Greetings from China Analysis

September 23, 2019

Mikhail Tikhomirov, Pavel Kunyavsky
Discover Singapore 2019

Moscow Workshops ICPC

DISCOVER Singapore 2019

Schaffhausen Institute of Technology

NUS National University of Singapore

Acronis

A. Array

Given an array, perform range operations:

- $x \to x + c$;
- $x \to \min(x, c)$;
- $x \to \max(x, c)$;
- find $\min x$ and $\max x$.

A. Array

We will use lazy propagation segment tree. The idea is:

- Store range minimums and maximums, as well as transformations $T_v(x)$ in each node v of the segment tree. If v is a leaf node, and $p_0, \ldots, p_k = v$ are all his ancestors starting from the root, then we assume that $T_{p_0}(T_{p_1}(\ldots(T_{p_k}(x))\ldots))$ should be applied to a_v.
- Whenever we have to go to children of a node v, replace $T_u(x)$ with $T_v(T_u(x))$ for each child u, and $T_v(x)$ with identity tranformation (push routine).

A. Array

How do we represent a transformation compactly? One can see that any combination of $x + c, \min(x, c), \max(x, c)$ can be represented as a single transformation $clamp_{l,r}(x) + c$, where $clamp_{l,r}(x) = \max(l, \min(r, x))$.

A. Array

How do we represent a transformation compactly? One can see that any combination of $x + c, \min(x, c), \max(x, c)$ can be represented as a single transformation $clamp_{l,r}(x) + c$, where $clamp_{l,r}(x) = \max(l, \min(r, x))$.

To combine transformations, observe that:

- $clamp_{l,r}(x + c) = clamp_{l-c,r-c}(x) + c$;

A. Array

How do we represent a transformation compactly? One can see that any combination of $x + c, \min(x, c), \max(x, c)$ can be represented as a single transformation $clamp_{l,r}(x) + c$, where $clamp_{l,r}(x) = \max(l, \min(r, x))$.

To combine transformations, observe that:

- $clamp_{l,r}(x + c) = clamp_{l-c,r-c}(x) + c$;
- $clamp_{l,r}(clamp_{l',r'}(x)) =$
$$\begin{cases} clamp_{l,l}(x) & \text{if } r' \leqslant l \\ clamp_{r,r}(x) & \text{if } r \leqslant l' \\ clamp_{\max(l,l'),\min(r,r')}(x) & \text{otherwise} \end{cases}$$

A. Array

When applying a range update $T(x)$, we have to change $T_v(x)$ to $T(T_v(x))$ for some of the nodes v and update their minimums/maximums. Since any T is monotonous, then we simply go $\min \to T(\min)$, $\max \to T(\max)$.

A. Array

When applying a range update $T(x)$, we have to change $T_v(x)$ to $T(T_v(x))$ for some of the nodes v and update their minimums/maximums. Since any T is monotonous, then we simply go $\min \to T(\min)$, $\max \to T(\max)$.

All of this is $\log n$ per query.

B. Chromatic Number

Given a graph with n vertices and at most $n+8$ edges count number of ways to color it in c colors.

While there is a vertex of degree 1, remove it, and multiply answer by $c-1$.

After that, there is only 16 verties have degree more than 2. Let's calculate number of ways to color path of length k such that it ends have same color, and have different colors.

Now problem is reduced to following: There is a graph on 16 vertices and 24 edges, for each edge it's known how many ways to color it, if vertices have same color and how many ways to color it, if vertices have diffrent colors. How many ways to color full graph?

Can be solved by dynamic programming on subsets in $O(3^16)$ time. Close to time limit, can pass, can not pass.

B. Chromatic Number

Let's choose dfs-tree for out graph, and calculate dp on it.

$dp_{v,S}$ is number of ways to color subtree of v, with parents of v colored as S.

$$dp_{v,S} = \prod_{u \in sons(v)} \left(\sum_{c} dp_{v,S+[c]} \cdot \prod_{t \in up(u)} ways(c, S[t], L(u,t)) \right)$$

This works in $O(c^{16})$ time. Need to merge equiavalent states.

First of all, exact values of colors doesn't matter, we can renumerate them from 1 to number of used colors. This leads to $B_{16} * 16 \approx 10^{11}$ solution, which is still too slow.

Next, if there is no upper edge to the vertex, it's color doesn't metter. So only at most 9 vertex is intersting, other can be removed. Which leads to $B_9 * 16$ solution, which is fast enough.

C. Nearest friend

Given a weighted graph with several marked vertices, for each marked vertex find the closest out of other marked vertices.

Let's find the closest marked vertex for each vertex. This can be done by running Dijkstra algorithm starting from all marked vertices.

We claim, that if a marked vertex b is closest to a marked vertex a, there exists an edge (u, v), such that a is the closest marked vertex to u, and b is the closest marked vertex to v. It allows to just check pairs for all edges.

Let's find the only edge (u, v) on a shortest path from a to b, for which $d(a, u) <= d(b, u)$ and $d(a, v) > d(b, v)$. If $c \neq a$ is closest to u, then $a \to u \to c$ is shorter than $a \to u \to v \to b$. If $d \neq b$ is closest to v, then $a \to u \to v \to d$ is not longer than $a \to u \to v \to b$.

D. Sequence Sorting

Given a lot of sequences sort them in lexicographic order.

Just put everything in trie, and do a dfs. To make memory usage lower, lists and count sorting can be used instead of vectors. Probably, a lot of other solution exists.

E. New Point

There are n distinct points in the plane. Add one more point distinct from all others so that to maximize the number of right-angled triangles with vertices in the points and legs parallel to the axes.

E. New Point

How to count the number of such triangles? Let $c_x(t)$ and $c_y(t)$ be the number of points with $x = t$ or $y = t$ respectively. Then the answer is $\sum_{(x,y) \text{ is in the set}} (c_x(x) - 1) \cdot (c_y(y) - 1)$.

E. New Point

How to count the number of such triangles? Let $c_x(t)$ and $c_y(t)$ be the number of points with $x = t$ or $y = t$ respectively. Then the answer is $\sum_{(x,y) \text{ is in the set}} (c_x(x) - 1) \cdot (c_y(y) - 1)$.

What is the effect of adding a new point (x, y) to the set S?

E. New Point

How to count the number of such triangles? Let $c_x(t)$ and $c_y(t)$ be the number of points with $x = t$ or $y = t$ respectively. Then the answer is $\sum_{(x,y) \text{ is in the set}} (c_x(x) - 1) \cdot (c_y(y) - 1)$.

What is the effect of adding a new point (x, y) to the set S?

- There are $(c_x(x) - 1) \cdot (c_y(y) - 1)$ new triangles where (x, y) is opposing the hypothenuse.

E. New Point

How to count the number of such triangles? Let $c_x(t)$ and $c_y(t)$ be the number of points with $x = t$ or $y = t$ respectively. Then the answer is $\sum_{(x,y) \text{ is in the set}} (c_x(x) - 1) \cdot (c_y(y) - 1)$.

What is the effect of adding a new point (x, y) to the set S?

- There are $(c_x(x) - 1) \cdot (c_y(y) - 1)$ new triangles where (x, y) is opposing the hypothenuse.
- There are $s_x(x) := \sum_{(x,y') \in S} (c_y(y') - 1)$ new triangles where (x, y) is opposing the horizontal leg.

E. New Point

How to count the number of such triangles? Let $c_x(t)$ and $c_y(t)$ be the number of points with $x = t$ or $y = t$ respectively. Then the answer is $\sum_{(x,y) \text{ is in the set}} (c_x(x) - 1) \cdot (c_y(y) - 1)$.

What is the effect of adding a new point (x, y) to the set S?

- There are $(c_x(x) - 1) \cdot (c_y(y) - 1)$ new triangles where (x, y) is opposing the hypothenuse.
- There are $s_x(x) := \sum_{(x,y') \in S} (c_y(y') - 1)$ new triangles where (x, y) is opposing the horizontal leg.
- There are $s_y(y) := \sum_{(x',y) \in S} (c_x(x') - 1)$ new triangles where (x, y) is opposing the vertical leg.

E. New Point

We'll start by compressing coordinates and computing all values of $c_x(x), c_y(y), s_x(x), s_y(y)$.

E. New Point

We'll start by compressing coordinates and computing all values of $c_x(x), c_y(y), s_x(x), s_y(y)$.

The problem now is: choose x, y such that $(c_x(x) - 1) \cdot (c_y(y) - 1) + s_x(x) + s_y(y)$ is largest, and (x, y) is not present in the input set.

E. New Point

We'll start by compressing coordinates and computing all values of $c_x(x), c_y(y), s_x(x), s_y(y)$.

The problem now is: choose x, y such that $(c_x(x) - 1) \cdot (c_y(y) - 1) + s_x(x) + s_y(y)$ is largest, and (x, y) is not present in the input set.

Let us group y's by their value of $c_y(y)$, and let $L(c)$ be the list of all y with $c_y(y) = c$, sorted by decreasing of $s_y(y)$.

E. New Point

We'll start by compressing coordinates and computing all values of $c_x(x), c_y(y), s_x(x), s_y(y)$.

The problem now is: choose x, y such that $(c_x(x) - 1) \cdot (c_y(y) - 1) + s_x(x) + s_y(y)$ is largest, and (x, y) is not present in the input set.

Let us group y's by their value of $c_y(y)$, and let $L(c)$ be the list of all y with $c_y(y) = c$, sorted by decreasing of $s_y(y)$.

Let us fix x and $c = c_y(y)$. To maximize the answer, we then have to choose the earliest $y \in L(c)$ such that (x, y) is not banned.

E. New Point

We'll start by compressing coordinates and computing all values of $c_x(x), c_y(y), s_x(x), s_y(y)$.

The problem now is: choose x, y such that $(c_x(x) - 1) \cdot (c_y(y) - 1) + s_x(x) + s_y(y)$ is largest, and (x, y) is not present in the input set.

Let us group y's by their value of $c_y(y)$, and let $L(c)$ be the list of all y with $c_y(y) = c$, sorted by decreasing of $s_y(y)$.

Let us fix x and $c = c_y(y)$. To maximize the answer, we then have to choose the earliest $y \in L(c)$ such that (x, y) is not banned.

We can do this as follows: as soon as we choose x, go over all points (x, y) and mark them as banned in their respective lists $L(c_y(y))$. To find the first non-banned point, use linear search in $L(c)$.

E. New Point

What is the complexity of this approach?

- We need $O(n \log n)$ time to compress coordinates, construct lists, etc.

E. New Point

What is the complexity of this approach?

- We need $O(n \log n)$ time to compress coordinates, construct lists, etc.
- The total number of linear search skips in all $L(c)$ does not exceed the number of banned points, i.e. $O(n)$.

E. New Point

What is the complexity of this approach?

- We need $O(n \log n)$ time to compress coordinates, construct lists, etc.
- The total number of linear search skips in all $L(c)$ does not exceed the number of banned points, i.e. $O(n)$.
- Since the sum of $c_y(y)$ over all distinct y is equal to n, the number of distinct values of $c_y(y)$ is $O(\sqrt{n})$. Thus, for each x there will be $O(\sqrt{n})$ of lists $L(c)$ to try, and the time we need for the last phase is $O(n\sqrt{n})$ (excl. complexity of linear search analysed above).

E. New Point

What is the complexity of this approach?

- We need $O(n \log n)$ time to compress coordinates, construct lists, etc.
- The total number of linear search skips in all $L(c)$ does not exceed the number of banned points, i.e. $O(n)$.
- Since the sum of $c_y(y)$ over all distinct y is equal to n, the number of distinct values of $c_y(y)$ is $O(\sqrt{n})$. Thus, for each x there will be $O(\sqrt{n})$ of lists $L(c)$ to try, and the time we need for the last phase is $O(n\sqrt{n})$ (excl. complexity of linear search analysed above).

The total complexity is now $O(n\sqrt{n})$.

F. Planar Graph Connectivity

There is a planar graph. Edges are removed one by one online. One need to answer queries on number of connected components and check if two verices are connected.

Let's not only remove edges, but add edges to dual graph. Connected component in graph is equivalent to a face in dual.

Let's maintain dsu in the dual graph. If an edge connects different components — nothing should be done. Otherwise it creates a cycle in the dual graph. That means, its endpoints in the original graph are not connected anymore. This allows to calculate number of components.

To answer connected queries, we need to store component id for each vertex. When an edge splits a component, we can run bfs from both verices in parallel, and stop whenever we visit one of the halves completely. It will work in $O(n \log n)$ total time, for the same reason as small-to-large merging.

G. Popo Sort

There is a loop swapping elementh in given positions in given order while at least one changes happens. Is it correct sorting?

If there exists i, such that $a_i > b_i$, sorting is not correct, because on the sorted sequence at least one change happens.

If there exists pos such that there is no pair $pos, pos + 1$, then sorting is not correct, because on this permutation with these two positions swapped it will do nothing.

Otherwise the sorting is correct, because number of inversions only decreases, and decreased by at least 1 on each step.

H. Power of Three

Given n bit vectors. Bits having pairwise distinct weights $\pm 3^k$, find a vector in a subspace of $\mathbb{Z}_2^n$ generated by these bit vectors with maximal total weight.

Let's transform our vector to 135-bit. $(3k)$-th bit is set if there is a bit with weight 3^k, $(3 \cdot k + 1)$-th bit is set if there is a bit with weight -3^k and $(3 \cdot k + 2)$-th bit is xor of two previous ones.

Then total weight 3^k for two bits is enforced by having $(3 \cdot k)$-th bit as 1, and $(3 \cdot k + 1)$-th bit as 0, while total weight 0 enforced by having $(3 \cdot k + 2)$-th bit as 0.

H. Power of Three

Exponential grouth of cost allows using greedy. Let's maximize total cost for power of three bits of bits one by one, starting from largest.

Let's do Gaussian elimination with smart pivot choosing. Previouse steps provide us some current vector value, and set of vectors without largest bits.

Let's say we can enforce value to a bit, if there is vector with this bit enabled or bit already has correct value. In first case, this vector should be made only one having this bit by making a round of elimination, to be sure nobody will change bit value in future.

H. Power of Three

To make total score for this power of 3 equal to 3^k we should enforce bit $3 \cdot k$ to value 1, and bit $3 \cdot k + 1$ to value 0. If both can be done — we can get positive score. Note that if only one bit can be forced, round of elimination for him should be reverted.

Next, we can try to enforce value 0 to bit $3\dot{k} + 2$ to achieve total score zero for this power of 3.

Otherwise, do nothing, and go to next step. To be done, we need to show, that future changes won't be able to change value for this power of 3.

H. Power of Three

If we can't enforce bit $3 \cdot k + 2$, than it's 0 in all of remaining vectors. That means, that bits $3 \cdot k$ and $3 \cdot k + 1$ are equal in all remaining vectors. So each of them change either none of them or both.

If current value is 0, we can't change it because of parity.

If current value is negative, no vector could change value of this bits, or we would be able to enforce postive value by adding vector changing both.

In both cases value for this degree is independent of future, so we can skip it.

I. Remove Obstacles

There is a grid of size $2 \times n$ with obstacles. One need to go from $(1, 1)$ to $(2, n)$, going up, down and right, without visiting same cell twice. It's possible to go, not entering obstacle cells. How many cells can there be on path if it's allowed to remove no more than m obstacles.

Obstacles splits grid to parts, with forced start and finish cells. Depend or parity, we can pass either to all cells in part, or to all except one. Note, that merging with a block of wrong parity, doen't change parity of other block.

If we can remove all obstacles — do it. Otherwise, pass through blocks, and remove obstacles after blocks of wrong parity. Each of removed obstacles give us one extra cell. If we have extra removals, we can earn one cell by removing two consecutive obstacles.

J. Salesmen

There is a tree. People in vertex i can buy w_i items. k salesmans can sell at most c_i items on path. How many items can be sold total?

We need to find max flow between salesmans and vertices. Only problem is naive approach will lead to quadratic number of edges.

Let's create an extra vertex for each vertical path of length 2^k. Edges of infinite capaticy leads from this vertex to two it subpathes of length 2^{k-1}. In particular, paths of length 1 is initial vertices.

Now we can add $O(\log n)$ vertices for each path.