Problem Tutorial: "The One Polynomial Man"

Assume $0 \notin V$ (this case can be considered individually)

The main idea is to notice the following fact:

$$f(a,b) = \left(\prod_{z\in V}\left(\frac{(2a+3b)^2+5a^2}{(3a+b)^2}+\frac{(2a+5b)^2+3b^2}{(3a+2b)^2}-z\right)\right) \equiv 0 \implies$$

$$\left(\prod_{z\in V}\left(\frac{(2ad+3bd)^2+5(ad)^2}{(3ad+bd)^2}+\frac{(2ad+5bd)^2+3(bd)^2}{(3ad+2bd)^2}-z\right)\right) \equiv 0$$

Therefore, $f(a,b) \equiv 0 \iff a/b \in L$, for some set L that can be easily determined. So, the problem reduced to the following:

Count, for how many pairs of $a, b \in S : a/b \in L$. Let's find an arbitrary generator g under modulo p. Then $a/b \in L \iff g^x/g^y \in L \iff g^{x-y} \in L$ that can be count by Fast Fourier Transformation.

Problem Tutorial: "Alexey the Sage of The Six Paths"

Formal version of the statement: You have to add some edges to the bipartite graph. The cost of the resulting graph determined by the power of each vertex. You have to add edges in a way such that total cost is minimized and the size of maximal matching is at least l and at most r.

Let's add fake vertices source and sink. Then, we want to add some edges between source and first half, first half and second half, second half and sink and **direct** them in a way such that the are no path from source to sink. Therefore current matching will be maximum because augmenting path does not exist. It appears, that we can run two independent dynamic programmings for each half. Of course, the *transition* will be a little bit different in each of the dynamics, but, the idea and *states* will be same.

We will show how dynamic works for the first half.

Let's define $dp(i, left, right, edgesDirectedToFirstHalf, edgesDirectedToSecondHalf)$

as follows:

We want to assign edges to the vertices $i..n$ such that exactly *left* vertices will be to the left of the *cut*, exactly *right* vertices will be to the right of the *cut*. Although, we want to assign exactly *edgesDirectedToFirstHalf* + *edgesDirectedToSecondHalf* edges to some vertices with the respective meaning. In that terms, the size of maximal matching equals to *edgesDirectedToFirstHalf*. If proper implemented, each state of this dynamic calculates in $O(N+M)$ time, therefore, the total time complexity will be $O(N^3 \cdot M^2 \cdot (N+M))$.

Problem Tutorial: "Steel Ball Run"

Let's build a centroid decomposition of the tree and answer the queries by descending on it.

To calculate the answer we store several values. For each vertex we know its parents on all heights, also for each centroid and all its subtrees we maintain the number of chips, the total distance from all chips in the subtree to the centroid and the closest vertex to the centroid. Also, for each vertex we store the numbers of chips in the subtrees in set (in order to get the "heavy" subtree fast). It's easy to recalculate these values while updating.

Let k be a a current number of chips. Let we are in the vertex v.

If there are a centroid subtree with at least $k/2$ chips, it is easy to prove that the optimal answer lies there. Otherwise, the answer is v and we can add to the answer the precalculated total distance and stop the process. If the optimal answer lies in a subtree, we descent to this subtree and move all chips from all other subtrees to the closest vertex u in this one. After this we create a "fat" vertex with a weight equal to number of moved vertices. Then, while finding the heaviest subtree we will have to consider subtrees with these "fat" vertices. Fortunately, there can be only $\log n$ "fat" vertices.

If we implement distance calculation in $O(1)$, the solution will work in $O(n \log^2 n)$.

Problem Tutorial: "The Jump from Height of Self-importance to Height of IQ Level"

First Solution

Let's maintain permutation p in a cartesian tree. In each *node* we will store the following information:

- minimum in the subtree of *node*
- maximum in the subtree of *node*
- minimum element min_2 such that there exists element y to the left of min_2 and $y < min_2$
- maximum element max_2 such that there exists element y to the right of max_2 and $y > max_2$
- Does the corresponding segment for this *node* contain $p_i < p_j < p_k$ where $i < j < k$. In other words, the answer for this segment.

How to merge the above information of two nodes? Let's call *left* and *right* as nodes we want to merge, respectively.

1. If there is a desired triplet in a *left* or *right* node, then just break.
2. If no, then update minimum and maximum by taking minimum or maximum from the *left* and *right* nodes.
3. In order to update min_2, we must look for all pairs of elements $i < j$ such that $p_i < p_j$ and find such one where p_j is minimized. First, update min_2 with min_2 in the *left* and in the *right* node. However elements with indices i and j might lie in a different nodes. Therefore, we must take the minimum in the *left* node and perform *lower bound* by this minimum in a *right* node. The crucial idea is that, **the elements that greater than minimum in the *left* node form a decreasing sequence in the *right* node**, otherwise, we would break at the step 1. So, it is possible to perform a *lower bound* by going down in the cartesian tree. The max_2 can be found by a similar process.

Asymptotic: $O((n + q) \cdot \log^2 n)$ time, $O(n)$ memory.

Second Solution

It can be proved that the permutation, that does not contain a desired triple, is a combination of two decreasing sequences. More precisely, one of the sequences lies under diagonal $(i, n - i + 1)$ and, other one lies above this diagonal. Therefore, the solution is to maintain two sequences of decreasing records and check whether the sum of their sizes is equal to n. However, elements $p_i = n - i - 1$ must be considered carefully.
Asymptotic: $O((n + q) \cdot \log^2 n)$ time, $O(n)$ memory.

Problem Tutorial: "Minimums on the Edges"

First, let's precalculate, for each subset of vertices, the number of edges between them. It can be done in $O(2^n \cdot n \cdot n)$.

Then, let's solve the problem using dynamic programming. Let's calculate $dp[i][j]$: the maximum possible sum if we consider vertices in mask i and distribute j tokens between them. So there are only several transitions:

- First, we can delete one node from the mask because we don't want to add more tokens to it.
- Second, we can add one token to each node from the mask, and add the number of edges inside it to the answer.

So, the values of the dynamic programming function can be calculated in $O(2^n \cdot s \cdot n)$.

Problem Tutorial: "IQ Test"

The following algorithm performs no more than 43 moves:

Let's define $f(n)$ as function which would add element n to the S with possibly some intermediate additions. Let's take $x = \lceil \sqrt{n} \rceil$ and $y = x^2 - n$, run $f(x)$, $f(y)$ and finally, perform addition of $x^2 - y$ to the set S.

Problem Tutorial: "AtCoder Quality Problem"

Let's define $R'(S) = \sum_{T \subset S} R(T)$ and $B'(S) = \sum_{T \subset S} B(T)$ (Sum convolution over all submasks).

Let $dp(S)$ be the answer if we only consider coloring subsets of S. Without loss of generality, we will color subset S red. Then, there **must** be some element $e \in S$ such that all subsets of S that include e are red. *Proof:* if there is no such element, then set S can't be red, because the union of all blue sets will give set S. Therefore, we can calculate $dp(S)$ as follows:

$$dp(S) = \min_{e \in S} R'(S) - R'(S - e) + dp(S - e).$$

Problem Tutorial: "Mex on DAG"

Let's rephrase the problem statement first: if edge i has number $\lfloor \frac{i}{2} \rfloor$ in it, then all numbers from 0 to $n - 1$ occur exactly twice.

So let's solve the problem using binary search. Suppose that you want to check if there exists a simple path with mex equal to x. Then all numbers from 0 to $x - 1$ must lie on this path.

So let's make x variables for each number. The i-th variable will be true if we take the first occurrence of the i in the edge, or false if we take the second one.

Let's first precompute for each pair of edges if they can lie on the same path. Because the given graph is acyclic, it can be done in $O(n^2)$ using depth-first search.

After that, we can add several conditions: for each pair of numbers from 0 to $x - 1$, if some pair of edges with these numbers can't lie on the same path, let's add an edge in the new graph from the variable corresponding to taking the first edge (state true) to the variable corresponding to not taking the second (state false). So finally you need to check that the 2-SAT here has a solution, which can be done in $O(n^2)$ in one iteration of binary search. So, the overall complexity is $O(n^2 \log n)$.

Problem Tutorial: "Find the Vertex"

Let's take a look at each vertex with distance 0 modulo 3. One of them is our answer. After that you can notice, that there is only one such vertex in a graph with distance 0 modulo 3 such that all adjacent vertexes have distance 1 modulo 3. Because all nodes except the start one will also have at least one adjacent node with distance 2 modulo 3.

Problem Tutorial: "Yet Another Mex Problem"

At first, let's learn how to keep information about mex value of all segments ending in some prefix of the array. So let's keep it in the following format: make a set of segments such that segment correspond to the left endpoints of the suffixes of the current prefix with the same mex value.

Now we need to update this set efficiently while moving to the right in our array. We will maintain a segment tree of the last occurrences of the each value in our array. So using it we can easily find for each mex value what is the rightmost suffix which has mex value not smaller than ours.

So suppose that the new value added to our prefix is k. That segment with $mex = k$, if it exists, disappears into oblivion. After that there can be added several segment with mex bigger than k and deleted some more segments from the set. But because every time all segments have increasing mex values and we can't increase the number of different values infinitely, it can be proven that we add and delete $O(n)$ segments in total.

So now let's compute dp_i, maximum possible profit of dividing prefix of length i of the array into some subarrays. We need to recalculate through all such j smaller than i as maximum of the dp_j plus the value $(prefsum_i - prefsum_j)$ multiplied by mex value of the segment $(j, i]$.

Let's rewrite this formula: $dp_j - prefsum_j \cdot mex + prefsum_i \cdot mex$. So you can see that for the fixed mex value we can independently calculate maximum possible value of the first part on the corresponding segment where this mex value is achieved. We can do it using segment tree of dynamic convex hulls in $O(n \log^2 n)$, storing there lines with coefficients $(-prefsum_j, dp_j)$. So after that we can see that for all mex values we can get another bunch of lines with coefficients (mex, optimum $dp_j - prefsum_j \cdot mex$ for this mex value). We need to add this lines and delete them from some structure each time some segment of mex is added or is deleted when we go through the array. It's important to notice that because we can't divide our arrays into subarrays with length more then k we need to keep these mex lines only for the suffix of current prefix. To do so we can simply check the leftmost current mex segment: if it is outside of our suffix, we can just delete it forever, if it is partly inside, we can delete old line and add new recomputed line in a segment with left border equal to $i - k + 1$.

So one structure to perform such operations can be done using square root decomposition quite efficiently, because in every moment of time there is only one line with some value of the angle coefficient. So let's keep $O(\sqrt{n})$ convex hulls using stacks and when we add or delete line we will just rebuild it for the corresponding block. So to find the value of the dp_i we need to go through all blocks and calculate maximum for the $prefsum_i$. But because all numbers are positive in the array than prefix sums are only increasing and we can keep a pointer for the optimal line in each block and move it when we need to find the maximum. So this part can be done in $O(n\sqrt{n})$. And overall complexity is $O(n \log^2 n + n\sqrt{n})$.