Problem 0

Problem 1: 許睿尹

Problem 2: 林佑辰

Problem 3: all by myself!

Problem 4: 黃千睿, 李沛宸, and https://web.ntnu.edu.tw/~algo/Graph.html

Problem 5: 王彥宗

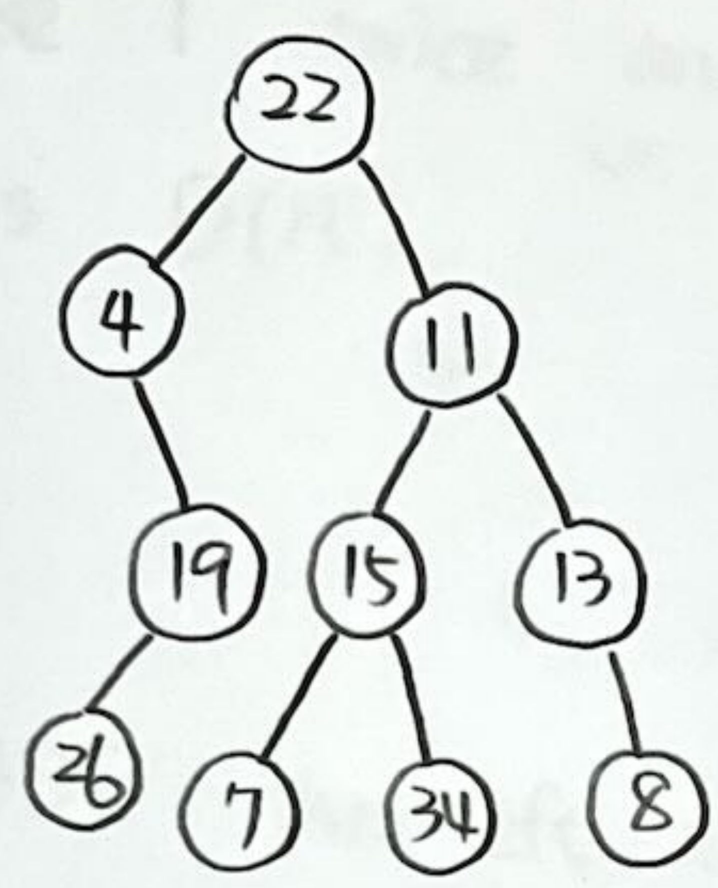
Problem 1 :

1. inorder : L, M, R

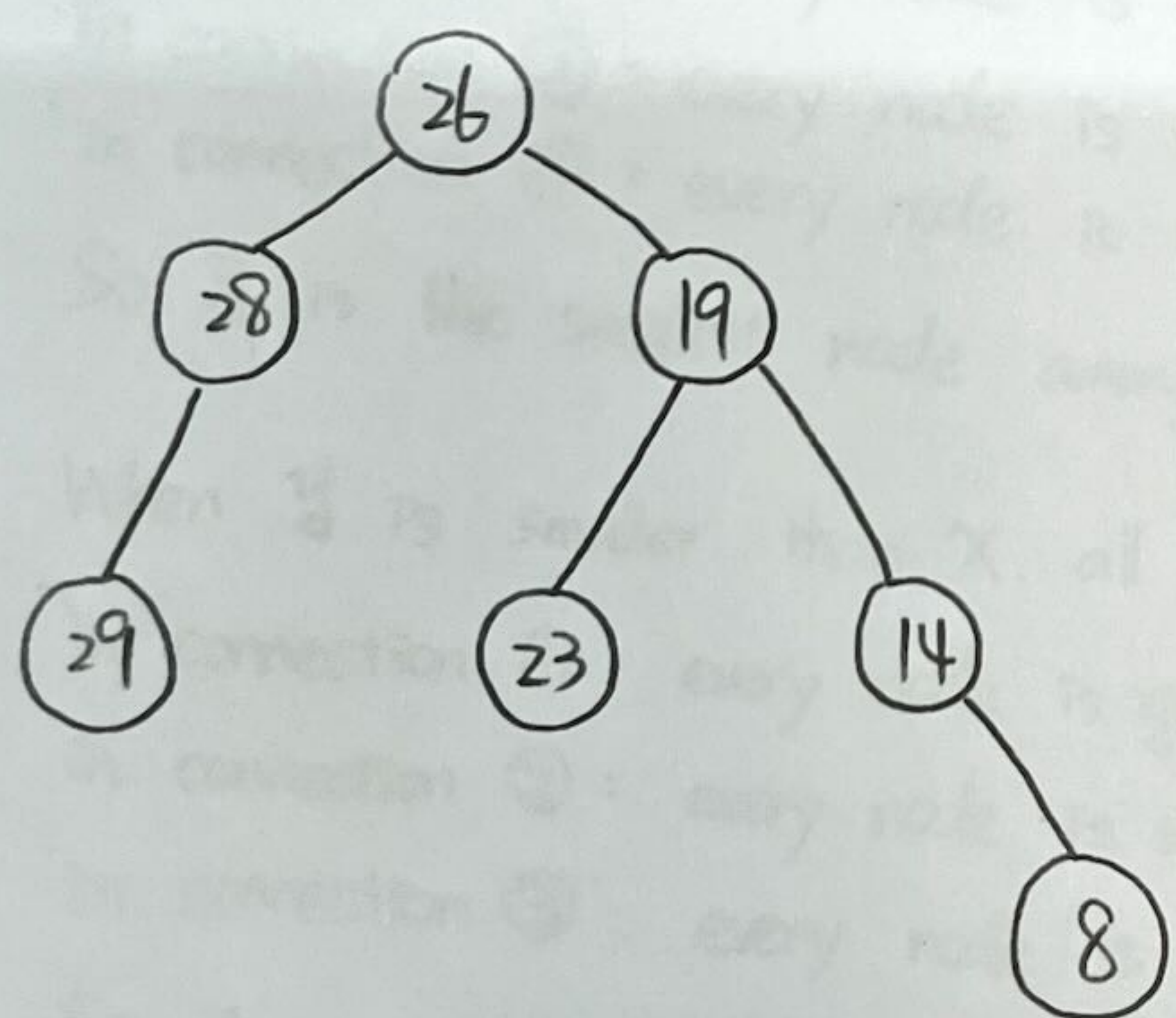
postorder : L, R, M

so we know that the last element of postorder traversal is the parent M, then those elements before M in inorder traversal is the left sub-tree and the elements after M is the right sub-tree.

It's a recursive definition and by applying the definition again and again, we can get the whole tree.



2.



3.

Since T' has the same connections as T and the value of nodes in T' $b_i = a_i + \sum_{v \in S_i} v.val$, $S_i = \{ v, v \in T, v.val > a_i.val \}$

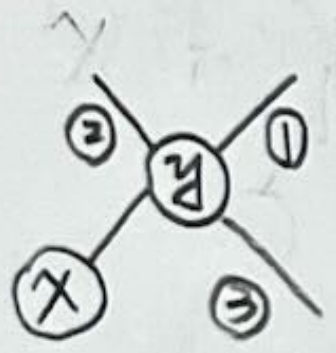
We can first inorder traverse T once and store the values into an array Val. Then we traverse the array from $i = n-1$ to $i = 1$, with every value $Val[i] = Val[i] + Val[i+1]$.

Now we inorder traverse T again and replace every value with $\underbrace{Val[i]}_{i \text{ from } 1 \text{ to } n.}$. Then after the traversal, T will become T'.

Since we only traverse T twice and traverse array Val once, the time complexity is $O(n)$.

4.

The definition of a BST : the left sub-tree is smaller than the parent, and the parent is smaller than the right sub-tree.



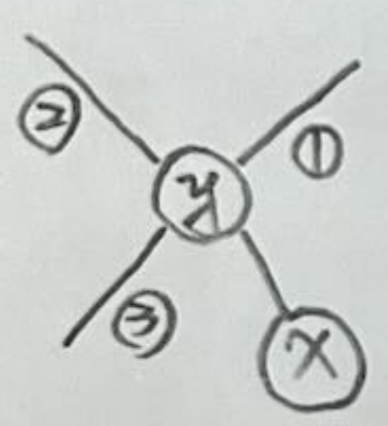
when y is greater than X, all the possible connections look like this,

in connection ① : every node is greater than y.

in connection ② : every node is smaller than X.

in connection ③ : every node is greater than y.

So y is the smallest node among all nodes larger than X.



When y is smaller than X, all the possible connections look like this,

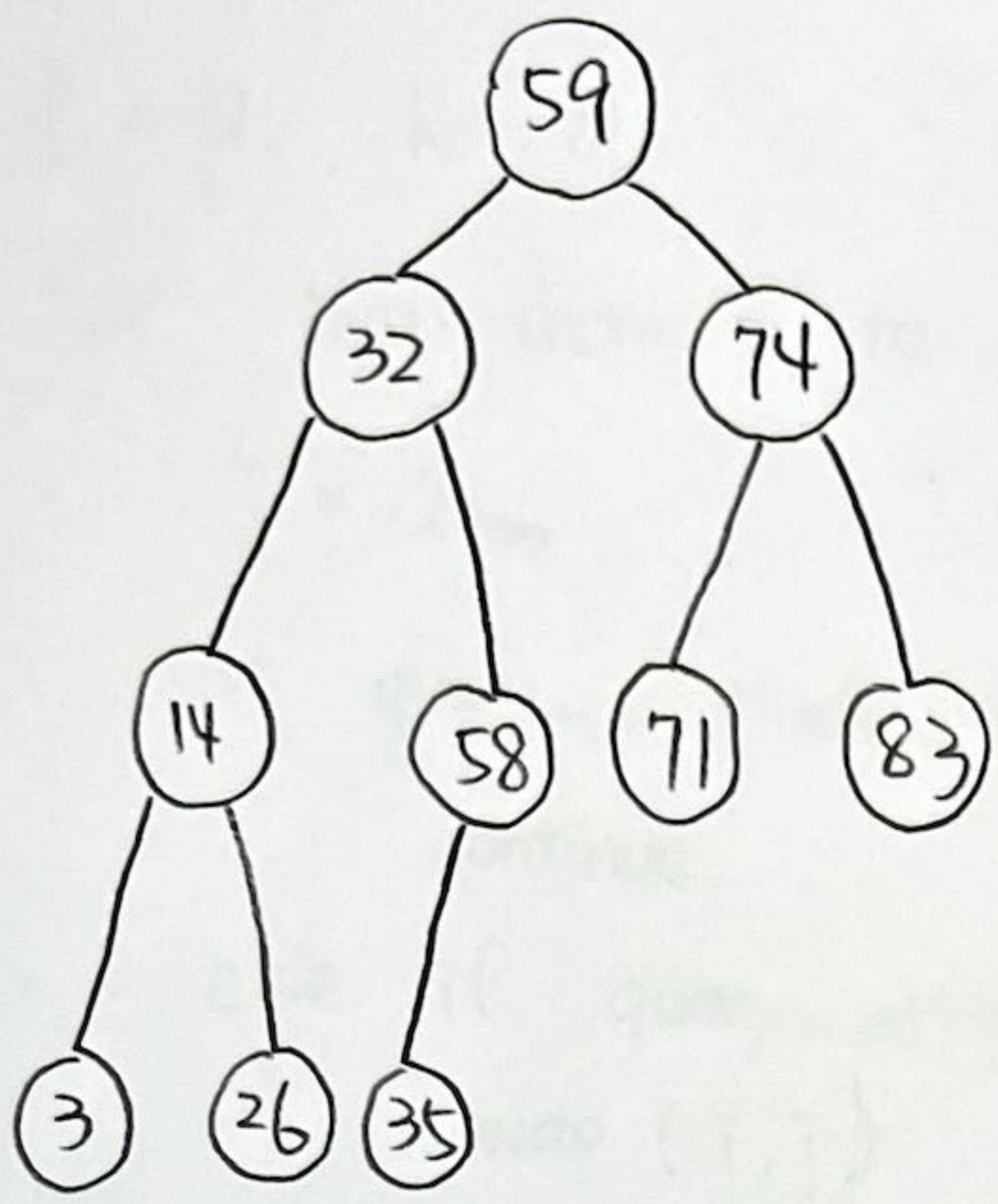
in connection ① : every node is greater than X.

in connection ② : every node is smaller than y.

in connection ③ : every node is smaller than y.

So y is the greatest node among all nodes smaller than X.

5.



6.

```
Max = 1
index = 1
Func_traverse (node, index, Max)
    if index > Max
        Max = index
    if node -> left != NULL
        Func_traverse (node -> left, index × 2, Max)
    if node -> right != NULL
        Func_traverse (node -> right, index × 2 + 1, Max)
    return. Max.
```

the number of X will be Max − n. #

Since the left index = 2 × parent's index and the right index = 2 × parent's index + 1

we can find out the greatest index Max and M − n is the number of X.

Since we traverse all the nodes in the tree, the time complexity is O(n).

Problem 2 :

1.

$i = a_1$, $k = a_2$

```
for tmp from 3 to n
    j = a_tmp
    if query_attitude_value (i, j, k) == true
        continue.
    else if query_attitude_value (j, i, k) == true
        swap (i, j)
    else if query_attitude_value (i, k, j) == true
        swap (j, k)
return (i, k)
```

Since the for loop runs about n times, the time complexity is $O(n)$.

2.

```
for i from 2 to n-1
    left_bnd = 1        left = 1
    right_bnd = i       right = i
    insert ( left, a_{i+1}, right )  ⇐ the function in sub-problem 3.
return.
```

Since the time complexity of the function insert is $O(\log n)$, and the for-loop runs n times, so the total time complexity is $O(n\log n)$.

3.

```
left_bnd = 1        left = 1
right_bnd = n       right = n
Func_insert ( left, a_{n+1}, right )
    mid = floor ( (left + right) / 2 )
    if query_attitude_value ( a_mid, a_{n+1}, a_{mid+1} ) == true
        return (mid, mid+1)  ⇐ insert between mid & mid+1
    else if query_attitude_value ( a_mid, a_{mid+1}, a_{n+1} ) == true
        if mid+1 = right_bnd
                return ( right_bnd, 0 ) ⇐ insert after n
        else
            left = a_{mid+1}
            insert ( left, a_{n+1}, right )
    else
        if mid = left_bnd
                return (0, left_bnd) ⇐ insert before 1
        else
            right = a_mid
            insert ( left, a_{n+1}, right )
    return.
```

Since everytime we run the function, we can reduce the range by a half, the time complexity is $O(\log n)$.

5.

14

6.

Sorts all presentation groups such that their attitude value are monotonic.

$a_1, a_2, a_3, a_4 \cdots$

Sorts all presentation groups such that their terrible value are monotonic.

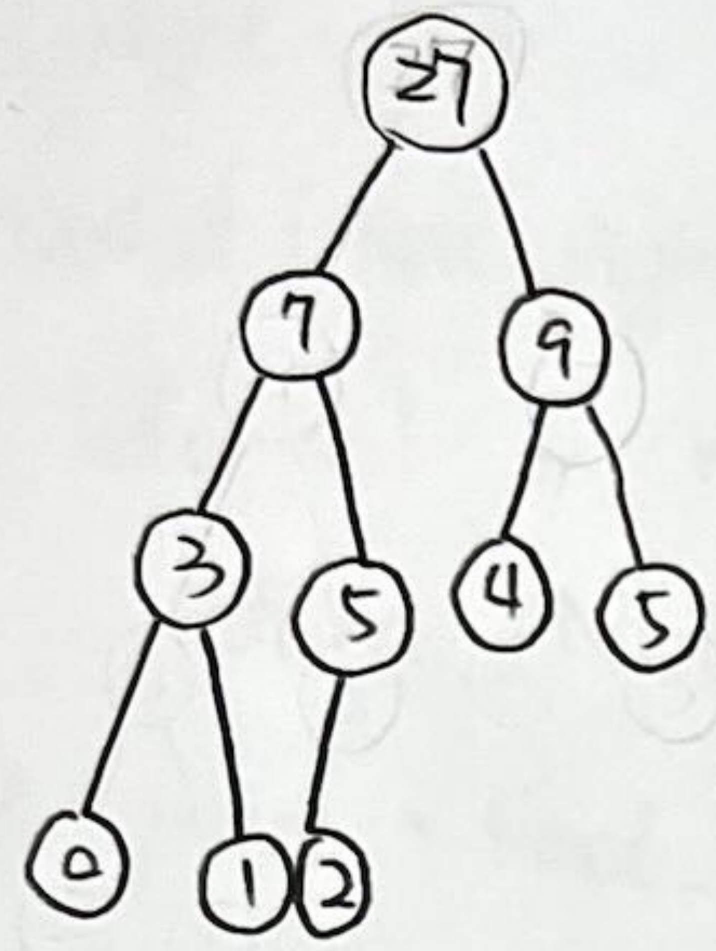
$t_1, t_2, t_3, t_4 \cdots$

lists all the possible sets of $group_i$, $group_j$, $group_k$ and check whether they are good triplets.

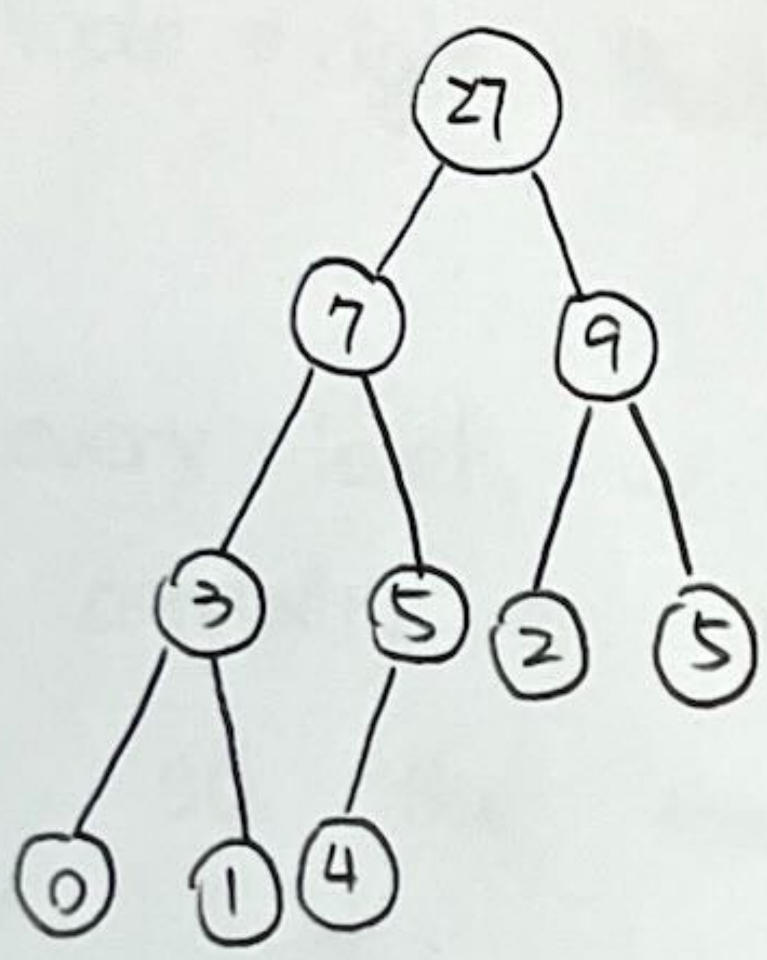
Since we only call the functions when sorting, the time complexity is $O(n\log n)$. #

Problem 3:

1.



2.



3.

```
left = 1   right = len
Func_Build ( left, right )
    if   left < 1   or   right > len   or   left > right
          return   NULL.
    Node → value = Find_Max ( left , right ) ← find the maximum from arr[left] to
                                                arr[right]
    Node → index = Max_index .
    Node → left = Build ( left, Max_index - 1 )
    Node → right = Build ( Max_index + 1 , right )
```

In every level, we will traverse at most n elements, so the time complexity is $O(n)$. And there are h levels in the tree, so the total time complexity is $O(n \cdot h)$.

4.

```
Node = root
while ( Node → index != index )
      if    index  >  Node → index
               Node  =  Node → right
      else
               Node  =  Node → left
return  Node → value.
```

In the worst case, we will traverse from the top to the bottom, and the depth is $O(\log n)$, so the time complexity is $O(\log n)$.

5.

use the tree built in sub-problem 3.

```
Node = root
while (!(Node -> index > left && Node -> index < right))
    if Node -> index < left
        Node = Node -> left
    else
        Node = Node -> right
return Node -> value.
```

In the worst case, we will traverse from the top to the bottom, and the depth is $O(\log n)$, the time complexity is $O(\log n.)$

6.

```
Node = root        cnt = 0
while (Node != NULL)
    building [cnt] = Node -> index
    height [cnt] = Node -> value
    Cnt ++
    Node = Node -> left
for i from cnt-1 to 0
    print building[i], height[i].
```

Since we traverse from the top to the bottom, and the depth complexity is $O(\log n)$, the time complexity is $O(\log n)$.