IOI'94 - Day 2 - Solution 1: The Clocks

[Introduction] [Problem Statement] [Test Data]

Problem Analysis

At first sight, this may look like a backtrack problem, in which the program systematically tries move sequences until one is found that turns the dials to the desired state (12 o'clock). There is, however, a difficulty in this approach.

In which order should we try the move sequences? We need to make sure that we do not miss the right one(s). If we cannot give an upper bound on the length of candidate move sequences, then we should, for instance, try them in order of increasing length. First we deal with all move sequences of length zero (only one), next all of length one (only nine), then length two, etc. This is called a *breadth-first search*. It is often more complicated then a *depth-first search* (where the move sequences are tried in some lexicographic order).

If the desired move sequence is sufficiently short, then it can be found quickly by a breadth-first search. But the problem statement did not say that all input would result in short move sequences. We have to take longer sequences into account as well.

Let us assume for a moment that we need no more than k moves of each of the nine move types. In that case the maximum length of a move sequence is 9k and there are (9k)!/((k!)^9) such maximal move sequences. For k=1 this equals 9!=3.6e5, for k=2 we get 18!/2^9=1.3e13, and for k=3 it is 27!/6^9=1.1e21. Of course, all the shorter move sequences should be considered as well. Therefore, these numbers do not look promising. For k=3 we cannot hope to try all sequences within the time limit. Even for k=2 we are hard pressed. Hoping for k=1 is wishful thinking.

We need a better idea. The effect of a move is to turn a subset of the dials over 90 degrees (clockwise). The net effect of two moves executed one after the other is cumulative. If move 1 turns clocks A and B over 90 degrees, and move 2 turns B and C over 90 degrees, then their combination turns A and C over 90 degrees and B over 2*90=180 degrees. Therefore, the net effect does not depend on the order of execution. (This resembles the *superposition principle* for waves, fields, etc. in physics.) Consequently, for determining the net effect of a move sequence on the dials, we need to know only *how many* moves of each type are involved and not *in what order* the moves are made.

The observation above drastically reduces the number of candidate move sequences to be considered. First of all, it is now obvious that each move type need not appear more than three times, because four quarter turns is the same is no turn at all. Secondly, since each of the 9 move types can appear from 0 to 3 times, the number of candidate move sequences equals 4^9=262,144. This can be accomplished in the given time.

Program 1

Program 1 tries all candidate move sequences in nine nested for-loops until a solution is found. (N.B. It was stated in the Competition Rules that every input file would have a solution.) Program 1 relies on the fact that doing one type of move four times is the same as doing nothing.

This program has not been optimized in any way. It shows that a rough idea may work all right. Anything you do to improve it is a waste of (your) time, since Program 1 provides the solution well within the time limit.

However, Program 1 leaves a number of interesting questions unanswered. Furthermore it is still not very efficient, even though it is fast enough for our purpose. There are two questions that the jury needed to answer in order to judge the programs:

1. Can every starting configuration be solved?
2. How many solutions (modulo 4) does a solvable starting configuration have?

Observe that the number of starting configurations is 4^9, because for each of the nine clocks there is a choice among four states. This is the same as the number of candidate move sequences for a solution. Therefore, either (i) each starting configuration has a unique solution, or (ii) there are some starting configurations without solution and some with multiple solutions.

Exercise: Modify Program 1 to generate all solutions. Because the problem statement does not restrict the input, we have assumed in Program 1 that each starting configuration has a solution, which then is unique modulo 4.

Given a move sequence it is easy to compute its net effect on the dials. Assume move type j occurs tj times in the sequence. Figure 2 in the problem statement implicitly tells us how clock A is affected by the moves. Here is that table again in a slightly different format:

```
Move   Affected clocks

 1     A B . D E . . . .
 2     A B C . . . . . .
 3     . B C . E F . . .
 4     A . . D . . G . .
 5     . B . D E F . H .
 6     . . C . . F . . I
 7     . . . D E . G H .
 8     . . . . . . G H I
 9     . . . . E F . H I
```

We conclude that the net effect on dial A is t1+t2+t4 quarter turns. Here is a complete table giving for each clock the moves that affect it (essentially obtained by reflecting the matrix above in the upper-left-to-lower-right diagonal, also called *transposition*):

```
Clock   Affected by moves

 A      1 2 . 4 . . . . .
 B      1 2 3 . 5 . . . .
 C      . 2 3 . . 6 . . .
 D      1 . . 4 5 . 7 . .
 E      1 . 3 . 5 . 7 . 9
 F      . . 3 . 5 6 . . 9
 G      . . . 4 . . 7 8 .
 H      . . . . 5 . 7 8 9
 I      . . . . . 6 . 8 9
```

Let us denote the state of dial V before the move sequence by v and its new state after executing the sequence by v'. The new states are related to the old states by the following equations (where addition is taken modulo 4):

```
 a' = a + t1 + t2 + t4
 b' = b + t1 + t2 + t3 + t5
 c' = c + t2 + t3 + t6
 d' = d + t1 + t4 + t5 + t7
 e' = e + t1 + t3 + t5 + t7 + t9
 f' = f + t3 + t5 + t6 + t9
 g' = g + t4 + t7 + t8
```

```
h' = h + t5 + t7 + t8 + t9
i' = i + t6 + t8 + t9
```

Consequently, the problem our program has to solve is now translated into solving a system of nine *linear algebraic equations* with nine unknowns (t1, ..., t9), where the vector (a, ..., i) is the given initial state and the final state (a', ..., i') is the all-zero vector. All the coefficients of the unknowns are apparently either 0 or 1. We have put the coefficient matrix in text file matrix.dat.

Program 2

There are numerous ways to solve a system of linear algebraic equations. Program 2 is based on a straightforward method often learned in school, called *Gauss-Jordan elimination*. It reads the coefficients from matrix.dat.

Note that the execution time of Program 1 depends on the actual input (in the worst case it takes 4^9 steps through the inner loop, in the best case only one). The execution time of Program 2 is almost independent of the input (procedure Solve takes on the order of 9^3=729 steps because there are three nested for-loops with at most 9 steps each). Program 2 is often faster than Program 1, but it is also more complicated and offers more opportunities for making mistakes when implementing it. Therefore, I cannot recommend it in the context of IOI'94.

Program 2 does help answer the above questions. The method it uses to solve the system of linear equations depends only in part on the input. The manipulations with the matrix A of coefficients (in procedure Solve) are independent of the input (only array b depends on the input). If the program is able to solve the problem for one particular input file, then we know that it will also succeed in solving the problem for every other input file. A simple trial run reveals that Program 2 works for the example input file input-0.txt. This proves that each input file has a solution and hence we know that the solutions are unique (modulo 4).

Program 3

Because the manipulations with the coefficient matrix A do not depend on the input, they can be done in a separate pre-processing phase. In particular, the coefficient matrix for the system of linear equations can be *inverted*. The inverse matrix can then be used to compute solutions very quickly. Program invert.pas reads the coefficients from matrix.dat and writes the inverse matrix to the text file inverse.dat. It is obtained from Program 2 by replacing the linear array b in procedure Solve by a square array B initialized to the identity matrix.

Program 3 uses this inverse matrix to compute a solution by a single matrix-vector multiplication (which takes on the order of 9^2=81 steps). This solution is extremely fast. It would be even faster if the coefficients would not be read from a file but were incorporated in the program to initialize the matrix. In Turbo Pascal this is not so difficult, but for other Pascal versions that may require 81 assignment statements. In fact, if you strip the program to the bone (you don't have to use an array) you can squeeze it into a few lines. Here is a solution to the above system of nine equations in (t1,...,t9) for given (a,...,i) and (a',...,i')=(0,...,0) derived from the inverse matrix (addition and multiplication are modulo 4):

```
t1 = 8+ a+2b+ c+2d+2e -f+ g -h
t2 =    a+ b+ c+ d+ e+ f+2g+   2i
t3 = 8+ a+2b+ c -d+2e+2f    -h+ i
t4 =    a+ b+2c+ d+ e+    g+ h+2i
t5 = 4+ a+2b+ c+2d -e+2f+ g+2h+ i
t6 =   2a+ b+ c+    e+ f+2g+ h+ i
t7 = 8+ a -b+   2d+2e -f+ g+2h+ i
t8 =   2a+   2c+ d+ e+ f+ g+ h+ i
```

```
t9 = 8    -b+ c -d+2e+2f+ g+2h+ i
```

Program 4, based on this solution, may be instructive but it is not a recommended IOI practice.

Variants of this problem

This problem derives from Rubik's clock puzzle. In Rubik's puzzle all clocks have 12 states. Each move sets the (affected) clocks one hour forward (clockwise). But there are also the inverse moves that set the (affected) clocks backward one hour (counterclockwise).

Solve the problem for Rubik's (12-hour) clock puzzle, minimizing the number of moves (a backward move counts as one move).

Tom Verhoeff
Eindhoven University of Technology