Day 2 Contest: GP of NorthBeach

Moscow International Workshop 2020

November 29, 2020

A. Arrange and Count!

Keywords: string algorithms, hashes.

There are only $2n$ possible resulting sequences. Use polynomial hashes to count how many of them are unique. Take enough independent hashes to avoid collisions.

A hash-free solution that analyses the string period/palindromicity is also possible (?).

Complexity: $O(n \log n)$, or $O(n)$ (?).

B. Build More 2020's!

Keywords: binary search, greedy.

Forget the 1's.

Binary search on the answer k.

With enough exchange arguments, we can show that if an answer with k subsequences $s_1, \ldots, s_k$ exists, then s_i start at the k first 2's in order, and finish at the k last 0's in order (both from left to right).

Initially assign all s_i to only contain the first 2.

To assign remaining characters, go left to right, and try to append each character to the earliest possible s_i, until $s_i = 202$. If in the end any of s_i are unfinished, or the characters are not in order (say, the last 2 in s_i goes after the last 0), then the answer k is not achievable.

Complexity: $O(n \log n)$.

C. Choose Two Subsequences

Keywords: DP.

For all i, j compute $lcs(i, j)$ — the longest common subsequence of $s_0 \ldots s_{i-1}$ and $t_0 \ldots t_{j-1}$. This a standard DP application.

Let x and y be the chosen subsequences. Suppose that x is a prefix of y, x ends before s_i, and the equal prefix of y ends before t_j. In this case update the answer with $2lcs(i, j) + (|t| - j)$, since we are free to take the entire suffix $t_j \ldots$.

On the other hand, if $s_i < t_j$ is the first mismatch between x and y, update the answer with $2lcs(i, j) + (|s| - i) + (|t| - j)$ (take everything after s_i and t_j).

Complexity: $O(n^2)$.

D. Determinant Strikes Back

Keywords: linear algebra.

Given two n-vectors a, b and a number x, we have to compute $\det(ab^T + xI)$, where I is the identity matrix.

Either use Matrix determinant lemma, or the fact that $\det A$ is equal to the product of A's eigenvalues.

ab^T has rank at most 1, thus $n - 1$ of its eigenvalues are 0. The remaining eigenvalue is $a^T b$ as shown be the eigenvector relation $(ab^T)a = (a^T b) \cdot a$ (still true if $a = 0$).

Adding xI to any matrix increases all its eigenvalues by x, thus the answer is $(x + a^T b)x^{n-1}$.

Complexity: $O(n)$.

E. Efficient Data Structue

Keywords: data structures.

One can see that $c_i = \max_{j \leqslant i}(a_j + b_{j+1} + \ldots + b_i)$.
If $sb_i = b_1 + \ldots + b_i$, then $c_i = \max_{j \leqslant i}(a_j + sb_i - sb_j) = sb_i + \max_{j \leqslant i}(a_j - sb_j)$.
Denote $d_j = a_j - sb_j$. Maintain segment trees for sequences sb_i and d_j.
Answering a query = a single element sb_i + prefix maximum of d_j.
Updating a_i = updating a single d_i.
Updating b_i = range additions to d_i and sb_i.
If lazy propagation is used, the **complexity** is $O(\log n)$ per query.

F. Fibonacci Suffix Array

Keywords: suffix structures, patterns.

No full explanation, just some directions.
Compressed suffix automaton of fib_n is regular, and can be obtained from the ones of fib_{n-1} and fib_{n-2} with minor tweaks.
If we have the automaton, then finding the x-th suffix lexicographically is standard.
Since n is large, this can not be done explicitly, but only $\log \max X$ steps of the process are needed.

G. Greatest Square

Keywords: sweep-line, sweep-line, sweep-line, sweep-line, sweep-line.

Consider a query point $p = (x, y)$. Consider the 45° angle to the right of p between the horizontal and the diagonal lines. Any $q = (x', y')$ of the polygon boundary P in this angle limits the largest square size $s \leqslant x' - x$.
A similar constraint applies for y-coordinates of points of P in the angle between vertical and diagonal lines.
Constraints in both angles can be found independently, and similarly after swapping x and y.

Let a, b be the first points of P along the directions $(1, 0), (1, 1)$ as seen from p. Then it suffices to consider the part of P between a and b.
If we find these collision points for all queries, the problem reduces to RMQ for the round-trip of P.
To find horizontal collision points (a) for all queries, we can use right-to-left sweep-line, with range updates for vertical sides of P.
We can look for diagonal collisions with vertical and horizontal sides independently. Say, for vertical sides, apply the coordinate transformation $(x, y) \to (x, y - x)$. Sides stay vertical, but $(1, 1) \to (1, 0)$, thus the same method can be applied. Similar for horizontal sides.
Coordinate compression/treap is needed on every step.

Ultimately, **complexity** is $O((n + q) \log n)$.

H. Hamming Distance

Keywords: combinatorices, divide-and-conquerish.

Let $S^i + j$ be the sequence S^i with all elements increased by j. Observe that $S^{i+1} + j$ can be obtained from $S^i + (j + 1)$ by inserting $j + 1$ in all odd positions.
Let us find the smallest Hamming distance of a sequence s to a substring of $S^i + j$. If s consists of the only element x, then the smallest distance is 0 if $j + 1 \leqslant x \leqslant j + i$, and 1 otherwise.
If s has more than one element, split s into a sequence of even and odd positions s_0, s_1. One of these is matched with $j + 1$'s, and the other with elements of $S^{i-1} + (j + 1)$. Try both options recursively.
The depth of this "divide-and-conquer" is $O(\log n)$, and the **complexity** is $O(n \log n)$.

To find the total Hamming distance, count the number of matches for each element s_i against all substrings.
In general, s_i occurs 2^{m-s_i} times in S^m. We only need to subtract the occurences for illegal shifts, that is, against the first $i - 1$ and last $n - i$ elements of S^m (these can be treated symmetrically).
One can see that the number of occurences of x among the first y characters of S^m is $\lceil y/2^x \rceil$.
Thus the second part is doable with linear **complexity**.

I. Integers and Ranges

Keywords: DP.

Suppose that the string of digits is prepended with $d_0 = 0$.
Place digits from left to right. Let (i, j, k) be a state such that:

- digits $d_0, \ldots, d_i$ have been placed;
- the smallest suffix product divisible by 3 is $d_j \times \ldots \times d_i$;
- the smallest suffix product divisible by 9 is $d_k \times \ldots \times d_i$.

Ranges in the input provide lower bounds lb_i such that states (i, j, k) with $k < lb_i$ are illegal.

Now, consider all transitions:

- $(i, j, k) \to (i + 1, j, k)$ ($d_{i+1} = 1, 2, 4, 5, 7, 8$; 6 ways);
- $(i, j, k) \to (i + 1, i + 1, j)$ ($d_{i+1} = 3, 6$; 2 ways);
- $(i, j, k) \to (i + 1, i + 1, i + 1)$ ($d_{i+1} = 0, 9$; 2 ways).

Divide all coefficients by 6, and multiply by 6^n in the end.
Now, for the current i maintain a matrix $dp_{j,k}$ of the number of ways to reach (i, j, k).
Consider moving $i \to i + 1$. Then, transitions of the first type don't do anything.
Transitions of the second type put $dp_{i+1,j}$ to be the sum in the column j, divided by 3.
The final transitions put $dp_{i+1,i+1}$ to be the sum of all elements, divided by 3.
Additionally, we have to zero out all elements with $k < lb_{i+1}$.

Let us maintain column and total sums as we update the elements. Then, the row $i + 1$ can be filled in $O(n)$ time.
If an element is zeroed out due to $k < lb_i$ constraint, it will remain zero. Thus, by moving a pointer p to the last zeroed-out column, we can ensure that each element is zeroed out at most once.

The total **complexity** is $O(n^2)$.

J. Jailing

Keywords: prefix sums, inclusion-exclusion, case work.

For each bounding box B_y, let's try to count sum of bounding boxes B_y contained in B_x, and containing B_x.
Choose a representative of each connected component arbitrarily, and find the sum of representatives inside B_x. If $B_y \subseteq B_x$, we must have accounted for y.
If y is illegitimately counted, then y must occur outside of B_x, next to the boundary. Traverse the boundary, and fix these miscalculations.
The time to process B_x this way is proportinal to its perimeter $P(B_x)$. We can see that $P(B_x) \leqslant cnt(x)$, where $cnt(x)$ is the number of occurences of x in the matrix. Summing over all x, we can see that the time this step takes is $O(nm)$.

In $B_x \subseteq B_y$, then the top-left corner of B_y is in the top-left part with respect to the top-left corner of B_x, and so on.
Find the sum of all such corners in all four regions. Each bounding box is counted $0, 1, 2$, or 4 times.
With enough inclusion-exclusion and similar 2D prefix summing in corners we can rule out the cases of $0, 1, 2$.
This part also takes $O(nm)$.

With enough luck, sketchy solutions in $O(nm \min(n, m))$ time could also pass.