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.