Problem Tutorial: "The Best Problem of 2021"

Authors: original idea by 7dan, solution by 244mhq and Um_nik

TBD

Problem Tutorial: "Random Interactive Convex Hull Bot"

Author: ~~RICH B~~ 998batrr

Since we don't have access to point coordinates, it should be hard to use standard algorithms that rely on finding the leftmost point. Also, sorting the points should require more queries. Thus we need a different approach.

Let's insert points one by one and maintain the current convex hull. If the point is inside the current convex hull, we shouldn't change anything, otherwise, we should insert it in the right place and then remove all the points that are no longer on the convex hull.

We can interpret our query as the question "In which halfplane relative to line (P_iP_j) point P_k lies?" We want to check whether the new point lies in the same halfplane as the rest of the polygon relative to all the lines containing the sides of the polygon, and if not, find the bad side. Linear search uses too many queries, but since it is an interactive problem, we should use binary search instead.

Let's choose a diagonal of our polygon and ask in which halfplane the new point lies. Then we can remove all the sides of the polygon that are in the other halfplane. If we choose a diagonal that splits the polygon in half, we will remove almost half the points. Thus in $\log_2 H$ queries (where H is the current size of the hull) we can reduce the problem to the triangle. In 3 more queries we can understand whether our new point lies inside the triangle.

If it is inside, we are done. If it is outside, we now understand where to insert it, and then we can remove the points near it while it is not convex.

Let's estimate the number of queries better. Let's say we will have to insert a new point k times. Then we will remove points at most k times, thus we will spend at most $3k$ queries after understanding that we are outside the triangle. In the triangle part, we don't have to query the sides that are diagonals of the original polygon, because we have already queried them. That would reduce the number of queries to $\max(3, 2 + \log_2(H - 2))$. With some tweaks, you can reduce it to $\max(3, 1 + \log_2(H - 1))$, but it is not strictly required to get AC.

Thus the total number of queries is bounded by $2n+3k+\sum_{i=3}^{n} \log_2(H_i-1)$. It is known that the expected size of the convex hull of random m points chosen from a square is $O(\log m)$, and since the prefix is a (more or less) random set, this sum should be $O(n \log \log n)$. The probability that we have to rebuild the hull after adding i-th point is roughly $\frac{H_i}{i}$, thus k is $O(\log^2 n)$. This fits into the query limit, our different implementations use between 24 000 and 27 000 queries.

Problem Tutorial: "Record Parity"

Author: Um_nik

Let's look at two consecutive elements x and y such that $x > y$. Consider all subsequences that contain x. y is not a record in them, therefore, if we split all the subsequences containing x into pairs that differ only in y, in each such pair the subsequences have the same number of records, but different parities. Therefore, each pair contributes 0 to the sum, so we can remove element x from the permutation altogether and the answer will not change.

Repeat that while you can. In the end, you will get an increasing sequence. Actually, the element will remain in the sequence if and only if it is smaller than everything to the right of it. In this sequence, every subsequence will have all its elements as records, so we are interested in subsequences of k elements. Then it is easy to see that the answer is $\binom{m}{k}(-1)^k$ if m is the number of elements remaining in the sequence.

Problem Tutorial: "XOR Determinant"

Author: `Um_nik` ~~stole from adamant~~

Let's look at the binary representation of each element of A. The binary representation explains how to write this element as the sum of powers of 2. Let's say that $A_{ij} = \sum_{k=0}^{K-1} 2^k A_{ij}^{(k)}$. Then we can define K 01-matrices $A^{(k)}$ so that $A = \sum_{k=0}^{K} 2^k A^{(k)}$.

But each of $A^{(k)}$ is defined only by k-th bits of b and c, and each row is either $c^{(k)}$ or $c^{(k)} \oplus \mathbf{1}_n = \mathbf{1}_n - c^{(k)}$. That means that each row of A can be written as a linear combination of $c^{(0)}, c^{(1)}, \ldots, c^{(K-1)}$ and $\mathbf{1}_n$. Therefore, $\mathbf{rank}(A) \le K+1$, and if $n > K+1$, $\mathbf{det}(A) = 0$. Otherwise, we can calculate the determinant in $O(n^3)$ trivially.

Problem Tutorial: "Egor Has a Problem"

Author: `244mhq`

First, you should note that the division in the checker is in `long long`, so it rounds the result down (takes floor). Then it should be clear, that if we have a lot of elements, we cannot double the value each time, so there will be two pairs with the floor of quotient equal to 1. Precisely, if we have at least $\log_2 C + 3$ elements, we will find 2 pairs with the floor of quotient equal to 1. Otherwise, we can just try all $\binom{n}{4}$ options.

Problem Tutorial: "Is This FFT?"

Authors: original idea and $O(n^4 \log n)$ solution by `Um_nik`, $O(n^4)$ solution by `bicsi`

TBD

Problem Tutorial: "MIT"

Authors: original idea and solution for non-weighted tree by `998batrr`, solution for weighted tree by `998batrr` and `Um_nik`

TBD

Problem Tutorial: "Exact Subsequences"

Author: `998batrr`

Let's consider how we would calculate the number of different subsequences for a binary string given in the way described in the output section. Say that before a group of t zeroes, there are x different subsequences ending not in 1 (ending in 0 or empty), and y different subsequences ending not in 0. It is easy to show that $x \le y$. After adding 1 zero, the number of different subsequences ending not in 1 will become equal to $x + y$. After adding 2 zeroes — $x + 2y$. After adding t zeroes — $x + ty$.

Now we can notice that this process in reverse is the Euclid algorithm. Instead of calculating the number of different subsequences left-to-right, we can do the same right-to-left, then doing the Euclid algorithm will write down groups left-to-right.

Empty string has 1 (empty) subsequence, thus its pair (x, y) is $(1, 1)$. Therefore, we should get to $(1, 1)$ at the end of the Euclid algorithm. That will happen if and only if we start with a pair (x, y) with $\texttt{gcd}(x, y) = 1$. At the beginning of the Euclid algorithm, our string should have n different non-empty subsequences, so $x + y = n + 2$. These two conditions are necessary and sufficient. $\texttt{gcd}(n + 2, y) = \texttt{gcd}(n + 2 - y, y) = \texttt{gcd}(x, y) = 1$. This, for example, means that the number of these strings is $\varphi(n + 2)$. But we still need to understand how to get k-th of them.

Let's call the string we get from applying the Euclid algorithm to (x, y): $S(x, y)$ (we are assuming $\texttt{gcd}(x, y) = 1$). $S(1, 1) = \varepsilon$, for $x > y$: $S(x, y) =$"0"$+S(x - y, y)$, for $x < y$ $S(x, y) =$"1"$+S(x, y - x)$.

Now I want to prove that if $x_1 > x_2$ and $y_1 < y_2$, then $S(x_1, y_1) < S(x_2, y_2)$. Proof by induction on $x_1 + x_2 + y_1 + y_2$. If their first characters are different, then $S(x_1, y_1)$ starts from 0 and $S(x_2, y_2)$ starts

from 1. Otherwise they either both start with 0, or both start with 1, these cases are similar, let's assume they both start with 0. Then $S(x_1, y_1) =$"0"$+S(x_1-y_1, y_1)$, $S(x_2, y_2) =$"0"$+S(x_2-y_2, y_2)$. $x_1-y_1 > x_2-y_2$, so we can apply the induction hypothesis to suffixes.

Since all our interesting pairs have the same sum, we can just sort them in increasing order of y. Thus we have reduced the problem to finding k-th number coprime with $n+2$.

To do this, we will factorize $(n+2)$ ($O(\sqrt{n})$ is enough), then apply binary search on answer, and use inclusion-exclusion on prime divisors to calculate the number of numbers on a given prefix that are coprime with the given number.

Problem Tutorial: "SPPPSPSS."

Author: `998batrr`

It's never bad to sort some subsegment.

Let's take an optimal answer. If it has two consecutive operations of the same type, the first of them is useless, so we can switch it to the other type without making the answer worse. Then if we switch to the rightmost such pair, we will surely move the rightmost pair to the left (or remove all of them altogether). Thus we can eliminate all such pairs. This proves that there is an optimal answer with alternating operations. There are only two such sequences, let's try both of them.

We can implement the sort in $O(n)$ (counting sort). While the prefixes and suffixes we are sorting are not intersecting, they don't affect each other, thus we can only sort the largest prefix and largest suffix. Thus if the answer is at most $n/2$, we can do the binary search on the answer, checking in $O(n)$.

If the answer is greater than $n/2$, we can do the first two intersecting sorts, after that our sequence is always a concatenation of two increasing sequences. Moreover, we can maintain the current sequence in the following form: `[correct prefix] [sorted segment from the first half] [sorted segment from the second half] [correct suffix]`. Each time we sort some prefix, we will cut a prefix from the third part, and merge it into the second part, but then all those values will actually become correct, so they will enlarge the correct prefix. So the operation will result in cutting off prefixes from the second and the third parts. The same is true about sorting the suffix and cutting suffixes from two segments. Both operations can be implemented by moving left or right borders of the segments, and each time we move the border, the size of the segment decreases by one. Therefore, this process requires $O(n)$ operations.

Another way to solve the second part is to understand that there will be at most $\sqrt{n}$ operations after $n/2$, and we can implement them in $O(n\sqrt{n})$. Depending on the constant factor, it might pass.

Problem Tutorial: "Kth Lex Min Min Min Subpalindromes"

Authors: `353cerega`, `244mhq` and `998batrr`

$m = 1$ is trivial.

For $m \geq 3$ we can make each character to be not equal to the two previous characters, thus not creating any palindromes of length greater than 1. Then independently of the previously made choices, there are m options for the first position, $(m-1)$ options for the second, and $(m-2)$ for all other positions. From here it should be easy to find k-th of them.

For $m = 2$ you can write bruteforce and see that for $n \geq 10$ there are exactly 12 valid strings, each of them having period 6.

Problem Tutorial: "4"

Author: `Um_nik`

Let's sort the vertices in increasing order of their degree. Now for each vertex let's only consider the edges to the vertices on the right. Let's call the number of these edges from i-th vertex d_i. Obviously, $d_i \leq deg_i$, $\sum_{i=1}^{n} d_i = m$, $\sum_{i=1}^{n} deg_i = 2m$.

For vertices with $deg_i \le \sqrt{2m}$, clearly $d_i \le \sqrt{2m}$. There can only be at most $\sqrt{2m}$ vertices with $deg_i > \sqrt{2m}$, and they will be in the end of the list sorted by degree, thus $d_i < \sqrt{2m}$.

Let's fix the first vertex u in a possible K_4. Then in $O(d_i(d_i + \sqrt{2m})) = O(d_i\sqrt{m})$ we can find all the triangles (i.e. K_3) from this vertex using bruteforce. Let's now build another graph, only on neighbours of the fixed vertex that are on the right, where edge (xy) means existing of triangle (uxy) in the original graph. Now K_4 $(uxyz)$ in the original graph will be corresponding to the triangle (xyz) in the new small graph.

So we want to calculate the number of triangles in this small graph. If it has n_u vertices and m_u edges, this is easily done in $O(\frac{m_u n_u}{\omega})$ using bitsets. n_u is equal to d_u, and m_u is the number of triangles starting from u.

$\sum_u \frac{m_u n_u}{\omega} \le \frac{\sqrt{2m}}{\omega} \sum_u m_u$

$\sum_u m_u = O(m\sqrt{m})$, because it is the number of triangles in the graph.

Thus this solution works in $O(\frac{m^2}{\omega})$.

Problem Tutorial: "5"

Author: `244mhq`

TBD