Petrozavodsk Programming Camp Contest 6

olmrgcsi and his friends

August 30, 2023

A: Abstract

- Let $c_i = \sum_{(i,j)\in E} 2c_j$ if i has at least one outgoing edge. Otherwise, $c_i = 1$.
- Let $S = \sum a_i c_i$.
- Observe that after each second, S become $\lfloor \frac{S}{2} \rfloor$. And, the process get stopped when $S = 0$.
- Hence, the answer is $1 + \lfloor \log_2 S \rfloor$.
- Note that S can be a $O(n)$-bit integer, so you might need to implement big integers.
- The total time complexity is $O(\frac{nm+n^2}{k})$ where k depends on the implementation of big integers.

B: Bocchi the Rock

TODO

C: Computer Network

- If there are two consecutive $+$s before a $/$, you should move them after the $/$ to minimize the number of operations.

C: Computer Network

- If there are two consecutive $+$s before a $/$, you should move them after the $/$ to minimize the number of operations.
- Enumerate the number of times $/$ is used.

C: Computer Network

- If there are two consecutive $+$s before a $/$, you should move them after the $/$ to minimize the number of operations.
- Enumerate the number of times $/$ is used.
- Each a_i, b_i corresponds to some inequalities once you write the whole thing down.

C: Computer Network

- If there are two consecutive $+$s before a $/$, you should move them after the $/$ to minimize the number of operations.
- Enumerate the number of times $/$ is used.
- Each a_i, b_i corresponds to some inequalities once you write the whole thing down.
- See if the inequalities share a solution, it is also easy to find out the number of $+$s needed to fulfill the inequalities.

D: Digit DP

- Easiest way to fit the initial large array into a data structure is to use persistent segment tree.

D: Digit DP

- Easiest way to fit the initial large array into a data structure is to use persistent segment tree.
- For all $m < n$, if we have two copies of the persistent segment tree formed by $a_1, \ldots a_m$, we can add a_{m+1} to all values in one of them and merge the two persistent segment trees to one. This way, we get the persistent segment tree formed by $a_1, \ldots a_{m+1}$.

D: Digit DP

- Easiest way to fit the initial large array into a data structure is to use persistent segment tree.
- For all $m < n$, if we have two copies of the persistent segment tree formed by $a_1, \dots a_m$, we can add a_{m+1} to all values in one of them and merge the two persistent segment trees to one. This way, we get the persistent segment tree formed by $a_1, \dots a_{m+1}$.
- Range queries can be handled the same way it is usually done in ordinary segment trees. Total complexity is $O(nq)$.

E: Except One

- Let's consider calculating all elementary symmetric polynomials for $1, 2, \cdots, p-1$

E: Except One

- Let's consider calculating all elementary symmetric polynomials for $1, 2, \cdots, p-1$
- Let q be a primitive root of p.

E: Except One

- Let's consider calculating all elementary symmetric polynomials for $1, 2, \cdots, p-1$
- Let q be a primitive root of p.
- For a term x of degree $m < p-1$, since $\{1, 2, \cdots, p-1\} = \{q, 2q, \cdots, (p-1)q\}$, we should have $x = xq^m$ modulo p, but that means $x = 0$ $(q^m - 1 \neq 0 \bmod p)$

E: Except One

- Let's consider calculating all elementary symmetric polynomials for $1, 2, \cdots, p-1$
- Let q be a primitive root of p.
- For a term x of degree $m < p-1$, since $\{1, 2, \cdots, p-1\} = \{q, 2q, \cdots, (p-1)q\}$, we should have $x = xq^m$ modulo p, but that means $x = 0$ $(q^m - 1 \neq 0 \bmod p)$
- By comparing the coefficients, we have $f(p, k, t) - kf(p, k, t-1) = 0$,so the answer is $(-k)^t \bmod p$

F: Fun on Tree

- Given a rooted tree with (possibly negative) weights on both vertices and edges. For each update, we will add v_i to y_i's subtree vertices. After each update, you need to output $\text{argmax}_j(d_{x_i,j} - a_j)$ and $\max_j(d_{x_i,j} - a_j)$. Here $d_{x_i,j}$ means the distance between x_i and j.

F: Fun on Tree

- Given a rooted tree with (possibly negative) weights on both vertices and edges. For each update, we will add v_i to y_i's subtree vertices. After each update, you need to output $\mathrm{argmax}_j(d_{x_i,j} - a_j)$ and $\max_j(d_{x_i,j} - a_j)$. Here $d_{x_i,j}$ means the distance between x_i and j.
- A classic problem. Many possible solutions exist. (e.g. TopTree, Centroid Decomposition, etc.)

F: Fun on Tree

- Given a rooted tree with (possibly negative) weights on both vertices and edges. For each update, we will add v_i to y_i's subtree vertices. After each update, you need to output $\text{argmax}_j(d_{x_i,j} - a_j)$ and $\max_j(d_{x_i,j} - a_j)$. Here $d_{x_i,j}$ means the distance between x_i and j.
- A classic problem. Many possible solutions exist. (e.g. TopTree, Centroid Decomposition, etc.)
- We will describe an algorithm that only requires Heavy-Light Decomposition and Segment Tree. This algorithm runs in $O(N \log N + Q \log^2 N)$ with $O(N)$ space used.

F: Fun on Tree

- Consider all possible LCA of x_i and j. It is clear that it must be either x_i or x_i's ancestor.

F: Fun on Tree

- Consider all possible LCA of x_i and j. It is clear that it must be either x_i or x_i's ancestor.
- Case 1: The LCA is x_i (i.e. j is in x_i's subtree). Then, the answer is $\min(d_{1,j} - a_j) - d_{1,x_i}$. Therefore, we can maintain a Segement Tree with $d_{1,j} - a_j$ inside. For each update and query, you simply do range-update and range-query on the given subtree range.

F: Fun on Tree

- Consider all possible LCA of x_i and j. It is clear that it must be either x_i or x_i's ancestor.
- Case 1: The LCA is x_i (i.e. j is in x_i's subtree). Then, the answer is $\min(d_{1,j} - a_j) - d_{1,x_i}$. Therefore, we can maintain a Segement Tree with $d_{1,j} - a_j$ inside. For each update and query, you simply do range-update and range-query on the given subtree range.
- Case 2: The LCA is x_i's ancestor. Then, the answer is $\min(d_{1,j} - a_j) - 2d_{1,g_{x_i,j}} - d_{1,x_i}$. Here, $g_{x_i,j}$ is the LCA of x_i and j. We also know that j must be in one of the subtrees of $g_{x_i,j}$ except the one x_i is in.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.
- Conduct HLD on the tree. For each node u, we store $\min(d_{1,j} - a_j) - 2d_{1,u}$ on it and build a Segment Tree for these chains. Here, j is in one of the subtrees of u except the one the heavy child of u is in.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.
- Conduct HLD on the tree. For each node u, we store $\min(d_{1,j} - a_j) - 2d_{1,u}$ on it and build a Segment Tree for these chains. Here, j is in one of the subtrees of u except the one the heavy child of u is in.
- For each query, climb the tree from x_i to 1.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.
- Conduct HLD on the tree. For each node u, we store $\min(d_{1,j} - a_j) - 2d_{1,u}$ on it and build a Segment Tree for these chains. Here, j is in one of the subtrees of u except the one the heavy child of u is in.
- For each query, climb the tree from x_i to 1.
  - If the edge is a heavy edge, the value we stored previously is the answer we want. Therefore, we can ask the Segment Tree to get the answer.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.
- Conduct HLD on the tree. For each node u, we store $\min(d_{1,j} - a_j) - 2d_{1,u}$ on it and build a Segment Tree for these chains. Here, j is in one of the subtrees of u except the one the heavy child of u is in.
- For each query, climb the tree from x_i to 1.
    - If the edge is a heavy edge, the value we stored previously is the answer we want. Therefore, we can ask the Segment Tree to get the answer.
    - If the edge is a light edge connecting p_o and o, we can use the Segment Tree in case 1 to answer the query.

F: Fun on Tree

- The answer of case 2 can be found in $O(\log^2 N)$.
- Conduct HLD on the tree. For each node u, we store $\min(d_{1,j} - a_j) - 2d_{1,u}$ on it and build a Segment Tree for these chains. Here, j is in one of the subtrees of u except the one the heavy child of u is in.
- For each query, climb the tree from x_i to 1.
  - If the edge is a heavy edge, the value we stored previously is the answer we want. Therefore, we can ask the Segment Tree to get the answer.
  - If the edge is a light edge connecting p_o and o, we can use the Segment Tree in case 1 to answer the query.
- Update to the Segment Tree in case 2 can also be done in a similar way. Since we only need to care about light edges for each update, we can reuse the Segment Tree in case 1 to update light edges.

G: Game

TODO

H: Harumachi Kaze

- From now on we only assume the largest case, $n \leq 16384$ and $k \leq 32767$

H: Harumachi Kaze

- From now on we only assume the largest case, $n \leq 16384$ and $k \leq 32767$
- Because of the conditions on the ks in the queries, we can construct $a_1, \cdots, a_8$ and $b_1, \cdots, b_8$ such that for all ks in the query, we can find i, j such that $a_i + b_j + k = 32767$.

H: Harumachi Kaze

- From now on we only assume the largest case, $n \leq 16384$ and $k \leq 32767$
- Because of the conditions on the ks in the queries, we can construct $a_1, \cdots, a_8$ and $b_1, \cdots, b_8$ such that for all ks in the query, we can find i, j such that $a_i + b_j + k = 32767$.
- For i from 1 to 8, we construct a segment tree of size 32768 for the array A, but with a_i zeros filled in the beginning. For B, we do the same. So we have 16 segment trees, and have to modify on 8 of them in every modification.

H: Harumachi Kaze

- From now on we only assume the largest case, $n \leq 16384$ and $k \leq 32767$
- Because of the conditions on the ks in the queries, we can construct $a_1, \cdots, a_8$ and $b_1, \cdots, b_8$ such that for all ks in the query, we can find i, j such that $a_i + b_j + k = 32767$.
- For i from 1 to 8, we construct a segment tree of size 32768 for the array A, but with a_i zeros filled in the beginning. For B, we do the same. So we have 16 segment trees, and have to modify on 8 of them in every modification.
- For a query, if we have $a_i + b_j + k = 32767$, then only operate on the two segment trees in this query (the ith version of A and the jth version of B).

H: Harumachi Kaze

- From now on we only assume the largest case, $n \leq 16384$ and $k \leq 32767$
- Because of the conditions on the ks in the queries, we can construct $a_1, \cdots, a_8$ and $b_1, \cdots, b_8$ such that for all ks in the query, we can find i, j such that $a_i + b_j + k = 32767$.
- For i from 1 to 8, we construct a segment tree of size 32768 for the array A, but with a_i zeros filled in the beginning. For B, we do the same. So we have 16 segment trees, and have to modify on 8 of them in every modification.
- For a query, if we have $a_i + b_j + k = 32767$, then only operate on the two segment trees in this query (the ith version of A and the jth version of B).
- Now it's equivalent to having $k = 32767 = 2^t - 1$, so we know **exactly one** of A', B' uses a prefix of length 2^{t-1}. We can determine which one to take in one compare query. Then we have $k = 2^{t-1} - 1$ and goes one level deeper in both the segment trees. Since the size of all nodes in the segment trees is some power of 2, we can continue this procedure and answer the query in O(log n) queries.

H: Harumachi Kaze

- To summerize, we use 16 segment trees, each needs to call *add* $n-1$ times in the preprocessing. For every modification, we use *add* $8\log(n)$ times.
  And for every query, we have $\log(n)$ levels to process and each costs 2 *add* queries and 1 *cmp* query. So the total number of queries is
  $8n\log(n) + 5000 \times 8\log(n) + 15000 \times 3\log(n)$

H: Harumachi Kaze

- To summerize, we use 16 segment trees, each needs to call *add* $n-1$ times in the preprocessing. For every modification, we use *add* $8\log(n)$ times.
  And for every query, we have $\log(n)$ levels to process and each costs 2 *add* queries and 1 *cmp* query. So the total number of queries is
  $8n\log(n) + 5000 \times 8\log(n) + 15000 \times 3\log(n)$
- This is the end of my clown fiesta solution.
  From the participants' solutions, we learned that it is actually possible to do binary search on two segment trees in $O(\log n)$ queries, where the form of the segment trees can be arbitrary, and the conditions on k are also not needed, rendering my algorithm useless. The key to that is to also do case analysis (whether the current k is too big or too small). This algorithm also supports binary searching on more than two trees. To my current understanding, the only advantage of my algorithm is it uses half the number of *cmp* queries, but is still worthless since we have to do much more preprocessing in order to make it work.

I: Interval Addition

- Consider the difference array. $b_i = a_i - a_{i-1}$ for $1 \leq i \leq n+1$, assuming $a_0 = a_{n+1} = 0$

I: Interval Addition

- Consider the difference array. $b_i = a_i - a_{i-1}$ for $1 \le i \le n+1$, assuming $a_0 = a_{n+1} = 0$
- Applying range addition to $[l, r]$ is equal to freely moving the value of b_l and b_{r+1} while preserving their sum.

I: Interval Addition

- Consider the difference array. $b_i = a_i - a_{i-1}$ for $1 \leq i \leq n+1$, assuming $a_0 = a_{n+1} = 0$
- Applying range addition to $[l, r]$ is equal to freely moving the value of b_l and b_{r+1} while preserving their sum.
- If we draw an edge between all such ls and $r+1$s, we can see that the sequence of operation can result in an all 0 sequence if and only if the sum of elements in every connected component is 0. If there are k of them, there is a way of using only $n-k$ operations.

I: Interval Addition

- Consider the difference array. $b_i = a_i - a_{i-1}$ for $1 \leq i \leq n+1$, assuming $a_0 = a_{n+1} = 0$
- Applying range addition to $[l, r]$ is equal to freely moving the value of b_l and b_{r+1} while preserving their sum.
- If we draw an edge between all such ls and $r+1$s, we can see that the sequence of operation can result in an all 0 sequence if and only if the sum of elements in every connected component is 0. If there are k of them, there is a way of using only $n-k$ operations.
- We can have the following bitmask dp for maximizing the number of components, $dp[i] = max(dp[j]) + [\text{sum of elements in } i \text{ is } 0]$, where j is a submask of i. In fact, only js that differ from i in one bit need to be considered, so the total complexity is $O(n2^n)$.

J: Joining Cats

- First observation is that we can always fix the position of one cat and let the other cats merge to it.

J: Joining Cats

- First observation is that we can always fix the position of one cat and let the other cats merge to it.
- So if we enumerate which of the cats it is, we can get a slow *dp* solution. Let $dp[i][j]$ be the maximum number of the rightward cats we can merge to the center if we've used the first i gust of winds and have merged j leftward cats to the center. Total transitions cost $O(n^2)$, so total complexity is $O(n^3)$

J: Joining Cats

- First observation is that we can always fix the position of one cat and let the other cats merge to it.
- So if we enumerate which of the cats it is, we can get a slow *dp* solution. Let $dp[i][j]$ be the maximum number of the rightward cats we can merge to the center if we've used the first i gust of winds and have merged j leftward cats to the center. Total transitions cost $O(n^2)$, so total complexity is $O(n^3)$
- If we view the problem in reverse order, we can have a similar *dp* but don't need to enumerate the cat in the center (just check if for some j, $dp[i][j] + j \geq n - 1$), so we get a $O(n^2)$ solution.

K: Keychain

- N different circles' center are given as $p_1, \ldots, p_N$ on 2D plane. Their radius are all r. Determine the minimum value of r such that we can draw a *generalized* circle Γ intersects with all N circle.
- The key idea is, if the center of the circle Γ is fixed to a certain point O, then optimal radius of Γ is $\frac{1}{2}(\max(d(p_i, O)) + \min(d(p_i, O)))$, and the corresponding minimized r would be $\frac{1}{2}(\max(d(p_i, O)) - \min(d(p_i, O)))$
- So we need to find a point O such that the difference between the distance from O to the farthest and nearest point among p_i is minimized.

K: Keychain

- Let's first enumerate all $\mathcal{O}(n^2)$ pairs of point as the farthest and nearest, say p_{far} and p_{near}.
- The region
$$\{O | d(p_{near}, O) \leq d(p_i, O) \forall i\}$$
can be described as the half-plane-intersection of $n-1$ bisectors. The farthest point case is similar.
- So, the region such that p_{far} is the farthest point and p_{near} is the nearest point is convex (but maybe unbounded). Let's call it C.

K: Keychain

- Claim: The minimum difference

$$\min_{O \in C}\{d(O, p_{far}) - d(O, p_{near})\}$$

  appears at vertices, or the limit to infinity through unbounded edge (infimium).
- Rough idea:
  - For a fixed D, the graph of $d(O, p_{far}) - d(O, p_{near}) = D$ is a branch of hyperbola. When $D = 0$ it's the bisector of p_{far} and p_{near}.
  - C completely lies in one side of the bisector (the side that is closer to p_{near}).
  - Imagine we increase D until the hyperbola touches C. Then either it exactly touches some vertices, or the asymptote is parallel to the unbounded edge.
- Unbounded case corresponds to the line case, and we can solve it by building convex hull and using rotating calipers algorithm.

K: Keychain

- Actually the vertices of C of each pair of p_{far} and p_{near} is the intersection of Voronoi Diagram and Farthest Point Voronoi Diagram.
- Each diagram is a partition of 2D plane. We can use Euler's planar graph formula to prove that they both have linear number of edges (about $6n \times 2$, though).
- We can enumerate pairs of cells in VD and FPVD, and for each vertex of their intersection, calculate the difference in $\mathcal{O}(1)$.
- Each cell is described by some half-planes, so if we take $\mathcal{O}(A + B)$ time to calculate the intersection of a cell with A edges and a cell with B edges, then the time complexity would be $\mathcal{O}(n^2)$ in total. This can be done by preprocessing sorted half-planes and merge them in linear time when doing half-plane-intersection instead of sort them directly.

K: Keychain — Building Diagram

- Both diagram can be calculated with half-plane-intersection in $\mathcal{O}(n^2 \log n)$.
- Here's the problem: can we get the two diagram in $o(n^2 \log n)$?
- 3D convex hull in $\mathcal{O}(n^2)$ might work.
- Nearest Point Voronoi Diagram in $\mathcal{O}(n \log n)$ is a well known geometry template.
- Sketch of building Farthest Point Voronoi Diagram in $\mathcal{O}(n^2)$: notice that only the points on the convex hull are useful in Farthest Point Voronoi Diagram. So it can be done by $\mathcal{O}(n)$ times half-plane-intersection of $\mathcal{O}(n)$ edges without sorting (the angle-sorted bisectors list can be obtained easily when we have the counterclockwise sorted convex hull list).
- Actually there is a linear time randomized incremental algorithm to do the farthest one [1].

K: Keychain — Implementation Details

- Be careful of degenerated case. $n = 1$, all points colinear, e.t.c.
- Common half-plane-intersection S&I implementation requires a boundary to ensure that it won't be unbounded. The unbounded case is done by rotating caliper, but we still need to decide the size of boundary so that all intersections are considered. The maximum coordinate of the intersection of two bisectors can reach $\Theta(C^2)$ where $C = 10^5$ in this problem. So we might need a 10^{10} boundary and one should implement it carefully to avoid overflow (or just use floating numbers instead?).

L: Lines

$$F_i(t) = it + \max_{\substack{0 \le x,y,z \le n \\ x+y+z=i}} (a_x + b_y + c_z)$$

$$\max_{0 \le i \le 3n} F_i(t) = \max_{0 \le i \le 3n} \max_{\substack{0 \le x,y,z \le n \\ x+y+z=i}} (a_x + b_y + c_z + tx + ty + tz)$$

$$\max_{0 \le i \le 3n} F_i(t) = \max_{0 \le x,y,z \le n} (a_x + b_y + c_z + tx + ty + tz)$$

$$\max_{0 \le i \le 3n} F_i(t) = \max_{0 \le x \le n} (a_x + tx) + \max_{0 \le y \le n} (b_y + ty) + \max_{0 \le z \le n} (c_z + tz)$$

So the graph of $\max_{0 \le i \le 3n} F_i(t)$ changes slope whenever the convex hull of A, B, C changes slope. We can construct the answer easily then by seeing the graph as the **sum** of the convex hulls.

References

[1] L. P Chew.
Building voronoi diagrams for convex polygons in linear expected time.
Technical report, USA, 1990.