Informatics olympiads: Approaching mathematics through code[1]

Benjamin A. Burton

Benjamin Burton is the Australian team leader for the International Olympiad in Informatics, and has directed the Australian training programme for the past eight years. He also has a keen interest in mathematics enrichment, having spent thirteen years teaching at the National Mathematics Summer School and five years helping train the IMO team. As a student he won a gold medal at the IMO in Moscow, 1992. His research interests include low-dimensional topology and information security.

Many readers are familiar with the International Mathematical Olympiad (IMO), a pinnacle in the yearly calendar of mathematics competitions. Here we introduce its cousin in computer science, the International Olympiad in Informatics (IOI).

The International Olympiad in Informatics is one of the newer Science Olympiads, beginning in 1989 in Bulgaria under the leadership of Petar Kenderov. In its short lifespan it has also become one of the largest, with 74 countries participating in 2006 [16]. Like the IMO, the competition is targeted at secondary school students who compete on an individual basis.

[1]This paper is based on a presentation by the author at the WFNMC Congress, 2006.

In this paper we take the reader on a brief tour of the IOI, paying particular attention to its relationships with traditional mathematics competitions. We begin in Section 1 with the structure of a typical IOI problem, including an outline of the mathematical ideas that lie behind such problems. For illustration, two problems are worked through in detail in Section 2. In Section 3 we compare the different ways in which mathematics and informatics contests encourage students to think mathematically. Section 4 closes with suggestions for students and teachers who wish to become involved in the IOI.

It should be noted that the IOI is not the only international competition of this type. Other examples include the ACM International Collegiate Programming Contest for university students [1], and the privately-run TopCoder contests [14]. For an overview and comparison of these and related competitions, see [7].

Thanks must go to Margot Phillipps, the IOI team leader for New Zealand, for her valuable comments on an earlier draft of this paper.

1 Structure of an IOI Problem

A typical informatics olympiad task requires competitors to create an algorithm that can solve some given problem. For instance, they might need an algorithm to calculate the nth Fibonacci number, or to compute the area of the intersection of two polygons. Any algorithm will not do, however; this algorithm must be both *correct* and *efficient*.

The concepts that lie beneath informatics problems are often mathematical in nature, and for harder problems students need a sound mathematical mindset to succeed. However, unlike a typical mathematics contest, students do not communicate their solutions by writing proofs—instead they communicate their solutions by writing computer programs.

In a similar vein, solutions are marked by computer. The contest judges prepare an *official data set*, consisting of several input scenarios that the students' programs must handle. Students receive marks according to whether their programs answer these scenarios correctly and within a given time limit (thereby testing correctness and efficiency).

The official data set essentially takes the place of a traditional marking scheme, and so the judges must take great care to construct it well. It typically consists a range of cases from easy to hard, including both small and large input scenarios to test for efficiency, and both "ordinary" and pathological input scenarios to test for correctness.

A typical problem statement has the following components:

- *The task overview.* This explains precisely what task the students' algorithms must solve, and often wraps it in a real-world (or fantasy) story, or *flavourtext*.
- *The input and output formats.* These explain the precise formats of the text files that students' programs must read and write. Each scenario from the official data set is presented to the program in the form of a given *input file*, which the program must read. Likewise, each program must write its corresponding solution to a given *output file*, which is given back to the judging system for marking.
- *Limits on time, memory and input data.* These allow algorithms to be tested for efficiency. Each program is given a small amount of time in which it must run (the *time limit*, typically no more than a few seconds) and a maximum amount of memory that it may use (the *memory limit*). Competitors are also given upper and lower bounds for the input data (which the scenarios in the official data set promise not to exceed), so they can estimate whether their programs are likely to run within the time and memory limits even for the most difficult scenarios.
- *Sample input and output files.* These sample files illustrate some simple input scenarios and their solutions. The problem statement should be perfectly understandable without them; they are merely given as examples so that students can be sure that they understand both the task and the input/output formats correctly.

Figure 1 shows a problem statement that, whilst simple, illustrates each of the components listed above. A sample solution using the Pascal programming language is given in Figure 2 (though this solution is not optimal, as seen in the following section).

Pascal's Triangle

Pascal's triangle is a triangular grid of numbers whose rows are numbered $0, 1, 2 \ldots$ and whose columns are also numbered $0, 1, 2, \ldots$. Each number in the triangle is the sum of (i) the number immediately above it, and (ii) the number immediately above it and to the left. The numbers along the boundary all have the value 1. The top portion of the triangle is illustrated below.

Row 0 $\rightarrow$ 1
Row 1 $\rightarrow$ 1 1
Row 2 $\rightarrow$ 1 2 1
Row 3 $\rightarrow$ 1 3 3 1
Row 4 $\rightarrow$ 1 4 6 4 1
$\vdots$ $\ddots$

Your task is to write a computer program that can calculate the number in row r and column c of Pascal's triangle for given integers r and c.

Input: Your program must read its input from the file `pascal.in`. This file will consist of only one line, which will contain the integers r and c separated by a single space.

Output: Your program must write its output to the file `pascal.out`. This file must consist of only one line, which must contain the number in row r, column c of Pascal's triangle.

Limits: Your program must run within 1 second, and it may use at most 16 Mb of memory. It is guaranteed that the input integers will be in the range $0 \leq c \leq r \leq 60$.

Sample Input and Output: The sample input file below asks for the number in row 4, column 2 of the triangle. The corresponding output file shows that this number is 6 (as seen in the triangle above).

`pascal.in`:

```
4 2
```

`pascal.out`:

```
6
```

Figure 1: A simple problem that illustrates the various components

```
program Triangle;

var
    r, c : longint;

function pascal(i, j : longint) : longint;
begin
    { Compute the value in row i, column j of the triangle. }
    if (j = 0) or (i = j) then
        { We are on the boundary of the triangle. }
        pascal := 1
    else
        { We are in the interior; apply the recursive formula. }
        pascal := pascal(i - 1, j - 1) + pascal(i - 1, j);
end;

begin
    assign(input, 'pascal.in');
    reset(input);
    assign(output, 'pascal.out');
    rewrite(output);

    readln(r, c);
    writeln(pascal(r, c));

    close(input);
    close(output);
end.
```

Figure 2: A correct but inefficient solution to Pascal's Triangle

Correctness and Efficiency

As discussed earlier, algorithms for informatics olympiad problems must be both *correct* and *efficient*. This means that students must think about not only the theoretical concerns of obtaining the correct answer, but also the practical concerns of doing this without placing an excessive burden upon the computer.

For example, consider again the solution to Pascal's Triangle presented in Figure 2. Whilst it certainly gives correct answers, it is not efficient—in the worst case the recursive routine `pascal(i,j)` may call itself on the order of 2^r times as it works its way from the bottom of the triangle to the top. For the maximal case $r = 60$ this cannot possibly run within the 1 second that the program is allocated. A simple but inefficient solution such as this might score $\sim 30\%$ in a real competition (that is, approximately 30% of the scenarios in the official data set would be small enough for this program to run in under 1 second).

The solution can be improved by keeping a lookup table of intermediate values in different positions within the triangle. That is, each time `pascal(i,j)` finishes computing the value in some position within the triangle, it stores this value in a large table. Conversely, each time `pascal(i,j)` is called, the program looks in the table to see if the value in this position has already been calculated; if so, it avoids the recursive descent and simply pulls the answer from the table instead.

In this way we reduce the number of recursive calls to `pascal(i,j)` from the order of 2^r to the order of r^2 instead, since there are $\leq r^2$ distinct positions in the triangle that need to be examined en route from bottom to top. Using this lookup table we can therefore reduce an exponential running time to a (much better) quadratic running time. For the worst case $r = 60$ we are left with ≤ 3600 function calls which are easily done within 1 second on a modern computer. A solution of this type should score 100%.

Mathematical Foundations

In one sense the IOI is a computer programming competition, since students are required to write a computer program to solve each task. There are no handwritten or plain-language solutions; the computer programs are the only things that are marked.

In another sense however, the IOI is a mathematics competition. The tasks are algorithmic in nature—the difficulty is not merely in the programming, but in devising a mathematical algorithm to solve the task. The computer program then serves as a means of expressing this algorithm, much as a written proof is the means of expressing a solution in a traditional mathematics contest.

Looking through recent IOIs, the following mathematical themes all make an appearance in either the problems or their solutions:

- Case analysis and simplification
- Combinatorics
- Complexity theory
- Constructive proofs
- Cryptanalysis
- Difference sequences
- Induction and invariants
- Game theory
- Geometry
- Graph theory
- Optimisation
- Recurrence relations

This list is of course not exhaustive. Although the IOI does not have an official syllabus as such, moves are being made in this direction; see [17] for a recent proposal by Verhoeff et al. that to a large degree reflects current practice.

2 More Interesting Problems

Whilst the example problem given in Section 1 is illustrative, it is not overly difficult. It is certainly useful at the junior level for highlighting the benefits of lookup tables, but the mathematics behind it is relatively simple.

It is often the case that more interesting mathematical ideas do not surface in informatics olympiad problems until the senior level. This is partly because (at least in Australia) algorithm design is rarely taught at secondary schools, and so challenging algorithmic problems must wait until students have had a little training.

In this section we turn our attention to senior level problems with more interesting mathematics behind them. Two of the author's favourite problems are discussed in some detail in their own sections below. For other examples of real olympiad problems, see [16] for an archive of past IOI problems, or see [9] for a thorough discussion of one particular IOI problem including an analysis of several potential solutions.

Polygon Game

This problem is not from an IOI per se; instead it is from the 2002 Australian team selection exam. It is included here because of its relation to the Catalan numbers, a combinatorial sequence with which some mathematics olympiad students might be familiar.

The problem itself is outlined in Figures 3, 4. Essentially students are given a polygon to triangulate; each triangulation is given a "score", and students' programs must determine the largest score possible. Given the bound $n \leq 120$ and the time limit of 1 second (and the speeds of computers back in 2002), an efficient algorithm should run in the order of n^3 operations or faster.

Brute Force Solution

A simple "brute force" solution might be to run through all possible sequences of valid moves. This does guarantee the correct answer, but it certainly does not run within the order of n^3 operations—instead the running time is exponential in n (with some arithmetic it can be shown that 4^n is a reasonable approximation).

Indeed, even if we note that the order of moves does not matter, the worst case $n = 120$ still leaves us with $\sim 5 \times 10^{67}$ possible sequences to consider.[2] Assuming a modern machine and extremely fast code, this is unlikely to finish in 10^{50} years, let alone the 1 second that the program is allowed. Thus the brute force solution is correct but incredibly inefficient, and would probably score 5–10% in a real competition.

Greedy Solution

For a faster strategy we might use a "greedy" solution that simply chooses the best available move at each stage. That is, the program looks at all lines that can be legally drawn and chooses the line with the

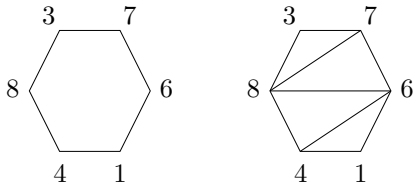
[2]The number of ways in which an n-gon can be triangulated is the Catalan number $C_{n-2} = \binom{2n-4}{n-2}/(n-1)$. Using the approximation $C_k \sim 4^k/k^{3/2}\sqrt{\pi}$, this gives $C_{118} \sim 4.86 \times 10^{67}$ for the case $n = 120$. See [15] for details.

Polygon Game

(B. Burton, G. Bailey)

You are given a regular polygon with n vertices, with a number written beside each vertex. A *move* involves drawing a line between two vertices; these lines must run through the interior of the polygon, and they must not cross or coincide with any other lines. The *score* for each move is the product of the numbers at the two corresponding vertices.

It is always true that after making precisely $n-3$ moves you will be left with a set of triangles (whereupon no more moves will be possible). What is the *highest possible score* that can be achieved through a sequence of valid moves?



As an example, consider the polygon illustrated in the first diagram above. Suppose we join the leftmost and rightmost vertices, the leftmost and top right vertices, and the rightmost and bottom left vertices, as illustrated in the second diagram. This gives a total score of $8 \times 6 + 8 \times 7 + 4 \times 6 = 128$, which in fact is the highest possible.

Input: The input file `polygon.in` will consist of two lines. The first line will contain the integer n, and the second line will list the numbers at the vertices in clockwise order around the polygon.

Output: The output file `polygon.out` must consist of one line giving the highest possible score.

Limits: Your program must run within 1 second, and it may use at most 16 Mb of memory. It is guaranteed that each input file will satisfy $3 \leq n \leq 120$.

Figure 3: The problem *Polygon Game* from the 2002 Australian team selection exam

Sample Input and Output: The following input and output files describe the example discussed earlier.

`polygon.in`:

```
6
3 7 6 1 4 8
```

`polygon.out`:

```
128
```

Figure 4: The *Polygon Game* input and output

greatest score (and then repeats this procedure until no more lines can be drawn).

This is certainly more efficient than brute force. For each move there are roughly n^2 possible lines to consider (more precisely $n(n-1)/2$), and this procedure is repeated for roughly n moves (more precisely $n-3$). Thus the entire algorithm runs in the order of n^3 operations, and is thereby fast enough for our input bounds and our time limit.

However, is this algorithm correct? Certainly it works in the example from Figures 3, 4. The best line that can be drawn inside an empty polygon has score 8×7, and the second best has score 8×6. The third best has score 7×4, but we cannot draw it because it would intersect with the earlier 8×6; instead we use the fourth best line, which has score 6×4. At this point no more lines can be drawn, and we are left with a total score of 128 as illustrated in the problem statement.

Nevertheless, a cautious student might worry about whether choosing a good move early on might force the program to make a *very* bad move later on. Such caution is well-founded; consider the polygon illustrated in Figure 5. The greedy solution would choose the 5×5 line (which is the best possible), and would then be forced to choose a 5×1 line for its second move, giving a total score of $25 + 5 = 30$ as illustrated in the leftmost diagram of Figure 5. However, by choosing a smaller 20×1 line for the first move, we open the way for another 20×1 line for the second move, giving a greater score of $20 + 20 = 40$ as shown in the rightmost diagram.

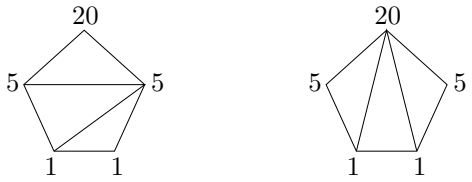


Figure 5: A case that breaks the greedy solution

This greedy solution was in fact submitted by a large number of students in the 2002 Australian team selection exam, where this problem originally appeared. It is a significant task when teaching informatics olympiad students to help them understand the difference between solutions that "feel good" (such as the greedy solution) and solutions that can be *proven correct* (such as the solution we are about to see). In an informatics olympiad where written proofs are not required, students need a healthy sense of self-discipline to spend time with pen and paper verifying that their algorithms are correct.

Correct Solution

It seems then that what we need is an algorithm that tries all possibilities—either directly or indirectly—but that somehow manages to identify common tasks and reuse their solutions in a way that reduces the running time from the exponential 4^n to the far more desirable n^3.

This is indeed possible, using a technique known as *dynamic programming*. Dynamic programming is the art of combining generalisation, recurrence relations and lookup tables in a way that significantly improves running time without any loss of thoroughness or rigour. We outline the procedure for Polygon Game below.

(i) *Generalisation:* Instead of solving a single problem (find the maximum score for the given polygon), we define an *entire family* of related "subproblems". Each subproblem is similar to the original, except that it involves only a portion of the original polygon. These

subproblems become our common tasks that can be reused as suggested earlier.

Suppose the vertices of the given polygon are $v_1, v_2, \ldots, v_n$. For any $i < j$ we define the subproblem $P_{i,j}$ as follows:

> *Consider the polygon with vertices $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$, as illustrated in Figure 6. Find the largest possible score that can be obtained by drawing lines inside this polygon (including the score for the boundary line $\overline{v_i v_j}$).*

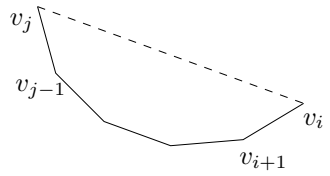


Figure 6: The polygon used in the subproblem $P_{i,j}$

In the cases where i and j are very close ($j = i + 1$ or $j = i + 2$), the subproblem $P_{i,j}$ becomes trivial since no lines can be drawn at all. On the other hand, where i and j are very far apart ($i = 1$ and $j = n$) we are looking at the entire polygon, and so $P_{1,n}$ is in fact the original problem that we are trying to solve.

Note that the boundary line $\overline{v_i v_j}$ causes some sticky issues; in most cases it receives a score, but in extreme cases (such as $j = i + 1$) it does not. We blissfully ignore these issues here, but a full solution must take them into account.

(ii) *Recurrence relations:* Now that we have our family of subproblems, we must find a way of linking these subproblems together. Our overall plan is to solve the smallest subproblems first, then use these solutions to solve slightly larger subproblems, and so on until we have solved the original problem $P_{1,n}$.

The way in which we do this is as follows. Consider the polygon for subproblem $P_{i,j}$. In the final solution, the line $\overline{v_i v_j}$ must belong to some triangle. Suppose this triangle is $\Delta v_i v_k v_j$, as illustrated in Figure 7. Then the score obtained from polygon $v_i \ldots v_j$ is the score

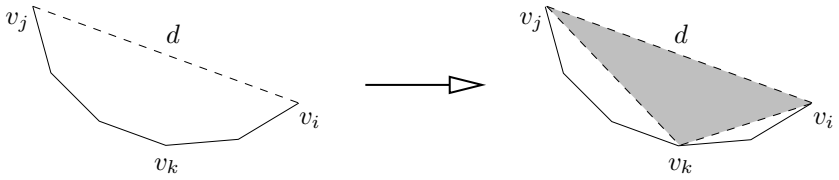


Figure 7: Splitting $P_{i,j}$ into smaller subproblems

for line $\overline{v_i v_j}$ plus the best possible scores for polygons $v_i \dots v_k$ and $v_k \dots v_j$. Running through all possibilities for vertex v_k, we obtain:

$$\begin{aligned}\text{Solution}(P_{i,j}) = \text{Score}(\overline{v_i v_j}) \\ + \max_{i<k<j} \{\text{Solution}(P_{i,k}) + \text{Solution}(P_{k,j})\}.\end{aligned}$$

Students familiar with the Catalan numbers might recognise this recurrence relation, or at least the way in which it is obtained. The Catalan numbers arise from *counting* triangulations of polygons (rather than maximising their scores). The recurrence relation that links them is based upon the same construction of Figure 7, and so takes a similar form (the max becomes a sum, additions become multiplications, and the subproblem $P_{i,j}$ becomes the Catalan number C_{j-i-1}). The Catalan numbers are a wonderful sequence for exploring combinatorics and recurrence relations, and interested readers are referred to [15] for details.

(iii) *Lookup tables:* Now that we have a recurrence relation, our overall plan is straightforward. We begin by solving the simplest $P_{i,j}$ in which i and j are very close together ($j = i+1$ and $j = i+2$, where no lines can be drawn at all). From here we calculate solutions to $P_{i,j}$ with i and j gradually moving further apart (these are solved using our recurrence relation, which requires the solutions to our earlier subproblems). Eventually we expand all the way out to $P_{1,n}$ and we have our final answer.

At first glance it appears that this algorithm could be very slow, since the recurrence relation involves up to $(n-2)$ calculations inside the $\max\{\dots\}$ term, each involving its own

smaller subproblems. However, we avoid the slow running time by storing the answers to each subproblem $P_{i,j}$ in a lookup table, similar to what we did for Pascal's Triangle in Section 1.

There are roughly n^2 subproblems in all (more precisely $n(n-1)/2$). For each recurrence we simply look up the answers to the earlier subproblems in our table, which means that each new subproblem requires at most n additional steps to solve. The overall running time is therefore order of $n^2 \times n = n^3$, and at last we have our correct and efficient solution.

Utopia Divided

Our final problem is from IOI 2002, and indeed was one of the most difficult problems of that year's olympiad. Summarised in Figures 8, 9, this is a fascinating problem that could just as easily appear in a mathematics olympiad as an informatics olympiad. We do not present the solution here; instead the reader is referred to [10] for the details. We do however present an outline of the major steps and the skills required.

1. *Simplifying the problem:* The first crucial step is to simplify the problem to a single dimension. In this case the input becomes a sequence of n distinct positive integers (not $2n$ integers) and a sequence of n plus or minus signs (not n quadrants). Your task now is to rearrange and/or negate the n integers to create a one-dimensional path along the number line; this path must begin at 0 and travel back and forth between the $+$ and $-$ sides of the number line in the given order.

For example, suppose the integers were $6, 8, 10, 11$ and the signs were $+, -, -, +$. A solution might be to use steps 8, -10, -6 and 11 in order; here the path begins at 0 and then runs to the points 8, -2, -8 and 3, which matches the signs $+, -, -, +$ as required.

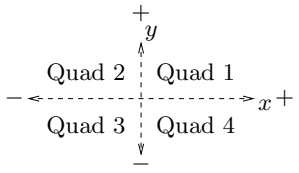
Certainly the one-dimensional problem is simpler to think about than the original two-dimensional version. More importantly, the one-dimensional solution is easily generalised to two dimensions—arbitrarily choose n of your integers to be x coordinates and the remaining n integers to be y coordinates, solve the problems in the x and y dimensions independently, and then combine the x and y solutions to obtain your final n two-dimensional steps.

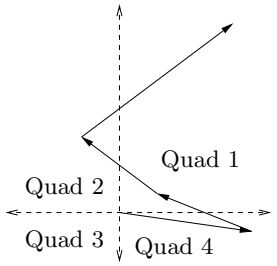
Utopia Divided

(S. Melnik, J. Park, C. Park, K. Song, I. Munro)

Consider the (x, y) coordinate plane, and number the four quadrants 1–4 as illustrated in the leftmost diagram below. You are given a sequence of $2n$ distinct positive integers, followed by a sequence of n quadrants. Your task is to create a path that begins at $(0, 0)$ and travels through these n quadrants in the given order.

You must create your path as follows. Your $2n$ positive integers must be grouped into n pairs; each of these pairs becomes a single (x, y) step along your path. You may negate some (or all) of these integers if you wish.





For example, suppose that $n = 4$ and you are given the sequence of integers `7 5 6 1 3 2 4 8` and the sequence of quadrants `4 1 2 1`. You can solve the problem by rearranging the integers into the following pairs:

$$(7, -1)\ (-5, 2)\ (-4, 3)\ (8, 6)$$

Following these steps from $(0, 0)$ takes you to the point $(7, -1)$, then $(2, 1)$, then $(-2, 4)$ and finally $(6, 10)$, as illustrated in the rightmost diagram above. This indeed takes you through quadrants 4, 1, 2 and 1 in order as required.

Figure 8: The problem *Utopia Divided* from IOI 2002

Input: The input file `utopia.in` will consist of three lines, containing the integer n, the sequence of $2n$ positive integers, and the sequence of n quadrants respectively.

Output: The output file `utopia.out` must consist of n lines, giving the n steps in order. Each line must list the x and y integer components of the corresponding step.

Limits: Your program must run within 2 seconds, and it may use at most 32 Mb of memory. It is guaranteed that each input file will satisfy $1 \leq n \leq 10\,000$.

Sample Input and Output: The following input and output files describe the example discussed earlier.

`utopia.in`:

```
4
7 5 6 1 3 2 4 8
4 1 2 1
```

`utopia.out`:

```
7 -1
-5 2
-4 3
8 6
```

Figure 9: The problem *Utopia Divided* input and output

2. *Broad plan:* After some thought one can formulate a broad strategy, which is to use larger numbers to change between + and − sides of the number line, and to use smaller numbers to stay on the same side of the number line. Indeed this is what we see in the example above—the smallest integer 6 is used when the path must run through the − side twice in a row, and the largest integers 10 and 11 are used when the path must change from + to − and from − to + respectively.

It then makes sense to split the sequence of integers into "large numbers" and "small numbers" (just how many large and small numbers you need depends upon how many times you must change sides). Whenever you need to change sides, you pull an integer from your large set; otherwise you pull an integer from your small set.

An important factor to consider in this plan is the *order* in which you use the large and small numbers. With some thought it can be seen that the large numbers should be used in order from smallest to largest, and the small numbers should be used in order from largest to smallest. This ensures that, as your position becomes more erratic over time, the large numbers are still large enough to change sides, and the small numbers are still small enough to avoid changing sides.

3. *Details and proofs:* Some important details still remain to be filled in; this we leave to the reader. In particular, when using a small number, the decision of whether to add or subtract requires some care.

Moreover, once the algorithm is complete, it is not immediately clear that it works for all possible inputs—some proofs are required. One approach is to place upper and lower bounds upon the kth point along the path, and prove these bounds using induction on k. The details can be found in [10].

As mentioned earlier, this was a difficult IOI problem—it received the lowest average score of all six problems in IOI 2002 (and in the author's opinion, IOI 2002 was one of the most difficult IOIs of recent years). Nevertheless, like the solutions to so many mathematics problems, the algorithm is simple and sensible once seen. It is the process of identifying the algorithm and proving it correct that creates so much enjoyable frustration during the competition.

3 Benefits and Challenges

As discussed in Section 1, informatics olympiad problems are often mathematical in nature, with regard to both the content and the skills that they require. In the author's view, an informatics olympiad can indeed be viewed as a type of mathematics competition, in which solutions are communicated through code (computer programs) instead of the more traditional written answers.

With this in mind, it is useful to examine both the benefits that informatics olympiads offer over traditional mathematics competitions,

and the difficulties that they present. We spread our discussion across three broad areas: enjoyment and accessibility, skills and learning, and judging.

The points made here merely reflect the author's own opinion and experiences; there are of course many variants of both mathematics and informatics competitions with their own different strengths and weaknesses. Readers are heartily encouraged to participate themselves and come to their own conclusions!

Enjoyment and Accessibility

Here we examine the ways in which students are encouraged to participate in different competitions, and the barriers that prevent them from doing so.

On the positive side, many students enjoy working with computers and find computer programming to be fun. In this way, informatics competitions can expose them to mathematical ideas using what is already a hobby—although the students are not explicitly setting out to study mathematics (which some of them might not even enjoy at school), they nevertheless develop what are essentially mathematical skills and ideas.

Likewise, a number of students are afraid of writing mathematical proofs. In an informatics olympiad they only need to submit computer programs, which for some students are rather less onerous to write—computer programs are concrete things that students can build, run, test and tinker with. Nevertheless, as problems become harder students must develop the underlying skills of proof and counterexample, in order to distinguish between algorithms that *feel* correct and algorithms that *are* correct.

On the negative side, there are some very clear barriers to involvement in informatics olympiads. The first (and highest) barrier is that students cannot participate unless they can write a working computer program from scratch within a few hours. Unfortunately this alone rules out many bright and otherwise eager students. Another barrier is the fact that informatics olympiads are difficult for schools to run—they require computers, compilers and sometimes network access. In

general informatics olympiads cause far more trouble for teachers than a mathematics competition where all a student needs is a desk and an exam paper. Recent moves have been made by countries such as Lithuania [8] and Australia [5, 6] to work around these problems by offering multiple-choice competitions that test algorithmic skills in a pen-and-paper setting.[3]

Skills and Learning

Although competitions are intended to be fun, they are also intended to be educational. Here we examine the skills that students discover and develop from participating in such competitions.

A strong advantage of informatics competitions is that they expose students to fields that in many countries are rarely taught in secondary schools. Certainly in Australia, schools tend to focus on using computers as a tool—courses in computer programming are less common, and even then they tend to focus on translating ready-made algorithms from English into computer code. The *design* of algorithms is almost never discussed.

A positive side-effect of automated judging is that informatics olympiad problems are often well suited for self-learning. Several countries have online training sites [4, 11], where students can attempt problems, submit their solutions for instant evaluation and refine them accordingly. Even without online resources, students can critique their solutions by designing test cases to feed into their own programs.

Informatics olympiads also develop a sense of rigour, even in the easiest problems. This comes through the hard fact that students cannot score any points without a running program—even with slow or naïve algorithms, a certain attention to detail is required. Combined with the fact that official data sets typically include pathological cases, students learn early on that rigour is important.[4]

[3] Both contests are now offered outside their countries of origin; see [2] and [3] for details.

[4] In some events such as the ACM contest [1], students score no points at all unless their programs pass *every* official test case. Whilst this might seem harsh, it puts an extreme focus on rigour that has left a lasting impression on this author from his university days, and that has undeniably benefited his subsequent research into mathematical algorithm design.

A clear deficiency in informatics olympiads is that they do not develop communication skills. Whereas mathematics students must learn to write clear and precise proofs, informatics olympiad students write computer programs that might never be read by another human. Bad coding habits are easy to develop in such an environment.

Another problem is that, since solutions are judged by their behaviour alone, one cannot distinguish between a student who guesses at a correct algorithm and a student who is able to *prove* it correct. For harder problems guesswork becomes less profitable, but for simpler problems it can be difficult to convince students of the merits of proof when their intuition serves them well.

Judging

Competitions are, when it comes to the bottom line, competitive. In our final point we examine the ways in which students are graded and ranked in different competitions.

On the plus side, weaker students in informatics competitions can still obtain respectable partial marks even if they cannot find a perfect algorithm. For instance, students could still code up the brute force or greedy solutions to Polygon Game (Section 2) for a portion of the available marks.

On the other hand, a significant drawback is that students cannot score *any* marks without a running program. A student who has found the perfect algorithm but who is having trouble writing the code will score nothing for her ideas.

In a similar vein, the marking is extremely sensitive to bugs in students' programs. A perfect algorithm whose implementation has a small bug might only score 10% of the marks because this bug happens to affect 90% of the official test cases. It is extremely difficult to design official data sets that minimise this sensitivity but still subject good students to the expected level of scrutiny.

As a result of these sensitivities, exam technique is arguably more important in an informatics olympiad than it should be. In particular, students who have found a perfect algorithm might be tempted to code

up a less efficient algorithm simply because the coding will be quicker or less error-prone. There has been recent activity within the IOI community to work around these problems (see [7] and [12] for examples), but there is still a long way to go.

4 Getting Involved

For students eager to become involved in the informatics olympiad programme, there are a number of avenues to explore.

- *Books:* Steven Skiena and Miguel Revilla have written an excellent book [13] specifically geared towards programming competitions such as the IOI. The book is very readable and full of problems, and includes outlines of the most prominent competitions in the appendix.
- *Online resources:* As discussed in Section 3, several countries have produced online training sites through which students can teach themselves. The USACO site [11] is an excellent resource with problems, reading notes and contest tips. The Australian site [4] is also open to students worldwide.
- *National contacts:* Students are encouraged to contact their national organisations for local contests and training resources. The IOI secretariat [16] has links to national organisations, as well as archives of past IOI problems and resources for other olympiads such as the IMO.

Informatics olympiads are often seen as belonging purely to the domain of computer science. However, when seen from a mathematical point of view, they offer new challenges and activities that complement traditional mathematics competitions. Certainly the author, trained and employed as a mathematician, has gained a great deal of pleasure from his involvement in informatics competitions; it is hoped that other teachers and students might likewise discover this pleasure for themselves.

References

[1] ACM ICPC, *ACM International Collegiate Programming Contest website*, `http://acm.baylor.edu/acmicpc/`, accessed September 2006.

[2] Australian Mathematics Trust, *Australian Informatics Competition website*, `http://www.amt.edu.au/aic.html`, accessed June 2007.

[3] Beaver Organising Committee, *Information technology contest "Beaver"*, `http://www.emokykla.lt/bebras/?news`, accessed July 2007.

[4] Benjamin Burton, Bernard Blackham, Peter Hawkins, et al., *Australian informatics training site*, `http://orac.amt.edu.au/aioc/train/`, accessed June 2007.

[5] David Clark, *Testing programming skills with multiple choice questions*, Informatics in Education **3** (2004), no. 2, 161–178.

[6] David Clark, *The 2005 Australian Informatics Competition*, The Australian Mathematics Teacher **62** (2006), no. 1, 30–35.

[7] Gordon Cormack, Ian Munro, Troy Vasiga, and Graeme Kemkes, *Structure, scoring and purpose of computing competitions*, Informatics in Education **5** (2006), no. 1, 15–36.

[8] Valentina Dagiene, *Information technology contests—introduction to computer science in an attractive way*, Informatics in Education **5** (2006), no. 1, 37–46.

[9] Gyula Horváth and Tom Verhoeff, *Finding the median under IOI conditions*, Informatics in Education **1** (2002), 73–92.

[10] IOI 2002 Host Scientific Committee (ed.), *IOI 2002 competition: Yong-In, Korea*, Available from `http://olympiads.win.tue.nl/ioi/ioi2002/contest/`, 2002.

[11] Rob Kolstad et al., *USA Computing Olympiad website*, `http://www.usaco.org/`, accessed June 2007.

[12] Martins Opmanis, *Some ways to improve olympiads in informatics*, Informatics in Education **5** (2006), no. 1, 113–124.

[13] Steven S. Skiena and Miguel A. Revilla, *Programming challenges: The programming contest training manual*, Springer, New York, 2003.

[14] TopCoder, Inc., *TopCoder website*, `http://www.topcoder.com/`, accessed September 2006.

[15] J. H. van Lint and R. M. Wilson, *A course in combinatorics*, Cambridge Univ. Press, Cambridge, 1992.

[16] Tom Verhoeff et al., *IOI secretariat*, `http://olympiads.win.tue.nl/ioi/`, accessed September 2006.

[17] Tom Verhoeff, Gyula Horváth, Krzysztof Diks, and Gordon Cormack, *A proposal for an IOI syllabus*, Teaching Mathematics and Computer Science **4** (2006), no. 1, 193–216.

Benjamin A. Burton
Department of Mathematics, SMGS, RMIT University
GPO Box 2476V, Melbourne, VIC 3001
AUSTRALIA
Email: bab@debian.org