Problem A

Let's first solve a simpler problem – when the sequence c is *cyclic*, i. e. when $c_i = c_{i \bmod N}$, for $i \geq 0$.
This simpler version is similar to *Fibonacci sequence*. Actually, for $N = 1$ and $c_0 = 1$, it is the Fibonacci sequence. To find the K^{th} number of these kind of recursive sequences fast we should first write them in their matrix form. Matrix form of this sequence is:

$$\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = \begin{pmatrix} c_{i-1} & c_{i-2} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix}$$

Expanding this, we can see that

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{K-1} C_{K-2} \dots C_2 C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{ where } C_i = \begin{pmatrix} c_i & c_{i-1} \\ 1 & 0 \end{pmatrix}.$$

How do we calculate this efficiently?

For relatively small K, and we will take $K < N$ for this case, we can do this just by multiplying all the matrices.
For large K ($K \geq N$), we will take advantage of the fact that c is *cyclic*. Since c is *cyclic* with cycle of length N, we know that $C_{N-1} C_{N-2} \dots C_1 C_0 = C_{iN+(N-1)} C_{iN+(N-2)} \dots C_{iN+1} C_{iN}$, for $i \geq 0$ (note that $C_0 = \begin{pmatrix} c_0 & c_{N-1} \\ 1 & 0 \end{pmatrix}$).
Let's define this product of matrices as $S = C_{N-1} C_{N-2} \dots C_1 C_0$.

Now, we can write a formula for F_K that can be calculated quickly:

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{a-1} C_{a-2} \dots C_1 C_0 S^b C_{N-1} C_{N-2} \dots C_2 C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{ where } b = (K - N) \, div \, N \text{ and } a = K \bmod N.$$

We can calculate S^b in $O(\log b)$ steps using ***exponentiaton* by squaring**, and then we can just multiply everything in the expression to get F_K quickly.

Let's get back to the full problem, when c is *almost cyclic*. In this case, we cannot just use S^b in the formula above, because some matrices in S^b may not respect the *cyclic* property. Instead of S^b, we will have something like

$$S \cdot S \cdot \dots \cdot S \cdot E_1 \cdot S \cdot S \cdot \dots \cdot S \cdot E_2 \cdot \dots = S^{t_1} \cdot E_1 \cdot S^{t_2} \cdot E_2 \cdot S^{t_3} \cdot \dots$$

where E_i denotes the product of matrices of the cycle, with some matrices being different than the matrices of the original cycle. Also, $i \leq 2M$ since

each of the M values of c different than values of the original cycle appears in exactly two matrices, so at most $2M$ of cycles are affected.
We can still calculate each S^{t_i} quickly, using exponentiation by squaring. Since there are at most $2M$ of those, total complexity of this would be $O(MlogK)$.

Now we only need to calculate each E_i. Naive way would be to just multiply all matrices of E_i. Since the number of matrices is N, this would be $O(NM)$ worst case, which is too slow. To quickly calculate E_i, we will initially create a **segment tree** of matrices, with matrices of original cycle in the leaves. Internal nodes of the tree will represent the product of their children. This means that the root will represent the product of all matrices in the cycle. To calculate E_i, we will just update our segment tree with those values that are different than the original values of the cycle. We will do this by updating corresponding leaves of the tree, moving up to the root and updating the products in the internal nodes. After we're done updating the tree with all of the matrices that are different than matrices of the original cycle, we will just use the product in the root of the tree. Finally, we will update the tree back with matrices of the original cycle in order to reuse the segment tree for E_{i+1}.

Since there are $O(N)$ nodes in the segment tree, the complexity of updating is $O(logN)$. The total complexity is then $O(MlogN + MlogK)$. We should also mention that the constant factor is not very small, since we operate on matrices and not just integers.
Note that we need to find $F_K \bmod P$ and P may not even be a prime number. However, this does not affect us since we only deal with operations of addition and multiplication throughout the whole procedure and we can just do them all modulo P.

Problem B

Let us first provide a suitable interpretation of the task description. The country can obviously be represented as an undirected graph, where vertices are towns and edges are roads. We see that it is connected (the description mentions that every city is reachable), but we also know that there are $N - 1$ edges, where N is the number of vertices. From this it follows that the graph is, in fact, a tree. For the sake of simplicity, let us make this tree rooted at node 1.

Let us consider just an $a \rightsquigarrow b$ transfer. From the previous assertion it follows that the cheapest path from a to b will always be the shortest path

from a to b – which is, in fact, the only path from a to b that does not have any repeated vertices. Borna's trip is thus uniquely defined by all of his stops. Getting from town a to town b requires that Borna first goes to the lowest common ancestor (LCA) node of a and b, and then descends to b (note that the LCA can also be any of the nodes a and b!). Computing the LCA of two vertices is a well-known problem and may be solved in several different ways. One possible approach is to use the equivalence between LCA and the range minimum query (RMQ) problem and then compute the LCA of any two vertices in constant time. Another one is based on heavy path decomposition. In any case, we need to be able to compute the LCA in $O(1)$ time.

Let us now define the notion of a **banned (directed) edge**: a directed edge $a \to b$ is *banned* if it requires paying a bribe. If a is the parent of b for a banned edge $a \to b$, then we call $a \to b$ a **down-banned edge**. Similarly, we may define **up-banned edges**. If Borna traveled along a banned edge p times, then he will have to prepare $1 + 2 + \cdots + 2^{p-1} = 2^p - 1$ thousands of dinars for bribing the police. Hence we need to determine the number of times every edge was traversed. This depends on whether the edge is down-banned or up-banned.

Before delving into these two cases, we need to compute the following three properties for every town x:

- $ends\_down$: the number of times x was the final stop in a path, this is equal to the number of occurrences of x in the array of stops;
- $ends\_up$: the number of times x was the highest stop in a path, this is equal to the number of times x was the LCA of two consecutive stops;
- $gone\_up$: the number of times x was the first stop in a path.

Now we consider the cases:

- If an edge $a \to b$ is up-banned, then the number of times it was traversed is equal to the number of times any vertex in a's subtree was an initial stop, minus the number of times any vertex in a's subtree was the highest stop (i.e. sum of all $gone\_up$'s minus the sum of all $ends\_up$'s). We may compute these parameters for all vertices at once using just one post-order tree traversal. Thus we can compute the 'bribe contributions' of all up-banned edge in linear time.
- If an edge $a \to b$ is down-banned, then the number of times it was traversed is equal to the number of times any vertex in b's subtree was a final stop, minus the number of times any vertex in b's subtree was the highest stop (i.e. sum of all $ends\_down$'s minus the sum of all

$ends\_up$'s). Similar to the previous case, we can compute the 'bribe contributions' of all down-banned edges using only one post-order tree traversal.

Hence, by first computing $ends\_down$, $ends\_up$ and $gone\_up$ for every vertex, and then traversing the tree, we are able to compute the answer. The final complexity depends on the implementation of LCA. The asymptotically optimal solution to this problem has $O(N + K)$ time complexity, but even an $O(N \log N + K)$ approach is acceptable given these constraints.

Problem C

The most naïve solution would be going through C(N, N/2) combinations of assignments half of people to Friday and another half to Friday and every time run Hungarian algorithm and just output the best answer among them. However time complexity of this solution would be O(C(N,N/2)*N^3) and O(N^2) memory – this won't pass time limit. To explain solution of this problem let's describe the scheme of Hungarian algorithm:

```
Hungarian(…)
{
        for (int i = 0; i < n; i++)
        {
                hungarian_iteration();
        }
}
```

Hungarian algorithm allows rows to be added one by one, in $O(n^2)$ each. So in this problem the only thing you had to do is to recursively go through all sets of binary masks with equal number of 0s and 1s (where 0 in i-th position means that i-th person would go partying on Friday, 1 – Saturday) and run Hungarian algorithm for this matrix. If done recursively complexity would be O(C(N+2,N/2+1) * N^2). You can see editorial for problem H to see why the number of vertices in that trie would be C(N+2, N/2+1).
Dp cannot be used here because of low ML.

Problem D

First notice that parity of thief's X coordinate changes every time he moves. Assume that at the beginning X coordinate of thief's position is odd, and check districts $(1, 1)$ and $(1, 2)$. The next day check districts $(2, 1)$ and $(2, 2)$ and so on until 1000^{th} when you check districts $(1000, 1)$ and $(1000, 2)$. What is achieved this way is that if starting parity was as assumed, thief could have never moved to district with X coordinate i on day $i + 1$, hence he couldn't have jumped over the search party and would've been caught. If he wasn't caught, his starting parity was different than we assumed, so on 1001^{st} day we search whatever (1 and 1001 are of the same parity, so we need to wait one day), and then starting on 1002^{nd} day we do the same sweep from $(1, 1)$ and $(1, 2)$ to $(1000, 1)$ and $(1000, 2)$ and guarantee to catch him.

Shortest possible solution is by going from $(2, 1)$ and $(2, 2)$ to $(1000, 1)$ and $(1000, 2)$ twice in a row, a total of 1998 days, which is correct in the same way. First sweep catches the thief if he started with even X coordinate, and second sweep catches the thief if he started with odd X coordinate.

Problem E

First we simplify the problem by replacing the initial set of points with the set of all points where some fan might appear after one second, call that set *S*. Every point in *S* has a probability that a specific player will appear there. Consequently, for every point **P** in *S* we know the expected number of fans at it after one second, call it **P**_e .

Now, for some arbitrary circle that passes through some three points of *S* (which doesn't violate the rules of the problem), the expected number of fans caught on camera is the sum of all **T**_e, where **T** is a point on or inside the circle. Our goal is to find a circle that maximizes that sum.

After drawing a few examples, we can notice that we can always catch most of the points that are possible locations of some fan or even all of them. We can write a brute-force solution that will increase our suspicion that all fans can be caught, no matter how they move.

Now let's try to find the largest circle of those that surely catch all fans and don't violate the rules in the problem. It is easy to see that three fixed points that determine the circle must lie on the convex hull of *S* (otherwise we surely wouldn't catch all points of *S* with that circle).

Convex hull can be computed in $|S|\log(|S|)$ which might be too slow if unnecessary points are not eliminated from S.

Notice that for every fan in input, if his speed is v, he might appear at $O(v^2)$ points, so convex hull algorithm would have $O(Nv^2\log(Nv))$ complexity, which is too slow.

The trick is to take only convex hull of those $O(v^2)$ points, which will have $O(1)$ points. All other points should be eliminated from S as they don't have a chance of appearing on convex hull of S. Contestants need to be careful with edge cases when a fan potentially goes out of the field.

After computing the convex hull of S (call it $H(S)$), we hope to find the circle that will pass through some three points on that hull and contain all other points inside it or on it.

These two claims can be proven geometrically:

1) For a convex polygon, the largest circle among all circumcircles of triangles determined by the polygon vertices will surely contain all vertices of the polygon on it or inside it.
2) For a convex polygon, the largest circumcircle of some triangle that is determined by vertices of the polygon is a circumcircle of a triangle that contains three consecutive vertices of a polygon.

With 1) and 2) we conclude:

The largest circle among those that are circumscribed around triangles that are composed of three consecutive vertices of $H(S)$ contains all of the points of $H(S)$ (and then obviously of S) and no other circle that contains all those points can be larger.

This means that we can finish the problem easily in linear time (with respect to the size of convex hull).

Problem F

First, let's consider solution in $O(n * maxX)$ complexity. It's a simple dynamic programming approach.

We have an array dyn (size $maxX$), which should say how costly would it be to be in this position at the end of the turn. At initialization step each position is set to $maxValue$ (some big number), except the initial position which is set to be 0. This should represent that in the initial moment it's impossible to be anywhere except in the initial position.

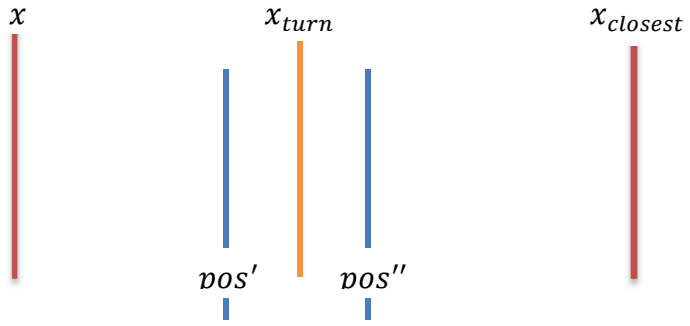
Before each turn we can change position, so before each turn we could calculate the best cost of being in that position when the bulbs light

up. We can do this by passing this array twice – once from the left and once from the right. Consider the case we're passing from left to right (smaller index i to larger index i, the other case could be explained in the similar fashion). While passing, we can keep the $bestVal$, which would be initiated to $dyn[0]$, and updated as $bestVal = min(bestVal + 1, dyn[i])$, $dyn[i] = bestVal$ for each $i > 0$. Rationale: if we came from the $i - 1$ position we have the same cost of that position $+1$, if we didn't we have the same cost we had before this exercise. Now we only need to add distance to the closest bulb for each position and we finished this turn. When we finish each turn we pick the lowest value in the array, and that's our solution. Simple enough.

But this solution is too slow for us. We want more.

Statement: We never have to move to the position which is not beginning or the end position of one of the light-ups.

Let's consider following situation: We're at the position x. If x is going to be inside of the next turn shining bulbs, there's no point of moving at all (it would be the same as moving after the turn). So, consider x is outside those bulbs, and $x_{closest}$ is the closest bulb to x that will be lighten-up. Also, consider $x < x_{closest}$ (the other case could be explained in the similar fashion). Consider all the remaining light-up points (light-up beginning and end positions) are in the array pos, which is sorted. Take a look at the following picture:



First thing we could notice is the fact that our cost for this turn is going to be $x_{closest} - x$, if we finish the turn anywhere between x and $x_{closest}$ inclusive. Going left from x or right from $x_{closest}$ doesn't make any sense, because we would have same or bigger cost than staying in x or $x_{closest}$ and moving from it in the next turn.

Next, let's consider we haven't ended our turn on some light-up endpoint, but between two neighboring endpoints, pos' and pos''. Let's call that position x_{turn}. Let's also introduce $x_{closest2}$, which is the closest bulb from the x_{turn} in the next turn light-up.

If x_{turn} is shining, then pos' is shining as well, so we could have finished our turn there. If $x_{closer2} \leq pos'$ we would be better off or equally well if we finished our turn in pos'. In that case we would have $x_{turn} - pos'$ smaller cost for the next turn. If afterwards we need to go to x_{turn}, total cost would not exceed the cost of going straight to x_{turn} in the initial turn. If $x_{closer2} \geq pos''$ we would be better off or equally well if we finished our turn in pos'', similar to the explanation for pos'. So, in each turn we could stay in the place or go to the closest light-up endpoint and we could still get the optimal solution.

We can use this fact to make a $O(n^2)$ solution – instead of each position we should take consider only light-up endpoints and initial position. Everything else is the same as in original solution.

Dynamic programming solution is enough to pass within the constraints for the program, but this problem can be solved in linear time as well.

Let's look at the values of array dyn. We can notice that this array actually has only one local minimum at each turn. What this means is that we have a range $[l, r]$ and that all of the values from $dyn[0]$ to $dyn[l]$ are monotonically decreasing, all values from $dyn[r]$ to $dyn[10^9]$ are monotonically increasing, while all of the values $dyn[l], dyn[l + 1], \dots, dyn[r]$ have the same value and represent the minimum summed cost until this turn. We can use this property to create a linear time algorithm.

Our linear algorithm will be as follows:

At the beginning, our optimal range will be $[x_{start}, x_{start}]$ and minimum cost will be 0. At each turn, we will update this optimal range and minimum cost.

If the range of shining bulbs in the next turn intersects with our optimal range, we can easily see that our new optimal range will be this intersection. The minimum cost will stay the same as cost for previous turn, since we don't need to move if we are located somewhere in this intersection, as we will already be located at a bulb that is shining. Anywhere outside of this intersection, the cost would increase since either the distance to the closest shining bulb would be larger than 0, or because of moving from our optimal range to somewhere outside of it, or both.

If the range of shining bulbs in the next turn does not intersect our optimal range and is left from our it, we will set that our optimal range is from the rightmost shining bulb to the left end of our previously optimal range. Our minimum cost will increase for exactly the distance between these positions – if we don't move from the left end of our previously optimal range, our cost increases for this distance. If we move from the left end of our

previously optimal range by one position to left, we decrease our distance in the next turn by 1, but increase the cost by 1 because of the move. Same goes for moving two positions to left, and so on until movement to the rightmost shining bulb in the next turn (at which point our distance to shining bulb will be 0, but our cost for moving will be the same as distance when we didn't move at all). It is easily seen that the minimum cost for a position that is left from this new optimal range is larger by 1, and the minimum cost for a position that is right from this new optimal range is also larger by 1. Moving further to the left or right, this cost increases more and more, resulting in only one local minimum as we described previously.
If the range of shining bulbs does not intersect our optimal range and is right from it, we can do a similar thing as we do when the range is left from our optimal range.

After all the turns, we have our optimal range and the minimum cost possible. The total complexity is $O(n)$, since in each turn we update the range in $O(1)$.

Problem G

The problem can be restated as follows: given an undirected graph. Consider any path with edge lengths $l_0, l_1, \ldots, l_{len-1}$. It's cost is $l_0 + 10l_1 + \ldots + 10^{len-1}\{l_{len-1}\}$. We need to find the cost of the cheapest path from 0 to $n-1$ and output it.

For now let's ignore the leading zeros. Notice that the distance after walking the edge (u, v) can be obtained by putting the length of this edge in front of the distance to get to u . So the trivial solution is to run BFS from vertex 0 and store for each vertex the smallest number to get to it. But it will run in $O(N^2)$ time because we will have to compare large number in order to get the best one for each vertex.

In order to avoid it we can store the equivalence classes instead of actual numbers, so that we could compare their classes, not actual numbers. The vertex 0 will have class 0. We will split the graph into layers.
The k^{th} layer will have vertexes with length of distance exactly k. Now process the graph layer by layer. For k^{th} layer we know all equivalence classes, let's obtain the $\{k+1\}^{st}$ layer and all the equivalence classes for it. Imagine we walked some edge (u, v) with length l. The distance to v d(v) will be $\overline{ld(u)}$. If we need to compare two numbers with the same

length, the equivalence classes allow us to replace it with just a pair (l,□class(u)). Store the minimal pair for each vertex of the next layer, sort and shrink them obtaining the equivalence class for new layer. This is easily done in O(nlog(n)), but also a O(n·alphabet) solution exists.

So how do we handle zeros? They behave almost the same way as other digits with only one exception: when they are in front of distance (leading zeros). For example, the path with length 001 is smaller than 11 despite being longer. The problems in the middle of the path don't cause any problems to solution described above. To solve this problem, let's process them in different manner: calculate for each vertex the smallest number of zeros to get to it from n□-□1. Now the answer can be obtained as follows: walk from 0 to vertex x by the shortest path and go from x to n□-□1 only by zero edges .

Problem H

Problem naturally can be transformed to a more formal way: how many vertices will a trie contain if we add all possible strings with length 2*N with equal number of zeros and ones to it.
So, first of all, it is obvious that upper half of this tree would be a full binary tree. Let's take a look on N=3: level 0 – 1 vertex, level 1 – 2 vertices, level 2 – 4 vertices, level 3 – 8 vertices.

Starting from N-th level not every vertex will duplicate: only those that haven't spent theirs 0s or 1s will.

So, here is how to calculate how many vertices will be there on level i+1:

- Lets' assign Number_of_duplicating_vertices_from_level to PD(i)
- $Count_{i+1} = PD(i)*2 + (Count_i - PD(i))$.
- And PD can be calculated pretty easily with binomial coefficients: PD(i) = 2*C(i, N).
- Everything else is implementation techniques: inverse module arithmetic's + some fast way to calculate these C(i, N) and sum Counts

Problem I

To solve this problem, we should first take a look at one easier problem. Consider having the same problem statement, but with rectangles instead of triangles. The solution to this problem is pretty straightforward. We will store a matrix representing points in our coordinate system. For each type 1 query, given the rectangle $((x_{ul}, y_{ul}), (x_{ur}, y_{ur}), (x_{ll}, y_{ll}), (x_{lr}, y_{lr}))$, we would add -1 to the points $(x_{ul}, y_{ul} + 1)$ and $(x_{lr} + 1,\ y_{lr})$, and 1 to the points (x_{ll}, y_{ll}) and $(x_{ur} + 1, y_{ur} + 1)$. Notice that we expanded the given rectangle when adding +1s and -1s, this is because the point is considered in the rectangle even when it is on the border! This way, when type 2 query is received, to find the answer we simply sum every value in the rectangle $(0, 0)$ to (x, y). Since simply summing the points in rectangles is $O(n^2)$, we should use binary indexed tree for this, hence getting the sufficient time complexity $O(\log^2 n)$ per query.

We will use this approach with some modification to solve the original problem. For handling the triangles we need to introduce new coordinate systems. Depending on the type of triangle (types 1 and 4 are analogous, so are types 2 and 3) we will make two new coordinate systems, where the corresponding hypotenuses are parallel to one axis. For types 2 and 3, point (x, y) in original coordinate system would map to $(x + y, y)$, and for types 1 and 4, point (x, y) would map to $(x + n - y - 1, y)$, where n is the size of coordinate plane given in the input.

The problem is now somewhat abstracted to the simple rectangle problem. This time we will need three matrices, one representing original coordinate system and two representing the introduced coordinate systems. For each triangle we need to border it with +1s and -1s, similarly as in the rectangle case, only using 2 matrices for each triangle. When adding border for the cathetus we use the original coordinate system and for the hypotenuse we need one of the two introduced coordinate systems, depending on the type of the triangle. Remember, the point belongs to the triangle even when it is on one of catheti or hypotenuse, so be careful.

For calculating the answer for type 2 query, we need to sum the values in the rectangle from $(0, 0)$ to (x, y) in all three matrices (of course, (x, y) needs to be mapped accordingly to the coordinate system each matrix represent). Again, to not exceed time limit we need to use binary indexed trees.

Since we are using binary indexed tree, the time complexity of this solution is $O(Q * \log^2 N)$.