Jobs

The problem can be represented as a rooted tree with each job being a separate vertex and vertex 0 being the root. For each job i, the job it depends on p_i represents its parent in the tree.

Lines

We begin with subtask 3: Suppose that every vertex except 0 has at most one child. So we are basically given multiple stacks. And in every step we can pop the top element from some stack and add it to our score. This version of the problem can be solved with a simple greedy idea:

We start by popping some elements from some stack S_1 and then we switch to some other stack S_2. Now, it doesn't make sense for us to switch stacks before we made profit. I.e. we don't want to pop elements with negative sum from S_1 and then switch to S_2. So we limit the set of actions which we are allowed to perform to: *Pop elements from some stack until their sum becomes positive.* If it is possible to perform such action, do it.

For example, suppose that there stacks are $S_1 = [-1, -1, 4, -2]$ and $S_2 = [-1, -3, 9]$ and $s = 2$. We don't want to pop a -1 from S_1 and then a -1 from S_2. By switching stacks early, we just block later options. Instead, we want to keep popping from S_1 until the sum of popped elements becomes positive. Thus, we pop -1, -1, 4, but not -2. This increases our score to 4. Next, we pop all elements from S_2, increasing our score to 9. After this, no possible actions remain.

A naive implemetation of the above algorithm leads to a $O(N^2)$ solution: We perform at most N actions and for every action we check all stacks. This can be easily optimized to $O(N \log N)$ with a priority queue: For every stack, we maintain the minimum score required to perform an action on it (pop elements until their sum becomes positive). If there is any possible action, then we can definitely perform the one whose required score is minimal.

Slow Tree Solution

To motivate the full solution, let's look again at the $O(N \log N)$ solution for the line problem: The order in which we take the vertices is independent of s. The value of s only determines when we have to stop. So we basically transformed a problem with multiple lines (stacks) into one which consists of only one line. And the order of the elements on that line corresponds to the optimal ordering of vertices.

So one has hope that such optimal order exists in the tree case as well. And this is true! Every subtree can be transformed into a line. Let the values on this line be denoted by $A(v)$ for the subtree of v. For a leaf $A(v) = [x_v]$. What about non-leaves? Suppose that v has children $c_1, ..., c_k$. We first have to take x_v. Afterwards, we are in the case of the previous section, because we can transform the subtree of c_i into the line $A(c_i)$. And from the previous section we also know how to merge multiple lines into a single one (we reuse the priority queue idea for it). Let $\text{merge}(L_1, ..., L_k)$ denote the result of merging k lines. Then we see that

$$A(v) = [x_v] + \text{merge}(A(c_1), ..., A(c_k)) \tag{1}$$

where "+" denotes concatenation of lists.

Since every merge requires $O(N \log k)$ time and the sum of all k is N, we obtain a $O(N^2)$ time complexity.

Full Solution

To speed up the previous solution, notice that we are not actually interested in $A(v)$. We don't care for individual elements on the lines. We always take contiguous blocks of positive sum. So let's instead store a list of pairs (a, b) representing a block such that:

- We need to have a score of at least a to take all of its elements (i.e. $-a$ is the minimum prefix sum of the block).
- The sum of all elements of the block is $b > 0$.

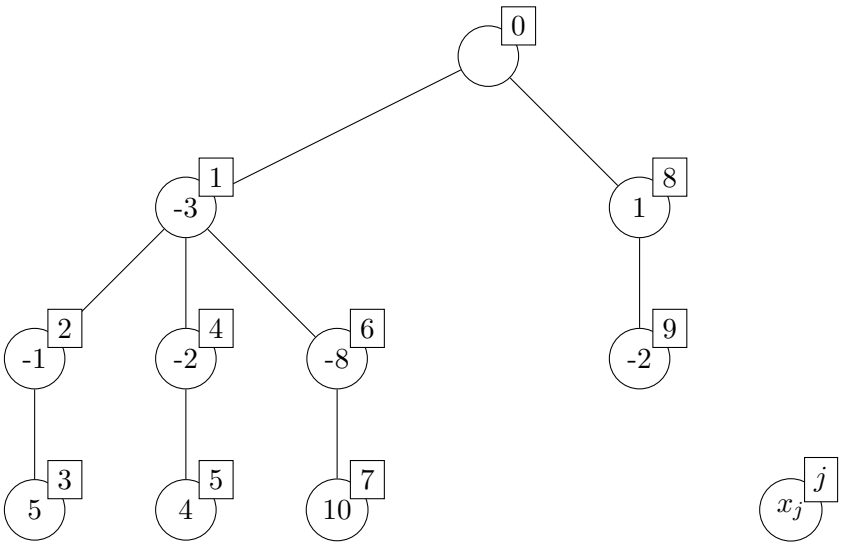
Let $B(v)$ be the block representation of $A(v)$. Now, let's see how our greedy algorithm for merging lines works with this representation: We pop the block with the smallest a value and append it to the result. Now, suppose that $B(v)$ contains two consecutive blocks of the form $(a_1, b_1), (a_2, b_2)$ where $a_1 > a_2$. When block (a_1, b_1) is popped, the second block is popped directly after it. Since those blocks are always taken together, we can combine those blocks to a single one: $(a_1, b_1 + b_2)$. By performing this combination step repeatedly, we can make $B(v)$ sorted in non-decreasing order by a. So now assume that $B(c_i)$ is sorted. Next, notice that the process of calculating merge$(B(c_1), ..., B(c_k))$ basically performs the the merge function from merge-sort on the lists $B(c_1), ..., B(c_k)$ (where a is the key by which we sort). Thus, the result is simply a sorted version of $B(c_1) + ... + B(c_k)$. When merging, we can thus maintain the sorted order of the blocks in a data structure like `std::set` and insert the smaller set into the larger one. With this well-known trick, we can achieve a $O(N \log^2 N)$ complexity for the merge part of the solution.

The step of inserting vertex v at the beginning of the list is done as follows: First of all, the block which just contains v is $(\max(0, -x_v), x_v)$. If $x_v > 0$ everything is fine. Otherwise, we have to combine it with the next block until the sum becomes positive. I.e. if $(a_1, b_1), (a_2, b_2)$ are the first two blocks and $b_1 \leq 0$, we have to combine them to $(\max(a_1, a_2 - b_1), b_1 + b_2)$. When $b_1 > 0$, we might still not be done, because we might have $a_1 > a_2$. So we keep combining the first two blocks as long as $a_1 > a_2$. If in the end only one block is left and it doesn't satisfy $b > 0$, we can simply discard it and say that $B(v)$ is empty, i.e. it is impossible to make profit in the subtree of v.

Notice that we don't actually need the full power of `std::set`. All we need is a structure, which allows us to extract the smallest element. So we can actually achieve the better time complexity $O(N \log N)$ by using a data structure like a *randomized heap* which allows us to extract the minimum and to merge two heaps both in $O(\log N)$.

Example

To illustrate the algorithm, we consider the following example:



The values of $B(v)$ are as follows:

v	$B(v)$
3	$(0, 5)$
2	$(1, 4)$
5	$(0, 4)$
4	$(2, 2)$
7	$(0, 10)$
6	$(8, 2)$
1	$(4, 3), (8, 2)$
9	
8	$(0, 1)$
0	$(0, 1), (4, 3), (8, 2)$

In particular, we calculate $B(1)$ as follows: First, we merge $B(2), B(4), B(6)$ to get $[(1, 4), (2, 2), (8, 2)]$. But then we have to insert $(3, -3)$ at the beginning: $[(3, -3), (1, 4), (2, 2), (8, 2)]$. We don't want blocks with negative sum, so we combine the first two blocks to $(\max(3, 1 - (-3)), -3 + 4) = (4, 1)$. This gives $[(4, 1), (2, 2), (8, 2)]$. But now the required score for the first block is higher than that of block 2. So we combine them: $[(4, 3), (8, 2)]$. Now the blocks are sorted and we are done.

For $B(9)$ note that in an intermediate step we get $[(2, -2)]$. But this block has negative sum and we can't combine it with any other block. So, we discard it say that $B(9)$ is empty. Assume $s = 3$. Then, we can recover the maximum profit from $B(0)$: since our score is 3, we can take the block $(0, 1)$. Now, our score is 4. So we can take block $(4, 3)$, making our score 7. But $7 < 8$, so we can't take block $(8, 2)$. Therefore, the maximum possible score is 7 and the profit is 7 - 3 = 4.

Credits

- Task: Lucas Schwebler (Germany)
- Solutions and tests: Justas Klimas, Dovydas Vadišius (Lithuania)