IOI 2024 Solutions: Problem Tree

Problem Information

Problem Author: Pikatan Arya Bramajati (Indonesia)

Problem Preparation: Mohammadreza Maleki, Franciszek Witt, Kian Mirjalali

Editorial: Agustín Santiago Gutiérrez, Mohammadreza Maleki, Yiping Liu

Solution description

Various lemmas will be used throughout this solution detailed explanation. As their proof is generally quite involved, they are placed in a separate section after the solution description, so that it is easier to follow the explanation of the solution ideas.

Let $S[i]$ be the sum of $C[j]$ for all vertices j that are in subtree of vertex i. Thus, an array C is valid if for every vertex i, $L \leq S[i] \leq R$.

Subtasks 1,2,3 : $Q \leq 10$

We will explain an algorithm, which we will simply call BottomUpGreedy, that allows us to solve the first subtasks. Subtasks 1 and 2 allow for simpler or inefficient implementations of BottomUpGreedy.

- (Lemma 1) There is always an optimal solution where leaves have $C[i] = L$.
- (Lemma 2) There is always an optimal solution where $C[x] = L$ if x is a leaf node and $C[x] \leq 0$ otherwise.

Due to these lemmas, we can enforce both restrictions and pretend that the original problem requires us to set $C[i] = L$ at leaves and to choose values $C[i] \leq 0$ for the non-leaf nodes, while satisfying all the restrictions with minimum cost.

BottomUpGreedy is the following algorithm:

- Initialize $C[i] = 0$ at every node.
- Set $C[i] = L$ at every leaf node i.
- Process nodes in a bottom up fashion (e.g. by decreasing id)
- At each node i, while $S[i] > R$, choose a valid node j in the subtree of i having the minimum possible value of $W[j]$, and decrement $C[j]$ by one. A node j is called valid if after decrementing $C[j]$ by one, all of the restrictions still hold, i.e. $S[k] \geq L$ still holds for all ancestors k of j after decrementing.

The lemmas directly justify the first steps of BottomUpGreedy, but a detailed proof of correctness is quite involved.

- (Lemma G) BottomUpGreedy is optimal.

Subtask 1 allows us to implement BottomUpGreedy in a very simple way because we can always choose $j = i$.

Subtask 2 allows us to implement BottomUpGreedy in quadratic time, by explicitly searching for the minimum weight valid node in the subtree.

Subtask 3 requires an efficient implementation that can quickly find the minimum weight valid node in the subtree. This can be done with standard data structures such as priority queues / sets / segment trees.

Subtasks 4,5 : $W[i] \leq 1$

The problem with the previous solution in the case in which we have many queries is that we would need to run BottomUpGreedy from scratch for each query. This is because the $C[i]$ computed during the first steps of the algorithm are needed for later steps, and all of those values depend on L and R, which change every query.

We can develop a more involved algorithm, whose execution flow does not heavily depend on the values of L and R, and thus it can be run a single time to precompute enough information to solve any $[L, R]$ query. We need some additional lemmas:

- (Lemma 3) If $W[i] = 0$ for non-leaf node i, then "cutting i from its children" does not change answer. That is, adding the solutions to the resulting trees as if they were each an independent instance of the problem gives an optimal solution to the original.
- (Lemma 4) An optimal solution exists with the form of lemma 2, and having either $C[i] = 0$ for every non-leaf node i, or else $S[0] = R$.

By Lemma 4 we can deduce that answer to subtask 4 is given by the formula $kL + max(0, kL - R)$, where k is the number of leaves in the tree.

We can combine this knowledge with lemma 3 to have a solution for subtask 5: after we cut at nodes having $W[i] = 0$, we just have to add many instances of subtask 4. Efficiently handling this sum requires some prefix-sum techniques, similar to what is explained for the general solution.

General solution

We need a final critical lemma to be able to reach an efficient full solution:

- (Lemma 5) Let k be the number of leaves in the tree, and m be the minimum $W[i]$ among all the **non-leaf** nodes i. If $kL - R > 0$, then the total cost of an optimal solution is $(kL - R)m + s$, where s is the cost of the optimal solution to a modified input problem in which m is subtracted from $W[i]$ for every **non-leaf** i.

We can create a new TopDownGreedyRecursive algorithm.

Suppose for now that L and R are known and fixed just as before. To solve a subtree (initially we go for the whole input tree), we first check if $kL - R \leq 0$ and if so, we could just stop and return L times the sum of leaves' weights. Otherwise we find a minimum weight **non-leaf** node v in the subtree and set the answer to $(kL - R)W[v]$, in accordance with lemma 5.

Then we subtract $W[v]$ from $W[i]$ for each **non-leaf** node i in the subtree. Crucially this causes the new minimum weight to be $W[v] = 0$ and thus by lemma 3 we can now split the problem and recursively solve each subtree, adding all those results to our final answer.

The base case is that of a leaf node (and in that case the subtree has no non-leaf node to choose as minimum), for which the answer is just $W[i]L$.

For both efficiency and simplicity of implementation, it is better not to subtract to all weights explicitly, but rather when calling the subproblem recursively pass an integer indicating how much has to be subtracted from every weight in the calculations.

Also in order to be able to efficiently find the number of leaves k in the subtree and the minimum weight non-leaf node v, a segment tree with the so called "euler-tour tree-flattening" technique can be used. Note that a normal (non lazy) one-dimensional segment tree is enough, and it allows modifications which are necessary when cutting at v.

Specifically, the implementation can call the lower subtrees first (so one recursive call to each child of v) while the subtree containing the root is solved last, and so if the algorithm is made to "delete" (that is, set to neutral values) processed nodes from the segment tree, then the segment tree will contain precisely the right state for the final call and thus any "cutting" at node v is simply managed by consuming the tree in the correct order and doing single-node-deletions from the segment tree.

While all of the previous can be implemented in $O(N \lg N)$ time for a single run with known L and R, we need to see how to solve the problem with a single run of TopDownGreedyRecursive. The key to notice is that, apart from returning early from some call when $kL - R \leq 0$, the L and R values do not influence the decisions and execution of the algorithm at all: we always choose min weight node in subtree, cut it, and call recursively, a process which depends only on W and not on L, R.

So we can do a single run at the start where we keep iterating for generic L, R values, and we will end up with a sum of $(kL - R)W'[v]$ terms. Note that $W'[v]$ is the known "subtracted" weight at that call, which might differ from input $W[v]$. Also, each term has its own value of k, depending on the number of leaves in the subtree at that call.

Let s be the sum of weights of all leaves. A fixed factor of Ls must be added together with each of the previous terms to get the final answer. Note that each

$(kL - R)W'[v]$ term must be added only when the corresponding $kL - R > 0$, as otherwise the algorithm would not reach that sum.

This can be rewritten as $k > \frac{R}{L}$, and k can only take values in $[1, N]$. So our situation for a particular $[L, R]$ query is that for each term in the generic run of TopDownGreedyRecursive, we need to add either 0 or some known linear combination $aL + bR$ to the answer, depending only on the value of $min(N, \frac{R}{L})$. Using the prefix-sums technique, we can use the single generic run of TopDownGreedyRecursive to precompute arrays a, b of size N such that the answer to a query $[L, R]$ is $a[t]L + b[t]R$ where $t = min\left(N, \lfloor \frac{R}{L} \rfloor\right)$.

Proof of Lemmas

- (Lemma 1) There is always an optimal solution where leaves have $C[i] = L$.

Proof: to find one, take any optimal solution and take a leaf x having coefficient $L < C[x] \leq R$. By subtracting one to the coefficient in this leaf so that it becomes $C[x] - 1$, no condition $S[j] \leq R$ can fail because the $S[j]$ have not increased.

Also, condition $S[j] \geq L$ can only break at ancestors of x. If none breaks, then we have changed to a solution with lower or equal cost and decreased sum of leaf coefficients. Otherwise, if we consider the first (farthest from root, nearest to x) ancestor j where the $S[j] < L$, since that node has at least one child and all of those have sum at least L>0, then it must be the case that $C[j] < 0$. Thus we can increment $C[j]$ by one and the total cost does not increase, but the $S[k]$ values from j up to the root now remain unchanged from the original solution (since we decremented $C[x]$ but incremented $C[j]$).

In both cases, we are able to reduce the sum of leaf coefficients, and so by repeating this process we get to an optimal solution with $C[i] = L$ at every leaf node i.

- (Lemma 2) There is always an optimal solution where $C[x] = L$ if x is a leaf node and $C[x] \leq 0$ otherwise.

Proof: Take a solution with $C[i] = L$ at every leaf node i, by lemma one. Now suppose that there exists some non-leaf node i such that $C[i] > 0$. Let v be such a node that is as close to the root of the tree as possible. Let w be the deepest ancestor of v (that is, furthest from the root) having $C[w] < 0$ (thus, each ancestor i of v that is strictly between v and w has $C[i] = 0$). Note that if all ancestors of v have coefficient 0, then w does not exist.

After decrementing $C[v]$ by one, all $S[j] \leq R$ conditions will still hold. Notice also that no condition $S[j] \geq L$ can fail at nodes deeper than w because of this change: the node v itself has at least one child and thus by having $C[v] > 0$, then $S[v] \geq L + 1$ before modification, and the same holds for the ancestors of v below w which have $C[i] = 0$ and thus $S[i]$ directly equal to sum of children $S[j]$.

So if w does not exist, we can simply decrement $C[v]$ and get a new optimal solution. Otherwise, if apart from decrementing $C[v]$ we increment $C[w] < 0$, we also get a new optimal solution. By repeating this process we keep reducing the sum of the positive non-leaf coefficients, until there are none and thus our proposed optimal solution exists.

- (Lemma G) BottomUpGreedy is optimal.

Proof: Due to the previous two lemmas, a correct solution exists that can be reached by doing some decrements of the $C[j]$, and thus the algorithm can only fail because it incorrectly chooses which nodes to decrement.

Suppose that BottomUpGreedy is not correct for a test case, and consider the first time that it makes a mistake, that is, while processing some node i, it decrements $C[j]$ for valid node j of minimum $W[j]$ in the subtree of i, in a way that it becomes impossible to reach an optimal solution by doing more coefficient decrementing steps. We will call this point in time just before this incorrect decrementing of $C[j]$ "the splitpoint".

Take an optimal solution which matches the greedy up to the splitpoint, but may have done other decrement steps later. If the optimal solution did a decrement of that same $C[j]$ at any time after the splitpoint, then we could just swap order of decrements and the greedy choice would not have been incorrect.

Thus, we can assume that the optimal solution does not decrement $C[j]$ at all after the splitpoint. Since $S[i] > R$ at the splitpoint and this has to be fixed somehow, the optimal solution has to decrement $C[k]$ for some other node $k \neq j$ in the subtree of i after the splitpoint.

Take one such node k decremented after the splitpoint, such that $LCA(j, k)$ is as deep as possible (farthest from root, closest to j). We can change the optimal solution so that it does one less decrement of $C[k]$, and instead decrements $C[j]$. Note that $W[k] \geq W[j]$ and so if this change is valid we will still have an optimal solution. We will now prove that indeed it is valid.

Nothing outside the subtree of i is changed by this modification. In fact, the only modified $S[x]$ are those for nodes x in the path between nodes j and k (but **excluding their LCA**).

More specifically, an ancestor x of k in the path could in principle now fail the $S[x] \leq R$ check, but that will not be the case because $S[x] \leq R$ was already ensured by BottomUpGreedy before the splitpoint (note that $x \neq i$), thus the extra decrement of $C[k]$ was not needed for this.

On the other hand, an ancestor x of j might now fail the $S[x] \geq L$ check. However, because of the choice of k, the change will not break these conditions: Since j was a valid node at the splitpoint, if it is not possible to do the change it must be because of later decrements affecting node x, but any such decrement would be at a node w having a deeper $LCA(j, w)$ than $LCA(j, k)$, contradicting the choice of k.

Thus in all cases, we reach the conclusion that we can find a solution that agrees with the greedy algorithm for one more step than the splitpoint, contradicting our hypothesis that BottomUpGreedy made a mistake there.

- (Lemma 3) If $W[i] = 0$ for non-leaf node i, then "cutting i from its children" does not change answer. That is, adding the solutions to the resulting trees as if they were each an independent instance of the problem gives an optimal solution to the original.

Proof: $C[i]$ can be set to any value for free. Thus after we choose any solution whatsoever (optimal or not) for the subtrees below i independently, we can set $C[i]$ so that $S[i]$ becomes any desired value in range $[L, R]$ at no cost, and this is always the case regardless of what coefficients were choosen below node i.

Thus after having chosen all coefficients below node i, the remaining problem behaves exactly as if node i were a leaf node itself with $W[i] = 0$, and its solution does not depend at all on the previously chosen coefficients. So, the optimal solution is achieved when an optimal solution is used for each independent subtree below i, and then the remaining weights are chosen ignoring any node below i.

- (Lemma 4) An optimal solution exists with the form of lemma 2, and having either $C[i] = 0$ for every non-leaf node i, or else $S[0] = R$.

Proof: BottomUpGreedy creates such a solution. We can prove it by induction in the number of nodes in the tree. Notice that when processing node 0, if any child j of node 0 has $S[j] = R$, then we have $S[0] \geq R$ and so after fixing it will be exactly $S[0] = R$.

Otherwise, since can inductively assume that for each child j either $S[j] = R$ or the solution was all-zeros, we must now have that all of the non-leaf coefficients are still zero when reaching node 0 in BottomUpGreedy, and so the result also holds.

- (Lemma 5) Let k be the number of leaves in the tree, and m be the minimum $W[i]$ among all the **non-leaf** nodes i. If $kL - R > 0$, then the total cost of an optimal solution is $(kL - R)m + s$, where s is the cost of the optimal solution to a modified input problem in which m is subtracted from $W[i]$ for every **non-leaf** i.

Proof: The "all zero" solution does not work in this case as it would have $S[0] = kL > R$. Thus by lemma 4, we must have $S[0] = R$ instead and so the sum of $C[i]$ over all non-leaf nodes i is exactly $T = R - kL < 0$.

Similar to the reasoning for BottomUpGreedy correctness, we can think of the solution as made up of exactly $-T = kL - R$ "steps" of decrementing some non-leaf coefficient by one, when starting with the $C[i] = 0$ at non-leaves solution.

Let r be a **non-leaf** node of minimum weight, i.e. $W[r] = m$. As a counting trick, we will pretend that we start by setting $C[r]$ to T and the others to zero (note that it might not be a valid solution, but it does not matter as we are

just pretending). Then we already have the $(kL - R)m$ part of the final cost accounted for: it corresponds to this optimistic value where we only use the cheapest node.

Now again thanks to lemma 4, we know that the remaining problem is to choose **exactly** $-T = kL - R$ nodes to "swap" for the optimistic cheapest (note that a node chan be chosen multiple times). Thus when choosing a node i for decrement, its actual extra cost we will count as $W[i] - m$, as that is the actual cost that we incur when compared to our reference optimal solution by selecting i instead of r for a decrement.

Crucially, this final step in the proof only works because we know for sure that both before and after the weight adjustment, there are optimal solutions with exactly the same number of "decrements": exactly $kL - R$, which does not depend on the W values.

Alternative solution for subtasks 6,7

Let $F(C)$ be the sum of $|C[i]| \cdot W[i]$ for all vertices i.

The problem solution is A, the minimum value of $F(C)$, for all possible valid arrays C.

Let $B[u]$ be the number of leaves in subtree of vertex u.

By lemma 4 we know that for subtask 4, A equals to $B[0] \cdot L + \max(0, B[0] \cdot L - R)$

From now on, we might consider more than one weight array W.

Which means the only fixed parts are the given tree and a given query.

Let $F(C, W)$ be $F(C)$ for the given weight array W.

Let $A(W)$ be A for a given weight array W.

Observation: Consider 2 weight arrays W_1 and W_2, satisfying:

- For any i and j, if $W_1[i] = W_1[j]$, then $W_2[i] = W_2[j]$.
- For any i and j, if $W_1[i] < W_1[j]$, then $W_2[i] \leq W_2[j]$.

Let C be the coefficient array obtained from BottomUpGreedy. Then if we have $F(C, W_1) = A(W_1)$, we must also have $F(C, W_2) = A(W_2)$.

- *Proof*: At each step, if we can choose vertex v for W_1, then we can also choose v for W_2.

For a weight array W_1 and an integer T, let $H(W_1, T)$ be another weight array W_2, where $W_2[i] = 0$ if $W_1[i] \leq T$, and $W_2[i] = 1$ if $W_1[i] > T$.

Observation:

- $A(W) = \sum_{t=0}^{\infty} A(H(W, t))$

- *Proof*: Let C be the coefficient array obtained from BottomUpGreedy. Then for each t, $F(C, H(W,t)) = A(H(W,t))$.

$$\begin{aligned}
\sum_{t=0}^{\infty} A(H(W,t)) &= \sum_{t=0}^{\infty} F(C, H(W,t)) \\
&= \sum_{t=0}^{\infty} \sum_{u=0}^{n-1} [W[u] > t] \cdot C[u] \\
&= \sum_{u=0}^{n-1} C[u] \sum_{t=0}^{\infty} [W[u] > t] \\
&= \sum_{u=0}^{n-1} C[u] \cdot W[u] \\
&= F(C, W) \\
&= A(W).
\end{aligned}$$

Now we can work out $A(W)$ by calculating each $A(H(W,t))$. Notice that this value is 0 for $t \geq \max(W[0 \ldots n-1])$.

$H(W,t)$ contains only 0 and 1, which is the same as Subtask 5. The answer is equal to the summation of

$$\text{leaf} \cdot L + \min(0, \text{leaf} \cdot L - R)$$

over all connected components formed by vertices with $W[u] = 1$, where leaf is the number of leaves in this component.

Traverse t from $\max(W[0 \ldots n-1])$ to 0. In $H(W,t)$, each vertex u will change from 0 to 1 at point $t = W[u] - 1$. Thus, there will only be $O(n)$ different connected components over all t's, each will survive in a certain interval of t.

Say, there are $K = O(n)$ connected components, the i-th has leaf$[i]$ leaves, and its survival time is life$[i]$. Then the answer is

$$\sum_{i=0}^{K-1} \text{life}[i] \cdot (\text{leaf}[i] \cdot L + \min(0, \text{leaf}[i] \cdot L - R))$$

We can use DSU to work out leaf$[i]$ and life$[i]$ in $O(n \log n)$ time.

Subtask 6: 1. In this subtask, $L = 1$ is fixed. 2. Let ans$[R]$ be the answer for R. 3. For each connected block, its contribution to ans is equivalent to adding a linear function to a prefix and a suffix.

Subtask 7: 1. Rewrite the answer as

$$\left(\sum_{i=0}^{K-1} \text{life}[i] \cdot (\text{leaf}[i] + \min(0, \text{leaf}[i] - R/L))\right) \cdot L.$$

2. We can apply the same trick as in Subtask 6 for R/L.