

SORT THE LEAVES

Creating a tree sort algorithm for the Codechef question SORT THE LEAVES.

Problem Statement:

You are given a full binary tree. Each of the nodes have an index between 1 and N (total number of nodes). The root of the tree is vertex 1. Each of the leaf nodes have a value associated with them which lies between 1 and L (no. of leaf nodes) both inclusive. Also, to each leaf, we can assign a string in the following way:

Each leaf node can be associated with a simple path $S_i = v_1, v_2, \dots, v_l$ from the root to leaf v_l

- the string S_{v_l} assigned to leaf v_l has length $l-1$;
- for each valid i , S_i is 'L' if v_{i+1} is the left son of v_i or 'R' if it's the right son of v_i

Our aim is to make our tree leaf sorted i.e. for every pair of different leaves a, b : S_a is lexicographically smaller than S_b if and only if the value of a is smaller than the value of b .

Input:

The first line of the input contains a single integer T denoting the number of test cases. The description of T test cases follows.

The first line of each test case contains a single integer N .

N lines follow. For each i ($1 \leq i \leq N$), the i -th of these lines contains two space-separated integers l and r . If $l = -1$, vertex i is a leaf with value r . Otherwise, l and r respectively denote the left and right son of vertex i .

Output:

For each test case, print a single line containing one integer — the minimum required number of operations or -1 if it's impossible to make the tree leaf-sorted.

Algorithm: (Pseudo code)

```
# Assume the (min, max) range for the subtree is stored in Max_min[].
# Number_swaps returns -1 if the tree can't be sorted at given node
# else it returns minimum number of swaps.
```

```
int Number_Swaps(Tree, Max_min, index):
    if is_leaf(index):
        # leaf value is already sorted
        Max_min[index] = {leaf_value, leaf_value}
        return 0      # as we didn't do any swaps
```

else:

```
lft = solve(left_son(index))
rgt = solve(right_son(index))
```

```
if lft == -1 or rgt == -1:
```

```
    # which means we can't tree sort the subtree rooted at left or
    # right son
    return -1
```

```
if left_son(index).max < right_son(index).min:
```

```
    # no swap is required and the subtree will be sorted rooted at
    # index node.
```

```
    Max_min[index] = {left_son(node).min, right_son(node).max}
    # Reassigning the min and max values at that node.
```

```
    return lft + rgt
```

```
    # The lft and rgt subtrees were already sorted. So no swaps
    # required.
```

```
else if left_son(index).min > right_son(node).max:
```

```
    # Which means the left and right subtree can be swapped and
    # there is no overlap between those cases.
```

```
    # Greedily perform swap and tree will be sorted
```

```
    Max_min[index] = {right_son(index).min, left_son(index).max}
```

```
    return lft + rgt + 1
```

```
    # 1 specifies the swap done at the current index
```

else:

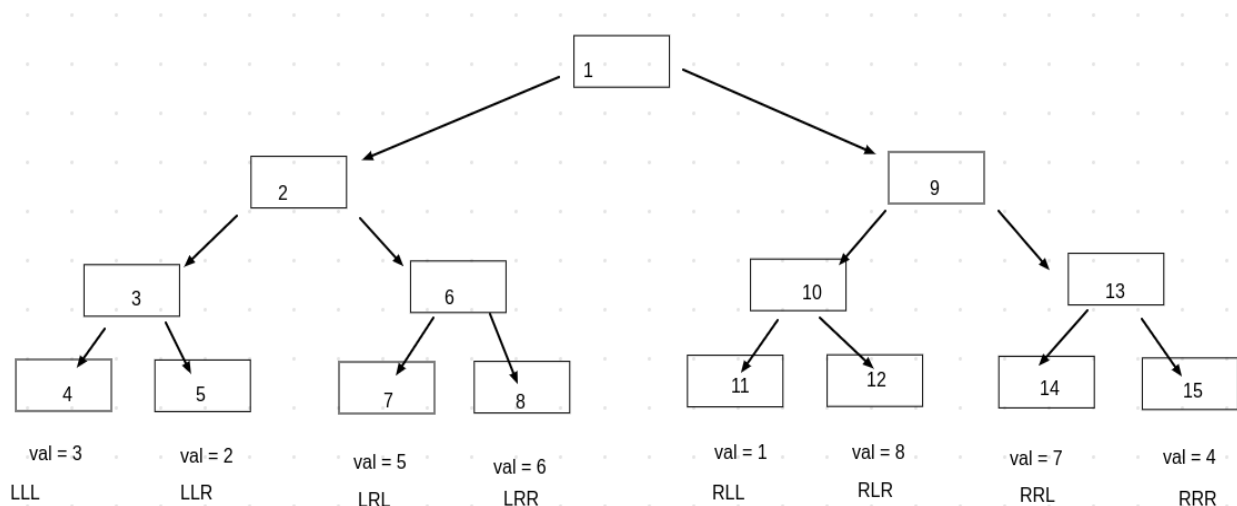
```
    # Even after swapping, some number in left son are
```

```
    # greater than some number in right son. So tree
```

```
    # can't be sorted.
```

```
    return -1
```

Description of the Algo:



We need to understand that the order of leaves is same as in the preorder traversal of the tree, Also it can be understood as the flattening(compressing) of the tree from the top. So in the above example, the order(after compression) will be values 3, 2, 5, 6, 1, 8, 7, 4. We need to make this a sorted order i.e the values of leaves should be 1, 2, 3, 4, 5, 6, 7, 8 in sync with the lexicographic order of the strings.

Here we have two choices:

1. We use top-down approach.
2. We use bottom-up approach.

Why can't we use top down approach?

Consider we use top-down approach. So, there might be a scenario where we swap initially, and then again we might have to swap it after doing some changes below the node. So we may land up swapping again or there might be no possible swaps although a solution might exist. We also observe that a bottom-up approach has the similar path but in reverse as for the preorder traversal. We can also see that swapping via bottom-up approach makes all the nodes below it move in blocks. So we use bottom-up approach.

Bottom-up Approach:

We have a full binary tree given to us. We also are given the values of leaf nodes. So at every node we store the max and min value in the subtree rooted at that node. For eg. The leaf nodes will have the max and min values same as the value of that leaf node. Gradually, we will propagate upwards and then we will construct max and min values for every node.

Now consider any two nodes A and B (where A is left leaf node and B is right leaf node)

Now there can be the following cases:

1. $\max(A) < \min(B)$
(which means all the values in the left subtree are less than all the values in the right subtree, hence no swapping required.)
2. $\min(A) > \max(B)$
(which means all the values in the left subtree are more than all the values in the right subtree, hence we can swap the left and right subtree and land up with a subtree sorted at current node.)
3. none of them.
(which means we have some numbers overlapping between left and right subtree, so we can't sort at this index- which means we can't sort the whole tree.)

Then we can update the Max_min values for the current node:

The Max_min values for the current node will be :

- case 1. (left_min, right_max)
- case 2. (right_min, left_max)
- case 3. in third case we don't have solution.

While propagating upwards we also propagate the count as well:

We use recursion for the same. We create two variables lft and rgt as the Number_Swaps(left_son) and Number_Swaps(right_son) respectively. The conditions we need to check are same as written in the pseudo code above.

Implementation of the algorithm:

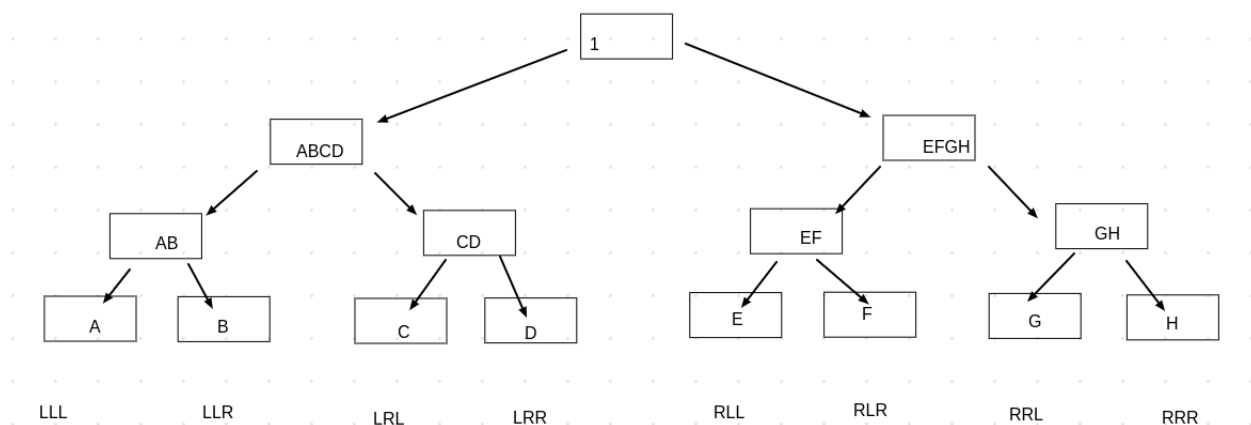
(Specifications of data-structures used).

For the implementation of the algorithm we used:

1. a C++ map for mapping the tree indices to its left and right son's indices.
(int) ---> pair(int, int)
2. a C++ map for mapping a node to the max and min values in the subtree rooted at that index.
(int) ---> pair(int, int)
3. Input is taken from the standard input (cin) and output is displayed at standard output (cout).

Proof and Correctness of the algorithm:

Consider the following diagram for reference,



Consider the depth of the the tree to be D.

Note: we are using the notation for depth to be distance from the farthest leaf node upto that level.

We will prove the algorithm by **mathematical induction**:

Induction base-case(depth = 0): We know that the every leaf node is tree-sorted as it has no left and right child to compare. It can be treated as a separate sorted unit.

Induction hypothesis: We assume that our algorithm gives correct number of swaps for depth 'd'. ($d < D$).

Induction Procedure:

We are given that our algorithm gives correct number of swaps for the depth 'd'.

To prove: The algorithm is correct for depth 'd+1'.

So for explanation purpose assume that our algorithm gave correct answer for the subtree rooted at 'ABCD'.

At ABCD, we see these 4 cases:

1. Any subtree of the tree rooted at AB or CD has no solution.
2. if $(\max(AB) < \min(CD))$
3. if $(\min(AB) > \max(CD))$
4. none of both.

We see that our cases are exhaustive which means we have covered all the cases.

If it is case 1:

Then we can't find the solution for the whole tree.

If it is case 2:

We don't need a swap between AB and CD because they are already sorted.

Claim: We don't need a swap between AB and CD.

Proof:

Suppose the max and min for subtrees AB and CD is mx_AB , mn_AB , mx_CD and mn_CD .

So we know that $mn_CD > mx_AB$.

Which means that:

$mn_AB < \text{all the numbers in subtree AB} < mx_AB < mn_CD < \text{all the numbers in subtree CD} < mx_CD$.

So it is already leaf-sorted at node ABCD.

If it is case 3:

Claim: We need a swap between AB and CD and this is a safe swap.

Proof:

Suppose the max and min for subtrees AB and CD is mx_AB , mn_AB , mx_CD and mn_CD .

So we know that $mx_CD < mn_AB$.

Which means that:

$mn_CD < \text{all the numbers in subtree CD} < mx_CD < mn_AB < \text{all the numbers in subtree AB} < mx_AB$. And we swap AB and CD.

Now it becomes leaf-sorted at ABCD.

If it is case 4:

Which means there are some overlapping numbers between AB and CD subtrees which prevent us from sorting it.

So the tree as a whole cannot be sorted.

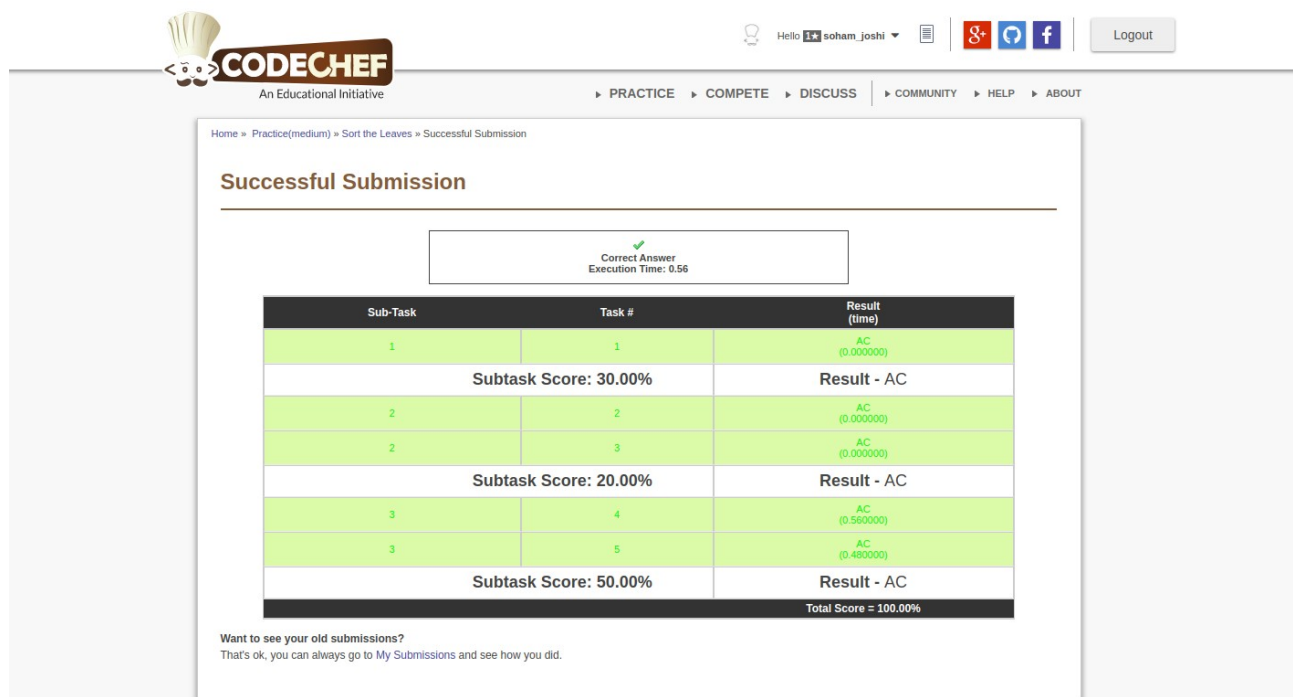
Which is what we return from the recursion call for ABCD and it gets propagated upward to its ancestors.

So we have now proved that we sort the tree at the depth 'd+1' if it is possible to sort and report whether it is impossible to sort.

Why our algorithm gives minimum number of swaps?

We are using the bottom-up approach. So we take into account the results of the subtrees rooted at children. Hence we don't do unnecessary swaps as in the top-down case. Also, we ensure the optimality of our swap i.e. we swap only when we are sure that it will give a subtree-sorted at current node. Hence, we can say we used minimum number of swaps.

Screenshot of the submission:



Home » Practice (medium) » Sort the Leaves » Successful Submission

Successful Submission

Correct Answer
Execution Time: 0.56

Sub-Task	Task #	Result (time)
1	1	AC (0.000000)
Subtask Score: 30.00%		Result - AC
2	2	AC (0.000000)
2	3	AC (0.000000)
Subtask Score: 20.00%		Result - AC
3	4	AC (0.560000)
3	5	AC (0.480000)
Subtask Score: 50.00%		Result - AC
Total Score = 100.00%		

Want to see your old submissions?
That's ok, you can always go to [My Submissions](#) and see how you did.

How to run the code (on a local machine):

1. Make sure you have g++ (a compiler for C++) installed on your local machine.
2. These are the steps for running the code:
 - i) Ensure you are in the project directory.
 - ii) Type the command 'g++ Coded_solution.cpp'.
 - iii) This will result in creating a a.out file.
 - iv) You need to run this file using command './a.out' (if you are using Linux terminal, but if you are using Windows type 'a.out')
 - v) Now you can give the input and corresponding output will be displayed after you give the input for a test case.

Contributions:

Saksham(IMT2018065) and me(Soham - IMT2018072) came up with the proof of the correctness of the algorithm.

Coding and preparing various testcases was done by Saksham.

I debugged the code and wrote this report of the project.

References and resources used:

1. GeeksforGeeks – map, pair and STL data structures.
2. Awwapp.com for the online scribble board used for discussions.