## OGRADA

Imagine a vertical line sweeping from the y-axis to the right. This line will encounter left and right edges of the given boards – each such encounter is called an event. Also, we say that by encountering the left edge of a board we open the board; similarly, encountering the right edge closes the board.

We will maintain two sets of boards: visible (we will keep these in the fence) and invisible (covered by visible boards). The set of open visible boards always forms a "staircase" going up and right. When a new event occurs, we need to determine if and how it affects the sets of open boards.

If the event is opening a new board, then the new board is visible if it is taller than the currently tallest open board and invisible otherwise (because it is covered by that tallest open board).

If the event is closing a board, then it must be removed from the set of open visible or invisible boards, whichever set it is in. Also, if this board was the tallest open visible board before removing, then it is possible that the tallest open invisible board becomes visible and must be moved to the other set.

The visible and invisible open boards must be kept in a data structure that allows efficient retrieval of the tallest board in the set. Examples of such structures are priority queues and binary search trees.

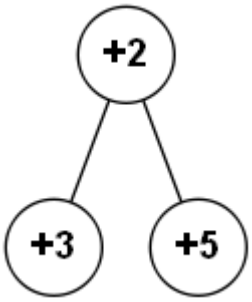
Additionally, we must carefully sort the events to correctly process events at the same x-coordinate.

## PODJELA

Optimization problems on trees can often be solved with dynamic programming. We assume that the tree is rooted, i.e. that one node is the designated root node and for each other node we know its parent (the first node on the path to the root). In this problem the given tree is not rooted, but it is possible to arbitrarily choose the root without changing the result. We need to determine a recursive relation in which the solution for a subtree rooted in node F can be found by combining the solutions for the subtrees rooted in node F's children.

First we make a simple transformation of the problem by calculating the "effective" amount of money each farmer has: this amount is the received amount X minus the deserved amount. In this new problem, we must make as few transactions as possible so that no farmers are left with a negative amount.

Let f(F, T) be the largest effective amount farmer F can have after exactly T transactions are made in his subtree, with the additional constraint that all farmers in his subtree (except for F himself) have zero or more money in the end. The following figure shows examples of values of f for various subtrees (F is the root of the depicted tree). The numbers inside nodes are initial effective amounts of money.

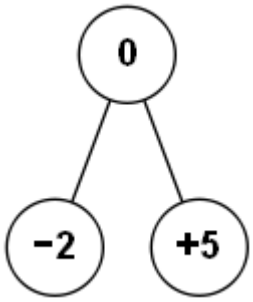



Farmer F got 2 extra units of money, his left child got 3, his right child 5 extra units.

f(F, 0) = +2. With T=0 we cannot make any transactions so we cannot extract the extra money from the child nodes.

f(F, 1) = +7. With one transaction available, we should take the extra money from the right child.

f(F, 2) = +10. With two transactions available we can take all of the extra money from the children.




Farmer F got exactly what he deserved, the left child got too little, and the right child got too much.

f(F, 0) = −∞. The left child must receive at least 2 money units, which is impossible without making any transactions. The condition from the definition of f, that all nodes in the subtree (except the root) must have at least zero, is not satisfied.

f(F, 1) = −2. We have one transaction available and our priority is to feed the left child. Farmer F sends 2 money units to the left child, which he will have to get back from his parent in the tree.

f(F, 2) = +3. Farmer F can receive 5 money units from the right child and forward 2 of those to the left child.

In calculating f(F, T), some of these T transactions will be used in the subtrees of the children (deeper in the tree), while some will be between F and his children. If we decide to make a transaction between farmer F and his child C, assuming C gets to make some number T' of transactions in his own subtree, then it is clear which direction that transaction will be in: if F(C, T') > 0, then child C will send F his excess money, otherwise C will have to receive money from F.

The optimal distribution of T transactions among children can be determined with a nested application of dynamic programming: g(F, T, n) is the largest f(F, T) if we have considered the first n children of farmer F so far. The values g(F, *, n) can be calculated by combining g(F, *, n−1) with f(child[n], *).

The smallest total number of transactions needed is the smallest T for which f(root, T) ≥ 0.

The complexity of this algorithm is not easy to analyse because it depends on the degrees of nodes. One upper bound is $O(N^3)$, but in practice the solution is faster than one would anticipate from this bound.

Based on the calculated values of f and g it is possible to determine where the transactions occur in the optimal solution. It is still necessary to determine a correct order for them. This reconstruction was worth 30 points. One elegant way was, for each node:

1. Recursively reconstruct the transactions in subtrees of children from which a node needs to get money;
2. Take excess money from children in step 1;
3. Distribute excess money to children in need;
4. Recursively reconstruct transactions in subtrees of children from step 3.