$D(x-i,y) \geq i$ equals the sum of $A(x')$ from $x' = x-s$ to $x' = x-1$. Doing updates to the $A$ array while being able to answer partial sum queries like that is exactly what a Fenwick tree does, in $O(\log n)$ time per update and per query. This results in an $O(n \log n)$ time per row or $O(n^2 \log n)$ time in total for the entire grid.

This problem had rather tight time limits, so one did need to be careful about constant factors in the implementation, and to make sure not to waste too much time on reading the input. The reason for this was not that we really wanted teams to make the best implementation, but simply that a reasonably optimized $\Omega(n^3)$ solution was rather fast for the grid size used (which already results in rather large input files, so we were hesitant to increase it even further), and we did want the $\Omega(n^3)$ solution to time out (the $n^3$ solutions we had were about 80% slower than the time limit and we wanted a decent margin to account for teams being better at low-level optimizations than us). Ultimately, we did succeed in preventing $n^3$ time solutions, but we also timed out several $n^2 \log n$ solutions forcing teams to optimize their constants a bit too much.

## Problem J: Uncrossed Knight's Tour

*Shortest judge solution: "1105" bytes. Shortest team solution (during contest): N/A bytes.*

This was the hardest problem in this set, and it was actually an open problem in mathematics; with people not knowing the optimal tour lengths for some of the chessboard sizes for which we posed this problem.

The first observation is that if the limits for $n$ were smaller (like, 50), we could attack this with a DP. The reason is that in order to extend a partial tour that is constructed up to some row, we don't need the full information on what happened previous to that row. The full state required to extend the tour, for a given row, is the following state:

- a representation of squares in this row: we either visited this square, both entering from and leaving to a square above this row, or visited entering from a square above this row, but not leaving above this row, or not yet visited this square
- a representation of moves from the row above to the row below (jumps of vertical length 2): we can have one of the two possible jumps (vertical and to the left, or vertical and to the right), or no jump at all, between any two cells.
- the connectivity information: we have to remember how the squares entered from above without leaving and the vertical jumps crossing our row are connected into pairs by the parts of the tour above our row.

In theory, this representation means that we hold one of three possible values for each of the 8 squares in a row, and each of the three possible values for each of the 7 spaces between squares, and additionally we have to connect up to 14 elements into pairs (which can be done in $13!! = 135135$ different ways), amounting to over $10^{12}$ states. However, in practice, most of these states are unreachable, and a program starting from the empty row will reach several hundred thousand states.

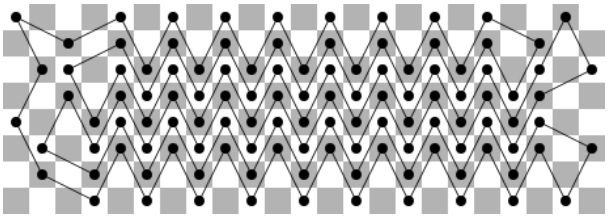
Representing the connectivity information in this state is a bit tricky, because we want to canonicalize it (that is, we want to make sure that we have only one valid representation for a given connectivity).

For any such state, we can then calculate what are the possible states in the next row, and how many visited squares do we add in such a transition. This can be done by a recursive

algorithm, where for each line between squares we just add a visited square in the next row, for each visited square with one incoming edge we branch out into the four options for the other edge from this square, and for each empty square we branch out into seven options – either it is really empty, or we visit it with both edges going downwards. We need to do this carefully to make sure the edges we add do not intersect, and that we maintain the connectivity information correctly.

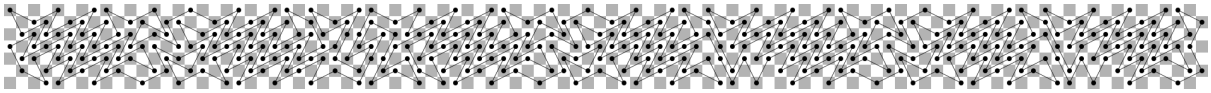
There are many technical details in the implementation of this algorithm (for instance, one has to distinguish the "empty row" state of "we have not started yet, there were no visited squares" from "we have already finished the tour", in order not to allow multiple disconnected tours as a solution). With an efficient implementation, an 8 by 50 chessboard can be solved in several seconds, and an unoptimized program will perhaps run for several minutes.

Now, the crucial observation is that the solutions actually start getting repetitive after a while. For instance, for board size 8 by $n$, after a while you get a pattern of 6 knights in a row all jumping left, and then all jumping right, adding 12 visited squares with each two added rows, as in the picture below.



This suggests that we should try to run the DP described above for a reasonable range of $n$, and look for this cyclic behaviour – that adding some $k$ rows to $n$ increases the answer by some $a$. And indeed – after running up to, say, $n \leq 200$, we will discover cycles for all possible values of $m$. Thus we can precompute these values locally and then submit a simple solution which has them hard-coded.

We found it interesting that for $m = 7$ one finds a somewhat surprising cycle length of 33 – there is a pattern of size $7 \times 33$ that can be repeated over and over in the solution. See, for instance, this diagram:



## Problem K: Wireless is the New Fiber

*Shortest judge solution: 709 bytes. Shortest team solution (during contest): 851 bytes.*

This is one of the easier problems in this problem set. After reading through the statement, we can see the problem is about constructing a tree where as many vertices as possibe have a given degree (note that the only information we care about from the input are the degrees of vertices in the input graph, and not the exact shape of the input graph).

The choice of which vertices preserve their degree can be made greedily. Since we have to construct a tree, the sum of degrees of all the vertices will be $2n - 2$, and each vertex will have degree at least 1. Thus, we are left with $n - 2$ spare degree increases to assign. In order to satisfy as many vertices as possible, we should assign these increments to the vertices with the