15295 Fall 2018 #10 Problem Discussion
October 31, 2018

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

The theme of this week's contest was disjoint sets, aka Union-Find.

A. New Year Permutation
"Beautiful" is equivalent to arranging the numbers in ascending order (as much as possible). So we can union all the connected positions, gather all the numbers in the component, and rearrange numbers into positions in ascending order.
----Yucheng Dai

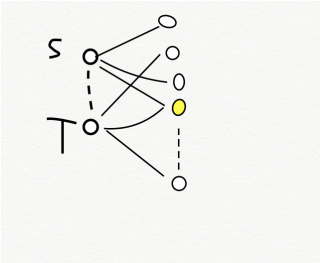
B. Correct Bracket Sequence Editor

C. Roads not only in Berland
This problem asks you to list out all of the edges needed to replace to connect a graph given as input. To do this, first initialize a union find over n vertices on the graph to be connected. For each edge, if the two vertices were not connected before, union them, otherwise store the edge for later (since this edge is redundant with respect to connectivity). Next, we want to check for any disconnections, so check each pair of vertices to see if they are disconnected. When we find a disconnected pair, make an edge between this pair and print it along with one of the edges from before which we stored (as we're replacing the redundant edge with a useful one), and union the two vertices which were disconnected.

This should run in O(n^2) time, since the union find operations are (nearly) O(1) and we check for each pair of vertices being connected. Since n <= 1000, this is not a problem.
-Eliot

In fact, the checking process doesn't need O(n^2) time as above. We can simply connect every root that is constructed in the union-find process since we only need to connect one point in every connected components. Specifically, we can choose the root. Then how to find the root? The root has a property that it is the only point in the component that satisfies x=parent[x]. In this way, we can use O(n) time to find the edges to be added.
----Yucheng Dai

D. st-Spanning Tree
In this problem, we can first union all the edges that is not connected to S or T because they have no limits. So now we acquire a graph like this:



The points to the right are components, and the dot line between S and T means there might be an edge there. The yellow points represent components that is connected to both S and T.

In this case, if such yellow points exist, say, point X, then it is clear that choosing S-X and X-T is better than choosing S-T and another edge. So we can simply ignore S-T. If such yellow point doesn't exist, then S-T must exist and must be chosen because it is a connected graph. Thus, for those yellow points, we choose one of them to connect to both S and T, and others connect to one of them (choosing which one is based on whether the degree reaches maximum).

Also, the edge connecting to those components which only connect to one of S and T must be chosen.

At last, we check if the degree exceeds maximum and whether the graph is connected by n-1 edges.

----Yucheng Dai

E. New Year Domino

I know there is a union find algorithm because I find it on the Internet, but I don't know how it is done. The blogger didn't explain clearly.

My solution is prefix doubling.

We first find out if one domino falls without extension, which domino it can reach after a series of chain reaction. To do this, we first find out the farthest domino itself can reach. This takes $O(n \cdot \log n)$ time.

Then, we maintain a list L of lists satisfying:

1. Every list in L starts with a domino that cannot reach other domino. That is to say, if we meet a domino that cannot trigger chain reaction, we create a new list in L.
2. For other dominos, we search from the first list in L to the last. If it can reach the last element in the list, it will be added to the end of that list. If we search from the first to the last, we can assure that the first element of that list is the farthest domino it can reach.
3. We add domino into L in descending position order. That is to say, from the largest position to the lowest position. Therefore, we can assure that every domino to the right of the current domino is in one of the list.

In fact, this needs O(n^2) time to construct in the worst case. However, it has a beautiful property that if the previous domino (the one to the right) sets itself at list i in L, then if this domino can reach others, it cannot set itself after list i (obviously). So in this way, we can abandon the lists after list i because they will no longer be used by any other domino afterwards. This optimus needs O(n) time. See program for more details.

Therefore, we have construct several chains. In every chain, reaching **ANY domino before the head of that chain** doesn't need any cost. The chain doesn't mean it can only reach

dominos in the chain, but rather a list of the dominos that “matters” in further calculations. To reach dominos to the right of the whole chain, we need to find the domino that reaches the farthest distance and extend that domino until it reaches the next domino to the right of the chain. In fact, we don’t need to check other domino outside this chain because if it reaches the head of the chain, it will be inside the chain. So when we construct the chain, we can maintain a maximum value until this domino. See program for more details.
Now we can calculate the minimum extension length from one domino to the next domino of its chain. And the next domino is in another chain, so it also has a minimum extension length to the next domino of its chain. Using prefix doubling, we can calculate the minimum extension from x to y in O(log n) time. Together, we need O(q·log n) time.
Link: **http://codeforces.com/group/KlrM1Owd8u/contest/231913/submission/45300032**
----Yucheng Dai

F. Imbalance Value of a Tree