Day 4 editorial, div. B

Ivan Smirnov
ifsmirnov@yandex.ru

September 29, 2018

A. Funky Numbers

There are only $O(\sqrt{n})$ triangular numbers less than n. Possible solution: put all of them into the set, bruteforce first summand x and check if $n - x$ is also triangular.

B. The Sum

Split the array into $\sqrt{n}$ blocks of size $\sqrt{n}$ and store the sum of elements in each block. For each query we bruteforce the elements in its boundary blocks and take the stored sum for indermediate blocks.

Pointwise update changes the sum only in one block.

C. Dot Product

In each block we store five values: sum_a, sum_b, inc_a, inc_b, res. That means that the sum of elements in the first array is sum_a and inc_a should be added to each of them. sum_b and inc_b have the same meaning for the second array. Finally, res is the value of the dot product of the elements in the block.

Whenever the sum or modification query intersects the block, we propagate the values and set $inc_a = inc_b = 0$.

To add x to the intermediate block we add it to inc_a or inc_b.

To get the dot product of the intermediate block, we take the value $res + sum_a \cdot inc_b + sum_b \cdot inc_a$.

D. Sum again...

Store two parts of the data structure: a sorted array and a "black box" with unsorted elements. When a new element is added, we put into the black box. Each $\sqrt{n}$ iterations we put all elements from the black box to the array and sort it.

To answer the query, do a binary search over the sorted array and bruteforce all elements from the black box.

E. K-th maximum

Several solutions exist. For example, one may use offline batching.

Once again, we store a black box and a sorted array. Each $\sqrt{n}$ iterations we rebuild the array, putting all elements that will participate in the next $\sqrt{n}$ queries to the black box. New elements are added to the black box.

To answer the query about k-th maximum, skip $\min(0, k - \sqrt{n})$ largest elements in the sorted array. Now the k-th maximum is either in the black box or in the next $\sqrt{n}$ elements of the array. Perform a merge of these two parts and return the element with the corresponding index.

F. Permutations

Split the array into $\sqrt{n}$ parts. In each part store the sorted array of the elements in that part.

Boundary blocks of the query are handled with bruteforce. For each intermediate block, two binary searches by k and l are necessary.

G. Permutations Strike Back

Each modification changes only one block, so we can insert the new element into the sorted array naively.

H. Important Guests

Given a bipartite graph, find the number of $K_{2,2}$ subgraphs (that is, cycles of length 4) in it. This problem is similar to counting triangles in a graph and is solved with the similar trick.

We consider only vertices in the left part. They may be small (with degree $< \sqrt{n}$) and large. There can be three kinds of pairs of vertices from the left part: small-small, large-small, large-large. Let us handle them separately.

Small-small For each small vertex v from left part, consider all pairs of its neighbors (x, y). For each (x, y) count the number of times this pair occured. Let it be k. That means that there are k small vertices in the left part that are adjacent to both x and y. It gives us $k \cdot (k-1)/2$ desired subgraphs.

Large-anything Fix some large vertex v. For every other vertex u, consider all its neighbors x such that x is the neighbor of v as well. If there are k such neighbors, it gives us $k \cdot (k-1)/2$ desired subgraphs. Be careful not to count large-large pairs twice.

I. Combinations Strike Back

- Let cnt_i be the number of elements equal to i.
- Let p_i be the number of cnt_j equal to i, i.e. the number of different elements with exactly i instances.
- The number of ways to pick exactly k elements from the set is equal to the coefficient a_k (between x^k) of the following polynomial:

$$P(x) = (1+x)^{p_1} \cdot (1+x+x^2)^{p_2} \cdot (1+x+x^2+x^3)^{p_3} \cdot \ldots \cdot (1+x+x^2+\ldots+x^n)^{p_n}$$

- Assuming that $\sum_{i=1}^{n} p_i \cdot i = n$, the above polynomial can be computed in $O(n \log^2 n)$ time using divide and conquer technique and Fast Fourier Transform algorithm.
- However, we also have to handle multiple queries and the polynomial for each of them may differ a bit. In particular, some value p_i is decreased by one and p_{i+1} increased by one.
- There are no more than $\sqrt{2n}$ different values of p_i not equal to 0. For each of them we can compute the above polynomial, thus the total complexity will be $O(n\sqrt{n} \log^2 n)$. Of course, this is still too slow.
- One can transform the solution to $O(n \log^2 n + n\sqrt{n})$ in a truly marvelous way, which this margin is too narrow to contain.

J. Lunch queue

Here we use the so-called dynamic sqrt-decomposition. In some of the first problems we split the array into blocks of size $\sqrt{n}$. Here we'll store the entire queue in a linked list of blocks of size around $\sqrt{n}$. For each block we know its size and the set of different teams of people in this block. When a new person is to be added, we go through the blocks until we find the block which contains his teammate and does not violate the impudence constraint. Then the newcomer may be inserted into this block in $O(block\_size)$ time.

The problem is that blocks may become too large. Here's how to fix it. If the block size after insertion becomes larger than $2\sqrt{n}$, we split this block in two smaller blocks naively. Thus all blocks will be of rather small ($< 2\sqrt{n}$) size.