A3_Report_122090337

Problem_1

1 Initial Analysis

The task here is to find a way to transverse through all the nodes (or say, edges) efficiently, and calculate the distance of **every pair** of the black nodes. So there are some questions to answer:

1. How to transverse the nodes/edges efficiently?
   A possible answer is to use depth-first-search. First go down to the bottom of the subtree, then return back to another subtree to continue to check.
2. How to "go down" and "return back", as mentioned in 1?
   A good way to "go down" is to recursively search the endpoint of the edge, each edge one by one. For "return back", it will be more efficient if we can record the returning point in advance when building the tree.
3. How to record and accumulate the distance between black nodes?
   That depends on finding a good formula. By drawing the picture and try to list some examples, it is easy to find that $blackNodeCounts[deeperIndex] * currentEdge.length$ in a subtree. This formula will be revised and used in the coding.

Now let us find out how to get the answer.

2 Possible Solutions for the problem

Brute Force

Checking each node by each node, get the distance for each node and then sum them up. The time complexity of $O(n^2)$ is unacceptable. Thus we will never consider it.

3 How to Solve the Problem

I construct two classes in the code: Node and Edge. Node is easy to understand. It will e easier for us to store the colour and order in it.

However, we need to be very clear that Edge is also important in this question. Since the distances between nodes are actually the sum of Edge.length, that's why for this question, directly operating on Edge and Edge array will be much more convenient.

We use array to store the Node and Edge, because reading will be quite fast.

First let me explain how to implement `dfs` as mentioned in the initial analysis.

```
private static long dfs(Node root, Edge currentEdge, long[] blackNodeCounts) {

    long deeperBlackCount = root.colour;
```

This is for initialize the deeperBlackCount, or say the number of black nodes in the subtree, including the root itself.

```
    while (currentEdge != null) {
        Node deeperNode = currentEdge.toNode;
        int deeperIndex = deeperNode.order - 1;
        deeperBlackCount += dfs(deeperNode, edges[deeperIndex], blackNodeCounts);
```

If the subtree is not null, then we set the head of subtree as new root for iteration. We convey the data to `dfs` again for recursion. This is actually to search deeper into the subtree. That's why we call it `dfs`.

```
        totalDistance += (totalBlackNode - blackNodeCounts[deeperIndex]) * blackNodeCounts[deeperIndex] *
currentEdge.length;
```

It is the formula for calculating the totalDistance of edges. By drawing the pictures, the relationship will be quite clear.

```
currentEdge = currentEdge.besideEdge;
```

When we reach the bottom of the subtree, we get back to the original root's other subtree, by going to the `currentEdge.besideEdge`.

```
int rootIndex = root.order - 1;
blackNodeCounts[rootIndex] = deeperBlackCount;
return deeperBlackCount;
```

Returning the deeperBlackCount (explained before), so that the calculation can work smoothly.

Now we finish talking about `dfs`.

Then let's see the constructing tree process is like this:

```
private static void constructEdgeTree(int totalNode, Edge[] edges) {
    for (int p = 1; p <= totalNode - 1; p++) {
        int nodeIndex = sc.nextInt() - 1;
        Node fromNode = nodes[nodeIndex];
        Node toNode = nodes[p];
        long length = sc.nextLong();

        Edge newEdge = new Edge(length, fromNode, toNode);

        int fromNodeIndex = fromNode.order - 1;
        if (edges[fromNodeIndex] == null) {
            edges[fromNodeIndex] = newEdge;
        } else {
            Edge lastEdge = findLastEdge(edges[fromNodeIndex]);
            lastEdge.addBesideEdge(newEdge);
        }
    }
}

private static Edge findLastEdge(Edge edge) {
    while (edge.besideEdge != null) {
        edge = edge.besideEdge;
    }
    return edge;
}
```

However, all questions on OJ are AC except question 8 is TLE.

After a desperate night of thinking, I found that the problem here is in constructing the tree. Initially I did't pay so much attention to constructing since I think `dfs` is the most important part for saving time.

Now the problem for the original version is: `findLastEdge` requires searching for null form current node to the bottom, again and again for each node. That would be really slow. Thus, `ends` is added to record the last edge, to make constructing much faster.

Now I will elaborate the AC version of constructing.

```
private static void constructEdgeTree(int totalNode, Edge[] edges, Edge[] ends) {
    for (int p = 1; p <= totalNode - 1; p++) {
        Node fromNode = nodes[sc.nextInt() - 1];
        long length = sc.nextLong();
        Edge newEdge = new Edge(length, fromNode, nodes[p]);

        int fromNodeIndex = fromNode.order - 1;
        if (edges[fromNodeIndex] == null) {
            // If no edge exists for this index, initialize with newEdge
            edges[fromNodeIndex] = newEdge;
            ends[fromNodeIndex] = newEdge;
        } else if (edges[fromNodeIndex].fromNode == newEdge.fromNode) {
            // If an edge exists and is from the same node, update besideEdge
            ends[fromNodeIndex].addBesideEdge(newEdge);
            ends[fromNodeIndex] = newEdge;
        } else {
            // If an edge exists but from a different node
            if (ends[fromNodeIndex] != null) {
                ends[fromNodeIndex].besideEdge = newEdge;
            } else {
                edges[fromNodeIndex].besideEdge = newEdge;
            }
            ends[fromNodeIndex] = newEdge;
```

```
        }
    }
}
```

It will be too trivial if I explain the code line by line. The point here is, we combine the Edges, and set the `fromNode` and `toNode` for reading Node order and colour in `dfs`. And more importantly, we store the end, or say bottom of each subtree in the array `ends`, and give end of `ends` a besideEdge, representing the beside subtree head of the original root. So `dfs` will wok smoothly later on.

The time complexity for constructing the tree and dfs are both $O(n)$, now it is acceptable.

Problem_2

1 Initial Analysis

First let us clearly find out what results are required in this problem. What we are given is a sequence of prices. Operation `BUY` is adding new price number after the sequence of prices. Now our task is to find out:

1. what is `CLOSEST_ADJ_PRICE`, i.e. the minimum of absolute value of difference between adjacent prices;
2. what is `CLOSEST_PRICE`, i.e. the minimum of absolute value of difference between any two prices.

Now the question is: what is the most efficient way to store and find the required data?

2 Possible Solutions for the problem

Brute Force

Intuitively, we come up with an easy idea that we can write three functions for each operations. `BUY` is just adding a number in the array or tree or something else. `CLOSEST_ADJ_PRICE` and `CLOSEST_PRICE` can be found by transversing the array or some other data structure.

Since this requires us to apply the function on every `CLOSEST_ADJ_PRICE` and `CLOSEST_PRICE` operations, the time complexity will go to $O(mnlogn)$. In OJ this will be awarded with ten desperate TLE. Where to improve?

Updating the data with AVL tree

Two things to improve: AVL tree; update result immediately after adding prices.

A wiser data structure is AVL tree. We can directly get the biggest number in the tree, or say in the price sequence after every inserting price operation (initial price sequence and `BUY`). That will be very easy for us to **update the `CLOSEST_ADJ_PRICE` and `CLOSEST_PRICE` exactly after adding the price.** Thus we don't need to waste time on transversing the tree again and again to find out the smallest difference. Now the time complexity will go to $O(nlogn)$, enough for AC.

Besides, AVL tree can be used simply by `import java.util.TreeSet;`, really easy and convenient.

Now it is time for considering how to code.

3 How to Solve the Problem

Function `getMinDiff`

It is used to find the two closest elements to the given element in a sorted TreeSet and calculates the minimum difference between them and the given element. Simply saying, it is to find the predecessor and the successor in the tree for the given inserted element, and store them in Long.

Function `main`

It is used for receiving the data given in the initial price sequence. The key part to elaborate is here:

```
for (int i = 0; i < numberOfPrices; i++) {
    Long currentPrice = sc.nextLong();

    if (previousPrice != -1L) {
        minAdjacentDiff = Math.min(minAdjacentDiff, Math.abs(currentPrice - previousPrice));
    }
    previousPrice = currentPrice;

    if (prices.contains(currentPrice)) {
        minDiff = 0L;
```

```
    } else {
        prices.add(currentPrice);
        minDiff = Math.min(minDiff, getMinDiff(prices, currentPrice));
    }
}
```

The loop iterates for `numberOfPrices` times to do the operation for each number in the initial price sequence.

1. It reads a Long-type `currentPrice` from the input. It will be the current price being processed.
2. It checks if `previousPrice` is not equal to -1L, which means it's not the first price in the sequence. If it's not the first price, it calculates the absolute difference between the `currentPrice` and the `previousPrice`. It then compares this difference to the current value stored in `minAdjacentDiff`, and it updates `minAdjacentDiff` to the smaller of the two values. This keeps track of the minimum difference between adjacent prices in the sequence.
3. It updates `previousPrice` with the value of `currentPrice` so that it can be used as the previous price for the next price insertaition.
4. It checks if the `prices` TreeSet already contains the `currentPrice`. If it does, it means the current price has already been encountered before. In this case, it sets `minDiff` to 0L because the minimum difference is now 0, as the current price is a duplicate.
5. If the `currentPrice` is not already in the `prices` TreeSet, it adds it to the TreeSet using `prices.add(currentPrice)`. Then, it calculates the minimum difference between the `currentPrice` and the prices already in the `prices` TreeSet by calling the `getMinDiff` function. It compares this minimum difference to the current value stored in `minDiff` and updates `minDiff` to the smaller of the two values. This keeps track of the minimum difference between the `currentPrice` and any price in the TreeSet.

Finally two variables are updated through the operation:

- `minAdjacentDiff`: tracks the minimum absolute difference between adjacent prices in the sequence;
- `minDiff`: tracks the minimum difference between the `currentPrice` and any price that has been encountered in the sequence so far.

Updating process id finished. Now the problem is solved.

Function `switch (operation)`

For `BUY` operation, the steps are exactly the same as each iteration in `Main`, since operating a price in the initial price sequence has the same meaning as `BUY`.

The other two operations are to print the result, because the result was updated to the newest.