Problem Tutorial: "Ring Road"

Let's start by transforming the input instance to a binary tree: For any node with degree greater than 3, you can introduce additional edges with weight 0 to make a binary tree with equivalent distances. The nodes are doubled, but it doesn't matter because TL is lenient.

Construct a centroid decomposition per the given binary tree. Let c be the centroid of current subproblem. Since c has at most 3 child, we have 3 subtrees as well.

The optimal path always takes the following form:

- It passes a centroid c.
- It does not pass a centroid c but uses some edges that connect different subtrees of c.
- None of the above.

The first case can be solved easily by computing a single-source shortest path from c, using Dijkstra's algorithm.

For the second case, the edges that connect different subtrees of c are ring roads, and you can observe that they are at most the number of c's children. This fact is obviously true for the original tree. For the subproblems, note that you can add at most one edge that the graph is still a tree where each leaf is augmented with ring roads. Therefore, there are at most 3 edges that connect different subtrees of c. Passing an edge means passing one endpoint of each edge, so you can compute 3 single-source shortest path from any arbitrary endpoint of such edges.

The third case only holds if the queried vertices belong to the same subproblem - proceed recursively.

Each queries belong to at most $O(\log N)$ subproblems, and for each layer you call Dijkstra's algorithm 4 times, which takes $O(N \log N)$ time. As a result, we obtain an $O(N \log^2 N + Q \log N)$-time algorithm.

To find the relevant ring roads in the second case, simple DFS is sufficient, and you don't need any analysis of the specific structure of the given graph.

Problem Tutorial: "Query on a Tree"

The quantity $strength(S)$ is a sum of $\binom{|S_i|}{2}$ where S_i are the connected components of S under the given tree. Naively, you can scan all tree edges where both endpoints belong to S and compute the components with graph search or disjoint sets, which requires $O(N)$ time per query and times out.

Performing DFS on the vertices of S may seem to work, but this approach may encounter many irrelevant edges, hence this also times out.

To improve the algorithm, root the tree arbitrarily, and compute the *parent* for all non-root vertices. An edge is relevant if and only if $par(v) \in S, v \in S$, therefore you can simply iterate through S and check if $par(v) \in S$ with a simple boolean array. The time complexity is $O(N + \sum K_i)$.

Problem Tutorial: "A+B Problem"

The graph is known as *Halin graph*, and the problem asks us to compute the *tree decomposition*. Tree decomposition is a generalized way of characterizing graphs that are *similar* to trees (such as cacti). This problem essentially asks to prove that Halin graphs are similar to trees (and hence many tree-specific techniques just work). From now on we will use a standard vocabulary:

- New trees are called *skeleton tree*.
- Revolutionary sets are called *bags*.
- $\max |X_i| - 1$ is called *treewidth* (We have to compute width-3 tree decomposition).

Let's assume that the input is a binary tree. If we ignore the ring road, we can construct a trivial decomposition:

- Skeleton tree is just the input tree.
- Each bag contains the node, and the parent (if it is not a root).

It's easy to verify that the decomposition is valid, and has width 1. The tree is binary, so supporting the ring road is simple: For each ring road (u, v), add the element u over the path between u and v. This modification yields a solution of width 4.

To reduce the width by one, we appropriately subdivide the edges of the skeleton tree and spread the element of bags. Let $dfs(v)$ be the recursive function such that:

- it computes the tree decomposition of subtree of v.
- if $(l_a, l_{a+1}), (l_b, l_{b+1})$ are the ring road that goes out of the subtree in left/right direction, returns a skeleton tree where the root node has bag (l_a, v, l_b).

For some child (w_1, w_2) of v, the return value of $dfs(w_1), dfs(w_2)$ will be $(l_a, w_1, l_b), (l_b, w_2, l_c)$, respectively. We can merge them in a following way:

- Attach a new root node (l_a, w_1, v, l_b) for w_1 (and similarly to w_2)
- Attach a new root node (l_a, v, l_b) for w_1 (and similarly to w_2)
- Merge two trees with new root (l_a, l_b, l_c, v).
- Attach a new root node (l_a, v, l_c).

We need 4 new nodes for merging two trees, hence the construction yields a skeleton tree of size at most $4N$. Generalizing this to a non-binary tree is trivial.

Remark. As noted, the tree decomposition of a graph makes tree-specific techniques applicable to general graphs. For example, problem A can be solved by constructing a tree decomposition as in this problem, and by applying your favorite way of computing distance in a tree (centroid decomposition or sparse table). Of course, it's much easier to use ad-hoc approaches to problem A, but author knows at least five problems that can be described as "find tree decomposition, and copy-paste distance computing template in bounded treewidth graph"(For example, NEERC 2015 C, GCJ 2020 R2D, JOI Spring Camp 2017 D2P3).

Problem Tutorial: "Building Bombing"

For buildings on the left to building L, we only have to blow up the buildings that block building L from being visible. The main problem is to solve for the buildings on the right. From now on, we will assume $L = 1$.

We can solve the problem using dynamic programming. Let $D[i, k, h]$ be the minimum number of buildings to blow up only considering buildings $1 \dots i$, and after blowing up, the number of visible buildings becomes k and the greatest height among them becomes h.

We will add buildings one by one and calculate the corresponding DP values. There are three cases to consider when adding building i to the set. We can make it visible by taking $k - 1$ visible buildings on the left, where the greatest height is less than h_i. Or, we can make it invisible either by blowing it up or hiding it behind another building. The recurrence relation is as follows.

$$D[i,k,h] = \begin{cases} D[i-1,k,h] + 1 & \text{for } h < h_i \\ \min(DP[i-1,k,h], \min_{h'<h} D[i-1,k-1,h']) & \text{for } h = h_i \\ D[i-1,k,h] & \text{for } h > h_i \end{cases}$$

This takes $O(N^2K)$ naively, so we have to do it efficiently. Suppose we have added buildings $1 \dots i$ so far. We will store K arrays $D[i, k, *]$ for $k = 1 \dots K$ in K segment trees. Now, the task of adding a building to the set becomes adding 1 to a range, querying the minimum in a range, and updating a point in a segment tree. Each task can be done in $O(\log N)$ per segment tree using lazy propagation. Since there are K segment trees and we will add N buildings in total, the time complexity is $O(NK \log N)$.

Problem Tutorial: "Double-Colored Papers"

We build a suffix automaton for S and T, A and B. Then, all possible double-colored paper is a combination of path in automaton of S starting from the root, and path in automaton of T starting from the root.

Our strategy is to fix the characters one by one: if we have found the prefix pfx of the answer string ans, we can find the next character of ans, or conclude that $pfx = ans$ if we can calculate the number of double-colored papers with having $pfx + c$ as prefix, where c is a arbitrary character.

To do this, we should analyze the possible pair of paths given the string pfx. Let u be the vertex in A reached when following the path pfx until there is no edge of desired direction, and v be the vertex in B reached when following the path pfx, following the link edge while there is no edge of desired direction.

All possible endpoints of the first path are the vertices in the path in A from root to u, and all possible endpoints of the second path are the vertices in the path in link tree of B from root to v. The two pair of vertices are a valid pair if sum of the lengths they present is $|pfx|$. Be careful that vertices in B correspond may present multiple lengths.

Note that we are finding the maximum possible length of prefix and suffix of pfx that is included in S and T by this procedure, and we can maintain u and v while adding characters to the end of pfx.

With dynamic programming in A and B, we can calculate the number of paths starting from each vertex in the automaton, cnt.

To count the number of pair of paths, note that the vertices in B that are included in a valid pair forms a suffix of the path: we can also store the first vertex not included in the suffix. Let's call it w. Then, the number of pair of paths is sum of cnt values multiplied by the number of lengths the vertex is representing, for the vertices in the path $w - v$ in the link tree of B. w and v should have extra care, as only some suffix of w is counted and some prefix of v is counted, depending on minimum and maximum length of possible suffix of pfx.

With dynamic programming in link tree of B, we can calculate this value after adding character c to the end, in a method similar to prefix sum. If we update u, v and w after each update, we can solve the whole problem in $O(26(|S| + |T|))$.

Problem Tutorial: "Making Number"

Let's define N_i as the i-th digit of N.

There is two cases for Z:

- $Z = Y$.
- There exists an unique index i, $Z_j = Y_j$ for $1 \le j \le i - 1$ and $Z_i > Y_i$.

The first case is checked easily by comparing counts of each digit.

For the second case, we can find an answer by binary searching the index i.

Let's define $I(j)$ as the maximum index satisfies following, corresponding to a given index j:

- $I(j) \ge j$
- $Y_k \ge Y_{k+1}$ for $j \le k \le I(j) - 1$

i.e. $I(j)$ is the maximum index such that $N_{j\sim I(j)}$ is non-increasing sequence.

It can be checked that $i \leq j$ for given index j with following steps.

1. Calculate $C_I(d)$, the count of digit d in $Y_{j\sim I(j)}$, for $0 \leq d \leq 9$.
2. Calculate $C_P(d)$, the count of digit d in $Y_{1\sim j-1}$, for $0 \leq d \leq 9$
3. Calculate $C_X(d)$, the count of digit d in X, for $0 \leq d \leq 9$.
4. If $C_X(d) < C_P(d)$ for some $digit$, result $i < j$.
5. Find the maximum digit d satisfies $C_P(d) \neq C_X(d) - C_I(d)$.
6. If there is no such d or $C_P(d) > C_X(d) - C_I(d)$, result $i < j$.
7. Otherwise, check the existance of d' satisfies $d' < d$ and $C_P(d') > 0$.
8. If d' exists, result $i \geq j$. If not, result $i < j$.

Each step can be proceed in the node of segment tree where the node of segment tree is consists of count of each digit in each interval, l to r and l to $min\{I(l), r\}$.

Time complexity is $O(D(N + Q \log N))$ where D is the number of digits.

Problem Tutorial: "Permutation Arrangement"

Suppose that there are non-adjacent 8 empty spaces left in the permutation. Then, each remaining number has at least 4 squares that it can go to and each remaining square has at least 4 numbers that can be in the square. Hence, from Hall's marriage theorem, there exist the valid arrangement of remaining numbers.

Hence, when there are 15 or more empty spaces it is always possible to arrange the remaining numbers. Therefore, the solution of greedily assigning numbers until there are 15 numbers left and using bit DP to find the optimal assignment of last 15 numbers works.

From case-work it is possible to bring the minimum number of empty spaces needed down and hence the brute force solution of testing all the permutations in lexicographical order also works when implemented well.

The time complexity is $O(N)$.

Problem Tutorial: "Squirrel Game"

Let d_i be the sequence of distances between the squirrels. That is, $d_i = x_i - x_{i-1} - 1$, assuming there also is a squirrel at $x_0 = 0$. A single move decreases d_i by some amount and increases d_{i+K} by the same amount for some i. (If d_{i+K} does not exist, we just ignore it.) We can decompose the game into K distinct chains of distances in the form d_{Kq+r}, for each r in $0 \ldots K-1$. Each chain is equivalent to a game where $K = 1$. If we can calculate the Grundy number of each chain, we can solve the entire game using the Sprague-Grundy theorem. However, it turns out that we don't even need the Sprague-Grundy theorem to solve this problem.

Let us first solve for $K = 1$. From the right, label the squirrels with R and L alternatingly. We can pair up two consecutive squirrels with the label L on the left and R on the right. We claim that the game is equivalent to a Nim game where the heap sizes are the distances between each squirrel pair.

Suppose you have a winning strategy and have played accordingly. Your opponent has two choices: Move a squirrel labeled L or labeled R. If your opponent moves a squirrel labeled R, it is equivalent to reducing the heap size in the corresponding Nim game. Therefore, you are still in a winning position. If your opponent moves a squirrel labeled L, which is increasing the heap size, you can move the paired squirrel of label R so that the heap returns to the original size. Therefore, you are back in the winning position.

Note that the game has to end in finite turns because each move reduces the sum of x_i. It means that your opponent cannot indefinitely stall the game by always increasing the heap size.

When $K > 1$, the game is equivalent to playing K distinct Nim games, which is just a single Nim game with all the heaps from each game.

Problem Tutorial: "Two Paths"

First, with dynamic programming, we can find the result of following query $F(u, v, w)$ in $O(1)$: the farthest vertex from u except for the directions v and w, where v and w are either vertex connected to u or 0(no direction).

For a single query, let P be the path from u to v. Let w_1 be the last vertex of P_1 inside P, and w_2 be the last vertex of P_2 inside P. If we fix some vertex c inside P, then we can divide the cases by three: both w_1 and w_2 is in c - u path, both w_1 and w_2 are in c - v path, or w_1 is in c - u path and w_2 is in c - v path.

With centroid decomposition, we make c be the root of the tree. Note that while P is inside the current tree, now P_1 and P_2 may not be fully included in the tree, so we use F query to find the actual farthest vertex in the tree.

First, let's solve the case where w_1 or w_2 is c, and the case where w_1 is in c - u path and w_2 is in c - v path. In this case, optimal choice of w_1 and w_2 is independent, so by calculating optimal w for each vertex using dynamic programming(Using the F query), this case can be solved in $O(1)$.

Now we solve the case where both w_1 and w_2 is in c - u path. The case where both w_1 and w_2 is in c - v path can be solved symmetrically. In this case, if w_1 is fixed, the optimal w_2 is fixed, and can be found by dynamic programming(Again using the F query). Also, the result value with w_1 and w_2 fixed can be expressed as a line $Ax + By + C$ for A, B values in each query, and C being a constant fixed for each query. Therefore, we can solve this case with Convex Hull Trick in tree. You should be careful with the case $u = w_1$.

The time complexity is $O(N \log^2 N + Q \log N)$.

Problem Tutorial: "Village Planning"

$K = 0$ is easy: answer is $A_0^{\binom{N}{2}}$ for each N.

$K = 3$ is a trick: the answers are equal with the case $K = 2$.

Now we only need to solve $K = 1$ and $K = 2$. First, we solve the case where $A_i = 1$ for all i: we just need to count the number of possible graphs.

For both $K = 1$ and $K = 2$, we first count the number of possible connected components for each size, c_i. Then, we can count the number of ways to merge the components.

Let's fix the number of components k, and label each component from 1 to k. Let size of component i be n_i, where $\sum_{i=1}^{k} n_i = N$. Then, the number of possible ways would be $\binom{N}{n_1 \cdots n_k} \prod_{i=1}^{k} c_{n_i}$. Therefore, the number of graphs with k connected components is $\sum_{n_1+\cdots+n_k=N} \binom{N}{n_1 \cdots n_k} \prod_{i=1}^{k} c_{n_i}$. Finally, to remove the order of the components, we divide the result by $k!$.

The above analysis tells us that if the exponential generating function of c is $P(x)$, then answer's exponential generating function is $\sum_{k=0}^{\infty} \frac{P(x)^k}{k!} = \exp(P(x))$.

When $K = 1$, each connected component is a tree. The number of connected components of size N is N^{N-2}.

When $K = 2$, each connected component is a tree or a unicyclic graph. Let's count the number of connected unicyclic graph of size N.

The unicyclic graph is formed by a cycle and a rooted trees with each vertex in the cycle being a root. Let $Q(x)$ be the exponential generating function of N^{N-1}. By a similar analysis to above, we can find the exponential generating function of the number of unicyclic graphs as

$\sum_{k=3}^{\infty} \frac{Q(x)^k}{k!} \times \frac{(k-1)!}{2} = \sum_{k=3}^{\infty} \frac{Q(x)^k}{2k} = \frac{1}{2}(-\ln(1-Q(x)) - Q(x) - \frac{Q(x)^2}{2})$.

Now, the general case where $1 \leq A_i < 998244353$ is not too different: for each step, we are merging components. Note that the number of simple paths with both vertices in the same component is fixed, and the number of simple paths with two vertices in different components is fixed. Using this fact, you can just modify the generating function before and after each step by multiplying $A_i^{\binom{N}{2}}$ and $A_i^{-\binom{N}{2}}$ in each terms appropriately.

Problem Tutorial: "Window Arrangement"

This problem can be modeled as MCMF as following.

1. Create one source vertex, one sink vertex, one vertex for each room, and one vertex for each places where we can put windows
2. Make edges connecting source vertex and rooms and give it capacity according to number of windows needed and cost zero.
3. Make edges connecting rooms and potential window locations. Give the edge capacity one and cost zero.
4. For each potential window location, connect it and the sink vertex twice. The first edge has capacity one and cost zero and the second one has capacity one and cost equal to the discomfort we get when we put two windows in this location (i.e. product of numbers of people in two rooms connected to this window location)

And now running MCMF algorithm on this graph solves the problem. The time complexity is $O(n^2m^2)$.