Problem Tutorial: "Solving problems"

Author and developer: Yan Olerinskiy

Let's consider two sequentially solved problems at t_i and t_{i+1}. If the time zone k may be the answer to the problem, then the problems should be solved either on the same day or on consecutive days. In other words, there should not exist a day d such that:

$$\begin{cases} t_i < d \cdot m + k \\ t_{i+1} > (d+1) \cdot m + k - 1 \end{cases} \iff t_i + 1 \le d \cdot m + k \le t_{i+1} - m$$

Taking both parts modulo m, we obtain that if $t_{i+1} - t_i > m$, then all k modulo between $((t_i + 1) \bmod m)$ and $(t_{i+1} \bmod m)$ satisfy this condition, and therefore cannot be the answer. Here there is a special case when $t_{i+1} - t_i \ge 2m$, but this is easy to handle as in this case, the answer is obviously -1.

Thus, each pair of sequentially solved problems blocks a certain cyclic interval of time zones. The minimum unblocked time zone can be found using scanline algorithm, which allows solving the problem in $\mathcal{O}(n \log n)$.

Problem Tutorial: "Yet another queries"

Author and developer: Dmitry Sosunov

Let's build a segment tree on the array a of size 2^n and on the array b of the same size, where $b_{i \oplus 2^k} = a_i$. How do these trees differ? It is claimed that the second segment tree differs from the first one only in the order of children of each node at the $(n-k)$-th level.

From here it easily follows that to get the segment tree builded on the array b, where $b_{i \oplus k} = a_i$, we need to change the order of children of each node on all levels i, if the $(n-i)$-th bit of k is 1.

This remark allows us to answer queries using a segment tree builded on the array a, but changing the order of children of each visited node, if necessary.

Thus, this problem is solved similarly to a lazy segment tree in $\mathcal{O}(2^n + qn)$.

Problem Tutorial: "Legs warm-up exercise"

Author: Kirill Kudryashov, developer: Igor Markelov

First of all, let's solve the problem on the tree without addition and removal of edges using heavy-light decomposition. For each vertex, we will maintain the number of paths and their total length starting at this vertex. But we will only consider the paths go down the tree, and their first edge will be light.

For each path in the decomposition, we will calculate the total length of the paths in the original tree, with the highest vertex belonging to this path. To achieve this, we will build a segment tree. When merging two vertices, it is necessary to calculate the total length of the paths passing between them. To do this, for each node of the segment tree, we need to maintain the total length and the number of paths for each side and direction. Also it is necessary to maintain the orientation of each edge in the tree. But this can be easily done with the same heavy-light decomposition.

This segment tree also allows us to orient a segment of edges in one direction, which is necessary for solving the problem. As for the implementation nuances, when changing the orientation of edges on some path of the decomposition, the total length and number of paths that need to be maintained for each vertex may change for some vertices higher up the tree. Therefore, when processing changes in the orientation of edges, it is necessary to consider not only the paths that the query path was decomposed into but also all paths higher up the tree. This solution can be implemented in $\mathcal{O}(n \log n)$.

To handle queries for adding and removing edges, instead of heavy-light decomposition, we will use a link-cut tree. In this case, the implementation will differ slightly from the standard one, as when reconstructing paths in the «expose» function, we need to carefully recalculate the total length and the number of paths starting from the lowest common vertex of the two paths in the tree. As a result, we obtain a solution in $\mathcal{O}(n \log n)$ with a very large constant due to the substantial amount of information that needs to be stored in the nodes of the link-cut tree.

Problem Tutorial: "Hard problem"

Author: Alexey Mikhnenko, developer: Alexey Datskovskiy

Let's notice that if we can solve the problem as a a-th problem and as a b-th one, then we can rearrange the problems to solve the given problem as the i-th, for any $a \le i \le b$. This is true, because when swapping two neighboring problems, the position of the problem in the sequence changes by at most 1. Therefore, we only need to find for each problem the minimum and maximum sequence number by which this problem can be solved.

To achieve the minimum sequence number, it is advantageous for us that during each iteration through the problems, Gena solves at most 1 problem, and he solves our problem as soon as it becomes available. To emulate this process, one can iterate through all the problems in descending order of difficulty and maintain a current «buffer» of problems (i.e., problems that are already available in terms of difficulty but have not yet been solved). Thus, Gena must solve one problem with every decrease in difficulty, and all newly available problems are added to the buffer.

To achieve the maximum sequence number, the strategy needs to be the opposite: it is not difficult to arrange things so that all the more difficult problems are solved earlier, but it is also necessary that before selecting a certain problem, as many problems with lower difficulty as possible have been solved. To accomplish this, one can maintain all the problems that are easier than the current one, available at the moment.

Altogether, in order to answer a query, it is only necessary to check whether the requested position falls within the achievable range. This solution can be implemented in $\mathcal{O}\left((n+q)\log n\right)$.

Problem Tutorial: "Colorful graph"

Author: Alexander Babin, developer: Alexey Vasilyev

First, we need to check if the graph is connected. If not, then the graph cannot be good.

Now, let's solve the problem for a fixed graph. Let's consider some subset of colors C and leave only the edges with colors from this set in the graph. In this graph, we will find a spanning forest, suppose it contains x_C edges. Then, if $x_C < |C|$, the graph cannot be good, because no matter how we choose one edge of each color from C, they will form a cycle. It is claimed that if the condition $x_C \ge |C|$ is satisfied for all subsets of edges C, then the graph is good. The proof of this fact will be given later.

Let's fix a specific subset of colors C and efficiently check this condition. Let the number of edges in the graph consisting only of edges from C be e_C. Notice that if $e_C > \frac{|C|\cdot(|C|-1)}{2}$, then $x_C \ge |C|$. Therefore, first check this condition, and if it is not satisfied, then naively find the size of the spanning forest in $\mathcal{O}(|C|^2)$. Thus, for a fixed graph, the problem can be solved in $\mathcal{O}\left(2^k \cdot k^2\right)$.

To solve the problem with updates, we need to efficiently store the set of edges of each color. This can be implemented using std::set, which allows solving the problem in $\mathcal{O}\left(q \cdot \left(2^k \cdot k^2 + \log m\right)\right)$.

Now let's prove that if $x_C \ge |C|$ for all C, then the graph is good. The necessity is obvious and has been proven earlier. It remains only to prove sufficiency.

Let's look at an arbitrary spanning tree of the entire graph. If this tree contains an edge of each color, then the answer is already found. Otherwise, there is some color c_1 that does not appear in the tree. Let's

consider an arbitrary edge of color c_1 and prove that there exists a spanning tree containing the color c_1, as well as all the colors that are already in the tree.

Notice that when adding this edge to the tree, it will form a cycle. If on this cycle some color appeared twice in our tree, then we can remove one of the edges of this color and add an edge of color c_1, which proves this fact. Otherwise, let the cycle contain colors $c_2, \ldots, c_a$. Then, according to our condition, the tree of edges only of colors $c_1, \ldots, c_a$ contains at least a edges. This means that somewhere outside our cycle there is an edge of one of the colors $c_1, \ldots, c_a$, which is not included in the tree. Then we will again look at the cycle it forms and try to remove one of the edges of the cycle and add an edge of c_1. If this cannot be done, we will do the same and so on. Notice that each time we increase the set of edges that have ever been on a similar cycle, and therefore the algorithm is finite. Therefore, we will definitely be able to find a tree in which there will be an edge of color c_1, as well as all the colors that are already in the tree.. This completes the proof.

A similar algorithm to the proof allows solving the problem in $\mathcal{O}(q \cdot (poly(k) + \log m))$, but this was not required in the problem. Also, similar asymptotics can be achieved using matroids intersection algorithm.

Problem Tutorial: "Tournament"

Developers: Anton Stepanov and Philip Gribov

First of all, let's something similar to the minimum search algorithm. To do this, initially we'll set the minimum to be equal to 0. Then, we'll iterate over the vertices in arbitrary order, and each time we'll make a query about the minimum and current vertex. If the edge is directed towards the minimum, we won't change anything, otherwise we'll assign the number of the current vertex to the minimum.

In total, we will spend $n-1$ queries to do it, after which we will obtain a structure in which the outdegree of each vertex, except for the minimum, is equal to 1.

After, we will go through all the vertices except for the minimum in arbitrary order. If we have not yet made a query about the minimum and the current vertex, we will make it. If the edge is directed towards the minimum, then the next vertex cannot be the answer, because its outdegree is definitely greater than 1. If the edge is directed away from the minimum and the outdegree of the minimum is already definitely greater than 1, then we will break the iteration.

If the outdegree of the minimum still hasn't exceeded 1 after this, then the minimum obviously is the answer. Otherwise, the outdegree of each vertex, which could potentially be the answer, is equal to 1. Thus, if we make a query about two such vertices, the outdegree of exactly one of them will exceed 1.

There is a problem: we have already asked about some pairs of vertices. However, if we carefully examine the structure of the queries we have made, we can notice that if we remove the orientation of the edges and delete all vertices that definitely cannot be the answer, we will obtain a forest (a set of trees). Thus, as long as in the forest there are at least 3 vertices, there are two leaves for which we have not yet made a query, so we will query about them and remove one of them.

After this step, there will be no more than two candidates for the answer. Up to this point, we have spent no more than $2n-2$ queries in total, because after the part of the solution involving the search for the minimum, each query, except possibly one, removed one vertex from the candidates.

Now we can naively check each candidate, using no more than $n-1$ additional queries for each. As a result, we have obtained a solution in no more than $4n-4$ queries. With a bit more careful assessment of the last part, we can obtain an assessment of $4n-7$ queries.

Problem Tutorial: "Space accident"

Author: Alexey Mikhnenko, developer: Victor Romanenko

Let's note that if it is possible to cool the reactor in k cycles, then it is also possible for any number of cycles greater than k. Therefore, binary search can be used to find the minimum number of cycles.

Now, we need to understand how to check whether a specific number of cycles is sufficient.

Suppose we want to check if m cycles are enough. We can break down the problem into 3 cases:

1. $A > B$
2. $A = B$
3. $A < B$

In any of the cases, let's first subtract m times B from each element.

In the first case, we will count how many times we need to subtract $A - B$ from each element to make it less than 0. If the total number of subtractions is $\leq m$, then m cycles are enough to cool the reactor.

In the second case, it is enough to check that all elements become negative.

The third case is similar to the first, but instead of subtracting $A - B$ from each element, we will add $B - A$ until the elements are less than 0. If the total number of additions is greater than m or if any element was already greater than or equal to 0, then m cycles are not enough.

Note that the additions/subtractions in the first and third cases for each element can be performed using a formula in $O(1)$. Therefore, the overall complexity of the solution is $O(n \log C)$, where C is the maximum value of t_i.

Problem Tutorial: "Lunch"

Author and developer: Stepan Kuznetsov

Let's divide the polygon into n triangles, where the i-th triangle is formed by point C, the i-th, and the $(i+1)$-th points of the polygon. Let $S_i = 1$ if the i-th triangle contains a special point, and $S_i = 0$ otherwise. Now, let's consider the moves we can make:

- If the i-th and $(i+1)$-th triangles still exist (meaning the i-th, $(i+1)$-th, and $(i+2)$-th points have not yet been replaced with C), then the point $(i+1)$ can be replaced with C if $S_i = 1$ or $S_{i+1} = 1$.
- If the i-th triangle no longer exists, but the $(i+1)$-th triangle still exists and $S_{i+1} = 1$, then the $(i+1)$-th point can be replaced with C.
- If the $(i+1)$-th triangle no longer exists, but the i-th triangle still exists and $S_i = 1$, then the $(i+1)$-th point can be replaced with C.

Therefore, the problem can be reformulated as follows:

- Given a string S of length n, consisting of zeros and ones. Two players take turns making moves, starting with Alice.
- On each turn, a player can choose two adjacent symbols in the string (the 1-st and the n-th symbols are considered adjacent) and, if at least one of the symbols is equal to one, replace both symbols with zeros.
- The player who cannot make a move loses the game.

If all the symbols in the string are equal to 1, then it doesn't matter what the first move Alice makes, so the problem is reduced to the case where the string contains at least one zero.

If the string contains at least one zero, let's find all the cyclic segments of ones. Note that any move contains at least one symbol from some such segment, and moreover, each move cannot immediately affect two segments of ones. Therefore, the game divided into independent games on such segments of ones.

For each segment of ones, it's easy to calculate the Sprague-Grundy function, since if the segment has a length of m, then in one move, you can either obtain a segment of length $m-1$, or create two new segments with lengths k and $m-2-k$ (where $0 \le k \le m-2$). Thus, for each m from 0 to n, we can calculate the Sprague-Grundy function for a segment of m ones in $\mathcal{O}(n^2)$.

To solve the problem, we only need to note that after adding or removing a point in the binary string, at most one symbol changes. Therefore, these segments can be easily maintained using, for example, std::set. Consequently, it's possible to maintain the bitwise XOR of the Grundy functions over all segments, which determines the winner of the game.

This yields a solution complexity of $\mathcal{O}(n + (m+q)\log n)$.

Problem Tutorial: "Paired roads"

Author: Alexander Babin, developer: Tikhon Evteev

To begin with, let's solve the problem in $\mathcal{O}(n^2)$ using dynamic programming on subtrees:

We will consider $dp[u][cnt][flag]$, where

- u —- vertex number
- cnt — number of constructed pairs of roads in the subtree
- $flag$ (0 or 1) — whether the edge $u - p(u)$ was taken in at least one of the pairs of roads.

The value of $dp[u][cnt][flag]$ is the maximum profit in the subtree, under the given conditions.

To recalculate the dynamic programming, we will need another one. It allows us to recalculate the value of the dp in the vertex u through the values in the sons of u.

We will consider $dp_1[u][cnt][i][flag][parity]$, where

- u — current vertex number
- cnt — number of constructed pairs of roads in the considered part of the subtree of u
- i — number of the first unconsidered son of u
- $flag$ (0 or 1) — whether the vertex u was taken as the central one.
- $parity$ (0 or 1) — the parity of the number of edges for which u is the central vertex.

The value of $dp_1[u][cnt][i][flag][parity]$ is again the maximum profit in the considered part of the subtree of u, under the given conditions.

A bit more about the parameter *parity*: We want to take some pairs of edges, with the central vertex u, but when recalculating the dp, we do not want to choose both sons forming a pair of edges with u at the same time. Instead, we will go through the sons one by one (parameter i), and keep in the parameter *parity* - whether we have an extra edge in the vertex u that still needs to find a pair to form a correct pair of edges with the center at u.

Let's give a couple of examples of transitions in the dp (there are a total of 11: 8 — do not take an edge in the son, 2 — take an edge in the son as the first in the pair, and 1 — as the second in the pair)

- $dp_1[u][cnt][i][1][0] \to dp_1[u][cnt + cnt_{son}][i + 1][1][0]$, using the value $dp[son_i][cnt_{son}][1]$ in the son i — do not take an edge in the i-th son, it has already been taken in one of the pairs with the center at son_i
- $dp_1[u][cnt][i][0][0] \to dp_1[u][cnt + cnt_{son} + 1][i + 1][1][1]$, using the value $dp[son_i][cnt_{son}][0]$ in the son i — take an edge in the i-th son as the first in the pair, while before that we did not take pairs of edges with the center at u
- $dp_1[u][cnt][i][1][1] \to dp_1[u][cnt + cnt_{son}][i + 1][1][0]$, using the value $dp[son_i][cnt_{son}][0]$ in the son i — take an edge in the i-th son as the second in the pair of edges with the center at u.

You may have noticed that the state $flag = 0, parity = 1$ is unreachable, so these two parameters can be combined in the code, so that instead of 4 states, there are only 3, but this is not necessary.

After we have gone through all the sons, we can update the value $dp[u][cnt][0]$ through $dp_1[u][last][cnt][0][0]$ and $dp_1[u][last][cnt][1][0]$, and the value $dp[u][cnt][1]$ through $dp_1[u][last][cnt][1][1]$

In such a dynamic, there are a total of $\mathcal{O}(n \cdot k)$ states, and if we limit the value of the parameter cnt in the subtree u by the number of edges in the subtree u, divided by 2 (+ 1), then all recalculations will take a total of $\mathcal{O}(n^2)$ time. (and not $\mathcal{O}(n \cdot k^2)$, as it may seem at first glance).

Finally, let's move on to the complete solution:

Consider the function $f(k)$ — the answer to the problem, depending on the required number of pairs of edges. It is claimed that on the interval $\left[0, \frac{(n-1)}{2}\right]$ this function is non-strictly convex. This statement can be proved by induction on subtrees, but the proof is much more complex than the rest of the problem solution. Also, all the values of the above-mentioned dynamics are convex with respect to the argument cnt (the number of taken pairs of edges).

This allows us to apply *lambda*-optimization to the described dynamic: Instead of storing cnt as a parameter, we will consider $dp[u][flag]$ and $dp_1[u][i][flag][parity]$, the values of which are: the maximum value [profit in the considered subtree minus $cnt \cdot Cost$], for some fixed value of $Cost$ — the "cost" of taking each pair of edges. For a fixed value of $Cost$, such dynamic is calculated in $\mathcal{O}(n)$

In addition to the maximum value, we will store the possible range of values of cnt at which this maximum is achieved.

By binary search on $Cost$, we will find such a value, at which k, given in the input data, lies in the range of possible values of cnt. Then the answer to the problem is the value of $dp + k \cdot Cost$.

To restore the answer (the actual pairs of edges, not just the number), we will go through the recalculation of the dp in reverse order, and if the next recalculation gives the maximum and can give the required value of cnt, then we will go through it, obtaining states from which the answer can be restored recursively. To do so, we need for each state of dp calculate not only the maximum profit, but also the minimum and maximum of cnt at which this maximum can be achieved.

The final time complexity of the solution: $\mathcal{O}(n \cdot \log(C))$.