XX Open Cup Grand Prix of Warsaw Editorial

Mateusz Radecki (Radewoosh)
Marek Sokolowski (mnbvmar)

September 15, 2019

A. Alakazam

Let $p_{t,v,i}$ be the probability that after t queries, $a[i] = v$. Consider a single random shuffle of $a[l..r]$ in $(t+1)$-st query. After the shuffle, v is equally probable to appear in each of $a[l], a[l+1], \cdots, a[r]$, and the probability of it appearing in any of the remaining cells doesn't change. Therefore, $p_{t+1,v,i} = \frac{1}{r-l+1} \sum_{j=l}^{r} p_{t,v,j}$ for $i \in [l, r]$, and $p_{t+1,v,i} = p_{t,v,i}$ for $i \notin [l, r]$.

However, the expected value of $a[i]$ after t queries, $E_{t,i} := \sum_{v \in \mathbb{N}} p_{t,v,i}$, is linear on each $p_{t,v,i}$. It means that we also have $E_{t+1,i} = \frac{1}{r-l+1} \sum_{j=l}^{r} E_{t,i}$ for $i \in [l, r]$.

Therefore, each shuffle is equivalent to finding the mean value M of all elements in some interval and then setting all values in this interval to M. This can be done either using a segment tree or by maintaining a sorted set of intervals with the same expected value. Either of these allows us to solve the problem in $O((n+q) \log n)$ time.

B. Bulbasaur

Let's solve a simpler version of the problem and calculate $F(1, n)$. Let's add layers one by one from the right to the left and for each subset of the vertices in the current layer maintain an information if we can simultaneously push a unit of the flow from each vertex in this subset to the right end of the graph. Let's call subsets which meet this property *good subsets*.

How to add a new layer quickly? We can use the fact the good subsets form a matroid and use something what I'll call a *generalized Hall's theorem*. Let's pick some subset A of the vertices from the current layer. Let's denote by $nei(A)$ a set of vertices in the next layer to the right which are connected to at least one vertex in A. Let's say that A is *OK* if and only if there exists some set B such that $B \subseteq nei(A)$, $|B| = |A|$ and B is a good set. (Notice that this definition differs from Hall's definition in that B has to be a good set here.) It turns out that A is good if and only if each subset of A is OK.

How to check for some A if it's OK? It's easy to observe that if some set of vertices is good, then also all its subsets are good. Thus, for each set of vertices from the same layer we can calculate the size of its largest good subset. With these values we can easily check if $nei(A)$ contains any good subset which is big enough.

This solution works in $O(n \cdot 2^k \cdot k)$. The k factor comes from the fact that we need to calculate the sizes of the largest good subsets (we need to propagate these values to the supersets) and from the fact that we need to propagate the fact of not being an OK subset.

We can notice that this solution also calculates answers for each suffix of the input sequence (as $F(i, n)$ is the size of the largest good subset after adding the i-th layer from the left). How to calculate the sum of the answers for each interval? We can notice that for constant i, $F(i, j)$ can only

decrease when we increase j. So, for each subset A of the vertices from the i-th layer, let's calculate the largest j such that we can push flow of size $|A|$ from A to the vertices in the j-th layer. It's obvious that these values will be enough to compute the answer – for each value of F, we'd know how many intervals with fixed left end have exactly this value of F.

So, we want to calculate for each set A the greatest j such that A would be good if the graph ended at the layer j (call it $good\_range(A)$). Let's similarly define a new meaning of being OK – let $OK\_range(A)$ be the maximum j such that A is OK in the graph ending at the j-th layer. Clearly, the $good\_range(A)$ is equal to the minimum OK_range over all subsets of A.

So, how to calculate an $OK\_range(A)$? We want to know the maximum value of $good\_range$ over all B such that $B \subseteq nei(A)$ and $|B| = |A|$. Here, a $2D$-array would be useful, let's call it $G[2^k][k+1]$. $G[S][j]$ stores the greatest value of $good\_range$ for any subset of set S of size j. Such an array can be calculated in time $O(2^k \cdot k^2)$. Therefore, the overall time complexity of the solution is to $O(n \cdot 2^k \cdot k^2)$, and the space complexity is $O(2^k \cdot k^2)$.

There also exists a solution which works in time $O(n \cdot \log(n)^2 \cdot \text{poly}(k))$, which is unfortunately too slow to pass the time limit. This solution uses flows as a black box and uses a divide&conquer technique.

There also exists a solution which works in $O(n \cdot k^3)$ or even in $O(n \cdot k^2)$ and pushes units of flow in a smart way. Firstly, let's split each vertex into two and add an edge from the left one to the right one. In this way we prohibit pushing more than one unit of flow though the vertex. Now, there are $2n$ layers in total.

Now, let's find the augmenting path which starts in some vertex from the first layer. It has to go as far as possible, so let's find how far we can reach. Let's say that it can reach layer p. Now, let's use another DFS to find this augmenting path. Of course, we need to reverse the edges on this path. Repeat the DFS for each vertex in the first layer. What is the sum of $f(1, i)$ for all i? It's the number of reversed edges which connect two parts of a split vertex (almost, we have to subtract the number of such edges in the layer in which we currently are, which is always equal to k). We should consider the sum of results only when we are in the odd layers, i.e. in the left parts of the split vertices.

What to do when we want to calculate the paths from the second layer? Firstly, there could be some reversed edges between the old layer and new one. Let's reverse them back and forget about them. Now, there can be a possibility to push more units of flow. So, let's again try to form as many long paths as possible.

Look's like $O(n^2 \cdot \text{poly}(k))$, huh? We can observe that when we reach layer p from layer q, we can do it in the complexity $O((p-q) \cdot \text{poly}(k))$! It can be seen that the layers to which we can push a fixed number of units of flow only move right, so everything amortizes to $O(n \cdot \text{poly}(k))$. Unfortunately, this solutions is a few times slower that the first (exponential) solution, but still should pass the time limit without any problems.

C. Cloyster

A problemset is incomplete without an interactive binary search problem... Yeah, that's it, again.

Assume that the board is surrounded from all sides by infinitely small shells. Let's bisect the square grid into two rectangles and try to find out which rectangle contains the largest shell. To do that, find the middle row and find the maximum shell in there (this requires n queries). Let M be this maximum. Let's also find the sizes of the shells of all 8-connected neighbors of this maximum. Now, there are a few cases:

- if M is larger than all its neighbors, then it's surely the largest shell in the board.

- if the largest shell found **so far** is to the top of the middle row, then the largest shell in the board is to the top as well — we can imagine that this largest shell is surrounded by a rectangle containing strictly smaller shells.
- Similar case holds also for the largest shell found **so far** which is to the bottom of the middle row.

The first bisection took $n + 6$ queries and created a board of size at most $n \times \lfloor \frac{n}{2} \rfloor$. The second bisection then takes $\lfloor \frac{n}{2} \rfloor + 6$ queries and leaves us with the board of size at most $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$. Therefore, in at most $\frac{3}{2}n + 12$ queries we managed to divide each dimension of the board by 2.

By iterating such bisection process over and over again, we'll arrive at the 1×1 board in at most $3n + 12 \log_2 n$ queries. We can easily convince ourselves that it's less than $3n + 210$ queries for our constraints.

Note that in our algorithm, it's not enough to simply compare M with its neighbors to find out which rectangle can be safely discarded. That's because the property that *each shell has a larger 8-connected neighbor* doesn't hold anymore after we discard a part of the board. Indeed, depending on the implementation, the second sample test might be a counterexample for this approach.

D. Dedenne

Firstly, $f(k) := \sum_{j=1}^{k} (\lfloor \log_2 j \rfloor + 1)$ can be computed quite easily in $O(\log k)$ time using some straightforward DP.

This leads us to a simple $O(n^2)$ solution: let's preprocess all $f(k)$ for $k \in [1, n]$. Now, if $D(n)$ is the minimum cost to build a tree with n leaves, then $D(1) = f(1)$ and $D(n) = f(n) + \min_{1 \le k < n}(f(k) + D(k) + D(n-k))$.

How to speed it up? We can show that D is convex, that is, $D(n) - D(n-1) \ge D(n-1) - D(n-2)$; f is convex as well. Therefore, $k \to f(k) + D(k) + D(n-k)$ is convex as well as it's the sum of three convex functions. It means, that given the values of $D(1), D(2), \ldots, D(n-1)$, we can compute $D(n)$ using the ternary search on the result. It needs $O(\log n)$ iterations, and each iteration needs to evaluate f in $O(\log n)$ time. Therefore, each $D(n)$ can be computed in $O(\log^2 n)$ time.

Obviously, it's still not enough. In order to make our solution even faster, we'll again use the fact that D is convex. Assume that we know $D(1), D(2), \ldots, D(n)$, and let $\delta := D(n) - D(n-1)$. Let's try to find the largest $m \ge n$ such that $\delta = D(n) - D(n-1) = D(n+1) - D(n) = \cdots = D(m) - D(m-1)$. We can actually achieve this using the binary search on m! This requires $O(\log n)$ evaluations of $D(\star)$ for each binary search, giving us $O(\log^3 n)$ time for the binary search. (Luckily, if implemented carefully, for n large enough, during evaluations for $D(\star)$, we never access yet non-computed values of D.)

Now, if for each δ we store the interval of arguments n for which $D(n) - D(n-1) = \delta$, we can easily find any value $D(\star)$ in $O(\log \delta_{\max})$.

We need to repeat the binary search for each value of δ occurring for $n \le 10^{15}$. We can prove that $\delta \sim O(\log^2 n)$ (or we can experimentally check that $\delta_{\max} < 1850$). We therefore conclude with an $O(\log^5 n)$ solution. Our implementation finishes within a second, however if any contestant's solution works too slow, they can easily do the precomputing and store all 1850 intervals in the code.

E. Eevee

Precompute all binomial coefficients and factorials up to nk, for sure they'll be useful.

Let's solve the problem for all permutations, not for intervals of them. Add a number $n+1$ at the end of every permutation and assume that we are also interleaving these new numbers, all of them must be at the end of the sequence (so we allow and even force them to be neighboring), and still no other number can have all its occurrences neighboring. This will make implementation easier. At the end we'll have to divide the result by $k!$, as we don't really choose the order of these k new numbers.

For each permutation, let its x-prefix be the prefix ending at x. Let's denote by $DP[x]$ (for $1 \le x \le n+1$) the number of ways to interleave the x-prefixes of all permutations so that the numbers x are neighboring at the end of the resulting sequence and there are no other numbers which have k neighboring occurrences. It's clear that the answer will be equal to $\frac{DP[n+1]}{k!}$. How to calculate $DP[x]$? The total number of ways to interleave all the x-prefixes so that all the numbers x appear at the end of the resulting sequence (but allowing other numbers to have k neighboring occurrences as well) can be easily computed – it's a product of some binomial coefficients and factorials. We then have to subtract the ways in which some other number has k neighboring occurrences. Let's iterate over first such occurrence y. This scenario is only possible if y appears before x in all k permutations. The number of ways to create such interleaving is also a product of some binomial coefficients, factorials and $DP[y]$.

So, this is a solution which works in complexity $O(n^2 \cdot k)$, so we already know how to solve the whole task in $O(n^2 \cdot k^3)$, which is way too slow. The first idea is to calculate the result faster for all prefixes of the permutation sequence (we'll then repeat the computation k times, iterating over the start of an interval). This can be done as the factor k came from the fact that for each pair (y, x) we had to calculate the product of factorials of distances between ys and xs in all permutations in the interval. We can do it faster by maintaining this product and updating it as we add new permutations to the prefix. This gives us a solution in $O(n^2 \cdot k^2)$ time – still too slow.

Now we have to use the randomness of the permutations. The complexity in the original problem wasn't exactly $O(n^2 \cdot k)$, it was $O(n \cdot k + (\textit{number of pairs } (y, x) \textit{ such that } y \textit{ in every permutation comes before } x) \cdot k)$, as it's the number of transitions in our dynamic programming. Let's estimate this number. If the prefix contains only one permutation, we have exactly $\binom{n+1}{2}$ such pairs. Nextly, after adding each new permutation to the prefix, the number of such pairs approximately halves. Therefore, the expected number of such pair over all prefixes is also $O(n^2)$. We can use this fact and implement the solution by maintaining the set of good pairs, and performing only necessary DP transitions. When implemented properly, the solution has complexity $O(n \cdot k \cdot (n + k))$.

F. Flaaffy

Let $n = R - L + 1$. Let's first write an $O(n^3)$ dynamic programming solution. The DP will have $O(n^2)$ states:

- $L(x, r)$: we've got number x on the display, and the hidden number is in the interval $[x + 1, r]$,
- $R(l, x)$: we've got number x on the display, and the hidden number is in the interval $[l, x - 1]$.

Let's speed it up. First, notice that the answer will always be rather low – indeed, we never need to do more than $M := 45$ shocks. Therefore, we'll compute the following DP states:

- $range\_left(c, x) = \min\{l \,:\, R(l, x) \le c\}$.
- $range\_right(c, x) = \max\{r \,:\, L(x, r) \le c\}$,

Let $dist(a, b)$ be the number of digits needed to change to turn a into b.

Let's compute all $range\_right(c, \star)$ states for given c at once, assuming we already computed all lower costs. Notice that $range\_right(c, x) \geq y$ if there exists some m such that $range\_left(c - dist(x, m) - 1, m) \leq x + 1$ and $range\_right(c - dist(x, m) - 1, m) \geq y$; m is the optimal next query.

When transitioning from m to x, we'll use the following process: change some digits of m into wildcards (?), each wildcard costing 1 shock. Then fill the wildcards with some digits to get x – this costs us another shock. E.g. $31337 \to 3??3? \to 35932$.

We'll compute $range\_right(c, x)$ for increasing x. We'll keep an array D mapping each of 11^5 possible numbers with wildcards to some integer (initially $-\infty$).

When computing the value for some x, we need to:

- For each possible distance $d \in [1, 5]$ and each possible $m \in [1, 10^5)$ such that $range\_left(c - d - 1, m) = x + 1$, consider all $\binom{5}{d}$ ways to change d digits of m into wildcards. For each such wildcard string w, set $D(w) := \max(D(w), range\_right(c - d - 1, m))$.
- We can now see that $range\_right(c, x)$ is the maximum value of $D(w)$ over all 2^5 possible wildcard strings w describing x.

The simplest way to implement the steps above is to first sort all pairs (d, m) by $range\_left(c - d - 1, m)$, and then to run the steps above for each x. This allows us to compute the DP values for each c in $5 \cdot 10^5 \cdot \log(5 \cdot 10^5) + 10^5 \cdot 32 + 11^5$ time. We fill $range\_left(c, \star)$ analogously.

This process needs to be repeated M times to populate the whole DP array. If implemented more-or-less efficiently, this should be enough to fit in the TL. It's still possible to get rid of the log factor, but it's a bit messy.

G. Gurdurr

Let's denote the types of layers. (□ – nonexistent block, ■ – existent block.)

$A = $ □■□

$B = $ ■■□ = □■■

$C = $ ■□■

$D = $ ■■■

A straightforward dynamic programming would have $O(4^n)$ states and would make transitions in $O(n)$ time, which is definitely too slow. Let's make some observations.

Firstly, if we have some layer of type C, then we cannot touch it anymore. On the other hand, it can be adjacent to layers of any other type, so we can think about such a layer as a separator between two independent games. Using this fact we can reduce the number of states to $O(3^n)$ if we always split the jenga towers on each layer of the type C and calculate NIMBERS for each possibility instead of only calculating whether some configuration is wining for the current player or not.

This is still too much. Now, let's look at the layers of the type A. If such a layer touches some layer of type B or C, we cannot touch them anymore. We could almost split towers on the layers of the type A, but remembering about the fact, that together with such layer we need to remove the neighboring ones. The problem is that if a layer of type A touches a layer of type D, we still can remove one block in the future. Fortunately, this one removable block is independent from anything other in the game and also it doesn't matter if we change the layer of type D into a layer of type B or C. In other words, this single block becomes an independent game with NIMBER equal to 1, which is easy to be considered. For instance, a stack of layers of types $BBDBADDBD$ is equivalent to a sum of three games: a jenga tower BBD, a jenga tower DBD, and a single independent game with a single move (as D is adjacent to A).

Getting rid of layers of type A and C we can decrease the number of states to $O(2^n)$ and the whole complexity to $O(2^n \cdot n)$, which is sufficient. Also note that after such preprocessing we can easily answer each query in $O(n)$ time.

H. Hypno

Compute the DP for each vertex: $T(v)$ – optimal expected time necessary to reach n from v. Of course, $T(n) = 0$. Consider a single vertex v. And assume we know T's for each neighbor of v, and these T's are sorted: $e_1 \leq e_2 \leq \cdots \leq e_k$. We can show that:

- If we didn't use an edge at all, it will succeed with $\frac{3}{4}$ probability ($\frac{1}{2}$ chance to be immune to a Hypno on that edge, and $\frac{1}{2} \cdot \frac{1}{2}$ to get lucky with a Hypno we're susceptible to).
- If we used an edge and it fails, we know we're susceptible to the Hypno on it. From now on, the edge will succeed with $\frac{1}{2}$ probability.
- The optimal strategy looks as follows: for some $j \in [1, k]$, try to go to e_1; if unsuccessful, try e_2 etc. If the edge leading to vertex e_j fails, repeatedly try to go to vertex e_1.
- $T(v) \geq e_j + \frac{2}{3}$. Therefore, if we compute the T's for each vertex in the increasing order of times, all neighbors of v *needed for our strategy* will be computed far ahead of v.

With the knowledge above, we can use a variant of Dijkstra algorithm to compute all T's: for each yet undiscovered vertex v, maintain the current candidate for $T(v)$ basing on the neighbors of v discovered so far. When another neighbor (s) of v is discovered, update the candidate. We can use the fact that $T(s)$ is the largest time discovered so far and therefore it can only be the last considered neighbor in the freshest strategy for v.

When implemented correctly, the solution can be implemented in $O(m \log n)$ time.

I. Infernape

We'll prove the following fact: *if the intersection of a set of balls is non-empty, then it's a ball centered at a vertex or a midpoint of an edge.*

Let's take two balls: $B(v_1, r_1)$, $B(v_2, r_2)$; v_i is either a vertex (and then r_i is integer) or an edge midpoint (and then r_i is a half-integer). Let d be the distance between v_1 and v_2. If $d > r_1 + r_2$, then the intersection is empty. If $r_1 \geq r_2 + d$, then the intersection is exactly the second ball. (Similarly for the first ball.) In the opposite case, put v_1 at point 0 on the number axis, and v_2 at point d, so that the shortest path between v_1 and v_2 is the $[0, d]$ segment. Consider the vertex/edge midpoint v at point $x := \frac{d+r_1-r_2}{2}$. (It can be shown that this point is either a vertex or an edge midpoint, and that it's between v_1 and v_2). We can now show that the intersection is a ball $B(x, r_2 - x)$.

To make the computations simpler, we can insert a vertex inside each edge of the tree. Now, each radius is integer, and each ball center is a vertex of the tree. In order to intersect two balls, we need to be able to find the $(2x)$-th vertex on the path between v_1 and v_2. This can be solved e.g. using jump pointers technique.

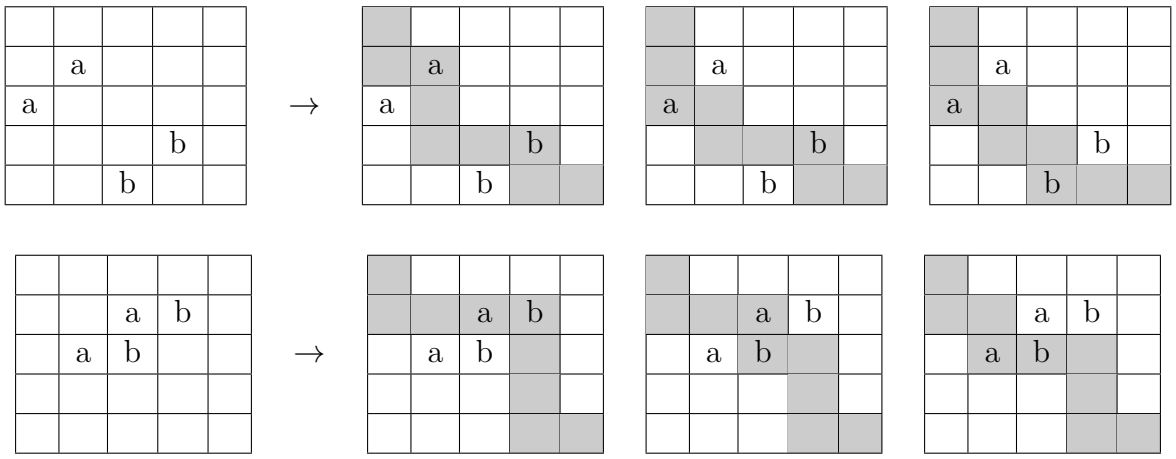
Let's make an observation. Let I_v denote the size of the intersection of all the balls except the v-th one and let I denote the size of the intersection of all the balls. The number of vertices which are covered by at least $k-1$ balls is $(\sum_{v=1}^{k} I_v) - (k-1) \cdot I$.

Now, take a single query $(v_1, r_1), (v_2, r_2), \ldots, (v_k, r_k)$. For each i, we want to intersect all balls except the i-th. It's quite straightforward when we compute the intersection of each prefix and suffix of the query. The only thing left is that for each query, we're left with a single ball and we want to

compute the number of (original) vertices inside the ball. This is a pretty simple exercise on centroid decomposition on the tree.

J. Jigglypuff

Notice there are two cases where the answer is positive. For each case, three gray paths create the same string:



Formally, let (r, c) be a *good* cell if the characters written in cells directly to its right and to its bottom are equal. Then the first case corresponds to two *good* cells (r_1, c_1), (r_2, c_2) such that $r_1 < r_2$, $c_1 < c_2$, and the second cell corresponds to two *good* cells such that $(r_2, c_2) = (r_1, c_1+1)$ or (r_1+1, c_1).

On the other hand, it can be shown that if none of the cases above occur, it's impossible to find three occurrences of any string. As it's straightforward to check the cases above in linear time, we can implement the whole solution in $O(nm)$ time.

K. Kecleon

There are many correct approaches to this problem. Let us mention the most notable of these:

Solution 1. $O(q\sqrt{q})$

Consider a static version of the problem (static string s, $|s| = n$, for each $k = 1, 2, \ldots, n$ determine how many times $s[1..k]$ occurs in s). This is a standard string problem which can be solved in linear time using either KMP or Z-function.

How can this be generalized to our problem? Every $\sqrt{q}$ operations, recompute the static version of the problem above. Now, consider a single query string $s[1..k]$. Each of its occurrences in s:

- either lies completely in the string and it's counted using the static version of the problem,
- or lies partially/completely in the most recent part of the string. However, as we recompute the static version of the problem every $\sqrt{q}$ operations, there are at most $\sqrt{q}$ substrings of length k that don't lie completely in the string. We can compare each of them with $s[1..k]$ using hashes.

Therefore, we answer each query in $O(\sqrt{q})$ time, and we recompute KMP/Z-function $\sqrt{q}$ times. The final time complexity is $O(q\sqrt{q})$.

Solution 2. $O(q \log q)$

For each $1 \leq j \leq |s|$, let $K[j] < j$ be the length of the longest border of $s[1..j]$. These values can be computed incrementally using KMP in $O(q)$ total time.

Construct a tree of size $|s| + 1$ rooted at 0. The parent of vertex v will have index $K[v]$ (this is actually a KMP suflink tree). Now, one can notice that the number of times that $s[1..j]$ occurs in s is the size of the subtree rooted at j.

In order to answer the queries dynamically, we need to be able to: (a) add a leaf to the tree, (b) find the size of a subtree. This is a fairly standard task which can be done e.g. by maintaining an Euler tour of the tree on some BST or a link/cut tree. However, as it might require your own implementation of a BST tree, this optimization isn't required.

Solution 3. $O(q \log q)$

Consider a KMP suflink tree from the previous solution. A sample $O(q^2)$ solution on it might look as follows:

- When adding a leaf to the tree, consider the whole path from the leaf to the root of this tree. Increase the sizes of the subtrees in each vertex on this path by 1.
- Each `get` query is simply asking for the size of the subtree stored in each vertex.

If we could use a heavy-light decomposition on this tree, we could easily answer the queries efficiently. However, as the tree is dynamic, the implementation of such structure is nothing but easy. We can however exploit the specific structure of the tree to construct a decomposition similar to HLD using the following lemma:

Lemma. *The set of borders of any board can be partitioned into $O(\log n)$ disjoint arithmetic progressions.*

Now, consider a vertex v, its parent p and its grandparent g. We'll say that the edge from v to p is *heavy* if $v - p = p - g$; otherwise it's *light*. Using the lemma above, we can prove that the path from each vertex to the root of the tree consists of at most $O(\log q)$ heavy paths. That's exactly the thing we wanted from HLD. We can now store Fenwick trees on each heavy path to store the sizes of the subtrees to get $O(q \log^2 q)$ runtime. By the deeper consideration of the structure of the borders, we can actually deduce it's $O(q \log q)$.

L. Lati@s

Firstly, let's notice that the tuples in the multiset are independent from each other. This gives us a hint, that we probably should calculate a NIMBER for each of them and check if their xor-sum is equal to zero or not. How to calculate this NIMBER value? It turns out that described game is well-known and the NIMBER for some tuple is equal to the NIMBER-multiplication of the elements of this tuple. Such multiplication can be done in $O(\log^2(A))$ time after $O(\log^3(A))$ preprocessing, where A is equal to the smallest number of the form 2^{2^k} for some integer k, which is strictly greater than any value in the input (so $A = 2^{2^6} = 2^{64}$, so $\log(A) = 64$).

NIMBERS less than A form a field, so addition and multiplication work in the same way as in any other field (e.g. rational or real numbers). Thanks to this property, the task is equivalent to computing the permanent of the given matrix in the NIMBERS field. Let's remember that the permanent of a square matrix is equal to $\sum_\sigma \prod_{i=1}^{n} a_{i,\sigma_i}$, where the sumation goes through every permutation of numbers from 1 to n. Calculating the permanent is hard, but due the property that

$a = -a$ in the NIMBER field (i.e., the field has characteristic 2), we can freely add minuses in any place in the formula above and change it to the formula for the determinant! The determinant can be calculated easily with Gaussian Elimination. To do so, we need to know how to calculate a multiplicative inverse of some number. The multiplicative group of the field contains $2^{64}-1$ elements, and therefore from Lagrange's theorem, $a^{2^{64}-1} = 1$ for each non-zero $a < A$. Therefore, $a^{-1} = a^{2^{64}-2}$, and this can be calculated efficiently via fast exponentiation.

Let's calculate the current time complexity. We have to calculate the multiplicative inverse n times, once for each row of the matrix. Each one requires $O(\log(A))$ multiplications, which gives complexity $O(n \cdot \log^3(A))$. Nextly, to normalize each row we need to multiply each number in it by the already calculated inverse. This gives us $O(n^2)$ multiplications, which so far gives complexity $O(n \cdot (n + \log(A)) \cdot \log^2(A))$. Nextly, we should add each row (multiplied by some number) to each other row. This would give us $O(n^3)$ multiplications, which is unfortunately too much. Fortunately, this can be sped up. When adding some row to some other one, we are multiplying each element in the former one by the same value. For one such value we can do preprocessing in $O(\log^2(A))$, which would allow us to multiply in the complexity $O(\log(A))$. Normal multiplication requires iterating over each pair of set bits – one bit in one number, second bit in the second number. As the set of set bits in the second number is constant (because we are multiplying by the same number), we can do the computations faster and avoid iterating over the bits of the second number (thanks to the preprocessing). Therefore, we can do each of $O(n^3)$ multiplications in $O(\log(A))$ time, and an $O(\log^2(A))$ preprocessing needs to be done $O(n^2)$ times.

The overall complexity is $O(n \cdot \log(A) \cdot (n + \log(A))^2)$.