CROATIAN OLYMPIAD IN INFORMATICS 2011.

ALGORITHM DESCRIPTIONS

COI 2011.	**Task SORT**

Most important observation is that there exists solution in which at most two cyclic swaps are needed. We can devide solution into few separate cases:

1. *Starting array is sorted* – in this case there is no need for any swap
2. *One swap is enough* – If there is number **a** at the position **P** in starting array and in position **a** there is number **b,** at position **b** number **c** ,..., at position **z** there is number **P**, then this configuration is called **interesting.** As example, starting array [5, 4, 1, 3, 2] is one big interesting configuration. In this case only one swap is necessary.
3. *Two swaps are enough* – it is easy to show that starting array can be partitioned into final number of interesting configurations (or subsets). If we make cyclic swap with one number from every interesting subset they all will colide into one big interesting configuration for which is then only one swap needed.

Necessary knowledge:

-

Category

Ad hoc

One possible solution of this problem consists of dividing an array which represents popularity of each seconds into several blocks. Number of seconds in a day is 86400. Let A be selected size of the blocks. Intervals which represent periods when someone has watched TV should be processed one by one and popularity array should be updated according to the interval data. When we are processing one interval two cases arise:
a) interval wraps around midnight - we split this interval on one that ends at midnight and the other that starts then
b) interval does not wrap around midnight - brute force update is optimized in the following manner: blocks that are fully contained in the processed interval are updated in one operation. Special case are the 'edge' blocks - those blocks are not fully contained, but overlap with the given interval. We solve this issue by updating each of the overlapping elements in the block one by one.

Queries are made in the same fashion as updates described above.

Speed of our algorithm depends on the parameter A. In this task, A was set to 100.

Sidenote: There is a faster approach with linear time complexity. Prefix sums are used to keep track of popularity of each second.

Necessary knowledge:
Buckets, prefix sums

Category
Data structures, ad hoc

It is clear that path with least number of returning is needed to be found. If in final path there exists returning from some village A to village B, then that interval is visited three times (once more moving from B to A). While without returning, that interval would be visited only once. So, intervals of returning are visited two more times and final solution will be:

M + 2 * complete_length_of_returning_intervals.

Goal is to find as short intervals of returning as possible. Persons that are traveling from left to right can be discarded: they will be transported in any case. Persons that are traveling from right to left are problematic: every one of them defines interval that is needed to be returning by (but maybe just as part of other longer interval).

Now, following solution is natural: we'll find **union** of every intervals on which we have to return (defined by persons that goes from right to left) and that union will be complete length of intervals we are returning by.

Final goal is to find union of given intervals very fast. This can be achieved using „sweep line" algorithm: moving from left to right we register events „start of interval" and „end of interval". Complexity is O(**N** lg **N**) for sorting.

Necessary knowledge:

Mathematical analysis of problem, sweep-line

Category

Sweep

COI 2011.	**Task KAMION**

Solution of this hard problem is going to be represented on simple version of this problem. In simple version there won't exist any roads of type 3, and the truck have to get empty to the city **N** in last step. In other words, it is allowed for truck to pass through city **N** on its trip as many times as it likes, but in the last step when it enters city **N** it has to be empty.

Rest of this description is going to be devoted to problem with these two restrictions, while the solution of original problem will be left to reader for exercise.

So, new problem which we are about to solve says: how to get from city **1** to city **N** using atmost **K** steps with restriction that truck has to get into town **N** empty. As solution dynamic programming is used. Lets mark number of different ways to get from city **a** to city **b** using exactly **k** steps starting (from city **a**) and ending (in city **b**) with empty truck as *dp[a][b][k].* First road which we have to pass in out trip will have to be road type 1 in which we load some kind of cargo. Lets mark this road as **a -> x**. Then we choose road on which we unload exactly that piece of cargo (which has to be road of type 2). Lets mark this road as **y -> z**. Then we can add to the solution all the paths, ie. *dp[x][y][k1] * dp[z][b][k2]* where *k1+k2+2 = **k**.* Unfortunately, solution implemented this way will have complexity O($\mathbf{E}^2\mathbf{K}^2$) which is not enough to get all points for this task.

Better solution is to use additional dynamic function *dp2[a][b][k]* which will contain number of different trips from city **a** and **b** using exactly **k** steps while starting and ending with empty truck (and never empty in the middle of trip).

Dynamic tables are calculated increasingly by number **k**. While calculating table dp we are finding first town and step in which our truck is going to be empty. If it happens in town **c** in step **k1**, we have::

*dp[a][b][k] += dp2[a][c][k1] * dp[c][b][k-k1]*

Parallel with that we are calcuating dynamic table dp2. Lets search for roads **a -> x** and **y -> z** where we are loading and unloading the same type of cargo. We have:

dp2[a][b][k] += dp[x][y][k-2]

Complexity of this solution can be represented as O($\mathbf{N}^3\mathbf{K}^2$) + O($\mathbf{E}^2$), which is enough to get all points for this task.

Necessary knowledge:
Advanced dynamic programming

Category
Graphs, dynamic programming

COI 2011.	Task LOVCI

Lets observe the board:

```
xxooxx
xxxxxx
xxxxxx
xxxxxx
xxxxxx
xxxxxx
```

Lets separate black and white cells (one bishop allways jumps by white cells, and the other by black cells):

```
x o x       x o x
x x x         x x x
 x x x       x x x
x x x         x x x
 x x x       x x x
x x x         x x x
```

Lets rotate the board by 45 degrees:

```
  xx            x
 xxxo          xxo
xxxxxx        xxxxx
 xxxx         xxxxx
  xx           xxx
                x
```

We can sort rows by length, by which the same set of cells will still be visible from every cell:

```
xxxxxx        xxxxx
 xxxx         xxxxx
 xxxo          xxx
  xx           xxo
  xx            x
                x
```

Same is applied to columns:

```
xxxxxx        xxxxx
xxxx          xxxxx
xxxo          xxx
xx            xxo
xx            x
              x
```

Now, we have to solve upper left sorted board. I.e. for every K, what is optimal solution if bishops make K steps on the board?

Lets take a look of R rows and C columns in which bishop will show at least once.
It is certain that first row from R has to cut with last column from C (in other ways last column isn't intersecting with anyone, so it can't be visited).
It is certain that last row from R has to cut with first column from C (in other ways last row

isn't intersecting with anyone, so it can't be visited).

It those assumptions are true then first row from R is intersecting with every column from S. Than it is possible to make these jumps (which will jumps into every row and every column exactly once)::
1.) Jump from starting row to first for in R.
2.) Jump to every column from S in any order, but in the end jump to first column in S (first row from R can see every column from S)
3.) Jump to every other row from R (first row from S can see every row from R)

It can be seen that bishop has to jump to same column in step 2 twice (if starting column is first one from S). So, if starting column is first column from S, then bishop won't jump in not even one other column and second step can be thrown out.

Further:
For k == 0 it is necessary to outtake starting cell of bishop because it doesn't count.
For k >= 2 it is possible to decide to jump only by allready seen rows/columns, so it is necessary to do dp[k] = max(dp[k-1], dp[k]).

Implementation:

1. Choose subset of columns that contains starting column of bishop in every way. For every row calculate how many points is there left.
2. Choose best row from which bishop can jump to every column in subset.
3. Choose starting row.
4. From now on, we add one by one best row in which bishop can jump from first column from subset.

For 100% points it is necessary to conclude that it is never good to jump in any corner, so step in which every subsets are choosen can evaluate 4 times faster. Without this optimization, 90% points can be achieved.

Necessary knowledge:

Dynamic programming

Category

Ad hoc