Editorial

It seems that I have failed to reduce the difficulty of the problems.

I hope that after reading the editorial you will see that they were not so hard after all.

You can already submit here: `http://domjudge.info.ucl.ac.be`.

Problem A

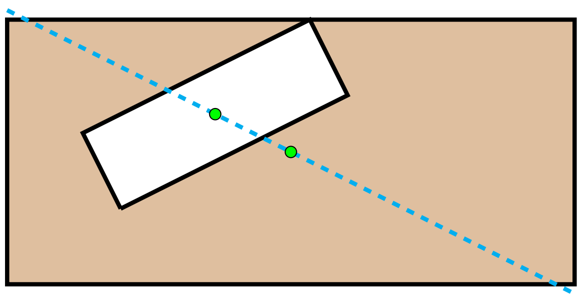
Clearly problem A was not hard. You can just implement what is said.

However, you can solve it even faster (in terms of coding time) and avoid to implement the rotations altogether. I will let you think about why this works but let $a[i]$ be the number of stones in the i-th column at the beginning.

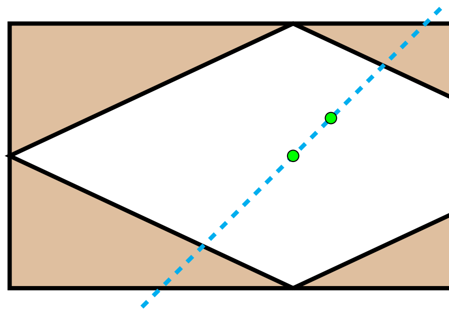
Making the rotations will actually sort the array a so you can just sort a and output the corresponding matrix.

Problem B

This problem is also trivial. Any line that passes in the center of a parallelogram will split it into two parts of equal area. Since we want to split the two parallelograms into two parts of equal area you can just output the two centers.



Just be careful that in some test cases the two centers are the same. In those cases just output one of the center and any other point.



Most of you tried to solve it with the cut polygon algorithm but you should always use all the information from a problem. If the shapes are parallelograms think about how to use that fact before starting to code blindly.

Problem C

For a given ride i denote by $s[i]$ the start time, $e[i]$ the end time $d[i]$ the ride duration. Also, denote by $dist(i, j)$ the distance between the end of ride i and the start of ride j.

First add a dummy ride at the start from $(0, 0)$ to $(0, 0)$ with time interval $[0, 0]$ and a dummy ride at the end from $(0, 0)$ to $(0, 0)$ with time interval $[\infty, \infty]$.

Compute for each ride the earliest ef time at which is can be finished:

$$ef[i] = \max(s[i], earliest[i - 1] + dist(i - 1, i) + d[i]$$

Compute for each ride the latest time ls at which we can start it:

$$ls[i] = \min(e[i], ls[i] - dist(i, i + 1)) - d[i]$$

Then loop over the rides to see if we can insert the new ride after it (except the last dummy one). You can add after ride i iff it does not make ride $i + 1$ start after $ls[i + 1]$. This can be computed in $O(1)$ by a simple formula.

Problem D

Add a dummy column with value 0 to the left and to the right.

This problem is quite tricky to get right as there are a lot of edge cases.

Lets assume that $s < e$ (the case $s > e$ is analogous to $s < e$ by reversing left and right). We will deal with $s = e$ after.

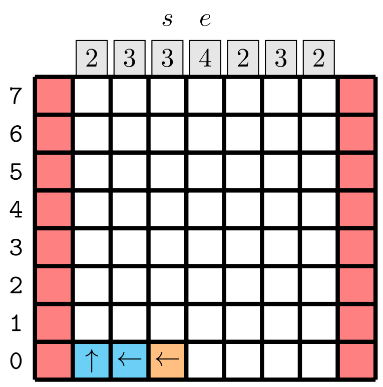
Since we want to go right, the idea is that we first go left to try to go up as much as possible. It is never a good idea to go back and forth.

So we go left until we reach a column i such that $a[i-1] \leq 1$. Then we go right and up until we reach column e. If we cannot reach column e then it is impossible.

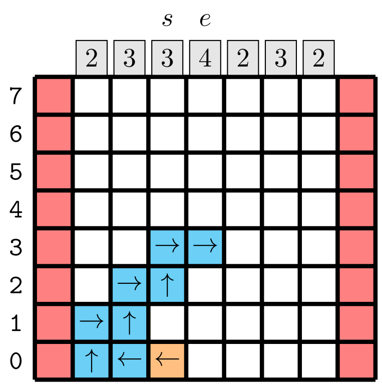
Otherwise, then we go right horizontally until we reach a column j such that $a[j+1] \leq 1$. We stop there because we can never come back if we cross such a column.

From there we go left and up until we reach e again. At this point we go up as much as possible on the end column. After that either we reached the top column and we have found a path or no path exists.

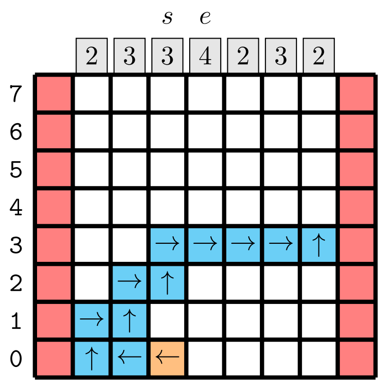
Note that we have already wasted all the steps to the left of column e by going right and up before.



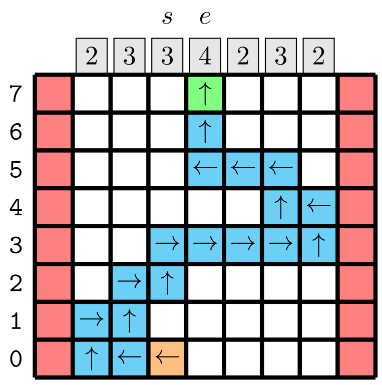
go left



go up-right



go right



go up-left then up at the goal

If $s = e$ then we can do the same unless $a[e] \leq 2$ If it is 0 or 1 then it is impossible. Otherwise we cannot cross the column tree times so we have to try both ways: 1. go left then up-right, 2. go right then up-left.

Problem E

Assume that $n = m$. The algorithm is the same, it just makes it easier to write the complexities.

Apparently you can solve this problem in $O(n^2)$. For more information ask Victor Lecomte. This solution uses the fact that the matrix is binary. The solution idea that I will present works even if the cells have different values assigned to them.

The intended solution was to reduce the problem to a maximum $2D$ sum. This problem consists of computing the maximum sum sub-matrix of a given matrix. In our case, we can set each tree location to $-\infty$ and the others at 1 and the maximum sum will give the maximum area land that you can get.

If you don't know how to solve the maximum $2D$ you can check it here: Max 2D sum

But our problem is more complicated: we want to select two sub-rectangles, not just one.

Since we require that they are disjoint we know that there exists a row or a column such that both of them lie on opposite sides. If we knew that row and column we could just compute the maximum $2D$ sum on each side and get the solution. But we don't know.

But that is ok, we can just try them all! For each row and column, compute the maximum $2D$ sum on each side, add them and output the maximum of all of these.

There is still a problem: there are $O(n)$ such rows and columns and since the maximum $2D$ sum algorithm is $O(n^3)$ the total complexity is $O(n^4)$.

However once you understand the maximum $2D$ sum algorithm is it not too hard to adapt it to keep track of the best solutions and update them in $O(n^2)$ while trying each line. Using this you will get an $O(n^3)$ algorithm.