hw2

- Problem 0 :

1.

 1-6:

 find the number of nodes of the binary tree:

 https://algorithms.tutorialhorizon.com/count-the-number-of-nodes-in-a-given-binary-tree/

 others are by myself

2.

 2-1 : by myself

 2-2: R09944059 劉懋澤

 2-3: R09944059 劉懋澤

 2-4: by myself

 2-5: R09944059 劉懋澤

 2-6: R09944059 劉懋澤

3.

 3-1: by myself

 3-2: by myself

 3-3: R10525102 李孟哲

 https://www.geeksforgeeks.org/cartesian-tree/

 3-4: R10525102 李孟哲

 3-5: R10525102 李孟哲

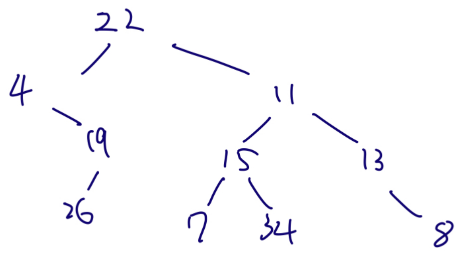
4.

 https://www.geeksforgeeks.org/c-program-to-implement-dfs-traversal-using-adjacency-matrix-in-a-given-graph/

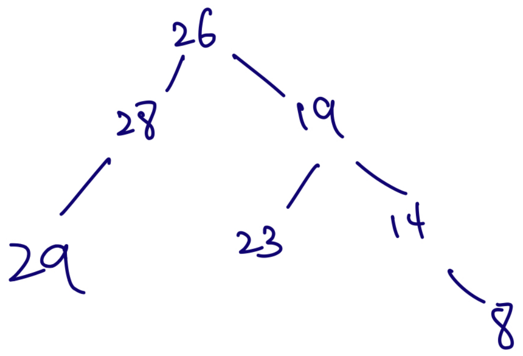
 R10525102 李孟哲

5. https://www.geeksforgeeks.org/stack-data-structure-introduction-program/
   https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
   R09944059 劉懋澤

- Problem 1 :
  1.
     We can make the construction process of the tree into different stages. In each stage, we determine the root of the subtree by choosing the last number of the postorder traversal. By pointing out the root of the subtree in the inorder traversal, you can tell the members of the left subtree and the right subtree. Repeat the process, and you will get the construction of the tree.



  2.



  3.

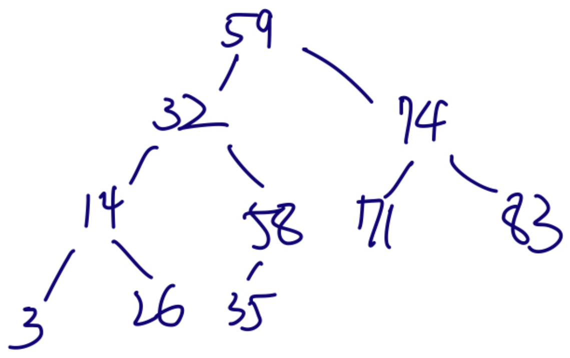
The larger nodes in T will be smaller in T' since there are less numbers of nodes will be added to the value of the larger nodes. In other words, the relationship among nodes will be opposite to the BST; If there is a node x, then x > x.right and x < x.left. According to the logic above, we can calculate the values of every nodes with the following function.

```
// Since x < x.left, we will have a value to add to the x.left.value
def function foo(x, to_be_added)
  x.value = x.value + to_be_added
  if (x.right ≠ null)
    x.value = x.value + foo(x.right)
  if (x.left ≠ null)
    return foo(x.left, x.value)
  return x.value

//Since we traverse from the right, we do not need to add value to the root.right
foo(root, 0)
```

4. If y is not the smallest node among all nodes larger than x, there is at least one node in the right subtree of x which contradicts to the statement of the question. Similarly, if y is not the largest node among all nodes smaller than x, there is at least one node in the left subtree of x which contradicts to the statement of the question. Through the descriptions above, we can prove that the answer is true.

5.



6.

```
//Time complexity is O(n) since we only visit every nodes once
def function find_the_largest(x, index)
  if (x.left ≠ null)
    a = find_the_largest(x.left, 2 * index)
  if (x.right ≠ null)
    b= find_the_largest(x.right, 2 * index + 1)

  if a > b
    return a
  else if (a < b)
    return b
  else
    return index

//Time complexity is O(n) since we only visit every nodes once
def function countNode(x)
  if (x == null)
    return 0
  return 1 + countNode(x.left) + countNode(x.right)


//Let the index of the root = 1.
//Get the number of wasted  nodes.
find_the_largest(root, 1) - countNode(root)
//Since we only run both of the functions once, the time complexity is O(n)
```

- Problem 2 :

1.

a. First, we put all groups into an array which can be represented as

   x = {a1, a2, a3, ... ,an}

b. for i = 1 to n-2, we put the permutation of index_of_a_of(x[i]), index_of_a_of(x[i+1]), index_of_a_of(x[i+2]) into query_attitude_value() to query til we get the ‘true’ return from the function. Once the ith true return, you have to make the (i+1)th set of queries return true with the same sorting order as the previous query set.

   e.g. If in the first set of queries, we get true when we query query_attitude_value(2, 1, 3); Then in the second set of queries, if a4 is larger than both of a1 and a3, or smaller than both of a1, a3, we should choose the true return from the query_attitude_value(1, 3, 4) instead of query_attitude_value(4, 3, 1). That is, to keep the sorting order stay the same through the whole process.

c. Once we finish step b, the x[n] will be the group which gets either the largest or the smallest value among groups.

d. Reverse the array, and repeat the process of step b with using the index_of_a_of(x[1]) as the first input in the first set of queries. Once we finish the process, the x[n] will be the group which gets either the large or the smallest value among groups that is opposite to the x[n] we get in the step c.

e. Return x[n] of steps c and d.

2.

a. Through mechanism of the previous question, we can get two groups which are the group with largest value and the group with the smallest value; We randomly choose one group, and let the group be B.

b. To sort all presentation groups such that their attitude values are monotonic, we divide array x of the previous question into n runs with normal merge sort mechanism.

c. Unlike the normal merge sort, we use other way to merge runs. In each merge, we use query_attitude_value() to compare values; However, we always put B in the first input of query_attitude_value() to make sure that the permutation of the groups in every query stay in the same sorting logic since B is the group which gets either the largest or the smallest value among groups. That is, makes each run stay monotonic.

d. After merging runs with the mechanism in step c, we will get an array in monotonic order. Since the normal version of merge sort is O(nlogn), we can say that our method is O(nlogn) as well because the only difference between these two methods is the mechanism of comparing numbers, and both of the mechanisms are O(1), which makes no difference to their time complexity.

3.

a. We use the concept of binary search to find the proper position for the an+1. However, the mechanism we use to compare values are different from the normal version of binary search.

b. When we determine whether we should put a(n+1) group to the right or the left, we use query_attitude_value(). If the middle index we choose is t, then we will use the permutation of a(n+1), at, and a(t+1) to query the query_attitude_value() until we get true with certain permutation in a strict rule.

   e.g. If we compare (an+1), at, and a(t+1), there will be three possible permutation that make the output of the query_attitude_value() true.

   When query_attitude_value(at, a(n+1), a(t+1)) == true, we know that it’s the right place for a(n+1).

   When query_attitude_value(at, a(t+1), a(n+1)) == true, we know that we should move a(n+1) to the middle of the left half to do the next query.

   When query_attitude_value(a(n+1), at, a(t+1)) == true, we know that we should move a(n+1) to the middle of the right half to do the next query.

   And even if something like query_attitude_value(a(n+1), a(t+1), at) return true, we won’t choose this permutation since the monotonic way is different from the original version.

c. If the query_attitude_value(at, a(n+1), a(t+1)) ≠ true, we will get the next middle index to try the next query. Repeat steps b, c til query_attitude_value(at, a(n+1), a(t+1)) == true.

   Since binary search is O(logn), our method is O(n) as well since the only difference between these two method is the mechanism of determining the next middle index, and both of the mechanisms are O(n).

4.

There are n! possible results if we sort n data. If we use decision tree to do sort, and regard the leafs as possible comparison result, there will be n! leafs. Since the tree is binary tree, the height ≥ ceiling(log(n!)) + 1. In a decision tree, number of comparison = height - 1 ≥ ceiling(log(n!)) = O(nlogn)

Since our algorithm is based on comparison, it’s impossible to find an algorithm with o(nlogn) query complexity to sort the groups by their attitude values monotonically.

5.

14 sets

6.

a. Use algorithm of 2-2 to sort groups with query_attitude_value to get an monotonic array. Process of this step is O(nlogn)

b. Since we are not sure about the actual values, we can not tell if two adjacent groups have same value.

Therefore, we do

```
1. Query first three groups of the array.
    if query_attitude_value(a1, a2, a3) == query_attitude_value(a2, a1, a3)
        then a1 == a2
    else
        a1 ≠ a2
2. for i from 1 to n-2 , we use ai, a(i+1), a(i+2) and  ai, a(i+2), a(i+1) to query.
    With this method, we can know that whether every two adjacent groups have same value.
Process of this step is O(n)
```

c. With steps above, we can do permutation to get valid groups since we know the relations between every groups even if adjacent groups have same value, and store the result of permutation. Since we don’t need to query in this step, we don’t count the time complexity of this step.

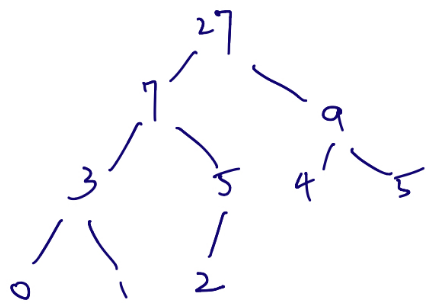
d. Repeat step a, step b, and c again with query_terrible_value() to get another result of permutations. The time complexity if as same as step a and step b

e. Compare results of query_attitude_value() and query_terrible_value() to choose permutations which is valid on both of the functions. The result will be the answer of this question. Since we don't need to query in this step, we don't count the time complexity of this step.

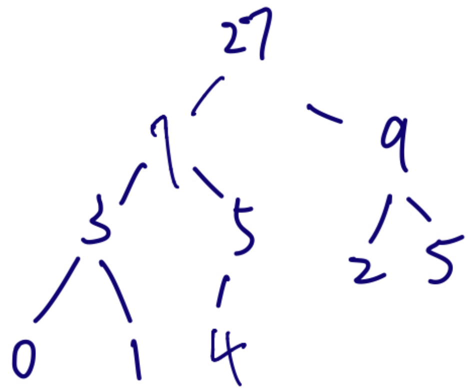
Time complexity is O(nlogn) + O(n) = O(nlogn)

- Problem 3 :

1.



2.



3.

```
def function buildCartesianTree (heights, n)

  Initialize parent[n],leftchild[n],rightchild[n] and make all of the elements -1
  root = 0
  last

  for i = 1 to n-1
    last = i-1
    rightchild[i] = -1

    while heights[last] <= heights[i] and last != root
      last = parent[last]

    if (heights[last] <= heights[i])
      parent[root] = i
      leftchild[i] = root
      root = i

    else if rightchild[last] == -1
      rightchild[last] = i
      parent[i] = last
      leftchild[i] = -1

    else
      parent[rightchild[last]] = i
      leftchild[i] = rightchild[last]
      rightchild[last] = i
      parent[i] = last

  parent[root] = -1

  then you can check the relationship between the nodes with three arrays,
  which are parent, leftchild, and rightchild
```

For we do for loop for every nodes only once and find relationship between a node and its child tree which is O(n), the total time complexity is O(n .h).

4.

```
Assume the root is given

def function find_index(heights, index, root)
  If root = index
    return heights[root]

  else if root > index
    return find_index(heights, index, heights[root].left)

  else
    return find_index(heights, index, heights[root].right)
```

Since we search the given root’s child not many times but once, the time complexity is O(logn) which equals to the height of the tree.

5.

```
The value of the root of every tree is supposed to be the largest value of the tree

def function find_largest(heights, root, left, right)
  if root is within left and right
    return heights[root]

  else if root > right
    return find_largest(heights, heights[root].left, left, right)

  else
    return find_largest(heights, heights[root].right, left, right)
```

Just like the previous question, since we search the given root’s child not many times but once, the time complexity is O(logn) which equals to the height of the tree.

6.

```
def function left_hand_sideview(heights, root, i, arr)
  if heights[root].left =! null
    arr[i] = heights[root] - heights[root].left.value
    i++
    left_hand_sideview(heights, heights[root].left, i, arr)

  else
    arr[i] = heights[root]

main (heights, root, i, n)
  i = 0
  init arr[n]
  left_hand_sideview(heights, root, i, arr)
  from the last to the first, print the elements of arr
```

Since we search the given root’s child not many times but once, the time complexity is O(logn) which equals to the height of the tree.