Problem Tutorial: "Participants entry"

Problem author: Alexey Mikhnenko

Problem developer: Artyom Agafonov

We will simulate the process. Let us iterate over the participants in increasing order of their numbers. If the color of the current participant is the same as the color of the previously released participant, then we put this participant into the queue. Otherwise, we mark that this participant is the next one to be released, and then release the participant waiting in the queue, if the queue is not empty. Finally, after we have processed all participants, we release all participants that are waiting in the queue. Thus, we obtain a linear simulation of the process, and therefore this solution works in $O(qn)$.

Notice that if, during such a simulation, the queue is empty and the color of the participant's T-shirt differs from the color of the previously released participant, then this participant will be released under his own number. Notice that all participants with numbers smaller than the current one do not affect the arrangement of this participant and all subsequent ones, so we may assume that for such participants the process starts anew, with their number taken into account.

Suppose we start the process from participant number i. How do we find the next participant from whom we can again assume that the process starts anew? Put $+1$ at the positions of participants whose T-shirt color is a_i and -1 at all other positions. Then the smallest index j such that the sum from i to j inclusive is 0 will be the desired one. For each position i, draw a directed edge to $p_i = j$. These edges form a directed forest.

Let us understand how to answer a query using this forest. Starting from the first participant, we will gradually follow the edges while the next participant has number at most y. Suppose we stop at i; then, by the property above, we may assume that the process starts from participant i. Notice that when arranging participants on the segment from i to $p_i - 1$, positions i, $i+2$, $i+4$, ..., $p_i - 1$ will be occupied by participants whose T-shirt color is a_i, and positions $i+1$, $i+3$, ..., $p_i - 2$ will be occupied by all the others. Therefore, to determine the position of participant y, we need to find out how many participants with color a_i stand before him. For this, we can use binary search over the array of positions of participants with T-shirt color a_i. Such an array can be easily maintained under update queries.

Thus, the problem is split into two parts: we need to dynamically maintain this forest when the colors of two adjacent participants change, and quickly find the edge covering position y. The constraints in some subgroups allowed simplifying the implementation of these parts of the solution. Below, we present the solution for the full score, using link-cut trees. Another option is root decomposition; with careful implementation, it also allowed earning full score in this problem.

The second part, which consists of finding the covering edge, is implemented by the expose operation from vertex 1, which makes it possible to extract the path containing all participants from whom the process will start anew. The participant of interest can then be found by descending the resulting splay tree.

If we determine for which participants and how the values p_i change, then these changes can be maintained in the tree using link and cut operations. Notice that for colors different from a_x and a_{x+1}, both participants contributed -1 to the corresponding sums. Therefore, swapping them does not affect the sums. If the colors a_x and a_{x+1} are the same, then nothing changes. Consider the case $a_x \neq a_{x+1}$. First, the outgoing edges from x and $x+1$ will change. Notice that for color a_x, the contributions of positions x and $x+1$ changed from $+1$, -1 to -1, $+1$. Now, when computing the sum starting from some position i of color a_x, the algorithm may obtain 0 at position x. This means that previously the sum up to position x was equal to 2. Knowing this, we can find the position where the prefix sum for color a_x turned out to be 2 less than at x. For example, this can be implemented by descending a segment tree for color a_x, where the leaf i stores the sum on the prefix up to the i-th participant with T-shirt color a_x (note that it is enough to store such sums, since between such participants the prefix sums decrease linearly). This position will be the one to which the edge would have pointed, and now we must redirect it to x. By the way the edges are drawn, there will be exactly one such edge. Similarly, we can analyze the case for color a_{x+1} — from it we will also obtain at most one candidate whose value in the array p changes. To find the

position to which the new edge will point, we need to make one more query in the corresponding segment tree to find the first occurrence after position i of a prefix sum that is 1 smaller than the prefix sum before position i.

Thus, we obtain a solution that works in $O(q \log n)$. Also, for full score it was possible to implement solutions working in $O(q \log^2 n)$, where the splay tree in the link-cut tree is replaced by a treap, or in $O(q\sqrt{n})$, where the whole array is divided into blocks of size $\sqrt{n}$ and for each element we maintain a compressed jump that points to the first element outside the current block that can be reached by moving upward along the edges. Then, for an update query, we need to update this jump for elements from at most 4 blocks, and the search for the covering edge is performed by following such compressed jumps until the block containing the query is reached.

Problem Tutorial: "Permutations and Queries"

Problem author: Alexey Mikhnenko

Problem developer: Alexey Vasiliev

Let us represent a permutation as a set of points (i, p_i). Then the first type of operation transforms all points (x, y) into $(n - x + 1, y)$. The second type of operation transforms (x, y) into $(x, n - y + 1)$. The third type of operation transforms (x, y) into (y, x).

Notice that using any number of these operations, we can obtain no more than eight different permutations, which we describe as follows. Let f_1, f_2, f_3 be three variables that take values 0 or 1.

Let p be our initial permutation, and then we apply the following modifications to it in order:

1. If $f_1 = 1$, then we perform the third type of operation (that is, we transform (x, y) into (y, x)).
2. If $f_2 = 1$, then we perform the first type of operation (that is, we transform (x, y) into $(n-x+1, y)$).
3. If $f_3 = 1$, then we perform the second type of operation (that is, we transform (x, y) into $(x, n-y+1)$).

Initially, we precompute the cost of the eight permutations corresponding to every possible set of variables f_1, f_2, f_3.

Now we maintain the current variables f_1, f_2, f_3, which are initially equal to zero.

- For a query of the first type, we replace f_2 with $1 - f_2$.
- For a query of the second type, we replace f_3 with $1 - f_3$.
- For a query of the third type, we replace f_1 with $1 - f_1$ and swap f_2 and f_3.

Thus, after each query, we maintain the current variables f_1, f_2, f_3 and output the precomputed cost for such flags. The running time is $O(n \log n + q)$, where the log comes from binary exponentiation used to compute the cost.

Problem Tutorial: "Tasks from Sasha"

Author and problem developer: German Perov

Restatement of the problem

We need to find the number of ways to split the vertices into $k + 1$ sets, such that the lowest common ancestor of the i-th set ($1 \le i \le k$) is equal to x_i (the set numbered $k+1$ contains the vertices corresponding to the residents whom Sasha will not invite to solve the problems).

Solution for $k = 1$

Notice that Sasha can invite residents only from the subtree of vertex x_1.

- if Sasha invites the resident from vertex x_1, then any resident from the subtree can be either invited or not invited. There are $2^{sz(x_1)-1}$ such ways
- if Sasha does not invite the resident from vertex x_1, then he must invite at least one resident from at least two subtrees. It is a bit more convenient to compute this value as $cnt_{all} - cnt_1 - cnt_0$, where cnt_{all} — the total number of ways to invite residents from the subtrees (that is, $2^{sz(x_1)-1}$), cnt_1 — the number of ways to invite at least one resident from only one subtree (that is, $\sum_{(x_1,u)\in E} (2^{sz(u)}-1)$), and cnt_0 — the number of ways to invite no residents at all (that is, 1).

Solution for $p_i = i - 1$

- each x_i must belong to the i-th set;
- a vertex v whose task will not be solved can belong to set j ($j \le k$) if and only if x_j is an ancestor of v. To find the number of ways to choose a set for a vertex, it is enough to count how many vertices above it are in the array x. There is also an option to put this vertex into set $k + 1$.

It is enough to compute the product of the number of ways to choose a set for v over all vertices.

Solution in $O(nk)$

Let us reformulate the problem so that instead of assigning vertices to sets, we color them. We need the lowest common ancestor of the vertices of color i to be equal to x_i (there will also be a dummy color $k+1$, for which no such condition is required).

We will define $dp[v][c]$ as the number of ways to correctly color the subtree of v, while having c extra colors (which will later be needed to cover vertices from x outside the subtree of v)

- if vertex v is not in x, then simply $dp[v][c] = c \cdot \prod_{(v,u)\in E} dp[u][c]$. This follows from the fact that to obtain c extra colors, it is enough to take the corresponding colorings for the subtrees and choose the color for v itself
- if vertex v is in x, then this vertex must be covered. We will process the children one by one and maintain three values: $cnt[c][0]$, $cnt[c][1]$, and $cnt[c][2]$, where $cnt[c][i]$ is the number of ways to end up with c free colors, while coloring at least one vertex in i child subtrees with the color of vertex v (states with $i \ge 2$ are indistinguishable for us, so we assume that $cnt[c][2]$ stores the total number of ways for all $i \ge 2$). Then adding the next child u is expressed as follows:
  - if a free color will be used in u (it is colored with the color of v) and c colors must remain, then $newcnt[c][min(i + 1, 2)]$ should be increased by $cnt[c][i] \cdot (dp[u][c + 1] - dp[u][c])$
  - if a free color will not be used in u, then $newcnt[c][i]$ should be increased by $cnt[c][i] \cdot dp[u][c]$

  Then $dp[v][c] = (cnt[c][0] + cnt[c][1] + cnt[c][2]) + cnt[c][2]$. The first three terms correspond to the case "v is colored with its own color, the subtrees can be colored arbitrarily". The second term corresponds to the case "v is colored with a free color, and vertices must be colored in at least two subtrees".

The answer to the problem is $dp[root][1]$ (the one means that we have one free color left for $k + 1$).

Thus, we obtained a dynamic programming solution computable in $O(nk)$.

Solution in $O(n + k^2 + k^{1.5}\sqrt{n})$

Let us call vertices from x bad, and vertices not in x good.

Suppose that a good vertex v has a good child w.

We obtain the formula

$$dp[v][c] = c \cdot \prod_{(v,u)\in E} dp[u][c] = c \cdot \left(\prod_{w\neq u\ (v,u)\in E} dp[u][c] \right) \cdot dp[w][c]$$

$$dp[v][c] = c \cdot \left(\prod_{w\neq u\ (v,u)\in E} dp[u][c] \right) \cdot c \left(\prod_{(w,t)\in E} dp[t][c] \right)$$

Notice that we can separately group the power of c and the product of dynamic programming states for a fixed c. This corresponds to the fact that if there is an edge (v, w) between two good vertices, then we can remove the lower vertex and attach the subtrees of w to v (while not forgetting that the vertex w itself can also be colored; for this, we propose maintaining a weight at vertices — how many vertices this one corresponds to).

After this processing, there will be no two good vertices connected by an edge. Therefore, in the resulting tree there will be $2k$ vertices plus some number of good leaves (possibly n). At the same time, note that the total weight of the good leaves cannot exceed n.

Thus, we obtained a solution in $O(n + k^2 + leaves \cdot k)$, where $leaves$ is the number of good leaves after the tree processing described above. It remains to learn how to efficiently process a vertex to which a set of good leaves of some weight is attached.

Let us compute the array $cnt[c][i]$ on good leaves. For convenience, let us assume that l leaves with weights $w_1, w_2, \ldots, w_l$ are attached to a vertex. For now, we know how to compute $cnt[c][i]$ for all c in $O(k \cdot l)$ and want to do it faster.

- $cnt[c][0] = c^{\sum w_j}$. This corresponds to simply coloring vertices in each subtree with c free colors.
- $cnt[c][2] = (c+1)^{\sum w_j} - cnt[c][0] - cnt[c][1]$. This follows from the fact that $cnt[c][0]$, $cnt[c][1]$, and $cnt[c][2]$ cover all possible ways to color the children's subtrees while leaving c free colors. The total number of such ways is, in turn, $(c+1)^{\sum w_j}$
- it remains to understand how to compute $cnt[c][1]$. Notice that when adding the next w, we get

  $$newcnt[c][1] = cnt[c][0] \cdot ((c+1)^w - c^w) + cnt[c][1] \cdot c^w = newcnt[c][0] \cdot \left(\left(\frac{c+1}{c} \right)^w - 1 \right) + cnt[c][1] \cdot c^w$$

  From this it follows that

  $$\frac{newcnt[c][1]}{newcnt[c][0]} = \frac{cnt[c][1]}{cnt[c][0]} + \left(\frac{c+1}{c} \right)^w - 1$$

  So, we can maintain the value $\frac{cnt[c][1]}{cnt[c][0]}$ and update it when a new weight is added. Thanks to this trick, we can add not one weight, but several equal weights at once.

Having the array $w_1, \ldots, w_l$, let us count how many times each weight appears and feed the algorithm not only the weights, but also their multiplicities. We obtain the computation of $cnt[c][i]$ for all c in $O(k \cdot distinct(w_1, \ldots, w_l)) \le O\left(k \cdot \sqrt{\sum w_j}\right)$ for a fixed vertex.

We obtained the bound $O\left(k \cdot \sqrt{\sum w_j}\right)$ for processing the good leaves of one bad vertex. Now let us estimate the total time for processing all leaves.

Let W_i be the total weight of good leaves attached to vertex x_i. Since the total number of vertices is at most n, we have the constraint $\sum_{i=1}^{k} W_i \le n$, and the running time is $\sum_{i=1}^{k} k\sqrt{W_i}$. The worst case is $W_i = \frac{n}{k}$ for all i, and the total time for processing leaves is $O\left(\sum_{i=1}^{k} k\sqrt{\frac{n}{k}}\right) = O\left(k^{1.5}\sqrt{n}\right)$

Thus, the total running time of the whole algorithm is $O\left(n + k^2 + k^{1.5}\sqrt{n}\right)$

Problem Tutorial: "Anxiety Before the Olympiad"

Author and problem setter: Alexey Mikhnenko

Suppose that Alexander has already talked to the first k participants, and the current answer is ans. Let P be the sum of the excitement levels of all participants at the current moment in time whom Alexander has already talked to. In this sum, we count the participants who have already entered the competition hall with coefficient $a_i + t \cdot b_i$, as if they were still standing in the queue.

Let us track the value $Q = \text{ans} - P$. Notice that if Alexander talks to a participant with excitement level x, then both ans and P increase by x, so this difference does not change. Thus, for a fixed value of t, the value Q remains unchanged.

Now let us consider what happens to Q when t increases by 1. If B is the sum of the values b_i of all participants Alexander has already talked to, then P increases by B, which means that Q decreases by B. Now consider the moment when Alexander finishes talking to the participants. Since initially $Q = 0$, at that moment $Q = -\sum B_t$ over all passed moments in time. Then $\text{ans} = Q + P$. That is, if we fix the moment when Alexander finished, as well as the number of participants he managed to talk to, the answer depends only on $\sum B_t$.

At the same time, there is no point in fixing the moment in time and the number of participants, since Alexander can always wait until all participants he has already talked to enter the competition hall. That is, it is enough to fix only the number of participants he eventually talked to. If we fix this value (let us call it k), it is beneficial for Alexander to always be at the minimum possible value of B_t for a fixed moment in time t. If there are no restrictions on Alexander's plan, then this minimum is the value B_i, where i ranges from t to k. If there are restrictions on the plan, then i is also subject to an additional upper bound, since Alexander cannot talk to some participants earlier than his plan allows.

For a fixed value of k, the problem can be solved in $\mathcal{O}(n)$ if we explicitly find these restrictions on i for each t and sum the minima on the corresponding segments. This makes it possible to solve the problem in $\mathcal{O}(n^2)$.

To improve the running time of this solution, one should notice that Alexander, between the fixed points of his plan, as well as after the last fixed point, will be located only at suffix minimum elements ignoring the last $n - k$ participants. Thus, if we iterate over the number of people he eventually talks to, from the last fixed point to n, we can explicitly maintain a stack of positions where he will be located. In parallel, we can maintain the value Q, which can then be used to compute ans. This approach makes it possible to solve the problem in $\mathcal{O}(n)$.