A3_Report_120090542

Question 1.

2.1 Solution & Implementation.

The solution is based on inclusion and exclusion algorithm and DFS of graph. Firstly, we construct the graph by attributes:

1. self.base: An array to record all the nodes inserted
2. self.sub_black: A dictionary, where each key is the node address and the value is the sum distance of black balls under this key node to this key node

For each node, it contains:

1. self.next: a dictionary of children, where key is the children's node and value is the distance between the children and the node
2. color, (=1 represents black)

Now, to address this, we use inclusion of exclusion algorithm:

We start from arbitrary node, say S:

```
def dfs(self,start:TreeNode):
        distance = 0
```

exclusion: We exclude S. Then, we will have all of its children are disjoint. We separately calculate each children's sum of distances of all the black pairs as below

```
        #Exclusion
        for child in start.next.keys():
            tmp = self.dfs(child)
            distance += tmp
```

Inclusion: Now, we include the node S. With include, the following things will happen:

1. If S is black, then, the inclusion will immediately increase the sum of distances of the nodes, which is indeed the distance of S to every other black nodes
2. The introduction of S will connect all the disjoint subtree of S. Assume for each subtree, we have

   $$(s_1, n_1), \cdots, (s_m, n_m)$$

   where m is the number of subtree and s_i is the sum of distances of black nodes to S. n_m is the number of black nodes. With trivial calculation, we can show the introduction of S will induce the increment in sum by:

   $$\sum_i \sum_j (n_i s_j + n_j s_i)$$

   this is achieved by code

```
def sum_of_sum_of_pairs_optimized(tuples_list):
    total_sum = 0
    running_value_sum = 0
    running_anti_count_sum = 0

    for value, anti_count in tuples_list:
        total_sum += (value * running_anti_count_sum) +
(anti_count * running_value_sum)
        running_value_sum += value
        running_anti_count_sum += anti_count

    return total_sum
```

   The time complexity is $O(n)$.

   We need the tuple list $(s_1, n_1), \cdots, (s_m, n_m)$, this is introduced by function

```
def next_black_dis(self,start:TreeNode):
        dis = []

        for child in start.next.keys():
            length = start.next[child]

            if child.color == 1:
                dis.append(length)
            k= [i+length for i in
self.next_black_dis(child)]
            dis+=k
            tmp = []
            length = start.next[child]

            if child.color == 1:
                tmp.append(length)
            tmp += k
            t = (sum(tmp),len(tmp))
            self.sub_black[start].append(t)

        return dis
```

this is $O(n)$ DFS.

3. 2 will lead to attribute sub_black to record the information. We have the inclusion code:

```
#Inclusion
        array = self.sub_black[start]
        distance += sum_of_sum_of_pairs_optimized(array)
        if start.color ==1:
            distance+=sum_of_tuple(array)

        return distance
```

Overall, the inclusion and exclusion takes two DFS, hence the time complexity is $O(n)$.

2.2 Advantages:

1. Using graph to represent the node and edges so that we can record the distance of each node to all the sub black balls respectively. We can use first DFS to record such information and in the second DFS, we can retrieve directly from the dictionary, which is efficient.
2. Overall, only two DFS traversals are used hence the time complexity is $O(n)$, significantly faster than the brute force.

Question 2.

2.1 Solution & Implementation

To get the desired result, we use Binary search tree as the data structure of our price manager class . In the Price_Manager class, we define operations:

1. Find Successor & Find Predecessor: Given a node with certain price, find the prices that closest to the node price. This is attained by $O(\log(n))$ approximately in BST. Ever time we input a new price, we can immediately find the only two prices that is close to the new price. Therefore, we no longer need to search for all data and we update the closest prices of all each time we have a new data
2. Define a top attribute, denoting the last element of the current array. each time we have a new price, we compare the absolute difference of the top value and the new price with the adjacent minimum before, and we update adjacent minimum each time. This is $O(1)$.
3. Each time we get a new instruction, we can output closest price $O(1)$, or insert (from bottom to tup) $O(\log n)$ or get adjacent closest price $O(1)$. Hence the overall complexity for each instruction is $O(\log n)$.

The price realization is achieved by

```
def add_value(self,value):
        self.adj_min=min(abs(self.top-value),self.adj_min)
        self.top = value

        z = Node()
        z.value = value
        self.bst.insert(z)

        s = self.bst.successor(z)
        p = self.bst.predecessor(z)

        if (s is None):
            diff = abs(p.value-z.value)
        elif (p is None):
            diff = abs(s.value-z.value)
        else:
            diff = min(abs(s.value-z.value),abs(p.value-z.value))
        self.all_min = min(self.all_min , diff)
```

Note that when the inserted price is the global minimum or maximum, we only retrieve one closest price.

Therefore, each time when we add and value, we update the adjacent min and closest min, to ensure for each retrieval, we can have $O(1)$ performance. The overall complexity is indeed

$$O(m \log n)$$

$O(\log n)$ falls on insertion.

2.2. Advantages

1. For closest price, using BST, we can access the successor and predecessor in $O(\log n)$ time and we do not care about other nodes. This strongly avoided the traversal of all the elements again.
2. The algorithm is sealed in class. The use of class could help to record the current adj_min and all_min each time we obtain a new data. We do not need to re-evaluate the two values every time, making the algorithm retrieval extremely efficient.