My Gold Editorials

Bribing Friends

This problem is a twist on the classic knapsack DP. However, whereas a knapsack has a time complexity of $O(NM)$ where N is the number of value-price pairings, and M is the amount of money one has. As we can see from this problem, instead of having only one money value, we have two. Using a "double-knapsack" we get a time complexity of $O(NM_1M_2)$, where N is the same, but M_1, M_2 represent the two different monetary values. In this problem, all values N, M, K are bounded at 2000. This makes our intuitive solution a bit too slow.

We can utilize some observations to make this problem a bit quicker. One such observation is that given a sequence of purchases, we always want to dump our ice cream on the cows with the cheapest ice cream costs. So one way to think of the problem is to first sort all the cows by the cheapest ice cream costs. Then we can iterate through the cows, making the decision to either purchase, in which we will dump as much ice cream as we can(as this cow is the best value), or we can skip. However, once we run out of ice cream, we don't really care and just switch to money. This way we essentially have just one currency, albeit one that dikes a bit of finking to use.

Mountains

We start off thinking about the naive approach to this problem. The naive approach is quite obvious, we just start off by getting our initial pairs in $O(N^2)$ fashion(this should be intuitive), then for each Q we just recalculate for a time complexity of $O(QN^2)$. This, however, is obviously a bit too slow. We realize that we need to speed up the process of updating the count, as N^2 is a bit too slow.

Then we go on trying to makes observations, but no matter what angle to look at the problem from, we realize that we need to somehow keep track of the existing pairs. Now given this(the official solution uses the more elegant monotonic sets) but pairs of (i, j) in a list sorted by i then j are the same idea. For some i, j such that $i < j$, for there to be a pair (i, j), we need that $\forall k | i < k < j$ the slope from i to k is less than the slope from k to j. In math, we write this as:

$$\frac{h_k - h_i}{k - i} \leq \frac{h_k - h_j}{k - j}.$$

Now, this provides a blueprint of how to calculate the initial pairs on $O(N^2)$ time, we start at an index, and go right, keeping track of the maximum slope met so far, and seeing if the slope at the current point is greater than that, if so, we create a pair by adding the current point to the monotonic set of the point we started at(IE: start from i iterating to the end, for each given j that works, we add j to the set of i), then we update the maximum slope.

However, there still raises a few questions on how to calculate for every Q in under $O(N^2)$ time. We realize that it is actually about $O(N \log N)$ time, as given an x, we update the x direct connections to x in about $O(N)$ time, meanwhile, for the indices less than x, we look to its monotonic set, then iterate through all the connections to j's greater than x and delete those that no longer work. The

reason this isn't N^2 is because for each Q we only increase a monotonic set by 1, meaning that if we somehow erase all j's for N time, we won't be able to do that again until N iterations of Q, thus the problem self bounds itself making our solution pass in $O(N^2 + Q(N + N \log N))$ time. Also notice that the slopes for a monotonic set are strictly increasing, meaning that there won't be wasted iterations. Either the iterator goes through and deletes the object in the set, or either it just stops.

In short, our algorithm is to first iterate through all i's, for each i we iterate through all j's. Such that $i < j$. Now we track max slope, for every $M \leq m_{ij}$, we have $M = m_{ij}$, and we add j to the set of i. Now we have the monotonic sets in sorted order. For each Q, we get an x that increases. First, to calculate the extra direct connections to x (notice that the direct connections will never decrease) we first calculate the number of direct connections to x before growth, then after growth. To update, we iterate through all $i < x$, then we update each i 's monotonic set, deleting all entries greater than x, which have a lesser slope than x. (This works well, as x 's slope is the only one that changes).

Strongest Friendship Group

This problem needs a Disjoint Set Union, or a DSU. A DSU is essentially a dfs, in which instead of given all the edges at once, you are given the edges one by one, and must make conclusions for every edge given.

One observation we make with this problem, is that the best way to approach the problem is to delete the "weak link" or the node with the minimum degree. Just using this observation, we are already close to the solution. We can make an algorithm, constantly removing the weak link, updating the rest of the orderd set(this runs in reasonable time because its bound by M) and then calculating the score.

This works, if not for one issue, the fact that it is possible for the weakest link to be connecting two subgraphs, deleting the weakest link would make computation much more difficult, as now we have essentially two graphs to work with. However, we can fall back on a weaker version of our observation: the fact that for every graph **we know for sure** who is the weak link, or who to delete. Now this is good, because we can at least get an ordering of the **order** of deletion.

(Note that we don't really care about the condition that it must count min degree within friend group right now, as the whole point is to trim the graph until we get the optimal group, furthermore, all outgoing edges from a node must count, as the nodes it connects to are part of the friend group.)

Using this ordering of deletion, we can reverse the process and recreate the graph. The key is that for each iteration, we focus on the node we just put, as well as its friendgroup. We can do this by tracking friendgroups by using a DSU, which is fed the edges that appear when our node appears(it will connect to everyone on its edges list that is at the time active). Then given this, we are able to get the **size** of the friend group at that iteration.

Remember how we need size multiplied by degree? Well we can get the strength by simply counting, but we can also store it from when we deleted the node.

Thus using these ideas we can find the solution by taking the max of everything calculated.