Olympiads in Informatics, 2013, Vol. 7, 23–35



# Pushing the Boundary of Programming Contests

Michal FORIŠEK
*Comenius University, Bratislava, Slovakia*
*e-mail: forisek@dcs.fmph.uniba.sk*

**Abstract.** In traditional programming contests the tasks almost always focus on design of efficient algorithms. In this paper, we discuss that this does not have to be the case in the future, and we give a significant number of concrete tasks that cover other areas of Computer Science. All of the tasks presented in this paper have actually been used in past programming contests.

**Key words:** task design, task types, sample tasks.

## 1. Overview

This section contains the overview of the paper's main topic: the set of tasks used in programming contests. We map the landscape of relevant programming contest, and discuss prior research related to the topic of this paper.

### 1.1. *Programming Contests Landscape*

In this paper we are dealing with programming competitions – competitions that involve writing computer programs. However, there are multiple kinds of programming competitions and we will only be interested in a particular subset of those: In terms of the terminology from Pohl (2011), in this paper we are interested only in *short-term* contests with clearly defined *tasks*.

(This leaves out competitions in which the contestants work on open-ended projects, such as the Imagine Cup or various types of robotics contests (Petronič, 2011). It also leaves out long-term competitions with clearly defined tasks, such as the Marathon matches track at TopCoder and contests like the CodeCup (Vegt, 2006) and ICPC Challenge where the contestants implement programs that play a game against each other.)

In other words, we shall consider short-term competitions that focus on creative thinking and problem solving. In these competitions, the contestants are given a particular set of problem statements, and the primary goal of the competition is to find solutions to the given problems. The computer programs produced by the contestants can then usually be seen as the "final proof" that a given problem has been solved.

Some of the largest worldwide competitions that fit into this group include:

- the ACM International Collegiate Programming Contest (ICPC),
- the International Olympiad in Informatics (IOI) along with the corresponding national olympiads for secondary school students,

– company-branded contests such as the Google CodeJam and the Facebook Hacker Cup,
– large portals that host regular contests, such as CodeForces, CodeChef and the Algorithm track at TopCoder.

Currently, all of these contests share a common characteristic: the competition problems focus on the design of efficient algorithms.

Most of these competitions acknowledge this focus openly. For instance, the Google CodeJam rules state that *The Google Code Jam Contest is a competition designed to engage programmers from around the world in algorithmic programming.* and the Facebook Hacker Cup official blurb contains the characteristic *In the Hacker Cup, programmers from around the world will be judged on accuracy and speed as they race to solve algorithmic problems to advance [...].*

The most popular track in TopCoder competitions is actually called *Algorithms* and competitions in this track are the traditional short algorithmic problem solving contests.

The ACM International Collegiate Programming Contest (ICPC) does not stress the algorithmic aspect and only focuses on programming in its official description: *The contest fosters creativity, teamwork, and innovation in building new software programs, and enables students to test their ability to perform under pressure. Quite simply, it is the oldest, largest, and most prestigious programming contest in the world.* Nevertheless, in the last decade the problems used at the ICPC World Finals are all of algorithmic nature.

The International Olympiad in Informatics (IOI) regulations only state the following objectives for the IOI:

– *To discover, encourage, bring together, challenge, and give recognition to young people who are exceptionally talented in the field of informatics.*
– *To foster friendly international relationships among computer scientists and informatics educators.*
– *To bring the discipline of informatics to the attention of young people.*
– *To promote the organisation of informatics competitions for students at schools for secondary education.*
– *To encourage countries to organise a future IOI in their country.*

One may note the complete absence of any mention of algorithms. On the other hand, each particular IOI competition has its own competition rules, and in the past years these rules always stated: *All of the tasks in IOI [year] are designed to be algorithmic in nature.* Almost without an exception, the IOI is using tasks that focus on algorithmic problem solving. See Verhoeff (2009) for more on IOI competition tasks.

As with all the rules, there have to be some exceptions. In this paper, we describe our effort to create some of those exceptions – to design programming competition tasks where the goal is other than just designing the most efficient algorithm. The tasks presented in the main sections of this paper come from two main sources:

1. The **Internet Problem Solving Contest (IPSC)**: an annual competition organized since 1999 by a group Comenius University faculty and students, including the author of this paper.

While primarily a programming / algorithmic problem solving contest, the IPSC strives to push the boundary of these competitions and include tasks that one would not expect in a regular programming contest.
IPSC is conducted online and open for everyone. In 2012, 1306 teams from 81 countries have registered for the contest.

2. The **Slovak Olympiad in Informatics (OI)** for secondary school students, and the related **Correspondence Seminar in Programming (KSP)** – a long-term Slovak national competition for secondary school students.

### 1.2. *"Algorithm Design" Does Not Equal "Problem Solving"*

The overwhelming preference of algorithmic problems is easily explained: Their nature makes them very suitable for short-term contests. It is easy to create "toy problems" with clearly defined statements that can be solved completely within the limited length of a contest.

One other benefit of using these algorithmic tasks in practical programming contests is the resulting scalability of these contests: the contestants' programs can be tested without the need of human interaction, which makes it possible to organize very large contests. (However, note that such evaluation is necessarily imprecise. There can always be false positives where an incorrect program is evaluated as correct. See Forišek (2006) for a discussion of some disadvantages of black-box testing in programming contests.)

There has been a substantial number of prior research publications dealing with various ways in which the spectrum of task topics for these programming contests can be broadened. Below we give a brief overview of these papers.

*Computer Science without the Computer*
As long as the contest is reasonably small, it is perfectly feasible to use theoretical tasks, have contestants write down their solutions on paper and have them checked manually. Some of the rounds of the Slovak OI (Forišek, 2007) work this way, and we believe that it is a good thing (Forišek, 2006). Pohl (2008) offers some more perspective on this topic and provides experience from the German OI (Bundeswettbewerb Informatik). Burton (2010) describes the pen-and-paper structure of the Australian Informatics Competition.

A lot of Computer Science can be introduced to the kids without actually using a computer – and starting at a surprisingly early age. The "Computer Science Unplugged" book of activities (Bell *et al.*, 1998) should be among the tools of any Computer Science educator. Some more activities in a similar spirit can be found in Forišek and Steinová (2010).

Kubica and Radoszewski (2010), van der Vegt (2012) and Ginat (2011, 2012) present a set of algorithmic tasks that are well suited for solving on paper, without programming.

*New Task Types for Programming Contests*
Kemkes *et al.* (2007) examine the use of open-ended tasks: clearly stated tasks without a known optimal solution, graded based on the quality of the solution produced by the

contestant. Ribeiro and Guerreiro (2007) suggest tasks that use graphics, in particular a Graphical User Interface (and also comment on the possibility of adding visualization to some traditional algorithmic tasks from past IOIs). Truu and Ivanov (2008) propose a way in which testing-related tasks can be used, and discuss several examples. Opmanis (2009) gives a classification of task topics used in a Latvian competition in mathematics and informatics. Among others, these topics include data mining tasks, word problems, logic problems, and tasks on analysis of algorithms. Skupiene (2006) examines the possibility of including the programming style into the grading scheme. Kulczynski *et al.* (2011) give examples of tasks that are solvable using precomputation and/or visualization of the problem.

Various related contests can serve as inspiration for new problem types in traditional programming contests. Here we would like to mention the First Spanish Parallel Programming Contest described by Almeida *et al.* (2012), and the multitude of interesting task types used in the Bebras (Beaver) contest Opmanis (2006).

Recently, Ragonis (2012) gave a fairly thorough classification of types of questions that may occur in Computer Science competitions (and Computer Science education, in general).

### 1.3. *Paper Structure*

The rest of this paper contains our original content: a selection of non-traditional tasks used in Slovak programming contests. The content is divided into multiple sections according to task type.

Due to the limited space, all task statements were shortened as much as possible. All original IPSC task statements are available online at `http://ipsc.ksp.sk/archive`. Whenever the statement summary in this paper seems inadequate, we recommend reading the full task statement *before* reading the corresponding solution section of this paper.

## 2. Areas Not Related to Efficient Algorithms

The tasks in this section are IPSC tasks that venture into other areas of Computer Science. Solutions to all of these tasks were submitted online and checked automatically.

### 2.1. *IPSC 2012: Invert the You-Know-What [Topic: Cryptography]*

*Statement*
You are given a password file: a collection of usernames and password MD5 hashes. Obtain some of the passwords.

*Solution*
Some of the most common MD5 hashes can easily be reverted simply by googling them/ looking them up in a database. To obtain others, some additional analysis of the given data was necessary. Existing password crackers such as John the Ripper could also be used for partial credit.

### 2.2. *IPSC 2012: Keys and Locks [Topic: Cryptography]*

*Statement Summary*
The problem statement describes one actual way how master keys for physical locks can be made. The contestants are then asked to devise an attack on such a physical system, armed only with a small set of "blank" keys and an iron file. The attack is then played out by interacting with the grading system.

*Solution*
The attack was apparently an open secret among locksmiths before being discovered and published by Blaze (2003). The amount of resources needed is surprisingly small – the attacker can easily and efficiently learn the heights of the "teeth" of a master key, one "tooth" at a time.

### 2.3. *IPSC 2011: Lame Crypto [Topic: Cryptography]*

*Statement*
Alice and Bob are communicating using a particular cryptographic protocol (described formally in the actual statement). You get to eavesdrop on their messages and change some of them. Show that the particular protocol is faulty by correctly inserting a false message into the communication.

*Solution*
The contestants were given access to a webpage where they could intercept and possibly modify the packets sent between Alice and Bob. They had to discover a weakness in the cryptographic protocol we used, and then actually exploit it in "real time".

### 2.4. *IPSC 2011: Jedi Academy [Topic: Computer Graphics]*

*Statement*
Given a view of a 3D scene with many triangles, compute the color of a particular pixel.

*Solution*
This would have been a real pain to solve as a programming task. However, there are simpler ways of getting the result. The intended solution was to use OpenGL (or a similar library) to actually model the scene and have it displayed.

### 2.5. *IPSC 2008: Hidden Text [Topic: Computer Graphics]*

*Statement*
You are given a picture. Part of the picture has been blurred (using an algorithm very similar to the standard Gaussian blur available in most graphics editors). The blurred part originally contained some text. Recover it.

*Solution*
There were two main approaches that lead to a correct solution: First, there was the option to use the tools available in a graphics editor to repair enough of the blur to make the text readable. For the second approach the contestants had to realize that they know the font used in the image, so they can compute the blurred forms of individual letters and compare them to the blurred part of the image.

### 2.6. *IPSC 2008: Comparison Mysteries [Topic: Data Representation]*

*Statement*
Declare a numeric variable `x` and initialize it to a non-zero value. Your variable must then satisfy `x == -x`.

Declare three numeric variables `x`, `y`, and `z`. Initialize them to any values. Your variables must then satisfy `x == y`, `y == z`, but they must not satisfy `x == z`.

*Solution*
One of the valid solutions for the first subtask: `int x = -2147483648`. In two's complement representation the range of integers is not symmetric, and the largest negative value has no positive counterpart. Instead, it is its own negation. (Formally, $2^{31}$ and $-2^{31}$ belong into the same remainder class modulo $2^{32}$, and this class is represented by the integer $-2^{31}$.)

One of the valid solutions for the second subtask: `int x = 1234567890; float y = 1234567890; int z = 1234567891;`. This solution uses the fact that integers and floating-point numbers have incomparable ranges. In the comparisons `x == y` and `y == z` the int gets converted to a float, and the resulting rounding causes both comparisons to evaluate as true. On the other hand, the two ints clearly differ by 1.

## 3. Non-Traditional Computational Models

We love to challenge our students to write "programs" for various non-traditional computational models. Among other reasons, this eliminates much of the pre-existing knowledge such as fluency in a given programming language, and thus all contestants start in the same place. Additionally, these different computational models often require a different way of thinking.

Traditionally, each year the Slovak OI contains a graduated series of tasks in some such model. In the first round (a long-term round solved at home) they are given a short introductory text that defines the model and shows some small examples. In all rounds of the contest they are then asked to solve progressively more and more difficult tasks in the model.

Some notable models used in the past include: many-one and Turing reductions (2013), log-space programs (2012), Fractran (2011), a-transducers (2007), alternating machines (2005), reversible programs (2002), Wang tiles (2001), and many more.

Below we present just a few examples from this large batch of tasks.

### 3.1. *OI 2012: Log-Space: Permutation Cycles*

*Statement*
You are given $n$ and a read-only array $A[1..n]$ that contains a permutation. Write a log-space program (i.e., a program that only uses $O(\log n)$ bits of memory) that counts the cycles of $A$.

*Solution*
Note that we cannot use the usual algorithm that uses graph traversal to count the cycles, because we cannot mark the elements of $A$ as visited.

For each $x$, we walk along the cycle that contains $x$, and we count this cycle iff $x$ is the smallest value it contains. In this way we are guaranteed to count each cycle exactly once.

(An alternate solution. For each length $\ell$ between 1 and $n$: For each $x$, verify whether the cycle that contains $x$ has length $\ell$. Let $c$ be the count such elements. Then we just found $c/\ell$ cycles of length $\ell$.)

### 3.2. *OI 2011: Fractran: Comparing Exponents*

*Statement*
A Fractran program is an ordered sequence of positive fractions. A Fractran computation is a sequence $S$ of positive integers. The first integer is the input. Each of the following integers is computed using a simple rule: Let $x$ be the current integer. We find the first fraction $f$ in $S$ such that $xf$ is an integer. The value $xf$ is the next element of the computation. The computation ends when no fraction produces an integer.

The input number $n$ is guaranteed to be of the form $2^x 3^y 5$. Write a Fractran program that terminates with the value 5 whenever $x = y$, or the value 7 whenever $x \neq y$.

*Solution*
One correct program:

$$\left( \frac{1}{6}, \quad \frac{7}{10}, \quad \frac{7}{15}, \quad \frac{1}{2}, \quad \frac{1}{3} \right)$$

Suppose that $x \neq y$. While both $x > 0$ and $y > 0$, the first fraction is used to decrease both by 1. If this brings us to the situation where $x > 0$ and $y = 0$, we can now use the fraction $7/10$. From this point on, the current value is not divisible by 5, but it is divisible by 7. Similarly, if $x = 0$ and $y > 0$, the first fraction that can now be used is $7/15$. Finally, we use the last two fractions to clear the remaining power of 2 or 3 in the current value, leaving only the 7.

And if $x = y$, we will use the first fraction $x$ times, and then the algorithm terminates with the current value being 5.

### 3.3. *KSP 2013: Regular Expressions*

*Statement (of one Subtask)*
Write a regular expression that matches strings of `a`s such that their length is a composite number.

*Solution*
One such regular expression is "`^(aa+)\1+$`". We take a sequence of at least two `a`s, and then use a back-reference to state that the rest of the input must consist entirely of a positive number of copies of the selected sequence. This regular expression can only be matched to a string of `a`s if its length has a non-trivial divisor.

### 3.4. *IPSC 2004: Gets and Puts*

*Statement (of one Subtask).*
Given is a simple programming language in which all you have are 26 integer registers and a single potentially-infinite queue. Write a program that is a quine (i.e., writes its own source code to the output).

*Solution*
The standard technique based on the recursion theorem can be used. Imagine a program that can be run once a queue already contains a sequence of integers. This program will do the following steps:

1. Insert an endmarker into the queue.
2. While not at the endmarker: take a number from the queue, **print** the command that inserts the number into the queue, insert the number back into the queue.
3. Throw away the endmarker. The queue is now in its original state.
4. While there is something in the queue: take a number, interpret it as an ASCII value, **print** the corresponding character.

Let this program be $P$. Let $Q$ be the sequence of ASCII values of characters of $P$. The solution is a program that first inserts all elements of $Q$ into the queue and then runs $P$.

## 4. Algorithm/Code Analysis

In this section we give several examples of tasks where the contestants are asked to analyze an algorithm, or its particular implementation. In addition to the traditional testing problem (find a bug in this program), there are many other variations possible: understanding what a program computes, improving its time complexity, ...

Included in this section is one example of a long-term task.

### 4.1. *IPSC 1999: Coins*

*Statement*
We have a set of coins. We want to pay the sum $s$ precisely. We have two different programs. Each of them claims that it can decide whether it is possible to pay $s$ or not. Find a counterexample for each of the programs.

*Solution*
The first program given to the contestants was the implementation of a simple greedy algorithm: always use the largest coin that does not exceed the remaining sum. There are very simple counterexamples, for example $s = 6$ and we have the following coins: $(5, 2, 2, 2)$.

The second program contained a slightly smarter algorithm. For each coin value $v$: Let $x$ be the number of coins with this value I have. For each $i$ between 1 and $x$: Pay $i$ coins worth $v$, then use the first greedy algorithm to pay the rest. If no pair $(v, i)$ works, return that it is impossible to pay $s$.

Probably the simplest counterexample combines two layers of the previous algorithm: let $s = 66$ and let the coins be $(50, 20, 20, 20, 5, 2, 2, 2)$.

### 4.2. *IPSC 2003: begin 4 7 add*

This task actually belongs both into this and into the previous section, because it deals with PostScript. PostScript is often considered to be just a language used to describe the layout of a page ... but in fact it is a Turing-complete stack-based programming language.

*Statement*
Given are two PostScript documents that produce the answers you have to submit. Obtain those answers.

*Solution*
The easier document just iterated a permutation of letters. It was sufficient to find the period, edit the file to use a smaller number of iterations, and open it in a PostScript viewer.

The more challenging document contained a Markov source that generated random words that were irrelevant. But at the same time the code computed the sum of all primes between 1 and 200,000, inclusive, which was the answer.

### 4.3. *IPSC 2011: BFS Killer*

*Statement*
Given is a particular implementation of breadth-first search (BFS) on a rectangular grid of cells, some of which contain obstacles. Find a particular input that causes the BFS to have a very long queue.

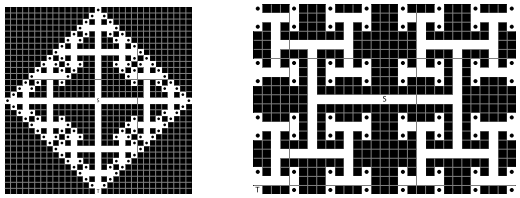

Fig. 1. Two mazes that require a large queue.

*Solution*
This task addresses a very common misconception[1] that the BFS, when executed on an $n \times n$ grid, will always use $O(n)$ additional memory. This is not the case. Below are two mazes that require a large queue – all cells marked by dots will be in the queue at the same time. For the maze on the right the maximum queue size is $\Theta(n^2)$.

### 4.4. *KSP 2007: Ordinary Sorting Algorithms*

Note: This is a large long-term task with many subtasks that can be used as separate task. The task involves both theoretical and practical (implementational) subtasks.

*Statement*
Three kids came up with weird algorithms to sort an array of length $n$.

Annie: I would just pick a random pair of elements and compare them. Whenever the one on the right is smaller, I would swap them. I would do this in a loop, and after every $n$ comparisons I would check whether the array is already sorted. If it is, I would terminate the algorithm.

Billy: I would use recursion. An array of length 1 is already sorted. If I have an array of a longer length, I would make three recursive calls: I would sort the first two thirds of the array (rounded up if necessary), then the last two thirds of the array, and finally the first two thirds of the array again.

Cecilia: I would take $n-1$ pieces of paper. On the $i$-th piece I would write the numbers $i$ and $i+1$. I would then proceed in rounds. In each round, I would shuffle the papers into a random order. Then, for each paper, I would read the two numbers on it, compare the elements on those indices, and swap them if necessary. At the end of each round I would check whether the array is sorted.

Implement the three algorithms. Create plots of their runtime as a function of $n$. Use those plots to guess their time complexity. Find and prove some lower bounds on their worst-case time complexity, – i.e., find inputs that are hard for that particular algorithm, and compute how many steps it would take for those inputs. Prove that algorithm B actually sorts. For extra credit, prove good upper bounds on their time complexity.

[1] A historical remark: a few years ago a task related to BFS was proposed to be used at the IOI. The author made a similar mistake in his intended solution, and the task had to be rejected.

*Solutions Sketch*
The implementations contain several challenges: In algorithm A, some contestants incorrectly implemented the step where a pair of elements is chosen uniformly at random. Some contestants were unable to implement the recursion in B. In C, incorrect and/or slow implementations are common for randomly shuffling the array of "papers".

The expected time complexity of algorithm $A$ is $O(n^2 \log n)$. This is related to the Coupon Collector problem. E.g., the array $(2, 1, 4, 3, \ldots, 2k, 2k-1)$ contains $n/2$ independent inversions, and we have to hit each of them at least once in order to sort the array.

The time complexity of algorithm $B$ is $\Theta(n^{\log_{3/2} 3})$, which is $O(n^{2.71})$. This can be shown by solving the recurrence $T(n) = 3T(2n/3)+O(1)$. The fact that $B$ actually sorts is left as an exercise. (Hint: color the largest $n/3$ elements red and follow them during the sorting.)

A clear worst case for algorithm $C$ is the reverse of a sorted array, which needs $\Omega(n^2)$ swaps of adjacent elements to become sorted. Algorithm $C$ randomly chooses one from a particular class of oblivious sorting algorithms. Using the 0-1 principle it can be shown that any such algorithm works in $O(n^2)$.

## 5. Lateral Thinking

Finally, we present three "lateral thinking" tasks. Part of the design of these tasks is to encourage the contestants to "think out of the box", as the saying goes.

### 5.1. *IPSC 2005: Alpha Centauri Tennis*

*Statement Summary*
A fairly long and detailed problem statement describes the scoring of a fictional sport – the "Alpha Centauri Tennis". This is essentially the same as traditional tennis, only without tie-breaks, and it is generalized to $n$ players. The essence of the rules remains the same: whoever scores enough points wins a game, whoever wins enough games wins a set, and whoever wins enough sets wins the match.

A match transcript is a string that records, for each ball played in the match, the player who scored the point.

The contestant is given a lot of match transcripts that are *guaranteed to be valid*. The task is to process them and determine the winner of each match.

*Solution*
As shown by our survey after the contest, a significant portion of teams implemented the full set of rules, computed the score of each match, and used that to return the winner. Needless to say, this approach was painful, error-prone, and slow.

Not to mention its complete uselessness. There is a much simpler solution: the winner of the match has to be the player who won the very last point in the match. Thus it was sufficient to output the last character of each match transcript.

### 5.2. *IPSC 2007: Know Your Crypto*

*Statement (of the Hard Subtask)*
We have to apologize for the hard problem, it is probably impossible to solve. This task was prepared at the last possible moment, and we didn't have any idea what to do with the hard input. So finally we decided that we will just randomly change all the letters of the plain text. This is the program we used:

(A listing of an incredibly simple C program followed. The program executed `srand(time(NULL));` and then shifted each letter of the input by a randomly selected amount.)

*Solution*
The problem intentionally misleads the contestants into thinking that the task is unsolvable, or that the solution lies in somehow reversing the pseudorandom `rand()` function using the fact that it is not perfectly random.

The correct clue is hidden in the words "at the last possible moment". This gives only a small set of values `time(NULL)` could have returned, and the contestant can try them all by brute force, and select the one that produces a readable plaintext.

(As an interesting note, similar poor choices of random seeds have been documented in practice, e.g., in Casino de Montréal in 1994.)

### 5.3. *IPSC 2012: (Blank)*

*Statement*

*Solution*
The statement above is printed correctly. It had precisely zero characters (a clear world record!).

Still, the task was solvable – by using the error messages from the grader. For your first submission, you were most likely to receive the following error message: "`Wrong answer: Not a~sequence of positive integers`". Obviously, your second submission would contain such a sequence. The easy subproblem was solvable by simply following the instructions, for the hard subproblem some additional insight was needed.

**Acknowledgements.** The author is grateful to all his colleagues and students who collaborated on preparing the competition tasks described in this paper. Task authors for individual IPSC tasks can be found on the IPSC website; all tasks are licensed under the Creative Commons Attribution-ShareAlike 3.0 license.

## References


Almeida, F., Pérez, V.B., Cuenca, J., Férnandez-Pascual, R., García-Mateos, G., Gimenéz, D., Guillén, J., Benito, J.A.P., Requena, M.-E., Ranilla, J. (2012). An experience on the organization of the first spanish parallel programming contest. *Olympiads in Informatics*, 6, 133–147.

Bell, T., Fellows, M.R., Witten, I. (1998). *Computer Science Unplugged ... Off-Line Activities and Games for all Ages*.

Blaze, M. (2003). Cryptology and physical security: rights amplification in master-keyed mechanical locks. *IEEE Security and Privacy*.

Burton, B.A. (2010). Encouraging algorithmic thinking withouty a computer. *Olympiads in Informatics*, 4, 3–14.

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

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

Forišek, M., Winczer, M. (2006). Non-formal activities as scaffolding to informatics achievement. In: Dagiene, V., Mittermeir, R. (Eds.), *Information Technologies at School*, 529–534.

Forišek, M., Steinová, M. (2010). Didactic games for teaching information theory. In: *Teaching Fundamentals Concepts of Informatics (Proceedings of ISSEP 2010)*.

Ginat, D. (2011). Algorithmic problem solving and novel associations. *Olympiads in Informatics*, 5, 3–11.

Ginat, D. (2012). Insight tasks for examining student illuminations. *Olympiads in Informatics*, 6, 44–52.

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

Kubica, M., Radoszewski, J. (2010). Algorithms without programming. *Olympiads in Informatics*, 4, 52–66.

Kulczyński, T., Łącki, J., Radoszewski, J. (2011). Stimulating students' creativity with tasks solved using precomputation and visualization. *Olympiads in Informatics*, 5, 71–81.

Opmanis, M. (2009). Team competition in mathematics and informatics "ugāle" – finding new task types. *Olympiads in Informatics*, 3, 80–100.

Opmanis, M., Dagienė, V., Truu, A. (2006). Task types at "Beaver" contests. In: Dagiene, V., Mittermeir, R. (Eds.), *Information Technologies at School*, 509–519.

P. Petrovińć. Ten years of creative robotics contests. In: *Proceedings of ISSEP 2011*.

Pohl, W. (2008). Manual grading in an informatics contest. *Olympiads in Informatics*, 2, 122–130.

Pohl, W. (2011). Computer science contests for secondary school students, approaches to classification. *Informatics in Education*, 5, 125–132.

Ragonis, N. (2012). Type of questions – the case of computer science. *Olympiads in Informatics*, 6, 115–132.

Ribeiro, P., Guerreiro, P. (2007). Increasing the appeal of programming contests with tasks involving graphical user interfaces and computer graphics. *Olympiads in Informatics*, 1, 149–164.

Skūpienė, J. (2006). Programming style – part of grading scheme in informatics olympiads: Lithuanian experience. In: Dagiene, V., Mittermeir, R. (Eds.), *Information Technologies at School*, 545–552.

Truu, A., Ivanov, H. (2008). On using testing-related tasks in the IOI. *Olympiads in Informatics*, 2, 171–180.

van der Vegt, W. (2006). The CodeCup, an annual game programming competition. In: W. Pohl Ed., *Perspectives on Computer Science Competitions for (High School) Students*.

van der Vegt, W. (2012). Theoretical tasks on algorithms; two small examples. *Olympiads in Informatics*, 6, 212–217.

Verhoeff, T. (2009). 20 Years of IOI competition tasks. *Olympiads in Informatics*, 3, 149–166.


**M. Forišek** is an assistant professor at the Comenius University in Slovakia. Since 2006 he serves as an elected member of the International Scientific Committee (ISC) of the IOI. He is also the head organizer of the Internet Problem Solving Contest (IPSC). His research interests include theoretical computer science (hard problems, computability, complexity) and computer science education.