Moscow Pre-Finals ACM ICPC Workshop 2018
Day 5, Division A, Contest Analys, Tuesday, March 27, 2018

A. Two Plates

Let’s find *O(n)* candidate pairs using sweeping line. Some of them might not be OK, but all valid pairs will definitely be there. The only thing left is to check which of the pairs are actually OK. To do that, lets find for each rectangle the total area of intersection with all other rectangles. If for the two elements of pair the area is equal to the area of their intersection, the pair is OK.

To find the total area of intersections for all rectangles, you can use sweeping line and segment tree. For that you need to split all rectangles in stripes with left side in minus infinity, and during sweeping line keep track of currently opened and closed rectangles in separate segment trees.

B. Mushrooms

Just compute the binary representations using greedy, like usual but with missing bits skipped. Then the answer is the resulting binary number in decimal.

C. Paint the Tree

Let’s use binary search to find the answer. So we want to find the number of ways to paint the tree such that the distance between each pair of vertices with same color is at least *x*.

Let’s process all vertices in BFS order from some root vertex. Let *v* be current vertex and S_v be the set of all already processed vertices *u* such that $dist(v,\ u) < x$. Let’s prove that for any two vertices $a, b \in S_v$: $dist(a, b) < x$. Denote $c_1 = lca(v, a),\ c_2 = lca(v, b)$ and consider the case when $dist(root, c_1) \le dist(root, c_2)$. Because it’s BFS order you can see that $dist(root, b) \le dist(root, v)$ and $dist(c_2, b) \le dist(c_2, v)$. From that we get:

$$dist(a, b) = dist(a, c_2) + dist(c_2, b) \le dist(a, c_2) + dist(c_2, v) = dist(a, v) < x .$$

This means that all vertices in S_v are colored in different colors. So the number of ways to color vertex v is $K - |S_v|$. Using this informations it’s easy to find the number of ways to paint the tree in a valid way.

D. Balls

Let’s count in how many ways some color *c* appears in the final coloring. Let’s use dynamic programming for that: *dp[step][count]* - number of ways such that after *step* steps *count* balls have color *c.*

All the colors are identical, so answer is $n * \sum_{i=1}^{n} dp[k][i]$.

E. Boring Days

Let's iterate over the size of the prefix which the resulting string and *t* will have in common. Also fix the next character, which should be smaller than the one in *t*. By finding the first valid occurrence for that character, you can calculate the number of characters you need to remove. Note that there’s the case when *s* is a prefix of *t*, which can be handled in a similar way.

F. Catch Them All!

Let’s iterate over all points and let’s say the current one (x_i) is the first that’s inside the intersection. First of all, we now have to consider only those segments that contain the point x_i. All of them can be split into two types - those that start before x_{i-1} and after. Two make sure x_i is the first point inside the intersection, there should be at least one segment of the second type. Using some simple combinatorics you can calculate the number of ways to do that.

G. Game with Strings

Let's just find the lex first subsequence in a greedy way. This way, all segments are OK except maybe the last one. If it not OK, let’s try and fix it by moving the last selected character to the right to the first position that fixes it. After that, the segment before it might become not OK, and it should be fixed in the same way. So continue the process, and if at the end the first segment is not OK, than the answer does not exist, otherwise you found it.

H. Things

Let’s use the following observation: for any set of segments one can select a subset of the segment with the same coverage and only consecutive segments intersecting. One way to do it is using greedy from the left to the right, each time selecting the segment that has greatest right point with the left point not messing the coverage.

Let’s split each segment in two halfs and treat them as separate segments, but keep track of the initial indices. Now we can find some subset of segments which would cover all initial points and no two of them belonging to the same initial segment. Note that using our observation above, two segment can have the same initial index only if they are adjacent. We can use that information to construct a dynamic programming with state *[current segment][previous segment]* over the list of all segments sorted by the left point. In the transition you have to add the next segment and make sure of the following:

1. The selected segments cover all initial points.
2. Only consecutive segments intersect or touch.
3. No two consecutive segments have the same initial index.

I. More Things to Do

Denote x_i be all beginnings and endings of segments sorted from left to right. If there are several coinciding points any possible order is chosen. Let c_k be the number of segments that cover $[x_k, x_{k+1}]$, $c_0 = 0$.

Let's construct such coloring of segments that there are exactly $[c_i/2]$ segments of one color between each pair of neighboring points. Such coloring achieves the maximum possible value of target function.

You can see that $c_i = c_{i-1} \pm 1$, so values c_i are alternately odd and even. When c_i is even then there should be exactly $c_i/2$ segments of each color. As c_{i+1} is either $c_i + 1$ or $c_i - 1$ there definitely are $(c_{i+1} - 1)/2$ segments of one of the colors. So we shouldn't care about this values. That means that we should care only about parts with odd c_i. That means that if both x_1 and x_2 are of the same type (beginnings or endings) corresponding segments should have different colors, and same colors otherwise. Same for x_3 and x_4, … , x_{2n-1} and x_{2n}. Let's build a graph with edges between two segments if they should have same colors (type 1 edges) or different colors (type 2 edges). Each vertex has degree 2 so this graph consists of several cycles.

Let's look at all edges of type 2 on some cycle in order they lie on cycle. Each such edge corresponds to 2 beginnings or 2 endings. Such types (beginnings of endings) should alternate. That means that there are even number of edges of type 2 on each cycle and it's possible to color vertices in 2 colors.

J. Quicksort

Let pos_i be the position of number i in the permutation. Let's divide all elements into groups in the following way. Numbers *L..R* lie in one group iff $pos_L < pos_{L+1} < ... < pos_R$ and we can't extend it to the left or right. Let *cnt* be the number of such groups.

Let *x* and *y* (*x* < *y*) be pair of element lying in different groups. Let's prove that the sequences of operations applied to this numbers should differ. Proof by contradiction. If $pos_x > pos_y$ then elements won't change their relative order, so we won't get a sorted permutation. If $pos_x < pos_y$ then there exists such element *z* that $x < z < y$ and $(pos_z < pos_x \text{ or } pos_z > pos_y)$. In such case it's impossible to place *z* in correct place. That means that pair of elements from different groups should differ in at least one operation. So the answer is at least $ceil(log_2 cnt)$.

To build answer with exactly that number of operations let select all groups with odd indices (group that contains 1 is group number 1, the next one has number 2 and so on). After such operation group 1 will merge with group 2, 3 with 4, etc. The new number of groups will be $ceil(cnt/2)$. So after $ceil(log_2 cnt)$ operation the number of groups becomes equal 1.

K. Password

The length of desired number is *ceil(n/k)*. Now we can add digits one by one from the most significant. At each step we have several numbers which are equal to current maximum. We can add smallest digits to such numbers and largest digits to all other numbers.

L. Triangle

It's possible to construct triangle from any triple of sticks iff sum of two minimum sticks is greater than maximum stick. This means that it's always better to break sticks in such way that smallest sticks are as large as possible and largest sticks are as small as possible. So if some stick with length L is broken into K parts that every part should have length $\frac{L}{K}$.

Denote $x = min(L_i)$. We can break all sticks such that length of each stick is in range *[x, 2x)*. This means that no stick should be broken into parts with length smaller than *x*, because breaking into such parts requires more operations and doesn't bring any profit. So we can fix smallest stick and exclude it from other sticks. Now our task is to find such maximum c that for each L_i exists k such that $\frac{L_i}{k} \in [c, c+x)$. We need to maximize *c* in order to minimize number of breaks needed. Instead of that let's find bad values of *c* for each L_i, such values that none of numbers L_i/k lie in $[c, c+x)$.. There are such intervals of values, first of them is $(L_i, +\infty)$ and the next ones $(\frac{L_i}{k+1}, \frac{L_i}{k} - x]$. The length of such intervals are $\frac{L_i}{k} - x - \frac{L_i}{k+1} = \frac{L_i}{k(k+1)} - x$. $x > 1$ so if $k > \sqrt{L_i}$ then such length is less than 0 and we don't need to consider them. We can generate all such intervals sort them and find maximum point c that doesn't lie in any of them. The overall complexity is $O(n\sqrt{L_i}log(n\sqrt{L_i}))$.

M. Lucky Numbers

If k ≥ 9 one can show that any number can be represented as a sum of lucky numbers, except some small numbers, which cannot be represented at all.

We have to find the number of positive integers smaller than *X* that can be represented as a sum of at most *K ≤ 8* lucky numbers. We will be constructing lucky numbers from lower digits to higher. For each digit place we are only interested in the following: the number of lucky numbers that have this digit place, the comparison with *X* (smaller, equal or bigger), the number of fours, the number of sevens and the carry from the previous digit place. Unfortunately, with this kind of DP state some numbers could be counted multiple time, so we have to improve that part.

Let's modify our state in the following way. The first two states are gonna be the same. The third one is a bit more complicated. We want to somehow categorize all the numbers that we could have already created so that no number is counted twice or more. The following parameter works just fine.

For each carry that can be carried from the current digit place to the next one calculate the maximum number of lucky numbers that can sum up to such a number with such a carry to the next digit place. This vector of numbers (of size ≤ *7* as carry is in range *[0..6]*) describes some class of numbers with given length and each number is present in exactly one such class. Having a class we can just try all possible carries and combinations of fours and sevens in current digit place and hence obtain transitions between states in the DP. Each carry and combination of number of 4s and 7s produces some final digit in the current position, and we should group our transition by this digit. All other parameters are the same in a single group, hence the state we go to is also the same for all numbers from the group.