All the solutions were made by the organisers of BOI'2001.

Excursion

Solution can be found in a separate .pdf file in Polish language only.

Box of Mirrors

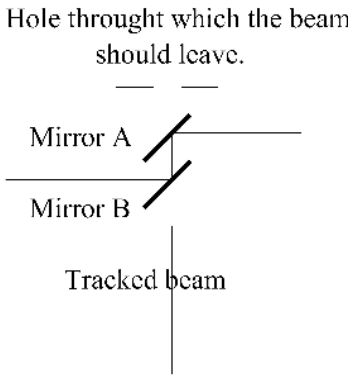
The correct solution is a greedy algorithm, that tries to put mirrors in such way that every light beam goes as much as possible upwards. We will successively track beams lighted into gaps with numbers 1,2,… n+m.

Let us consider the following algorithm:

1. Let x,y be the current column and row of the tracked beam. Let x',y' be the column and row of the gap through which the beam should leave.
2. If x=x' and y=y' then stop.
3. If beam goes vertically go to point 5.
4. If there is a mirror above the current position and $x \neq x'$ or $y = y'$ go one cell to the right: x := x+1.
Otherwise, place mirror in the cell x,y and go one cell up: y:= y -1 .
Go to point 2.

5. If there is a mirror in the current cell, go right: x:=x+1.
Otherwise go up: y:=y-1. Go to point 2.

To prove the correctness of this algorithm, we will consider the only bad situation:



The beam should leave with the gap above, but on the way there is a mirror B. This situation can not occur because there is no light reflected by the upper side of a mirror A. And in our algorithm the mirror is placed only when the beam is reflected.

Tests.
Tests are random, with different sizes and different number of mirrors.

Postman

Remarks:

– in graph presented in tasks always have Euler cycle (all vertex degrees are even),
– Euler cycle satisfy all conditions given in task,
– the total profit does not depend on chosen Euler cycle,

$$profit = \sum_{i=1}^{n}(w(i) - k(i)) = \sum_{i=1}^{n} w(i) + \sum_{i=1}^{n} k(i) = \sum_{i=1}^{n} w(i) + \sum_{i=1}^{n} i$$

– tour which visits all edges, and some of them more then once, is always worse then Euler cycle (because postman pays 1 euro for each road).
– so the optimal solution simply ignore all w(i) values, and returns Euler cycle (randomly choosen).

Prog/poschk.pas

Because for some tests there is more then one correct answer, external checker is needed. The checked depends on input and output file in current directory.

Tests

test 0 – example test
test 1 – simple test (manualy choosen), n=7
test 2 – random test (type1), n=10
test 3 – ladder + extra edges, n=20
test 4 – random test (type2), n=40
test 5 – loop (two side), n=100
test 6 – random test (type1), n=100
test 7 – flower (four connected loops), n=120
test 8 – random test (type2), n=150
test 9 – ladder, n=200
test 10 – random test (type2), n=200

Crack the Code

1. Problem Analysis
1.1. Coding program (crack.pas)
The crack.pas program codes (one by one) the successive characters form the input file. Characters that are not letters (of the English alphabet) are left unaltered. The key given by the user specifies the way characters are coded. Namely, the i-th character from the input file (counting from 1) is “rotated” in a “cyclic” English alphabet (consisting of capital letters only) by the value of the [(i-1)mod 10+1]-th element of the key. (Elements of the key are also numbered from 1).

1.2 Reverse engineering the coding algorithm

It can be easily seen, that knowing the key we can decode the message in a similar way it was coded–we can apply the coding algorithm given above for a key consisting of negated elements of the coding key. Hence, the problem reduces to finding the coding key. In case of texts written in the natural language, the relative frequencies of particular letters are different. So, we can use some statistical methods to determine the key. Basing on the uncoded piece of text we can calculate the probability with which particular letters appear in the text. We will denote by p_i the probability that (the next) character in the text is i , and by P the probability distribution itself. To find the i-th element of the key, we calculate the probability distribution for letters appearing in the coded text on positions i, (i+10), (i+20), ... We will denote by q_i the probability of appearance of letter i , and by Q the the probability distribution itself. Now, it is enough to find such a cyclic rotation of sequence (q_A, q_B, ..., q_Z) that is closest to the distribution P .

2. Exemplar Solution

prog/cra.pas.

Our solution is based on the method described above. We also need a measure of how "close" the distribution P is to R = (r_A, r_B, ..., r_Z) = (..., q_Z, q_A, q_B, ...) (some cyclic rotation of the sequence (q_A, q_B, ..., q_Z)). We will use here statistical test χ^2 , which can be expressed as:

$$n\sum_{i=A}^{Z}\frac{(p_i - r_i)^2}{p_i};$$

where n is the number of letters in the sample on which Q is based.

Remark. The test χ^2 should be used only in the cases when for each i the value np_i is not too small (let's say, not less than 0.1). Therefore, sometimes it maybe necessary to group a few events in the distribution P , so that their total probability is big enough, relatively to n .

3. Other Solutions

prog/cra1.pas

This solution differs from the one above only in the measure used to compare probability distributions. We can view the distributions P and R as vectors in a 26-dimensional space (26 is the number of letters in the alphabet). We can calculate their dot product and normalise it, dividing it by the square root of the product of lengths of the two vectors. The closer the result is to 1, the closer are the two distributions to each other.

4. Tests

There are 5 test cases. Each of them consists of two files:

–in/cra*n*.in – coded message,

–in/cra*n*.txt – text sample of the same origin as the coded message.

All the messages are in English. Texts differ in the difficulty level – the longer the text sample and coded message, the more reliable are the results of the statistical analysis. In case of test no 5, program cra1.pas fails to decode correctly all of the characters,

however most of them is decoded correctly and the resulting file can be corrected manually. Program cra.pas, however, solves this problem correctly.

Files out/cra*n*.out contain decoded messages form files in/cra*n*.in. The testing software should compare the output files delivered by contestants with the files in the /out directory.

Knights

1. Introduction
} The task is a hard one. Solution is based on the "Hungarian theorem". Basing on the theorem, the problem can be reduced to finding a maximal matching in a bipartite graph. Maximum data size (40000 fields, i.e. 40000 vertices) requires best of known algorithms, e.g. Hopcroft-Karp's algorithm.

2. Problem analysis.
We will build a graph, whose vertices are the fields on the chess-board, and edges represent single moves of the knight. In a single move knight is always jumping from a white field onto a black one, or vice versa, hence the graph is a bipartite one. The maximal independent set of vertices is what we are looking for.

The completion of the independent set of vertices form a vertex-covering of the graph. One of the formulations of the Hungarian theorem say, that in any bipartite graph the size of the smallest vertex-covering is equal to the size of the maximal matching. Hence, if M is a maximal matching in the graph, then the result is equal $|V|-|M|=n^2-m-|M|$. So, the algorithmic part of the solution reduces to finding the maximal matching.

Note, that the maximal degree of each vertex is 8. Therefore, $|E|\leq 8|B|, 8|W|$, where W is the set of white fields, and B is the set of black fields, and hence $|E|\leq 4|V|=O(|V|)$

3. Exemplary solution.
The solution uses the Hopcroft-Karp's algorithm. It is written in the file kni.pas.

Considering the big size of the graph, neither it nor the acyclic graph of extending paths can be represented in a straightforward way (at least not in DOS). However, we do not need to remember the set of edges E , since all the moves are allowed. It is enough to remember which fields have been removed and which are present.

The implementation of the auxiliary acyclic graph is more complicated. It can be seen, that we do not need to represent the set of its vertices explicitly. The starting vertices are unmatched white fields of the board, and the rest of them can be accessed through the edges. The lists of incident vertices are represented in a compressed way% –one bit per edge and one byte per vertex.
Since $|E|=O(|V|)$, complexity of this algorithm is $O((|E|+|V|)\sqrt{|V|}) = O(|V|^{2/3})$.

4. Other solutions.

We can find the maximal matching using the Edmonds-Karp's algorithm ([CLR], section 27.3) finding the maximal flow. In our case, this algorithm runs in O(|V|^2) time. Its simpler than the exemplary one, however it is significantly slower. Its written in kni1.pas file.

We can improve this algorithm. Instead of searching each time for the shortest extending path, we store the list of unmatched white fields and search for extending paths starting from each of these fields. There is no guarantee that we find the shortest path, but if we find a patch quick enough, and it is short, then we can extend the match doing only a few steps instead of Θ(|V|) . This solution is written in the file kni2.pas.

We can improve this algorithm even further, starting the next search from the field next to the one where the previous extending path started. Proceeding in such a way, if there were no extending paths starting from fields A1, A3, A5, A7, A9 and there was such a path starting from A11, then fields A1, A3, A5, A7 and A9 will be considered last. Such an improvement gives practically very good algorithm, that sometimes behaves better than the exemplary one. Moreover, such a solution can be devised even without the knowledge of Hopcroft-Karp's algorithm. This solution is in the file kni3.pas.

5. Wrong solutions.
One can approximate the solution as maximum of numbers of white and black fields. Such a solution is correct, for example, for chess-boards of size not grater than 3. Such a solution would score 0 points.

6. Tests.
Since the task is a hard one, and correct but slow solutions should score a significant amount of points, all the tests should be worth 10 points.

No	n	m	|V|	description
1	2	0	4	full board 2×2
2	2	1	3	2×2 board with one field missing
3	10	67	33	10×10 board, random fields missing
4	20	200	200	20×20 board, with regular cuts along two edges and some random fields missing,
5	28	276	508	as the previous one, 28×28
6	40	913	687	as the previous one, 40×40
7	80	4233	2167	as the previous one, 80×80
8	180	14632	17768	as the previous one, 180×180
9	200	4	39996	200×200 with four fields near the corner missing (in the shape of letter L)
10	200	10100	29900	200×200 random fields missing

7. Timeouts
Below are given execution times (for Pentium 166) and proposed timeouts. All the times are given in 1/100 s.

No	1	2	3	4	5	6	7	8	9	10

n	2	2	10	20	28	40	80	180	200	200
m	0	1	67	200	276	913	4233	14632	4	10100
$\|V\|=n^2-m$	4	3	33	200	508	687	2167	17768	39996	29900
KNI.PAS	ε	ε	ε	ε	ε	11	33	868	143	2225
KNI1.PAS	ε	ε	ε	ε	17	28	308	18405	66131	44105
KNI2.PAS	ε	ε	ε	ε	ε	11	110	13286	2109	18032
KNI3.PAS	ε	ε	ε	ε	ε	ε	ε	214	2098	1044
timeout	100	100	100	100	100	100	200	3500	8900	8900
points	10	10	10	10	10	10	10	10	10	10

8. Bibliography

[CLR] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest "Introduction to Algorithms".

Mars Map

mar.pas O(n\log c)) – Scan-line solution.
n – no. of rectangles c – maximum y-coordinate.

It passes all vertical lines in increasing order of their x -values and keeps a list of active y -intervals. At each "stop", i.e. for every vertical line, it does the following:

– multiply the x -distance to the previous stop with the size of the interval that is the union of all active y -intervals,
– add the result of this multiplication to the total result,
– update the set of active y -intervals.

The scan line is implemented as a full binary tree over range 0... 30000 (whole structure is hidden in array, like in standard heap implementation). Every node of the tree consists:
– number of edges which cover whole interval
– summary length of covered space in the interval defined by subtree of vertex

2. Other solutions

mar1.pas (O(nc)) – Scan-line solution with lazy implementation. The scan line is array over range 0... 30000 .

mar2.pas ($O(n^2)$) – Scan-line solution, (modification of previous solution with added y-axis compression).

mar3.pas ($O(n^3)$) – This solution first "compress" x and y-axis to used co-ordinates. Then we can think about plane as a bitmap and paint rectangles. This solution may not pass all test cases.

3. Tests

test0 – example from task description;
test1 – simple test, n=4;

test2 – simple test, n=7;
test3 – image with text: “BOI”;
test4 – random rectangles, n=100;
test5 – snail shape, made from rectangles, n=200;
test6 – random rectangles, n=1000;
test7 – large rectangles placed in X shape, n=2000;
test8 – many vertical and horizontal bars, n=5000;
test9 – two groups of squares, n=10000;
test10 – rhomb shape, n=10000;

Execution times (in seconds):

	mar.pas	mar1.pas	mar2.pas	mar3.pas
mar0.in	0.00	0.00	0.00	0.00
mar1.in	0.00	0.00	0.00	0.01
mar2.in	0.00	0.00	0.00	0.00
mar3.in	0.00	0.01	0.00	0.00
mar4.in	0.00	0.24	0.01	0.03
mar5.in	0.01	0.48	0.01	0.01
mar6.in	0.05	2.00	0.16	–
mar7.in	0.08	6.32	0.46	–
mar8.in	0.18	23.06	4.43	–
mar9.in	0.36	13.10	13.50	–
mar10.in	0.42	26.19	27.03	–

The images of tests can be found in a separate .pdf file.
In fact some of tests are much bigger, and images present scaled down tests.

6. Teleports,

1. Sample solution

tel.cpp

We will call Bornholm and Gotland A and B respectively. The first step of our algorithm is setting all the teleports on A in receiving mode, and all teleports on B in sending mode.

Why is this solution not good? The only thing that is wrong is that some receiving teleports on A are useless - there are no teleports sending to them. We will maintain this invariant - after each step the solution will be OK except that some receiving teleports on A will be useless.

In each step we choose a receiving teleport on A with no teleports sending to it. If there are no such teleports, we are finished. Let's call the teleport we have chosen x . We switch the teleport x to sending mode. Let y be the destination of x – y is a teleport on B . If y is in receiving mode, all is fine – the invariant is true. Otherwise we just switch y to receiving mode. We can make another receiving teleport on A useless this way, but the invariant is still true.

At the end there are no useless teleports left, so we have a correct solution.

At each step the number of receiving teleports on A decreases by 1 , so we will make at most m steps.

To be able to make each step in O(1) time, we introduce an array ***inDegree***, in which we store for each teleport on A the number of teleports sending to it. We also keep track of all useless receiving teleports on A – in the sample solution they are kept on a stack. In each step we just take a teleport from the stack, switch it to sending mode, possibly change the mode of its destination teleport and update one entry in the array ***inDegree***.

Reading the input and the initial step take O(m+n) time, therefore time complexity of the whole algorithm is O(m+n) . We also need O(m+n) space.

2. Other solutions.

One could just look for a useless teleport in a straightforward way: without the *inDegree* array, O(m+n) time for each step. This way we get an O(m(m+n)) solution – this solution is implemented in tel2.cpp.

3. Other files
telinv.cpp – input file format verifier;
telchk.cpp – solution checker ;
telgen.cpp – tests generator;

4. Tests

All tests were generated automatically. Most are random. Some are two lists, which can be one of the worst cases for some quadratic algorithms and for heuristics, because there is a unique answer and the whole list has to followed to determine the answer for a teleport.

test1 (ε sec.) m=n=10;
test2 (ε sec.) m=100 , n=107;
test3 (ε sec.) m=1000 , n=1005;
test4 (ε sec.) m=n=1000 , two lists
test5 (ε sec.) m=10006 , n=10000
test6 (0.1 sec.) m=n=10000 , two lists
test7 (0.2 sec.) m=50000 , n=1000
test8 (0.2 sec.) m=1000 , n=50000
test9 (0.4 sec.) m=50000 , n=49987
test10 (0.4 sec.) m=n=50000 , two lists tests

There can be more than one possible output, therefore a solution checker telchk.cpp is included.