Central European Olympiad in Informatics
Brno, July 1 - 7, 2007

NEWSLETTER

Official Opening Ceremony

The Official Opening Ceremony started with the speech of the dean of Faculty of Informatics Prof. Jiří Zlatuška. The speech focused on Faculty of Informatics and Masaryk University. The Chairman of the CEOI 2007 Dr. Tomáš Pitner said a few words about the history of the CEOI and the IOI and then welcomed the participants. The Opening Ceremony was closed by a string quartet Clawed Forehead which played some compositions based on known rock music band Apocalyptica.

After the Opening Ceremony the Chairman of Scientific Committee Dr. Daniel Kráľ presented important issues concerning competitions rules and some practical instructions about computer equipment for the competition.

The contestants also visited competition room to test the computer equipment for the next day.

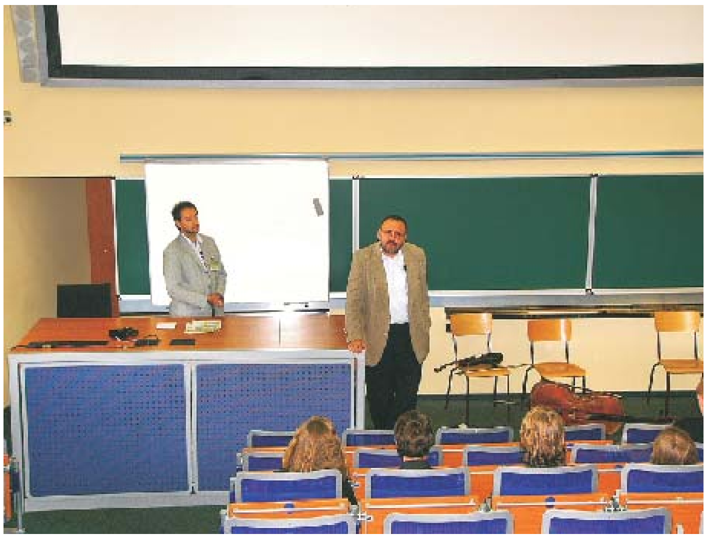
Fig. 1: The dean of FI MU and the Chairman of CEOI 2007

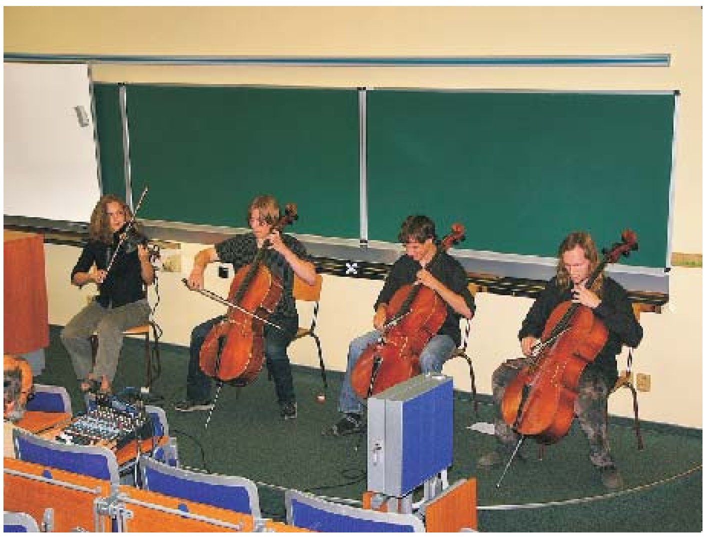
Fig. 2: Clawed Forehead band

Excursion to Brno

The program of contestants at the first day was closed by excursion to the center of Brno. The excursion led the contestants around the St. James' Church, Liberty Square, Old Town Hall, The Fruit and Vegetable Market to the Kapuchin St. Cross Church.

From there they walk over the hill to the Špilberk Castle and fortification. The old prison Casemates was visited at the end of the excursion.

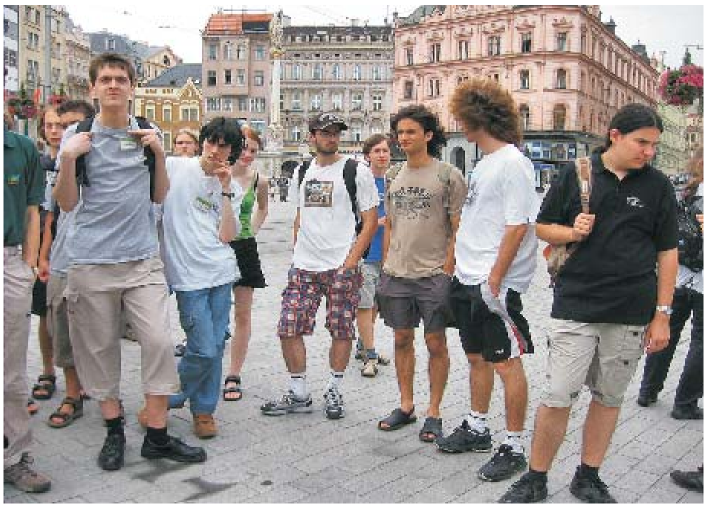
Fig. 3: Liberty Square

Algorithm Hints for Competition Tasks

Ministry

This task can be solved in a simple bottom-up way, that is by processing the departments in order of their increasing depths. To achieve that, we first represent the organizational tree by pointers to sons and parents, and sort the departments by their depths (which can be done in linear time by repeatedly plucking leaves of the tree). Then we observe that all departments of depth one are equivalent and we continue by establishing equivalence of the higher depth departments.

Let us assume we already know which departments of depths *1* to *d* are equivalent, for example we can have integer codes assigned to them in such a way that two de partments get the same code if and only if they are equivalent. Then we consider the departments of depth *d+1*. Two such departments are equivalent if and only if their bosses have the same number of subordinates and the subdepartments chair *d* by these subordinates are equivalent, except possibly for their order.

We can recognize that by sorting the subordinates of each boss by their already assigned codes and then by sorting the bosses by the triples of the subordinates' codes (if there are less than three subordinates, we can pad the triple by some special values). We finally scan the bosses in the sorted order, assign new codes to those who have different triples and proceed with the depth *d+2*.

This gives a nice $\mathcal{O}(n \log n)$ time solution, which can be further improved to linear time by employing three-pass bucket-sort for sorting the triples. This is a little bit tricky as you cannot afford scanning all the possible buckets all the time, but you can keep a list of active buckets. As we do not really need to have the triples sorted, only the equivalent ones have to be brought next to each other, we can keep the list of active buckets in any order. However, due to the large multiplicative constants in the linear time complexity, the difference between this solution and the previously mentioned $\mathcal{O}(n \log n)$ one is negligible and both of them could have achieved full score 👍.

If you have any questions or if you are just curious about some other applications of such multi-level sorting, feel free to ask Martin or Milan ☺.

Nasty

There are two main steps leading to a solution of this task: evaluating of expressions written in the postfix notation and reducing unnecessary calculations. Let us start with the former issue. The classical approach to evaluating such expressions is the use of a stack (e.g., implemented in an array) to store intermediate results of the calculation. You process the expression in a single sweep: if you encounter a number or the symbol for *x*, you put it on the stack, and if you find an operator, you remove the top two numbers from the stack and store back the result of the operator applied to these numbers. After processing the entire expression, the only number on the stack is the result.

The first paragraph already suggests an algorithm for calculating the value of *f(x)*, but... wait, we only need the last digit, right? Since we know that *f(x)* is non-negative, we have to compute the value of *f(x)* mod *B*. And since the mod *B* operation behaves well with all the operands which might occur in the expression, we can use *a* mod *B* instead of *a* in all calculations. In particular, we do not have to implement long number arithmetic. Finally, since the value of *f(x)* mod *B* depends only on *x* mod *B*, we can precompute the results for *x = 0, ..., B-1* and use the stored values instead of evaluation of *f(x)* for every *x* in order to save time.

Sail

This was clearly the most difficult task of the first day. The problem of reconstructing the sail is known to be NP-complete, which means that there is only a little hope that there exists a reasonably fast program for solving the problem in general.

However, for the open-data task, we prepared input data either small enough or satisfying some additional properties which make finding solution more feasible.

The first two input files were supposed to be solved by hand, using either the program `draw_sail` or a pen with a sheet of paper. The third input file contained eight types of triangles, each of them 64 times. The dimensions of the original sail were 64×64. A natural idea seems to be to check whether the eight triangles (representatives of each type) cover a sail of dimension 8×8. Since they do, a solution can be obtained by repeating the cover of the 8×8 sail 64 times.

Sail (continuation)

The fourth case requires a different approach. If you sum up the lengths of all edges of the triangles which are parallel either with the x or y axis, the obtained value is equal to the perimeter of the original sail. Hence, the sides of the triangles parallel to the x axis must form the two sides of the sail parallel to the x axis. An analogous statement is true for those parallel with the y axis. Since the triangles have only four different heights (with respect to the side parallel to one of the axes), one can suspect the existence of a single point X such that the sail was cut by several cuts starting in X and ending in a point at the perimeter. This can be verified with a simple program and drawing the resulting sail layout. The fifth input file contains a similar type of data, but instead of a single point inside, there is a triangle.

The sixth and seventh input files can be dealt with using the same approach. They contain several right-angled triangles with catheti parallel to the axes. The lengths of the catheti parallel with the x axis are half the width of the sail in the sixth input file and one third in the seventh. Moreover, the triangles can be paired to form rectangles. If the resulting rectangles could be partitioned into two/three groups in such a way that the sum of the heights of the rectangles of each group were equal to the height of the sail, you would get solutions of these two cases (the groups would form two or three columns of rectangles forming the original sail). Partitioning the rectangles into the groups is the well-known thieves problem. In this problem, several items of possibly different values should be divided into a given number of groups, each containing items with the same sum of values. The problem can be solved using dynamic programming (if the number of groups is fixed).

The last three instances could be solved by a clever backtracking algorithm. The eighth and ninth input files contain triangles with sides parallel to the y axis and with the heights (corresponding to the y axis) equal to the width of the sail. Hence, it is enough to stack up the triangles in a single column. Again, this might be quite hard in general, but the angles of the triangles contained in the eighth input file are all different and they are only few duplicates in the ninth case. Either way, it is easy to find the right order of the triangles in the column by backtracking. Finally, the sail corresponding to the tenth input file is comprised of three such columns with different widths.

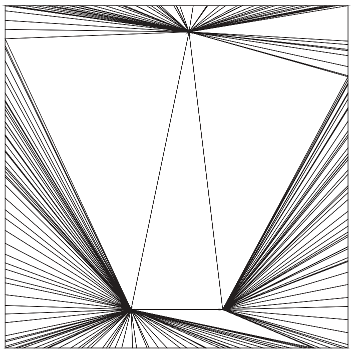
Fig. 4: Sail

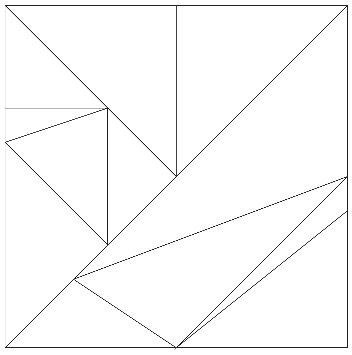
Fig. 5: Another Sail

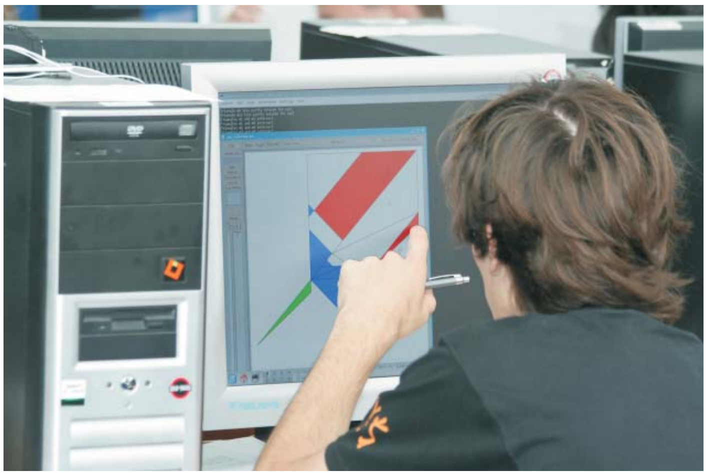
Fig. 6: Solution in a standard way

Fig. 7: Solution in an old school way I

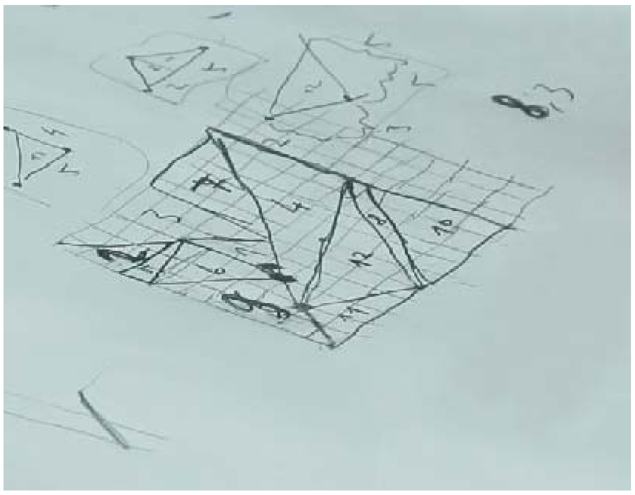
Fig. 8: Solution in an old school way II