Problem A. Moving Dots

To solve this problem, you had to come up with the following observations:

1. It is always optimal for each dot to pick the direction of its movement, and then move it. It is quite obvious, because if we want to move a dot from position x to $x + d$, then we need to spend only d moves, and all of them will be performed in one direction.
2. If a dot cannot be merged with another dot, then it does not have to move at all. We want to minimize the number of the dots left on the line, so if moving this dot does not influence the number of dots, then we may do not move this dot at all.
3. Let $x_1, x_2, x_3, \ldots, x_n$ ($x_i \leq x_{i+1}$) be the initial positions of the dots and let $d_i = x_{i+1} - x_i$. We can notice, that to merge two consecutive dots i and $i+1$ we have to spend at least d_i time. Let's prove, that for a chosen subset of dots A we can merge all the dots using exactly $\sum_{j \in A} d_j$ time. To merge all the dots using this amount of time, we can move all the dots, for example, to the right. Let's iterate over all indexes i from 1 to $n-1$, if i does belong to the A, then we can move it to the right and merge it with another dot.

We know, that for given subset of moving dots A we can make answer equal to $n - |A|$ and spend $\sum_{j \in A} d_j$ time. We can minimize the required time by picking the smallest possible d_j. One way to implement it is to make array d, sort it, make another array $pref_i = \sum_{j=1}^{i} d_j$ and binary search the maximal size of A we can achieve.

Total time complexity is $\mathcal{O}((n+q)\log n)$.

Problem B. Mex Permutations

We want to find the number of permutations a that have $f(a) = F$ for a given array F.

Let array a be empty at the beginning. We are going to fill it with values while iterating through F. If we are now checking position i of array F ($F_0 = 0$), then we have to place $F_i - F_{i-1}$ values in the array a. We have to place these values in the first i positions of the array a, because in other way they do not influence on the value of mex of i-th prefix of array a. So, we have to pick $F_i - F_{i-1}$ positions out of unused positions on i-th prefix, let this value be $free_i$, and we can support it after placing elements and changing prefix. So, the answer for the problem equals to the product of $A(free_i, F_i - F_{i-1})$ over all i.

To get value of $A(n, k)$ fast, we have to know values of $n!$ and $\frac{1}{(n-k)!}$. First values can be precomputed in linear time. Inverse factorials can be precomputed in linear time as well. Let $inv_n = n!^{md-2}$, where $md = 998244353$. Then we can say that $inv_i = inv_{i+1} \cdot (i + 1)$.

Total complexity of the solution will equal to $\mathcal{O}(n)$.

Problem C. Boring Problem

We are given some definitions from the statement.

$$f(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j|$$

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} f(s[i..j])$$

We can expand the *cost* formula to the following state:

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} f(s[i..j]) = \sum_{l=1}^{n} \sum_{r=l}^{n} \sum_{i=l}^{r} \sum_{j=i}^{r} |s_i - s_j|$$

Now it can be noticed, that the *cost* can be represented as:

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j| \cdot k_{i,j}$$

For some integers $k_{i,j}$ — the number of times $|s_i - s_j|$ occurs in the *cost* formula. Actually, for fixed i, j we can say that $k_{i,j} = i \cdot (n - j + 1)$, because only segments that contain both of these elements can have this sum, and each of these segments contain it exactly once.

$$cost(s) = \sum_{i=1}^{n} \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1)$$

Now, let's introduce the function $g(i)$, and represent the *cost* using g:

$$g(i) = \sum_{j=1}^{i} |s_i - s_j| \cdot j \cdot (n - i + 1) + \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1)$$

$$2 \cdot cost(s) = \sum_{i=1}^{n} g(i)$$

$$cost(s) = \frac{\sum_{i=1}^{n} g(i)}{2}$$

This way, if some element changes, then we can update the *cost* in the $\mathcal{O}(g)$ — the time we are able to compute $g(i)$. If we update position p and change its character to c, the change of cost will equal to $cost_{new} = cost - g(p) + g'(p)$, where $g'(p)$ is the value of $g(i)$ after changing s_p to c.

Without any difficult data structures and calculating g in $\mathcal{O}(n)$, we get the solution in $\mathcal{O}(n^2 + nq)$, which is able to get 34 points.

Now we will try to find $g(i)$ in $\mathcal{O}(\log n)$ with $\mathcal{O}(n \log n)$ preprocessing. To do this, we are going to expand all brackets and absolute values.

$$g(i) = \sum_{j=1}^{i} |s_i - s_j| \cdot j \cdot (n - i + 1) + \sum_{j=i}^{n} |s_i - s_j| \cdot i \cdot (n - j + 1) =$$

$$= \sum_{j=1,\, s_j<s_i}^{i} (s_i - s_j)\cdot j\cdot (n-i+1) + \sum_{j=1,\, s_j\geq s_i}^{i} (s_j - s_i)\cdot j\cdot (n-i+1) +$$

$$+ \sum_{j=i,\, s_j<s_i}^{n} (s_i - s_j)\cdot i\cdot (n-j+1) + \sum_{j=i,\, s_j\geq s_i}^{n} (s_j - s_i)\cdot i\cdot (n-j+1) \; =$$

$$= (n-i+1)\cdot \Big(\sum_{j=1,\, s_j<s_i}^{i} (s_i\cdot j - s_j\cdot j) + \sum_{j=1,\, s_j\geq s_i}^{i} (s_j\cdot j - s_i\cdot j)\Big) +$$

$$+ i\cdot \Big(\sum_{j=i,\, s_j<s_i}^{n} (s_i\cdot (n-j+1) - s_j\cdot (n-j+1)) + \sum_{j=i,\, s_j\geq s_i}^{n} (s_j\cdot (n-j+1) - s_i\cdot (n-j+1))\Big) \; =$$

$$= (n-i+1)\cdot \Big(\sum_{j=1,\, s_j<s_i}^{i} s_i\cdot j \; - \sum_{j=1,\, s_j<s_i}^{i} s_j\cdot j + \sum_{j=1,\, s_j\geq s_i}^{i} s_j\cdot j \; - \sum_{j=1,\, s_j\geq s_i}^{i} s_i\cdot j \Big) +$$

$$+ i\cdot \Big(\sum_{j=i,\, s_j<s_i}^{n} s_i\cdot (n-j+1) - \sum_{j=i,\, s_j<s_i}^{n} s_j\cdot (n-j+1) + \sum_{j=i,\, s_j\geq s_i}^{n} s_j\cdot (n-j+1) - \sum_{j=i,\, s_j\geq s_i}^{n} s_i\cdot (n-j+1)\Big) \; =$$

$$= (n-i+1)\cdot \Big(s_i\cdot \Big(\sum_{j=1,\, s_j<s_i}^{i} j - \sum_{j=1,\, s_j\geq s_i}^{i} j\Big) \; - \sum_{j=1,\, s_j<s_i}^{i} s_j\cdot j + \sum_{j=1,\, s_j\geq s_i}^{i} s_j\cdot j \Big) +$$

$$+ i\cdot \Big(s_i\cdot \Big(\sum_{j=i,\, s_j<s_i}^{n} (n-j+1) - \sum_{j=i,\, s_j\geq s_i}^{n} (n-j+1)\Big) \; - \sum_{j=i,\, s_j<s_i}^{n} s_j\cdot (n-j+1) + \sum_{j=i,\, s_j\geq s_i}^{n} s_j\cdot (n-j+1)\Big)$$

Now we have achieved the sum formulas where each sum depends only on j. Using these formulas, we can support these values in a data structure that allows adding at the point and asking sum on the segment — Segment Tree or Binary Indexed Tree, for example. But still, we don't know to find sum over all j such that $s_j < s_i$ or $s_j \geq s_i$. The solution for this is quite easy, let's support 26 such Data Structures in order to support these values, i-th data structure represents the i-th lowercase letter of English alphabet.

Problem D. Graph? Are you sure?

Let's firstly solve the problem with only first and second query types, and query of second type are being asked after all queries of first type were executed. We will increase restrictions on c further.

Firstly, we need to check whether vertexes a and b belong to the same component. We can use DSU (Disjoint Set Union) for this. If they are from different components, we need to output -1. From now on, let's assume that a and b belong to the same component.

1. $c = 1$. A simple path is considered good if each value written on the edge has an even number of entries. As here we have $c = 1$, all number written on edges are equal, so we have to check whether the length of the simple path is even. The length of the path between vertexes a and b equals to $d_a + d_b - 2 \cdot d_{lca(a,b)}$, where d_v denotes distance from v to some root vertex, and $lca(a, b)$ denotes lowest common ancestor of a and b. As here we need only parity of length, we can say that parity of length of the path is the same as the parity of $d_a + d_b - 2 \cdot d_{lca(a,b)} = d_a + d_b$ ($= d_a \oplus d_b$, where $\oplus$ is XOR operation). We can pick any vertex as root, calculate d_a and d_b from it and use for calculating parity afterward.

2. $c \leq 8$. Using the idea from the previous solution, we can achieve a hash-like algorithm. Let's say that if color i has the odd number of entries on the path, then i-th bit of the number will equal to 1. Using this logic, we can extend the previous algorithm to XOR of distances.

3. $c \leq 4 \cdot 10^9$. If we tried to apply the same trick here, it would consume 2^c memory, which is obviously too much. If we started hashing, let's continue hashing? Let's say that $hash(value) = x$, where x is a random number in range $[0; 2^{63})$. Now we can check whether the path is good by looking at XOR of hashes of values. If it is equal to zero, then this path is good and no otherwise. Why would it work? Actually, it does not work with 100% probability. However, we can prove that the chance of this algorithm having collision is so small, that it can be omitted.

   How to prove this? Firstly, let's assume that all number that are being XORed are pairwise distinct. Let's look at each bit separately. When we XOR with new number, we XOR each bit with either 1 or 0. Thus, the new value of each bit will be either 0 or 1 with equal probability — $\frac{1}{2}$. Thus, having 0 in each bit at the same time is equal to $(\frac{1}{2})^b$, where b is a number of bits in the number.

   As we have $\mathcal{O}(n^2)$ number of ways in the graph and each way may have a collision, the chance of collision happening is $\frac{n^2}{2^b} \approx 2 \cdot 10^{-14}$.

Now we are going to solve the problem with full constraints. We want to somehow maintain the XOR value of hashes on the way from any vertex to its root (as there may be more than 1 component). We can do this using DSU and smaller to larger merging. Let's introduce some definitions:

- $root$ — $root_v$ is the root vertex of vertex v;
- xr — xr_v is XOR of hashes on the way from v to $root_v$;
- sz — sz_v is the size of the component, with v being the root of the component;
- $cnt_{root,xor}$ — number of $xr_v = xor$, such that $root_v = root$;
- ans_v — number of good paths in component with root v;
- $total$ — total number of good path in all components.

When we want to add edge from a to b with number num on it, we are going to do the following:

1. Let's say that $A = root_a$ and $B = root_b$;
2. If $sz_A > sz_B$, then we are going to swap a and b, and A and B;

3. Now we are going to add all vertexes v from component with root A, to component with root B:
   (a) Make $root_v = B$;
   (b) Make $xr_v = xr_v \oplus xr_a \oplus hash(num) \oplus xr_b$. It can be expressed as the following: we go from vertex to its root, then to vertex a from the root, then pass the new edge and go from vertex b to its root. If we pass some vertexes twice, they will not be counted in XOR;
   (c) Add cnt_{B,xr_v} to ans_B and $total$;
   (d) After all vertexes were processed, add 1 to cnt_{B,xr_v} for each v in component with root A.
4. Add sz_A to sz_B;
5. Add ans_A to ans_B.

Now to answer the queries, we are doing the following:

- For first type query, do the algorithm described above;
- For second type query, check whether they belong to the same component, $root_a = root_b$, and whether $xr_a = xr_b$. If the XOR values are equal, then the answer is `YES`, otherwise the answer is `NO`;
- For third type query, print ans_{root_u};
- For fourth type query, print $total$.

Why is this fast enough? The small to large merging trick is well-known. It works in $\mathcal{O}(n \log n)$ because each element from the set will change its root no more than $\mathcal{O}(\log n)$ root vertexes. When we change the root vertex, it means that the size of the set we are adding these vertexes to is greater or equal to the current size. Thus, the size of the resulting set is at least twice as large as size of the smaller set.

As for storing cnt array, we would use `map`, as the XOR values may be way too big, the resulting and total complexity of the problem is equal to $\mathcal{O}(n \log^2 n)$.

Problem E. Ball momentum

This problem can be solved using two pointers approach. Let's sort both arrays, so $w_i \leq w_{i+1}$ and $p_i \leq p_{i+1}$. Now, for each element of array w we are going to find the smallest element in array p with which these balls will form a good pair. A pair $(i; j)$ is called good if any other pair $(i'; j')$ ($i \neq i'$ and $j \neq j'$) has difficulty of catching less or equal to the difficulty of catching the pair$(i; j)$. Obviously, the bigger w_i is, the smaller p_j we need. Thus, let's keep the pointer on the smallest element in p that makes a good pair with i. While we can make a pair $(i; j-1)$, we can decrease j. Thus, the answer is just the sum of $n - j + 1$ over all values j for each i.

Problem F. Typical Query Problem

Firstly, let's analyze the conditions of segment being good provided in the statement:

- $len(S) \geq L$ — for a segment $[l; r]$ we need to have $r - l + 1 \geq L$ or $r - L + 1 \geq l \Leftrightarrow l \leq r - L + 1$;
- $avg(S) \leq C$ — $\frac{\sum S_i}{|S|} \leq C \Leftrightarrow \sum S_i \leq C \cdot |S| \Leftrightarrow \sum (S_i - C) \leq 0$.

Thus, we can subtract C from each element of a in the beginning and check whether there exists a segment with sum less or equal to zero.

We can use prefix sums to fast compute the sum on the segment, so let $pref_i = \sum_{j=1}^{i} a_j$, thus $sum[l; r] = pref_r - pref_{l-1}$. As we want to minimize this sum, we can fix r and take $val = \max_{i=1}^{r-L} pref_i$, thus $pref_r - val$ will be the minimal sum of the segment S with $len(S) \geq L$. If for any r this sum is less or equal to zero, then the answer is `YES`.

Doing this results in $\mathcal{O}(n \cdot q)$ complexity.

Let's make the first and the most important observation: if at some moment of time the answer is `YES`, then after any update the answer will still be `YES`.

We can use binary search on the point of first entrance of `YES` in answers. Let's apply first t operations on the array and check whether there is a good segment S.

How we can check this in linear time? Let's make an array $b_i = a_i - a_{i-1}$ ($a_0 = 0$), thus, if we add x to segment $[l; r]$ array b changes only in two positions: $b_l = b_l + x$ and $b_{r+1} = b_{r+1} - x$. We can restore array a as following: $a_i = \sum_{j=1}^{i} b_i$.

After finding first moment t we have to print t `NO` answers and $q - t$ `YES` answers.

Considering the solution described above, we have achieved the final complexity of $\mathcal{O}(n \log q)$.

Problem G. Tree queries

Let's denote *cur* as XOR value of all vertexes in the initial graph and sz_v as number of vertexes in the subtree of vertex v. Let's track how *cur* changes during the queries. If sz_v is odd, then *cur* value will change to $cur \oplus x$, where $\oplus$ denotes XOR operation, or it will remain unchanged otherwise. Remembering two properties $0 \oplus x = x$ and $x \oplus x = 0$, this observation is obvious.

Now let's look at *cur* by bits. We can count the number of trees, where this bit is set, let it be $f(bit)$, then $ans = \sum_{bit} f(bit) \cdot 2^{bit}$. How we can count the $f(bit)$? Consider some cases:

1. no query changes the value of *bit*-th bit:
   (a) *cur* has *bit*-th bit set — $f(bit) = 2^q$;
   (b) otherwise — $f(bit) = 0$.
2. $f(bit) = 2^{q-1}$ otherwise.

First two cases are quite obvious, now we should prove last case. Let cnt_0 be the number of queries that does not change the *bit*-th bit in *cur*, and $cnt_1 = q - cnt_0$. Let's consider two cases, when we have *bit*-th bit set in *cur* and do not have it set.

Suppose *bit*-th bit is set, then we have to choose even number of queries that change *bit*-th bit in *cur*. Although, we can use any queries that do not change *bit*-th bit in *cur*.

$$f(bit) = 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1, 2i) = 2^{cnt_0} \cdot \sum_{i=0} (C(cnt_1 - 1, 2i - 1) + C(cnt_1 - 1, 2i))$$

$$= 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1 - 1, i) = 2^{cnt_0} \cdot 2^{cnt_1 - 1} = 2^{cnt_1 + cnt_0 - 1} = 2^{q-1}$$

If we suppose that *bit*-th bit is not set, then we can follow the same logic: we can choose any queries that do not change *bit*-th bit in *cur*, and choose **odd** number of queries that change *bit*-th bit.

$$f(bit) = 2^{cnt_0} \cdot \sum_{i=0} C(cnt_1, 2i + 1) = 2^{cnt_0} (2^{cnt_1} - \sum_{i=0} C(cnt_1, 2i))$$

$$= 2^{cnt_0} \cdot 2^{cnt_1 - 1} = 2^{cnt_1 + cnt_0 - 1} = 2^{q-1}$$

This finishes the proof. Total complexity of the solution is $\mathcal{O}(q \log x)$ what is $\mathcal{O}(60 \cdot q)$ in this problem.

Problem H. Anton the Guard

Let's look through some solution with different time complexity.

The first solution has the time complexity of $\mathcal{O}(n!)$. We can brute force the order of visiting vertexes using `next_permutation`. Then check whether $deep_{order_i} \leq deep_{order_{i+1}}$ for all $1 \leq i < n$. Here, $deep_v$ denotes the minimal number of roads that need to be traveled to reach vertex v from vertex 1. To calculate the distance between two vertexes, we can either use LCA in $\mathcal{O}(\log n)$, or just use BFS in $\mathcal{O}(n)$.

The second solution involves a similar idea. We can do bitmask dp, as there are only $\mathcal{O}(2^n)$ states, and we can easily traverse, going to the vertex we can go to and have not been there before. It works in $\mathcal{O}(2^n \cdot n \cdot \log n)$.

Let's spot the first critical observation. We can look at the graph by layers. On layer i there will be all vertexes v with $deep_v = i$. Let's denote the set with these vertexes v as $layer_{deep_v}$. Let dp_v be the minimal distance to visit all vertexes u such that $deep_u \leq deep_v$ following the condition in the statement. We can choose the starting vertex in the previous layer and bruteforce all possible orders of visiting vertexes in the next layer. We can use two approaches that were already mentioned above. First algorithm will have complexity of $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot |layer_{i+1}|!)$, and the second one has complexity of $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot 2^{|layer_{i+1}|})$.

We can use the idea from the previous two subtasks — we have to try to make a transition from layer on depth i to layer on depth $i+1$. Let's make another critical observation:

For sets of vertexes $T = layer_i$, $S = layer_{i+1}$ with $|S| > 1$, with $L = lca(S)$, we have to traverse all the edges of subtree of vertex L till vertexes on layer $i+1$ one or two times, let sum of edges in this subtree be sum_edges. Specifically, if we start at vertex u on the layer i and finish at vertex v on layer $i+1$, then $dp_v = dp_u + 2 \cdot sum\_edges - dist(u, v)$. We can prove it in the following way. When we come to any vertex p that has unvisited vertexes $r \in S (r \neq v)$, then we have to go to this subtree and visit all vertexes r before leaving it. Thus, we go to the neighbor vertex and back, traversing each of these edges twice. As we visited the useful edges except of that edges that belong to the path, we will traverse them only once.

As we found the formula of recalculating values of dp on the following layer, we achieved complexity of $\mathcal{O}(\sum_{i=1}^{n} |layer_i| \cdot |layer_{i+1}| \cdot \log n)$. Let's analyze the formula $dp_v = \min dp_u + 2 * sum\_edges - dist(u, v)$. For all vertexes $v \in S$ calculate minimal value of $dp_u - dist(u, v)$, where $u \in T$. We can do this in lineal time. Firstly, split $dist(u, v)$ to $dist(u, LCA(u, v)) + dist(v, LCA(u, v))$, where $LCA(u, v)$ denotes the lowest common ancestor of u and v. Let us calculate the minimal value of $dp_u - dist(u, LCA(u, v))$, so we should store $minpath_{1,t}, minpath_{2,t}$ — two minimal values of $\{dp_u - dist(u, t); r\}$ in vertex t, where u is in the subtree of r and $u \in T$ and r is son of t. Assume that $minpath_{1,t,1} \neq minpath_{2,t,1}$. For $t \in T$ we store just $\{dp_t, t\}$, for $t \notin T$ we calculate minimum by finding two minimal values of $\{mindist_{1,r,0} - W_{u,v}; r\}$, where $W_{u,v}$ denotes the weight of edge from u to v. Then assume t equals $LCA(u, v)$, then

1. $\min dp_u - dist(u, v) = \min mindist_{1,t,0} - dist(u, t)$ for $mindist_{1,t,1} \notin Parents_v$;
2. $mindist_{2,t,0} - dist(u, t)$ for $mindist_{2,t,1} \notin Parents_v$;

Where $Parents_v = \{parent_v \cup Parents_{parent_v}\}$.

We have recalculated $minpath$ values from bottom to up. Now let us go in the opposite direction — from top to bottom and support $value$ — the minimal value of the part of the dp formula. Let us push values from v' to its children, then for all $u' \neq mindist_{1,v',1}$, and $parent_{u'} = v'$ we push into u' minimal value of $mindist_{1,v',0} - W_{v',u'}$ and $value - Wv', u'$, while for $u' = mindist_{1,v',1}$, we push minimum of $mindist_{2,v',0} - W_{v',u'}$ and $value - Wv', u'$. As a result we have $value$ in the $u' \in S$, that denotes the minimal value of dp transition formula.

Thus, we can solve this problem in $\mathcal{O}(\max deep_v \cdot n)$, which is likely to be $\mathcal{O}(n^2)$.

Now we have almost working solution, let's try to think of some vertexes that do not really influence on the answer. We can delete such vertexes. Let's say that vertex v during recalculating dp on layer i to layer $i+1$ is important if it has at least two important sons. Vertexes on layer $i+1$ are important. If the

vertex is unimportant, it can de deleted from the graph. Let's prove, that while having c vertexes on layer $i+1$ the total number of important vertexes on layers less or equal to i is at most c. As each important vertex has at least two important sons, the number of vertexes on the previous level decreases in at least 2 times. Thus, $c + \frac{c}{2} + \frac{c}{4} + \frac{c}{8} + \cdots = 2 \cdot c$.

We can spot that we delete some vertexes on the previous layer. There are two such cases:

- Vertex v has 0 important sons — we can prove that it is never optimal to start from v on the previous layer and finish at some vertex u on the next layer. Consider the number of times we pass the edge from v to its parent. As we finished in v we passed it once on the previous layer. If we want to start from vertex v, then we have to pass it again. Thus, it will be visited 2 times. Although, if we try to start from some vertexex u that has important sons, the number of times we pass each edge will not change, except for an edge from v to its parent.
- Vertex v has 1 important son — this vertex is deleted, by we use it for the recalculating the dp values.

We have to be careful with some things during the implementation:

1. There is only 1 vertex on the next layer. We have to recalculate the dp using the parent of the vertex;
2. The root of the tree may be deleted — be careful with it.

Thus, if we can compress the graph using the algorithm described above we can achieve the complexity of $\mathcal{O}(\sum_{i=1}^{n} 2 \cdot |layer_i|)$, which equals $\mathcal{O}(2 \cdot n) = \mathcal{O}(n)$.

Finally, we solved this problem in linear time. For better understanding of the editorial recommend you to check author solution.

Link to the my solution: `https://pastebin.com/TJLQWjzC`