Question #1

(a)

A screenshot of a computer

Description automatically generated

A screenshot of a email

Description automatically generated

Question #2

(a)

The Union Find/ Disjoint Set Algorithm. The main idea is to split nodes into different unions, and check if some nodes are connected or not. If and is complementary to each other, then any node in the union must not relate to any node in . The pseudocode is as follows. Here, n represents the length of elements to consider.

Can-Satisfy-Constraints (n, m1, m2):

# Initialize Union-Find data structures, by:

# Firstly, setting the parent of every element to itself.

parent = [0, 1, …, n - 1]

# Then initialize every element’s rank as 0.

rank = [0, 0, …, 0]

# Define helper functions for Union-Find

# The find function is to find the top-parent of one element.

Find(x):

# If the parent of x is not x itself,

if parent[x] != x:

# then set x’s parent as recursively found parent, the top-parent.

parent[x] = find(parent[x])

# If we found the element whose parent is itself, it must be the top-  
# parent, so return it. Then every child of x will have their parent as x.

return parent[x]

# The union function is to link two elements based on their rank.

Union (u, v):

# Find two elements’ parents.

root\_u = find(u)

root\_v = find(v)

# If their parents are not the same:

if root\_u != root\_v:

# Case 1, u’s parent rank is greater than v’s.

if rank[root\_u] > rank[root\_v]:

# In that case, we put v as u’s child.

parent[root\_v] = root\_u

# Case 2, v’s parent rank is greater than u’s.

elif rank[root\_u] < rank[root\_v]:

# Then we do vice versa.

parent[root\_u] = root\_v

# Case 3, u’s parent is the same rank as v’s.

else:

# Then it’s the same. We just pick u as the parent and   
# update its rank.

parent[root\_v] = root\_u

rank[root\_u] += 1

# Process equality constraints:

for (xi, xj) in m1:

# Link all elements existed in equalities.

union(xi, xj)

# Check inequality constraints

for (xi, xj) in m2:

# Check inequalities. If two elements have the same parent, it means   
# they must be in the same union. Then it is impossible to satisfy both   
# equalities and inequalities.

if find(xi) == find(xj):

return False

return True

(b)

For answer this question, I referred to a [YouTube video](https://www.youtube.com/watch?v=aBxjDBC4M1U).

The total running time of above algorithm is .

Firstly, initializing parent and rank arrays takes time.

Secondly, the Fund function takes time. What we did in Fund function is called path compression, which compresses the graph of a long linear node link into a dense 2-layer graph. So theoretically speaking we would not encounter a long linear node link graph. In practice, can be considered nearly as constant time.

Thirdly, the Union function takes time as well, since this function calls Fund function, and other operations are constant time.

Last, iterating with Union function will take time and iterating with Find function to check if there is any contradiction with will take time, since we need to iterate all conditions in them.

So, combining all parts, the total running time of the algorithm is:. Since is considered nearly constant, we could say the running time of this algorithm is .

Problem #3

(a)

The overall idea for a greedy approach is to compare which restricted box is cheaper for pack items contiguously, and then repeat this process until all boxes are packed. Following this idea, here is a brief description of the algorithm:

1. While there are still boxes waiting to be packed, compare the number of items we could pack into one value-restricted box and one weight-restricted box .
2. Pick the box that holds more items and start from the next un-packed item.
3. Repeat above process until all items are packed.

This greedy approach ensures that we always pack more items into one box then move on to the next, minimizing the cost of total boxes.

(b)

The time complexity of this algorithm is and the pseudocode is as follows:

Greedy-Min-Cost (items, V, W):

number\_of\_items = len(items)

i, total\_box = 0, 0

while i < number\_of\_items:

current\_value\_item, current\_weight\_item = i, i

current\_box\_value\_sum, current\_box\_weight\_sum = 0, 0

while (current\_value\_item < number\_of\_items

and current\_box\_value\_sum + items[current\_value\_item].value <= max\_value):

current\_box\_value\_sum += items[current\_value\_item].value

current\_value\_item += 1

while (current\_weight\_item < number\_of\_items

and current\_box\_weight\_sum + i tems[current\_weight\_item].weight <= max\_weight):

current\_box\_weight\_sum += items[current\_weight\_item].weight

current\_weight\_item += 1

if current\_value\_item > current\_weight\_item:

i = current\_value\_item

else:

i = current\_weight\_item

total\_box += 1

return total\_box

The variable initializations take time. Inside the main while loop there are two while loops, which in the worst case will iterate all items in the list, so they will take time. However, since i is assigned to a larger index from above two while loops, all in all every item will be only iterated twice at most. Therefore, the time complexity of this algorithm is .

Problem #4

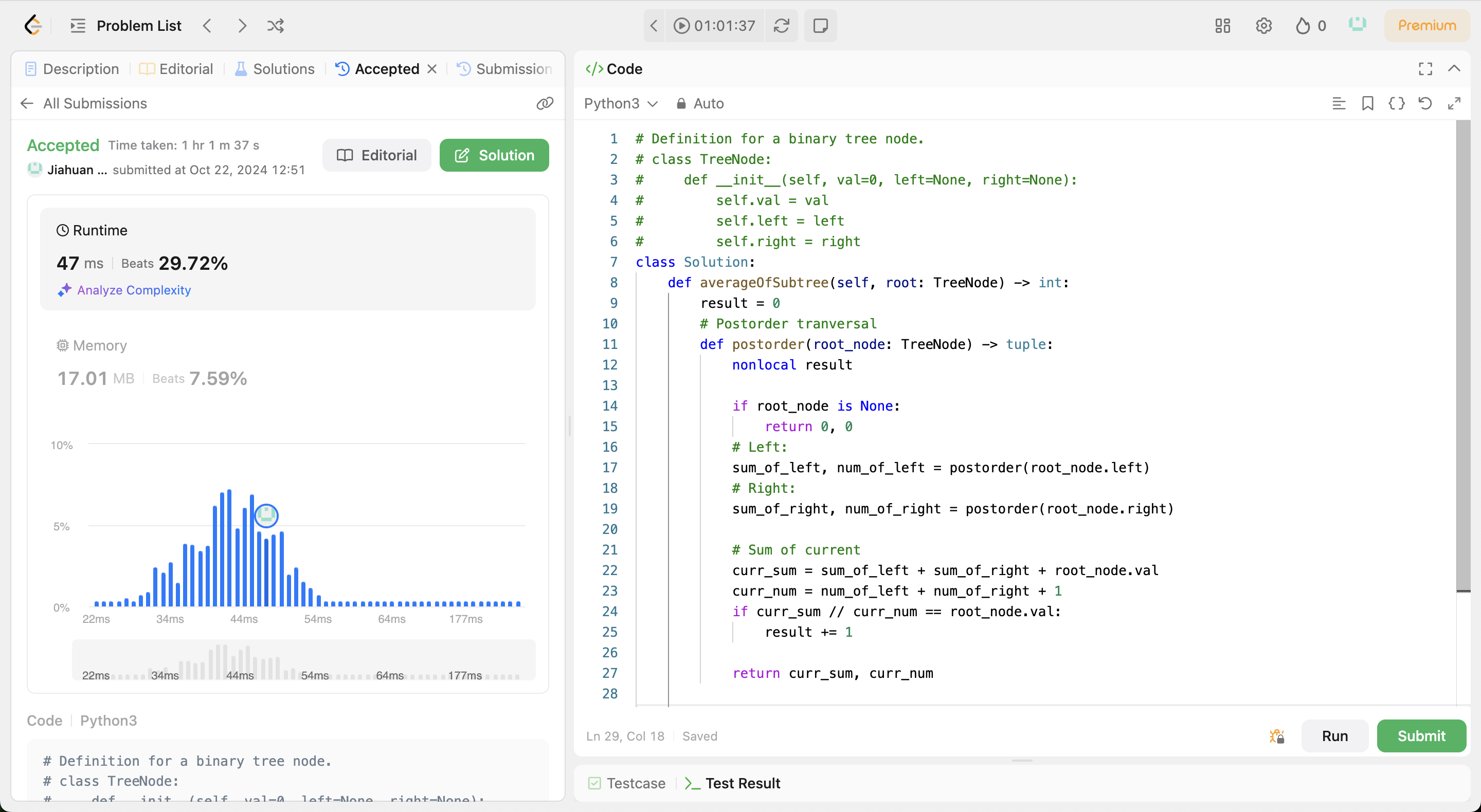
(a)

[**1382. Balance a Binary Search Tree**](https://leetcode.com/problems/balance-a-binary-search-tree/) (Medium), I used 50:41 minutes to solve it.

A screenshot of a computer

Description automatically generated

[**2265. Count Nodes Equal to Average of Subtree**](https://leetcode.com/problems/count-nodes-equal-to-average-of-subtree/) (Medium), I used around 40 minutes to solve it (I forgot to stop the last problem, so the total time here shows 1 hr 1 mins).



(b)

I would like to share my solution with [**2265. Count Nodes Equal to Average of Subtree**](https://leetcode.com/problems/count-nodes-equal-to-average-of-subtree/). At first, I didn’t know how to solve this problem, so I tried to put all nodes into a queue and pop one node a time to check if it has left child and/or right child, which didn’t work at all. Then I thought about three ways of transversal of a graph – the post order transversal is exactly what this problem requires: calculate the sum of one node’s children’s value and itself and divide it by the number of nodes involved. So, I turned to the post order transversal. I didn’t remember it really well, so I still kept my queue. After looking through lines of code, I suddenly thought to myself: the queue didn’t seem useful here, what would happen if I removed it? The answer is nothing. And after I got rid of the confusing queue, the answer seemed obvious: a recursive call to post order transversal of the graph will satisfy this problem, and I finally got it right.

This was actually the second problem I tried for this question in HW4, and honestly, I had practiced DFS problems before, so this kind of problems was not as scary as DP to me. Yet I didn’t really think I could solve this problem at first. Thanks to this question, I had a chance to restart practicing Leetcode problems and review the three basic ways of transversing a graph.

On the other hand, I have never practiced any problems of MST or Shortest Paths, which I should definitely try next time. I am planning to practice more other than assignments.