we can see how these techniques relate to well-known global optimisation methods such as simulated annealing.

As optimisation is an extensively-studied field of algorithmics, it could be interesting to take other optimisation algorithms and try to apply them to the task creation process. This in turn might help the community to produce new and original tasks into the future.

References

Burton, B.A. (2007). Informatics olympiads: Approaching mathematics through code. *Mathematics Competitions*, **20**(2), 29–51.

Charguéraud, A. and Hiron, H. (2008). Teaching algorithmics for informatics olympiads: The French method. To appear in *Olympiads in Informatics*.

Diks, K., Kubica, M. and Stencel, K. (2007). Polish olympiad in informatics – 14 years of experience. *Olympiads in Informatics*, **1**, 50–56.

Ginat, D. (2002). Gaining algorithmic insight through simplifying constraints. *J. Comput. Sci. Educ.*, April 2002, 41–47.

Kedlaya, K.S., Poonen, B. and Vakil, R. (2002). *The William Lowell Putnam Mathematical Competition 1985–2000: Problems, Solutions, and Commentary*. Mathematical Association of America.

Larson, L.C. (1983). *Problem-Solving Through Problems*. Springer, New York.

Skiena, S.S. (1998). *The Algorithm Design Manual*. Springer, New York.

Skiena, S.S. and Revilla, M.A. (2003). *Programming Challenges*: *The Programming Contest Training Manual*. Springer, New York.

Sloane, N.J.A. (2008). The on-line encyclopedia of integer sequences. Retrieved March 2008 from `http://www.research.att.com/~njas/sequences/`

van Lint, J.H. and Wilson, R.M. (1992). *A Course in Combinatorics*. Cambridge Univ. Press, Cambridge.

Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science*, **4**(1), 193–216.

Wilf, H.S. (1994). *Generatingfunctionology*. 2nd edition. Academic Press, New York.

B.A. Burton is the Australian IOI team leader, and has chaired scientific committees for national contests since the late 1990s. In 2004 he co-founded the French-Australian Regional Informatics Olympiad with M. Hiron, and more recently he chaired the host scientific committee for the inaugural Asia-Pacific Informatics Olympiad. His research interests include computational topology, combinatorics and information security.

M. Hiron is the French IOI team leader, and is the co-founder and president of France-IOI, the organisation in charge of selecting and training the French team for the IOI. He creates tasks on a regular basis for national contests and online training programs, as well as for the French-Australian Regional Informatics Olympiad, and occasionally for other international contests such as the IOI or the APIO. He is a business owner working on projects ranging from web development to image processing and artificial intelligence.

Olympiads in Informatics, 2008, Vol. 2, 37–47

Romanian National Olympiads in Informatics and Training

Emanuela CERCHEZ

Informatics College "Grigore Moisil"
Petre Andrei 9, Iasi, 700495 Romania
e-mail: emanuela.cerchez@gmail.com

Mugurel Ionuţ ANDREICA

Politehnica University
Splaiul Independentei 313, Bucharest, 060032, Romania
e-mail: mugurel_ionut@yahoo.com

Abstract. The paper discusses the Romanian experience, including two on-line forums which have helped improve the participation and quality of the informatics experience in Romania.

Key words: informatics, olympiad, training, problem.

1. Informatics Secondary Education. Background

This section presents a general view of the secondary education in Informatics in Romania.

Secondary education in informatics in Romania started in 1971, when 5 informatics high-schools were founded, one in each of Bucureşti, Iaşi, Cluj, Braşov and Timişoara. Another informatics high-school was founded afterwards in Petroşani. Until 1989, only these 6 schools provided secondary education in informatics.

After 1989, the number of informatics schools or informatics classes in ordinary schools increased rapidly, since students and parents became more and more interested in this discipline.

Over time, changes have frequently occurred, both in informatics education and in the educational system. Every minister of education tried to do a major transformation in the educational system, at different levels.

15 years ago, a student studied informatics for 8 hours/week. Today, in an ordinary mathematics-informatics class, a student studies 1 hour/week in the 9th and 10th grades (representing the age range of 15–17 years), and 4 hours/week in the 11th and 12th grades. There are only a few special (intensive) classes for 4 and 7 hours/week respectively.

This situation has an obvious negative impact on the quality of education in the field.

2. Alternative Training

In Romania, the major problem in informatics education is the lack of teachers and especially the lack of qualified and/or proficient teachers. We have had to find alternative ways to train students.

2.1. *Centre of Excellence for Young People Capable of High Performance*

In 2001, the Centre of Excellence for Young People Capable of High Performance was founded by the order of Minister of Education and Research. Gifted students were declared a national wealth, having specific educational needs.

In the whole country, 9 Regional Centers of Excellency were founded, each coordinating 5–6 counties, each having its own coordinator and developing its own training program on 6 disciplines: mathematics, informatics, chemistry, biology, physics and geography. The main idea was to gather highly-skilled students with the best teachers and to develop special activities as a response to their higher educational needs.

Is this program a success?

After 7 years, the Center of Excellence still lacks national coordination, budget or headquarters. In some counties this program has never worked (the person in charge did not organize the activities of the Center of Excellence: teachers' registration, students' registration, training programs, etc.). In some there are hesitating attempts to make it work (the activities of the Center of Excellence are organized, but not for all the 6 disciplines, training activities are scheduled too late or not every week). But in some counties this program really works, year after year. For instance, every academic year, over 1000 students in Iaşi choose to spend week-ends working at the Center of Excellence. Almost 100 teachers train students for excellence. Passion and enthusiasm make things work.

2.2. *.campion*

`http://campion.edu.ro`

In 2002 we started .campion, an online training program for performance in informatics, supported by teachers with remarkable results and brilliant students, former winners of international informatics olympiads.

The program is supported by SIVECO Romania, a leading software company, in cooperation with the Romanian Ministry of Education and Research.

Nowadays .campion is part of Siveco Virtual Excellence Center, together with Siveco Cup, the National Contest for Educational Software.

The main goal of .campion training program is to offer all students an equal chance to participate in a high level training in computer science. The specific objectives of our training are:

- to develop algorithmic thinking,
- to develop programming skills,
- to develop competitive spirit.

Students are divided into 3 training groups, according to their level/age:

- group S (Small) – beginners in computer science, the first year in programming; age should not exceed 16 years (9th grade or primary school students);
- group M (Medium) – intermediate level, one or two years experience in programming; age should not exceed 17 years (10th grade);
- group L (Large) – advanced in programming (11th and 12th grade).

The training program consists of rounds: alternating training rounds and contest rounds.

For each round, students receive 2 problems, to be solved in 10 days for a training round, and in 3 hours for a contest round.

Solutions are graded using an automatic online grading system. After grading, on the website are published:

- for each student: personal grading sheets;
- for each group: rankings for the current round and also total rankings, including all the rounds;
- for each problem: solutions, solution descriptions, grading tests.

All past problems are organized in an archive (task description, test, solutions).

Each year, the best 50–60 students, participate in an on-site final round, consisting of a single contest day. The main goal of the final round is to offer students the opportunity to know each other, to compete in a real, not virtual environment, to be awarded and acknowledged.

This is .campion from a student's point of view. Between 1000 and 1500 students register every year. About 500 consistently participate. The feedback we collect every year indicates that 30%-75% of the training for the best students (participating to the Final Round) is provided by .campion. All Romanian IOI (International Olympiad in Informatics), CEOI (Central-European Olympiad in Informatics), BOI (Balkan Olympiad in Informatics), and JBOI (Junior Balkan Olympiad in Informatics) medalists are .campion finalists.

In 2005 and 2006 we began international cooperation, between Balkan countries, providing an English version of .campion. The cooperation was excellent with Bulgaria and Moldova, but from a general point of view, international .campion was not considered a success, and we lost support for this initiative.

What does .campion represent from a teacher's point of view?

First of all, of course, it represents a resource for self-conducted students' training or for the teacher-conducted training. But, for us, the National Committee, .campion represents a practical way to train teachers. As we previously emphasized, teachers are the critical resource.

We select teachers from all over the country (according to the results of their students) and propose they cooperation in a .campion training program. Most of them agree and send problems according to some technical specifications. Afterwards, we analyze the problem, the solutions, the grading tests, and the checkers. We suggest improvements, we correct errors, and we constantly communicate by e-mail with the teacher proposing the problem. Definitely, a successful cooperation in .campion leads to a successful

cooperation with the committee preparing the national olympiads in informatics and the national informatics team training.

2.3. *infoarena*

`http://infoarena.ro`

Together we learn better! This statement, written on the Infoarena website, describes in a simple, yet precise way, Infoarena.

Infoarena is a training website made by students for students. On this website, students organize programming contests, publish educational materials, and exchange ideas. A problem archive is available on the website, including Infoarena contests problems, but also problems from different stages of the National Olympiad in Informatics. Online grading is available for Infoarena contests, and also for the archive problems. The Infoarena team has implemented a rating system, reflecting the performances of Infoarena users. Ten to twenty students are supporting the Infoarena website, but a lot of volunteers are helping them, adding to the Infoarena archive problems used in various national and regional contests. Infoarena is a dynamic program, continuously improving. A remarkable educational community, Infoarena yearly attracts students eager to learn and also willing to help others to learn.

3. Romanian Olympiads in Informatics: 30 Years of Experience

This section presents a general view of the Romanian Olympiads in Informatics. Comparing the beginning and the current state of the olympiads might give us a real perception of the changes that happened over 30 years.

3.1. *The First Olympiad in Informatics*

In 1978 the first Olympiad in Informatics was organized. A few facts from the first olympiad are that there were about 60 participants who had to write both a handwritten and computer contest. Their choice of languages for the computer portion was Fortran, Cobol and ASSIRIS.

The general procedure of the contest was: students wrote programs on a special sheet of paper called a “programming form”;

- the programming form was given to the operators, who punched the cards;
- the program written on the punched cards was run on the computer (IBM 360 compatible) twice; after the first run the errors made by the operators were corrected;
- the listing obtained after the second run was handed to the National Committee; the National Committee unfolded the listing (usually in a long hall) and corrected it “by hand”.

Sample task

Consider a sequence of n ($1 \leqslant$ n $\leqslant 100$) positive integers less than or equal to 50. For each distinct integer in the sequence determine the number of appearances in the sequence. Draw an horizontal histogram to represent the number of appearances of each distinct integer, using a corresponding number of *s.

3.2. *Informatics Olympiads after 30 Years*

Nowadays the National Olympiad in Informatics has three stages and two divisions. The first division is for gymnasium students (5th to 8th grade), while the second division is for high-school students (9th to 12th grade).

For each division 3 stages are organized: a local, regional and national stage.

The local stage is organized in each town, using contest problems proposed by local teachers. The best students qualify for the regional stage. Actually, the selection is not very tough, the local stage being a good practice competition. A regional stage is organized in each county, using contest problems created by the National Committee. According to the results of this contest, each county selects a team to participate in the National Olympiad. The number of members in the team is between 3 and 11, and is established according to the results obtained at the National Olympiad in the past years by the teams of the county.

The National Olympiad in Informatics gathers about 300 high-school students and about 160 gymnasium students. The contest is held over two consecutive days and consists of 3 problems for each contest day. After a day break, half of the students may participate in another 2 contests, having the purpose to select the national informatics teams (10 students in the Junior Team, 24 students for the Senior Team).

Sample task for selection of Junior National Informatics Team

Consider two groups, each of them containing n digits ($1 \leqslant n \leqslant 9$). In any group, digits may appear more than once. Using all the digits of the first group we build a positive integer (n1). Similarly, using all the digits of the second group, we build another positive integer (n2).

Determine n1 and n2 so that the difference n1-n2 is greater than or equal to 0 and minimal. In case there is more than one solution, choose the solution with minimal n1.

Sample task for selection of National Informatics Team

Zaharel draws N points on a plane ($N \leqslant 100000$). Being a peculiar person, he also chooses M points on the X-axis ($M \leqslant 200000$). All the coordinates of the points are less than 10^9. Then, he asks himself for each of the M points which of the N points is closest (situated at minimum distance). The distance between two points (x_1, y_1) and (x_2, y_2) is considered $(x_1-x_2)^2+(y_1-y_2)^2$.

Determine for each of the M points the closest of the N points.

3.3. *Training Camps*

Two training camps are organized for the national teams. Training consists of theoretical courses and also problem solving sessions. During the training camps students have

selections contests (3 for each camp), in order to select the teams for IOI, BOI, CEOI and JBOI.

4. Task Definition and Development

The task definition and development process is a crucial part in the organization of any contest, whether regional, national or international, because the contest results (and winners) are established based on the sum of scores obtained for each task in the contest task set. Since the purpose of a contest is to rank the competitors accurately according to their abilities, several aspects are considered when developing tasks for a particular contest (like the difficulty level of the tasks, the age group of the competitors, the contest duration and others). These aspects are considered from two perspectives, during two stages of the task definition and development process. During the first stage, each member of the contest committee develops one or more tasks individually and then he/she submits them to the other members, as candidate tasks for the contest task set. During the second stage, several tasks are chosen from the set of candidate tasks and these tasks will form the contest task set. We will describe the structure of the contest committee, the contest and task types and the syllabus for each contest type. Afterwards, we will discuss aspects of both stages of the task definition and development process. Finally, we will present some grading related aspects.

4.1. *The Contest Committee*

The members of the contest committee belong to one of the following categories:

- remarkable high-school professors, teaching informatics both in classes and within Centers of Excellence;
- university professors and teaching assistants, working in the domain of mathematics and computer science;
- former medalists of international olympiads in informatics and former members of the Romanian national informatics team, who are currently bachelor, master or PhD students.

The responsibility of each member of the contest committee is to create at least one contest task (candidate task) and to participate in the process of selecting the candidate tasks which will form the contest task set.

4.2. *Syllabus, Contest Types and Age Groups*

In Romania there are local, regional and national olympiads, for seven age groups:

- gymnasium pupils – four age groups (5th, 6th, 7th and 8th grade);
- high-school pupils – three age groups (9th grade, 10th grade, 11th–12th grades).

The pupils belonging to the seven age groups vary significantly in terms of their algorithmic and programming skills and knowledge. For this reason, in order for the contest

results to be meaningful, the difficulty levels of the tasks must be suitable for the type of contest (local, regional, or national) and the age group of the competitors. For each age group, there is a loosely defined syllabus. The syllabus for the 11th–12th grades is the most inclusive and includes of the following general topics: greedy algorithms, divide and conquer techniques, dynamic programming, graph algorithms, data structures, combinatorics, computational geometry, string algorithms and game theory. For the other age groups, some of these topics are excluded from the syllabus. For instance, graph algorithms are not included in the 10th grade syllabus and dynamic programming algorithms are not included in the 9th grade syllabus. Furthermore, each topic contains basic and advanced techniques. The advanced techniques can be considered only for the larger age groups (11th–12th grades). Officially, the syllabus is defined loosely, i.e. the topics and their contents are broadly defined. Unofficially, however, the syllabus is quite detailed; these details are filled by the experience, knowledge and common sense of the contest committee members. We should mention that the detailed syllabus for the informatics olympiads is far more advanced than the topics the pupils are taught at school. Thus, a lot of emphasis is put on the individual supplementary training of each competitor (alone and under the guidance of a teacher) and on the participation in the alternative training programs (.campion, Infoarena and others).

The local and regional olympiads are qualifying contests for the national olympiad in informatics, which is the most important informatics contest in Romania. This fact is acknowledged, among other things, by granting to the prize winners free admission to universities with a computer science department. Winning a prize in the national olympiad in informatics is challenging, as there are only three prizes awarded per age group (or 5, in case the total number of competitors is large enough). Based on the results at the national olympiad and another selection contest, a small number of competitors are chosen to be part of the national informatics team. The pupils who are part of the national informatics team take part in several subsequent selection contests, from which participants at the international informatics competitions are selected. The seniors are selected for participating in IOI, CEOI, BOI and Tuymaada International Olympiad. The juniors are selected for participating in the Junior Balkan Olympiad in Informatics (JBOI). It is worth mentioning that the JBOI and the junior informatics team are quite new (the JBOI started in 2007 and the junior informatics team was established in 2008) and that they represent an important means for stimulating the interest of secondary school pupils in informatics. The syllabus for the seniors is a superset of the one for the 11th–12th grades' national olympiad, including more advanced concepts for each topic. In general, it is well aligned with the one proposed in [1], but also contains several more advanced techniques such as hashing, max flow algorithms, bipartite matching, interval trees, and AVL trees. The curricula for the junior level is only loosely defined and is now under active development.

The Informatics contests are characterized by several parameters. Two of them have already been mentioned: the geographic scope (local, regional, or national) and the age group. The number of rounds, duration per round and number of tasks in a contest round are some other important factors. The local and regional contests consist of only one round with two tasks and their duration is usually 3 hours. The national olympiad consists

of two rounds, held in two consecutive days. Each round lasts for about 4-5 hours and consists of three tasks. The contests selecting the junior and senior national informatics teams have the same format as the national olympiad (two rounds, 3 tasks and 4-5 hours per round). The pupils participating in the IOI, CEOI and BOI are selected based on 6 other contest rounds, having the same format. Overall, an IOI candidate has to solve 30 tasks, starting with the national olympiad and finishing with the last selection contest. Every task is awarded the same number of points (100).

Other types of contests are online contests, with variable duration, different number of tasks per round and different grading styles (IOI style or ACM ICPC style), like those hosted by .campion [2] and infoarena [3].

4.3. *Task Types*

Just like in the IOI, the contest tasks are of three types: batch, reactive and output-only. The batch tasks are the most common and they are the ones the competitors are most accustomed to. Batch tasks are also the easiest to prepare and test. Reactive tasks require special support from the evaluation system. In Romania, an evaluation system capable of supporting batch, reactive and output-only tasks is used. Output-only tasks are generally not appreciated by the competitors and they have rarely made the pupils' skills and knowledge stand out. For this reason, only one or two of them are used in the IOI selection contests. All the tasks are algorithmic in nature and must be implemented in a programming language, like C/C++ or Pascal.

4.4. *Creating a Contest Task*

During the first stage of the task development process, the authors compose and prepare their tasks. The first thing to be considered is the syllabus, which depends on the contest type (local, regional, national, or international) and age group (school grade). Based on the syllabus and the task author's experience and preferences, a task idea is developed. The most important part of the task at this point are: **the algorithm** required for solving it (the reference solution), an estimation of **the difficulty level** of the task and an estimation of the **duration** a competitor who is above average would require for finding and implementing the algorithm. The difficulty level is expressed as the percentage of the total number of competitors who should have the necessary knowledge for developing the required algorithmic techniques and who should be able to find them and implement them in a programming language in a reasonable time (typically ranging from a quarter to half of the contest duration).

After the task idea and solution are defined, alternative solutions are searched for, both equally efficient and less efficient than the reference solution. Occasionally, the reference solution may not be the most efficient one for the task, particularly in the case of contests for younger age groups or when the most efficient solution is either very difficult to find or too complicated to implement. After finding all (or most) of the alternative solutions, the author decides the number of points each solution should be awarded, based on its

efficiency in terms of time and memory. The next steps consist of developing a clear problem statement, implementing all the solutions and creating several test cases. Each test case is assigned a number of points (the sum of the points of all the test cases should be 100). The most common situation is represented by 10 or 20 equally weighted test cases. At this point, time and memory limits are set, based on the behavior of the reference solution. The test cases and their associated points are chosen in such a way that each solution receives the desired number of points. The author may also choose to award partial scores for some of the test cases. The duration estimation can also be updated, based on the unforeseen problems he/she encountered while implementing the reference solution.

At any moment during this stage, the author may collaborate with other members of the contest committee. The collaboration usually consists of writing alternative solutions, improving the task statement and, rarely, developing test cases.

This stage of creating tasks may start several months, although usually less than a month, before the contest date and finish several days (one or more) before the contest. The stage usually consists of several iterations, in which the problem statement, algorithm implementations and the test cases are gradually developed and improved. Small parts of the problem statement are changed particularly often, in order to help the competitors understand the task requirements clearly, as well as to avoid potential typing mistakes.

4.5. *Selecting the Contest Task Set*

We consider that the set of contest tasks must have all of the following properties:

- it must contain tasks of different levels of difficulty (from easy to difficult);
- the range of algorithmic topics covered by the tasks must be broad (i.e. multiple tasks should not be solvable by similar techniques);
- all the tasks should be solvable within the allotted contest time, by above average competitors – this does not mean that we expect this situation to actually occur, because there are many other factors involved;
- the tasks should make a good distinction between highly skilled, average skilled and poorly skilled competitors.

A few days (or, rarely, even weeks) before the contest date, after the tasks have been developed (at least partially) and submitted as candidate tasks, the process of selecting the contest tasks begins. The tasks are classified based on their level of difficulty, the types of required algorithmic techniques (greedy, dynamic programming and others) and the estimated time for solving them. A balanced set of radically different tasks is selected, which has all the properties mentioned before. The tasks are chosen democratically, based on the votes of the contest committee members. Each member votes the tasks he/she thinks should be part of the contest task set. The period between the classification of tasks and the casting of votes is reserved for discussions among the contest committee members. By expressing their opinions before the vote, all perspective on the tasks can be shared. It often happens that issues which were otherwise disregarded come up during this discussion phase. The discussion phase usually helps the opinions of the committee

members to converge to a large degree. This way, after the votes are cast, it is often the case that more than half of the chosen tasks are voted in favour by more than 75% of the members. Another issue which is taken into consideration is the number of rounds of the contest. In this situation, votes are cast for selecting problems for each round. However, the task set is final only for the next round to come with changes being allowed for the future contest rounds.

4.6. *Grading the Tasks*

The tasks are graded automatically using an evaluation system which supports batch, reactive and output-only tasks. For the batch tasks, a grading program and the test cases are all that is required. The grading program checks the solution and outputs the score for each test case to standard output, using a well defined format, in order to properly interact with the rest of the evaluation system. When the solution is unique, and no partial score is awarded for a test case, a default grading program may be used. Setting up reactive tasks requires a number of test cases and two programs: the grading program and a program which interacts with the contestant's solution. Once the grading process is started, it can be stopped and resumed at any time. The evaluation system generates individual score sheets for each contestant.

5. Results

Romania participates in the International Olympiad in Informatics, Balkan Olympiad in Informatics, Central-European Olympiad in Informatics (the last two competitions being initiated by Romania), Tuymaada International Olympiad and in Junior Balkan Olympiad in Informatics (a new competition, initiated in 2007 by Serbia).

Romania has been participating at the IOI since 1990, winning a total of 67 medals (19 Gold, 32 Silver and 16 Bronze [4]).

References

[1] Verhoeff, T., Horvath, G., Diks, K. and Cormack, G. (2006). A Proposal for an IOI Syllabus. *Teaching Mathematics and Computer Science*, **IV**(I).
[2] .campion. `http://campion.edu.ro`
[3] infoarena. `http://infoarena.ro`
[4] `http://www.liis.ro/~marinel/Statistica.htm`

E. Cerchez is the scientific coordinator of the Romanian National Informatics Committee. She is currently teaching in Informatics College "Grigore Moisil" in Iasi. She has been involved inthe Romanian National Olympiads since 1996 and in IOI since 2001. In 2002 she initiated .campion training program and she is coordinating this program ever since. She published over 15 books for secondary education in informatics. Since 2004 she has been involved in creating educational software.

M.-I. Andreica is a teaching assistant at the Polytechnic University of Bucharest (PUB), in Romania. He is currently pursuing his PhD degree in computer science, on the topic of communication in distributed systems. His research has been recognized as valuable by being awarded a prestigious IBM PhD fellowship. Mugurel has been a member of the Scientific Committee of the National Olympiad in Informatics since 2002 and has also contributed each year to the selection process of the Romanian IOI team. Furthermore, since 2005, he has been the coach of the student teams competing for PUB in the ACM ICPC regional contests. Other training activities include his participation in the .campion national project as a contest task author and his active involvement in the organization of preparation contests on several training web sites. As former IOI and ACM ICPC World Finals medalist, Mugurel has a significant amount of experience regarding programming contests and he makes use of this experience within his teaching and training activities, as well as in order to compose meaningful and interesting contest tasks. It is also worth mentioning that his experience as a former participant in informatics contests and olympiads has also had a great beneficial impact upon his research activity.

Olympiads in Informatics, 2008, Vol. 2, 48–63

Teaching Algorithmics for Informatics Olympiads: The French Method

Arthur CHARGUÉRAUD, Mathias HIRON

France-IOI
5, Villa Deloder, 75013 Paris, France
e-mail: arthur.chargueraud@gmail.com, mathias.hiron@gmail.com

Abstract. This paper describes a training curriculum which combines discovery learning and instructional teaching, to develop both problem solving skills and knowledge of classic algorithms. It consists of multiple series of exercises specifically designed to teach students to solve problems on their own or with the help of automated hints and intermediate problems. Each exercise is followed by a detailed solution similar to a lecture, and synthesis documents are presented at the end of every series. The paper then presents a structured problem solving method that is taught to the students throughout this curriculum, and that students can apply to organize their thoughts and find algorithms.

Key words: teaching algorithmics, problem solving techniques, guided discovery learning.

1. Introduction

France-IOI is the organization in charge of selecting and training the French team to the International Olympiads in Informatics (IOI). As main coaches of the French team since its early days in 1997, our main focus is to develop teaching materials and improve training methods. Since there are no computer science curriculums in French high schools, the students who come to us are usually self-taught programmers with almost no experience in the field of algorithmics.

All the teaching materials that have been produced are made available to the public on the training website (`http://www.france-ioi.org`), where the students can study all year long. Intensive live training sessions are held several times a year for the top students. The long-term objective is to reach a maximal number of students, so the online training follows the same principles as the live training sessions. The aim of this paper is to describe these principles. Section 2 gives an overview of the training curriculum. Section 3 then presents the structure of the training website. Finally, Section 4 describes the problem solving method that has been developed along the years and is now taught throughout the website.

2. Philosophy of the Training Curriculum

In this section, we explain how the training curriculum is designed around series of problems for the teaching of classic algorithms and data structures as opposed to providing lectures. We then present the benefits of this approach.

2.1. *Introduction to the Structure of the Training*

The training curriculum we present differs from those that can be found in many algorithmic courses. Most curriculums follow a common pattern:

1. *Provide basic knowledge*. Teach algorithms and data-structure through lectures, providing pseudo-code and complexity analysis.
2. *Show examples*. Illustrate how to apply this knowledge to practical situations.
3. *Give exercises*. Solidify what was taught, and try to develop solving skills.

This approach counts mostly on a student's innate abilities when faced with new, more difficult problems. Our thesis is that *improving skills to solve new, difficult problems is what really matters in algorithmics*. Therefore the true aim of a teaching curriculum should be to improve those skills. Providing the student with knowledge and experience is necessary, but should not be the first priority.

The interactive training is structured around sets of exercises with each set focusing on one particular area of the field. Within a set, each exercise is a step towards the discovery of a new algorithm or data-structure. To solve an exercise the students submit their source-code and the server automatically evaluates the code, in a fashion similar to grading systems that are used in many contests. If the students fail to solve an exercise, they may ask for automated hints or new intermediate problems, which in turn ease the progression. Once the students succeed, they get access to a detailed solution that describes the expected algorithm and presents a reasonable path to its discovery. At the end of each sequence of exercises, a synthesis of what has been learned is provided.

Overall the curriculum can be seen as a guided step-by-step discovery completed with instructional material as opposed to more standard curriculums which are typically based on instructional material and very little discovery learning.

2.2. *Teaching the Problem Solving Method*

While training students along the years, it was tried as much as possible to give reusable hints, i.e., hints valuable not only to the current problem but for others as well. Reusable hints were then collected and sorted in an organized document. In recent years, the authors have improved this document by carefully analyzing the way they come up with ideas for solving difficult problems. As a result, a *general problem solving method* was developed. This method is not a magical recipe that can be applied to solve any algorithmic problem, but someone who is sufficiently trained to apply this method will be able to solve more difficult problems than he/she could have solved without it.

As the document describing the method appears very abstract at first, asking one to *"apply the method"* is not effective. Instead, the teaching of the method is spread throughout the training curriculum in three ways: [1]

1. The hints given to students when they need help are (in most cases) hints that could have been obtained by applying the method.
2. Each detailed solution is presented in a way that shows how applying steps of the method leads to the expected algorithm.
3. Advanced students are provided not only with the reference document describing the problem solving method, but also with versions specialized to particular areas.

To teach this method, we follow an inductive approach rather that a deductive one, since it is extremely hard to explain without many examples to rely upon.

2.3. *Advantages over Traditional Curriculums*

To improve skills for solving unknown hard problems, it is necessary to practice solving such hard problems. So in a way, explaining an algorithm in a lecture can be seen as wasting an opportunity to train that skill. It gives away the solution to a problem before giving the students a chance to come up with a few good ideas on their own. By finding some, if not all of the ideas behind an algorithm by themselves, students not only learn that algorithm but also improve their problem solving skills. Also, by making these ideas their own, they are more likely to adapt them to future problems.

Once students have spent time thinking about a problem and have identified its difficulties, they are in a much better position to understand and appreciate the interest of the reference solution that is then presented to them. More importantly, students can compare the process they have followed to find the solution against the process that is described in the lecture. They can then update their strategies for the next exercises.

Also, discovering by oneself the principles of breadth-first search, binary trees, and dynamic programming generates a great deal of self-satisfaction. Even though the previous problems and reference solutions made it much easier by putting the students in the best conditions, they get a sense of achievement for having come up with their own ideas, gain confidence about their own skills and a strong motivation to go further.

Discovery learning is often proposed as an alternative to the more traditional instructional teaching, but has been strongly criticised recently (Kirschner *et al.*, 2006). Our purpose is not to advocate the *pure* discovery learning that these papers focus on and criticize extensively. The curriculum we present is indeed based on *guided* discovery learning, and is completed with extensive and carefully designed instructional teaching. With a structure designed to ensure that each step is not too hard for students, and which reinforces what is learned after each exercise, the risks associated with pure discovery learning, such as confusion, loss of confidence and other counter-productive effects are avoided.

[1] Note: the training material is constantly evolving, and what is presented here corresponds to the authors' current vision. At the time of writing, many existing elements of the website still need to be updated to fit this description completely.

To summarize, we teach problem solving techniques by relying in the first place on the good aspects of discovery learning, and then consolidating the insight acquired by students through instructional teaching. Teaching of pure knowledge, which is a secondary goal, also follows the same process. This is the complete opposite of standard curriculums which start by teaching knowledge in instructional style, and then leave students to develop solving techniques mainly on their own.

3. Structure of the Training Curriculum

In this section, we give the global structure of the training curriculum, and describe each of its components in more detail.

3.1. *Overview of the Structure*

The training website aims at being an entirely self-contained course in algorithmics. The curriculum is divided into three levels of difficulty: beginner, intermediate, and advanced. The only requirement for the "beginner" level is to have some experience of programming in one of the languages supported by the server. At the other end, the "advanced" section contains techniques and problems corresponding to the level of the IOI. Note that the training website also includes instructional programming courses and exercises for the C and OCaml languages.

Each of the three levels contains several series of exercises of the following kinds:

1. *Step-by-step discovery.* A sequence of problems designed to learn about a set of classical algorithms on one particular topic. Each problem comes with a detailed solution, as well as optional automated hints and intermediate exercises.
2. *Application problems.* A non-ordered set of problems of various topics involving variations on algorithms and techniques previously learned through step-by-step discovery.
3. *Rehearsal exercises*. Students are asked to write their best implementation of each standard algorithm studied in a given domain. Reference implementations are then provided.
4. *Practice contests*. Students are trained to deal with limited time and get them used to zero-tolerance on bugs.

When students complete a series of exercises, they are granted access to a synthesis document that recapitulates all the encountered concepts in a well-structured presentation. Once students complete an entire level, they are provided with various elements of the problem solving method and a document summarizing techniques that have been implicitly presented throughout the reference solutions.

Little by little, these documents constitute a sort of reference manual that a student can look back to on a regular basis.

3.2. *Hints and Intermediate Problems*

To improve students' problem solving skills, they are trained on problems that are hard enough to make them think seriously and apply problem solving strategies, but feasible enough to have a good chance at finding a solution within a reasonable amount of time without the risk of losing motivation. Clearly there is no hope that a single set of problems will suit all students and their disparate levels.

The issue is addressed in the following way: a set of "main problems" that are non-trivial even for the best students is given. For each problem, intermediate problems and/or hints are given to students who cannot find the solution after a certain amount of time. They are carefully selected to not only help the students reach the solution, but to also make them realize they could have thought of the solution by themselves with the right strategy, that they will then apply to solve the following problems.

From a practical point of view, a hint consists of information that is added at the end of a task and may be remarks on properties of the task, a simplified formulation of the problem, a well chosen graphical representation of an example, or a suggestion about the kind of algorithm to look into. The last hints provided give the main ideas of the solution and make sure students do not get stuck and give up on the task.

Sometimes intermediate problems are provided instead of hints. They are typically simplified versions of the original problem or versions with weaker constraints for which the students can submit a program and obtain a detailed solution. This gives them a strong basis to attack the original problem.

One can compare each series of exercises to a climbing wall that leads to the discovery of the algorithms in a particular area: when a hold is too high to be taken the students are helped out by being provided with one or more extra intermediary holds. If this is not enough, the student has the option to contact a coach. Overall the structure adapts itself to the level of the students by providing steps of difficulty corresponding to their levels, which help them to learn as much as possible.

3.3. *Example: a Series of Problems Introducing Graph Algorithms*

In this section, we illustrate the notion of step-by-step discovery with an example series of problems that introduces students to very basic graph algorithms. This series appears roughly in the middle of the beginner level of the training curriculum. Its only prerequisites are the completion of a set of problems that covers basic data structures (introducing stacks and queues), and another that introduces recursive programming.

The tasks from this first graph series all take a maze as input: a two-dimensional grid in which each cell is either an empty square or a wall. The locations of the entrance and exit are fixed (respectively at the left-top and bottom-right corners). The following list describes these tasks and the list of hints and/or intermediate problems that the website provides on demand.

1. Read a maze and print the number of empty squares that have respectively 0, 1, 2, 3, and 4 empty adjacent squares. The purpose of this problem is to introduce the

notions of neighbors and degree, and to show how to manipulate these notions in the code in an elegant way.

   a. Hint (Intermediate problem): "read a maze and print the number of empty squares". The aim of this new problem is to check that input data is read correctly.

2. Read a 10×10 maze and print the number of different ways to go from the entrance to the exit without walking through the same cell twice within a single path.
   a. Hint: the following question is asked: "what question can you ask on each of the neighbors of the first cell in order to compute the total number?"
   b. Hint: the answer to the previous hint is given, and insists on the fact that there is a set of cells one cannot use in the rest of the path.
   c. Hint: the main idea of a recursive function is given, that maintains the current set of cells that cannot be used for the rest of the path.
3. On a maze of up to 1000×1000 cells, give the total number of cells that can be reached from the entrance.
   a. Intermediate problem: same exercise with a 10×10 maze. This problem comes with a hint of its own, telling how to reuse the idea from the previous problem by marking every visited cell. It also comes with a detailed solution of a working exponential algorithm.
   b. Hint: the following suggestion is provided: "try to apply the algorithm given in the previous hint by hand on an example and find out how to reduce the amount of work involved".
   c. Hint: running of the algorithm is demonstrated with an animation that clearly shows sequences of steps done several times.
4. Given a 10×10 maze, find the longest path going from the entrance to the exit without traversing the same square twice. Print the path as a sequence of letters ('W', 'N', 'E', 'S'). When there is more than one such path print the one that is first in alphabetical order.
   a. Intermediate problem: any longest path is accepted as an output.
      i. Intermediate problem: only output the length of the path
      ii. Hint: the following question is asked: "in the solution provided for the previous intermediate problem, when can you say that the square corresponding to the current step of the recursive exploration is part of the longest path found so far?"
      iii. Hint: the answer to the previous question is provided, and demonstrates a way to record the steps of the longest path.
   b. Hint: the following question is asked: "in the solution given for the first intermediate problem, in which order should you try to explore the neighbors?"

This structure allows students to manipulate algorithms such as exhaustive search, depth first search, and printing the path corresponding to an exhaustive search. Strong students can do this quickly by solving 4 problems while weaker students can go at a slower pace with a total of 8 problems.

3.4. *Detailed Solutions*

Access to the detailed solution of each problem is only given once the problem has been solved by the student and checked by the server. The students are encouraged to read these analysis documents carefully. Indeed, having solved the problem successfully does not necessarily mean one has understood everything about the algorithm he/she discovered.

This document contains complete instructional material on the algorithm studied that consists of the following elements:

1. A step by step description of a thought process that leads to the key ideas of the solution. The intent is to have the students realize that they could have applied this process entirely on their own.
2. A well chosen graphical representation of the problem. Such diagrams help the students to see the interesting properties of the problem and can be reused later when working on similar problems.
3. A clear presentation of the algorithm. First through a few sentences giving the big picture and then through precise pseudo-code. The rationale of why the solution works is given, but no formal proof.
4. A complexity analysis of the solution.
5. Implementations of the reference solution, in C++ and OCaml that are as elegant and simple as possible.

These elements are given not only for the expected solution of the problem, but also for valid alternative solutions. Solutions usually start with a description of the principles and complexity analysis of algorithms that are correct but not sufficiently efficient, since they are typically intermediate steps in the thought process. Counter-examples to frequently proposed incorrect algorithms are sometimes presented to explain why those algorithms cannot work. This is done in a way that teaches the students how to create their own counter-examples.

This combination of elements gives the students a solid basis of knowledge and techniques on which they can then rely to solve other problems.

3.5. *Application Problems*

When working on problems from the step-by-step discovery series, students are in an environment which is particularly prone to bringing about new ideas. For instance, in the middle of the graph algorithms series, the students expect each problem to be a new graph problem and moreover expect its solution to be partially based on the elements discovered in the previous exercise. While such series of exercises are great to learn about graph algorithms, they do not train recognition of graph problems among a set of random tasks.

So once the basic knowledge of each field is acquired it is important to train students to solve tasks outside of an environment that suggests a given type of solution. Therefore each level ends with sets of problems of various kinds that cover most of the elements encountered throughout the discovery series. These tasks train the students in three ways.

First, they train to recognize the nature of a problem without any obvious clues. Second, they train to apply the knowledge and solving techniques acquired to variations of standard algorithms. And third, more difficult problems would typically require a combination of algorithms coming from different fields.

These application problems also come with hints and detailed analysis that insist on the different steps needed to find the solution. Also, these solutions often describe some useful programming techniques which help making the code shorter and cleaner, thus less error-prone.

4. Introduction to the Problem Solving Method

This section gives an introduction to the problem solving method that we have developed along the years and now teach throughout our website. We do not attempt to describe the whole method in details since this would be way beyond the scope of this paper, but instead we try to convey its main principles. To that end, we first explain how this method has been obtained, then illustrate its working on three particular steps, and finally give an overview of the other steps that it involves.

4.1. *Origins of the Method*

Throughout the years spent training the students on a regular basis through intensive live sessions, sets of problems on the website or frequent email interactions, there have been numerous occasions to look for the best advice to help students solve a given problem without giving them the solution itself, or even part of the solution. On each of these occasions, it could be determined which advice were the most successful.

It became apparent that some types of advice were very efficient over a variety of problems. Little by little a collection of techniques was synthesized which led to a full method for solving algorithmic problems.

It was then observed that on various occasions including IOI competition rounds, students applying the new method in a systematic manner would find the right ideas for difficult tasks more often than students who only counted on their intuition. Since then, improving this method and the way it is taught throughout the training program have been the top priorities. Every time coaches or contestants solve a hard problem, time is spent analyzing what helped to get the right idea. This is then taken into account to update the method when appropriate.

4.2. *Dimensions of the Problem*

This section describes a process that not only is a key to the application of several solving techniques, but which also helps to get a clear understanding of a task.

The objective is to build an exhaustive list of the dimensions of the problem. The word “dimension” is to be taken in its mathematical sense; informally it corresponds to everything in the problem that can take multiple values (or could if it was not settled to a specific value). Dimensions should be ordered by type: they can be the values from

the input data, from the output data, or intermediate values that are implicitly necessary to be manipulated in order to get the output. Beginners are given a technique to ensure no dimension has been missed (due to lack of space, this is not described here). Trained students do this step as they are reading the problem statement.

For each dimension in that list, the range of possible values should be indicated. The constraints for the input dimensions are usually given in the task. For other dimensions, some calculations (or approximations) might be needed.

To illustrate the process, consider the following problem:

> "*You are given the coordinates* (x_1, y_1) *and* (x_2, y_2) *(*$0 \leqslant x_1, x_2, y_1, y_2 \leqslant 100\,000$*) of two diagonally opposite corners of each of* N *(*$0 \leqslant N \leqslant 10\,000$*) rectangles. Write a program that computes the maximum number of rectangles that have the same area.*"

Table 1 describes the dimensions for this problem. Notice that there is more to consider than just two dimensions (x, y) of the plan since x_1 and x_2 as well as y_1 and y_2, can be changed independently and have different meanings. Note the potential overflow for the surface of rectangles which means 64 bits integers will be needed at some point in the implementation.

Filling the second column may not always be obvious, and one has to be careful not to blindly write down what is given in the problem statement. In particular, the range of values for a dimension can be significantly reduced when it is impossible to reach all the values from the range given in the task.

Having this table of clearly listed dimensions at the top of one's paper is very useful both while looking for ideas and during the implementation phase. The next two sections will show how this list can be used as a first step to some very effective techniques.

Table 1
Dimensions of a problem

Dimension	Range of values
Input dimensions	
id of a rectangle	[0..9999]
x_1	[0..100,000]
y_1	[0..100,000]
x_2	[0..100,000]
y_2	[0..100,000]
Output dimensions	
Number of rectangles of a given surface	[1..10,000]
Implicit dimensions	
Width of a rectangle	[0..100,000]
Height of a rectangle	[0..100,000]
Surface of a rectangle	[0..10^{10}] (overflow!)

4.3. *The Power of Simplification*

The most recurring advice given to the students when they are stuck on a hard problem can be synthesized in the following way:

> "*Simplify the problem, and try to solve each simplified version as if it was a new, independent problem.*"

This simple advice is undoubtedly the most powerful problem solving technique in the method. It is effective in that its sole application suffices to resolve many hard problems and in that there is a very precise recipe to generate simplified versions of a given problem.

The idea that simplifying a task may help to find the solution of a problem is not new. It is described for instance by Ginat (2002), and mentioned in the analysis of various tasks such as "Mobiles" from IOI 2001 (Nummenmaa *et al.*, 2001). What is presented here is a technique to look for every useful simplification of any task, that students are asked to apply systematically.

There can be many different ways to simplify a given problem and not all of them give as much insight into the complete problem. Moreover, some useful simplifications may not come to mind immediately. So what is needed is a way to come up with the most useful simplifications in a quick manner. The following recipe can be used to go through every simplified version of a problem.

1. For each dimension of the task (see Subsection 4.2) try to simplify the problem by either: (a) removing the dimension, (b) setting the value for this dimension to a constant, or (c) reducing the range of possible values for that dimension.
2. Among the resulting simplifications, rule out those which clearly lead to a non-interesting problem. Then, for the sake of efficiency, sort the remaining simplified problems according to their interest – this is to be guessed by experience.
3. For each simplification considered, restate the corresponding problem as clearly as possible and try to solve it as if it were a completely independent problem. This may involve applying the problem solving method recursively, including a further simplification attempt.
4. Try to combine the solutions of the different problems in order to get some ideas for the complete problem. (There are some specific techniques to help with this step).

Notice that there is no need to simplify more than one dimension at a time since the simplification recipe is called recursively when necessary.

The fundamental idea behind this technique is that although solving a simplified version of the problem is almost always easier than solving the whole problem, it is often very helpful. Indeed, the complete solution needs to work at least on instances of the simplified problem, so any algorithm or clever observation required in the simplified solution will most likely be a part of the complete solution.

Finding the solution to a simplified version has an effect akin to getting a very big hint. Given that the simplification technique is so useful to produce hints and that it is so easy to apply (at least with some experience) it is tried to have the students apply this technique as a reflex.

Going from the solution(s) of one or more simplified versions of a problem to a solution for that problem can be difficult, and a separate document provides with techniques that make it easier. The content of that document is out of the scope of this paper.

As an example, consider the task “A Strip of Land” (France-IOI, 1999) where, given a 2-dimensional array of altitudes, the goal is to find the rectangle of maximal surface, such that the difference between the lowest and highest altitude within that rectangle is stays below a given value.

Examples of simplified versions of this task that are obtained by applying the described method are:

1. Remove the y dimension; the task is then, given a sequence of integers, to find the largest interval of positions such that the difference between the minimal and the maximal value from that interval is less than a given bound.
2. Reduce the altitude dimension to only 2 values, 0 and 1; the task is then to find the largest rectangle containing only zeros.
3. Apply both simplifications; the task is then to find the maximal number of consecutive 0s in a sequence of 0s and 1s.

Each of these simplified versions appears much easier to solve than the original task, and a solution to the original problem can be obtained by combining ideas involved in the solutions of these simplified problems.

4.4. *Graphical Representations: Changing Points of View*

Drawing different examples on a piece of paper and looking at them can be a very good way to get ideas. Drawing helps to show many properties clearly and all the elements that are laid on the paper are elements that do not need to be “maintained” in the limited short term memory. That memory can then be used to manipulate ideas or imagine other elements moving on top of the drawing. For a given problem, however, some drawings are much more helpful than others.

When students are working on hard problems they typically write some notes about the problem on a piece of paper and draw a few instances of the problem as well as the corresponding answers. Then they stare at their paper for a while thinking hard about how to solve the problem. When a student do not seem to be moving forward on a problem, coaches look at how he drew his examples and often think “no wonder he can’t find the solution, with such a drawing no one could”.

The students’ drawings represent examples that are too small for anything interesting to appear and they are asked to try with larger ones. Other times the problem lies with their choice of a graphic representation for the problem. Students tend to draw things in a given way and often stick to that representation throughout their whole thinking process. They sometimes make a similar drawing multiple times hoping that new ideas will come. Their mistake is to forget that there can be several different ways to draw the same example and the first one that comes up is seldom the one that does the best job at bringing ideas.

To illustrate the point, consider the following task:

> "*Given a sequence of positive integers, find an interval of positions within that sequence to maximize the product of the size of the interval and the value of the smallest integer within that interval of positions.*"

Faced with this task most students will naturally write a few sequences of integers on their paper and compute the product for various possible segments.

However, there is a way to represent such sequences that is much more expressive: draw each number of the sequence as a vertical bar whose height corresponds to the value of that number. Segments that are potential answers for a given instance of that problem can then be represented as rectangles whose height is the minimum height among all bars along its length. The answer then corresponds to the rectangle with the largest area. This new representation displays some of the properties of the problem in a much clearer way, and makes it easier to solve the task (Fig. 1).

The most important advice that is given regarding graphical representations is the following:

> "*Don't stick to your first idea of a graphical representation and try different ways to draw examples*".

Applying this advice in an efficient way is not as easy as it seems. Students may quickly think about several representations, but often miss the most interesting ones. Students are taught to apply a simple technique to enumerate the most interesting graphical representations and select the best ones.

The main idea behind that technique is to observe the following fact: there are only two dimensions on a piece of paper. So only two dimensions of the problem can be presented in a very clear ordered way, where values can be compared in an instant, by looking at their relative positions. Selecting the two dimensions of the problem that will be mapped to the x and y axis of the sheet of paper is an essential part in selecting the right representation. The following steps summarize this process:

1. Among all the dimensions of the task (see Subsection 4.2) identify the most important ones starting with the dimensions that correspond to values that need to be compared easily. Consider grouping the dimensions that are compatible as one (x_1 and x_2 may both be represented on the same axis). This first step is used to optimize the chances of finding the best representation quickly.
2. For every possible pair among these dimensions consider a graphical representation that maps each dimension of the pair to the x and y axis of the paper and find a reasonable way to represent the others in the resulting grid.

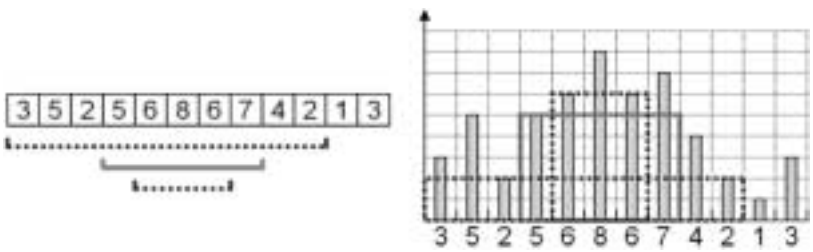


Fig. 1. Two graphic representations for the same problem.

3. Among these possible representations, apply each of them on a simple example. By comparing the results, it is usually clear which representation is the most helpful to visualize the problem and bring ideas.

In most cases, there are only a few pairs of dimensions to enumerate so this technique can help to quickly find the best representations. When the problem is more complex and the number of dimensions is higher, there can be quite a few pairs to enumerate. This may seem like a long process, but with some experience and a couple of rules of thumb to discard the least interesting representations, drawing an example for each potentially interesting pair can be done fairly quickly.

Of course, selecting these two dimensions is not enough to get a good representation and much advice can be given depending on the type of problem. In most cases though, what should be represented on the paper to get ideas is mostly the same: the components of the example, and the answer for that example. Students often do fine naturally at selecting which elements to draw, but may need some advice on what example to draw, and how to draw it.

Once the students have a nice and carefully drawn graphic representation of a good example in front of them it is observed that they are much more likely to get the right idea. We often observe students spend a lot of time working on a problem only to see the solution "appear" to them as soon as they are asked to use a graphic representation that can be obtained by the aforementioned technique. All that was needed for them to find the solution was to change their point of view on the problem. Forcing oneself to try out different graphical representations is a very effective way to try new points of view.

4.5. *General Structure*

So far, three particular elements of the problem solving method have been presented. In this section, we give an overview of its structure by briefly describing each of the main steps.

The method can be divided in two parts. The purpose of the first part is to help to bring a better understanding of the problem and to bring solution ideas. The second part contains steps to apply on each idea that comes up during the first part. Unless the task is really easy and the solution is obvious, students should at least go through the first four steps of Part 1 to make sure they have a clear understanding of the problem before working on a given idea. After that, they may jump to the second part at any time and come back later to Part 1 if necessary.

The following steps should be attempted in the order they are listed. It might however be necessary to come back to a previous step at some point to spend more time on it. For example, it is often useful to come back to Step 4 and generate some new examples.

1. Restate the problem (or sub-problem) in the simplest way, discarding superficial elements of the story, to get a clear idea of its essence. A good formulation usually involves two sentences: the first one describes the data and the second one the question that must be answered on this data. Someone who has not seen the full task should be able to start looking for a solution after reading this description only.

2. Write down a list of all the dimensions involved in the task and the corresponding constraints, as described in Subsection 4.2. This further helps to get a clear idea of the task and is necessary both to find a good graphical representation (Step 3) and to simplify the problem (Step 6).
3. Find the best graphical representations for the task by applying the technique described in Subsection 4.4. This can make working on examples (Step 4) much more effective at bringing ideas.
4. Generate different kinds of examples and solve them carefully and entirely by hand. These examples should be large enough so that the answer is not obvious and needs some effort to be determined. This step has many benefits. First, it helps to get all the details of the problem clearly in mind. Also since the brain is naturally lazy, ways to avoid doing too much work will often come automatically, which can lead to good ideas. These examples will be used again later to test new ideas and check the final implementation, so writing them rigorously is rarely a waste of time.
5. Look for an algorithm that gives a correct answer to the problem, without worrying about efficiency, and describe it clearly. The purpose is to separate the concern of correctness from the concern of efficiency and this often helps to clarify the recursion involved in the task. In some cases, writing it as pseudo-code may be a good idea. Depending on the form of this "naive" solution, different specific methods can then be applied to transform it into an efficient solution. Note that it is sometimes useful to implement such a brute-force solution, to generate data from some examples that one can then analyze to look for specific properties.
6. Simplify the problem and try to solve each simplified version as if it were a new, independent problem, as explained in Subsection 4.3. Then try to combine the different ideas into a solution for the original problem.
7. Try to see the task from different point of views by listing standard algorithms and wondering how they could be applied. One may ask questions like "can the problem be seen as a graph problem?", "could it be solved with a shortest path algorithm?", or "how could a sweep-line technique be applied?" and so on. Even in the case where the solution is not a standard algorithm, changing point of view on the task in this way may help to bring new, original ideas.

For each promising idea obtained during this first part the students are asked to go through the following steps. Note that Step 5 should be applied on any "reasonable" idea that is found to be incorrect.

1. Describe the solution in a couple of sentences. The objective is to make it clear enough that anyone who knows the task and is experienced with algorithmics can understand the idea. Depending on the type of the algorithm, the method provides standard ways to describe the solution.
2. Find a good graphical representation of the algorithm. In a similar fashion to what we described in Subsection 4.4, there is often one or more good ways to represent the dynamics of an algorithm graphically. This can help to understand why it works and brings attention to special cases or possible optimizations. Again, the method provides standard representations for certain classes of algorithms.

3. Using this graphical representation, try to execute the algorithm by hand on a couple of examples. This gives a better feeling of how it works and may bring up elements to think about during the implementation.
4. Look for counter-examples to this algorithm, as if it were a competitor's solution that you want to prove wrong (i.e., forget that you really hope it works). If a counter-example can be found, then it often offers an excellent example to use when looking for better ideas. It is also useful to explicitly state why the idea does not work.
5. After spending some time looking for counter-examples without finding any, it usually becomes clear why there is no chance of finding a counter-example. In other words, it becomes clear why the algorithm is correct. While it is too hard and too long to carry out a formal demonstration of correctness, stating the main invariants that make the algorithm work reduces the risk of implementing an incorrect algorithm.
6. Determine the time and memory complexities as well as the corresponding running time and actual memory needed. At this point you may decide if it is worth going ahead with this algorithm or better to keep looking for more efficient and/or simpler solutions.
7. Write the pseudo-code of the algorithm, try to simplify it and test it before going ahead with the actual implementation.

On many occasions during this process, students may encounter sub-problems that need to be solved. For example it could be a simplified version of the original problem or something that needs to be done before starting the main algorithm. When students encounter such sub-problems, they tend to work on them with less care and apply less efficient strategies than when they work on the original task. It is important that faced with such a sub-problem, they work on it as if it were the original problem and apply the method (recursively) on it, starting with Step 1.

The structure of the curriculum aims at teaching the students how to apply all of these steps. The hints and intermediate problems often correspond to applying steps of the first part. The detailed solutions try and follow the method closely. Documents synthesizing algorithms and data-structures providing domain-specific techniques are designed to help out during this process, particularly during Step 7 of Part 1. Finally, each step of the method is described in great detail in an independent document.

5. Conclusion

We described a complete curriculum for teaching algorithmics, organized around the aim of teaching problem solving skills. Unlike most traditional curriculums which follow an instructional approach, this curriculum combines the benefits of both guided discovery learning and instructional learning, using the first to introduce new notions and relying on the second to consolidate the knowledge acquired. Thanks to a system of hints and intermediate problems, the discovery learning component is effective for students of different levels.

Throughout the structure, we teach the application of the problem solving method that we have developed along the years. This is done by following an inductive approach: the application of the method is illustrated through the hints, intermediate problems and solutions to the tasks, and then generalized into synthesizing documents.

This training website has introduced algorithmics to hundreds of students over the years, many of whom developed a strong interest in the field. Motivated students often solve more than 200 of the problems in our series within one or two years, going from a beginner's level with only some experience in programming to a level that allows some of them to get medals at the IOI, ranging from bronze to gold.

This success shows that such a curriculum can be a very effective way to teach algorithmics and more specifically, problem solving skills. In the future, we aim at improving this curriculum by adding more series of problems to cover a wider range of domains, by optimizing the structure it is based on, and by perfecting the method that it teaches.

References

Charguéraud, A. and Hiron, M. *Méthode de résolution d'un sujet.*
`http://www.france-ioi.org/train/algo/cours/cours.php?cours=methode_sujet`

Ginat, D. (2002). *Gaining Algorithmic Insight through Simplifying*. JCSE Online.

France-IOI. *France-IOI Website and Training Pages.*
`http://www.france-ioi.org`

Hiron, M. *Méthode de recherche d'un algorithme.*
`http://www.france-ioi.org/train/algo/cours/cours.php?cours=methode_recherche_algo`

Kirschner, P.A., Sweller, J. and Clark, R.E. (2006). Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, **41**(2), 75–86.

Nummenmaa, J., Mäkinen, E. and Aho, I. (Eds.) (2001). *IOI' 01 Competition.*

A. Charguéraud is the vice-president of France-IOI. After his participation at IOI'02, he got involved in training the French team to the IOI. He has designed many tasks and tutorials for the training website, as well as tasks for contests. He is a PhD student, working at INRIA on formal verification of software.

M. Hiron is a co-founder and president of France-IOI. He has selected and trained French teams for International Olympiads since 1997, and is the co-author of many elements of the training website. As a business owner, he works on projects ranging from web development, to image processing and artificial intelligence.

Olympiads in Informatics, 2008, Vol. 2, 64–74

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 lifecycle. 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.

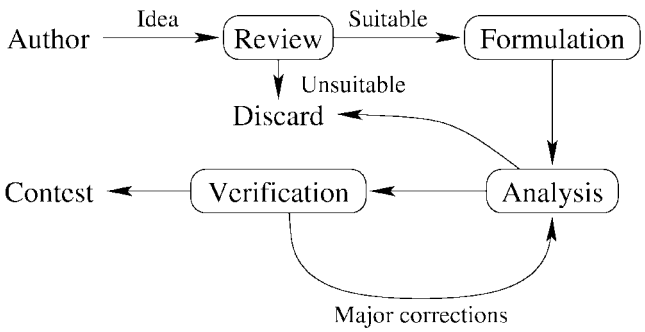


Fig. 1. Task life-cycle.

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.

Olympiads in Informatics, 2008, Vol. 2, 75–89

Tasks and Training the Youngest Beginners for Informatics Competitions

Emil KELEVEDJIEV
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences
Akad. G. Bonchev str., block 8, 1113 Sofia, Bulgaria
e-mail: keleved@math.bas.bg

Zornitsa DZHENKOVA
Mathematical High School
2 Elin Pelin str., 5300 Gabrovo, Bulgaria
e-mail: zornica.dzhenkova@gmail.com

Abstract. Training children for participation in informatics competitions and eventually in the IOI has been moving to younger ages and now is starting in many countries at a level of about 5–6th grades (about 11–12 years old). The main tools for teaching and preparation are tasks. We present the experience and problems given in the Bulgarian national competitions in informatics for school students in the mentioned age group. Some features of the Bulgarian system for the preparation of the youngest school students are discussed. The study covers a period from 2001 up to present. In the paper, an attempt is made to arrange and classify tasks by keywords. As examples, selected task descriptions and comments are given.

Key words: tasks in competitive informatics, informatics for the youngest school students.

1. Introduction

In recent years, the competitions in informatics have been continually expanding and involving more and more younger students. This process can be observed in Bulgaria, as well in many other countries in the world. An example for this development is the establishment in 2007 at Belgrad, Serbia, of a new kind regional Balkan Youth Olympiad in Informatics for the students up to 15.5 years old. In Bulgaria after 2001, several age group systems have been applied to divide school students for the national informatics competitions (the Autumn, Winter, and Spring Tournaments, as well for the three rounds of the National Olympiad in Informatics).

In 2001, we had an age group of 5–7th school grades (11–13 years old), which we denoted at that time as a “youth age group”. Starting in 2002, groups were introduced with letter names: A, B, C, and D, which comprised 11–12, 9–10, 7–8, and 4–6th school grades, respectively (In Bulgarian schools the mentioned grades correspond to 18–19, 16–17, 14–15 and 11–13 years old students, respectively). Starting in 2004, an additional group for the youngest students was introduced, group E, comprising the 4–5th grades.

This modified the age division among groups A, B, C, and D, as 12, 11–10, 9–8 and 6–7th grades. Later, our observations showed that it would be better to change slightly this division principles and starting in the autumn of 2007, we have groups A, B, C, D, and E, that cover 11–12, 9–10, 7–8, 6, and 4–5th school grades, respectively.

A permanently open question, often asked by teachers and trainers, who are involved in the preparation of students from the youngest age group, is the question: how to choose suitable tasks? The goal is to cover such material that might be expected in real competitions. Of course, the style of the olympiads does not always allow good prediction about the task types even for the youngest students. Nevertheless, it is possible to outline some set of themes and task types, which can serve as preparation tools. One important starting point to do this selection is examining the tasks, given at the previous real competitions. Classifying them, it becomes possible to make up manuals and handbooks. In Bulgaria, recently published books (Kelevedjiev and Dzhenkova, 2004) and (Yovcheva and Ivanova, 2006) are successfully used in the preparation process for the mentioned age group including school students of about 4–6th, or even up to 7th grade.

2. Classification

After having accumulated enough tasks (Bulgarian web portal site for competitions in Informatics, 2008) previously given in competitions, it becomes possible to start an attempt for classification using keywords.

The chosen keywords indicate some basic features from 3 different points of view:

a) basic concepts of the programming language (mainly concerning C/C++ language) together with the simple data types: numbers, symbols, strings, text (as a set of strings and delimiters), one- and two-dimensional arrays, arrays of strings, and some special attention is emphasized on the sequences of input data elements;

b) basic control constructions that form a program: simple computation by a chosen formula, conditional operator (“if” operator), loop with a counter (“for” cycle), loop with a condition (“while” cycle), combination of a loop and an “if” operator, embedded loops, recursion, and reasonable use of procedures in programming (functions in C/C++ language);

c) algorithms (with respect to the involved subject): whole numbers and divisibility, digits of a number, long numbers, combinatory analysis, sorting, recursion, geometry (rectangular shapes with sides which are parallel to the coordinate axis), modeling (including date and time intervals, informative processing of texts, etc).

The choice and the amount of the keywords are not strictly determined in our next presentations. We rather assume keywords as abbreviations to point out what is the main essence of the task.

3. Exemplary Tasks

The following tasks are chosen to illustrate the use of keywords. They also present several main topics and trends in the competitive informatics for the youngest age group in Bul-

garian national competitions. At some tasks, simple input and output examples are given in order to clarify what the used keywords mean (especially for the task that require the output of a figure or digits:

3.1. *Keyword: Conditional Operator*

Task "Brick" (4–6th grades, Spring Tournament, 2002). A brick has a form of a regular parallelepiped with length x, width y, and height z. These sizes are expressed as whole numbers, less than 1000. Write a program, that inputs x, y, and z, and outputs a number, which is equal to the value of a minimal area that should be cut in sheet iron, so that the brick can be moved through the hole. While moving we assume that brick's sides remain parallel to the edges of the hole.

3.2. *Keyword: Embedded Cycles*

Task "Different ways" (4–6th grades, Winter Competition, 2002). Write a program that inputs a positive integer S, $5 \leqslant S \leqslant 50$, and outputs how many ways there are for the integer S to be presented as a sum of 3 different integers. Example input: 10, output 4. Explanation: $10 = 1 + 2 + 7 = 1 + 3 + 6 = 1 + 4 + 5 = 2 + 3 + 5$.

3.3. *Keyword: Printing out a Figure of Characters*

Task "Decreasing numbers" (4–6th grades, Round 1 of the National Olympiad, 2004). Write a program that inputs number N, $1 \leqslant N \leqslant 9$, and outputs the following figure: on the first row – all whole numbers from 1 trough N; on the second row – all whole numbers from 2 trough N; and in a similar way up to the Nth row, where should be placed the number N only.

Example input: 5
Output:
12345
2345
345
45
1

3.4. *Keyword: Dates and Hours*

Task "Airplane" (6–7th grades, Round 1 of the National Olympiad, 2007). An airplane departs at K hours and M minutes, and arrives at L hours and N minutes. Write a program that finds out how many hours and minutes the airplane has been flying, and which time (that of the departure or of the arrival) is earlier in the twenty-four-hour day period. The flight lasts less then 24 hours. Departure and arrival times are assumed to be in a same time zone. Program's input consists of four integers K, M, L, N, on a line, separated by spaces ($0 \leqslant K \leqslant 23$, $0 \leqslant M \leqslant 59$, $0 \leqslant L \leqslant 23$, $0 \leqslant N \leqslant 59$). The output has to

contain two lines. On the first line, two integers for the flight duration have to be written and they have to express hours and minutes. On the second line, one of the letters: D or A, has to be written, depending on what is earlier: departure or arrival.

3.5. *Keyword: Strings*

Task “Leftmost” (5–6th grades, Spring Tournament, 2001). Given is a string of length N, $50 \leqslant N \leqslant 255$, containing small and capital Latin letters and digits. Some characters may occur repeatedly. Write a program that inputs the string and determines which pair of equal characters is leftmost placed. That is, the found pair should have the following property: there are no identical characters placed before the first (rightmost) character of the found pair. The output should contain two integers in the range from 1 trough N, namely the positions of both found characters in the pair.

3.6. *Keyword: Texts*

Task “Words” (4–6th grades, Autumn Tournament, 2003). Write a program that inputs text of length up to 80 characters. We call a “word” a sequence of consecutive characters which does not contain spaces, and the word has to be separated by spaces from the other words. Your program has to output the same text as input but with the places of the longest and the shortest words exchanged. In case there is more than one longest and/or shortest word, the program has to exchange the last longest word with the first shortest one. If all the words have the same length, the program has to output the same text as input.

3.7. *Keyword: Modeling and Generating*

Task “One or Zero” (4–5th grades, Spring Tournament, 2006). Let us consider numbers 1, 10, 100, 1000, 10000, and so on. That is, we consider numbers, each of them starting with 1, followed by zeros. Now take number 1 and join 10 to its right-hand side, then join 100 again to the obtained new right-hand side, then join 1000, and so on, doing this many times. We can obtain a very long number: 110100100010000100000... Write a program that inputs integer N, $0 < N < 65000$, and outputs the Nth digit of the above defined long number.

3.8. *Keyword: Recursion*

Task “Sticks” (6–7th grades, Spring Tournament, 2006). We have a large enough quantity of two types of sticks – one with a length of 1 m, and the other, with a length of 2 m. The sticks of both types cannot be distinguished, except by length. Taking several sticks, we arrange them tightly in a line with a total length of N meters. In how many ways we may do this? Write a program that inputs N, $0 < N < 30$, and outputs the answer.

3.9. *Keyword: Geometry*

Task “Rectangles” (4–5th grades, Round 3 of the National Olympiad, 2006). Given are two rectangles with sizes a by b, and c by d respectively. We have to put both rectangles side by side, without overlapping, so that the obtained figure has the least possible perimeter. Write a program that inputs the values of a, b, c, and d, as whole numbers, less than 1000, and outputs the least perimeter. Example input: 5, 7, 6, 3. Output: 30.

3.10. *Keyword: Sorting*

Task “Arranging by the sum of digits” (4–6th grades, Winter Competitions, 2003). Given is an integer N, $1 < N < 20$, and a sequence of N different positive integers, whose values are less than 1000. Write a program that inputs this data and outputs the sequence with the given integers, arranged in an increasing order by the sum of their digits. If there are two integers with the same sums of the digits, the smallest integer should be placed first (to left-hand side of the biggest one). Each two subsequent integers should be separated by a space in the output.

3.11. *Keyword: Counting*

Task “Sum” (6–7th grades, Round 3 of the National Olympiad, 2006). Given are N, $1 < N < 20$, different positive integers $a_1, a_2, \ldots, a_N$, with values less than 1000. Consider all sums, in which each given integer occurs at most once. Write a program that outputs how many different values of the considered sums are possible. The program has to read by the standard input the value of N, followed by $a_1, a_2, \ldots, a_N$, all integers separated by spaces. The program has to output the result as an integer on the standard output.

3.12. *Keyword: Table*

Task “Table” (6–7th grades, Spring Tournament, 2007). Given is a table with m rows and n columns ($1 < m < 100$, $1 < n < 100$) with cells containing “0” or “1”. The cell in the upper left corner contains 1. We call two cells neighbors, if one of them is placed directly above, bellow, to the left, or to the right of the other. We call that a set of cells is contiguous, if we can start at any cell of this set and go to any other cell moving only through neighbor cells. Let us denote by S the largest contiguous set of cells containing only “1”, which includes the upper-left cell. In how many ways we can translate the set S within the table boundaries, so that each cell of S covers again a cell that contains “1”? Write a program that outputs this quantity. The program has to read by the standard input values of m and n, separated by a space and followed by m lines in the input, each containing n characters “0” or “1” without delimiters among them.

Table 1
Data types

Keyword	Number of Tasks
Numbers	62
String	27
One-dimensional array	22
Sequence	13
Characters	10
Text	9
Two-dimensional array	8
Array of strings	3
Stack	2

Table 2
Control constructions

Keyword	Number of Tasks
Loop	73
Embedded loops	35
Loop and conditional operator	18
Conditional operator	17
Function	12
Input and output files	3
Computation by formula	1

4. Study of Keywords

In (Kelevedjiev and Dzhenkova, 2008) we published a table with detailed description built on keywords for each task from the complete collection with 148 tasks, which were given at the National competitions in informatics for the age groups of 4–7th grades in Bulgaria during the period 2001–2007. The reader may refer to the English translated copy of the table in the Appendix 2. We give the cumulative data (Tables 1–3).

5. Trends

We present diagrams to illustrate observed tendencies for monotonic or periodic trends in time appearance of task types (by means of several chosen keywords) during the period 2001–2007 in the scene of the Bulgarian national competitions in informatics for the age groups of 4–7th grades (Figs. 1–6).

Table 3

Algorithms

Keyword	Number of Tasks	Keyword	Number of Tasks
Sequential processing	17	Combinatorial analysis	2
Digits from a number	16	Dynamic programming	2
Print out a figure of characters	12	Games and strategies	2
Counting	11	Geometry	2
Divisibility	10	Number systems	2
Text processing	10	Palindrome	2
Optimal elements	9	Rectangular figures	2
Logical	7	Recursion	2
Dates	6	Decomposing numbers	1
Long numbers	6	Exhaustive search	1
Sorting	4	Fractional numbers	1
Modeling	3	Parity	1
String of digits	3	Raising to a power	1

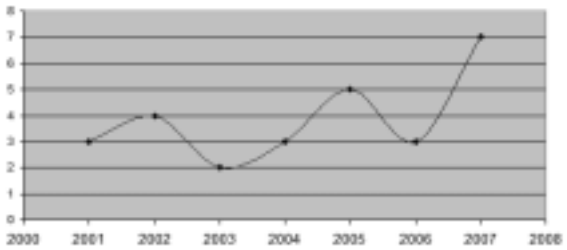


Fig. 1. Keyword: *String*.

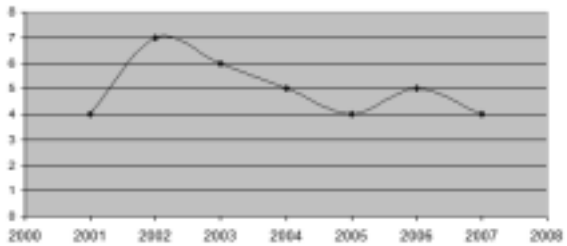


Fig. 2. Keyword: *Embedded loops*.

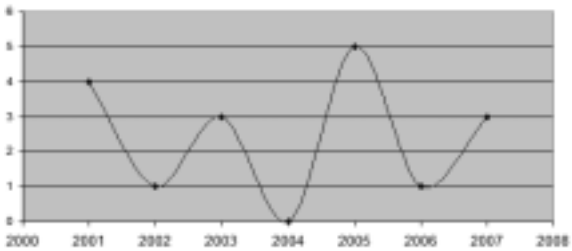


Fig. 3. Keyword: *Sequential processing*.

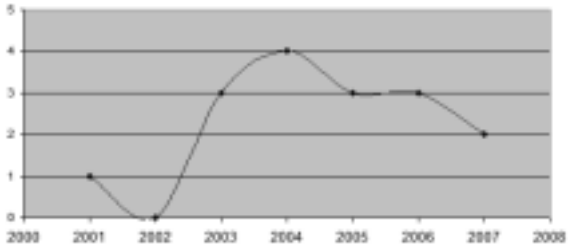


Fig. 4. Keyword: *Digits from a number*.

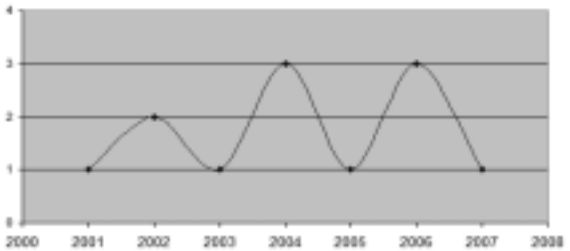


Fig. 5. Keyword: *Print out a figure of characters*.

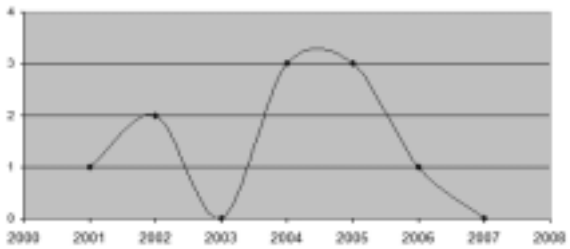


Fig. 6. Keyword: *Divisibility*.

6. Conclusions

Although the presented data as above graph samples are not statistically significant, they give us some ideas about the variety of themes.

Assigning keywords to each task is influenced by personal feelings, tastes, or opinions, but there are some more or less steady principles to choose these keywords. In many cases the keywords are self-descriptive and publishing information about tasks together with keywords is easily understandable and can help teachers in their training education process for competitive problem solving.

The authors of tasks for the Bulgarian competitions could find useful information about the history of tasks from the previous competitions in order not to duplicate or sometimes intentionally repeat some kinds of problems. In more broad sense, the study of the keywords might be applied for initializing appropriate changes and improvements in

the national curriculum which is used now as a recommendable list of themes in all the set of local out-of-class forms for young student preparation in Bulgaria. In the Appendix 1 the reader may find parts of this curriculum ((Bulgarian web portal site for competitions in Informatics, 2008; Bulgarian site for school competitions in Informatics, 2008)).

Appendix 1

Curriculum used about 2004–2005 school years:

Group E

Programming: Environment for C/C++, Branch and loop operators, Integers and Characters. One-dimensional array. Standard input and output.

Algorithms: Whole numbers arithmetic. Dates.

Geometry: Straight line coordinates.

Group D

Programming: Extended study of the programming language. Inroduction to pointers.

Data structures: Arrays and Strings. Multi-dimensional arrays. Stacks and Queues.

Methods for algorithms desing: Simple exhaustive search. Recursion. Introduction to dynamic programming. Binary search in a sorted array.

Aritmetic: Divisibility. Euclid‘s algorithm. Long integers. Number systems.

Sequences: Searching, sorting, Merging, Polynomials.

Combinatorics: Counting, Generating combinatorial configurations.

Graphs: Representations, Grid of squares.

Geometry: Coordinates in the plane. Rectangles with sides parallel to the axes.

Games: Strategies, Parity, Symmetry.

Curriculum for the National out-of-class school for preparation in informatics competitions during the 2007–2008 school years:

6th grade

	Themes	Study hours
1	Functions in C language.	2
2	One-dimensional array	2
3	Sorting	4
4	Strings	4
5	Divisibility. Prime numbers	4
6	Euclid‘s algorithm. Common Fractions	4
7	Strings in C++ style.	4
8	Two-dimensional arrays	6
9	Rectangles with sides parallel to the axes	4
10	Structures in C language	4
11	Recursion.	2
12	Number systems	6
13	Long integers	6
14	Backtracking	7
15	Grid of squares	6
	Total	65

7th grade

	Themes	Study hours
1	Parameters of the functions in C	3
2	Introduction to the standard library	2
3	Sorting – fast algorithms	2
4	Searching – binary search	2
5	Introduction to complexity of algorithms	2
6	Introduction to object-oriented programming	2
7	Combinatorial configurations	2
8	Extended Euclid‘s algorithms	3
9	Roman numerals	2
10	Polynomials	4
11	Pointers in C	2
12	Stack and Queue	3
13	Linked Lists	2
14	Searching substrings in strings	3
15	Games with numbers – using symmetry and parity	4
16	Rectangles	3
17	Bitwise operations	2
18	Long integers	3
19	Backtracking	4
20	Introduction to Dynamic Programming	5
21	Introduction to Graphs	5
	Total	60

Appendix 2

Table 4 presents all tasks given at the Bulgarian competitions during the years 2001–2007. In the column “Competition”, the names of the Autumn, Winter and Spring Competitions are abbreviated, and the three rounds of the National Olympiads in Informatics are denoted by NOI–1, NOI-2, and NOI-3, respectively.

Table 4

Tasks given at the Bulgarian competitions during the years 2001–2007

	Year	**Competition**	**Age Group**	**Task name**	**Keywords**
1	2001	Autumn	D	Stars	Characters, Embedded loops, Print out a figure of characters
2	2001	Autumn	D	Equal	Sequence, Loop and conditional operator, Sequential processing
3	2001	Autumn	D	Numbers	Numbers, Embedded loops, Digits from a number
4	2001	Winter	D	Competition	One-dimensional array, Loop, Sorting
5	2001	Winter	D	Study Circle	One-dimensional array, Loop and conditional operator

To be continued

Table 4

Tasks given at the Bulgarian competitions during the years 2001–2007 (continued)

	Year	Competition	Age Group	Task name	Keywords
6	2001	Winter	D	Text	Text, Loop, Input and output files, Text processing
7	2001	NOI-2	D	Rectangle	Numbers, Input and output files, Divisibility
8	2001	NOI-2	D	Numbers	String, Input and output files, Long numbers
9	2001	Spring	D	String	String, Embedded loops
10	2001	Spring	D	Leftmost	String, Embedded loops
11	2002	Autumn	D	Unique	String, Embedded loops, Sequential processing
12	2002	Autumn	D	Ruler	Numbers, Loop, Divisibility
13	2002	Autumn	D	Triangles	Characters, Embedded loops, Print out a figure of characters
14	2002	Winter	D	Date	Numbers, Loop, Function, Dates
15	2002	Winter	D	Largest	Numbers, Text, Embedded loops, Function, Long numbers, Combinatorial analysis
16	2002	Winter	D	Different ways	Numbers, Embedded loops, Decomposing numbers
17	2002	NOI-1	D	Longest word	Text, Loop, Function, Text processing
18	2002	NOI-1	D	Prime factors	Numbers, Embedded loops, Divisibility
19	2002	NOI-1	D	Exchanges	One-dimensional array, Loop, Combinatorial analysis
20	2002	NOI-2	D	Crossword	String, Embedded loops
21	2002	NOI-2	D	Multiplication	String, Loop, Long numbers
22	2002	NOI-2	D	Different	Array of strings, Embedded loops, Text processing
23	2002	Spring	D	Find	Numbers, Loop and conditional operator
24	2002	Spring	D	Sum	String, Loop and conditional operator, Long numbers
25	2002	Spring	D	Brick	Numbers, Logical
26	2003	Autumn	D	Words	Text, Loop and conditional operator, Text processing
27	2003	Autumn	D	Knight	Two-dimensional array, Embedded loops, Sequential processing, Geometry
28	2003	Autumn	D	Car park	Numbers, Embedded loops, Digits from a number
29	2003	Winter	D	Histogram	String, Embedded loops, Print out a figure of characters
30	2003	Winter	D	Arranged	One-dimensional array, Embedded loops, Digits from a number, Sorting
31	2003	Winter	D	Hotel	One-dimensional array, Embedded loops, Modeling
32	2003	NOI-1	D	Odd numbers	Sequence, Loop and conditional operator, Parity, Sequential processing
33	2003	NOI-2	D	Spiral	Numbers, Loop
34	2003	NOI-2	D	Trade	Numbers, Loop and conditional operator
35	2003	NOI-2	D	Cake	Numbers Embedded loops
36	2003	Spring	D	Minimax	Sequence, Loop and conditional operator, Optimal elements

To be continued

Table 4

Tasks given at the Bulgarian competitions during the years 2001–2007 (continued)

	Year	Competition	Age Group	Task name	Keywords
37	2003	Spring	D	Sum	String, Loop and conditional operator, Long numbers
38	2003	Spring	D	Street	One-dimensional array, Loop, Sequential processing
39	2004	Autumn	D	Painter	Characters, Embedded loops, Print out a figure of characters
40	2004	Autumn	D	Safe	Numbers, Loop, Digits from a number
41	2004	Autumn	E	Inequality	Numbers
42	2004	Autumn	E	Windows	Numbers
43	2004	Autumn	E	Safe	Numbers, Digits from a number
44	2004	Winter	D	Words	Text, Embedded loops, Function, Text processing
45	2004	Winter	D	Smart	Numbers, Function, Recursion
46	2004	Winter	D	Multiplication	Numbers, Loop, Divisibility
47	2004	NOI-1	D	Divisibility	Numbers, Loop, Digits from a number
48	2004	NOI-1	D	Half	String, Loop, Sequential processing
49	2004	NOI-1	D	Decreasing	Numbers, Embedded loops, Print out a figure of characters
50	2004	NOI-2	D	Game	Numbers, Loop, Divisibility
51	2004	NOI-2	D	Rooks	Two-dimensional array, Embedded loops
52	2004	NOI-2	D	Football	Numbers, Loop
53	2004	Spring	D	Fractions	Loop, Divisibility
54	2004	Spring	D	Triangles	Characters, Embedded loops, Print out a figure of characters
55	2004	Spring	D	King Artur	One-dimensional array, Loop, Digits from a number
56	2005	Autumn	D	Words	Text, Loop, Text processing
57	2005	Autumn	D	Calendar	Numbers, Embedded loops, Dates, Print out a figure of characters
58	2005	Autumn	D	Millionaire	Numbers, Loop, Dynamic programming
59	2005	Autumn	E	Bonbons	Numbers, Logical
60	2005	Autumn	E	Guess a digit	String, Loop, Function, Digits from a number
61	2005	Autumn	E	Numbers	Numbers, One-dimensional array, Loop, Digits from a number
62	2005	Winter	D	Game	String, Loop
63	2005	Winter	D	Crossword	Two-dimensional array, Embedded loops, Function
64	2005	Winter	D	Travel	Stack, Loop
65	2005	Winter	E	Windows	Numbers, Logical
66	2005	Winter	E	Minimax	Sequence, Loop and conditional operator, Optimal elements
67	2005	Winter	E	Reciprocal	Numbers, Loop, Digits from a number
68	2005	NOI-1	D	Code	Numbers, Loop, Number systems
69	2005	NOI-1	D	height	Numbers, Loop
70	2005	NOI-1	D	Triangular	One-dimensional array, Loop

To be continued

Table 4

Tasks given at the Bulgarian competitions during the years 2001–2007 (continued)

	Year	Competition	Age Group	Task name	Keywords
71	2005	NOI-1	E	Competition	Numbers
72	2005	NOI-1	E	Estimations	Sequence, Loop, Sequential processing
73	2005	NOI-1		Clock	Numbers, Divisibility
74	2005	NOI-2	D	Platforms	Two-dimensional array, Embedded loops
75	2005	NOI-2	D	Rectangle	One-dimensional array, Loop, Geometry
76	2005	NOI-2	D	Lotto	One-dimensional array, Loop
77	2005	NOI-2	E	Coating	Numbers, Divisibility
78	2005	NOI-2	E	Bus lines	Sequence, Loop, Digits from a number
79	2005	NOI-2	E	Auto	Sequence, Loop and conditional operator
80	2005	NOI-3	D	Arithmetic	Numbers, Characters, Loop
81	2005	NOI-3	D	Intervals	String, Loop
82	2005	NOI-3	D	Crossword	Array of strings, Embedded loops
83	2005	Spring	D	Game	Games and strategies, Divisibility
84	2005	Spring	D	Calendar	Numbers, Loop, Dates
85	2005	Spring	D	Monopoly	Numbers, Loop
86	2005	Spring	...	Calendar	Numbers, Dates
87	2005	Spring	E	Divisors	One-dimensional array, Loop, Divisibility
88	2005	Spring	E	Trip	Sequence, Loop and conditional operator
89	2006	Autumn	D	Library	Numbers, Loop
90	2006	Autumn	D	Trains	Numbers, Embedded loops, Print out a figure of characters
91	2006	Autumn	D	Will	Text, Loop, Text processing, Long numbers
92	2006	Autumn	E	Dates	Numbers, Dates
93	2006	Autumn	E	Text	Characters
94	2006	Autumn	E	Golden Rush	Numbers
95	2006	Winter	D	Joda	Text, Loop, Text processing
96	2006	Winter	D	Curtain	Numbers, Loop, Divisibility
97	2006	Winter	D	MAX3	One-dimensional array, Loop
98	2006	Winter	E	Animal problem	Numbers, Loop, Counting
99	2006	Winter	E	Sets	One-dimensional array, Number systems
100	2006	Winter	E	Snowflake	Characters, Embedded loops, Print out a figure of characters
101	2006	NOI-1	D	Chicken decoder	String, Loop
102	2006	NOI-1	D	meteorologist	String, Loop, Counting
103	2006	NOI-1	D	Points	Numbers, Loop, Geometry
104	2006	NOI-1	E	Arithmetic	Numbers
105	2006	NOI-1	E	Holydays	Numbers, Loop and conditional operator, Dates
106	2006	NOI-1	E	Maximal	Sequence, Loop and conditional operator, Geometry
107	2006	NOI-2	D	Diary	Numbers, Loop and conditional operator
108	2006	NOI-2	D	Roads	Numbers, Loop
109	2006	NOI-2	D	Neighbors	Two-dimensional array, Embedded loops

To be continued

Table 4

Tasks given at the Bulgarian competitions during the years 2001–2007 (continued)

	Year	Competition	Age Group	Task name	Keywords
110	2006	NOI-2	E	Square	Characters, Embedded loops, Print out a figure of characters
111	2006	NOI-2	E	Martenitza	Sequence, Loop and conditional operator, Fractional numbers
112	2006	NOI-2	E	Numbers	Sequence, Loop and conditional operator, Sequential processing
113	2006	NOI-3	D	Zig-zag	Two-dimensional array, Embedded loops
114	2006	NOI-3	D	Summer School	One-dimensional array, Loop, Sorting
115	2006	NOI-3	D	Sum	One-dimensional array, Loop, Modeling
116	2006	NOI-3	E	Cycle	Numbers Loop, Digits from a number
117	2006	NOI-3	E	Rectangles	Numbers, Geometry
118	2006	NOI-3	E	Three-digit numbers	Numbers, Loop, Digits from a number
119	2006	Spring	D	Zeros	Numbers, Loop, Raising to a power
120	2006	Spring	D	Sold	One-dimensional array, Loop
121	2006	Spring	D	Sticks	Numbers, Loop, Function, Recursion
122	2006	Spring	E	One or Zero	String, Loop, Modeling
123	2006	Spring	E	Prime factors	Numbers, Loop, Function, Counting
124	2006	Spring	E	Lucky tickets	Numbers, Loop, Digits from a number
125	2007	Winter	D	Bank Accounts	String, Loop, Digits from a number
126	2007	Winter	D	Seagull	Numbers, Characters, Loop
127	2007	Winter	D	Numbers	Loop, Sorting
128	2007	Winter	E	Text	String, Loop, Palindrome
129	2007	Winter	E	Accuracy	Numbers, Dates
130	2007	Winter	E	Ruler	Sequence, Loop and conditional operator, Geometry
131	2007	NOI-1	D	Picture	String, Embedded loops
132	2007	NOI-1	D	Teams	Numbers, Loop
133	2007	NOI-1	D	Airplane	Numbers, Loop, Dates
134	2007	NOI-1	E	Bulls	String, Digits from a number
135	2007	NOI-1	E	Coding	String, Loop
136	2007	NOI-1	E	Triangles	Numbers, Logical
137	2007	NOI-2	D	Sequence	One-dimensional array, Loop
138	2007	NOI-2	D	Group	Numbers, Loop, Function
139	2007	NOI-2	D	Paper	Text, Loop, Text processing
140	2007	NOI-2	E	Sum	Sequence, Loop, Sequential processing
141	2007	NOI-2	E	Numbers	String, Loop, Text processing
142	2007	NOI-2	E	Password	Numbers, Loop, Digits from a number
143	2007	Spring	D	Mushroom	Two-dimensional array, Embedded loops
144	2007	Spring	D	Melody	One-dimensional array, Loop
145	2007	Spring	D	Table	Two-dimensional array, Embedded loops
146	2007	Spring	E	KGB	String, Loop, Divisibility
147	2007	Spring	E	Rating	Array of strings, Embedded loops, Sorting
148	2007	Spring	E	Coloring	One-dimensional array, Loop, Sequential processing

References

Bulgarian web portal site for competitions in Informatics. Retrieved 21 April 2008 from `http://infoman.musala.com`

Bulgarian site for school competitions in Informatics. Retrieved 21 April 2008 from `http://www.math.bas.bg/infos`

Kelevedjiev, E. and Dzhenkova, Z. (2004). *Algorithms, Programs, Problems*. Manual for beginner's trainning in competitions and olympiads. Regalia, Sofia (in Bulgarian).

Kelevedjiev, E. and Dzhenkova, Z. (2008). Competition's tasks for the youngest school students. In *Mathematics, Informatics and Education in Mathematics and Informatics*. Spring Conference of the UBM, Borovetz, 2008.

Yovcheva, B. and Ivanova, I. (2006). *First Step in Programming with C/C++*. KLMN, Sofia (in Bulgarian).

E. Kelevedjiev is a research fellow in the Institute of Mathematics and Informatics at the Bulgarian Academy of Sciences. His field of interests includes algorithms in computer science, operation research, digitization techniques, etc. He is a member of the Bulgarian National Committee for Olympiads in Informatics since 1993; leader or deputy leader of the Bulgarian teams for many IOI's and BOI's.

Z. Dzhenkova is a teacher in the Mathematical High School in Gabrovo, Bulgaria. She is coauthor of a manual for beginner's training in competitions and olympiads in informatics. Her field of scientific interests includes education in informatics and information technology; leader of school student teams and instructor in competitive informatics.

Olympiads in Informatics, 2008, Vol. 2, 90–104

Tasks on Graphs

Krassimir MANEV
Department of Mathematics and Informatics, Sofia University and
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences
5, J. Bourchier blvd., 1164 Sofia, Bulgaria
e-mail: manev@fmi.uni-sofia.bg

Abstract. Despite missing of the topic in standard school curricula, tasks on graphs are among the most used in programming contest for secondary school students. The paper starts with fixing the used terminology. Then one possible classification of tasks on graphs is proposed. Instead of the inner logic of Graphs Theory, which is typical for some profiled textbooks, the classification approach of the general textbooks on algorithms was used. The tasks on graphs from last ten editions of the Bulgarian National Olympiad in Informatics were considered and classified in order to check the usability and productivity of the proposed classification.

Key words: graph structures, directed and undirected graphs and multi-graphs, classification of tasks on graphs, algorithmic schemes.

1. "In the beginning ..."[1]

It is a fact that the task proposed in the first International Olympiad in Informatics (IOI), held in Bulgaria in 1989, was a task on a graph. The statement of the task, as given to the students, was published in (Kenderov and Maneva, 1989) and a more simple and contemporary form of the statement of the task was included in (Manev *et al.*, 2007, p. 114).

Two years before the first IOI, an open competition on programming for school students was organized just before and in connection with the Second International Conference and Exhibition "Children in the Information Age" of IFIP, which took place in Sofia, Bulgaria, from May 19 till May 23, 1987. The contestants were divided in three age groups (junior – under 14 years, intermediate – under 16 years, and senior – older than 16 years). It is interesting that the task proposed to students of the senior group was also a task on a graph. The statement of the task, as given to the students, was also published in (Kenderov and Maneva, 1989). We would like to give here a contemporary form of this task too.

***Task 1* (*Programming contest "Children in the Information Age"*, 1987).** Let the bus stops in a city be labeled with the numbers $1, 2, \ldots, N$. Let also all bus routes of the city be given: $M_1 = (i_{1,1}, i_{1,2}, \ldots, i_{1,m1})$, $M_2 = (i_{2,1}, i_{2,2}, \ldots, i_{2,m2}), \ldots$, $M_r =$

[1] The Bible, Genesis 1:1.

$(i_{r,1}, i_{r,2}, \ldots, i_{r,mr})$, $1 \leqslant i_{j,k} \leqslant N$, $i_{j,k} \neq i_{j,l}$ when $k \neq l$. Each bus starts from one end of its route, visits all stops of the route in the given order and, reaching the other end of the route, goes back visiting all stops in reverse order. Write a program that (i) checks whether one can get from any stop to any other stop by bus; (ii) for given stops i and j prints all possible ways of getting from stop i to stop j by bus; (iii) for given stops i and j finds fastest possible way of getting from stop i to stop j by bus, if times for travel from stop to stop are equal and 3 times less than the time to change busses.

During the years after the first IOI tasks on graphs became a traditional topic in the programming contests for school students – international, regional and national. Something more – the number of the tasks on graphs given in these contests is significant.

Why this subject, which was never included in school curricula, is so attractive for programming contests? When and how do we have to start introducing of graph concepts and algorithms on the training of young programmers? Which tasks and algorithms on graphs are appropriate for the students of different age groups, and which concepts have to be introduced in order that students are able to understand and solve corresponding tasks? How should we present the abstract data type graph (and related to it abstract type tree) in data structures? These are only few of the questions that arise in the process of teaching the algorithmics of graphs. In this paper we will try to give answers of a part of them, based on more than 25 years experience of teaching algorithms on graphs, as well as preparing tasks, solutions and test cases.

In Section 2 we will introduce some basic notions, not because the reader is not familiar with them but just to escape misunderstanding, because the different authors use different terminology. In Section 3 one possible classification of tasks on graphs is presented, based on the character of used algorithms. The effectiveness and productivity of proposed classification was checked on a set of tasks on graphs from the Bulgarian olympiads in informatics and the results are presented in Section 4.

2. "... What's in a name? ..."[2]

Speaking about the terminology, we are strongly influenced by the remarkable book (Harary, 1969). As it was mentioned in Chapter 2 of this book, which is dedicated to the terminology, most of the scientists that work in the domain are using their own terminology. Something more, the author supposed that Graph Theory will never have uniform terminology. Back then, we totally accepted the terminology of Frank Harary and believed that uniformity was possible. About 40 years later we have to confess that he was right. Uniform terminology in Graph Theory still does not exist and those used by us, even if strongly influenced by that of Harary, is different from the original.

2.1. *Graph Structures*

Each *graph structure* $G(V, E)$ is defined over a finite set V of *vertices* as a collection of *links* E (see Fig. 1) such that each link is connecting a couple of vertices. The link

[2] W. Shekspeare, *Romeo and Julet*, Act II, Scene 2.

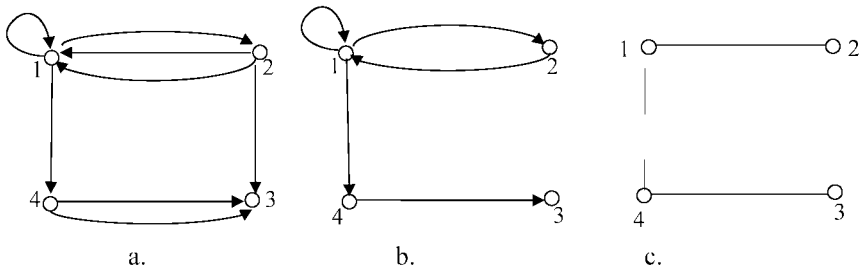


Fig. 1. Directed multi-graph (a), directed graph (b) and undirected graph (c).

is *directed* when the couple is ordered and *undirected* when the couple is unordered (2-elements subset of V). We will call undirected links *edges* and directed links – *arcs*. Unfortunately, most of the authors of books on graph structures denote the edge that links vertices v and w by the ordered couple (v, w) instead of more precise set notation $\{v, w\}$. Such notation sometimes leads to misunderstandings. We will be happy to see the tradition changed, but for the moment it seems impossible.

If E is a set of links (edges or arcs) we are speaking of *graphs*. If a multi-set E of links is considered, i.e., a repetition of elements of E is permitted, than we are speaking of *multi-graphs*. Applying ordering and repetition principles of Combinatorics, four kinds of graph structures are obtained – *directed graphs* (or *digraphs*), *undirected graphs* (simply *graphs*), *directed multi-graphs* (simply *multi-digraphs*) and *undirected multi-graphs* (simply *multi-graphs*). Distinguishing the four kinds of graph structures in the training process is important because sometimes algorithms solving the same task in different structures are different. A trivial example is the procedure of presenting graph structure in a *matrix of incidence* `g[][]`. When the structure is given in the input file as a *list of links* it is enough to assign `g[v][w]=1` for the arc (v, w), but for the edge (v, w) both assignments `g[v][w]=1` and `g[w][v]=1` will be necessary.

Special kind of links (v, v) are called *loops*. Our experience suggests that it makes sense to allow loops in directed graph structures and do not allow them in undirected, which will be our assumption through the paper. But this does not mean that loops should not be included in undirected graph structures at all. It is quite possible that an interesting task idea could presume existing of loops in a graph or multi-graph.

2.2. *Traversals in Graph Structures*

The idea of "moving" in a graph structure, passing from a vertex to another vertex that is linked with the first, is one of the most fundamental in the domain. We will call such moving in a graph a *traversal*. It is worth separating directed traversals from undirected. We will call the sequence $v_0, v_1, \ldots, v_l$ of vertices of a directed graph structure a *course of length l from v_0* to v_l, if there is an arc (v_i, v_{i+1}) in E for $i = 0, 1, \ldots, l-1$. When $v_0 = v_l$ then the course is called a *circuit*. We will call the sequence $v_0, v_1, \ldots, v_l$ of vertices of an undirected graph structure a *path of length l from v_0* to v_l, if there is an arc (v_i, v_{i+1}), for $i = 0, 1, \ldots, l-1$ and $v_{i-1} \neq v_{i+1}$ for $i = 1, 2, \ldots, l-1$. When $v_0 = v_l$

then the path is called a *cycle*. The constrain $v_{i-1} \neq v_{i+1}$ in the undirected case is crucial. Without this constrain it will happen that the undirected graph from Fig. 1(c.) contains the cycle 1,2,1 which is unacceptable. In the digraph from Fig. 1(b.), for example, the same sequence 1,2,1 is a circuit and even the sequence 1,1,1,2,1 is a circuit.

A graph structure in which there is a course, respectively path, from each vertex to each other vertex is called *connected.* Directed graph structures in which for each two vertices there is a course in at least one of the two possible directions are called *weakly connected*. When a graph is not connected, then it is composed of several connected sub-graphs called *connected components*.

Some authors prefer to call courses and paths that not repeat links and/or vertices with specific names (sometime different for both cases – lack of repeated links and lack of repeated vertices only). Others prefer to call them *simple courses* and *simple paths*, respectively (for the circuits and cycles $v_0 = v_l$ is not considered a repetition of vertex). We will use the short names courses and paths for the most frequent case when the repetition of vertices is not allowed (which imply no repetition of links too). For rare cases of repetitions other names, longer and even self-explaining could be used.

Traversals of graph structures that pass trough each link once are called *Euler traversals*. Traversals of undirected graph structures that pass trough each vertex once are called *Hamilton traversals.*

2.3. *Graph Structures with Cost Functions*

On each graph structure it is possible to define *cost function* on vertices c_V: $V \rightarrow C$, *cost function* on links c_E: $E \rightarrow C$, or both, where C is usually some numerical set of possible values (natural numbers, rational numbers, real numbers, etc. or subset of those sets). Values of the cost functions, beside *cost*, are called also *length, weight, priority, etc.* depending on the situation. If a graph structure has no cost function defined then we will presume that the cost of each vertex and link is 1. Cost functions are usually extended in some natural way on sub-graphs and other sub-structures defined in the graph structure. For example the cost of a path in a graph is usually defined as a sum of costs of its vertices, of its edges or both of vertices and edges (if applicable).

The notion *path* (or *course*) *of minimal cost*, called also *shortest path* (or *course*) is fundamental for algorithmics on graphs. Both tasks mentioned in Section 1, the task from IOI'89 and the task from the contest organized in parallel with the International Conference and Exhibition "Children in the Information Age", included searching of shortest path. Let us now reformulate them in terms introduced here.

***Task 2* (*IOI'1989*):** Let V be the set of all strings of length $2N$ composed of $N-1$ letters 'A', $N-1$ letters 'B', and 2 consecutive letters 'O'. Two strings (vertices of V) are linked by an edge if one of the strings could be obtained from the other by swapping letters 'O' and two other consecutive letters, conserving their order. The strings in which all letters 'A' are leftmost of all letters 'B' (does not mater where the letters 'O' are) are called *final*. Write a program that for a given string S finds and prints one path of minimal length (trivial cost of each edge is 1) from S to some final string. If there is no path between S and a final string, the program has to print the corresponding message.

Task 3 (***Programming contest, 1987***): A graph $G(V = \{1, 2, \ldots, n\}, E)$ is given. The set E of edges is defined by r of its paths of length $m_1, m_2, \ldots, m_r$, respectively in such a way that each edge of G is included in at least one of the given paths. The cost of each vertex is 3 and the cost of each edge is 1. Write a program (i) to check whether the graph is connected; (ii) for given two vertices v and w, to generate all paths between v and w; (iii) for given two vertices v and w, to find the path between v and w with minimal cost.

Let $G(V, E)$ be a graph with cost function c_E: $E \to C$, where C is a numeric set with non negative values. Then the function d: $V \times V \to C$, where $d(v, w)$ is the cost of the shortest path from v to w is a *distance* in classic mathematical sense of the word because (i) $\forall v, w \in V$, $d(v, w) \geqslant 0$ and $d(v, w) = 0$ iff $v = w$; (ii) $\forall v, w \in V$, $d(v, w) = d(w, v)$; (iii) $\forall v, w, u \in V$, $d(v, w) \leqslant d(v, u) + d(u, w)$.

Introducing of distance function gives us the possibility to consider the graph $G(V, E)$ as a geometric object and to define the corresponding notions. For example, a *center* of the graph G is each vertex v, which minimize $D(v) = \max\{d(v, w)|w \in V\}$ and the *diameter* of the graph G is $D(G) = \max\{d(v, w)|v, w \in V\}$. The analogy between the "geometry" of a graph and the geometry of well known Euclidean space is an origin of interesting tasks on graphs.

2.4. *Graph Structures and Relations*

Let A and B be arbitrary sets. Each subset R of the Cartesian product $A \times B$ is called a *relation*. School and university curricula in mathematics provides a large amount of useful relations: among numbers ("x is less then y", "x is less then or equal to y", "x is equal to y", etc.), among geometric objects ("the point p lies on the line l", "the line l passes trough the point p", "lines l and m are parallel" etc.), among subsets of some universal set ("A is a subset of B", "A and B intersect", etc.).

A lot of relations we could find outside mathematics, in the world around us. For example, relations among people – "x is a son of y", "x likes y", "x and y are in the same class", etc; or the very popular relation among villages "the village x is linked with the village y by a road" (similar relations could be established among city crossroads linked by streets, railway stations linked by railway roads, electricity sources, distribution centers and customers linked by electricity lines, etc.). That is why many different tasks can arise, in a natural way, in connection with a specific finite relation – abstract (mathematical) or from the real world.

Unfortunately, school curricula (and even some university curricula) use many relations without to consider the notion itself and its properties – especially the properties of relations over Cartesian squares $A \times A$ – *reflexivity, symmetry, anti-symmetry, transitivity*. Some specific relations over Cartesian squares – equivalences (reflexive, symmetric and transitive), partial orders (reflexive, anti-symmetric and transitive) and total orders (reflexive, strongly anti-symmetric and transitive) are significant both for mathematics and algorithmics.

The notion *finite relation* coincides with the notion digraph. Indeed, each digraph $G(V, E)$ could be considered as a relation $E \subseteq V \times V$ and vice versa. A finite relation

$E \subseteq V \times V$ that is symmetric (and optionally reflexive) is really a graph. That is why, each task connected with some relation could be considered as a task on a digraph or graph. Let us consider some examples. It will be helpful to fix the set V to be $\{1, 2, \ldots, n\}$.

***Task 4*:** Let $E \subseteq V \times V$ be equivalence. Find the number of classes of equivalence of E. Is this number equal to 1? If the number of classes is great than 1, find the classes of equivalence of E.

This task (really set of very similar tasks) is classic for relations of equivalence. Because equivalence is reflexive and symmetric relation, $G(V, E)$ is a graph. From the graph point of view this task could be formulated as: "How many connected components has the graph $G(V, E)$? Is the graph connected? If not, then find the vertices of each connected component of G".

***Task 5*:** Let $E \subseteq V \times V$ be total order (we will denote $(x, y) \in E$ with $x \leqslant y$) and $V' \subseteq V$, $|V'| = M$. Find a sequence $a_1, a_2, \ldots, a_M$ of all elements of V' such that $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_M$.

Of course, this is the task for *sorting* a subset of elements of a given total order. It is so popular that a specific branch of the Theory of Algorithms is dedicated to it. Anyway, the task could be formulated as a task on digraph. It is well known that relations of ordering, considered as digraphs, have no circuits. So the task for sorting a given subset of a totally ordered set will look like: "Given a digraph $G(V', E')$ without circuits. Find a course with a maximal length in G". Relation E' in this formulation is, obviously, the *restriction* of E on V'. Digraphs without circuits are very popular and have specific name – *dag* (abbreviation of **d**irected **a**cyclic **g**raphs, because some authors use the notion cycle for digraphs too).

***Task 6*:** Let $E \subseteq V \times V$ be a partial order which is not total (we will denote $(x, y) \in E$ with $x \leqslant y$ again). Find a sequence $a_1, a_2, \ldots, a_M$ of elements of V with maximal length such that $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_M$.

Formulation of this task as a task in digraph will be: "Given a dag $G(V, E)$. Find a course with a maximal length in G".

The examples given above concerned relations over the Cartesian square. But relations over the Cartesian product of two different domains could also be considered in graph formulation.

***Task 7*:** Let $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times A$ are such that $(a, b) \in R_1$ if and only if $(b, a) \in R_2$. Find a subset $\{(a_1, b_1), (a_2, b_2), \ldots, (a_M, b_M)\}$ of R_1 (or $\{(b_1, a_1), (b_2, a_2), \ldots, (b_M, a_M)\}$ of R_2, which is the same) with maximal numbers of elements such that $a_i \neq a_j$ and $b_i \neq b_j$, $1 \leqslant i < j \leqslant M$.

Relations R_1 and R_2 with mentioned above property are called *mutually reversed*. Examples of mutually reversed relations are the above mentioned couple "the point p lies on the line l" and "the line l passes through the point p". An example from real life could be the couple "the person p could do the work w" and "the work w could be done by the person p". For each couple of mutually reverse relations we can build a graph $G(V = A \cup B, R_1)$ (or $G(V = A \cup B, R_2)$, which is the same) considering elements of R_1 (R_2, respectively) as not ordered. Such graph is called *bipartite*. Searched subset M of edges such that each vertex is an end of at most one edge in M is called *matching*.

In graph formulation the task will be: "Given a bipartite graph $G(V = A \cup B, R_1)$. Find one maximal matching of G".

2.5. *Trees and Rooted Trees*

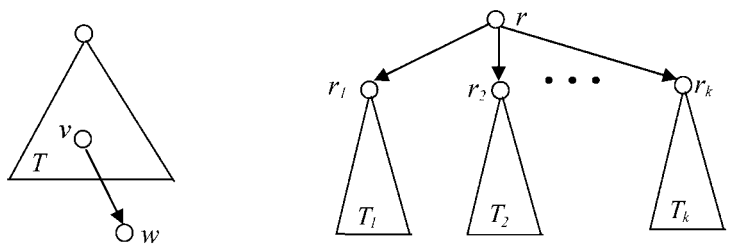
Discussion of tasks on graph structures is impossible without introducing the notion *tree*. By the classic definition, graph $T(V, E)$ is a tree if it is connected and has no cycles. For the purposes of algorithmics the notion *rooted tree* is more helpful. Two equivalent inductive definitions of rooted tree are given below.

DEFINITION 1. (i) The graph $T(\{r\}, \varnothing)$ is a rooted tree. r is a *root* and a *leaf* of T; (ii) Let $T(V, E)$ be a rooted tree with root r and leaves $L = \{v_1, v_2, \ldots, v_k\}$. Let $v \in V$ and $w \notin V$; (iii) Then $T'(V' = V \cup \{w\}, E' = E \cup \{(v, w)\})$ is also a rooted tree. r is a *root* of T' and leaves of T' are $(L - \{v\}) \cup \{w\}$. This definition (Fig. 2(a.)) is more appropriate for building of rooted trees.

DEFINITION 2. (i) The graph $T(\{r\}, \varnothing)$ is a rooted tree. r is a *root* and a *leaf* of T; (ii) Let $T_1(V_1, E_1), T_2(V_2, E_2), \ldots, T_k(V_k, E_k)$, be rooted trees with roots $r_1, r_2, \ldots, r_k$, and leaves $L_1, L_2, \ldots, L_k$, respectively. Let $r \notin V_1 \cup V_2 \cup \ldots \cup V_k$; (iii) Then $T'(V' = V_1 \cup V_2 \cup \ldots \cup V_k \cup \{r\}, E' = E_1 \cup E_2 \cup \ldots \cup E_k \cup \{(r, r_1), (r, r_2), \ldots, (r, r_k)\})$ is also a rooted tree. r is a *root* of T' and leaves of T' are $L_1 \cup L_2 \cup \ldots \cup L_k$. Rooted trees $T_1, T_2, \ldots, T_k$ are called *subtrees* of T'. This definition (Fig. 2(b.)) is more appropriate for analyzing rooted trees. Introducing the notion sub-tree it is leading to natural recursive procedures.

By definition rooted trees are undirected graphs. Anyway, Definition 1 is introducing an *implicit direction* on the edges of the rooted tree. We could say that v is a *parent* of w and that w is a *child* of v. Obviously each rooted tree is a tree and each tree could be rebuild as rooted when we choose one of the vertices for its root.

If $G(V, E)$ is a graph and $T(V, E')$ is a (rooted) tree such that $E' \subseteq E$ than T is called a *spanning* (*rooted*) *tree* of G. Graph G is connected if and only if it has a spanning tree. So, the most natural way to check whether the graph G is connected is to try to build a



a. b.

Fig. 2. Two equivalent definitions of rooted tree.

spanning tree of G. If c: $E \to C$ is a cost function on edges of $G(V, E)$ we could extend it to spanning trees of G, defining $c(T(V, E')) = \sum_{e \notin E'} c(e)$. Each spanning tree T of G with minimal (maximal) $c(T)$ is called *minimal* (*maximal*) *spanning tree* of G.

2.6. *Presentation of Graph Structures and Trees*

As in each other domain, presenting abstract data types *graph*, *digraph*, *multi-graph*, *multi-digraph* and *tree* in data structures is crucial for creating of efficient algorithms. Traditionally, the graph structures are given in input files, as it was mentioned above, in form of a *list of links* preceded by the number n of its vertices and the number m of its links and an instruction that links have to be interpreted as undirected (edges) or directed (arcs).

When the essential part of the algorithm that will be implemented on a graph structure $G(V, E)$ is an iterative instruction of form

`for e` $\in$ `G do` $\{\ldots\}$

then the list of links is a perfect presentation and the implementation will have time complexity $\mathrm{O}(m)$. If the same algorithm is implemented over a presentation of G with an *adjacency matrix* (i.e., 2-dimentional array `g` such that `g[v][w]` is the number of the links between v and w) than the time complexity will be $\mathrm{O}(n^2)$ and the implementation will be slower for graph structures with a relatively small number of links.

If the essential part of the algorithm is an iterative instruction of the form

`for` $v \in V' \subseteq V$ `do{for` $w \in V'' \subseteq V$ `do{...if` $(v, w) \in E\{\ldots\}\}\}$

then the implementation with list of links will be of complexity $\mathrm{O}(|V'||V''|.m)$ and with an adjacency matrix – of complexity $\mathrm{O}(|V'||V''|)$, which is much better.

If the essential part of the algorithm is an iterative instruction of the form

`for` $v \in V$ `do{for` w such that $(v, w) \in E$ `do`$\{\ldots\}\}$

then the implementation with adjacency matrix will be of complexity $\mathrm{O}(n^2)$. In such case it will be more appropriate to use another presentation of the graph structures – *lists of neighbors* (in undirected case) or *lists of children* (in directed case) which will give us an implementation of complexity $\mathrm{O}(m)$.

Especially for rooted trees, we would like to mention the presentation *list of parents* – an array `g[]` such that `g[i]` is the parent of `i` for each vertex that is not the root `r` and `g[r]=0`. This presentation of rooted trees is very convenient when it is necessary to build a rooted tree (spanning, minimal spanning, etc.) or to maintain it.

It is worth also mentioning that different specific tree structures (heaps, index trees, suffix trees, etc.) are an inevitable part of efficient implementation of many algorithms but discussing of such specific tree structures is far beyond the scope of this paper.

3. Classification of Tasks on Graphs

Classification of tasks on graphs is important due to different reasons. Good classification could be very helpful for preparing curricula and organizing the training process – for deciding which classes of tasks are to be taught and in which order, just to have a smooth

passing from more easy to more difficult tasks. Classification could be very helpful for preparing contests too – for avoiding tasks of a same class in one contest or similar tasks in two consecutive contests. In this chapter we will consider first some classifications of tasks on graphs and then will discuss the place of tasks on graphs among other classes of tasks used in programming contests.

3.1. *Classifications of the Profiled Textbooks*

Profiled textbooks – see, for example, (Christophides, 1975) and (Gondran and Minoux, 1984) – prefer to classify tasks on graphs following the inner, graph-theoretical, logic of the book. Our preferable way is another but it is worth discussing briefly these classifications.

In (Christophides, 1975) the classification is based totally on the theory. The following main classes of tasks are considered:

Connectivity and accessibility; Independent and dominating sets of vertices; Colorings; Centers (radii, diameters); Medians; Spanning trees; Shortest paths; Euler traversals; Hamilton traversals and traveling salesman problem; Flows; Matchings.

The approach of (Gondran and Minoux, 1984) is a bit different. The textbook first separates the following classes of tasks, for which good (polynomial) algorithms exist:

Connectivity; Shortest paths; Spanning trees; Flows; Matchings; Euler traversals.

Then the authors consider a group of tasks, for which polynomial algorithms still do not exist. Some algorithmic concepts which are useful for approaching algorithmically hard tasks are also considered – greedy, backtracking (branch and bound), dynamic programming, etc.

Classifications on the base of the inner, graph-theoretical, logic have their reasons. But they are not convenient for the education and training of young programmers. Such an approach can sometimes hide important common features of significantly different (from graph-theoretical point of view) tasks. For example, both sources referred to above consider as different topics *connectivity, spanning trees* and *shortest paths.* But the simplest way of checking the connectivity of an undirected graph structure is to try to build a spanning tree of this graph. From the other side, a spanning tree of a graph with root r, built *in breadth*, is a *tree of shortest paths* from r to each other vertex of the graph.

Without underestimating theoretical classifications of the tasks in graphs, we prefer the "algorithmic" classifications – such that collects in one class tasks, solvable by similar algorithms or, more general, by same *algorithmic scheme*. As we will see it is possible that some class of tasks could be a result of both classification approaches. This will happen when tasks of some, "theoretically" identified, class are solvable with a specific algorithmic scheme, not applicable at all or inefficient for another kind of tasks – *Euler traversals*, for example.

3.2. *Classifications of the Textbooks in Algorithms*

Classification of tasks on the basis of the used algorithms is an approach that is typical for the general textbooks on algorithms. These books are not aimed at considering the

tasks of a specific mathematical domain but to introduce the general principles and approaches of design (and analysis, of course) of algorithms. That is why these textbooks are trying, usually, to identify as much as possible tasks that are solvable by the algorithm (or algorithmic scheme) in consideration.

As a base of our attempt for classification we used the leader among the textbooks in algorithms (Cormen *et al.*, 1990) and compared it with some other popular textbooks – (Reingold *et al.*, 1977; Aho *et al.*, 1978; Sedgewick, 1990; Brassard and Bratley, 1996), as well as the most popular in Bulgaria (Наков и Добриков, 2003). It is obvious that the chapter on graphs of (Reingold *et al.*, 1977) is organized in a similar way as the profiled books on graph algorithms, so we will not consider it.

The part dedicated to graphs of (Cormen *et al.*, 1990) starts with the chapter "Elementary Graph Algorithms", which discusses the *traversals* of the vertices of graph structures called *Breadth-First search* and *Depth-First search* (BFS and DFS). Both approaches are applicable for solving different tasks related to connectivity of graph structures and the accessibility of a vertex from another vertex. But these two algorithmic schemes are used for solving some specific tasks also. BFS, for example, is building a *tree of shortest paths* in graphs without cost function on the edges and DFS is a basic step for efficient *topological sorting*, finding of *strongly connected components*, *articulation points* and *edges*, etc. All other mentioned above books consider both BFS and DFS. That is why BFS and DFS will be different classes in our classification.

The second chapter of (Cormen *et al.*, 1990) is dedicated to algorithms for finding minimum spanning tree of graphs (MSP). This topic is included in all of considered textbooks. So, *Minimum spanning tree* will be a class in our classification too. It is worth to mention that (Brassard and Bratley, 1996) discuss algorithms for MST in the chapter on greedy algorithms in a special section dedicated to applying greedy scheme on graphs (we will discus this fact later).

The next two chapters of (Cormen *et al.*, 1990) are "Single-Source shortest paths" and "All-Pairs Shortest Paths". We will suppose that the splitting of the topic *Shortest paths* in two is made by the authors just to limit the size of the chapters. No other among the considered textbooks makes the distinction. And let us append two remarks. First, the word "shortest" has to be considered in a larger sense – tasks for finding the *largest path, more reliable path, etc.*, are solvable with the same approach – *the relaxation approach.* And second, the tasks for finding the center(s), the median(s), the radius (or diameter), etc., of graphs (considered in depth only in (Наков и Добриков, 2003) are also solvable by the relaxation approach.

The last chapter of (Cormen *et al.*, 1990) is "Maximum Flow". Beside some basic algorithms for finding *Maximum flow in a network*, the chapter also considers the strongly related but specific task for finding *Maximum matching in bipartite graphs*. (Brassard and Bratley, 1996) does not consider these two topics at all and the other textbooks consider them separately.

Something that is missing in (Cormen *et al.*, 1990) but is included in all other textbooks is the *Exhaustive search* – the general way for solving a huge amount on *NP*-complete problems in Graph Theory. The tasks for finding *Hamilton traversal* of graph

and closely related *Traveling salesman* problem are the most used examples for this class of tasks. Speaking of exhaustive search in graphs we usually have in mind *backtracking* traversals. But in this class could be included any task, for the solving of which it will be necessary to generate all permutation of the vertices, all subsets of vertices (or edges), etc.

Only (Наков и Добриков, 2003) includes a chapter dedicated to such specific topic as *Euler traversal* of multi-graphs and related problems. None of the textbooks consider the topic *Games in graphs* (of type Nim and similar). From graph-theoretical point of view these tasks could be classified in the topic *Independent and Dominating Subsets*. There is a specific approach for solving these tasks (functions of Sprague-Grundy and splitting a game in sum of more simple games). We will include such specific topics in our classification too.

3.3. *Tasks on Graphs in the General Classification of Tasks*

As it was mentioned above, general textbooks on algorithms have always a chapter (or few chapters) dedicated to tasks on graphs (and corresponding algorithms). Anyway, some authors are inclined to put some graph tasks in other chapters of their books. One example, which was mentioned above, was classifying MST task in (Brassard and Bratley, 1996). The book is considering the *algorithm of Prim* and the *algorithm of Kruskal* for finding MST as *greedy*.

Such classification has a very serious reason. In the *cyclic matroid* [3] of a graph the spanning trees, and only they, are maximal independent sets. As it is well known from the theory, greedy algorithms that search a maximal independent set of a matroid with some optimal property always find the optimum. Following such logic, the algorithm of Dijkstra for the task Single-source shortest path is also greedy. Its goal is also a maximal independent set of the cyclic matroid of the graph (i.e., rooted tree) with additional optimum property – to be a tree of the shortest paths from the source.

Let us mention some other examples. In (Келеведжиев, 2001), which is a short introduction to Dynamic Programming (DP), the *algorithm of Dijkstra* for finding the *shortest path* was considered as an example for applying the DP approach and there is a reason for this too. Dijkstra's algorithm is maintaining a table of vertices for which the shortest path from the origin is found (i.e., of sub-tasks solved to the moment). Solving the task for the remaining vertices is reduced (by relaxation steps) to already solved subtasks.

As another example let us consider the exceptional book (Кирюхин и Окулов, 2007), which collects statements and solutions of tasks from the first eighteen IOI. The above mentioned task from the first IOI, that by our classification is a typical BFS task, is classified in the book as a task on a sequence. With the same success authors could classify it as *Exhaustive search*. In the way, as exhaustive search authors classified, for example, both the task for first day of IOI'1991 (*Hamilton path*) and the task "Camelot" from second day of IOI'1998, which we prefer to classify as *Breadth-first search*. Different classifications of tasks on graphs give us different points of view to the ways the task could be solved.

[3] For short introduction to Theory of Matroids see, for example, (Welsh, 1976).

3.4. *Graphs in the Proposed IOI Syllabus*

In (Verhoeff *et al.*, 2006) a proposal for a Syllabus of IOI was published. It is interesting to see the place of the specified above topics in the Syllabus. Briefly, the authors explicitly suggest excluding from the topics of IOI *matching in bipartite graphs* and *flows in networks*. And more, the Syllabus does not mention at all (and so exclude implicitly) *games of type Nim* (and related) in graphs. All other topics are covered in one or another form.

We would not like to guess the reasons of authors to exclude (explicitly or implicitly) these specific topics – general reasons for excluding topics from the Syllabus are given in the mentioned paper. We would like only to stress that tasks from excluded classes are proposed in national olympiads and could appear in task sets of future IOIs too, because the Syllabus of IOI has to be instructive, rather then restrictive. Discussion of this topic and especially attempts to exclude some topics from the Syllabus of IOI is going beyond the scope of this paper but put in front of the community very serious question: what kind of mathematical knowledge we have to give to the new generation of mathematicians – the computer programmers?

4. Tasks on Graphs in the Bulgarian Programming Contests

To conclude this paper we would like to consider the place of tasks on graphs in Bulgarian programming contests. For this purpose we checked large amount of tasks from all Bulgarian programming contests from the last 10 years published in (Infoman, 2008). A set of 85 tasks was identified and each task was classified in one of the classes that we specified in previous section. Results are given in Table 1. This will help us to realize which classes of tasks are most used in programming contests.

The tasks of each class were additionally classified by the age group for which they were proposed. This will help us to realize when the tasks of a specific class appear for the first time in competitions and which the preferred tasks for each age group are. The definitions of age groups in Bulgarian programming contest changed over the years (see (Manev *et al.*, 2007)) so we are using the following average definition: group C – 14–15 years; group B – 16–17 years; group A – 18–19 years. When a task was given during a contest for selection of Bulgarian national team, it was classified in group A.

From Table 1 it is obvious that the most preferable class of tasks in Bulgarian programming contests is *Shortest path* for graphs with cost function on edges or on edges and vertices. From the two cases – single-source and all-sources – the first dominates (19 versus 6 tasks). Tasks with classical formulation (solvable with algorithm of Dijkstra or algorithm of Floyd-Warshal) are very few – the two tasks for group C and two of the tasks for group B.

The usual way to escape classical formulation is to define the graph implicitly or to put some additional obstacles (or optimization criteria) on vertices and/or links. Sometime the shortest path task is combined with some task from different domain. As an example of such combination let us mention the following task.

Table 1

Tasks on graph from the Bulgarian programming contests for last 10 years

Class of tasks	Number of tasks	Age group		
		C	B	A
Breadth first search and related (including connectivity checking, identifying connected components, shortest path in graphs without cost function, etc.)	15	6	3	6
Depth first search and related (including topological sorting + some optimization, strongly connected components, etc.). **Remark.** Tasks solvable both by BFS and DFS are classified in the previous group.	15	2	4	9
Euler traversals	3	1		2
Minimum spanning tree	2			2
Shortest path (both *single-source* and *all-sources)* and related	25	2	7	16
Matching and flows in networks	6		2	4
Games in graph *(of type Nim)*	2		1	1
Exhaustive search	15	1	2	12
Difficult to classify	2			2

Task 8 (***Winter tournament 2000, group A***) (Infoman, 2008). Vertices V of a graph are points of the Euclidean plane and edges are line segments linking some of the vertices. Let $C \subseteq V$ are the vertices of the graph from the convex hull of V. For each vertex v of V find the closest to v vertex of C.

It is not unexpected that the second place is shared by the BFS and DFS. Because tasks solvable both by BFS and DFS (i.e., connectivity, accessibility, etc.) are classified only in BFS, it is possible to say that, in Bulgarian national contests, the couple *BFS&DFS* is even more popular than *Shortest path*. Something more, exploring a graph in depth is, or may be, the first task on graphs that young programmers have to solve. This is easy to explain – with a recursive implementation of DFS we could escape introducing the abstract data type *stack*. Objectively the BFS had to be easier to understand in age 14–15 but it will need introduction of the abstract data type *queue*.

Tasks for BFS and DFS in which the graphs are explicitly given are very rare. Usually the graph is extracted from *mazes of squares*, *spaces of situations* with an operation for transforming one situation in another (like the task from the first IOI), some *relation* (for example, the interval $[a, b]$ is included in the interval $[c, d]$), etc. The most frequently used task that is solvable by DFS is *longest course* in a dag.

The fourth most popular category in Bulgarian national programming contests is *Exhaustive search.* It is obvious that in the process of creating of tasks it is impossible to escape the numerous NP-complete tasks of Graph Theory. Especially because there are many situations from the real life, which are modeled as NP-complete tasks (traveling

salesmen, splitting group of people in cliques, etc.). It seems normal that the topics recommended for excluding in the Syllabus of IOI (matching, flows and games of type Nim) are rare. But it is strange that the tasks of the categories *Euler traversals* and *Minimal spanning tree* are very rare. We have not reasonable explanation of this fact.

We did not succeed to classify only 2 of considered tasks. One of them – *Lowest common ancestor* in a rooted tree is a popular tasks, but a specific approach for solving it is necessary. We would like to present here the second of unclassified tasks.

Task 9 (Autumn tournament, 2005) (Infoman, 2008)**.** A set V of vertices and the positive integers $d(v, w)$ for each $v,\ w \in V,\ v \neq w$ are given. Find a graph $G(V, E)$ with minimal number of edges and a positive integer length of each edge in such way that the shortest path for each couple of vertices $v, w \in V$ is equal to the given $d(v, w)$.

5. Conclusions

Graph structures are **an important origin of tasks** for olympiads in informatics. They are modeling real life situations and so the tasks become more natural and easy for understanding. Graphs are not included in classical mathematical school curriculum but most of the notions and concepts are understandable by relatively young students. As it was mentioned, tasks on graphs appear in Bulgarian national contests for student aged 14–15 years. So teaching of graph concepts and algorithms really starts at the age of 12–13 years.

The classification of tasks on graphs, proposed in this paper, is **one of many possible**. It could be discussed and ameliorated. It is possible a classification of tasks on graphs to be based on another principles. But some **classification of tasks on graphs is necessary** for each team of teachers that is coaching contestants in programming. On the base of the proposed classification we could conclude that in the "Bulgarian model" of teaching graphs concepts and algorithms we are starting with BFS and DFS on age 14–15 years. At the age of 16–17 years shortest path tasks are included in the training process. At the age of 18–19, beside algorithmically hard tasks, solvable by different kind of exhaustive search, some specific topics, like Matching in bipartite graphs, Flows in networks and Games of type Nim, also appear in the sets of contests' tasks.

Classification of tasks together with analysis of results during the contests could help us to better organize both the training process and the contests (national and local) – to identify kinds of tasks that are not appropriate for some age group, to identify kinds of tasks that are included in the tasks sets more or less frequently than usual, etc.

We would like to thanks numerous authors of task on graphs for Bulgarian programming contests as well as all Bulgarian contestants for their work, which made this research possible.

References

Aho, A., Hopcroft, J. and Ulman J. (1978). *Data Strucures and Algorithms*. Addison-Wesley.
Brassard, G. and Bratley, P. (1996). *Fundamentals of Algorithmics*. Prentice Hall.

Christophides, N. (1975). *Graph Theory. An Algorithmic Approach.* Academic Pres.
Cormen, T.H., Leiserson, Ch.E. and Rivest R. L. (1990). *Introduction to Algorithms*, Second Edition. The MIT Press.
Gondran, M. and Minoux, M. (1984) *Graphs and Algorithms*. John Wiley & Sons.
Harary, F. (1969). *Graph Theory*. Addison-Wesley Publishing Company.
Infoman (2008). Bulgarian portal for competitive programming. `http://infoman.musala.com` (visited in February2008)
Kenderov, P. and Maneva, N. (Eds.) (1989). *Inernatonal Olympiad in Informatics*. Sofia.
Manev, K., Kelevedjiev, E. and Kapralov, S. (2007). Programming contests for school students in Bulgaria. *Olympiads in Informatics*, **1**, 112–123.
Келеведжиев, Е. (2001). *Динамично програмиране.* Анубис, София.
Кирюхин, В.М. и Окулов, С.М. (2007). *Методика решения задач по информатике.* Международные олимпиады. БИНОМ, Москва.
Наков, Пр. и Добриков, П. (2003). *Програмиране = + + Алгоритми.* Top Team Co, София.
Reingold, E.M., Nivergelt, J. and Deo, N. (1977). *Combinatorial Algorithms. Theory and Practice*. Prentice Hall.
Sedgewick, R. (1990). *Algorithms in C*. Addison-Wesley.
Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A Proposal for an IOI Syllabus. *Teaching Mathematics and Computer Science*, **VI**(I), 193–216.
Welsh, D.J.A. (1976). *Matroid Theory*. Academic Press.

K. Manev is an associate professor in discrete mathematics and algorithms at Sofa University, Bulgaria. He is a member of the Bulgarian National Commission for Olympiads in Informatics since 1982 and was a president of the commission (1998–2002). He was a member of the organizing team of first (1989) and second (1990) IOI, president of the SC of Balkan OI'1995 and 2004, leader of the Bulgarian national team for IOI'1998, 1999, 2000 and 2005 and BOI'1994, 1996, 1997, 1999 and 2000. In 2007 he was a leader of Bulgarian team for First Junior Balkan OI. From 2000 to 2003 he was an elected member of IC of IOI. In 2005 he was included again in IC of IOI as a representative person of the host country of IOI'2009. He is author of more than 30 scientific papers, 1 university textbook and 9 textbooks for secondary schools.

Olympiads in Informatics, 2008, Vol. 2, 105–114

Challenges in Running a Computer Olympiad in South Africa

Bruce MERRY
ARM Ltd
110 Fulbourn Road, Cambridge, CB1 9NJ, England
e-mail: bmerry@gmail.com

Marco GALLOTTA, Carl HULTQUIST
Department of Computer Science, University of Cape Town
Private Bag, Rondebosch 7701, Cape Town, South Africa
e-mail: marco@gallotta.co.za, chultquist@gmail.com

Abstract. Information and Communication Technology (ICT) in South Africa lags behind that of the developed world, which poses challenges in running the South African Computer Olympiad (SACO). We present the three-round structure of the SACO as it is run today, focusing on the challenges it faces and the choices we have made to overcome them. We also include a few statistics and some historical background to the Olympiad, and sample questions may be found in the appendix.

Key words: digital divide, olympiad in informatics, problem solving, South Africa, SACO.

1. Introduction

For historical reasons, the level of ICT infrastructure in South African schools spans a wide range. At schools in affluent suburbs, computers are available to students, Internet access is common, and most students can take optional classes in computer studies (either at their own school or a nearby centre). On the other hand, poorer schools lack the most basic of facilities, and students have no access to computers or the Internet. This is sometimes referred to as the *digital divide*.

This makes organising a representative computer olympiad challenging. We would like to involve as many students as possible, to foster interest in computer science and computer programming amongst talented students. But how can one run a computer olympiad for students with no access to computers? South Africa is larger than most European countries, so gathering students in one place is neither practical nor cost-effective. And even if this barrier can be overcome, reliable Internet access is even less common than computers, so coordinating and marking is a further challenge.

At the same time, South Africa takes part in the International Olympiad in Informatics (IOI), and we need a mechanism to select teams. We thus need to run a contest of comparable standard in order to select and train a team to represent South Africa at the IOI.

Today, the South African Computer Olympiad (SACO) features three rounds, described in detail in the following sections. This is followed by some statistics showing correlations of scores between rounds.

2. First Round

The first round is aimed at involving as many students as possible. It is a pen-and-paper round, similar to a mathematics olympiad, but with more focus on logic and programming. The question paper is mailed out to schools (via postal service, not e-mail) in advance, and teachers at the schools administer and mark the submissions. Answers are designed to be objective (often multiple choice or a number) rather than subjective (for example, an essay), so that teachers do not require any computer knowledge.

The round is offered in two divisions, junior and senior. The senior division is aimed to students in grades 10–12 (roughly 15–18 years old), while the junior division is restricted to students in grade 9 and lower. In the South African education system, subject choices are made when entering grade 10, so schools and students can use the results of the junior division to guide subject choices, while the senior division is helpful in making career choices.

The same paper is used for both divisions in this round and they are marked in the same way. The divisions are only distinguished when the students are ranked against one another. This is because having separate papers every year would add more difficulty in setting them, and in getting the teachers to photocopy and administer them. The paper is made to be like an aptitude test and the results are therefore valid for a range of ages – one just expects less from the average junior. Since a single paper is used for a wide age distribution and such a large variation in skill level within a division, the aim is to broaden the difficulty of the questions as much as possible.

Trying to gather and collate all the results from the hundreds of schools taking part would be an enormous task. Instead, certificates for the top three seniors and top three juniors are sent to each school, and results are not further compared. With enormous differences in education standards between advantaged and disadvantaged schools, a student who obtains 50% in a rural disadvantaged school probably has more potential than one who obtains 80% in an affluent urban school. The ranking within schools recognises this issue.

This format was first introduced in 2003, where it attracted 11 123 participants (South African Computer Olympiad, 2007). The junior division was added in 2006, and participation immediately increased to 31 926. In 2007, participation was 33 893.

3. Second Round

The second round of the SACO requires a computer. It is open to anyone, regardless of participation in the first round – this removes the need to ensure correct and consistent marking of the first round between schools.

The style of the problems is very loosely similar to the IOI, in that problems are algorithmic and intended to reach a specific answer, rather than testing the ability to build a user interface, database or other type of system. As with the first round, the objective nature of the answers makes it easier for teachers who have no experience in programming to mark solutions.

The input test data are included in the problem description, and students are required to submit both their source code and printouts of test runs on those test cases. The advantage of known data is that minimal work is required by teachers marking the paper, in that they do not need to compile or run submissions. An IOI-style automatic marking system is infeasible, as it would require Internet access, and would also cause difficulties for students not used to dealing with strict input and output formats or issues arising from differences between their local setup and the marking server. The main disadvantage of known data is that we usually have at least one test case that can be solved by hand, and some students tweak their programs until the desired answer is obtained without regard for the underlying bugs in their programs. Some students go further and simply hard-code answers into their code.

Because the results of the second round are used to select participants for the third round, the papers are re-marked centrally. Rather than re-mark all papers, schools are asked to send in their best result, and only asked for their second-best result when it is possible that the school may have two participants in the third round. Results are sent by postal service, and include the printouts of source code and test runs. The bulk of the points are awarded for producing correct output to the specified test cases. A small number of points are awarded for programming style, largely as a mechanism to break ties, but also to penalise solutions that have been written to solve only the test cases specified in the question.

An unfortunate consequence of using the postal service to gather results is that many students and teachers fail to follow the instructions, and by the time this is discovered it is too late to do anything about it. It is quite common to receive solutions that are missing either the source code or the sample run printouts, and these submissions are regrettably discarded.

As with the first round, a junior division, called *Start*, is offered to students in grade 10 and below. The change of age groups from the first round is due to programming being introduced in the second round. Schools do not teach any programming skills before grade 10 (and very few in grade 10). Trying to have a junior division limited to grades 9 and below would result in very few entries.

These students participate for enjoyment and experience, and are not eligible for the third round. Certificates are also sent to schools for the top three participants in each division. Unlike the first round, separate papers are set for the juniors and seniors, although some questions are shared between the divisions to reduce the amount of work required in problem-setting.

The second round dates back to 1987, when 1 750 students participated, with a similar format to today (although it was the first round until 2003, when the current first-round format was introduced). Perhaps surprisingly, participation has not grown steadily, but

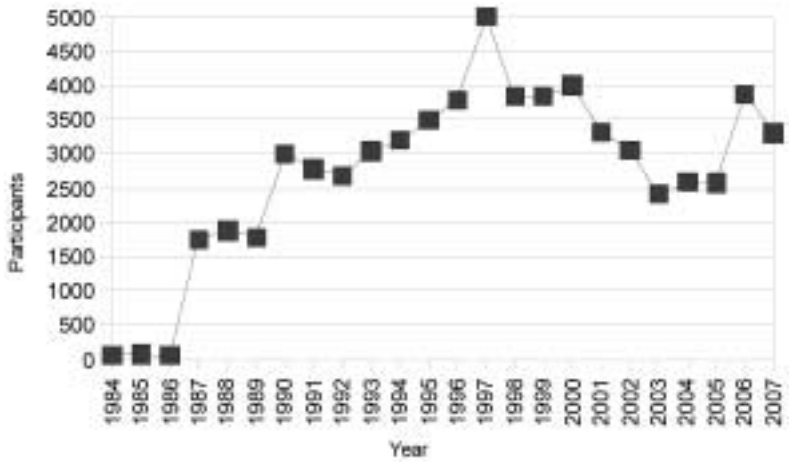


Fig. 1. SACO Second Round participation.

has varied considerably over time. The peak of 4 994 registrants occurred in 1997, when South Africa hosted the IOI, after which participation decreased. In 2003 the pen-and-paper first round was introduced, which reduced participation in the second round from 3 056 to 2 409. The junior division, introduced in 2006, has proved popular, with participation increasing to 3 873 that year. Fig. 1 shows participation in the second round since its inception.

4. Third Round

The best contestants from the second round are invited to participate in the third and final round of the SACO. The exact number of participants varies from year to year, with the aim of using a natural cut-off in the scores rather than forcing ties to be broken. Typically, 15 to 20 contestants are invited, but this number varies depending on funding.

The final round is an on-site event, hosted over the last ten years at the University of Cape Town (UCT). Students from outside Cape Town are provided with flights and accommodation, so cost does not prevent anyone from taking part in the final round. As most former IOI participants in South Africa carry on to study at UCT, there is never a shortage of on-site judging staff.

As the final round contains problems that are at a level far greater than the students encounter at school, they are sent training material prior to the final round to help them prepare. This includes printed copies of previous final round papers and pointers to various online resources. Due to participants being spread across the country, personal training cannot be provided and they therefore often resort to self-training.

The competition format of the final round follows that of the IOI quite closely, and new trends in task descriptions, types, compilers and so on are quickly adopted. As with the IOI, the contest consists of two days, each with a five hour contest featuring three

problems, and with a similar automated online grading system. While the problems are similar in style to the IOI, they are of slightly lower difficulty so that all the contestants are able to at least attempt them.

The only major technical difference from the IOI is in the languages offered: the IOI-standard languages of C, C++ and Pascal are available, but Java and more recently Python are also provided. Java is the main language taught in South African schools; Python was added due to the backing of a sponsor that wished to grow the language in South Africa, and it has proven extremely popular as it is easy to learn and offers powerful features not easily accessible in Java. In fact, although the top six Python users in the second round are guaranteed a place in the final (so that six Python prizes can be awarded – see below), these top six have always done well enough to earn an invitation without this provision.

We have found that although Java programs are usually somewhat slower than equivalent C programs (a frequent objection whenever a proposal is made to introduce Java to the IOI), the speed is sufficiently comparable that we can use the same time limits for Java as the other languages. Python, on the other hand, is a scripting language and is 1–2 orders of magnitude slower than the other languages. We have thus implemented different time limits for Python. The ratio of time limits between Python and other languages is reviewed each year based on the performance of reference solutions. In the 2007 SACO, the ratio used was 10.

At the IOI, the afternoons after the contests are free time for the contestants to review their scores and make appeals. At the SACO, this time is somewhat more structured. The judges lead a discussion of proposed solution methods (often leading out of more informal discussion during the lunch break), and after the first day of competition there is commonly some training on general topics.

The SACO offers prizes, and winners are often offered scholarships and bursaries, so the judging is as strict and impartial as the IOI, with no opportunity to "just fix one bug" or "just correct the file name". Unfortunately, this also limits the degree to which it can be used as a training opportunity, because we are unable to help with programming questions during the contest, and also cannot provide one-on-one help beforehand. Since the introduction of junior divisions into the earlier rounds, we have also had a semi-official junior division of the final round, to which a few (around six) top-performing juniors from the second round are invited. As this is a for-fun event with no cash prizes, the judges are free to provide hints and advice during the contest, and this forms a valuable learning experience for the contestants. We believe this approach has been successful, with several junior contestants returning as regular final-round contestants in later years.

Table 1 provides statistics on the language usage at the final round over the past. Before Python was introduced, the majority of students used Pascal and Java, the languages taught at schools. The gradual increase in Java usage corresponds to the increased number of schools moving from Pascal to Java as a teaching language. The small number of C++ users are typically former IOI participants who were required to learn either C++ or Pascal for the IOI.

The sudden shift to Python upon its introduction in 2005 is immediately evident from the data. This is mostly due to the sponsorship of cash prizes for the top Python users, which are significantly larger than the standard prizes.

Table 1
Language distribution in the third round, 2002–2007

	C/C++	Pascal	Java	Python
2002	0 (0%)	8 (80%)	2 (20%)	
2003	3 (25%)	6 (50%)	3 (25%)	
2004	3 (25%)	3 (25%)	6 (50%)	
2005	0 (0%)	5 (21%)	4 (17%)	15 (63%)
2006	0 (0%)	0 (0%)	3 (20%)	12 (80%)
2007	0 (0%)	1 (5%)	6 (29%)	14 (67%)

5. Statistics

We have collected a limited set of data from the 2007 contest, consisting of the scores for student whose second-round papers were centrally graded. Since this is only done where there is a chance that a paper is within the top hundred schools (for example, submissions that only attempted one question are not graded), this is not a statistically random sample. Nevertheless, some interesting results can be obtained from this data.

Fig. 2 shows a scatter-plot of scores in rounds 1 and 2, for seniors in round 2. It should be noted that of the 126 students for whom data was captured, only 76 entered the first round, and only those results are shown in the figure (the reason for this is not known, but some students with the ability to do well in the second round may not consider the first round sufficiently challenging to interest them). The triangular shape of the figure is interesting: it suggests a high score in the first round indicates a capability with problem-solving, but that this not does always translate into the ability to implement solutions. Many students in South Africa do not have access to a programming course at high school, and potentially talent is being wasted due to lack of education.

We also computed the correlation of the scores between these rounds. The Pearson product-moment correlation coefficient was 0.369. Under the assumption of a normal distribution, this would be highly statistically significant, but this is not necessarily a valid assumption as capturing results only for top students will skew the distribution. The Kendall tau (Kendall, 1938) value (a non-parametric measure of correlation) is 0.245, and this is also highly statistically significant ($p = 0.0024$).

Fig. 3 shows the same comparison between the second and third rounds. Here the data is complete, as we have second and third round results for all 22 participants in the third round. The graph suggests that a very strong performance in the second round is a good indicator of a strong performance in the third round, but that a weaker performance in the second round is not necessarily indicative of performance in the third round.

The Pearson's product-moment correlation coefficient in this case is 0.54, and is highly statistically significant ($p < 0.01$), although again this may not be statistically valid as the second round performances will not be normally distributed. The Kendall tau value is 0.31 and the p-value is roughly 0.05; R (R Development Core Team, 2007) warns that it cannot compute an exact value when there are ties.

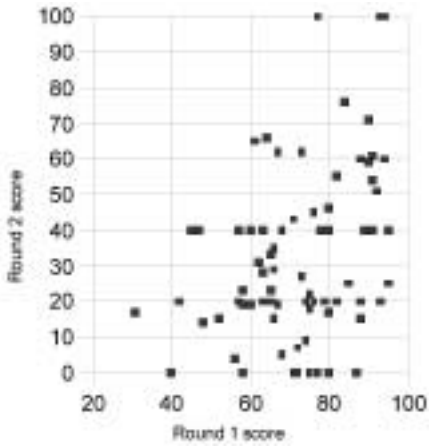


Fig. 2. Scores in round 2 against round 1.

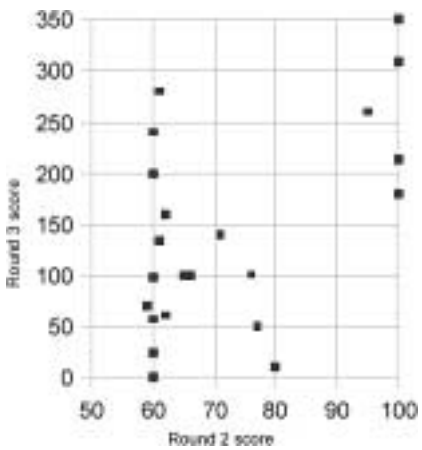


Fig. 3. Scores in round 3 against round 2.

6. Conclusions

In spite of the limited ICT infrastructure in South Africa and South African schools, we are able to run a large and successful computer olympiad that hopefully encourages students to pursue careers in computing. This is achieved by limiting the use of technology in each round to what is generally available. The first round requires no computer, and so it is accessible to all students even though few schools have computers. Our statistical analysis also shows that the first round has provided a more accessible medium for students with strong problem-solving abilities, who have not yet developed the skills to master programmatic problem-solving.

Of course, a programming contest should not be run completely without computers, and they are required for the second round. However, we use printouts rather than internet access for submission, and attempt to keep marking as simple as possible so that participation is possible even when there is no trained computer studies teacher at the school. We can also observe from Fig. 2 that students who perform well in the second round, have similarly strong performances in the first round – indicating overall strong problem-solving abilities.

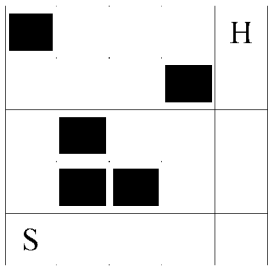
By the final round, we can provide an experience similar to the IOI, as the small number of contestants affords us the ability to bring them all to a single site and use a web interface on the local network. Fig. 3 shows the value of this round in selecting an IOI team: while there is some correlation between second- and third-round performances, many of the finalists had similar scores in the second round (around 60), but could be separated by the more challenging conditions of the final.

7. Appendix: Sample Problems

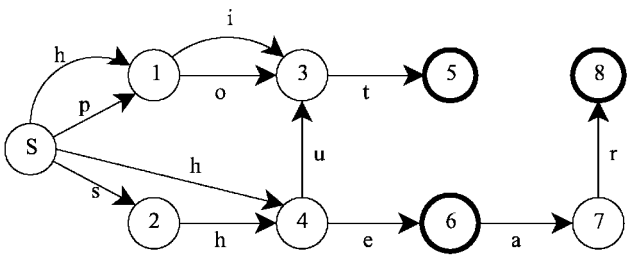
7.1. *First Round*

The following are samples of easy, medium and difficult problems from the 2006 Olympiad:

1. Imagine a country called SACO that uses 5c and 7c coins. Which of the following amounts cannot be paid using only 5c and 7c coins?
   1. 27
   2. 26
   3. 24
   4. 23
2. Sally (S) wants to go home (H). She can only move up or right one square each time. She is not allowed to go through black squares. How many paths can she pick from to go home?



3. A finite-state machine (FSM) is a . . . (explanation of an FSM follows). What words does the following FSM recognise?



7.2. *Second Round*

The following is a sample question taken from the 2007 Olympiad.

Description

Strings are just a series of characters “strung” or joined together. Substrings are strings that are, in fact, just a part of a larger string. One might, for various reasons, wish to find

if a string is merely a substring of another string, sometimes disregarding such things as case (UPPER and lower) or punctuation.

Task

Your task is to write a program that finds and prints all occurrences of a word (substring) within a piece of text. This word may be hidden, it may contain spaces or punctuation, and it might appear with different capitalization. The program must accept 2 strings, the first being the main string, and the second the substring that is to be searched for in the main string. If no substrings are found, "No strings found" must be printed.

Constraints

The length of each string will be < 255 characters.

Sample Run

Input

```
It's behind the intercom. Put erasers to one side computer
```

Output

```
com. Put er
```

Test your program with

```
This suit is black!!
not

"You thought your secrets were safe. You were wrong." - Hackers
gh

Donald likes Mall shops where he and his friends discuss
idealism all day long.
Small
```

7.3. *Third Round*

Third round questions are typical of the IOI and similar contests, so we do not include any samples here.

References

South African Computer Olympiad (2007). *History and Statistics – South African Computer Olympiad.* Retrieved 9 March 2008 from http://www.olympiad.org.za/history.htm

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org

Kendall, M. (1938). A new measure of rank correlation. *Biometrika*, **30**, 81–89.

B. Merry took part in the SACO from 1995 to 2000. Since then he has been involved in running the contest, as well as South Africa's IOI training program. He obtained his PhD in computer science from the University of Cape Town (UCT) and is now working as a software engineer at ARM.

M. Gallotta is the deputy coach of the SA IOI team. After participating in the IOI in 2004, Marco is currently in charge of task selection and the technical side of running the SACO, as well as being the coach of the ACM ICPC team at UCT. Having recently obtained a BSc (Hons) in computer science at UCT, he is now there temporarily as a research assistant working on RoboCup.

C. Hultquist has been involved with the SACO since 1995, when he first started taking part as a contestant. Upon commencing his studies at UCT, he became involved in the running of the contest and the training of South Africa's IOI team. He recently started working for D.E. Shaw & Co. (U.K.), Ltd, and is also putting the finishing touches to his PhD in computer science.

Olympiads in Informatics, 2008, Vol. 2, 115–121

Naturalness in Tasks for Olympiads in Informatics

Pavel S. PANKOV
International University of Kyrgyzstan
A. Sydykov str., 252, Apt. 10, 720001 Bishkek, Kyrgyzstan
e-mail: pps50ster@gmail.com, pps50@rambler.ru

Abstract. There are two main ways to invent tasks for olympiads of a high level: one way is to invent or choose an effective algorithm and compose a corresponding subject, and another way is to think out a real situation or task and try to formalize it. The second way is more difficult because it is multi-stage: the author needs to find some effective algorithm for a task obtained; if the best algorithm is obvious or the only algorithm seems to be exponential then we need to rework the formulation, etc. But by our opinion the second way is preferable because it can yield original tasks with natural, short and elegant formulations and give less advantage to experienced participants. We shall consider the second way in detail in this paper.

Key words: olympiads in informatics, tasks, naturalness.

1. Natural Ways to Generate Tasks

Diks (2007) mentioned the following phases of the task preparation process: review of task ideas, formulation, analysis, verification and calibration. This sequence may be applied not only to informatics but to other sciences as well. Since we have not found publications on generating of task ideas in informatics, we shall review a way to generate ideas, basing on actions in spaces which we call *natural*, and include our own experience.

1.1. *Choosing a Space*

Firstly, a space is to be chosen. Traditionally, there are used the following types of spaces:

S1) integer numbers (one-dimensional grid);
S2) pairs of integer numbers (plane grid) (often, it is introduced as ”grid of streets in the host town of the olympiad”);
S3) triples of integer numbers (space grid), but tasks on such space are usually too difficult for solving;
S4) a graph;
S5) space of solids rolling on a plane (an intermediate between S2 and S3);
S6) ring (a segment with glued ends) with finite number of elements (an intermediate between S1 and S4).
We propose also:
S7) two connected rings (figure-of-eight);

Certainly, S7 is a kind of S4 but the general graph demands a vast description since ”figure-of-eight” is self-explanatory.

S8) integer grids on non-Euclidean spaces, e.g., topological torus, Moebius band (Weeks, 1985). They are defined not as manifolds in Euclidean space but as arrays with gluing of edges as follow:
Define a rectangular grid
$G = \{(X, Y) | 0 \leqslant X \leqslant M, 0 \leqslant Y \leqslant N; M, N, X, Y$ are integers$\}$.
A *Moebius band* is obtained from G when, for all $Y = 0, 1, \ldots, N$, points $(0, Y)$ and $(M, N - Y)$ are glued. A *topological torus* is obtained from G when, for all $Y = 0, 1, \ldots, N$, points $(0, Y)$ and (M, Y) are glued and, for all $X = 0, 1, \ldots, M$, points $(X, 0)$ and (X, N) are glued.

Since the 1970s, the topological torus has appeared in computer games naturally: if an object disappears beyond one of the edges of the screen then it appears from the opposite edge.

1.2. *Choosing Actors*

Further, ”actors” (moving, changing objects) are to be chosen. Involving more than one actor provides a *game*. But many interesting tasks with more than one actor could be generated without the idea of being games, for instance, by means of cooperation of actors. Something more, the organization connected with the proposal of a game-task during an olympiad in informatics (with preparing auxiliary libraries, describing interfaces, etc.) is too difficult. But some games can be imitated by means of formulating the aim of a task as minimax.

Traditionally, we have the following configurations of actors:

A1) a point is used as an actor (a point can move by 1 or ”jump” but the rules of jumping must be very simple);
A2) some rectangles.
We propose also:
A3) two or three points (or many points with very simple conditions);
A4) moving ”train” of a length of one edge or one arc within S4;
A5) moving ”train” of a given length within S2, S6, S7.

1.3. *Choosing Actions*

The main action proposed is natural *moving*. Moving a train along a graph or a grid is a consecutive passing of its vertices by the head of the train, by its intermediate points (if its length is greater than 1) and by its tail. Simple *cutting*, *gluing*, *deleting* and *adding* are also natural actions.

1.4. *Choosing Conditions, Restrictions and Obstacles*

Conditions may be natural, i.e., actors must/cannot

C1) coincide;
C2) pass;
C3) overlap;
C4) touch;
C5) be seen;
C6) cross;
C7) cross itself (for a train).
For A3 natural conditions are also:
C8) be near/far each from other.
Another natural kind of restrictions is the following:
C9) an actor can make only a given number of steps.
Traditional type of obstacles is a labyrinth but some points or rectangles can replace it.

1.5. *Aims and Composing of Tasks*

The aims may be as following:

G1) to reach/build/compose something in a minimal number of steps;
G2) to build/compose the least/greatest object;
G3) to find the shortest/longest way;
G4) to catch another actor; to escape from another actor in a minimal time; to escape with minimal number of steps or expenditures for all possible actions of another actor (i.e., if its behavior is optimal).

By describing the space, the actors with their possible actions, the conditions, restrictions and obstacles, and declaring the aim, we obtain a task of optimizing or determining if the goal is "impossible or possible" and optimizing if the goal is possible.

Even if the author is sure the aim is attainable in all cases of the task, the output meaning "impossible" must be provided in the description of all permitted outputs of the task.

2. Grading System

Certainly, the author must try to find the best algorithm to solve the task. If s/he is sure that the best algorithm has been found (constructed), s/he is to think about possible weaker algorithms or even wrong algorithms such as a "greedy" ones, and compose the set of tests (as it is described in some papers, Diks (2007)). Also, the author must keep in mind that sometimes the best algorithm cannot be found within a restricted time. For example, the task in the paper Pankov (2005) has a solution of time complexity $O(1)$ operations but it is impossible to find it during a few hours.

At the same time, if the author cannot find the best algorithm then the natural tasks give the following ways of composing the set of tests.

2.1. *"Evident" Solutions*

For some of natural tasks the answer (the best answer) is "seen" by a human for all initial data within given restrictions. If the author is certain that the answer is self-evident then s/he may compose as many sufficiently different cases as necessary covering all aspects of the task. A good solution (algorithm) by a competitor must solve many of them. In other words, we propose the hypothesis: if the answers for all initial data are evident for a human then there exists a good algorithm solving the task on the modern computer during an appropriate time.

2.2. *Open-Ended Tasks*

Grading of such tasks is considered, for example (Kemkes, 2007). We will mention some known procedures.

Compose as many as necessary sufficiently different cases covering all aspects of the task. Consider them for a task on maximization. Let M be the maximal number of points for any test.

One of the ways is to write a simple algorithm yielding any boundaries A_-, A_+ for the (unknown) result. If $A \leqslant A_-$, then a result A obtains 0 points else $(M(A - A_-)/(A_+ - A_-)$ rounded down) points.

Another way is to compare the result with the records of all other contestants. Let $A_{\max}$ be the best result of all contestants in this test. If $A \leqslant A_{\max}/2$ then a result A obtains 0 points else ($MA/A_{\max}$ rounded down) points.

3. Examples of Tasks

Some of tasks built by means of above techniques were published in (Pankov, 2000; 2003; 2007). Tasks 1, 2, 3, 4 listed below are generalizations of ones given at olympiads in informatics of different levels in Kyrgyzstan in 2004–2008.

Task 1. Given a graph, its vertices are "houses" (less than 7). The Instrument has counted mice under each of houses at different moments. During all this measuring, each mouse could pass to another neighbor house only once. Write a program to find the least possible number of mice.

Example. Six houses form a ring. Input: 9, 0, 1, 0, 0, 2. Output: 10. [Two mice under the first house and two mice under the sixth one could be the same].

Generation: S4 or S5; A3; C9 (one step); G2.

Task 2. A graph is given. Firstly, the head H and the tail T of a train are in two neighbor vertices. Write a program finding one of the shortest ways to be passed by the train (moving forward only) in order to put its head to the primary position of T and its tail to the primary position of H.

Example. The graph contains vertices A, B, C, D and edges AB, BC, BD, CD. Firstly, $HT = BA$. One of the shortest ways for H is $B - C - D - B - A$.

Generation: S4; A4; G3.

Task 3. Let the streets in the city form a rectangular grid (of given size). The firm Logic [sponsor] is situated at a given crossing (X, Y). Two friends wish to come to the firm. Now the first is at the crossing $(X1, Y1)$, the second is at the crossing $(X2, Y2)$. Because of plentiful snowing they wish to minimize the trampled path (the sum of paths trampled by the first, by the second and by the both going together). Write a program calculating the minimal length of such path.

Generation: S4; A3; G3.

Task 4. At night, a mouse is anywhere within a long ditch of "figure-of-eight" of length 2008 meters, the first ring of the ditch is numbered from 0 till 1004 (from the cross to the cross) and the second ring is numbered from 1004 till 2008 (the points with numbers 0, 1004 and 2008 coincide). The mouse can run quickly but cannot climb out. Two men with sacks stand at given points $X1$ and $X2$. The men's velocity is 1 meter/second. Write a program calculating the minimal time to catch the mouse in any case.

Example. Input: $X1 = 500,\ X2 = 504$; output: 1006.

Generation: S7; A3; C5; G4.

Task 5. A piece of the upper half-plane is cut by broken (not self-touching) line connecting N points: $(X[1], Y[1] = 0), (X[2], Y[2] > 0), \ldots, (X[N-1], Y[N-1] > 0), (X[N], Y[N] = 0)$. Given $(2N-2)$ integer numbers $X[1], X[2], Y[2], \ldots, X[N-1], Y[N-1], X[N]$, write an algorithm detecting whether this piece can be extracted from the half-plane by means of motion within the plane.

Generation idea: what surfaces can be punched? But "surfaces" are too difficult; after discussion they were changed to "figures" in S2.

Comment: The author had in mind but did not write "by means of parallel shift" because he was sure that the motion to extract could be parallel shift only and including these words would be a prompting.

This task was given at the III All-USSR olympiad in Informatics, held in Kharkov at 1990, and was solved by pen-and-paper. During the olympiad, all organizers and contestants also thought that the only possible motion was a parallel shift and all algorithms investigated possibility of such a motion only. But just after the closing ceremony one of the contestants found an example of "upper half of a (narrow) crescent" which can be extracted by rotation!

This task demonstrates both dangers and interest arising while implementing the proposed approach.

4. Proposals and Conclusion

We propose some tasks built within the proposed approach.

Task 6. Given a graph (with less than 10 vertices) and two sets B(initial positions of wanderers) and E of its vertices, $|E| \geqslant |B|$. Write a program finding the least number of

steps to move all wanderers from B to E under the condition that they do not meet each other.

Generation: S4; A3; C8; G3.

Comment: This is one of the simplest examples of possibilities of A3.

Task 7. Consider a rectangular grid $0 \leqslant X \leqslant M, 0 \leqslant Y \leqslant N$.

A) For all $Y = 0, 1, \ldots, N$, points $(0, Y)$ and $(M, N - Y)$ are the same; or

B) For all $Y = 0, 1, ..., N$, points $(0, Y)$ and (M, Y) are the same; for all $X = 0, 1, \ldots, M$, points $(X, 0)$ and (X, N) are the same.

Write a program calculating

C) the shortest way between two given points $(X1, Y1)$ and $(X2, Y2)$; or

D) the shortest cycle connecting three given points $(X1, Y1)$, $(X2, Y2)$ and $(X3, Y3)$ along the grid.

Generation: S8; G3.

Comment: This is an example of possibilities of S8.

Task 8. The head H of a train of length N is at the point $(0, N)$ and its tail T is at the point $(0, 0)$. The train can move (forward only) along edges of the rectangular grid (pairs of integer numbers) not self-touching and cannot pass given points $(X1, Y1), (X2, Y2), \ldots, (XK, YK)$. Write a program finding one of the shortest ways to be passed by the train in order to put H at the point (XH, YH) and at the same time to put T at the point (XT, YT). Main restriction: $|XH - XT| + |YH - YT| \leqslant N$.

Generation: S2; A5; C4; G3.

Task 9. Cut a given rectangle with integer sides by two segments parallel to its sides (to three or four rectangles with integer sides) and

A) compose of these rectangles (without overlapping) a polygon (with all angles right) of the least possible perimeter; or

B) shift and overlap them to compose a polygon of the least possible area. (Only parallel shift is permitted).

Generation: S2; A2; C3; G2.

Let us demonstrate composing a task on a given theme. There is the Fig. 1 *Regions that hosted finals of the NOI (15 towns)* in the paper Dagiene (2007). *Idea*: "Two friends with bicycles decided to make photos of these towns for the illustrated history of the NOIs". Choose the endpoints: "Now (in the morning) they are in Vilnius (the capital; 16th town) and must return here". Further, the array of distances (may be, in hours rather than kilometers) between some pairs of these 16 points must be given. Also, choose the time necessary for making photos in every town (for instance, 3 hours). To make the task more realistic, add: "one can ride or make photos not more 12 hours a day." Thus, we obtain the *Task 10*: "Write a program calculating the minimal number of days for such enterprise under given conditions".

Generation: S4; A3; C2; G3.

We hope that tasks built in such a way would yield short and elegant formulation (Dagiene, 2007), would be interesting for young people and attractive for prospective

sponsors. Also, such tasks give less advantage to experienced participants because they would not be able to use known algorithms immediately. Analysis of programs written by contestants within conditions of Subsection 2.1 as it was proposed by Verhoeff (2006) and is seen from *Task 5* would yield interesting and unexpected results.

References

Dagiene, V., Skupiene, J. (2007). Contests in programming: quarter century of Lithuanian experience. *Olympiads in Informatics* , **1**, 37–49.

Diks, K., Kubica, M., Stencel, K. (2007). Polish olympiads in informatics: 14 years of experience. *Olympiads in Informatics* , **1**, 50–56.

Kemkes, G., Cormack, G., Munro, I., Vasiga, T. (2007). New task types at the Canadian computing competition. *Olympiads in Informatics* , **1**, 79–89.

Pankov, P., Acedanski, S., Pawlewicz, J. (2005). Polish flag. In: *The 17th International Olympiad in Informatics (IOI'2005). Tasks and Solutions*. Nowy Sacz, 19–23.

Pankov, P.S., Oruskulov, T.R. (2007). Tasks at Kyrgyzstani olympiads in informatics: experience and proposals. *Olympiads in Informatics* , **1**, 131–140.

Pankov, P.S., Oruskulov, T.R., Miroshnichenko, G.G. (2000). *School Olympiads in Informatics (1985–2000 years)*. Bishkek (in Kyrgyz & Russian).

Pankov, P.S., Oruskulov, T.R., Miroshnichenko, G.G. (2003). *Olympiad Tasks in Informatics, Devoted to Kirghiz Statehood, History of Kyrgyzstan and Great Silk Road*. Bishkek (also in Kyrgyz & Russian).

Verhoeff, T. (2006). The IOI is (not) a science olympiad. *Informatics in Education*, **5**(1), 147–159.

Weeks, J.R. (1985). *The Shape of Space*. Marcel Dekker, Inc., New York.

P.S. Pankov (1950), doctor of physical-math. sciences, prof., corr. member of Kyrgyzstani National Academy of Sciences (KR NAS), is the chairman of Jury of Bishkek City OIs since 1985, of Republican OIs since 1987, the leader of Kyrgyzstani teams at IOIs since 2002. Graduated from the Kyrgyz State University in 1969, is a main research worker of Institute of Mathematics of KR NAS, a manager of chair of the International University of Kyrgyzstan.

Olympiads in Informatics, 2008, Vol. 2, 122–130

Manual Grading in an Informatics Contest

Wolfgang POHL
Bundeswettbewerb Informatik
Ahrstr. 45, 53175 Bonn, Germany
e-mail: pohl@bwinf.de

Abstract. Bundeswettbewerb Informatik, the central informatics contest in Germany, from which German IOI participants are chosen, is not an olympiad in informatics (OI) in a strict sense. It has a wider range of tasks than OIs (including tasks without programs), and it uses a manual grading approach with grading schemes. Such a scheme is described for two example tasks, one of them an OI-style task, the other a data modeling task without programs and programming involved. Finally, some thoughts are added on how manual grading and tasks without programs could be applied to IOI.
Key words: informatics contest, grading, manual grading.

1. Black-Box Testing in IOI Competitions

In the annual IOI competitions, tasks are of an algorithmic nature. They require the participant to write a program that is to constitute a solution of the task. For each task, the source code of the corresponding program is submitted. The quality of a submission is determined by checking submitted programs against a defined set of test data. Each test case is determined by input data and output data. A test case is satisfied by a submission if the submitted program outputs the test output data when applied to the test input data. Hence, if a submission achieves full score, it can be said to reproduce the input-output relation given by the test data – no more, no less. The contestant, who aimed at solving the given problem, cannot count on a full score to confirm the solution to be perfect.

There has been criticism of black-box testing in general and the IOI grading approach in particular; see, for instance, Cormack (2006), Forisek (2006), Verhoeff (2006). It can surely be said that quality and choice of test data influence scores dramatically. There may be a flaw in submissions that will remain undetected because there are no suitable test cases, and positive properties of submissions may not be rewarded. In addition, the black-box testing approach regularly leads to severe punishment of a submission that implements correct ideas but shows a slight implementation mistake.

Recently approaches were suggested and tried to improve that situation. For instance, test case bundles were introduced, where each bundle should be designed to cover a specific desired property of solutions. Thus, test case design ought to become more focused on qualitative assessment of a submission, in contrast to the more quantitative focus of mere efficiency testing with test cases of different size.

2. Manual Grading in Bundeswettbewerb Informatik

In Bundeswettbewerb Informatik (short: BWINF, Engl.: Federal Contest in Computer Science), a manual grading process is used, which focuses on qualitative assessment of submissions. This contest was described in (Pohl, 2007); by using some of the dimensions for characterising contests suggested by Pohl (2006), it can be summarised as a task-based contest with homework rounds, mixed submissions of texts and programs (source code plus executable), and manual grading.

In the first two rounds of that contest, a submission consists of the following parts:

1. A required part of any submission to a task is a written description of the solution approach.
2. The majority of the tasks also requires the submission of a program; in this case the written part should also explain how the solution approach was implemented, typically by short descriptions of the most important program components.
3. Contestants are demanded to demonstrate the functionality of their submission with examples. Often, task formulations contain a set of required examples, but contestants are always asked to invent and demonstrate their own examples or test cases.
4. This is complemented by printouts of source code, which may be inspected by judges if the other parts of the submission leave doubt about how to grade the submission, or if they want to understand the reason for a mistake, etc.

Grading of BWINF submissions is done on a grading weekend, when all jury members meet. The group event allows the jury members to immediately clarify possible open issues with the jury chairman. Submissions are graded individually and sequentially by two jury members. The grading itself is organised as follows: For each task, a set of grading criteria (a grading scheme) is developed by the task committee. This set may be refined after looking into selected submissions; real submissions may often contain unexpected flaws or, rarely, unexpected solution approaches.

BWINF uses a “negative grading” scheme: Grading criteria are formulated to discover errors and weaknesses of submissions. In the first round, for instance, judges start with a score of 5 points for each task. For each grading criterion that is met, 1 or sometimes 2 points are subtracted. The overall score must not be negative; hence, task scores range from 0 to 5 points. Participants will be informed about grading results with respect to the criteria: They receive an individual score sheet that lists all grading criteria and states which of these were met by their submission and, hence, led to score deductions. In addition, they receive a text that explains solution approaches and the meaning of the grading criteria.

3. Example Tasks

In this section, two example tasks will be described, each from a first round of BWINF. The first could have been an olympiad task as well, it presents a typical algorithmic problem. The second task is of a completely different style; it does not require a program, but asks for information and process models.

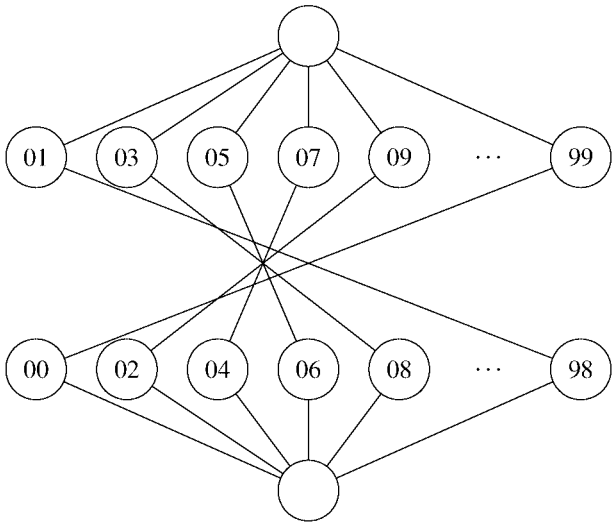


Fig. 1. A partial view of the constant size flow graph (102 nodes, 350 edges), with nodes for cent values only.

3.1. *Example 1: Prämienjagd (Bonus Hunt)*

3.1.1. *Problem*

This task was given in the first round of the 26th BWINF 2007/2008. It could have been an olympiad task as well, since it presents a typical algorithmic problem. The task can be summarised as follows:

- Imagine yourself in a supermarket, at the cashier. You have to put your goods on the belt, one after the other. The supermarket will grant you a bonus for every neighbouring pair of goods the prices of which sum up to an amount with a decimal part of 11, 33, 55, 77, or 99 cent. Each good can contribute to only one pair. Given a list of prices, write a program that computes an order of the prices so that the number of bonuses generated by matching pairs is maximal. It should output the pairs.
- *Example*: For the price list [1.99, 4.13, 6.64, 8.98, 9.91, 1.99], two matching pairs can be found: (4.13, 8.98) and (6.64, 9.91).
- Test your program with the input data given on the BWINF home page and with your own meaningful test data.

3.1.2. *Solution Approaches*

There are many solution approaches that find the best order and hence a maximum set of pairs. But for larger inputs, only few solutions worked. The given test data contained up to 500,000 entries. However, it was stated on the home-page that, in order to achieve full score on this task, it was not necessary to submit a perfect solution which would work for even the biggest cases.

First, it can be observed that a price with an even cent value can only be combined with a price with an odd cent value – and vice versa. So one solution would be to compute a maximum matching in the bipartite graph of prices with odd and even cent values, with edges between prices that can be paired. But the crucial observation is that only cent values need to be considered. Then, a flow graph can be constructed with nodes for all cent values, edges between possible pairing values (like 33 and 44, since they add up to 77), and source and target nodes, with edges between source and even values, and between odd values and target, respectively (see Fig. 1). The number of prices with some specific cent value determines the capacity of the edge between the price node and the target or source node. Capacities of edges between price nodes can be infinitely high. To such a graph, a network flow algorithm can be applied to compute the maximum number of pairs, and this computation takes a time that is independent from the size of input data. In the end, prices can be paired according to the computed flow.

Interestingly, the problem could be solved with a greedy algorithm, too. Also in the case of applying a greedy algorithm, constant run time can be achieved by considering cent values only.

3.1.3. *Grading Scheme*

For grading this task, judges were given a grading scheme with a list of criteria and the corresponding score malus (or bonus, in one case). They are listed in Table 1, with explanations given in italics if necessary. The table also states, how many percent of the 292 submissions to this task met each individual criterion. It is interesting to see that, also in this case of manual grading, test cases play a crucial role. Both the response to the given test cases and self-invented test cases were found to be lacking in about half of submissions. Furthermore, a statement about performance was missing in 37 % of submissions. It looks like many contestants did not find it necessary to investigate into the complexity of their solution, neither by theoretical argument nor by running their solution on given or self-invented test cases – or had the idea that such an investigation might discover the suboptimalities or errors of their solution.

3.2. *Example 2: Supermarkt* (*Supermarket*)

3.2.1. *Problem*

This task was given in the first round of the 25th BWINF. Its story refers to a supermarket setting, too – but be assured that, in general, BWINF task settings are taken from a wide range. The task can be summarised as follows:

> In a food market, at the cash-point price labels are scanned. Non-packed goods like fruit and vegetables will be put on the scales, and the clerk will manually input a product number. The cashier system should cover the following business cases:
>
> - output a receipt;
> - output a list of goods where the amount in store is lower than a given threshold;

Table 1
Grading Scheme for BWINF-task “Prämienjagd”

Criterion	Malus	Applied to
submission does not discover that pairings can be limited *not all possible combinations of prices need to be considered; odd-cent-prices can be paired with even-cent-prices only, and better: a price can be paired with another one only if their sum has one of the legal cent values.*	−1	19%
program too inefficient for other reasons *Even if the approach does not consider too many price combinations, it may still use to much memory, or fail to solve larger cases for other reasons.*	−1	19%
submission does not make a statement about performance *Since it was obvious that performance was a crucial issue in this task, participants should make at least a rough assessment of the (time) performance of their solution.*	−1	37%
incorrect results *The submitted program is expected to produce correct results. This basic criterion and its score malus should only be applied when incorrect results could not be attributed to other criteria.*	−2	31%
program does not output the pairs themselves	−1	2%
submission does not contain outputs to the published test cases	−1	46%
self-invented test cases are missing or are not meaningful	−1	51%
problem reduced to looking at cent amounts only *A very extraordinary case in a BWINF grading: Submissions that discovered and presented the crucial idea that led to an optimal performance were rewarded with a score bonus.*	+1	30%

- output the top-seller list of the month, with goods sorted by product groups;
- output labels with addresses of those clients who used their bonus card and bought a significant amount of wine recently. The labels shall be used for a promotional letter.

What data does the system use and how should it be organised? How can (on that basis) the output jobs be done? What do you think about the last business case?

That is, this task did not ask for a program. A submission was required to describe a data model as well as processes based on that data model. In addition, it is a general BWINF requirement that each submission should present a number of examples sufficient to illustrate and explain their ideas.

3.2.2. *Solution Approaches*

The starting point for a data model is the product number. In the task formulation, it is only mentioned for non-packed goods, but also packed goods should have such a number. This number is the key to all further information about the products of the supermarket. Then, the sub-tasks can be solved as follows:

receipt For each product number, we need to know a name and a price. The price is per pack or per kg (for non-packed goods). For non-packed goods, the price on the receipt is computed from the price per kg and the weighing result.

shortage list For each product number, we need to know the current amount and the minimum amount in store. To make the list informative, we would also like to know whether the product is packed or non-packed; then the shortage list may contain "pack" or "kg", resp. The information needed to update the current amount can be obtained directly from the cashier.

top-seller list This business case requires product groups, so that product numbers need to be related to product groups. Furthermore, every day for each product number the amount sold on that day is stored. That is sufficient to compute the sold amount per month (and might help in cases where you would like to produce top-seller lists per week-day etc.). From this information and the relationship between product number and product group, the top-seller list can be generated.

promotion addresses In this case, individual client data are needed. Similar to the product number, a client number is introduced. For clients with bonus card, address data are known and related to the client number. Moreover, for each client and each product, we store how much the client bought of that product, e.g. since the last promotional letter concerning that product (there may be smarter solutions, but this works for our case). For every product group, there is a minimum amount a client should have bought in order to receive a promotional letter.

An assessment of the last business case should not only consider economical aspects, but also take a critical position concerning privacy aspects. Of course, promotional letters must only be sent upon clients' consent.

3.2.3. *Grading Criteria*

The grading scheme for this task with its list of criteria is given in Table 2. Again, the table also states, how many percent of the 218 submissions to this task met each individual criterion. Interestingly, the more formal aspects of the task (the data model and its description) appeared to be less problematic than the assessment of the business case or the (very basic) difference between packed and non-packed goods. The last grading criterion helped to detect the lacking interest or awareness of the contestants concerning privacy issues.

Table 2
Grading Scheme for BWINF-task “Supermarkt”

Criterion	Malus	Applied to
no difference between packed and non-packed goods *The data model does not make a difference between packed and non-packed goods. That is a mistake, since non-packed goods are treated differently in many cases.*	−1	35%
lacking data model description *The model should be described using some (semi-)formal notation. The description must identify the key data, how other data can be accessed using key values, and how separate sets of information (products, product groups, clients) are linked to each other. Depending on how severely a submission misses these requirements, judges may subtract 1 or 2 points.*	−1 / −2	21%
lacking data model *The data model does not allow to produce (some of the) outputs that were specified in the business cases. Depending on how severely a submission misses this requirement, judges may subtract 1 or 2 points.*	−1 / −2	28%
no description of output procedures *Not only product and client data, but also procedures operating on that data are needed to produce the required outputs. Those procedures must be explicitly described, too.*	−1	6%
lacking assessment of last business case *The assessment of the last business is not acceptable if it does not consider privacy aspects or does not require clients' consent.*	−1	39%

4. New Ideas for IOI?

4.1. *Manual Grading*

BWINF experience shows that manual grading can be applied successfully in an informatics contest. The requirement to create a list of grading criteria (whether negative, positive, or both) forces task committees to make their reasoning about the problem and about the quality of a submission explicit. Such a grading scheme also allows for feedback to the participants. And it can be applied to a wide range of tasks, for which black-box testing is impossible.

However, the BWINF approach cannot easily be applied at IOI, because it heavily relies on the participants explaining their solutions in natural language, and on jury members being able to read and understand that solution. But without a suitable mechanism for (partial) translation of submissions (at least of source code), black-box testing remains the only choice. Manual grading is possible in international contests, like the example of the International Mathematics Olympiad shows. In an international contest, a grading process involves translation work of delegation leaders or even grading work of delegation leaders and is prone to be biased by their interference. For IOI, it would be important to design a manual grading process which would avoid such bias. For instance, delegation

leaders could be asked to translate only, not to grade. The grading then could be done on the basis of the translation, of-course double-blind, by leaders of other delegations.

Very interesting suggestions along this line were made by Verhoeff (2006). Verhoeff's central proposal is to introduce a "thoroughly prepared and motivated grading scheme, supported by measurements". This would follow the example of BWINF, and I fully support Verhoeff's argument.

4.2. *Tasks without Programs*

Here, we are speaking of tasks that would not involve any computer program. This is different from tasks without programming; e.g. a task that would require contestants to test and detect the flaws of a given program with their own test cases, would be a task without programming, but not a task without programs.

For tasks without programs, the formats used in answers to the task's problem(s) strongly determines how the task can be handled and submissions can be evaluated. If the set of possible answers is clearly defined, or the format of a correct answer can be automatically detected, automatic grading is possible. In all cases where the answer set cannot be clearly defined (like with natural language answers), manual grading is required, and all above-mentioned problems of manual grading apply.

Furthermore, in tasks without programming, the internationally understood "linguae francae" of Informatics, the programming languages, disappear. For tasks that involve proofs concerning properties of more or less mathematical constructs (like graphs), the usual mathematical notations could be used. For tasks that involve data models (like our example above), UML or ER-diagrams might help – but should IOI require knowledge of such notations from its contestants?

My personal opinion is that tasks without programs are possible within IOI. However, the BWINF task above is not a perfect example. In particular, its grading criteria are fairly vague and offer jury members too much freedom in grading.

References

Cormack, G. (2006). Random factors in IOI 2005 test case scoring. *Informatics in Education*, **5**(1), 5–14.

Forišek, M. (2006). On the suitability of programming tasks for automated evaluation. *Informatics in Education*, **5**(1), 63–75.

Pohl, W. (2006). Computer science contests for secondary school students: Approaches to classification. *Informatics in Education*, **5**(1), 125–132.

Pohl, W. (2007). Computer science contests in Germany. *Olympiads in Informatics*, **1**, 141–148.

Verhoeff, T. (2006). The IOI is (not) a science olympiad. *Informatics in Education*, **5**(1), 147–159.

W. Pohl was educated in Computer Science, and received a PhD in 1997 from the University of Essen, Germany. For many years, he investigated the use of Artificial Intelligence techniques for the improvement of interaction between humans and machines. In 1999, he changed position and perspective by becoming Executive Director of the German Federal Contest in Computer Science. Among his responsibilities is to coach the German IOI team and lead the German IOI delegation. Now, his interest lies in improving Computer Science contests, establishing new ones, and work on diverse other projects, everything in order to popularise Computer Science among youth. Hence, he co-ordinates the German participation in the international contest “Bebras”. From 2003 to 2006, he was elected member of the IOI International Committee, and briefly held the position of Executive Director of IOI in 2006.

Olympiads in Informatics, 2008, Vol. 2, 131–148

Competitive Learning in Informatics: The UVa Online Judge Experience

Miguel A. REVILLA
Applied Mathematics Department, University of Valladolid
Prado de la Magdalena s/n, 47011-Valladolid, Spain
e-mail: revilla@mac.cie.uva.es

Shahriar MANZOOR
Computer Science and Engineering Department, Southeast University
24, Kemal Ataturk Avenue, Dhaka, Bangladesh
e-mail: shahriar_manzoor@yahoo.com

Rujia LIU
Department of Computer Science and Technology, Tsinghua University
Qinghua Yuan, Haidian District, 100084 Beijing, China
e-mail: rujia.liu@gmail.com

Abstract. The UVa Online Judge is probably the oldest and one of the most recognized programming contest training sites for ICPC format contests. It is an automatic judging system where anyone from around the world (regardless of being a contestant or not) can submit his solution to the archived problems to check its correctness and improve his programming skill in the process. Although the judge was initially developed to be used as a trainer site for potential competitors in the international programming contests (mainly ACM ICPC), we have observed that it is a very good tool for self-study. In the present paper some facts from the history of the site are given. Then the paper focus to the nature of self-competitive learning by analyzing the more frequent response sequences to the users from the judge along these 10 years. And by doing so we identify the main differences between the behaviors of the users when they are just training and when they are competing.

Key words: competitive learning, programming contests, online judge, informatics.

1. Introduction

Programming contests are probably the fastest expanding co-curricular activity related to computer science. The main reason could be that the new Technologies of Information and Communication (TIC) allow us to arrange all kind of interactive activities without too much infrastructure. Of course, the educational processes are an ideal target to use these tools as they offer multiple options to the teachers as well as to the students. It seems evident that one of the favourite topics to focus the modern e-learning systems must be informatics, as it is the base of most of the involved and developing tasks. The fact is that the programming lovers, whether they are secondary, high-school, or university students have a lot of choices to attend to programming contests. For example a university

student can participate in ACM ICPC, the national contests of his own country, local programming contest of his university or programming contests arranged by TopCoder and different online judges like UVa, SPOJ, etc. Moreover, it is an activity that can provide a method for attracting interest in computer science, as it is accessible to beginning students.

It's clear that a programming contest is, by its own definition, a competitive activity, where there are winners and others (not really losers, in general). Usually it's an additional and also co-curricular activity and in that sense they can be seen as a good model of competitive learning. Moreover, many of the programming contests are team competitions and they involve a lot of collaborative work to prepare them. In fact, the training process involves several interesting learning strategies that have nothing to do with the real competition, but with systematic pedagogical methods, which can be very positive for the student's formation and maybe neutralize the negative effects that many people impute to any kind of competitive learning activity.

There exist many online judges on the internet that can play a very important role here. An online judge is in general a server, which contains descriptions of problems from different contests, as well as data sets to judge whether a particular solution solves any of these problems. A user from anywhere in the world can register himself (or herself) with an online judge for free and solve as many problems as he likes. He can send as many solutions as he want till receiving satisfactory information, not only about the verdict, but also about the time that the code takes to run after improving the program and/or the algorithm used to solve the selected challenge. One of the main distinctive trait of the online judges is that they allow the users this self-competitive behaviour to learn informatics, not only algorithms but also programming.

2. A Brief Story of the History of the UVa Online Judge

First of all, let's remember here the name of the person mainly responsible for the existence of the University of Valladolid (UVa) Online Judge: Ciriaco García de Celis. He was a student of informatics when in November of 1995 the first version of the judge started working a few hours before the first local qualifying contest to select a team of the UVa for going to compete in the ACM-ICPC South Western European Regional Contest (SWERC). For more than eight years he was the wizard inside the judge. He, worked almost alone, designed and implemented the kernel of the judge and he also maintained the system as well as the successive migrations from one computer to another, from one version to the other. But maybe the harder work was to fight and win against the many hackers we have had as normal and habitual users. That initial version (written using Unix standard **sh** scripts) was partially rewritten in order to add some improvements to support a 24-hours judging system, capable of working without the presence of a system operator. For example, an automatic system needs to be able to detect and skip e-mail loops.

However, Unix scripts were not powerful enough to support a true reliable judging system. For example, it was not possible (at least under Linux) to limit the memory used

by a submitted program when being executed. And the judge architecture was not designed to generate events reporting its status (external utilities showed the internal judge state by polling it periodically).

For this reason, new judge software was developed. This new judge was able to work as a 24-hour Online Judge and a programming contest judge. However, it still missed many components required by a general conception for Contest Judge, as then it matched almost completely with the ICPC (the granddaddy of the programming competitions, as far as we know) model both for the problems style and for the contests dynamics and rules. However, Fig. 1 shows us that a lot of services were already included in the planning in order to provide to the users a tool to use for learning, while they train for the contests.

After checking the system for a short period of time, when some students in Algorithmics at the University of Valladolid were the only allowed users, the UVa Online Judge started its open period in 1997 with a hundred problems and a little promotion, on a day of April 1997-04-15 14:31:48 (UTC) is the date of the first submission (it was made by Ciriaco and was successful, of course). But, any person in the world can get access to it via the internet for free. The site then began to be more and more known, as programming contests were becoming more and more popular. For about two years the judge didn't really change, except minor bug fixing, or adding three or four new volumes (a volume is a set of a hundred problems) taken from different web sites, mainly corresponding to ACM-ICPC regional and final contests.

In November 1999 the University of Valladolid hosted the official SWERC and then we realized that we must work on the environment of the judge. It's the first time we were aware about a bunch of services that we still needed to develop before having a site able to support the quickly increasing number of users and submissions and to host online contests. A group of voluntary students collaborated to plan out and then to implement a lot of new services: an electronic board, a friendly interface, a detailed set of statistics,

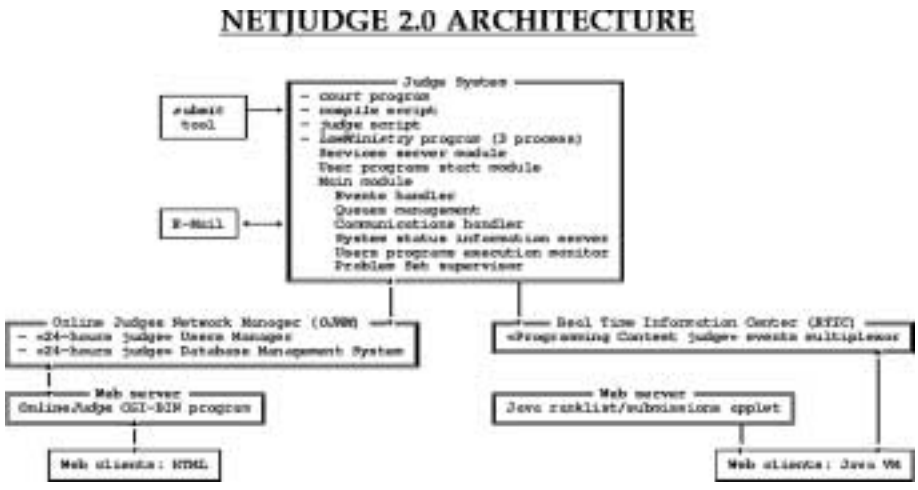


Fig. 1. Diagram of the different Online Judge modules. Each module can be located in a different computer, and are communicated by TCP connections.

rankings, etc. That team of people was the base of the users' community of the UVa judge; they started the big task to analyze every possibility we had to transform our practice and test site in a real project with plenty of objectives. Their enthusiasm and also many of their ideas are still present today.

Of course, the most important and urgent work to do was to implement a robust contest system to be used by the real time contest. After the contest some of the members of that team continued working to consolidate the tasks already done and to develop the main goal we had talked about: to have our own regular online contests. As UVa had to arrange the SWERC in November 2000 also, we decided to do it as fast as possible and by the month of July, exactly on 2000-07-14 at 14:00:00 (UTC) the first test contest was open to all the world and 5 hours later finished successfully.

There are many other important dates and facts in the life of the UVa Online Judge. The main evolutions were due to Fernando Nájera that included in 2002 the use of a SQL real database to keep all the information ready for the users in real time and the PHP tools to manage the interface easily and, of course, to Carlos Casas who is quite well known for all the users as he is still a very active member of the UVa site. A special mention is deserved for our online contests. Under the management of Shahriar Manzoor have increased the level of the problems on our site in quantity but especially in quality. In fact, as of today the set of problems specifically written for our judge are probably the main asset we have and surely our main pride. Many other people arranged some high quality online contests from the early stages such as Professor Gordon Cormack of University of Waterloo, Rujia Liu of Tsinghua University and Md. Kamruzzaman of BUET (now he is at UCSD).

Fig. 2 shows the classification by languages of the 5899124 programs submitted by 63351 users from about 180 different countries till the date 2007-09-06, 17:19:45 (UTC),

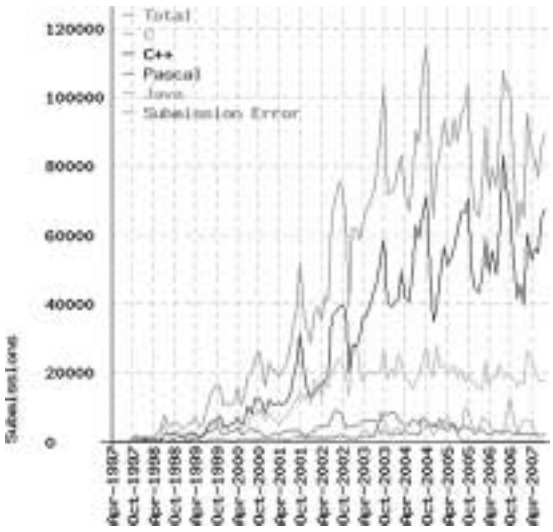


Fig. 2. Robot Judge submissions by Programming Language throught September 2007.

when after 170 online contests (more or less a half of them arranged by our own team) the old judge was definitely stopped (UVa Online Judge). Two hours later a completely new robot restarted working at a new server at the Baylor University, the headquarters of the ACM-ICPC contest. It has been developed and implemented by Miguel Revilla Jr. and it incorporates the whole history of these ten years, all the amazing information about this extraordinary experience. So the UVa Online Judge continues its new journey at the CLI website (CLI).

3. The Analysis of Statistics

The immense amount of data from users in different parts of the world provides the opportunity of making lots of analysis and it opens up the possibility of getting some important results. These results may enable us to find out what aspects of online judging or the programming contests need to be changed to make it more meaningful, how practice and experience improves the performance of a contestant and many other interesting issues. It will also create new openings on which we can continue our study in future. For example, it will help us to identify the geographic locations where it's almost impossible to compete in programming contests and then we can take initiatives for those regions, and try to extend our internet community to those parts of the world.

Although there is no academic or social bondage between the members of this online judge community, they are still training in our judge, testing (individually or by groups) their programming skills, self-competing or arranging contest in our server, discussing on our board and participating in our Online Contests just to be better programmers and to learn more complex and new topics. Many of them may not even see each other but still they are good friends, helping each other out in many occasions. An extreme example may be that the three authors of this paper have been collaborating since year 2001, but Miguel and Shahriar first met in 2005 and Shahriar and Rujia also first met in 2005. And the occasion was the 2005 ACM ICPC World Finals that took place in Shanghai. Miguel and Shahriar still meet only annually, while they are still waiting for their second opportunity to meet Rujia Liu.

The main conclusion we have made is that we are managing a tool with a very large potential not only for training but also for teaching informatics, a merge of competitive and cooperative learning, let's say a kind of collaborative competition. Moreover, it can be successful where the conventional systems often fail: to make students curious, to make them do non-academic works that often seems more interesting or to enhance their creativity. As everybody can access to these online judges and start practicing with easy and funny problems, it could be a good way to attract the newcomers (for example, secondary school students) to the world of programming and to computer sciences in general. It is very important in a world where many of us including some software giants have a perception that the young generation is losing interest in computer science.

In subsequent steps, from the perspective of algorithm design, the programming contest is a treasure trove. There appear to be numerous ways to solve the same problem. But

also for software reliability engineers this is the case: there are even more ways to not solve the problem. Most authors first submission is incorrect. They take some trials to (in most cases) finally arrive at the correct solution.

Suppose one submits a program in a contest and gets accepted, another contestant submits a program he gets wrong answer and then he submits again he gets accepted, another contestant submits a program six times and every time he gets wrong answer. Which one of these three events is more likely to happen in a programming contest? To find it out we analyzed all the submissions of UVa site and found out which are the most common response sequence for a contest. We actually took a method like digraph, trigraph analysis of a text. First we tried to analyze which submission response is most common for a problem. And the most frequent response sequences are given in the Tables 1 and 2. These tables are based on the analysis of the first 4 millions submissions to the UVa till October 2005, but the amount is enough to be statistically significant.

In fact, the partial analysis we have done later show that the order of popularity is more or less the same although the number of submissions now is near of seven millions. Only the number of PE (Presentation Error) verdicts has decreased proportionally as we have purged the data sets to fix some trivial mistakes.

Table 1
A table for most popular response sequence

Monograph	**AC**	**WA**	**CE**	**TL**	**PE**
Frequency	465516	214187	104952	76806	73526
Digraph	**WA|WA**	**WA|AC**	**AC|AC**	**CE|CE**	**TL|TL**
Frequency	164521	71018	49743	39732	30830
Trigraph	**WA|WA|WA**	**WA|WA|AC**	**CE|CE|CE**	**TL|TL|TL**	**AC|AC|AC**
Frequency	92545	32765	20049	14436	14203
Tetragraph	**WA|WA|WA|WA**	**WA|WA|WA|AC**	**CE|CE|CE|CE**	**RE|RE|RE|RE**	**TL|TL|TL|TL**
Frequency	55504	16518	11566	7947	7474

Table 2
A table for most popular responses ending with an AC

Monograph	**AC**				
Frequency	465516				
Digraph	**WA|AC**	**CE|AC**	**TL|AC**	**PE|AC**	**RE|AC**
Frequency	71018	18099	10612	9213	8205
Trigraph	**WA|WA|AC**	**CE|CE|AC**	**TL|TL|AC**	**CE|WA|AC**	**RE|RE|AC**
Frequency	32765	4685	3540	3511	2620
Tetragraph	**WA|WA|WA|AC**	**CE|CE|CE|AC**	**CE|WA|WA|AC**	**TL|TL|TL|AC**	**RE|RE|RE|AC**
Frequency	16518	1750	1636	1340	1158

Many comments can be made based on these tables. But some things are obvious:

a) When individual contestants make a particular type of mistakes for a problem they tend to make the same mistake again, which encourage the group to try working together for the contests. Let's say one more time, competitive and cooperative learning in informatics are not opposite but complementary.

b) We can say that if someone gets five consecutive wrong answers then in the next submission he is four times more likely to get a wrong answer than an accepted verdict. That means that after four or five errors, the best is to analyse carefully what happen as, probably, the mistake is not trivial.

c) It is very important the influence of the kind of mistakes in these sequences. That is, mainly, because some responses of the judge give us information about the error and others tell us nothing at all. This is very important in order to improve our system judge to become a real learning tool, by adding new features.

As of now, we can divide the judge responses into two types: Informed Response and Uninformed Response. These divisions will help us to propose a combined system to bring IOI and ICPC closer later on. Informed responses are the responses that allow the contestants to know whether their program logic is correct or not correct: AC (for ACcepted), PE (for Presentation Error) and WA (for Wrong Answer) are such types of responses. The other three TL (for Time Limit exceeded), RE (for Run-time Error) and CE (for Compilation Error) are uninformed responses, because it is not known what would have happened if the program, in case it starts, was allowed to run longer or not crashed. And, of course, unless we give one test case per file as input it would be impossible to judge the 'degree of correctness' of the submissions that get TL or RE in the present ICPC system. Including new models of judging, the grading system used at IOI is the main project we are working on.

Similar statistics could be done about the submissions to the online contests arranged on our server, but after 135 contests over five years, we saw that the great figures were very similar. But there are other very interesting details to analyze in the contests. In fact, whether it is an Online Contest or in the 24 Hour Online Judge the acceptance rate is around 30%. But this acceptance rate is not so bad when we consider the statistics of accepted problems only. For example suppose there are eight problems in a contest A, B, C, D, E, F, G and H. One team solves problem A, B and G and attempts problem C and D. In this section we will not consider the judge responses for problem C and D for that team.

Table 3 shows the judge response statistics, but considering only the initial submissions from a team for which they finally got an accepted verdict. It is found that its probability of getting Accepted in the first submission is 44.16%. The percentage of informed responses is 80.89% and uninformed responses is 18.14%. But more important is the fact that percentage of informed errors is 36.73% and of uninformed errors is 18.14%. So their ratio is roughly 2:1.

Table 3

Judge response statistics for accepted problems/team only

Verdict	Percentage	Informed vs uninformed response	Informed vs uninformed errors
AC	44.16	80.89%	Not considered
PE	3.08		36.73%
WA	33.65		
TL	8.03	18.14%	18.14%
RE	3.72		
CE	6.39		
Others	0.97	Not considered	Not considered

4. How Does Practice Change Things?

The most important part of the analysis of the millions of programs we have received at the UVa Online Judge is to know our users and try to learn from them as many details as we need to improve our services. And not only about the demographic distribution, as we told above, but also about their evolution along the time. We certainly hope that almost all of them have improved their skills in programming and algorithms, but it's interesting to quantify this fact. And, as far as it's possible, try to get an idea about the different patterns of the verdicts in function of a user is a newcomer or an expert in the use of the judge. Of course, there are many details we can't be sure about most of these users, we don't know if they are individuals, a regular team or a variable group. It would be a tuning process, almost impossible to do, of selection of submissions in order to get more categorical conclusions.

Table 4 shows the error rate of people with different experience. The first column on the left actually describes the experience of the user that is being considered. Experience means the number of problem he has solved in the judge, it does not consider anything

Table 4

Based on all problems

Solve Range	**AC**	**PE**	**WA**	**TL**	**RE**	**CE**
0–49	23.76	4.93	36.13	8.36	8.01	12.24
50–99	33.81	5.57	34.18	7.33	7.54	6.35
100–149	35.08	6.41	33.59	6.70	7.50	5.62
150–199	37.02	4.95	33.01	7.07	6.90	5.70
200–249	37.74	5.01	32.85	7.11	6.83	5.31
250–299	39.90	4.60	32.41	6.89	6.16	5.17
300–349	40.86	4.08	32.56	7.34	5.87	4.63
350–399	42.03	4.30	32.21	6.51	5.97	4.49
400–449	41.96	4.03	32.16	6.86	6.37	4.05
450–499	41.82	3.65	31.50	7.10	5.98	4.68
500+	42.36	3.53	31.83	8.06	5.42	4.06

about the time he is associated with the judge. Each of the next six columns actually shows the rates of six major judge responses in a programming contest. For example the third row of the table says that the contestants who have solved more than 50 and less than 100 different problems has 33.81% acceptance rate, the rate for wrong answer is 34.18 and so on.

Table 4 can have some interpretation troubles, because their can be some confusions: as people solve more problems they have less easy problems to solve (assuming that people tend to solve easy problems first). When someone has already solved 400 problems he has no more easy problems to solve, so his acceptance rate can go down a little. But as he is more experienced the acceptance rate does not go down but remains similar. In Table 5 and Table 6 we have put the same results but this time separated by the 'experimental' difficulty of the problems based on their low or high acceptance rate.

Table 5

Based on problems with low (less than 25%) acceptance rate

Solve Range	AC	PE	WA	TL	RE	CE
0–49	11.09	1.71	41.73	14.62	12.43	11.52
50–99	17.45	2.15	42.48	13.25	12.00	6.88
100–149	18.98	2.69	42.13	11.85	12.44	6.16
150–199	20.29	2.37	41.61	12.47	10.79	6.23
200–249	20.86	2.46	42.17	12.78	10.15	5.77
250–299	23.09	2.37	41.91	12.43	9.19	5.16
300–349	24.24	1.94	42.17	12.46	8.92	5.01
350–399	24.15	2.54	42.99	11.30	9.44	4.92
400–449	25.61	2.33	41.32	11.42	9.51	4.47
450–499	27.21	2.09	38.57	12.36	8.89	5.38
500+	27.20	1.65	41.04	13.53	7.20	4.24

Table 6

Based on problems with high (more than 50%) acceptance rate

Solve Range	AC	PE	WA	TL	RE	CE
0–49	40.81	6.67	26.37	4.03	4.00	11.79
50–99	53.86	7.47	21.77	2.99	3.37	5.94
100–149	53.97	9.18	21.18	2.51	3.31	5.38
150–199	58.33	7.21	18.66	2.57	3.10	5.33
200–249	59.67	6.66	19.25	2.39	3.02	4.78
250–299	62.30	6.24	18.25	2.29	2.39	4.51
300–349	64.56	6.40	16.42	2.12	2.65	3.92
350–399	64.44	5.01	17.48	2.12	2.44	3.91
400–449	65.17	6.26	17.74	2.23	2.13	2.84
450–499	63.15	5.19	17.50	2.10	2.72	4.68
500+	67.73	4.31	15.46	2.22	2.33	3.97

Tables 5 and 6 indicate that with practice the acceptance rate increases a lot, mainly for the problems with high acceptance rate, and also compilation errors decreases a lot and quickly for all the three categories. But, surprisingly, wrong answer and TL percentage does not change that much, even for the group of very expert users. So does this indicate no matter how experienced you are you can always get wrong answer? Of course, every person will always have a harder problem to solve, a new programming challenge to face in order to continuously increase his skills in informatics.

Usually, by 'programming ability' people means coding, debugging and testing. Though, these individual abilities greatly affect cooperative works too (it's easy to suppose that many of our users work in group, being a team or not). It's better for the team members to use the same language and similar coding conventions. In such way, if one cannot find his bug then one can ask another person to read her/his code. Though programming is the very first skill, it needs improving all the time. For example, coding complex algorithm can only be trained after studying these algorithms.

Most people got started by solving easy problems. Here, by easy problems, we mean the problems in which you only need to do what you're asked to do, i.e. a direct implementation of the problem description. For example, do some statistics, string processing or simulation. These problems mainly require some coding ability but not any sophisticated algorithm, deeper mathematics or logical insights. When getting started, practice is much more important than theory. (Practice, practice and practice).

Everyone is encouraged to program as much as he can, as long as enthusiasm is perfectly kept. But there is one thing you need to know first: ICPC, IOI and most of the existing contests concentrate on problem solving and apparently the enjoy of programming comes from solving easy problems in which you only need to do what you are asked to do, i.e. a direct implementation of the problem description. But keep the limits. Trying to solve more problems is good, but the quantity is not the most important thing. When you've managed to solve 50 easier problems somewhere, it's better to seek for more challenges. In other word it is better to solve many problems of various kinds and difficulty. In real contests and online judges, there are a large number of problems that require a few lines of code but more maths and algorithmic thought. So when you are challenged with problems that are more interesting and difficult, you will find it necessary to think about something serious: becoming a great contestant.

5. The EduJudge European Project

The users of Online-Judge are demanding a greater pedagogic character for this tool (at least one request per week is sent to the UVA On-line Judge creator via email, also some requests are available in the forum `http://online-judge.uva.es/board/`). For example, teachers would like to use it as one more activity for their official courses. This requires the possibility of managing courses and students and an extension of the current functionalities of the Judge so that it can provide gradual evaluation or different difficulty levels of problems. On the other hand, the set of problems is continuously being

incremented but it is necessary to give the problems an adequate and common structure, adding metadata and creating a search engine so that the problems are more accessible for the community of teachers.

It is easy to understand that these sets of achievements are only possible within the frame of a collaborative project involving experts from several countries and different areas of knowledge. This is the origin of the EduJudge project. EduJudge is an innovative system based on ICT that can be incorporated into the learning processes in the mathematical and programming field and is addressed to higher education students and secondary education students. It has been managed and coordinated by CEDETEL (Centre for the Development of Telecommunications in Castilla y León), a non-profit Technology Centre located in Spain. The project has been funded with support from the European Commission into the frame of the Lifelong Learning Programme of the European Union. Other than the University of Valladolid, there are three more partners from different European countries: the University of Porto (Portugal), the KTH Royal Institute of Technology (Stockholm, Sweden) and the Institute of Mathematics and Informatics (Vilnius, Lithuania).

The main goal of the project is to give a greater pedagogic character to the UVA Online Judge, and adapt it to an effective educational environment for higher and secondary education. We want to give the Online Judge a pedagogical character by means of a redesign, improvement of contents and its integration into an e-learning platform. All these will contribute to the development of quality lifelong learning, and also to promote innovation providing new methods of teaching through contests, instead it being focused exclusively on competition. The work packages UVa leads will make it more suitable for its use on a learning environment. The different tasks are:

- Solution quality evaluation: the user will receive a more complete feedback from the system, not only indicating that the problem is solved (or not) but grading the quality of the solutions. This can range from a no significant solution (less than 50% of correctness) to a completely correct solution (100%).
- Generic Judge Engine: the system will support several problem formats, allowing different kinds of learning approaches. By allowing different formats of problems the system is not limited to a right/wrong evaluation method. There can be problems in which the challenge is not only solving a problem, but solving it in the most efficient way. Also there can be cases where a student's solution must 'compete' against another student solution in an interactive way. Having a generic judge engine that can easily be extended to support more problem formats will make this possible.
- Automatic Test case Generation: the automatic generation of test cases will allow different levels of difficulty in the solving of the problems. Having good quality and heavily checked test cases is essential for a good evaluation of a solution. The creation of such cases by hand is a difficult task. The automatic system should be able to generate good test cases based on a given set of rules describing the format of the test case. There can be another interesting situation with automatic generation of test cases. We just give a trivial example here. Suppose a user is

trying to solve "The closest pair Problem" – given n points find the distance of the closest two points. Now the user finds that he is struggling to solve the problem with $n <= 10000$. So using the generator he can generate test cases with smaller values of n and check whether his program works for that. Or we can even propose a WIKI system where users can submit their own generator, and our input tester will check whether the submitted generator generates according to the specified rule before passing the input to other users' solution. In other words if the problem's input statement specification is editable by the user, he can even generate his own test cases with different values of the parameters and actually solve a very different problem that the original problem setter did not intend to solve. Of course all these changes will be within certain limits so that the original author's solution can solve it. All these will help the teachers to create easier problems for their weaker and/or younger students. Also the teacher can specify in which format he wants the test cases to be IOI, ICPC or any other new format. But to implement all these a good number of dedicated people are needed to be involved with it.

6. IOI vs. ICPC. Is the Convergence Possible?

Looking at the tasks mentioned above, it's clear that one of the more important ideas behind the work package to be developed by the University of Valladolid in the frame of EduJudge, is trying to find a meeting point between IOI and ICPC, as far as it's possible. The reason is they are the two most "academic" of the programming contests existing now and in fact there is a continuity from the first to the second, as many of the contestants of the second had their first contact with this activity in the IOI. Probably a more interesting statistics would be how many have actually won a medal in ICPC, without participating in IOI.

From the point of view we are implied, the automated judging, the first problem we have to face is to try and overcome is the issue of grading. It's evident that this is an additional trouble for the problem setters, because the test cases need to be more carefully selected in most of the problems in order to produce a gradual punctuation correlated with the correctness of the code. Even the description of the problems need to be analyzed in detail to allow different sets of inputs that make it reasonable to claim that a program is 50% correct and to prevent the criticisms about from the people that defend the strict binary system of the ICPC: a program that fails in solving an only case is not correct. Certainly any kind of conventional grading system is closer to our competitive learning objective than the 0/1 approach of ICPC.

The IOI is more positive than ICPC because (i) It allows partial marking unlike the 0/1 approach of ICPC, and (ii) It requires the contestants to solve only three problems in five hours which is a lot of time (even though the contest is by individuals). So anyone with a bad start can make up, because there is no penalty on submission time. So the speed of a contestant is not a strong factor. But the ICPC, in spite of its very strict 'either correct or incorrect', still has some very good sides: it gives real time feedback to contestants about the correctness of their solution and also it is not bad to give some credit

Table 7

Judge response statistics based on accepted problems/team only

Subm. Serial	Cumulative Acceptance Percentage	Acceptance Percentage	Cumulative Number of Acceptance	**Subm. Serial**	Cumulative Acceptance Percentage	Acceptance Percentage	Cumulative Number of Acceptance
1	53.622455	53.622455	24358	**11**	98.908090	0.305999	44929
2	72.686846	19.064392	33018	**12**	99.119428	0.211337	45025
3	82.875069	10.188222	37646	**13**	99.317556	0.198129	45115
4	88.920198	6.045129	40392	**14**	99.493671	0.176114	45195
5	92.631811	3.711613	42078	**15**	99.583930	0.090259	45236
6	94.996147	2.364337	43152	**16**	99.667584	0.083654	45274
7	96.398459	1.402312	43789	**17**	99.749037	0.081453	45311
8	97.367089	0.968630	44229	**18**	99.806274	0.057237	45337
9	98.093561	0.726472	44559	**19**	99.856907	0.050633	45360
10	98.602091	0.508531	44790	**20**	99.894331	0.037424	45377

to the contestants for their speed. Moreover, the three member team structure promotes the cooperative learning added to the competitive situation, because it requires an active interaction between them, which results in a positive interdependence.

So to eliminate the short comings of these two major types of contests we need a contest that (a) Gives partial marks to contestants. (b) Gives real time responses to contestants. (c) Possibly informs the contestant which test cases match (only the serial of test case) and which don't. (d) If we don't use separate files for each set of input no information regarding correctness will be available if the submitted program does not run within the time limit (TL) or crashes (RE) for any one of the inputs. In continuation to this discussion a new probable approach will be proposed after we see some interesting statistics related to the UVa Online Judge programming contest.

Every year in the prize giving ceremony the Chief Judge (aka Head Jury) often loves to say how a team failed to solve a problem after submitting it 30 (thirty) times, or another team got a problem accepted in their 20th attempt. These types of things are mentioned because they are rare events in a programming contest. Before proposing a new model of contest, we tested these kinds of events in our Hosting Contest Service. We were afraid they would be significant more frequent as the users play for nothing really important, but to check their competitive level. Our interests were to extrapolate the new ideas about possible new models of contest by simulating what would be the result with our contest. Then we needed to check our online contest with real ones.

The Table 7 shows the statistics on how many submissions are required to get a problem accepted based on the first 135 online contests of Valladolid Site. We can see that in 10 or less submissions almost 98.6% accepted verdicts are found. It means on average in a programming contest only 1.4% of total accepted problems require more than 10 submissions. But, even more important, almost three from each four contestants get an AC verdict on their first or second submission.

It has already been said that an ideal contest model should have partial credits like IOI

and also real time feedback like ICPC. But ICPC allows the contestant to submit problem infinite times. But a proposal of a contest model giving partial credit and infinite time submission is a bit too much because in each submission the contestant has the option to try different kinds of tests and moreover if he is allowed to know which test cases are getting wrong he might use one of his solution to produce output for some test cases and another solution to produce outputs for other cases just depending on the case number. In our study we also found that the ratio of informed and uninformed errors is roughly 2:1. So we can set a new limit that a team will be allowed to make total eight wrong submissions per problem and another four uninformed responses will be allowed. So a team can get 4 RE and 8 WA for a problem but he cannot get 9 WA because maximum 8 informed errors will be allowed. In other words we can say that total 8 errors will be allowed and first four uninformed errors will not be counted in these eight errors. With this new rule the statistics of Table 7 becomes as in Table 8.

As we are allowing 8 errors if the ninth submission is an accepted verdict, it will be granted. However if a team fails to get the problem accepted in these submissions he will be given the highest points that he obtained among these submissions. Now the question comes how can we prevent poorly written solutions to get good scores? – in this model the answer is simple. As we are allowing the contestant to fix his mistakes we don't need to be as lenient as the current IOI, so partial marks will only be given if someone gets more than 60% of the marks, otherwise he will get a zero. Now the question that may come how weak coders will get marks as there is no lenient rule like the classical 50% rule, and the answer is just to give an easy problem to the contestants to solve so that they can get some marks and let the hard ones remain hard. The total number of problems can also be increased (Say five problems in five hours) to include easy and easy medium problems.

The problem with an ideal programming contest model is that it needs to be fair but it also needs to be simple because the same model will be followed in regional (ICPC) and national contests (IOI). Also some of the models are extremely popular so it will take

Table 8

Judge response statistics ignoring first four uninformed responses and allowing maximum eight informed errors

Subm. Serial	Cumulative Acceptance Percentage	Acceptance Percentage	Cumulative Number of Acceptance	**Subm. Serial**	Cumulative Acceptance Percentage	Acceptance Percentage	Cumulative Number of Acceptance
1	63.077600	63.077600	28653	**10**	99.225096	0.323610	45073
2	80.061640	16.984040	36368	**11**	99.392405	0.167309	45149
3	88.453495	8.391855	40180	**12**	99.509081	0.116676	45202
4	93.021464	4.567969	42255	**13**	99.643368	0.134287	45263
5	95.601541	2.580077	43427	**14**	99.720418	0.077050	45298
6	97.076500	1.474959	44097	**15**	99.795267	0.074849	45332
7	97.932856	0.856357	44486	**16**	99.843698	0.048431	45354
8	98.507430	0.574573	44747	**17**	99.876720	0.033021	45369
9	98.901486	0.394056	44926	**18**	99.898734	0.022014	45379

some time to replace them. All online judges are written in the existing rules and it will take some time to change them as well. Many regions and nations are still struggling to adopt the present simple contest models so the new more complex models can be impossible for them to follow. So a new full proof system can first be followed in international level and then in course of time poured into national and regional level.

7. About the Categorization of Tasks

About classification, serious solvers are not interested in doing some classification of the UVa archive. They think that making it public would take away the fun part. Many times the more important task for solving the problem is to decide the type of the designing technique to use. So, some of our previous attempts have failed. Of course, grouping problems into specific categories is very useful, especially for beginners, teachers or for those who want practice on a particular problem type. The users can then try to solve all variety of problems of the same type to master the technique. And, in the frequent case, when a problem can be assigned to several classes it's also very useful to learn new algorithmic concepts behind the technique itself.

However, maintaining such a kind of list is a really hard task, especially when the number of problems is as big as 2500+, and to do it well is very troublesome. First of all we need a prototype controlled list where to classify the problems. Even though there is an almost standard universally accepted list, the experience shows us that the contribution of the users must be managed if we want to prevent a real chaos. The first version of our judge allowed to the users fill a field to write the algorithm they had used in the code, and many of the people didn't use or made an undesirable use of it and that made the field useless. There are too many details to decide before to go on with these helping tools, because it could be negative and confusing if we are not careful enough.

Talking about difficulty level, the problem is even worse as for most of the problems it is very subjective opinion depending of the expertise of the person making the decision. Probably most the people agree about trivial and very hard classes, but the opinions for intermediate levels can be almost impossible to fix. And this is a very important detail for the learning efficiency of the site. A user trying and trying an "easy" problem without getting a positive answer probably lives a traumatic experience in his mind. Then it must be very clear that the difficulty level is a relative concept and a good idea is that the user completes this kind of information with additional data, mainly heuristics consequences of the statistics. In fact, the only published classifications we have done are included in the book that the first author wrote with the Professor Skiena. After arguing for long and working a lot we decided that talking about popularity was better than difficulty and success rate was better than difficulty.

In fact, there are several pages with information about our UVa Online site (and maybe much more we don't know about, as we can't control every thing, of course). For example, one of the most popular pages dedicated to algorithms and programming contest (Pi algorithmist) contains a section specifically dedicated to the UVa Online Judge with several links to some of those pages, as well as an open subsection about categories. The

sites of Felix and Steven Halim are excellent, with a lot of interesting information, but maybe the users like more the site managed by Igor Naverniouk (Igor's UVa tools) where the problems are classified by difficulty level (trivial, easy, medium, hard, very hard, IMPOSSIBLE) and it allows to compare with each other user as well as to get a tip about the following problems to try in function of the past history.

Of course, we have this information and we check it from time to time, but they are not managed by us. Although we contact the responsible staff of these sites, and try to help them to develop their initiatives, they are independent. But all of them have some common characteristics. For example, there is almost general agreement about labeling only a few problems into each of the categories. At this moment there is not an 'official' categorization of the site even though the data base is ready to do it, and we hope that all these features will be included as a built-in feature in the UVa Online Judge with the collaboration of all these persons.

8. Conclusions

Competitive learning in informatics, as we understand it in the present paper (training to participate in programming contests by using online judges and taking part in internet contests) can be an adequate method to learn algorithms and programming, as it is free of the most frequent criticisms that many other methods have. It's true that the final objective is the competition, and probably a hard competition, but there are a lot of constructive outcomes on the way. It is something like climbing mount Everest. One may not be able to reach the top, but the courage, physical ability required to reach even half the height is praise worthy and requires a lot of skill. It doesn't the matter whether the contest is individual or by teams, most of the work to do is self-competitive as well as cooperative.

Depending on the contest the students are preparing for and the actual stage of training they are in, the teachers in charge may promote different kinds of activities, by group or by individual, to prevent as far as possible negative consequences of competitive learning. From this point of view, probably the ICPC and the team competitions in general, are a level under the individual ones, as IOI, as the 'learning team' criteria require that the common work and the individual effort must go together.

Of course, we can't forget that at the end there is only one (person or team) winner of the real contests. It's clear that winning must be the main goal for all the contestants, but the statistical analysis of the millions of submissions to our UVa online judge shows that many times the users, whatever there is behind, try and try the same problem till they get a successful verdict and/or CPU time, by using the informed responses of the judge as well as the electronic board of the site for checking with the other users results. And in the end the self improvement of individual users is the most important outcome of the practice, not the one champion that we get. Many students qualify for the big events of ICPC, IOI and TopCoder but many more students never qualify for a bigger event, but behind this tangible failure, they become better programmers and thinkers, which may in future help them to become something special.

In any case, remember that we are talking about learning informatics for free (we mean here algorithms and programming) as the main step of the process is the training period and it can be scheduled as a really funny work. Let's cite the starting words of the Programming Challenges book (Skiena and Revilla, 2003): "There are many distinct pleasures associated with computer programming (...). The games, puzzles, and challenges of problems from international programming competitions are a great way to experience these pleasures while improving your algorithms and coding skills."

Acknowledgements

The activities described in this article are part of the project "Integrating On-line Judge into effective e-learning". This project has been funded with support from the European Commission. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

CLI. Competitive Learning Institute at the Baylor University of Texas (USA).
`http://icpcres.ecs.baylor.edu/onlinejudge`
Igor's UVa tools.
`http://shygypsy.com/acm/`
Liu, R. (2008). *Training ICPC Teams: A Technical Guide*. CLIS, Banff.
Manzoor, S. (2006). *Analyzing Programming Contest Statistics*. CLIS, San Antonio.
Pi Algorithmist. The Algorithmist is a resource dedicated to anything algorithms.
`http://www.algorithmist.com/index.php/Main_Page`
Skiena, S.S. and Revilla, M.A. (2003). *Programming Challenges. The Programming Contest Training Manual*. Springer-Verlag, New York.
UVa Online Judge. Online Judge and Contest system developed by the University of Valladolid (Spain).
`http://online-judge.uva.es/problemset`

M.A. Revilla is a professor of applied mathematics and algorithms at the University of Valladolid, Spain. He is the official website archivist of the ACM ICPC and creator/maintainer of the primary robot judge and contest-hosting website. He is involved with the ICPC contest for more than ten years, and now is member of the International Steering Committee of the ACM. He received the 2005 Joseph S. DeBlasi Outstanding Contribution Award.

S. Manzoor was born in Chittagong, Bangladesh on 12th August, 1976. He is probably the first person with the concept of arranging monthly ACM ICPC format online contests. He is also first person to arrange ACM ICPC World Finals Warmups with the help of many other persons and these contest have been arranged for consecutive eight years (2001–2008) via UVa Online Judge. He is also a ACM ICPC World Finals Judge for six consecutive years (2003–2008). He is the chairman of Computer Science and Engineering Department of Southeast University, Bangladesh.

R. Liu is a coach of IOI China national training team – a team consisting of 20 students from which the final national team is selected) since 2002. Being a contestant, he participated in the 2001–2002 ACM/ICPC, winning the champion of Shanghai regional contest in 2001, and then a silver medal (the 4th place) in the world finals, Hawaii in 2002. Being a problem setter, he authored over 10 problems for the national Olympiad, winter camp and IOI team selection contests in the past (2002–2006). Currently he's still active in creating problems for online contests in UVa Online Judge and other programming contests.

Olympiads in Informatics, 2008, Vol. 2, 149–162

Early Introduction of Competitive Programming

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
Gambelas, 8005-139 FARO, Portugal
e-mail: pjguerreiro@ualg.pt

Abstract. Those who enjoy programming enjoy programming competitions, either as contestants or as coaches. Often coaches are teachers, who, aiming at better results in the future, would like to have more and more students participating, from earlier stages. Learning all the basic algorithms takes some time, of course; on the other hand, competition environments can be introduced right from the beginning as a pedagogical tool. If handled correctly, this can be very effective in helping to reach the goals of the course and, as a side-effect, in bringing larger crowds of students into the programming competition arena.

Key words: programming contests, computer science education, automatic evaluation systems, competitive programming, introductory programming, IOI, International Olympiads in Informatics.

1. Introduction

The most popular programming competitions are geared to imperative languages and to input-output problems. The automatics judges that handle submissions have been customarily designed for that environment. In our own programming courses, at our universities, we are using one such judge as a learning tool. Students submit their programming assignments to the judge, with the benefit of immediate feedback, more reliable than what could be provided by a teaching assistant. Moreover, the automatic judge publishes a ranking of the students by number of accepted submissions, and this induces a healthy competitive attitude that many students enjoy and that makes them work harder and in a more disciplined way than they would otherwise.

This approach has shown to be very effective in increasing student productivity, measured by the number of programs actually written by the students during the courses, but it is not yet completely satisfactory. In fact, one cannot use the customary approach of the judge before students have learned how to do input and output. In some languages (C and Java, for example) this is not a trivial task, and it is never a trivial task when large sets of data have to be handled. Furthermore, it makes little sense to spend time on the details of formatting numbers or splitting strings when problem solving using computers is

barely being introduced. In any case, it is a pity not to use the automatic judge to validate the very first pieces of code with which the novice programmers are struggling, because they are incomplete programs, and because they do no input-output. It is a pity because there is a large number of small tasks that the students could try out and which could not reasonably be checked by a human with the promptitude that we seek, and because we have observed that students derive a special pleasure when the results of their efforts are immediately rewarded with an "accepted" message from the judge.

On the other hand, there is a trend towards using functional languages such as Haskell in introductory programming courses. These languages run on an interactive "calculator", in which functions are called with supplied arguments and results are displayed automatically. The same situation occurs in logic programming, with Prolog. In these environments there is no input-output, in the conventional imperative programming sense, with reads and writes, at least in the initial stages. This means that functional programming and logic programming can be used with the automatic judge, from the very beginning, since the input and output are handled automatically, so to speak. This has completely modified the first labs: the initial exercises are now online, meaning they are to be submitted to the judge for feedback, whereas in the past students would pass to the next exercise after "manually testing" once or twice only. In addition, the competitive attitudes that we observed made most students want to solve all the exercises, to get more points, and be in the top positions of the informal rankings.

Indeed, the lessons learned from approaching programming using these "unconventional" languages can be brought back to the imperative style of programming. With Pascal, C, C++ or Java, we can use the judge in an "advanced" way, similar to the one used for Haskell or Prolog, with incomplete programs that do no input or output, even if this requires some hacking to overcome its original design of the judge. With this, we have been able to provide to our students, in various courses, an early introduction to competitive programming.

This paper is organized as follows. After this introduction, we briefly describe, in Section 2, the competition environment that we use in our programming courses, which is based on the automatic judge Mooshak (Leal and Silva, 2003; Mooshak site). Then we discuss how we have tweaked it into being able to use problems with no input-output in conventional terms, first with Haskell, in Section 3, then in Prolog, in Section 4. In Section 5, we bring those techniques to conventional programming languages, allowing novice students, even those who are programming for the first time, to immediately have their small, incomplete, programs automatically checked. In Section 6, we report on the feedback we had from students who were exposed to this approach in the introductory programming course with Haskell. A conclusion follows.

2. Competition Environment

There are several systems for automatically assessing programming assignments (Douce *et al.*, 2005; Ala-Mutka, 2005). Each one has strengths and weaknesses. Given our own

background, it was only natural that we chose for our courses an automatic evaluation system related to programming contests: Mooshak. We have been using Mooshak to manage the programming competitions that we organize for more than five years, and we know exactly how to take advantage of its strong points, avoid the dark corners, and even extend it to do things it wasn't originally designed for. In all of our courses, these days, we make extensive use of this publicly available platform.

Mooshak grew from previous experience with a late 90's web-based learning environment for Computer Science called Ganesh (Ganesh Site). Mooshak appeared first on the context of the ACM ICPC (ICPC Site). In this family of contests, typically you have teams of three students solving problems in C, C++ or Java. Solving means creating a program that will compile under some given compilation flags and that, when it runs, produces the correct output for the secret input files associated with the problem. Mooshak was first published in 2000 and since then it has matured into a very robust, flexible and accurate platform. In its initial realm of ICPC, it has been used successfully in several editions of local, national and international contests, such as the Southwestern Europe Regional ACM Programming Contest (*SWERC 2007* site).

Since the beginning, Mooshak architecture was designed having in mind different kinds of programming competitions. For example, in ICPC a program is considered correct only if it passes all tests; otherwise it does not count. In the Olympiads, there is a partial credit scheme and points are allocated depending on which tests passed; i.e., a contestant will get points even if his program fails some tests. Mooshak provides a way to give a value to each test, and ranks the contestants accordingly. Moreover, it has way of creating customized ranked schemes. So, one could for example grade the submissions according to the source code size, or to the memory usage. Finally, it is possible to run in a mode that permanently displays the current ranking, useful for ICPC, or that hides it, useful for the Olympiads.

In terms of feedback, Mooshak is more complete than other automatic judges. Instead of just giving the result of a submission as a crude "Wrong Answer", "Runtime Error" or "Compile Time Error", Mooshak allows us to define more detailed reports that really give the students more opportunities to understand what went wrong. This is especially important with compilation errors. In an on-site contest, all contestants are using the same environment as the judge, but on a distributed contest, or in a classroom environment, it is common that student use heterogeneous platforms. Even if very strict compilation flags are used, to ensure adherence to the language standard, it is not uncommon for a program that compiles successfully at the student's computer to fail in the judge. In this case, Mooshak can present the compiler error message, guiding the student in solving the issue. This, by the way, has the nice side effect of leading students to better appreciate the advantages of following standards. In the case of runtime errors, the system can be set to provide the submitter with the exact kind of exception that the program rose. This can be important from an educational point of view, because we don't want the students to be frustrated for committing mistakes that are common to beginners, but instead we want them to learn from errors, helping to achieve correct solutions.

Mooshak's user interface is completely web-based. It has four basic profiles: contestant, judge, administrator and guest. For contestants (or students, as in our case), the

interface shows the problem descriptions, and allows them to submit their code and ask questions. The judges (the teachers) can go through the questions and answer them using the platform. Thus, everyone will see all the questions and all the answers, in an organized way that also fosters interaction between students and teachers. The system also has the capability of issuing global warnings, and this is very useful in an "emergency". Judges have access to a detailed report of every submission, and can check in detail what result was obtained in each test case. If necessary, they can even manually change the result of a submission or ask for an automatic re-evaluation (for example, in case the tests are modified after the contest started). The administrator profile is used to set up a contest and to upload all files concerning it. Finally, the guest profile is available to the public and does not require authentication. With a guest login, anyone can take a look at rankings and at the submissions as they are being made. Indeed, we have observed students proudly giving the URL of their course's Mooshak site to friends, in order to show off their achievements. An example screenshot of the contestants' view can be seen on Fig. 1.

Perhaps the strongest point of Mooshak is its simple yet very flexible way of evaluating programs. Basically after some code is submitted it passes through three phases. The first one is compilation. Compilation is completely customizable in the sense that the administrator can set up the command line that calls the compiler. Normally it is used precisely for compiling the source code and obtaining an executable, but nothing forbids us from doing whatever else we want in this phase. Mooshak considers a compilation to be successful if the program (or programs) run in the compilation phase do not write anything to the standard output. In case the compiler does output something and we want to ignore it, there is a mechanism for that, based on regular expressions. Although the compilation phase is meant for compiling, we have been using it for other purposes as well. For example, in some occasions, we use Mooshak for accepting other types of as-

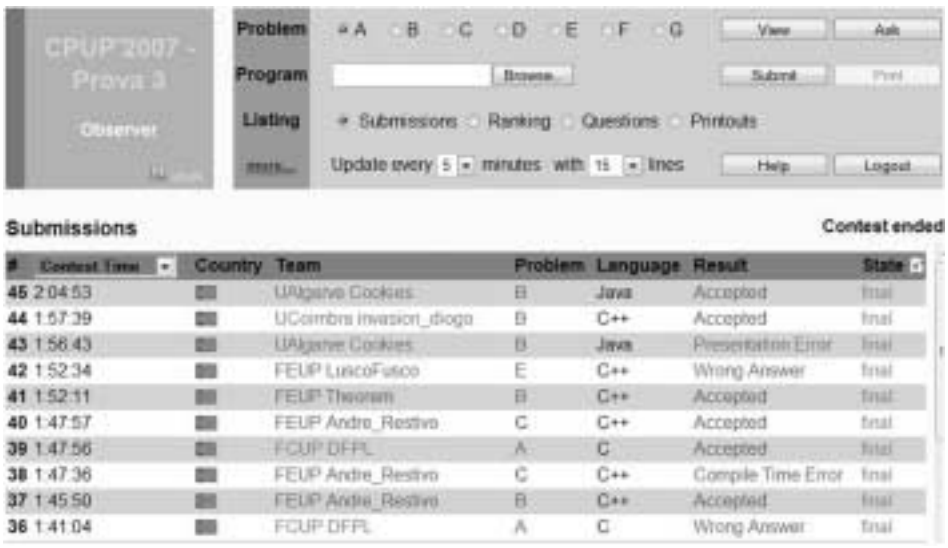


Fig. 1. Example screenshot of Mooshak in contestant mode.

signment, and we use the compilation phase to check whether the submitted zipped file contains all the specified deliverables, in the required format.

The second phase is execution. The program is run once for each test case, again by way of a customizable command line. Mooshak provides a safe sandbox for execution of programs, so we do not have to worry about the students trying to do anything nasty with their code. We can create boundaries for the sandbox, specifying things like the maximum CPU time allowed or the memory limit. For each test case, Mooshak feeds the command line with the content of corresponding input file and it stores the output in a temporary file.

The third phase is the evaluation itself. Mooshak provides a default evaluator which compares the output that was obtained with the expected output. For flexibility, it also allows us to define a custom command to be applied to the output file, thus evaluating it directly.

With this simple versatile scheme, Mooshak can be tuned to a variety of programming languages. Indeed, it has been successfully and routinely used with the standard competition languages – C, C++, Java and Pascal – but also with other imperative languages such as Perl and Python. Moreover, it is able to cope with functional languages such as Haskell and logical languages such as Prolog, as we will see in more detail. Colleagues are also using it for evaluating shell programs.

Although Mooshak was initially developed for a particular kind of programming contests, it was designed in such a flexible way that it can be used for a multitude of things. Basically, we can think of it as a platform that provides a sandbox for safe execution of user code, together with an interface for submitting that code and a user management system. In what concerns us here, this set of facilities also makes it a very fine pedagogical tool.

3. Using Competitive Programming with Haskell

Over the years, we have been using an imperative-programming first approach to teaching programming, at university level (CC2001). More recently, we introduced the automatic judge Mooshak as a pedagogical tool, and this has proven to be invaluable. With it, not only the amount of programming done by the students increased several times, but the courses themselves became more joyful, and also more popular, more transparent and more spoken of. It also made students more diligent programmers, because deadlines, being enforced automatically, are more difficult to circumvent, and more rigorous programmers, because of the greater effort required perfecting the programs so that they pass all the secret tests held by Mooshak.

The first contact of students with Mooshak tends to be turbulent. Usually students react angrily when Mooshak refuses to accept their solutions, which they could swear were bulletproof. Moreover, at the beginning of the semester, good students are striving for visibility in the class. It must be very disappointing to have your programs rejected and not being able to hide it.

Indeed, by having Mooshak evaluate students' programs, we are creating a new obstacle to students: they must solve the programming exercise and they must do it a way that Mooshak accepts. Take, for example, the problem of solving a second degree equation. The problem is not difficult to understand, but it has its own small programming complications: the three coefficients must be read, the roots computed, not forgetting that the equation may have no real roots, and then the roots must be written, if they exist, or else some message must be produced. Since we are handling real numbers, for automatic evaluation to be possible, we must specify precisely the number of digits after the decimal point, which root to be written first and how to separate the two roots.

Most likely, as teachers, we would like our students to concentrate on the function that solves the equation. However, students are eager to see the results, and if they are still insecure about functions, they will try to do everything – reading, solving, writing – in the main program.

Conventional input-output automatic judging does not help here: it only cares about the input-output behavior. In a way, automatic judging somehow works against us, teachers, in the elementary stages, from that point of view. On the other hand, it makes life harder for the students, unnecessarily, by requiring them to master the details of input of variables and output of expressions, which we might want to downplay initially.

More recently, we adopted the functional-first approach to programming, using Haskell. Of course we wanted to continue using automatic judging, with Mooshak, but for that we knew we would have problems on two fronts. On the one hand, Mooshak was designed for compiled languages. We would be using Haskell with an interpreter in the course and we would want Mooshak to rely on that interpreter, to avoid superfluous mismatches. On the other hand, input-output in Haskell is an advanced topic, one that typically is not addressed until the final weeks of the course. It is not necessary initially because functions are called in the interpreter, acting as an interactive calculator, with arguments typed in as needed, and results are written back, automatically.

In order to be able to run autonomously, Haskell programs must have a main function. However, unlike C, for example, we can live without the main function for a long time. Our programs are simply collections of functions and we can call any of them, from the interpreter, providing the arguments each time. This is fine, from a pedagogical perspective, because it impels students to organize their solutions as collections of functions, instead of long lists of intricate instructions within the sole main function. It also frees them from the worries of input, since the values for testing are typed in as arguments of the function, and of output, since the result of the function is written automatically, using a default format. Again, this clear separation between computation and input-output can have a very positive influence in the early habits of the students.

For Mooshak, a compilation is successful if there are no error messages on the standard output. The interactive interpreter is not useful for this. Instead, we used a standalone version that loads the source file and runs its main function. Since the file submitted by the students does not have a main function, Mooshak adds one, via an ad-hoc script, just for the sake of compilation. This main function is empty. Therefore, if the submitted file was syntactically correct, the program does run in the interpreter and produces no

output, just like a successful compilation. If something is wrong, the stand-alone interpreter outputs a message, and that is enough for Mooshak to understand that there was an error.

After the submitted program passes the compilation stage, the function to be tested must be called. Typically, we want to call it several times, with different arguments. Recall that normal Haskell functions, the ones we are considering in this discussion, perform no input-output. So, instead of using an input file from where the data is read, we use a Haskell source file with a main function calling the function to be tested, using the desired arguments. Indeed, for flexibility and simplicity, this file replaces the input file that is provided for each test in the standard operation of Mooshak. Our source file is appended to the submitted program, via a script, and the result is run by the stand-alone interpreter. The result of each function call is written on the standard output and can be compared to the expected output.

By hacking Mooshak this way we were able to use it from the first day of our Haskell course. At the beginning, the programming exercises dealt with simple functions, such as counting the odd numbers in a list, checking if a given number appears in a list, etc. Students first write the programs using a text editor and experiment on their own with the interactive interpreter and then submit to Mooshak. For each accepted submission, students earn one point. For these simple programs, it seemed easy to have an "accepted" and that helped students gain confidence in the system. This contrasts with our experience in previous courses, where, on the contrary, most early submissions were erroneous, which left students very uneasy.

We used this setup for the most part of the course, not only for exercises involving simple functions, but also for larger assignments and for programming problems similar to those used in competitions. Only in the final part did we introduce input-output in Haskell, and we came back to a more conventional use of Mooshak, with input and output handled explicitly by the program and a main function that is called by the interpreter, by default. At this stage, students were more capable of overcoming the details of input-output than they could have been in the beginning. However, the presence of the main function requires a change in the compilation script, because we do not want the program to actually run in the interpreter in the compilation phase, which is what would happen because the provided main function was not empty, as the one we added before. Well, the new script edits the source file, replaces the identifier main by another unlikely identifier and then passes it on to the old script.

Overall, in this course, there were 69 problems to be submitted and evaluated by Mooshak. This is much more than could have been handled manually by teaching assistants. Not all students had all their submissions accepted, of course, but all of them certainly got a lengthy exposure to the automatic judge. The points earned by the accepted submissions counted for 30% of the final grade.

Although we did not stress the competitive aspect of the assignments, many students thrived in it. We observed that many of them enjoyed being on the first places on the ranking and made an effort to solve the problems as soon as possible. As anecdotal evidence on this, at one occasion, we made a mistake when setting up a new contest in Mooshak

which caused the ranking to disappear. We immediately received complaints from the students that "without the ranking, it was no fun".

As a side effect of their involvement with an automatic judge designed for managing programming competitions, some of these students developed a liking for this style of competitive programming. Perhaps some will participate in competitions in the future. However, most programming competitions still use only the usual programming languages – C/C++, Pascal and Java – and these students are not prepared for them just yet.

4. Using Competitive Programming with Prolog

Very much in the way described in the previous section, we were able to use Mooshak in a course with the main focus on logical programming, using Prolog. Instead of just testing "by hand" their Prolog programs, students use Mooshak for that. At the same time, teachers obtained a detailed picture of the successes and failures of the students and part of this information was used for grading.

Like in the case of functional programs, we do not usually compile logic programs; instead, we load them on an interpreter, in our case Yap (Yap site), a fast Prolog interpreter developed also at Universidade do Porto. Contrary to imperative languages, in Prolog we declaratively express a program using *relations*, called *predicates* in Prolog, in a way similar to databases. Executing a Prolog program is actually running a *query* over those predicates, and typically this is done using the command line of the interpreter. We test by making queries about the relevant predicates. The Prolog environment then uses inference rules to produce answers.

A Prolog program therefore does not need a "main function": it is simply a collection of predicates. In the context of Mooshak, compiling can be seen as loading the program in the database. Executing a program is done by running a series of queries. We use a metapredicate that collects the list of all answers that satisfy the goal, comparing that list with the expected list of solutions (ignoring the order). Often, the solution contains only one possible answer (for example, the length of $[a, b, c]$ is always 3), but this approach effectively lets us natively test predicates that accept, and should produce, multiple solutions in a very simple fashion. In programming contests there are situations where several solutions have to be handled by ad-hoc correctors or else the burden is passed to the contestant itself, by adding a redundant layer of complexity to the problem, asking the solutions to be presented in a specific order, or a solution with specific properties to be printed.

In concrete terms, we run a customized predicate in which the rule part is made of the conjunction of all the query calls we want to test and a final query which uses Prolog's I/O capabilities to write some simple sentence to the standard output (it can be for example a simple "ok"). So, this sentence is written only if all the queries are successfully satisfied, that is, if all tests are passed. If the expected output in Mooshak is exactly the sentence that we use, then the default evaluator gives "accepted" if and only if the program passes

all the tests, thus mimicking what happens in a conventional evaluation. If we want to make Mooshak aware of exactly which queries are passed, we can just create several input tests, each of them with the desired queries.

With this scheme, we can use Mooshak from the very beginning of the logic programming course. As with Haskell, students do not need to write complete programs in order for them to be tested by Mooshak. Actually, with this in mind, we have put together a set of more than 60 predicates and made them available through Mooshak permanently. Different subsets of this set are viewed as different "contests". For example, the "Lists" contest asks the students to program predicates to compute things such as the maximum of a list, the average of a list of numbers, the concatenation of two lists or the removal of duplicates. This invites students to become self-sufficient in their struggle with the language, relying on the automated feedback for validating the solutions they invent. We published additional problems that went beyond the goals of the course, as challenges to the most diligent students. Actually, some of these problems were taken from real programming competitions, namely the Portuguese Logic and Functional Programming Contest (CeNPLF, 2007). We observed that many students accepted the challenge and made an effort to solve those problems which, at the time, were on the limits of their programming capabilities.

Like in the case of Haskell, this competitive environment fostered a desire to solve all the problems proposed, in order to reach a higher position in the ranking and get the satisfaction of being up to the challenge. Indeed, having the program that you tentatively and hesitantly wrote to solve a difficult problem finally "accepted" by Mooshak creates a sense of fulfillment that we, as teachers, must not underestimate. In fact, students that would otherwise just look at the exercise and put it aside as too easy or too difficult, have now the incentive of actually programming it, gaining more experience and more wisdom in the process, because things are not usually as easy or as difficult as they seem. While we must contradict the pitfall of equating being accepted by Mooshak and correctness, we should note that the validation provided by the automatic judge does help many students to gain confidence in themselves and their emerging programming capabilities.

For the purpose of direct student assessment, in some years we have used Mooshak in "submission labs". Students were handed in a predicate, and they had to program it and submit it to Mooshak within a determined amount of time. If it was accepted, they would earn some points. This is very much like a programming competition, with an immediate, tangible result.

Mooshak was also used to guide students through more complex programming assignments. Instead of building a complete program from scratch on their own and testing it at the end, students follow directions to implement intermediate predicates, which they can test with Mooshak, before they advance. This way, students are more confident of the correctness of the building blocks of their programs. The intermediate predicates were graded directly by Mooshak, and the success of the overall assignment indirectly benefited from the automatic evaluation.

Typically, we leave all contests open, even after the course finishes. Students of the course can come back to the early exercises, in preparation for the exam, for example. We

also have had cases of students from previous years who resorted to the "old" and familiar Mooshak to refresh their Prolog, years later. This is made possible by an optional facility of automatically accepting new registrations, thereby opening Mooshak to all interested, when we are willing to allow that. Actually, some current students use that facility, which allows them to experiment their code anonymously.

5. Using Competitive Programming with Imperative Languages

The more conventional imperative programming languages can also benefit from schemes similar to those we describe for Haskell and Prolog. In fact, when we use Mooshak in courses for teaching programming in Java, C, C++ or Pascal, we usually do it in the ordinary form of having the submitted programs perform their own input and output. In other words, we prepare the contests in the normal way, setting them up for doing the automatic evaluation of the complete programs student will write. This is fine, but we cannot follow this approach in the very first labs of the semester, because students who are taking the first steps in programming do not have the knowledge to correctly construct a complete meaningful working program. For that, among other things, they would need to know how to use the input and output functions of the language, what is always tricky and confusing for beginners. Besides, they would have to respect precisely the specification of the output given in the problem description. We know by experience that this is difficult at first, when one is not used to that kind of painstaking detail.

Whereas with Haskell and Prolog we did not have input and output at the onset, with C, C++, Java and Pascal, we do, but we are willing to waive it. The beauty of this idea is that we can easily use and adapt Mooshak to evaluate only pieces of code and not complete programs, also for these more conventional languages. Technically it is even simpler than with Haskell and Prolog. We ask the student to program a certain function and we provide, inside Mooshak, a program that calls that function. For example: suppose we require a function that receives three real numbers as arguments and returns the average. The program we supply calls that function directly several times, with different arguments, each time writing the result on the output stream, exactly as we did with Haskell functions.

For doing this, we only have to tweak the compilation phase. Now we have not only the source file that was submitted, containing the definition of the required function, but also a complete program prepared beforehand with our own customized main function and all library declarations in place, and a header file with the prototype of the required function. Care must be taken in order to avoid such name clashes, but this approach is certainly feasible. Therefore, we compile the submitted source file plus the supplied program.

After this, running a program would simply be calling the executable created after the compilation. Since we wrote the main function ourselves, we can make it work the way we want. We can hardwire all the tests inside our own program, calling the evaluated function for each set of arguments to be tested, or we can write a loop that reads a set the arguments from a data file and calls the function with those arguments.

No matter what technique we choose, the remarkable fact to observe is that it is possible to do this in a very early stage: even the most primitive concepts and technique that we teach can be automatically evaluated, immediately. This can be done with any programming language, imperative, functional or logic.

This early introduction of automatic evaluation for imperative languages has a number of benefits, other than enticing students into the programming competitions, and the early exposure to the joys of having your program accepted by a robot, and earning points by it. First, it allows us to use the first classes to stress program development using functional decomposition, without the distraction of input and output. Second, it leads to more submissions being accepted in first labs, raising the spirits, and making students trust the teachers more that they would if what the teachers presented as an easy procedure proved to be embarrassingly difficult and distracting. In fact, the usual approach of submitting complete programs becomes very frustrating for beginners, precisely because very often the error is not in the function we want to test but in the part of the program that reads and writes. In this part, one has to worry about formatting details, about parsing the input line, about dealing to the end of data, and this can be overwhelming and is always distracting.

As a side-effect, this approach forces those students who have some knowledge of programming from secondary schools, and who are able to write long sequences of instructions, happily mixing input, computation and output, to drop that practice, and understand that the real challenges of programming are elsewhere.

6. Feedback from Students

We have used the approach described in Section 3 recently, in a first-year introductory programming course. There were students taking the course for the first time, but there were many repeating it who, having failed in the past editions, were taught differently. We initially observed frenzy amongst students, as they got acquainted with the automatic judge and began to become conscious of the way their assignments were to be submitted and graded. After a while, the agitation increased due to the perception that there seemed to be more assignments than usual. This was true: with the automatic judge, all the assignments were to be submitted and each successful submission did count, very little, in the final grade. Eventually, things calmed down, with practice and routine.

At the end we carried out a survey, to ascertain the students' reactions and overall impression of out approach. We had 115 responses, from more than 90% of the students taking the course. For most questions we used the 5-point agree-disagree scale. Table 1 summarizes the responses to the questions relevant to this paper.

In the questions, "platform" means the automatic judge, Mooshak, plus the learning management system that we were using, Moodle (Moodle site).

We observe that the students are not very assertive, except when giving advice to teachers, as in the forth question. For all questions, more than half of the students either "strongly agree" or "agree". Even for the third question, that shows a certain discomfort with Mooshak, only less than 20% of the students consider that Mooshak is not an effective learning tool.

Table 1
Results of survey, abridged

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree	Don't know
Programming assigments are more interesting because they are supported by Mooshak	26%	50%	17%	6%	1%	1%
Assessment by the platform is generally fair	15%	62%	18%	4%	1%	1%
With Mooshak we learn more than without Mooshak	20%	35%	25%	11%	7%	2%
Other teachers should be encouraged to use a similar platform	61%	25%	7%	3%	2%	2%

This survey confirms results that we have observed in similar surveys for other courses, but is especially interesting because it was given to a class that used Mooshak for the first time, starting immediately on the first lab, with a functional programming language, as we explained in Section 3.

7. Conclusion

It is a cliché that computers excel at performing boring, repetitive tasks. For teachers, one of those boring, repetitive tasks is reviewing students' homework, especially when there are many students and many assignments. Computer help here is most welcome. In many disciplines, however, computers are not up to the task, at least not yet, but for programming, we are lucky that we can take advantage of tools like those used for managing programming competitions to take care of that chore.

Automatic judges have been designed mostly for imperative languages; they traditionally work by inspecting the input-output behavior of programs. Yet, we have been using them with Haskell and Prolog, which are not imperative, and for which input and output is not a technique that is learned in the earlier stages. For making automatic judges work with Haskell and Prolog we had to adapt them and in the process we discovered that, avoiding the complications of input and output, we were able to start using automatic evaluation much earlier in the courses. This possibility promotes a new approach of teaching, in which all the exercises and assignments, from day one are to be submitted to the judge, with immediate feedback to the students. This way, the productivity of students, measured in programs written and submitted, increases significantly. Also, the visibility of everybody's successes and failures aids in creating a challenging working environment and a healthy sense of community. This makes learning more enjoyable and more rewarding.

As a side-effect, students got acquainted with competitive programming, via the tools used to evaluate their programs. Many derive a special pleasure in appearing in the top places in the rankings kept by the automatic judge, even if that has no importance in the context of the course. We can expect that some of them will have been bitten by the competitive programming bug and will now create teams to participate in real programming tournaments.

References

Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, **15**(2), 83–102.

CC2001 (2001). *Computing Curricula 2001, Computer Science Volume.*
`http://www.sigcse.org/cc2001/`

CeNPLF (2007). *Concurso/Encontro Nacional de Programação em Lógica e Funcional.*
`http://ceoc.mat.ua.pt/cenplf2007/`

Douce, C., Livingstone, D. and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* (*JERIC*), **5**(3).

ICPC site. *The ACM-ICPC International Collegiate Programming Contest.*
`http://icpc.baylor.edu/icpc/`

Ganesh site. *Ganesh – An Environment for Learning Computer Science*.
`http://www.dcc.fc.up.pt/~zp/ganesh/`

Leal, J.P. and Silva, F. (2003). Mooshak: a Web-based multi-site programming contest system. *Software Practice & Experience*, **33**(6), 567–581.

Moodle site. *Moodle – A Free, Open Source Course Management System for Online Learning.*
`http://moodle.org/`

Mooshak site. *Mooshak Contest Management System*.
`http://mooshak.dcc.fc.up.pt/`

SWERC (2007). *Southwestern Europe Regional ACM Programming Contest 2007.*
`http://ctp.di.fct.unl.pt/SWERC2007/`

Yap site. *Yap Prolog – Yet Another Prolog.*
`http://www.dcc.fc.up.pt/~vsc/Yap/`

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 programming 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 data structures and algorithms, artificial intelligence and distributed computing.

P. Guerreiro is a full professor of Informatics at Universidade do Algarve. 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 2008. He is the author of three popular books on programming, in Portuguese. His research interests are programming, programming languages, software engineering and e-learning.

Olympiads in Informatics, 2008, Vol. 2, 163–170

Japanese Olympiad in Informatics

Seiichi TANI
Department of Computer Science, Nihon University
Setagaya-ku Sakurajousui, Tokyo 156-8550, Japan
e-mail: sei-ichi@tani.cs.chs.nihon-u.ac.jp

Etsuro MORIYA
Department of Mathematics, School of Education, Waseda University
Shinjuku-ku Nishi-Waseda, Tokyo 169-8050, Japan
e-mail: moriya@waseda.jp

Abstract. The Japanese Committee for the IOI (JCIOI) is a nonprofit organization and one of its purposes is promoting the interest of Japanese secondary school students in computer science as well as computer programming through various activities including Japanese Olympiad in Informatics. This article describes the process of selecting and training the Japanese IOI team and the key issues that faces JCIOI.

Key words: olympiad in informatics, training, programming competition, IOI, International Olympiad in Informatics.

1. Introduction

The Japanese Committee for International Olympiad in Informatics (JCIOI) started the Japan Olympiad in Informatics (JOI) in 1994 and sent Japanese delegations to IOI 1994 (Sweden), IOI 1995 (Netherland) and IOI 1996 (Hungary). Unfortunately, due to financial difficulties, the activities of the JCIOI were suspended from 1997 to 2004.

When the JCIOI attempted to restart its activities, the Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT) decided to support school children participating in international science and technology contests including the IOI. This decision efficiently provided enough support needed to reinstate the JCIOI. The aim of MEXT is to encourage talented school students to take more interest in and orientation toward science and technology, through competitions and exchanges with other children of the same generation throughout the world.

At the time of reinstatement, Japan did not participate in any international science olympiads, except for the International Mathematical Olympiad. Consequently, the JCIOI was reorganized as a nonprofit organization sponsored by the Japan Science and Technology Agency (JST), an independent administrative agency affiliated with MEXT. In 2006, the JCIOI sent the Japanese delegation to the IOI 2006 (Mexico) for the first time since 1996.

In the two years of participation at the IOI, the Japanese Team won three gold, one silver and two bronze medals. Based on these unexpectedly good results, the IOI contestant selection procedure has been deemed successful. However, a certain number of issues remain to be resolved. One such issue concerns the quality and quantity of computer science education for younger students. Another issue is related to the fact that students possess adequate skills in either programming or mathematical reasoning, but not both.

The goal of the JCIOI is to improve the abilities of the students gifted in computer science as well as the overall population of secondary school students. The JCIOI also aims to generate greater interest in informatics in secondary schools. This article will outline some ways in which the JOI can achieve these goals. The paper is organized in the following way. Section 2 introduces the Japan Olympiad in Informatics and the IOI contestant selection procedure. Section 3 discusses key issues and future works. Section 4 shows the conclusions.

2. National Olympiad in Informatics and Selection of IOI Contestants

2.1. *Structure*

Activities of the JCIOI were restarted in 2005. That year, the JCIOI sent observers to IOI 2005 in Poland and held the first round of the Japanese Olympiad in Informatics (JOI) 2005/2006. At the present time, the JOI attendance fee is free because of the support of JST. While the committee works to improve the domestic olympiads, JOI has maintained a simple structure due to personnel, budget, and time constraints.

The JOI has two rounds. The first round is an open online programming contest, with the tasks provided only in Japanese. Students who qualify are invited to the final round which is held at a central location in Tokyo, so contestants from all over Japan are gathered at one place. The top students of the final round are awarded gold, silver or bronze medals according to their grades. The JCIOI invites students who perform well in the final round of the JOI to a training camp. The IOI contestant selection takes place at the training camp. The Japanese team for the IOI, along with other camp participants, continues the training process through correspondence from the end of the camp until the moment of departure for the IOI. The expense for travel to IOI is covered by JCIOI via JST aid.

2.2. *First Round of JOI*

The first round is usually held on a Sunday in December. Participants have to solve six tasks with the programming language that they wish to use for three hours. Five test data per task are distributed in the beginning of the contest and contestants have to upload the outputs for the test data. The grade of a participant is automatically decided based on the number of correct outputs uploaded. One of the objectives for adoption of such a system is to allow the highest number of contestants possible to enter the competition. Due to the

disparity in the level of ability, four of the tasks are relatively simple, while the remaining two require more challenging mathematical and algorithmic considerations.

The number of participants of the first round at JOI 2005/2006, 2006/2007 and 2007/2008 was approximately 80, 150 and 280 respectively.

2.3. *Final Round of JOI*

The final round of the JOI is an on-site programming contest. It is time constrained and the contestants are required to design efficient algorithms and to implement them appropriately. The most difficult tasks of the competition are intended to be as difficult as the least challenging IOI tasks.

The final round takes place in an examination venue in Tokyo on a holiday in February. Contestants from all over Japan meet at the site and the traveling expenses are covered by the JCIOI. The JCIOI invites high scorers of the first round who are under 20 years of age on the day of the final round and secondary school students, excluding high school seniors. Note that students usually graduate form high schools at 18 years of age in Japan. From JOI 2007/2008, top ranked students in each region are also invited to the final round. The numbers of students who proceeded to the final round at JOI 2005/2006, 2006/2007 and 2007/2008 were approximately 30, 40 and 50 respectively.

Five tasks were proposed to contestants, to be solved for three hours in JOI 2005/2006 and 2006/2007. The duration of the contest has been extended to four hours since JOI 2007/2008. For each task, contestants have to write programs in C/C++ or Java that solve the task and to submit them. Contestants are provided sets of sample inputs and corresponding outputs in order to confirm whether their solutions satisfy the output format and to assist in estimating the running time of their programs. After the examination, the submitted solutions are compiled and run with test data unknown to the contestants on an evaluation machine. The specification and the programming environment of the evaluation machine are the same as the machines used by contestants in the final round. The participant's grade is automatically calculated by the number of test data for which their solutions output correct answers within the time limits. The test data are set as distinguishing efficient solutions from inefficient solutions.

2.4. *Training Camp*

The training camp starts March 19th and ends March 25th every year. The JCIOI invites top ranked students from the final round of the JOI. The number of camp participants at JOI 2005/2006, 2006/2007 and 2007/2008 was 8, 13 and 16 respectively. In addition to IOI contestant selection, four lecturers are also invited to speak at the camp. These speakers include university faculty members, graduate students, and IT professionals. Table 1 sketches the schedule of the JOI 2007/2008 training camp. During the camp, former IOI contestants and ICPC contestants work as tutors. They communicate with the participants and support in organizing the camp.

The camp competitions are held in IOI competition format. Three tasks are set at each competition. The twelve tasks are intended to cover as many IOI problem types as

Table 1
The schedule of the JOI 2007/2008 training camp

March	Morning	Afternoon	Evening
19th		Arrival	Practice Session
20th	Competition 1 (3 hours)	Lecture 1	Comments and Discussion 1
21st	Competition 2 (4 hours)	Lecture 2	Comments and Discussion 2
22nd	Competition 3 (4 hours)	Lecture 3	Comments and Discussion 3
23rd	Lecture 4		Free Time
24th	Competition 4 (5 hours)	Awarding Ceremony Public Lecture	Comments and Discussion 4 Farewell Party
25th	Departure		

possible. The difficulty of the last competition is similar to those of the IOI. Almost all tasks are batch tasks, with a few being reactive and/or output-only. For tasks requiring source codes as solutions, C and C++ programs are accepted. The top four students are selected to represent Japan in the IOI.

Two of the lectures are focused on the design and implementations of efficient algorithms. The other lectures deal with topics that are not directly concerned with programming contests. For example, an introductory lecture for theoretical computer science is provided. The titles of lectures at JOI2007/2008 were as follows:

- Let's try to use STL;
- An introduction to computational complexity theory
  – Get a million dollars to solve the "P = NP?" problem;
- An introduction to information retrieval
  – Algorithms for Web search engine;
- Let's solve IOI problems with a functional programming language.

At the present time, the camp schedule is completely full. The IOI participant selection, programming contest ability improvement seminars, and computer science motivational presentations account for the full schedule.

2.5. *Correspondence Course*

After the training camp, a correspondence course starts for the camp participants, including the members of the Japanese IOI team. The participants solve problems from past IOIs and regional informatics olympiads, after which they submit their solutions to the team leader of Japan. After the deadline set for each task, the submitted solutions are posted on the course bulletin board system. Participants exchange views on submitted solutions for two weeks and the coordinator of each task gives them suggestions and summarizes the discussions. If a participant submits a solution after the deadline, the coordinator will post comments about the solution on the bulletin board system.

2.6. *Task Creation*

The scientific committee (SC) of the JCIOI controls the entire process of creating JOI tasks from the first round through the training camp. The SC attempts to maintain a balance between suitable qualifier selection and participant satisfaction. The steering committee of the JCIOI appoints the members of the SC. Initially the SC consisted only of faculty members when the JCIOI resumed activities. However today, former IOI contestants and ICPC contestants are involved in the SC.

The deliberations on task creation are held both at off-line meetings and on the bulletin board system for the SC. For JOI2007/2008, the off-line meetings were conducted ten times and approximately 400 messages were posted on the bulletin board system. Subversion, a version control system, has been used to maintain tasks, test data, solutions, commentary, etc. SC members commit drafts of their tasks to SC's Subversion. A task set for a contest is chosen from a pool of tasks stored on the Subversion that meets conditions relative to difficulty, area and type. If there are an insufficient number of tasks, the SC will create new ones.

3. Key Issues and Future Works

In the previous section, the IOI contestant selection procedure was mentioned. Currently, the procedure in place appears to be successful in choosing appropriate candidates. IOI contestant selection is not the only goal of the JCIOI, however. The aims of the JCIOI also include improving the abilities of the students gifted in computer science, as well as generating greater interest in informatics in secondary schools. At JOI 2005/2006 and 2006/2007, only IOI contestant selection and IOI participation training were held due to personnel, budget, and time constraints. Recently, the JCIOI situation has been improving. Evidence of this is as follows:

- the number of faculty members and former contestants who have joined the JCIOI continues to increase each year;
- in 2007, the JST began to support the promotion of sciences including computer science in secondary schools in addition to hosting domestic science competitions and sending Japanese delegations to international science competitions;
- Fujitsu Limited has supported the activities of the JCIOI since 2005. Since April 2008, NTT Data Corporation has started to sponsor the JCIOI.

Hence, the JCIOI has initiated new activities to achieve its aims. The remainder of the section will address two of these new activities, as well as the remaining challenges to be faced.

3.1. *Summer Camp*

M. Forišek mentioned in (Forišek, 2007): "We strongly believe that the thinking process (in other words, the problem solving process) is the most important skill we want to see in

our contestants. This is what they will need in their future lives, should they pick a career in computer science." The JCIOI agrees with Forišek's theory. The JCIOI considers the most significant challenges of the IOI competition format to be the following:

- a lack of consideration for the thinking process when grading solutions;
- a lack of open-ended problems to challenge higher level students;
- excessive focus on the importance for quickness when completing tasks;
- excessive focus on the importance for coding skills.

A student who produces an excellent solution, but is unable to submit it within the contest time frame will unfortunately not receive any points. The students who are able to solve all IOI tasks in five hours and five minutes seem to be as outstanding as ones who are able to solve the tasks in five hours. Every IOI task is capable of being solved in two hours or less. At the current time, the IOI and the JOI have implemented regulations that are biased towards students who are better able to write code and finish tasks quickly. These regulations regrettably neglect the thinking process needed to succeed in the field of computer science in the future. If the regulations are changed, another bias would undoubtedly be created as a result. Therefore, in 2007 the JCIOI has started a summer camp that does not include any competitions.

The camp is held for three or four days soon after IOI in late August. Approximately 20 students including past JOI training camp attendees participate. At the first camp last year, a faculty member gave a lecture about computability theory. In addition to that, the camp participants were divided into small groups and each group studied a computer science text book for undergraduate students. Each group gave a presentation about what they learned on the last day. The style of the summer camp may change in the near future.

3.2. *Introductory Course of Computer Science*

In Japan, computer science is absent from the secondary school curricula at the present time. In order to promote it in secondary schools, elementary educational materials should be provided to them. The first step in the realization of this goal is to develop a web site that introduces computer science to young people. This will be done in cooperation with Fujitsu Limited and an educational group to promote "Computer Science Unplugged" (Bell *et al.*, 2002; Kanemune *et al.*, 2007). Fujitsu Limited operates a web site for children. They plan to add an introductory course of computer science at the Fujitsu's kids site in the near future (Fujitsu, 2008).

3.3. *Future Works*

Besides JOI, a few other programming contests for secondary school students are held in Japan. The Supercomputing Contest (SuperCon) is a programming contest for high school students using a supercomputer system at the Tokyo Institute of Technology and/or Osaka University. SuperCon is a team competition, in which teams create a program to solve a given open-ended problem. Unlike the IOI and the JOI, there is no rigid time limit. Teams compete with their ideas and originality to design algorithms to solve a problem

for three or four days. One of the reasons why the Japanese team achieved positive results so quickly upon returning to the IOI is that Japanese students have been competing in SuperCon since 1995. The contest is usually held at the beginning of August. It is a great advantage to have competitions that are held in different seasons and differ in contest format. Therefore, the JCIOI does not intend to hold another contest for top ranked students at the present time.

However, it is necessary to hold contests for students who have little experience and/or are not very familiar with logical thinking. If students without adequate coding skills attend the JOI, their participation will be limited. The JCIOI realize the necessity of "theoretical" or "logical" (by pen and paper) tasks that many national information olympiads adopt. The theoretical or logical tasks are expected to play a key role in improving interest in computer science among secondary school students and teachers. In the first Olympiads in Informatics Conference, there were many fruitful discussions about the running and issues facing several national Olympiads (Dagienė *et al.*, 2007). Especially, the procedures of Brazil (Anido and Menderico, 2007), Italy (Casadei *et al.*, 2007) and Slovakia (Forišek, 2007) have been helpful for the JCIOI. In response to growing needs for logical thinking, the JCIOI has started to prepare a new contest without coding.

There are a lot of challenges facing the JCIOI. The committee intends to address each issue one by one. Possible solutions include cooperation with secondary schools and training secondary school teachers.

4. Conclusions

In this article, the activities for the Japanese Olympiad in Informatics and the IOI contestant selecting procedure have been presented. Some issues and future works have also been described. The main challenges involve providing theoretical background and promoting initiative to generate interest in computer science in secondary schools. The JCIOI will continue to improve its activities to make computer science popular and to encourage gifted secondary school students to be interested in it.

References

Anido, R.O. and Menderico, R.M. (2007). Brazilian olympiad in informatics. *Olympiad in Informatics*, **1**, 5–14.

Casadei, G., Fadini, B. and Vita, M.G.D. (2007). Italian olympiad in informatics. *Olympiad in Informatics*, **1**, 24–30.

Bell, T., Witten, I.H. and Fellows, M. (2005). *Computer Science Unplugged – An Enrichment and Extension Programme for Primary-aged Children*. See also `http://csunplugged.com/`

Dagienė, V., Cepeda, A., Forster, R. and Manev, K. (Eds.) (2007). *Olympiad in Informatics*, **1**.

Forišek, M. (2007). Slovak IOI 2007 team selection and preparation. *Olympiad in Informatics*, **1**, 57–65.

Fujitsu Limited (2008). *Fujitsu Kids*. `http://jp.fujitsu.com/about/kids/`

Japan Science and Technology Agency (2004). *Support for Participating in International Science and Technology Contests*. `http://www.ioi-jp.org/index-e.html`

Japan Committee for International Olympiad in Informatics (2008). `http://www.jst.go.jp/rikai/eng/contest/index.html`

Kanemune, S., Shoda, R., Kurebayashi, S., Kamada, T., Idosaka, Y., Hofuku, Y. and Kuno, Y. (2007). An introduction of "Computer Science Unplugged" – Translation and Experimental Lessons. In *Summer Symposium in Suzuka*, IPSJ SIG-CE (in Japanese).

Supercomputing Contest (2005). `http://www.gsic.titech.ac.jp/supercon/index-e.html`

S. Tani is a director of the Japanese Committee for IOI, and has served as a secretary of Information and System Society of IEICE (the Institute of Electronics, Information and Communication Engineers) since 2008. He received the BSc, MSc and PhD degrees from Waseda University in 1987, 1990 and 1996 respectively. He is currently a professor at Department of Computer Science, Nihon University. His research interests include computational complexity theory, computational topology, and knowledge discovery.

E. Moriya is the president of the Japanese Committee for IOI. He received the BS and PhD degrees from Waseda University, Tokyo, Japan, in 1970 and 1976, respectively. He is currently a professor of mathematics at Department of Mathematics, School of Education, Waseda University, Tokyo. His research area includes formal language and automata theory, and computational complexity theory.

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.

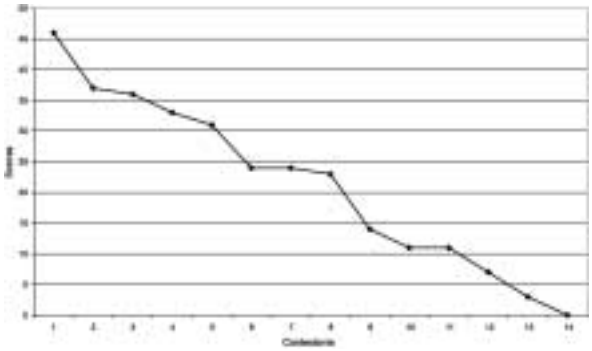


Fig. 1. Distribution of scores for the task IFTHENELSE.

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

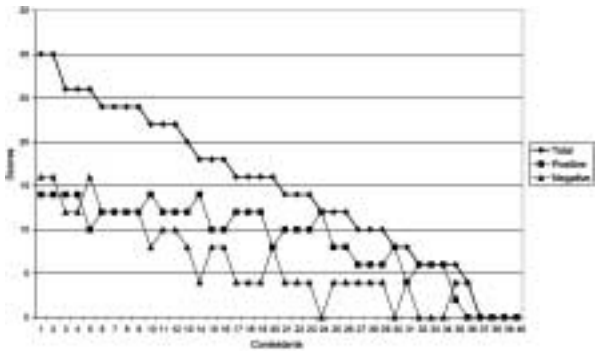


Fig. 2. Distibution of scores for the task RELAY.

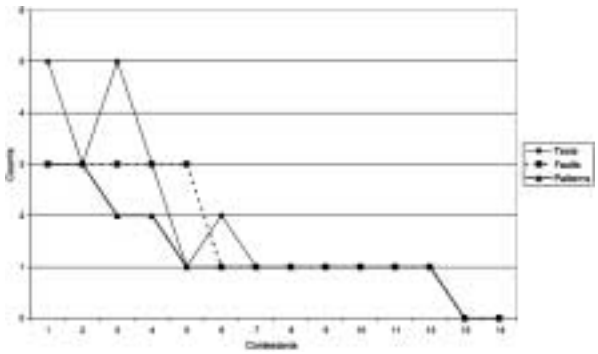


Fig. 3. Results for the task QUICKSORT.

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.

Olympiads in Informatics, 2008, Vol. 2, 181–191

What Do Olympiad Tasks Measure?

Troy VASIGA, Gordon CORMACK
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, N2L 3G1 Canada
e-mail: {tmjvasiga, gvcormack}@cs.uwaterloo.ca

Graeme KEMKES
Department of Combinatorics and Optimization, University of Waterloo
Waterloo, Ontario, N2L 3G1 Canada
e-mail: gdkemkes@math.uwaterloo.ca

Abstract. At all levels of difficulty, the principal focus of olympiad tasks should be problem solving. Implementation complexity, esoteric knowledge requirements and mystery distract from problem solving, particularly for problems deemed to be of low or intermediate difficulty. We suggest criteria for analysing potential tasks and illustrate these criteria by examining several tasks.

Key words: computing competitions, problem solving.

1. Introduction and Motivation

1.1. *Motivation*

At all levels of difficulty, the principal focus of olympiad tasks should be problem solving. Implementation complexity, esoteric knowledge requirements and mystery distract from problem solving, particularly for problems deemed to be of low or intermediate difficulty. We suggest criteria for analysing potential tasks and illustrate these criteria by examining several tasks.

We believe problem solving is important and fundamental to computer science since it satisfies several properties that tasks should aspire to. In particular, tasks should be *attractive*, *approachable*, *challenging* and *practical* (which we also mean *generalizable* or *extendible*). Moreover, when the problem solving aspect of competition tasks is reduced, the remaining elements lessen the attractiveness, approachability, challenge or practicality of the task.

Our thesis is that while it is tempting, one should avoid making problems hard by increasing the information processing aspects of the problem. Similarly it is common to create easy problems by removing the problem solving component, leaving information processing or memorization of algorithms. Neither the information processing nor memorization should overwhelm the problem solving aspects.

Before we proceed to examining these claims, we formally define our terminology.

2. Taxonomy

We now define the terminology we will use throughout the rest of this paper. By *problem solving*, we mean the use of creative, intelligent, original ideas in combination with prior knowledge when applied to a new situation. (A full description of problem solving can be found in (Antonietti *et al.*, 2000).) With respect to computing competition tasks, the "new situation" is the task itself, but the novelty is not sufficient to warrant problem solving; the task itself must have components which would draw upon prior knowledge and also cause creative, original and intelligent new thought processes to occur in the task solver.

In order to further clarify our definition of problem solving, we explicitly define what we *do not* mean by problem solving. For lack of a better term, we will classify the following concepts as *problem solving distractors*. To begin, we consider detailed information processing a distractor to problem solving. By *detailed information processing*, we wish to encompass knowing details of

- particular programming languages or libraries contained therein,
- tedious input/output formatting,
- optimizations that result in constant-factor time improvement in execution speed.

We do acknowledge that there is some information processing required to make a problem solving task into an computer science problem. However, the information processing requirement should be minimized to the largest extent possible in order to keep the crux of the problem solving task as clear as possible.

Another problem solving distractor is *detailed esoteric knowledge of algorithms*, by which we mean memorization of recent journal articles for leading edge algorithms (e.g., recent algorithms from such publications as ACM Symposium on Theory of Computing or IEEE Symposium on Foundations of Computer Science) or memorization of implementation details of complicated algorithms (e.g., implementing red-black trees).

The final problem solving distractor we consider is the distractor of *mystery*, by which we mean hidden aspects of a task or evaluation criteria that a competitor must guess at. Granted, there are two sides to the issue of mystery. Mystery is required, in a moderate sense, since programmers must learn to implement and test their programs with incomplete information. However, since olympiad tasks must be solved under very high time pressures, the challenge of performing deep testing is overwhelming, and thus subtle or even minor errors (typically resulting from information processing distractors) can cause a nearly-correct solution to obtain very few points. Specifically, coding under a veil of mystery is both frustrating and time consuming for the participant. Furthermore, if a task is unreasonably complex, the participant is in effect required to guess at how many marks a partial or partially debugged solution might yield.

When we state that a task should be *generalizable* or *extendible*, we mean that it is straightforward to add dimensions to the problem, remove constraints or decontextualize the problem to make it more abstract. If a task is extendible, it often is *practical*, by which we mean that there are real-world problems that can be solved using solutions to this particular task. As mentioned in (Cormack *et al.*, 2005), making problems practical is an important step to improve the view of the IOI through the eyes of spectators, public

and sponsors. If the tasks we are solving are useful, or at least can be seen to be useful, this would go a long way to improving the perception of the IOI and the discipline as a whole.

3. One Goal, Many Distractors

What is the goal of an olympiad task? Our view is that the goal of an olympiad task is to solve the task. By solving the task, we mean being able to communicate the underlying algorithmic process that will consume input in the desired way to produce the correct output. The core concept in solving a task is the application of problem solving skills. Unfortunately, solving a task is conflated with the problem solving distractors outlined in Section 2. Specifically, students need to pay particular attention to information processing details, such as

- whether to return 0 at the end of their main procedures in C/C++ ;
- dealing with newline characters in their input or output;
- dealing with file input and output nuances in particular languages;
- knowing particular libraries, and more specifically, the efficiency of particular libraries. For example, knowing runtime differences between `cin` and `printf` in C++.

These distractors point towards what we believe is a significant problem with IOI tasks. At the IOI, optimal efficiency of a correct solution tends to be the overriding goal of IOI tasks. Certainly, competitors must meet the constraints outlined in the problem statement. However, the goal of IOI tasks seems to be finding "the best" algorithm, going so far as to distinguish between $\mathrm{O}(N \log \log N)$ and $\mathrm{O}(N)$ solutions. This focus on efficiency (and how to measure it accurately) has been the main reason Java has not been added to the set of allowed languages at IOI: specifically, see the technical reasons stated in (Nummenmaa and Kolstad, 2006). This disproportionate focus on efficiency has led to many concerns with IOI tasks, such as:

- competitors guessing what non-solutions get the most marks: see for example evaluation of IOI tasks (Forisek, 2006) that have allocated an inappropriate percentage of marks for incorrect algorithms;
- the speed of programming being a factor in the scores of IOI competitors;
- the competitor who has memorized the "correct" set of algorithms has a clear advantage over those competitors who have broader algorithmic abilities.

In the next section, we discuss ways of mitigating these distractors which can move the goal of IOI tasks away from ultra-efficiency and closer to pure problem-solving.

4. Minimizing Distractors

The distractors outlined in the previous two sections tend to move the particular task away from the problem solving aspects and also artificially inflate the difficulty of the task. An

unfortunately corollary of these distractors is that when a task is deemed difficult, there is a tendency to eliminate the problem solving aspect to make the task easier, which results in having a task with an even higher proportion of distractors, or, in the worst case, a problem involving only distractors.

Further, the distractors may not be in the task itself, but some distractor external to the task, such as the programming environment in particular.

We propose the following solutions that minimize or mitigate the distractors described above.

1. **Provide feedback during the competition.** As outlined in (Cormack *et al.*, 2005), feedback on submissions for competitors can provide many benefits. For the purposes of this paper, the key benefit feedback provides is the minimization of mystery.
2. **Consider using output-only tasks.** Output-only tasks reduce the restrictions imposed by programming languages, leaving the idea of problem-solving as a remnant. There are several flavours of output-only tasks that can be used. One flavour, is to solve some small or restricted case of a problem by hand (as mentioned in (Cormack *et al.*, 2005)). Another variation on output-only tasks is to ask a theoretical question, such as *what is the minimum number of times event X can occur if some other event Y must occur in some previous step*. Yet another variation is for the "solution" to be a test case which examines some constraint of the problem: for example, what is the largest amount of money that cannot be formed using a set of coins in denominations 4 and 9? To state the benefit of output-only tasks another way, output-only tasks remove all information processing in a CPU sense and recast it in a *mental information processing* sense, which should be the core idea of informatics tasks.
3. **Provide a syllabus.** Providing a syllabus ahead of the competition will remove mystery and focus the competitors on applying problem solving skills of synthesis and creativity in their solutions, rather than rote learning and memorization of a correct set of algorithms. An example proposed syllabus in the IOI context is (Verhoeff *et al.*, 2006).
4. **Provide a standard IOI library of tools.** If multiple languages are allowed in a competition, and if the ability to measure efficiency of solutions matters, then providing an STL-like library, where students call particular routines with known runtime, mitigate the language-specific distractors. Specifically, a standard library will make solutions much less language dependent, opening up the possibility of additional languages being introduced at the IOI, and it will also mitigate information processing details that arise from input and output.
5. **Provide practice problems that test all input/output requirements.** If there is no library available for the students to use, and thus, they must use the input/output of their particular language environment, they should be told precisely the format of the input and output well before the actual competition, in order to move the implementation problems to an off-line ahead-of-time concern, not part of the task.
6. **Simplify the input format.** As specific examples, we mean tasks should eliminate file I/O, and eliminate multiple test cases from individual files. We are not advocat-

ing for the removal of batch testing. Rather, batch testing can be done by grading systems and test schemes that group test cases together. Standard input may be a good format of input, if we wish to be open to many different tools (this is what the ACM ICPC uses). The simplification of input will mitigate detailed information processing.

7. **Simplify the output format.** It is unclear what is the relative difficulty of returning a value, or outputting to a file, or outputting to the screen or outputting to standard output. Thus, make the output format as simple as possible by not relying on multiple streams/types of output (i.e., do not rely on both correct return values and correct output to files) or relying on multiple pieces of output. Simplified output will minimize detailed information processing.
8. **Ensure the problem statement is short.** Information processing does not simply involve processing information via the computer: the less reading the student needs to do, the more the student can focus on the problem solving of the task itself. Moreover, it is generally the case that tasks with long descriptions either have very little problem solving in them (i.e., they are using information processing in the task description as a distractor) or there is a significant amount of information processing required (i.e., the task is extremely complicated but perhaps not intellectually demanding).
9. **Ensure that solutions are short.** It should go without saying that tasks should have solutions written for them before they are given to competitors. If the solution for a task is longer than 50 lines of C code or if it requires use of the STL or other library functions, the task should be met with great suspicion.
10. **Attempt to make tasks practical, or show how they may be extended to practice.** Most problems in the "real world" have fairly simple descriptions, simple input, and simple output but have incredible problem solving cores. One example of such a problem is finding a subsequence in a DNA string: here the input is defined by a string consisting of four different letters, the output is a boolean value, yet the computational and problem solving core is very rich.
11. **Piloting the task.** We have mentioned that tasks should have solutions written for them before they are given to competitors. Furthermore, the solutions should be written by people other than those that created the problem. While the existence of a solution to a task is beneficial, we argue that the "pilot" (tester) of a task should also ensure that the points 6–10 above are satisfied to the greatest extent possible. Thus, we propose a checklist that includes the items shown in Fig. 1.

5. Examples

We now consider applying the ideas introduced in Section 4 to various examples in order to highlight the beneficial analysis and framework the points from Section 4 can provide.

By way of contrast, we will present a seemingly simple task description, and highlight how implementation details can make a task which is of poor quality. We begin by

- Do you understand the problem statement?
- Is there extra information in the problem statement that can be deleted?
- Can some descriptions in the problem tasks be simplified or clarified?
- Do you know how to solve the task?
- What do you consider the components of the task to be?
- If you solved it, was the programming effort tedious? What were the implementation (rather than problem solving) challenges you faced?
- Was your solution fewer than 50 lines of code?
- Describe your thought process in solving the problem. What false starts or incorrect attempts did you encounter while solving the problem?
- Can the input format be simplified?
- Can the output format be simplified?
- Can you imagine problems/circumstances/issues where this task may be generalized to?

Fig. 1. The pilot's checklist.

considering a simple, abstract task of "Read in a list of numbers and maintain two heaps, one a max-heap and one a min-heap." The implementation details involve maintaining pointers/references between all nodes in both trees, and successively updating each tree for each operation that is implemented. Notice that the problem solving aspect here is quite simple, and the task description is very short, but there is a tremendous amount of implementation detail to be worked out. Moreover, if there is an error in the implementation, even though the problem has been solved, there may be a significant amount of difficulty in debugging such errors.

For the remainder of this section, we focus on actual tasks that were used in competitions. For copyright reasons and to avoid negative implications of our analysis, we focus on problems that have been used either at the Canadian Computing Competition or local training contests used at the University of Waterloo for ACM ICPC team selection.

5.1. *Dick and Jane*

(This task was used at the University of Waterloo local competition, June 1998 (Cormack, visited 2008).)

Dick is 12 years old. When we say this, we mean that it is at least twelve and not yet thirteen years since Dick was born.

Dick and Jane have three pets: Spot the Dog, Puff the Cat, and Yertle the Turtle. Spot was s years old when Puff was born; Puff was p years old when Yertle was born; Spot was y years old when Yertle was born. The sum of Spot's age, Puff's age, and Yertle's age equals the sum of Dick's age (d) and Jane's age (j). How old are Spot, Puff, and Yertle?

Each input line contains four non-negative integers: s, p, y, j. For each input line, print a line containing three integers: Spot's age, Puff's age, and Yertle's age. Ages are given in years, as described in the first paragraph.

5.2. *Analysis of "Dick and Jane"*

This task has a short problem statement, short solution, but could be improved by reducing the multiple inputs in files. If libraries were given to the competitors, variables

could be eliminated by having values in a method call to programs. As a practical extension, students could learn something about linear programming generally, or integer linear programming in particular. It is worth noting that there may appear to be two pieces of redundant information in the task description: in fact there is only one redundant item. The parameter d is redundant, in that Dick's age of 12 is given in the problem statement. However, the parameters s, y and p are all necessary, since there are many "off-by-one" boundary cases that are meant to be considered by this task. This task would satisfy the majority of the Pilot's Checklist (shown in Fig. 1) and thus, would be a very good task.

5.3. *Space Turtle*

(This task was used at Canadian Computing Competition, Stage 2, 2004. (Canadian Computing Competition Committee, visited 2008)).

Space Turtle is a fearless space adventurer. His spaceship, the *Tortoise*, is a little outdated, but still gets him where he needs to go.

The *Tortoise* can do only two things – move forward an integer number of light-years, and turn in one of four directions (relative to the current orientation): right, left, up and down. In fact, strangely enough, we can even think of the *Tortoise* as a ship which travels along a 3-dimensional co-ordinate grid, measured in light-years in directions parallel to co-ordinate axes. In today's adventure, Space Turtle is searching for the fabled Golden Shell, which lies on a deserted planet somewhere in uncharted space. Space Turtle plans to fly around randomly looking for the planet, hoping that his turtle instincts will lead him to the treasure.

You have the lonely job of being the keeper of the fabled Golden Shell. Being lonely, your only hobby is to observe and record how close various treasure seekers come to finding the deserted planet and its hidden treasure. Given your observations of Space Turtle's movements, determine the closest distance Space Turtle comes to reaching the Golden Shell.

Input

The first line consists of three integers *sx*, *sy*, and *sz*, which give the coordinates of Space Turtle's starting point. Each of these integers is between -100 and 100. Space Turtle is originally oriented in the positive x direction, with the top of his spaceship pointing in the positive z direction, and with the positive y direction to his left. The second line consists of three integers *tx*, *ty*, and *tz*, which give the coordinates of the deserted planet. Each of these integers is between -10000 and 10000.

The rest of the lines describe Space Turtle's flight plan in his search for the Golden Shell. Each line consists of an integer d, $0 \leqslant d \leqslant 100$, and a letter c, separated by a space. The integer indicates the distance in light-years that the *Tortoise* moves forward, and the letter indicates the direction the ship turns after having moved forward. 'L', 'R', 'U', and 'D' stand for left, right, up and down, respectively. There will be no more than 100 such lines. On the last line of input, instead of one of the four direction letters, the letter 'E' is given instead, indicating the end of today's adventure.

Output

Output the closest distance that Space Turtle gets to the hidden planet, rounded to 2 decimal places. If Space Turtle's coordinates coincide with the planet's coordinates during his flight indicate that with a distance of 0.00. He safely lands on the planet and finds the Golden Shell.

5.4. *Analysis of "Space Turtle"*

There is a rather long-winded story in the description. While there can be circumstances where a story enhances the situation, other times it can be distracting. For this task, the story is a distractor. The input description is quite complicated, involving various types of data, and the description of the last line is ambiguous: it is unclear if there is a number before the letter 'E' or not. If there is no number before the letter 'E' then reading of such input is sophisticated. Output was constrained to a single value to make the problem easier to mark, but since the output did not show a derivation it was difficult to assign partial credit for students who made small mistakes. The simplified output also, in fact, complicated the problem statement. The solution to this problem is quite short, involving only a few simple arithmetic transformations during each iteration of a loop. This task can be generalized by relaxing the requirement of 90 degree motion. This task is an introduction to the rich subject area of three-dimensional geometry.

5.5. *Pinball Ranking*

(This task was used at the Canadian Computing Competition, Stage 1, 2005 (Canadian Computing Competition Committee, visited 2008)).

Pinball is an arcade game in which an individual player controls a silver ball by means of flippers, with the objective of accumulating as many *points* as possible. At the end of each game, the player's score and rank are displayed. The score, an integer between 0 and 1 000 000 000, is that achieved by the player in the game just ended. The rank is displayed as "r of n". n is the total number of games ever played on the machine, and r is the position of the score for the just-ended game within this set. More precisely, r is one greater than the number of games whose score exceeds that of the game just ended.

You are to implement the pinball machine's ranking algorithm. The first line of input contains a positive integer t, the total number of games played in the lifetime of the machine. t lines follow, given the scores of these games, in chronological order. Input is contained in the file `s5.in`.

You are to output the average of the ranks (rounded to two digits after the decimal) that would be displayed on the board.

At least one test case will have $t \leqslant 100$. All test cases will have $t \leqslant 100000$.

5.6. *Analysis of "Pinball"*

The problem solving essence of this task is to maintain rank statistics in a tree. However, the efficiency considerations cause the implementation details and memorization of

esoteric algorithms (i.e., range trees, balanced trees) to overwhelm the problem solving core. Also, notice that the output was forced to be a summary statistic, since the marking was done by teacher in classrooms (not in an automatic fashion). This constraint should be altered to give a trace of each step. This would provide both better traceable code for students (should they encounter an error in the program) and also remove one extra layer of obfuscation that hides the problem solving skill.

A possible alteration to provide a library would be to create an API consisting of a set of scores (S) where we have operations

- S = addScore(S, newScore)
- medianScore(S)
- subsetGreaterThanMedian(S)
- subsetLessThanMedian(S)
- count(S)

These operations remove knowledge of pointers, and remove the advantage of prior tailored experience.

The test suite for this problem was spoofable by using poor heuristics, and this was found out only after reviewing student submissions.

This problem is equivalent to a hard problem at IOI, but unfortunately, this problem has multiple flaws based in its current form.

5.7. *Long Division*

(This task was used at the Canadian Computing Competition, Stage 1, 1997 (Canadian Computing Competition Committee, visited 2008)).

In days of yore (circa 1965), mechanical calculators performed division by shifting and repeated subtraction. For example, to divide 987654321 by 3456789, the numbers would first be aligned by their leftmost digit (see Example 1), and the divisor subtracted from the dividend as many times as possible without yielding a negative number. The number of successful subtractions (in this example, 2) is the first digit of the quotient. The divisor, shifted to the right (see Example 2), is subtracted from the remainder several times to yield the next digit, and so on until the remainder is smaller than the divisor.

EXAMPLE 1.

```
     987654321
-    3456789    first successful subtraction
     ========
     641975421
-    3456789    second successful subtraction
     ========
     296296521  remainder
-    3456789    unsuccessful subtraction
     ========
     negative
```

EXAMPLE 2.

```
          296296521
      -    3456789
         ========
         261728631
         etc.
```

(a) Write a program to implement this method of division. See the input and output specifications below.

(b) If the dividend is n digits long and the divisor is m digits long, derive a formula in terms of n and m that approximates the maximum number of single-digit subtractions performed by your program.

Input Specification:

The first line of the input file contains a positive integer n, $n < 20$, which represents the number of test cases which follow. Each test case is provided on a pair of lines, with the number on the first line being the dividend, and the number on the second line being the divisor. Each line will contain a positive integer of up to 80 digits in length.

Output Specification:

For each pair of input lines, your output file should contain a pair of lines representing the quotient followed by the remainder. Output for different test cases should be separated by a single blank line. Your output should omit unnecessary leading zeros.

5.8. *Analysis of "Long Division"*

This task has a tedious, heavily information processing-based solution. The specific information processing issues are output with certain spaces, line breaks, multiple test cases in each file. Additionally, the output can be spoofed by simply performing the division and remainder using built-in operators in some programming languages. The problem statement is short, and the sub-task involving the number of operations is a very good problem solving question.

On balance, if the information processing issues were mitigated, there is an interesting problem solving core here, which can be extended and generalized to provide a good task. Specifically, the description of the division algorithm should be simplified and incorporated with more text, the input should be simplified to a single test case and the output should be modified to output the number of subtractions necessary for the entire process.

6. Conclusion

The goal of olympiad tasks should be to measure the problem solving abilities of competitors. Unfortunately, this goal is often hard to attain, due to distractors such as detailed information processing, mystery and esoteric prior knowledge of algorithms. In this paper, we have attempted to provide suggestions for improving competition environments

(such as using libraries, syllabi, feedback during competition) for mitigating distractors outside of tasks, as well as providing a methodology for analyzing tasks that will minimize distractors from within particular tasks. It is our hope that improving both the environment in which tasks are written and the tasks themselves will result in olympiads where students are better motivated, tasks are easier to understand and evaluators capable of more accurately measuring the problem solving abilities of competitors.

References

Antonietti, A., Ignazi, S. and Perego, P. (2000). Metacognitive knowledge about problem-solving methods. *British Journal of Educational Psychology*, **70**, 1–16.

Canadian Computing Competition Committee.
`http://www.cemc.uwaterloo.ca/ccc/`

Cormack, G. *University of Waterloo Local Competitions Pages*.
`http://plg1.cs.uwaterloo.ca/~acm00/`

Cormack, G., Munro, I., Vasiga, T. and Kemkes, G. (2005). Structure, scoring and purpose of computing competitions. *Informatics in Education*, **5**, 15–36.

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

Nummenmaa, J. and Kolstad, R. (2006). Java in the IOI. *IOI Newsletter*, **6**.
`http://olympiads.win.tue.nl/ioi/oed/news/newsletter-200606.pdf`

Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A proposal for an IOI Syllabus. *Teaching Mathematics and Computer Science*, **4**, 193–216.

G.V. Cormack is a professor in the David R. Cheriton School of Computer Science, University of Waterloo. Cormack has coached Waterloo's International Collegiate Programming Contest team, qualifying ten consecutive years for the ICPC World Championship, placing eight times in the top five, and winning once. He is a member of the Canadian Computing Competition problem selection committee. He was a member of the IOI Scientific Committee from 2004–2007. Cormack's research interests include information storage and retrieval, and programming language design and implementation.

T. Vasiga is a lecturer in the David R. Cheriton School of Computer Science at the University of Waterloo. He is also the director of the Canadian Computing Competition, which is a competition for secondary students across Canada, and has been the delegation leader for the Canadian Team at the International Olympiad in Informatics. He is the chair of the IOI 2010 Committee.

G. Kemkes has participated in computing contests as a contestant, coach, and organizer. After winning a bronze medal at the IOI and two gold medals at the ACM ICPC, he later led and coached the Canadian IOI team. He has served on the program committee for Canada's national informatics olympiad, the Canadian Computing Competition. Kemkes is currently writing his PhD thesis on random graphs in the Department of Combinatorics & Optimization, University of Waterloo.

Olympiads in Informatics, 2008, Vol. 2, 192–207

Programming Task Packages: Peach Exchange Format

Tom VERHOEFF

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands
e-mail: t.verhoeff@tue.nl

Abstract. Programming education and contests have introduced software to help evaluation by executing submitted taskwork. We present the notion of a *task package* as a unit for collecting, storing, archiving, and exchanging all information concerning a programming task. We also describe a specific format for such task packages as used in our system *Peach*, and illustrate it with an example. Our goal is to stimulate the development of an international standard for packaging of programming tasks.

Key words: programming education, programming contest, programming task, task package, grading support software, data format.

1. Introduction

Programming education and contests have introduced software to help evaluation by executing submitted taskwork. Typically, a programming task is understood to be a short text that describes the requirements on the program to be constructed. Such task descriptions can be found in various problem archives, such as the (UVa Online Judge, 2008). However, it takes more than just its task description to be able to (re)use a programming task. In this article, we present the notion of a *task package* as a unit for collecting, storing, archiving, and exchanging task-related information. Ideally, such task packages can be dropped into your favorite programming education and contest hosting system to configure it for the task.

We have used task packages for many years now in our programming education and contest hosting system, called *Peach* (Peach, 2008). Using these task packages has helped us ensure completeness and correctness of task data.

We will discuss the contents and format of task packages and also the interface and operations for task packages. Particular concerns are the support for

- multiple languages to express human readable texts (including, but not restricted to, the task description);
- multiple programming languages allowed for solving the task;
- multiple platforms to deploy tasks;
- diverse task styles;

- validation of package content;
- handling of relationships between tasks, e.g., where one task is a variant of another task;
- flexibility to allow easy incorporation of changes, such as changing the task's name.

2. Package Contents

In this section, we discuss the various pieces of information that (could) belong in a task package. As a running example, we use the task *Toy Division*. Here is its task description:

TOY DIVISION

PROBLEM

K kids together receive T toys. They wonder whether it is possible to divide these toys *equally* among them, that is, with each kid obtaining the same number of toys and no toys remaining. If this is possible, they also want to know how many toys Q each of them receives.

Write a program to answer the questions of these kids.

INPUT

A single line is offered on standard input. It contains two the numbers K and T, separated by a space.

OUTPUT

The answer must be written to standard output. The first line contains the string `'Yes'` if it is possible to divide the toys equally, and `'No'` otherwise. If equal division is possible, then a second line must be written, containing the number Q of toys each kid obtains. If there are multiple answers, then it does not matter which answer your program writes.

CONSTRAINTS

K, T, and Q are non-negative integers, less than 10^8. The execution time limit is 0.1 s.

EXAMPLE

Standard Input
`3 15`

Standard Output
`Yes` `5`

SCORE

There are 10 evaluation cases. Each completely solved case gets 10 points. Incomplete outputs get 0 points; there is no partial score.

(END OF TASK DESCRIPTION)

The following kinds of task-related information can be distinguished:

- textual information in natural language, such as the task description;
- data files, such as input data for evaluation runs;
- configuration parameters, such as resource limits;
- tools, such as a task-specific output checker;
- scripts, e.g., to build an executable or coordinate the evaluation process;
- submissions (good and bad), as exemplary solution and to help verify everything;
- metadata, e.g., classification of task type; some metadata could be textual.

It is good to be aware of the various **stakeholders** of task-related information. The terminology often depends on the setting.

Supervisor makes management-level decisions, e.g., about the task name, set of tasks to use together, presentation style, etc.; could also be called **owner**. In an educational setting, this role is played by the *teacher*; in a contest setting, by the *contest director* or *chair of the scientific committee*.

Author creates the task; makes technical decisions about the task; often needs to carry out various experiments to explore alternatives and tune parameters.

Solver attempts to carry out the task, i.e., solve the stated problem, resulting in work to be submitted for evaluation; could also be called *performer*. In an educational setting, this is the *student* participating in a course; in a competition, it is the *contestant*. Keep in mind that the solver is the primary stakeholder.

Grader is involved in evaluating the submitted work for a task. In an educational setting, this is often a teaching assistant; in a competition, this is nowadays often supported by an automated grading system, which is administered by a *grading system administrator*, who configures the system for specific tasks. Note that also *developers of automated grading systems* are to be considered as stakeholders.

Mediator helps interpret evaluation results. In an educational setting, this could be done by an *instructor*, sometimes *parents* play this role; in a competition, this is done by a *coach* or *team leader*.

Trainer helps in preparing the solver, e.g., through directed practicing. In an educational setting, this is often the job of a *teaching assistant*; in a contest setting, there are typically several trainers, each with their own area of expertise. It is also possible that solvers practice by themselves. This stakeholder has a specific interest in the ability to *reuse* tasks easily.

Researcher investigates tasks, submitted work, and evaluation results; this could be for historic reasons, but also to help improve education and competition practices, or with a purely scientific interest. This stakeholder is helped by consistent and complete archiving of tasks, similar to what (Sloane's OEIS, 2008) does for integer sequences.

2.1. *Textual Information*

The **task description** contains all information (or pointers to such information) that the task author wants to offer to the students or contestants confronted with the task. In (Verhoeff, 1988) some guidelines for creating programming problem sets are presented. The task description must specify unambiguously, precisely, consistently and completely what qualifies as an acceptable solution. Typically, each participant submits work on the task in the form of (one or more) files. For files with program code, the task description states the interfaces and the relevant contracts (assumptions and obligations), and provides at least one example.

Besides a task description the following texts are useful to include in a task package.

Hints In an educational setting, we find it useful at times to include a separate text with hints, which can be disclosed to participants under appropriate circumstances. Even for contests, it can be useful to have separate hints.

Background information This includes, for instance, historic and technical information (algorithms, data structures), and motivation for particular features, e.g., their relationship to a syllabus (Verhoeff *et al.*, 2006).

Grading information That is, information that concerns the process of grading submitted work for this task. In an educational setting, this could involve instructions for assistants doing the actual grading. In a contest setting, it could cover information useful in understanding the grading result.

Notes These are short texts about the contents of other items, for instance, summarizing the purpose of the task, evaluation cases, and test submissions, and they can be used to generate various overviews (also see Section 3 about operations on task packages). These notes belong to the category of *metadata*, which can include machine-readable information as well (treated in Section 2.7).

The hints for our example are:

> HINTS FOR TOY DIVISION
> Consider the integer equation $T = K * Q$ with unknown Q.
> What special cases are there?
> (END OF HINTS)

Some background information for the example:

> BACKGROUND INFORMATION FOR TOY DIVISION
> The purpose of this task is to present a simple, nontrivial programming problem. The cases $K = 0$ could be eliminated to simplify it even further, by constraining the input to $K > 0$.
> It has proven to be a good exercise in careful reading of specifications, and the use of integer division, including the modulo operator.
> (END OF BACKGROUND INFORMATION)

Some grading information for the example:

GRADING INFORMATION FOR TOY DIVISION

The proper case distinction needs to be made. For $K = 0$, the equation $T = K * Q$ degenerates to $T = 0$. That is, equal division is not possible if $T \neq 0$, it is possible if $T = 0$, in which case any Q (in range) will do.

For $K \neq 0$, the equation $T = K * Q$ is a linear equation in Q. It is solvable in integers if and only if K is a divisor of T, or, alternatively, if $T \bmod K = 0$. In that case, Q is uniquely determined by $Q = T/K$.

Manual graders should judge the layout, comments, names, and idiom. Particular points of further attention are:

- avoiding division by zero;
- use of 32-bit arithmetic (or better);
- use of integer division, rather than floating-point division (inaccurate) or repeated subtraction (too slow).

#	K	T	Q	Remarks
1	0	0	*	The only divisible case with $K = 0$
2	0	1	No	Smallest T with $K = 0$ that is not divisible
3	0	99999999	No	Largest T with $K = 0$ that is not divisible
4	1	0	0	Smallest case with $K > T = 0$
5	1	99999999	99999999	Largest Q that is divisible
6	2	1	No	Smallest case with $K > T > 0$
7	2	65536	32768	$T = 2^{16}$, fails with 16-bit arithmetic
8	99999998	99999999	No	Largest T, and K, that are not divisible
9	99999999	99999998	No	Largest K, and T, that are not divisible
10	99999999	99999999	1	Largest K and T that are divisible

Legend:

Identifier of evaluation case

K Number of kids (input)

T Number of toys (input)

Q Number of toys per kid (quotient), if equally divisible, else No (output)

* indicates that any value Q satisfying $0 \leqslant Q < 10^8$ is correct.

(END OF GRADING INFORMATION)

Each of these texts could be available in several languages. In our educational setting, we often have material both in Dutch and in English. In contests like the International Olympiad in Informatics (IOI, 2008), task descriptions are translated into the native language of the contestants, resulting in dozens of versions of the same information, whereas background and grading information is often presented in English only. Note that the tabular overview of evaluation cases in the grading information would ideally be generated from the actual evaluation data and summary texts (metadata).

A general concern with 'plain' text files is their **encoding**. For simple texts, ASCII suffices, but especially for non-English texts, additional characters are desirable. We recommend the use of **UTF-8** (RFC 3629, 2003), one of the *Unicode* standards.

Many texts, however, will not be written in a 'plain' text format, but some other format. Some relevant open format standards are:

- LaTeX, TeX, especially suited for texts involving mathematics (CTAN, 2008);
- OpenDocument, used by OpenOffice (OpenOffice, 2008);
- (X)HTML, used in web browsers (W3C, 2008);
- DocBook (DocBook, 2008);
- reStructured Text, used by Docutils (Docutils, 2008);
- various wiki formats.

Each of these open formats may have multiple variants. Note that these formats are aimed at flexible text entry and editing. They can be converted into various (open) presentation formats, such as PDF.

One should also be aware of the need for **version control** on texts. This issue is addressed further in Section 4.

2.2. *Data Files*

Besides human-readable texts, a task can also involve various other files, in both text or binary format. We call them data files, even though they could well be source files with program fragments, such as an interface definition for a library. These files could be part of the material that the solver receives along with the task description, but they could also be related to evaluation. Here is an overview:

- Data files accompanying the task description, possibly including source files. In our educational setting, we sometimes have assignments where we provide a program text 'with holes' to be completed by the students. Such a source file with holes is created by a generic tool on the basis of our own solution with special 'hole markers' in comments:

```
//# BEGIN TODO body of method TPointSet.Count
  ... author's solution, to be suppressed ...
//# END TO DO
```

- Input data for evaluation runs; per run there could be several files;
- Expected output data for evaluation runs of deterministic[1] programs, possibly modulo some simple equivalence relation. The equivalence could concern white space, upper versus lower case characters, the order of numbers in a set, etc.;
- Auxiliary data used in evaluation runs of nondeterministic programs. This could concern parts of the output that are deterministic, or some description of the expected output, e.g., in the form of a regular expression;
- Input data for tools that generate other files, such as large input for evaluation.

It is important that one can motivate the choice of data. A haphazard collection of inputs does not make a good grading suite. Make it a habit to write a *note* for each data file, summarizing its *purpose* (as opposed to its contents; for the latter, generator input or characterizer output is more useful, see Section 2.4 on tools). Such notes can be (partially) included in tabular overviews of data sets. This is especially useful for larger data sets. The overview can be attached to the grading information.

There are some platform-related issues to keep in mind:

[1] By deterministic we mean that the input uniquely determines the output.

End-of-line markers In text files, the end of a line is marked by a special character or combination of characters, depending on the platform. Unix uses a *line feed* (LF), Mac OS uses a *carriage return* (CR), and Windows uses the combination CRLF. This is particularly relevant for files made available to the solver, and files directly used in evaluation (e.g., to compare to the output of an evaluation run).

Byte order In binary files, the ordering of bytes in some multi-byte data items (such as numbers) may vary between the platforms. The two main flavors are *big-endian* and *little-endian*. The concerns are similar to those for end-of-line markers.

2.3. *Configuration Parameters*

The data files discussed in the preceding section play a specific role in grading the functionality requested in the task: input(-related) data and output(-related) data.

A task can specify more than just functionality, It can, for instance, also impose performance requirements. Such requirements are often expressed in terms of **resource limits**. In particular, the following resources have been limited:

- *size of submission* (total size of source files);
- *build time* (total time allowed for compilation and linking);
- *memory* (RAM, hard disk space);
- *execution time* (total run time, or response time to specific events);
- *number of times* that certain resources may be used, for instance, that some function in a library may be called.

Other things that can be treated as configuration parameter are: *compiler options* and *linker options*.

Such configuration parameters are intrinsic to the task, and are sometimes – but not always – communicated explicitly to the solver in the task description. They also need to be taken into account during evaluation runs. For automatic grading and for later analysis and comparison of tasks, it is useful to include configuration parameters in a task package in a *machine readable* way. They should be easy to tune at a late stage.

Note, however, that the meaning of such parameters depends on the actual platform used for evaluation runs. Platform information is discussed in Section 2.7 about metadata. Also, it is imaginable that not all evaluation runs use identical parameter values.

2.4. *Tools*

When solving a task and when evaluating work submitted for a task, various generic software tools are needed. Most notably these include editors, compilers, libraries, linkers, loaders, debuggers, file comparators, etc. Generic tools are discussed in Section 2.7 along with metadata.

There is often also a need for task-specific tools. These are to be developed by the task author (or assistants). One can think of the following kinds of task-specific tools:

Input generator to produce specific input cases, for instance large cases with special properties. Use of an input generator also helps ensure that valid data is created.

Input validator to check that input files satisfy the assumptions stated in the task description. These assumptions often include **format** restrictions: what types of input data appear in what order and in what layout (i.e., distribution over lines); but also concern **value** restrictions: range limits on values, specific relationships between values (e.g., a graph that needs to be connected).

Input data files need to be of high quality, and one should not simply assume that they are valid (unless they are automatically generated maybe, but even then it is useful to have the ability to independently check their validity). The application of an input validator needs to be automated, because otherwise it will not be used when it is most needed, viz. under pressure when last-minute changes are made. Also see Section 3 about package operations.

Output format checker to check that output conforms to the format requirements of the task. This tool can be given to the solver to help achieve the correct output format. Note that this tool will not give information about the actual correctness of the output. It can also be used during evaluation as a filter to ensure that a tool that checks for correctness does not have to deal with badly formatted data.

Input/output characterizer to summarize characteristics of data files, in particular, to generate, from actual input and output data files the tables appearing in the grading information. Such summaries are useful in determining to what extent evaluation runs cover various 'corners' of the input space. Doing this by hand is cumbersome and error prone.

Expected-output generator to produce expected output on the basis of given input data. This is useful when a task is (almost) deterministic. Note that in most cases a solution to the task can act as expected-output generator. But it need not satisfy the task's performance requirements; it can be run in advance (or afterwards) and even on a different platform.

Output checker to check that output[2] generated in an evaluation run corresponds with the input data in accordance with the requirements stated in the task description. An output checker takes as inputs the input data file, the output data file produced in the evaluation run, and sometimes also some preprocessed data (to avoid the need for recalculating certain information, e.g., concerning deterministic parts of the output; that way the checker can be kept smaller and more generic).

This applies especially to nondeterministic tasks. In case of a deterministic task, output checking can be done generically by comparing actual output to expected output, possibly modulo some simple equivalence relation.

Evaluation drivers and/or stubs to be combined with submitted program fragments to build executables used in evaluation runs. In particular, if the task does not require the solver to submit a main program (but, for instance, a module or library), then the task author needs to provide a main program (or more than one) to act as an evaluation driver of the submitted module or library. And, conversely, when the

[2] Occasionally, also the order of *input-output interleaving* needs to be checked.

task requires the solver to submit a main program with one or more holes (e.g., in the form of a pre-defined module or library), then the author may need to provide evaluation stubs to fill these holes.

Not every task will need each of these task-specific tools.

Such tools need to incorporate task-specific knowledge. Often it is a good idea to create a separate library with task-specific facilities (data types and related operations), rather than duplicating such definitions in each tool. Duplication hinders future changes, especially when a task is still under development.

Some tools can be combined, though this is not advisable. It is better to refactor common functionality into a task-specific library. For instance, an input/output characterizer needs to read input and output files, and so could also report on the validity of their format and contents. But combined functionality complicates the interface of the tool, and increases the risk that changing one piece of functionality will also (adversely) affect other pieces.

There is an opportunity to use generic libraries for functionality common to multiple tasks. For instance, **RobIn** (Verhoeff, 2002) was developed to assist in the construction of input validators and output (format) checkers, by providing some simple operations for *robust input*, that is, without making any assumptions about the actual input. RobIn was used by the author at IOI 2000 in China to validate the input files.

2.5. *Scripts*

Besides task-specific tools, there will also be various task-specific scripts. Tools concern task-specific technical operations, whereas scripts are more for management and for coordinating the application of task-specific tools. Scripts can

- coordinate the entire grading process of a submission for the task, involving such steps as
  1) preprocessing of submitted work,
  2) building various executables,
  3) running executables with various inputs, capturing output and interleaving,
  4) evaluation of the outputs of each run according to various criteria,
  5) scoring to obtain numeric results,
  6) reporting to present and summarize all results.

  Such a grading script should be runnable by a daemon in an automated grading system, but also by a task author or human grader in a stand-alone situation; a task author may want to explore how a particular submission is handled, and a human grader (teaching assistant) may want to re-evaluate a submission locally under several what-if scenarios by making manual changes;
- coordinate the generation of all evaluation data;
- generate various overviews;
- generate an archive of material to be presented to solver, especially when this consists of more than just the task description;
- validate the package contents, by evaluating all test submissions and checking the results.

2.6. *Solution and Test Submissions*

A task author not only needs to invent and describe the task, specify how it will be graded, and provide data and (where applicable) task-specific tools, but also needs to write an **exemplary solution** worthy of imitation. This solution is needed for pedagogical reasons, and it also serves as a test for the grading data and tools. However, package testing should not end there. In fact, solutions are needed in *all allowed programming languages*. Of course, the grading tools and data should also be tested with **imperfect solutions**, to check that these are graded in agreement with the intentions.

These test submissions (ranging from good to bad) belong in the task package, and must be used to validate the package contents and in particular, the entire grading chain. They also provide a means to test the installation of a package on a particular grading system. There should be sufficient variety in submissions to ensure a broad coverage.

As with data files, it is recommended to include a separate *note* with each test submission, motivating its purpose. These notes can be summarized in a tabular overview, together with actual and expected grading results.

The work submitted by solvers, when this task is actually used in a course or competition, does not belong inside the task package, but should be stored separately. The relationship between submissions and tasks does need to be recorded.

2.7. *Metadata*

We have come a long way in defining the package contents. What we have described so far would already allow one to run a nice programming course or competition. When one is involved in multiple events, year after year, the need arises to look at things from a somewhat different perspective. For these longer term interests, it is useful to include certain metadata in a task package from the very beginning. One can think of the following items.

Task-instrinsic metadata including

- *Summary*, describing the task in one sentence; this is useful when dealing with *task bundles*;
- *Task type*, for instance, *batch* (through stdio, files, sockets, ...), *reactive* (through stdio, using or providing a library, sockets, ...), *output file only* (for given input files), etc.;
- *Difficulty level*, possibly distinguishing *understanding* (what to do), *designing* (abstract solution), and *implementing* (concrete submission); this is, of course, a somewhat subjective judgment, relative to specific context parameters (skill of solver, amount of time for solving, resource limits, programming language allowed, development tools available, etc.); each of these could be expressed on a scale of *easy*, *medium*, *hard*, possibly extended with *easy-medium*, *medium-hard*. This is usually done in a review meeting;
- *Topic classification*, what topics are involved in the task description, what topics are involved in a (typical) solution; this can be done in terms of a syllabus (Verhoeff *et al.*, 2006);

- *Notes* for data files and test submissions, summarizing their purpose.

Author-related data such as *name*, *contact information*.

Event-related data such as *name* of (or even better, some standardized *identifier* for) the original event (course or competition) at which it has been or will be used; *date* of that event, *amount of time allowed for solving*, *number of solvers* involved, etc.

Solver-related data such as their *background* (educational level, experience), *platform* used by solvers, characterizing the hardware architecture (processor, memory hierarchy), operating system, but also compilers, linkers, standard libraries, possibly also specific other tools allowed in solving the task at hand. This metadata helps in interpreting such things as configuration parameters (time and memory limits), because these are expressed relative to a certain platform.

Grading-related metadata such as *grading scale* (e.g., accepted–rejected, numeric 0–100, numeric 1–5, letter A–F, ...); *amount of time* it typically takes to grade a single submission. If the grading *platform* differs from the solver's platform, then it must also be characterized.

Management-related data such as *status of development* (in preparation, already used; incomplete, complete; draft, ready for review, approved); *version information*, *revision log* of content and status changes, *comments* (by author and reviewers), and a *to-do list*. A supervisor might also be interested in the amount of effort (time) it typically takes to translate the task description (possibly relative to some standard).

What metadata to include will also depend on the *style* of the course or competition. Compare, for instance, the styles of (IOI, 2008) and (ACM ICPC, 2008). Some metadata will be the same for all tasks used together in the same event. Good tasks must be expected to be reused later in other events, for example, on a training camp. It is advisable to copy that common information in each task package, so that a task in isolation is still complete.

2.8. *Miscellaneous Considerations*

The preceding compilation of items that can be included in a task package is not claimed to be complete and final. On some occasions, it may seem overkill; on other occasions, one may wish to include additional information.

There is a trade-off between putting data inside the package or keeping it outside. When data is not directly incorporated in the package, one has the option of incorporating some form of **reference** (like a URL) to that information instead. Our system (Peach, 2008) can be configured with *time intervals* for when a task is available to solvers and when submissions are accepted. We keep this information outside the package, because it will differ for each reuse of the package, e.g., in next year's course.

In an international event like the (IOI, 2008), texts presented to solvers must be translated. This is a major effort because so many languages are involved. It can be useful to provide **translation support**, such as separate figures and a *glossary*.

Another issue to be addressed concerns **task bundles**, that is, sets of tasks used together in a course or competition. In a task bundle, one often strives for a certain level of

uniformity. This can be achieved by copying common information into each task package. However, this makes it harder to change common information easily and consistently. An alternative is to introduce a kind of *inheritance mechanism* for task packages, and *abstract packages*. In fact, task bundles call for **bundle packages**, that contain (references to) task packages, but in particular also contain common items, such as document templates to be used for all tasks. But his is beyond the scope of the present article.

3. Package Interface and Operations

In the preceding section, we have discussed the contents of a task package. When constructing a package, the author is mainly "working inside it". Once a package is completed, there are several different ways of using it. At that stage, the package users (often not the author) wish to abstract from all internal details, and concentrate on specific package-level operations, such as

viewing (a summary of) (parts of) the package contents;

validating the package contents (for internal consistency and completeness; this is what the test submissions are for);

generating various items from the package, e.g., an archive to be made available to solvers, or information for a mediator (like a team leader);

grading a submission for the task; this could be done locally on the user's platform, or remotely inside an automated grading system; grading can be done *completely*, that is, fully performing all grading steps (preprocess, build, execute, evaluate, score, report), or *partially*, that is, performing only some user-selected steps;

cleaning up a package by removing auxiliary and temporary files;

installing a package in an automated grading system, e.g., by a simple *drag-and-drop*.

Especially for the automated use of packages, it is necessary to have a well-defined, clear, and uniform interface for the package operations. The implementations of these operations are provided by the scripts and tools inside the package, involving various external facilities (like compilers and libraries). The interface is intended to protect (the integrity of) the package contents.

It could be useful to include in the interface some limited ways of modifying a package as well. *Renaming* a task is good candidate, as is *tuning* (some of) the configuration parameters.

4. Package Format

There are many ways in which the package contents can be stored and given an interface. By using appropriate wrappers, one can convert between formats. However, an abundance of different formats is far from convenient. We now briefly describe the format currently used in (Peach, 2008).

4.1. *Peach Exchange Format for Programming Task Packages*

Peach task packages are stored in a directory tree with a predefined structure, naming scheme, and files formats. Fig. 1 shows the main features.

There are separate subdirectories for

- evaluation data subdivided in cases;
- test submissions subdivided by programming language;
- texts subdivided by natural language;
- tools.

The subdirectories for texts are named by their (RFC 4646, 2006) code; this code is based on (ISO 639-1, 2002) alpha-2 language identifiers and (ISO 3166-1, 2006) alpha-2 country codes. Unfortunately, there is no international standard for programming language and dialect identification codes. We use common names in lower case, and currently do not distinguish dialects.

At present, human-readable metadata can be stored in one language only, and is distributed over the tree (e.g., in various `summary.txt` files). Scripts are not in a separate subdirectory but spread out as well. A Python script in the root coordinates the grading steps, including language-dependent builds through a generic module. This script can be run locally or by a daemon inside our grading system (Peach, 2008). Other scripts are in (Unix) makefiles, e.g., for building tools and cleaning up, and in (Unix) shell scripts for viewing evaluation cases, and generating expected output through a known-correct solution. Evaluation drivers and stubs are stored with the test submissions, because in most cases they depend on the programming language. When communication is via sockets, it could be language independent, in which case, they are put in a `generic` subdirectory.

The current format does not standardize storage of additional task-specific information (other than texts), and of scripts to generate an archive for solvers. These things are handled in an ad hoc way.

Platform dependencies and tool dependencies (like compilers and libraries) are handled implicitly by references. Submissions are graded on a Linux platform, whereas in our educational setting, most students use Windows. There are minor issues concerning compiler versions.

The current format does not support inheritance or sharing of common information. Related packages are mostly created by branching.

Some temporary files are created inside the package when using it. This limits the possibilities of *concurrent usage*. Evaluation-related files are stored in a working directory outside the package.

Peach^3, the latest version of (Peach, 2008), still uses the format introduced for Peach^2. We are working on an improved Peach Exchange Format to overcome the limitations mentioned above.

Our task packages are stored in a central Subversion repository for configuration management. This works quite well because most persistent data is stored in line-based text files. Each package is treated as a composite configuration item. Tags are used to record status changes, and branches are used for variants.

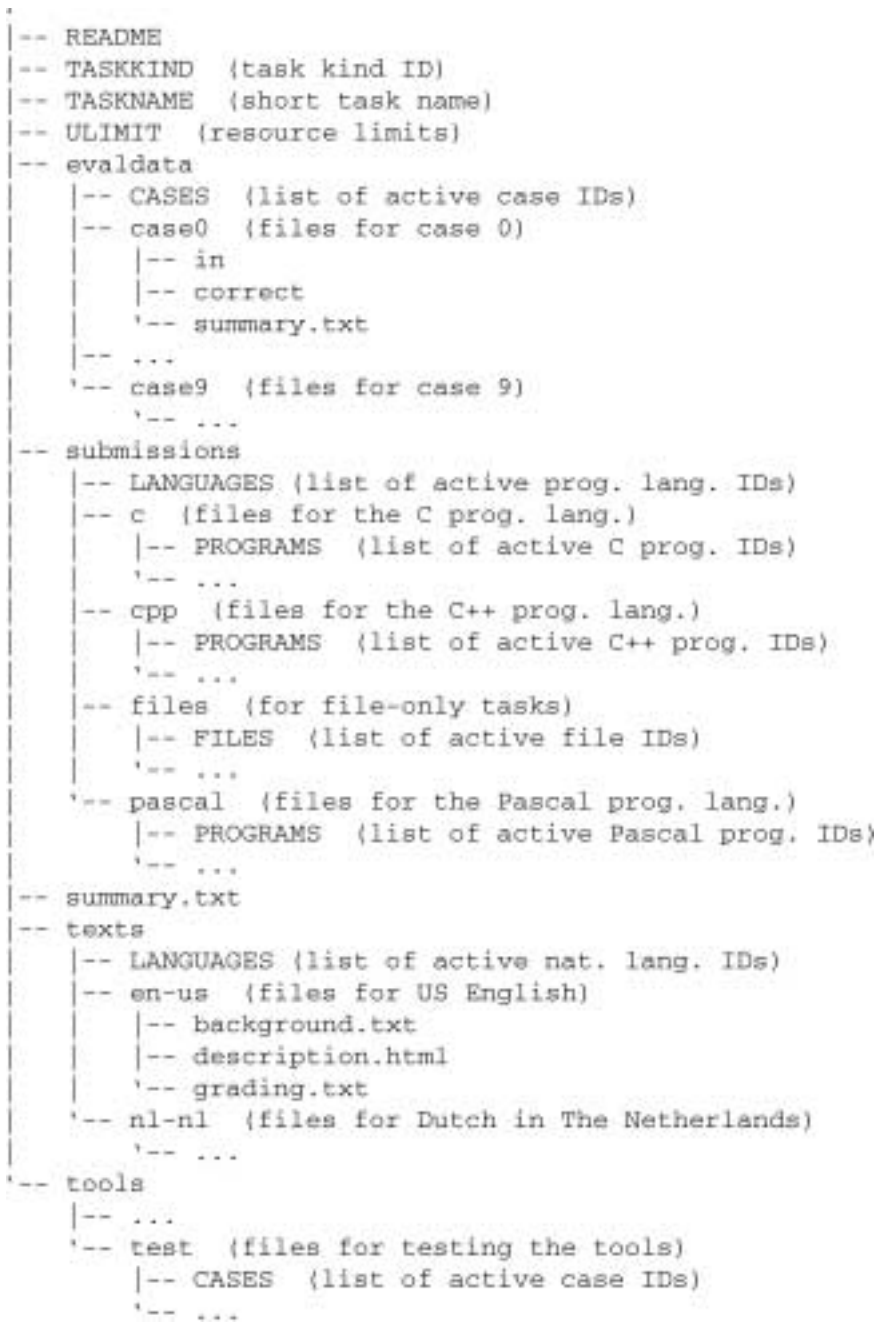


Fig. 1. Directory structure for information in Peach task package.

5. Conclusion

We have introduced the notion of a programming task package containing all task-related information, and serving as a unit for storage and communication. We have inventoried the stakeholders and contents of such packages, and the package interface and operations. This helps put in perspective the issues that arise when dealing with programming tasks on a larger scale.

Our automated programming education and contest hosting software (Peach, 2008) uses a package exchange format, which we have briefly described. The format is currently under revision to make it more generally usable. The Peach software is available under an open-source license.

At the moment, an application is lacking to handle task packages. Such an application should be supported on multiple platforms, and preferably should (also) provide a graphical user interface, possibly via a web browser.

Having a widely used task package format helps to improve the quality of programming tasks. But using task packages does not automatically lead to good quality. Task authors must still pay attention to many details when formulating a programming task; see for instance (Verhoeff, 2004).

We hope that this article will stimulate the development of an international standard for programming task packages. It would be good to standardize the interfaces of various task-specific tools as well. The *International Standard Task Number* and an *ISTN Registration Authority* (ISTN/RA) will then arise naturally.

Acknowledgments. The author is much indebted to Erik Scheffers, who co-designed Peach and who did most of the implementation work. We also wish to give credit to the more than 1500 users of Peach, who took part in dozens of courses and competitions, causing Peach to grade well over $25\,000$ submissions to date.

References

ACM ICPC (2008). *ACM International Collegiate Programming Contest*.
`http://icpc.baylor.edu/` (visited March 2008).

CTAN (2008). *Comprehensive TEX Archive Network*.
`http://www.ctan.org/` (visited March 2008).

DocBook (2008). `http://www.docbook.org/` (visited March 2008).

Docutils (2008). `http://docutils.sourceforge.net/` (visited March 2008).

IOI (2008). *International Olympiad in Informatics*.
`http://www.IOInformatics.org/` (visited March 2008).

ISO 639-1 (2002). *Codes for the representation of names of languages – Part 1: Alpha-2 code*.
`http://www.iso.org/iso/language_codes/` (visited March 2008).

ISO 3166-1 (2006). *Codes for the representation of names of countries and their subdivisions ÐPart 1: Country codes*.
`http://www.iso.org/iso/country_codes/` (visited March 2008).

OpenOffice (2008). `http://www.openoffice.org/` (visited March 2008).

Peach^3 by E.T.J. Scheffers and T. Verhoeff (2008). Technische Universiteit Eindhoven, The Netherlands.
`http://peach3.nl/` (visited May 2008).

RFC 3629 (2003). IETF Standard 63 concerning UTF-8, a transformation format of ISO 10646, Nov. 2003.
`http://www.ietf.org/rfc/rfc3629.txt` (visited March 2008).

RFC 4646 (2006). IETF Best Current Practice concerning *Tags for the Identification of Languages*, Sep. 2006. `http://www.ietf.org/rfc/rfc4646.txt` (visited March 2008).

Sloan, N. (2008). *Online Encyclopedia of Integer Sequences*. `http://www.research.att.com/ njas/sequences/` (visited March 2008).

UVa Online Judge (2008). `http://icpcres.ecs.baylor.edu/onlinejudge/` (visited March 2008).

Verhoeff, T. (1988). *Guidelines for Producing a Programming-Contest Problem Set*. Oct. 1988, expanded July 1990. `http://www.win.tue.nl/ wstomv/publications/guidelines.pdf` (visited March 2008).

Verhoeff, T. (2002). *RobIn for IOI I/O*, version 0.7. July 2002. `http://www.win.tue.nl/ wstomv/software/robin/Doc07.txt` (visited March 2008).

Verhoeff, T. (2004). *Concepts, Terminology, and Notations for IOI Competition Tasks*. Sept. 2004. `http://www.win.tue.nl/ wstomv/publications/terminology.pdf` (visited March 2008).

Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science*, **IV**(1), 193–216. `http://www.win.tue.nl/ wstomv/publications/ioi-syllabus-proposal.pdf` (visited March 2008).

W3C (2008). *World Wide Web Consortium*. `http://www.w3c.org/` (visited March 2008).

T. Verhoeff is an assistant professor in computer science at Technische Universiteit Eindhoven, where he works in the Group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI distinguished service award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.

Olympiads in Informatics

Volume 2 2008

1822-7732(2008)2;1-G