Olympiads in Informatics, 2008, Vol. 2, 5–15
© 2008 *Institute of Mathematics and Informatics, Vilnius*

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 white-board 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.

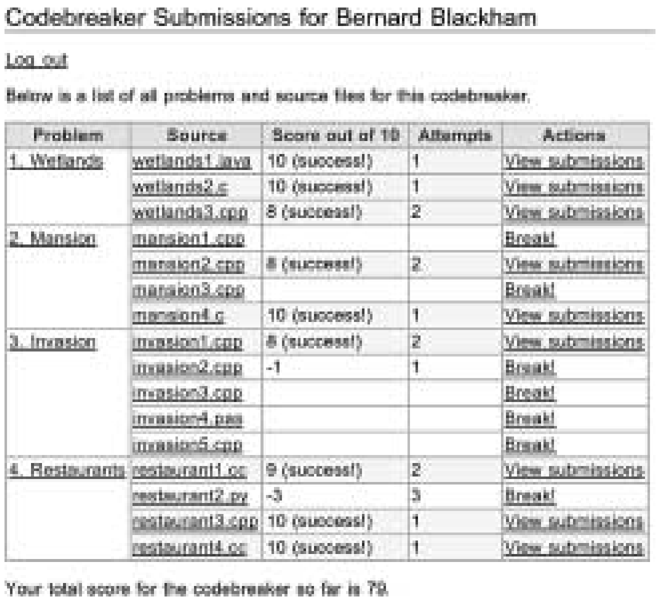
For these reasons, the codebreaker was created as a training school event in April 2005, and has been a fixture on the programme ever since. The codebreaker aims to help students develop the following skills:

- 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 designing 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.