Problem Tutorial: "Apollonian Network"

The solution to this problem is divided into two parts.

- Restore the graph building process

Let's assume that the boundary triangle is fixed.

Then there is only one vertex that is connected to each of these three vertices. After that, divide the triangle into three parts and solve the same problem recursively.

So we can brute force all triangles, there are $O(n)$ edges so $O(n\sqrt{n})$ triangles.

For this boundary, we can check that it is correct in $O(n^2)$ time. In the worst case, it is $O(n^3\sqrt{n})$, which is fast enough. And in fact, it is $O(n^3)$ because there are $O(n)$ triangles.

- Find the weight of the largest path in the graph

After that, we need to calculate the DP on the triangles that were generated in the first part.

The path is a connected set of vertices, such that all vertices besides the two endpoints have degree 2, and the endpoints have degree 1.

For each interesting triangle, we need to know the degrees of its vertices and also the connected components for the vertices of degree 1, assuming that we only use edges such that at least one endpoint of the edge lies strictly inside the triangle.

You can maintain all these values as the DP state, and merge the values of the three triangles on which the triangle is split.

Constraints were very friendly, so even a suboptimal realization of this DP should pass.

Problem Tutorial: "Bitwise Xor"

Several solutions exist, one of them is the following:

Using bitwise trie it is easy to prove that the minimum of $(a_i \oplus a_j)$ is achieved at some two adjacent numbers.

So you can sort everything, calculate dp_i is the number of good subsequences ending at element i, and recalc it as $dp_i += dp_j$ if $(j < i)$ and $a_i \oplus a_j \geq x$.

You can maintain all dp values in the bitwise trie (btw bitwise tree on binary words of length k is equal to segment tree on segment $[0, 2^k))$, to make transitions quickly.

Complexity is $O(kn)$, $k = 60$ in the problem.

Problem Tutorial: "Counting Cactus"

Let's calculate the answer using dp.

We will use three functions.

- $foo(v, s)$ means v is the root, the subtree is state s (without v)
- $bar(v, s)$ means v is the root, the subtree is state s (without v),and v is connected to a single node or v is in a single cycle.
- $baz(v, u, s)$ means we dp a cycle starting from v and ending at u, subtree is s

We can recalculate all these values in $O(2^{number\ of\ bits\ in\ s})$.

Overall time complexity is $O(3^n \cdot n^2)$.

Problem Tutorial: “Determinant”

Of course, the determinant is equal to the sum of $(-1)^{evencycles}$ among all decompositions of vertices of this graph into cycles.

This constraint is equal to “All edge-biconnected components have size $\leq k$”.

Let’s build the tree of edge-biconnected components and calculate DP on it.

Looking at a leaf in this tree, we have two possible choices.

- This block will be covered without using the outgoing edge from it, so we can delete this block completely, and multiply the answer by its determinant. ($type = 0$)
- This block will be covered using an outgoing edge from it, so we can delete all vertices of this block besides from the boundary vertex, and multiply the answer by the determinant of this block without the boundary vertex. After that, you should take cycle $v \to par \to v$ to the determinant decomposition. ($type = 1$)

Using these ideas we can calculate $dp_{v,type}$.

To recalculate it, in the current block we can fix the *type* which you need to calculate (and depending on this, decide whether you should delete the boundary vertex).

After that, for each vertex in the block, you should look at the outgoing edges from it to the blocks of its children. Either choose one child with $type = 1$ (this means that you connect this vertex to this chosen child) and set all others to $type = 0$ or set all of them to $type = 0$.

Unfortunately, if you will do it naively for each vertex in the block, it will be $exp(k)$.

But we can deal with it very simply! Let’s create for each vertex v a fictive one. If you are using the edge from a vertex to its fictive one, it means that you connected v to some child. Otherwise, it means that no children of v are using their boundary edge.

Let $s = \prod dp_{to,0}$, where *to* goes through all children of v.

Let’s look at what weights we need for vertex v with fictive u.

Set the weight of $u \to u$ equal to s.

The weight $v \to u$ is $\sum \frac{s}{dp_{to,0}} \cdot dp_{to,1}$. It is possible that $dp_{to,0} = 0$, so we should compute this sum without using division (ex. partial sums).

And, finally, weight $u \to v$ is 1.

After that, $dp_{v,type}$ is equal to the determinant of the obtained matrix.

The time complexity for a block of size k is $O(k^3)$, so the overall time complexity is $O(\frac{n}{k} \cdot k^3) = O(nk^2)$.

Problem Tutorial: “Easy Win”

Each connected component has a Euler circuit $\to$ each vertex has even degree.

For each edge (u, v) let’s create a vector of $n + 60$ bits, in the first n bits set $a_u = a_v = 1$, and the remaining 60 bits should contain the binary notation of the number of stones on the pile on the edge.

A subset of edges is good if and only if the vectors corresponding to them are linearly independent.

So in this problem, each time you add vectors you need to find the largest total weight of linearly independent vectors. Using some knowledge about matroid theory, you can note that each time you should add one vector to the answer for previous vectors and maybe delete one of them to make the answer as large as possible.

Using Gaussian Elimination with bitsets, you can do everything in $O(q\frac{(n+60)^2}{64})$.

Problem Tutorial: “Fast Spanning Tree”

- Useful note

If $a + b \geq s$, then $a \geq \frac{s}{2}$ or $b \geq \frac{s}{2}$.

- Solution

For each given edge, let's add events to the components a_i and b_i with weight $\frac{s}{2}$.

When one of these components has a weight which is at least the weight of some event on it, we will change this event to another one. If the current weights in components a_i and b_i are x_i and y_i, you will add events $x_i + \frac{(s-x_i-y_i)}{2}$ and $y_i + \frac{(s-x_i-y_i)}{2}$ to components a_i and b_i, respectively.

Of course, you will update the same event not more than $O(\log C)$ times, because each time $s - x_i - y_i$ is divided by two.

To maintain these events you should use Small To Large technique. When merging, just go by the list which contains a smaller number of events.

Total complexity will be $O(n \log n \log C)$.

Problem Tutorial: "Grammarly"

If the prefix of length $|s| - 1$ of string s is equal to the suffix of length $|s| - 1$ of string s, that means that all characters of s are equal.

It means that while s has at least two different characters, you have exactly two transitions in this graph.

Let's look at the first moment when the substring $s[l, r]$ will have only one distinct character.

If $s_{l-1} \neq s_l$ and $s_{r+1} \neq s_r$, it means that there are exactly $C(l - 1 + n - r, l - 1)$ paths to this substring, and after that $(r - l + 1)$ paths from it, so there are $C(l-1+.n-r, l-1) \cdot (r-l+1)$ where this substring is exactly the first moment when string started to has exactly one character.

Also, there are $O(n)$ such substrings.

Besides this one, all prefixes and suffixes of these substrings may be the first moments when the string has exactly one character.

There are $O(n)$ such substrings too, and for each of them, you can calculate the answer similarly.

We have not checked only those cases where the last string has at least two different characters.

For this, we need to calculate $\sum C(l - 1 + n - r, l - 1)$ for all substrings $[l..r]$, which contains at least two different characters.

For this, you can fix l and there exists a suffix of r, such that $[l..r]$ contains at least two different characters.

And you need to add $\sum C(l - 1 + n - x, l - 1)$ for $x \leq r \leq n$ to the answer.

Which means that you need to add $\sum C(a, l - 1)$ for $l - 1 \leq a \leq l - 1 + n - r$ to the answer.

To calculate $\sum C(x, k)$ for $l \leq x \leq r$, you need to find out how to calculate

$\sum C(x, k)$ for $0 \leq x \leq n$.

You can prove with simple induction that it is equal to $C(n + 1, k + 1)$.

Total time complexity — $O(n)$.

Problem Tutorial: "Honorable Mention"

- Convert into another, quite similar problem

We want to find a sequence $s_1, s_2, \ldots, s_n$. $(0 \leq s_i \leq 1)$.

And a sequence $y_1, y_2, \ldots, y_n$ $(0 \leq y_i \leq 1)$.

Such that $y_i = 1$ if $s_i = 1$ and $s_{i-1} \neq 1$, and $y_i = 0$ if $s_i = 0$.

For all other elements $0 \le y_i \le 1$, $\sum y_i = k$.

Let $f(k)$ be the largest possible $\sum(a_i \cdot s_i)$ under such constraints.

In fact $s_i = 1$ means that this element is covered by some segment and $y_i = 1$ means that this element is the beginning of some segment.

- Key lemma and its proof

Lemma: for fixed s_1 and s_n, $f(k) - f(k-1) \ge f(k+1) - f(k)$.

Proof: We can reduce this problem to max-cost k-flow, and as we all know, the cost of pushed flow in max-cost k flow is non-increasing.

- Precalculation

Let's build a segment tree. In each node, for fixed s_l and s_r we will maintain $f(1), f(2), \ldots, f(r-l+1)$.

This precalculation may be done in $O(n \log n)$.

For this, you can note that you can merge two children, for fixed s_l and s_r we should fix s_m and s_{m+1} and inside you should make something like "knapsack" for answers for l, m and $m+1, r$ with such s_l, s_m, s_{m+1}, and s_r.

Knapsack is — you have two arrays x and y, and you need to find another array z which is built as $z_{i+j} = \max(x_i + y_j)$.

In general case, it can't be done (at least science doesn't know how to do it) in $o(n^2)$, but x and y are convex! So we can merge the arrays of $x_i - x_{i-1}$ and $y_i - y_{i-1}$ into one sorted array, and in this order add guys to z. It is just a Minkowski Sum of two convex figures.

So you can merge some arrays of children for the chosen s_m, s_{m+1}, and relax the answers to the fixed s_l and s_r of the current vertex.

Also, if $s_m = 1$ and $s_{m+1} = 1$, you can drop the mark $y_m = 1$ and decrease k by 1.

So you can use the same merge in that case, but decrease indices by 1 and relax the answer in the same way.

So, the whole precalculation works in $O(16n \log n)$.

- Answering the queries

How are we answering the queries for the concave functions?

Yeah, we can use Aliens trick! Also known as wqs binary search.

You can binary search on x, increase answer by x for each y_i, and inside make some DP using the pre-calculated values in the segment tree.

For fixed array in the segment tree, we want to find the largest $kx + a_k$, where a_k is convex, does it looks familiar to you? Yeah, it is something like Convex Hull Trick queries, but the array itself is convex, so we don't need to build any hulls.

So, to answer the query for fixed x in $O(16log^2 n)$ time, you should split the segment in the segment tree into $O(\log n)$ for which you already have some precalculation. For these segments for each precalculated thing in $O(\log n)$ times do binary search to the find optimal $kx + a_k$ as in CHT.

After that, you need to recalculate dp for fixed x using the ideas similar to precalculation, when you are merging $X_{a,b}$ and $Y_{c,d}$ into $Z_{a,d}$ you can go to $X_{a,b} + Y_{c,d}$ or to $X_{a,b} + Y_{c,d} - x$ if $b = 1$ and $c = 1$.

Also, as always, to make Aliens trick work, you should maintain the smallest number of grabbed x that you can use for the best answer.

As you are doing Aliens trick, after that you need to do a binary search on x and find the moment when you will have exactly k segments.

Using this, we can solve the problem in $O(16n \log n + 16q \log^2 n \log C)$.

- Improving complexity

In this problem queries are offline, to decrease the amount of pain in the world.

Let's do a parallel binary search on x.

On the current iteration sort all queries by x, and instead of doing a binary search to find optimal $kx+a_k$, move a pointer to the right while $a_{k+1} - a_k \geq x$.

On one fixed iteration, all pointers will traverse $O(16n \log n)$ total, and you will need to make $O(16q \log n)$ merges to check the answer with current x on queries.

So, the total complexity will be $O(16(n+q) \log n \log C)$.

Problem Tutorial: "Interactive Vertex"

Let's ask all vertices adjacent to the centroid, if they all have the same distance, then it is 1 and the centroid is the answer! Otherwise, we can binary search the children of this centroid to find the required child, and after that proceed recursively. In the worst case, it will take $O(\log^2 n)$.

But instead of binary search let's make a weighted binary search.

Select an arbitrary order for the children. Let's find the first child, such that the total size of the children before it $\geq$ half of the total size of all children. If the answer is in it, just go to it. Else, check children to the left of it and the right of it, and find the children which you need to go to.

I claim that this works in $O(\log n)$. Why? Because inside this weighted binary search each time the number of candidate vertices that you are looking at is divided by two.

Problem Tutorial: "Jiry Matchings"

Solution 1:

This solution is pretty easy to come up with —

Let's maintain the orientation of edges of a tree as in Kuhn's algorithm for maximum bipartite matching. Each time you should find an augmenting path with the largest cost, and change the orientation of edges on this path.

For this, to find augmenting path one time, you can use standard *dp*, maintaining the largest cost of a path using some information about it as a state.

It is possible to update it using HLD with path reverting, using Dynamic Tree DP technique (ex. JOI Open 2018 Catdog).

Solution 2 (Official):

It is possible to use ideas from problem "Honorable Mention" about merging convex functions because the difference between successive answers is non-increasing. We'll also use small-to-large (HLD).

Root the tree at vertex 1. For each vertex v, let the heavy child of v be the child of v with the largest subtree size. We should store

- a vector with the max costs of all matchings for the subtree at v excluding the subtree of the heavy child of v and v itself
- a vector with the max costs of all matchings for the subtree at v excluding the subtree of the heavy child of v

- the cost of the edge connecting v to its parent

So when we're solving for vertex x, our steps are as follows:

1. solve for each child of x
2. for each non-heavy child c of x, combine the triples for every vertex on the heavy path ending at c into a single triple using divide and conquer (merge the bottom half of the path, merge the top half, then combine)
3. merge the triples for all non-heavy children of x using divide and conquer, and set the result equal to the triple for x

After calling solve(1), we should then merge the triples for all vertices on the heavy path ending at 1 to get our final answer.

Both these solutions may be implemented in $O(n \log^2 n)$.

Problem Tutorial: "K-pop Strings"

Some memoization solutions with precalc are possible.

The intended solution uses the inclusion-exclusion principle. Let's do it with brute force. However, if at some moment adding or not adding the current tandem repeat to the set of added does not change the connected components, this means that this branch of brute force will return 0 so you can exit.

Certain orders of tandem repeats allow the solution to work fast enough to pass all tests without precalculation. You can shuffle randomly or sort lexicographically in order of decreasing length for small n.