Moscow workshop - ICPC Asia Seoul - Editorial

Maksim Gorokhovskii, Ramazan Rakhmatullin

November 2021

A Ant colonies

Let's consider the case where we have only two types of nodes: turned on and off. Our query then becomes finding two closest nodes on a path that are both turned on.

This task is easily solvable with standard HLD, for a node in segment tree we store leftmost turned on node, rightmost turned on node, and best possible answer on a segment. Merging is obvious.

To solve the original task we have to use the fact that queries are offline. Now we have two solutions:

- create virtual tree (data structure) for each color beforehand (include query endpoints as well), then solve an easier version on each virtual tree independently.
- loop through colors and process queries on the original tree, clearing turned on nodes after finishing with the color.

B Double Rainbow

Let's fix left endpoint of a segment and iterate right endpoint in increasing order. Each time amount of elements inside and outside of segment changes only for one color, so we can find a moment when we have all the colors inside a segment and a moment when some color become absent outside of a segment.

Time complexity $O(n^2)$, though can be optimized to $O(n)$ using two pointers.

C Find The House

Read the statement and do what it says.

D Friendship Graphs

Let's construct a complement graph. Then we have to divide nodes in two parts such that the graph becomes bipartite and the absolute difference between sizes of parts is minimum possible.

If the graph isn't bipartite, then the answer is -1. Otherwise, each component is represented as a pair (a, b), and for each pair we assign one of the values for the first part, and the other value for the second part. The goal is to minimize the absolute difference between parts, which can be done via knapsack dp in $O(\frac{n^2}{32})$ or faster because there are at most $\sqrt{n}$ different values.

E Grid Triangle

We need to find the number of pairs of points (x, y, z) and (x_1, y_1, z_1) such that all x, y and z are pairwise distinct integers different from zero, both these points are contained in a rectangle $[-A;\ A] \times [-B;\ B] \times [-C;\ C]$ and three sets $\{|x|, |y|, |z|\}$, $\{|x_1|, |y_1|, |z_1|\}$, $\{|x - x_1|, |y - y_1|, |z - z_1|\}$ are the same.

First let's assume that $x, y, z > 0$ and in the end multiply answer by 8. Also let's say that $x > y > z$ and try all permutations of (A, B, C) instead.

Now we can notice that

- $x_1 > 0$, otherwise $|x - x_1| > x$
- at least one of y_1 or z_1 is negative, otherwise each of $|x - x_1|, |y - y_1|, |z - z_1|$ is less than x
- $x \neq x_1$, $y \neq y_1$, $z \neq z_1$, otherwise corresponding difference will be 0. This also means that $|y_1| = x$ or $|z_1| = x$, so if both y_1 and z_1 are negative, then at least one of $|z - z_1|$, $|y - y_1|$ is greater than x, which means that exactly one of y_1 and z_1 is negative

Let's consider both cases

- $y_1 < 0 \Rightarrow z_1 = x$

  $z_1 - z < x$, $x - x_1 < x \Rightarrow y - y_1 = x$

  1. $x_1 = z$, $y_1 = -y \Rightarrow x = 2y \Rightarrow z < \frac{x}{2} \Rightarrow |x - x_1| = x - z > y$, which is impossible
  2. $x_1 = y$, $y_1 = -z \Rightarrow x = y + z$
     Here we get two points $(y + z, y, z)$ and $(y, -z, y + z)$

- $z_1 < 0 \Rightarrow y_1 = x$

  Similarly we can get points $(y + z, y, z)$ and $(z, y + z, -y)$

Now we can iterate over y for example and find the range of suitable z for each one (keep in mind that $z < y$)

Total complexity is $O(3! \cdot C)$, where $C = 10^7$

F John's Gift

Given two arrays a and b, we need to remove one element from a to minimize the score given in the problem.

Let's sort a and b in increasing order. Then there exists the best matching in terms of a score that doesn't have crossing edges. For example, if we have n elements on each side, it means that the best matching is (a_i, b_i). On the other hand, if we remove element i from a, then the best matching would look like $(a_1, b_1) \ldots, (a_l, b_l), (a_{l+1}, b_{l+2}), \ldots, (a_{i-1}, b_i), (a_{i+1}, b_{i+1}), \ldots, (a_n, b_n)$ or in a similar way but the shift would be on the right side.

First, to handle both cases we solve the task twice, one more time for reversed arrays. Now we assume that the shift is placed left from i. Then we can calculate a simple dp_1 and dp_2: $dp_1[i]$ is the minimum score on prefix i of a if we had only (a_i, b_i) edges, while $dp_2[i]$ is the minimum score on prefix i of a if the last edge is (a_i, b_{i+1}). Using this dp, calculating answer is obvious.

G Logical Warehouse 2

If I had a dollar for every time I see this problem... *(I would have at least 4 dollars)*

Let's say that some vertex is covered if there is a warehouse with the distance at most k to the vertex. Consider a following greedy algorithm: root a tree and while there are some uncovered vertices, choose the deepest one, go k steps up (stop at the root if you can't go further), and put a warehouse there (let's say this vertex is u). It's easy to prove that this algorithm is correct, since we have to put at least one warehouse in the subtree of vertex u, and if it's not u, we can always move warehouse to the parent vertex without leaving some vertices uncovered.

To do this efficiently, let's maintain some *dp*. For a vertex v let's store the deepest uncovered vertex in it's subtree if there is one. Otherwise, store the distance to the closest warehouse. To merge *dp*-s of children, find closest warehouse and farthest uncovered vertex among their *dp*-s. Check if this warehouse can cover this vertex (sum of their depths from current vertex is at most k). Then put a warehouse in the parent vertex if there is still an uncovered vertex and it's depth is k and then update *dp* of the vertex accordingly.

Total complexity is $O(n)$

H Postman

This task is about handling a lot of corner cases. I will omit some corner cases like $w = 0$ or $w = 1$, so be sure to write stress test if you want to solve it.

Define a to be the array of positions from the input sorted in ascending order, it to be the 0-based index of the last element less than 0 (-1 if no such exist). For each element we need to assign how would we end here, $\rightarrow$ means coming from the left, $\leftarrow$ means coming from the right. We have to construct a path that contains exactly w $\leftarrow$. Let's say that $b_i = \{\leftarrow, \rightarrow\}$ defines the assignment.

Notice, that if $a_0 < 0$, then $b_0 = \leftarrow$. Same goes for a_{n-1}.

First, let's solve $t = 1$ case, where we can finish anywhere.

Consider the case where $it = -1$ (no elements to the left of the start).

- if $w = n$, then -1 since $a_{n-1} = \rightarrow$
- otherwise consider some assignment. If we have some segment $\leftarrow, \ldots, \leftarrow, \rightarrow$, then the best way to cover it would be to go to the $\rightarrow$, then going to the leftmost $\leftarrow$ while visiting all points, then proceeding forward to the right (the last part doesn't exist if $\rightarrow$ is actually the last element).
- if we want to calculate the score of any assignment b, then it would look like $a_{n-1} - 0 + \sum_{i:b_i=\leftarrow} (1 + [\text{closest } \rightarrow \text{ to the right is not the last}]) \cdot (a_{i+1} - a_i)$. So the task is to find an assignment with w $\leftarrow$ minimizing this sum. To do that, we can iterate where the second rightmost (not b_{n-1}) $\rightarrow$ would be, say i, then we have to find sum of $\max(0, w-(n-i-2))$ minimum elements among $a_j - a_{j-1}$ for $j \leq i$, which can be done with set.

Now we assume that $a_0 < 0$ and $a_{n-1} > 0$, otherwise we can reverse array and use previous case. We also solve the task twice, one more time for the reversed array, to make sure that we can assume that we first go to a_0, then go to a_{n-1}.

- if $w = 0$ or $w = n$, then -1 since $a_0 = \leftarrow$ and $a_{n-1} = \rightarrow$
- otherwise we can notice that if $w \leq it + 1$, then the answer is $(0 - a_0) + (a_{n-1} - a_0)$, by putting required $\leftarrow$ to the left.

- if $w > it + 1$, then assign $w = w - it - 1$, and solve the case $it = -1$ for the subarray $a_{it+1} \dots a_{n-1}$

Done with $t = 1$, now let's handle $t = 2$ case. Say fin to be the index of an element where we need to finish. We also assume that $fin > it$, reversing array if that's not the case.

Before continuing, we need to cover one tricky case. if $it \neq -1$ (we have elements on both side) and $w = 1$ and $fin \neq n - 1$, then our answer would look like $(a_{n-1} - 0) + (a_{n-1} - a_0) + (a_{fin} - a_0)$. Otherwise I claim (without proof) that the answer would always look like going to a_0 then going to the right part $a_{it+1} \dots a_{n-1}$ and doing some stuff there. We will utilize that.

We again start with $it = -1$ case.

- if $w = n$, then -1 since $a_{n-1} = \rightarrow$
- if $fin = n - 1$
  - if $w = 0$, then $a_{n-1} - 0$
  - otherwise, if $w = n - 1$ then -1
  - otherwise, $w < n-1$ and we just have to find w smallest elements among $2 \cdot (a_i - a_{i-1})$ for $i < n - 1$ and add them to $a_{n-1} - 0$
- otherwise, $fin < n - 1$
- if $w = 0$, then -1 since we must be able to return from a_{n-1}
- if $n - 1 - fin \geq w$, then we can assign $w \leftarrow$ to elements among $fin \dots n - 2$ and the answer is $(a_{n-1} - 0) + (a_{n-1} - a_{fin})$
- otherwise if $w = n-1$, then -1 since we know because of the previous point that $fin \neq 0$ so we need at least one additional $\rightarrow$
- otherwise set $w = w - (n - 1 - fin)$, since we put all $b_j = \leftarrow$ for $fin < j < n - 1$
- now just find w smallest elements among $2 \cdot (a_i - a_{i-1})$ for $i < fin$

Now we assume that $it \neq -1$, we have $a_0 < 0$ and $a_{n-1} > 0$

- if $w = 0$ or $w = n$, then -1
- otherwise answer is $2 \cdot (0 - a_0)$ plus the answer for the case $it = -1$ for subarray $a_{it+1} \dots a_{n-1}$ for $w = \max([fin \neq n - 1], w - it - 1)$

I Security System

First let's represent the polygon as two arrays $down[i]$ and $up[i]$ which describe the height of both horizontal segments intersecting line $x = i + 0.5$ ($down[i] < up[i]$), this will be possible after coordinate compression. Let's make two dp-s: $dp_1[i]$ is the smallest number of tracks needed to cover first i columns of polygon such that the last track is horizontal and it ends in column i. And $dp_2[i]$ is the same, except the last track is vertical and it is in the i-column.

We need four transitions, and I will do them backwards.

- horizontal — horizontal

  This is the easiest one. For each i calculate how far left we can extend horizontal track if we choose the best y-coordinate for it. We can extend it to $j \leqslant i$ if $\max(down[j..i]) < \min(up[j..i])$, this j can be found with binary search. So we can update $dp_1[i]$ with $\min(dp_1[j..i-1]) + 1$.

- vertical — horizontal

  We need to find the smallest j such that vertical track in j-th column and horizontal track, which ends in i together will cover everything from j-th to i-th column. For that we need to calculate for every vertical track where is the closest uncovered vertex to the right. One can see that this is exactly the smallest $j > i$ where $down[j-1] > down[j]$ or $up[j-1] < up[j]$. This can be done with linear precalculation, you only need to maintain smallest such index while going to the left. Then find farthest allowed vertical segment with binary search.

  Then as in previous case, update $dp_1[i]$ with $\min(dp_2[j..i-1]) + 1$.

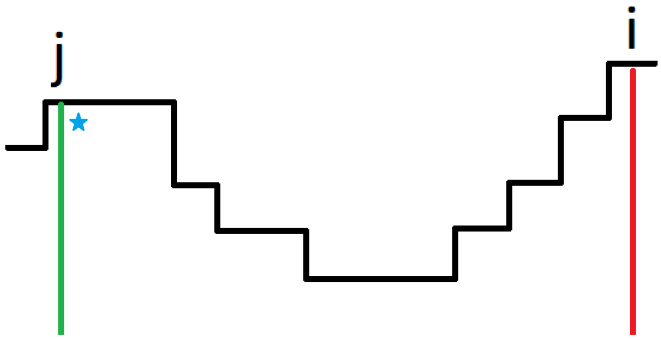
- horizontal — vertical

  This is similar to previous one. You can do the same precalculation to the left, and then just take $dp_1[j]$ from there

- vertical — vertical

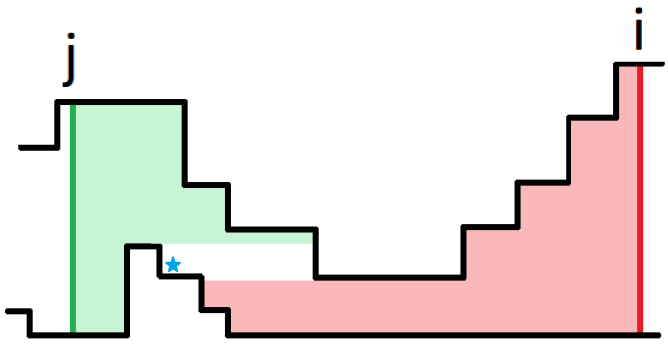
  Here for each i we need to calculate smallest j such that vertical tracks in columns j and i will cover everything between them.

  First go from i to the left while up is non-decreasing. Then go to the left while it's non-increasing.



  As in this picture, if we want to put a vertical track in column i, here is the smallest j we can take to cover everything between tracks, otherwise the blue star won't be covered.

  Do the same thing for $down$ and pick maximal index. However, this is not enough, since there is a case



  Which just means that we need to make sure that $\max(down[j..i]) \leqslant \min(up[j..i])$, this again can be easily found with binary search

Total complexity is $O(n \log n)$

J Squid Game

Suppose $X \leqslant Y \leqslant Z$ and let's try to decrease the smallest element by half using at most $\log C$ moves, $C = 10^9$.

Let's first consider a case where Y divides X. Then we can assume that $X = 1$. Now for each power of 2 from 2^0 to $2^{\lfloor \log Y \rfloor}$ if i-th bit in Y is 1, pour from Y to X. Otherwise pour from Z to X. It's easy to see that

1. before i-th iteration X is equal to 2^i
2. if i-th bit in Y is 1, then it will become 0 after i-th iteration, making $Y = 0$ in the end
3. pouring from Z to X is always possible, since in the beginning $Z \geqslant Y \geqslant X$

Now suppose $Y \mod X = R$. If we do the same operations as we would do in the case of $X' = 1$, $Y' = \left\lfloor \frac{Y}{X} \right\rfloor$, we will end up with R in the second bucket. So let's do that if $R \leqslant \frac{X}{2}$.

Otherwise, let's do operations as we would do them for $X' = 1$, $Y' = \left\lceil \frac{Y}{X} \right\rceil$. This can mean that the last operation for $i = 2^{\lfloor \log Y' \rfloor}$ is impossible. However, before this operation we actually have $2^i X$ water in the first bucket and $2^i X - X + R$ in the second one. Now if instead we pour water from the first bucket to the second one, we will get $X - R$ water in the first bucket in the end, which is again less than $\frac{X}{2}$.

We spend $\lfloor \log C \rfloor$ operations on each iteration and after each iteration the smallest element decreases by at least half, so there will be at most $\lfloor \log C \rfloor$ iterations. Thus, the total number of operations is at most $\log^2 C < 1000$

K Stock Price Prediction

Let's calculate some hash of an array so that we can use it to check whether two sequences are an R-match, and it is easy to maintain for a sliding window over array y.

For simplicity let's assume for now that all numbers are distinct. Consider a sequence p. Now build an array q such that $q[i]$ is the index of the i-th lowest element in p. Finally build an array $t[i] = q[i] - q[i-1]$, since it's easier to maintain. Now, two sequences p and p' are an R-match iff their corresponding arrays t and t' are the same.

Let's maintain a polynomial hash of an array t for a sliding window to calculate hash of any subsegment $y[i..i+m-1]$. And let's actually store not just $t[i]$, but $t[i] \cdot p^i$, where p is the base of our polynomial hash. When we add a new element to the sequence, we need to change some values of t in the place where we inserted it, and multiply everything to the right by p. Erasing element is similar. Notice that if we store t in some binary tree, we don't need any lazy propagation, since we are only interested in the value in the root, thus storing size of each subtree is enough. Also, we don't need any self-balancing trees, since we know all numbers in advance and set + segment tree of size n is enough.

To account for equal numbers, we can order them by their positions in the array. But also notice that in this case for sequences $[1, 1]$ and $[1, 2]$ resulting arrays t are the same, so instead make an array $t'[i] = (t[i], b[i])$, where $b[i]$ is a boolean meaning whether i-th and $(i-1)$-th smallest elements in p are the same or not.

Total complexity is $O(n \log n)$ with segment tree, or $O(n \log m)$ with a treap, for example

L Trio

Fix first element A of the triple and split all other numbers into groups by the subset of positions where the digit is equal to the corresponding digit of A. Now in each of these groups

we can forget about positions in this subset and focus on the rest. We have the following problem: given some digit strings with length up to 4 find the number of pairs of these strings such that strings in one pair don't have the same digit in any position.

This can be solved with inclusion-exclusion. The number of pairs with string s of length k is

$$\sum_{mask=0}^{2^k-1} (-1)^{popcount(mask)} \cdot \left| \underbrace{\{\text{strings t such that } mask[i] = 1 \Rightarrow s[i] = t[i]\}}_{D} \right|$$

To calculate the size of set D efficiently, we can use precalculation. For each string t and mask $mask$ define *masked string* as string t_{mask} where

$$t_{mask}[i] = \begin{cases} t[i], & mask[i] = 1 \\ 0, & mask[i] = 0 \quad \text{(since statement says that there is no 0 in string } t\text{)} \end{cases}$$

Now D is the number of such strings t that there exists mask m such that $t_m = s_{mask}$. To calculate that, iterate over all t and m and add 1 to corresponding masked string in some map or array of size 10^4.

The total complexity is $O(2^k n^2)$, where $k = 4$ is the length of a string.