Editorial

Tasks, test data and solutions were prepared by: Nikola Dmitrović, Dominik Fistrić, Bojan Štetić, Paula Vidas, Pavel Kliska, and Krešimir Nežmah. Implementation examples are given in the attached source code files.

Task Ljeto

Prepared by: Pavel Kliska
Necessary skills: for-loops, arrays/lists

Using an array $z[8]$, for each of the eight players we can keep track of the last time this player sprayed someone from the other team. Initially, we set each of these values to -11. Then, reading the input line by line, we add 100 points to the teams depending on if $a_i \leq 4$. To determine if a spray is a dobule-spray, it is sufficient to check if $t_i - z[a_i - 1] \leq 10$, and if so, we should add an additional 50 points to that team. Finally, for each line we should update the value $z[a_i - 1] = t_i$.

Task Kamenčići

Prepared by: Pavel Kliska and Krešimir Nežmah
Necessary skills: prefix sums, dynamic programming

Notice that the state of the game is completely determined by only three numbers: l - the leftmost pebble still in the game, r - the rightmost pebble still in the game and m - the number of red pebbles collected so far by the player whose turn it is. Using prefix sums on the number of red pebbles in the array, from these number we can easily determine which player will win. Since $n, k \leq 350$ and from each game state we can make only two possible moves, a natural idea is to use dynamic programming. The state will be $dp[l][r][m]$ which will be 1 if the current player can win, and 0 otherwise. The number of red pebbles collected so far by the other player can be determined from the information of the state: $h =$ (num. of red pebbles) $- m -$ (num. of red pebbles in interval l, r), where we used h to denote this number. The transition is defined as follows:

- $dp[l][r][m] = 0$, if $m \geq k$
- $dp[l][r][m] = 1$, if $h \geq k$
- $dp[l][r][m] = \quad !dp[l+1][r][h] \quad | \quad !dp[l][r-1][h]$, otherwise

Task Logičari

Prepared by: Krešimir Nežmah
Necessary skills: graphs, dynamic programming on trees

We will call nodes which will have blue-eyed logicians on them black nodes, and the other nodes will be called white nodes. Notice that every black node has exactly one black neighbour and that it is also that node's neighbour. Thus, the black nodes come in pairs as the endpoints of certaint edges.

For the first subtask, one should notice that the nodes in a cylce should alternate between two black nodes and two white nodes, so the solution exists only when n is divisible by 4 and the answer is then $n/2$.

The second subtask can be solved by trying out all possibilities for the black nodes with time complexity $O(2^n n)$.

For the remaining subtasks, one should notice that the graph contains exactly one cycle, where from each node of the cycle a tree might hang off rooted at that node. Using a DFS we can find that cycle and then remove one of the edges of the cycle. One of the ends we will call the *root*, and the other one will be the

special node. After removing the edge, the remaining graph is a tree, which we will root in the mentioned root node.

The task will be solved with dynamic programming on the obtained tree. We will fix one of the possibilities for the choice of color of the root and the special node (4 possibilites), and add this information to the state of our dp. The state will have the current node which determines the current subtree and an additional 4 flags: the state is $DP[x][me][up][rt][sp]$ which denotes the least number of black nodes in the subtree of node x if the color of node x is written in the flag me, the color of the parent of x is up, the color of the root is rt, and the color of the special node is sp. In the state we assume that the parent of x has already taken care of having a black neighbour, so that we are left to determine the colors of the nodes in the subtree of x.

The transition is as follows. First we determine the cases in which the flags in the state don't match up (so the answer is immediately `-1` for this state). This happens when:

- x is the root, and $me \neq rt$
- x is the special node, and $me \neq sp$
- x is the special node, and rt i up are black - then x is covered by two black neighbours.

Then we figure out if the current node x already has a black neighbour. This is true either when up is black, or if x is the root and sp is black, or if x is the special node and rt is black. If x happens to already be covered, none of his children are allowed to be black, so the solution is given by summing the expression $DP[v][white][me][rt][sp]$ over all children v of x. If x is not covered, we have to choose one of the children which will be black, and the rest of them will be white. We do this by calculating the same sum as above, except changing the me flag for one of the children to 'black'. We try this for each child and determine which one gives the least result.

Task Set

Prepared by: Krešimir Nežmah
Necessary skills: convolutions, FWHT and similar, mathematics

For the first subtask it is sufficient to try out every triplet of cards and check whether it forms a set in $O(n^3k)$.

For the second subtask, one should notice that if we choose two cards, the third card needed for a set is uniquely determined. Namely, for each position, if the corresponding characters in the first two cards are the same, then the third character has to be equal to them, and if they are different, the third character will be the remaining one. With time complexity $O(n^2k)$ we can now try out every pair of cards and check whether there exists a third card which will form a set with them. Of course, we should first record in an array of size 3^k for each type of card if it shows up in the input.

From now on, we will assume that the characters in question are 0, 1 and 2 instead of 1, 2 and 3, so that we can view each card as a number in base 3. Let's try to come up with a simple rule that associates a pair of characters with the third character needed for a set. In other words, we are looking for a rule that acts in the following way:

$$(0,0) \mapsto 0, \quad (1,1) \mapsto 1, \quad (2,2) \mapsto 2, \quad (0,1) \mapsto 2, \quad (0,2) \mapsto 1, \quad (1,2) \mapsto 0$$

We can notice that $(a,b) \mapsto -(a+b) \mod 3$, or equivalently, the characters a, b and c form a set if and only if $(a+b+c) \mod 3 = 0$. Now let's make an analogy with the bitwise xor operation. This operation is denoted by $\oplus$ and represents addition modulo 2 with the digits in base 2. In this problem we will consider addition modulo 3 with the digits in base 3, which we will denote by $*$. Having in mind the things mentioned above, three cards a, b and c form a set if and only if $a * b * c = 0$, where the cards are thought of as numbers in base 3.

Let's fix some card c and try to figure out how many pairs of cards a and b exist so that $a * b = -c$. For each individual c we can calculate this in $O(nk)$ so the total complexity is $O(n^2k)$, but we will show a way to find the answer for all cards c simultaneously in the complexity $O(3^k k)$. If instead of the operation $*$ we had the operation $+$, the problem could be solved by calculating the desired $+$-convolution with fast multiplication of polynomials using FFT. In this problem we will therefore try to modify this idea to calculate the $*$-convolution.

The operations $\oplus$ and $*$ are very similar and the $*$-convolution is calculated in the same manner as the xor-convolution. Thus, what follows is a description of the modification of the 'fast walsh-hadamard transformation' (FWHT) to work modulo 3. More about this can be found on this codeforces blog. (P.S. the day before the contest, another great blog on the topic appeared on codeforces: link)

At a high-level, the idea is the following:

- The given deck of cards is represented by a polynomial.
- Each term in the polynomial represents a certain type of card.
- The coefficients of the polynomial represent the number of times a card appears in the deck. In the beginning, all of the coefficients are either 0 or 1.
- We will square the polynomial to get new coefficients which represent the result of the desired convolution. (For now this corresponds to the + operation).
- Before multiplying, we will convert the polynomial from coefficient form to point value form as a sequence of calculated values $(x_i, P(x_i))$, which is more desirable for multiplication.
- The result of the multiplication should be converted back to coefficient form.

Regular multiplication of polynomials corresponds to the operation $+$, that is $(x^a, x^b) \mapsto x^{a+b}$. We would like to make a modification so that $(x^a, x^b) \mapsto x^{a*b}$. We need to make two modifications:

- 1) Addition should be done separately for each digit.
- 2) Addition should be done modulo 3.

Problem 1) can be solved by introducing a polynomial with k variables x_0, x_1, ..., x_{k-1}. For example, a pair of cards '1123' and '2321' (that is '0012' and '1210') is represented by a polynomial

$$P(x_3, x_2, x_1, x_0) = x_3^0 x_2^0 x_1^1 x_0^2 + x_3^1 x_2^2 x_1^1 x_0^0.$$

Multiplication now correspond to addition of the digits separately. Looking at each of the varibles separately, the polynomial is of degree at most 2, but when squaring the degree might grow larger.

To convert a polynomial (which remember has 3^k coefficients) to point-value form, we will calculate the value of the polynomial at 3^k different points. We will choose three values v_0, v_1 i v_2 and calculate $P(x_{k-1}, ..., x_0)$ for each possible combination where $x_i \in \{v_0, v_1, v_2\}$, of which there are also 3^k. In the implementation we will therefore have a transformation F that converts a sequence of 3^k coefficients to a sequence of 3^k calculated values.

Since the product of two polynomials has more coefficients than the original polynomials, if we wanted to calculated the true product, we would have had to extend the polynomials to bigger powers, making the new coefficients 0. However, to solve problem 2), that is precisely what we will not do. When doing the inverse transformation F^{-1} which returns the coefficient form, we will purposefully demand that the result has 3^k coefficients. Additionally, for the values v_0, v_1 i v_2 we will choose the third roots of unity (both real and complex), that is the numbers 1, $\frac{-1+\sqrt{3}i}{2}$ and $\frac{-1-\sqrt{3}i}{2}$, so that $v_0^3 = v_1^3 = v_2^3 = 1$. The effect of these two things is that the coefficients of larger powers in the resulting product (x^3, x^4, ...) will get

added to the coefficients of the smaller powers - precisely with the smallest power that is the same modulo 3 (because of the choice of v_i). Thus, the powers will reduce modulo 3 and we will get exactly the desired 3^k coefficients of the $*$-convolution.

Let us illustrate this on an example with $k = 2$. For the polynomial we take

$$\begin{aligned} P(x_1, x_0) = a_{00}x_1^0x_0^0 + a_{01}x_1^0x_0^1 + a_{02}x_1^0x_0^2 \\ +a_{10}x_1^1x_0^0 + a_{11}x_1^1x_0^1 + a_{12}x_1^1x_0^2 \\ +a_{20}x_1^2x_0^0 + a_{21}x_1^2x_0^1 + a_{22}x_1^2x_0^2, \end{aligned}$$

which can also be written as

$$P(x_1, x_0) = Q_0(x_0) \cdot x_1^0 + Q_1(x_0) \cdot x_1^1 + Q_2(x_0) \cdot x_1^2,$$

where

$$Q_i(x_0) = a_{i0}x_0^0 + a_{i1}x_0^1 + a_{i2}x_0^2.$$

First, we will apply the transformation F on each of the polynomials Q_i separately. Using the label $w = -\frac{1}{2} + \frac{\sqrt{3}}{2}i$, we have:

$$(a_{i0}, a_{i1}, a_{i2}) \mapsto (a_{i0} + a_{i1} + a_{i2}, \quad a_{i0} + wa_{i1} + w^2a_{i2}, \quad a_{i0} + w^2a_{i1} + wa_{i2})$$

In this way, the sequence of coefficients

$$(a_{00}, a_{01}, a_{02}, \quad a_{10}, a_{11}, a_{12}, \quad a_{20}, a_{21}, a_{22})$$

turns into a new list of coefficients, which we will label with

$$(a'_{00}, a'_{01}, a'_{02}, \quad a'_{10}, a'_{11}, a'_{12}, \quad a'_{20}, a'_{21}, a'_{22}).$$

We would like to obtain a list that has the following values in order

$$P(1,1), P(1,w), P(1,w^2), \quad P(w,1), P(w,w), P(w,w^2), \quad P(w^2,1), P(w^2,w), P(w^2,w^2).$$

What remains is to replace the triplets $(a'_{0i}, a'_{1i}, a'_{2i})$ with new values, in the same manner as we did when calculating the values for Q_i.

For bigger values of k, the process is analogous. In each of the k iterations, we gradually transform the coefficients in the described way, each time jumping by a larger power of 3.

The inverse transformation F^{-1} is almost identical to the original, having the formula

$$(a, b, c) \mapsto \frac{1}{3}(a + b + c, \quad a + w^2b + wc, \quad a + wb + w^2c).$$

It should be noted that in the implementation, there is no need to make calculations with complex numbers, whose real and imaginary parts are stored with a floating point type. Instead, we can notice that at each moment every number will be of the form $a + bw$, where a and b are whole numbers which fit in long long int. When calculating it is useful to keep in mind that $w^3 = 1$ and $w^2 = -1 - w$.

Task Volontiranje

Prepared by: Krešimir Nežmah
Necessary skills: longest increasing subsequence, greedy algorithms, amortized time complexity

We use 'LIS' as a shorthand for a longest increasing subsequence.

The first subtask can be solved by finding every LIS, and then find via dynamic programming with bitmasks a set of them which do not overlap.

For the rest of the subtasks, some more insight into the structure of the longest increasing subsequences is needed. The solution in short is the following: we can remove the subsequences one by one, each time greedily building the lexicographically smallest LIS. A naive implementation of that would be to slow, so it is necessary to do some backtracking and removing of certain elements for which we are sure that they will not help with the solution, so that the time complexity gets reduced.

A more detailed description and proofs of the observations are given, but first we will mention a couple of general properties of this type of configuration, which are common in these types of tasks.

For each index i ($1 \leq i \leq n$), let's define $LIS[i]$ as the length of the longest increasing subsequence ending at index i. This can be calculated with the formula $LIS[i] = 1 + \max\{LIS[j] : 1 \leq j < i, p_j < p_i\}$ using standard algorithms for finding a LIS. Furthermore, let l denote the length of a LIS, i.e. $l = \max_{1 \leq i \leq n} LIS[i]$. Also, for a fixed positive integer k let's define S_k to be the set of all indices i for which $LIS[i] = k$. Two important observations are:

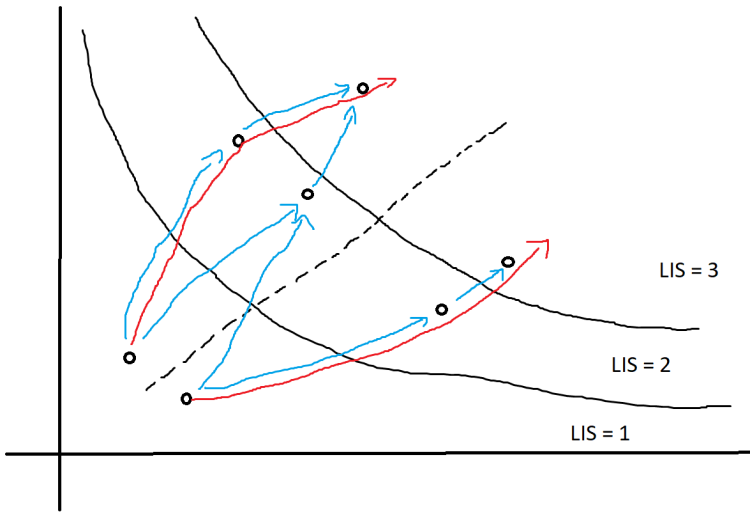
- *Claim*
  For all positive integers k, if we look at the values corresponding to the indices of S_k, they will be decreasing.

  Proof
  If there were $a, b \in S_k$ such that $a < b$ i $p_a < p_b$, the longest increasing subsequence ending at a could be extended to b and then $LIS[a] < LIS[b]$. □

- *Claim*
  For each LIS $x_1, x_2, ..., x_l$ (where $x_1 < x_2 < ... < x_l$) it holds that $LIS[x_i] = i$ for all $1 \leq i \leq l$. In other words, if we look at a fixed index j, it will always find itself at the same position in every LIS (precisely at position $LIS[j]$).

  Proof
  Just as in the previous claim, we conclude that $LIS[x_1] < ... < LIS[x_l]$. Noting that $LIS[x_1] = 1$ and $LIS[x_l] = l$, these l numbers must be exactly $1, 2, ..., l$. □

The things mentioned so far can be visualized in the following way (see image bellow): the given permutation can be interpreted as a set of points (i, p_i) in coordinate plane, and increasing subsequences can be thought of as path going through the points, moving 'up and to the right'. Having in mind the claims mentioned above, the sets S_k come in layers, and a LIS (red arrows) pass through one point in each layer.



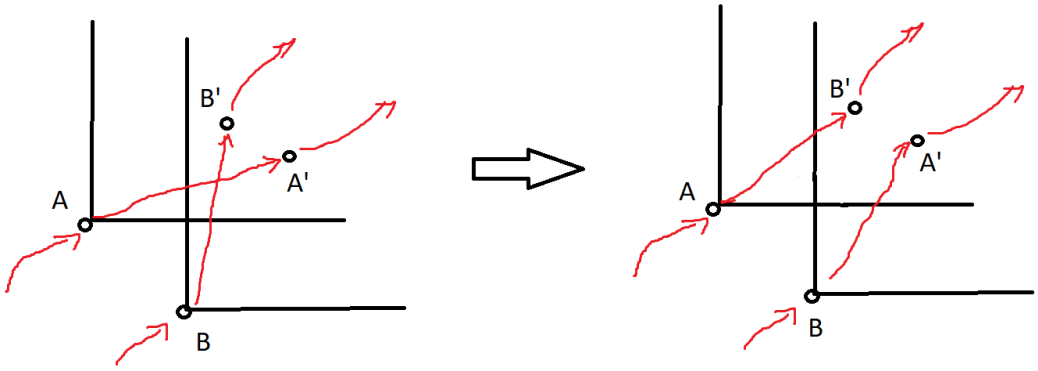
Between every two neighbouring layers blue edges are drawn between pairs of points where an increasing subsequence of the first point can be extended to the second point (those are actually pairs of points

where the second points is 'up and to the right' of the first point). Thus, a LIS is any path using the blue edges which starts in the first layer and ends in the last layer. Also, for each point, all of its neighbours form a segment of points in the next layer.

Now we will mention the claims specific to the problem.

Claim 1:
Assume that in the optimal solution the number of LIS's is equal to m, and that the indices of the i-th LIS are denoted by $x_{i,1}, x_{i,2}, ..., x_{i,l}$. Then, there exists an optimal solution such that for each j it holds that $x_{1,j} < x_{2,j} < ... < x_{m,j}$.
On the image this corresponds to the fact that it is possible to choose the red paths so that they do not intersect.

Proof:



Let's look at a situation where the paths intersect and let's label the points as in the image. Since the edges $\overline{AA'}$ and $\overline{BB'}$ intersect, the points A' and B' have to be in the intersection of the 'fields of vision' of the points A and B. Thus, instead of the current edges we can take the edges $\overline{AB'}$ and $\overline{BA'}$, untangling the paths. Using a sequence of such untanglings, we can make it so that the paths do not intersect. □

Since there exists an optimal solution in which the paths do not intersect, the idea comes to mind of trying to pick paths from left to right (it is possible to make a solution in the opposite direction) such that we take the paths that are 'as left as possible' so that we would have more room for the remaining paths. The following claim shows that the right choice is in fact the lexicographically smallest LIS, i.e. that it leaves the most room.

Claim 2:
Assume that from each layer we have removed a prefix of points, so that only the remaining points are allowed to be used when constructing a LIS. The lexicographically smallest LIS of the remaining points will then be the smallest for each layer separately.

Proof
Assume not. Denote the lexicographically smallest LIS by P. By the assumption there exists a path Q such that in the first layer it begins at the same point or later than P, but which finds itself earlier than P in some other layer. The first place where this happens is actually a crossing, which by claim 1, we can untangle, obtaining a lexicographically smaller path than P. □

A consequence of claim 2 is that each point that is to the left of the lexicographically smallest LIS (in any layer) can never be used for building a LIS. Thus, the solution can be obtained if at every step we choose the lexicographically smallest LIS and remove all of the points on or to the left of it. A solution that goes through all of the remaining points for each step and finds the desired LIS solves the second subtask.

To solve the third subtask, the searching process should be sped up. Say we're trying to build our current LIS and that we are currently at node v (the LIS is built in order from smaller to larger values, and we have so far built a prefix of the LIS). By the things mentioned, node v is the leftmost node in the current

layer that has the possibility of being extended to a LIS. All of the nodes in the next layer which are to the left of v can immediately be deleted, because if we can't reach them now, we won't be able to reach them in the future. Then, if from v we cannot reach the leftmost node in the next layer (i.e. if that node is bellow v), then no path from v can continue to the last layer. For that reason, we should remove node v, and the new current vertex will become v's predecessor, from which we repeat the process. Of course, if it is possible to go from v to the leftmost node in the next layer, the path will continue through it.

After we make the division into layers, which can be done with time complexity $O(n \log n)$, the mentioned searching procedure works in $O(n)$, which is enough for the third subtask.