第二部分

2.1.二叉树

94. 二叉树的中序遍历

```python
# Definition for a binary tree node.
# 94. 二叉树的中序遍历
from typing import List, Optional


class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []
        self.recur_mid(root, result)
        return result

    def recur_mid(self, root, result):
        if root is None:
            return
        self.recur_mid(root.left, result)
        result.append(root.val)
        self.recur_mid(root.right, result)


# 非递归版本
class Solution1:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        result = []
        tmp = [root]
        while len(tmp) > 0:
            node = tmp.pop(0)
            new_tmp = []
            if node.left is not None:
                new_tmp.append(node.left)
                node.left = None
                new_tmp.append(node)
                tmp = new_tmp + tmp
                continue
            else:
                result.append(node.val)
            if node.right is not None:
```

```
                new_tmp.append(node.right)
                tmp = new_tmp + tmp
                continue
        return result


def testCase():
    tn1 = TreeNode(val=1)
    tn2 = TreeNode(val=2)
    tn3 = TreeNode(val=3)
    tn1.right = tn2
    tn2.left = tn3

    sol = Solution1()
    ret = sol.inorderTraversal(tn1)
    print(ret)

if __name__ == "__main__":
    testCase()
```

100. 相同的树

```
# Definition for a binary tree node.
# 100. 相同的树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if p is None and q is None:
            return True
        elif (p is None and q is not None) or (p is not None and q is None):
            return False
        else:
            if p.val == q.val:
                return self.isSameTree(p.left, q.left) and
self.isSameTree(p.right, q.right)
            else:
                return False
```

102. 二叉树的层序遍历

```
# Definition for a binary tree node.
# 102. 二叉树的层序遍历
from typing import List
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_line(node_list, result)
        return result

    def recur_line(self, node_list, result):
        if len(node_list) == 0:
            return
        tmp_list = []
        lines = []
        for node in node_list:
            if node.left is not None:
                tmp_list.append(node.left)
            if node.right is not None:
                tmp_list.append(node.right)
            lines.append(node.val)
        result.append(lines)
        self.recur_line(tmp_list, result)
```

103. 二叉树的锯齿形层序遍历

```
# Definition for a binary tree node.
# 103. 二叉树的锯齿形层序遍历
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        node_list = [root]
        result = []
        self.recur_node(node_list, result, 1)
        return result

    def recur_node(self, node_list, result, depth):
```

```
        if len(node_list) == 0:
            return
        sub_nodes = []
        lines = []
        for node in node_list:
            if node.left is not None:
                sub_nodes.append(node.left)
            if node.right is not None:
                sub_nodes.append(node.right)
            lines.append(node.val)
        if depth % 2 == 0:
            lines = lines[::-1]
        result.append(lines)
        self.recur_node(sub_nodes, result, depth+1)
```

104.二叉树最大深度

```
# Definition for a binary tree node.
# 104.二叉树最大深度
from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        return self.getDepth(root)

    def getDepth(self, root):
        if root is None:
            return 0
        return max(self.getDepth(root.left), self.getDepth(root.right)) + 1

def testCase():
    tn1 = TreeNode(val=1)
    tn2 = TreeNode(val=9)
    tn3 = TreeNode(val=20)
    tn4 = TreeNode(val=15)
    tn5 = TreeNode(val=7)
    tn1.left = tn2
    tn1.right = tn3
    tn3.left = tn4
    tn3.right = tn5
    sol = Solution()
    ret = sol.maxDepth(tn1)
    print(ret)

if __name__ == "__main__":
    testCase()
```

105.从前序遍历和中序遍历构造二叉树

```
# Definition for a binary tree node.
# 105.从前序遍历和中序遍历构造二叉树
from typing import List
import sys
sys.setrecursionlimit(100000)

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if len(preorder) == 0:
            return None
        val = preorder.pop(0)
        node = TreeNode(val)
        in_idx = inorder.index(val)
        left_inorder = inorder[:in_idx]
        right_inorder = inorder[in_idx+1:]
        left_preorder, right_preorder = self.getLeftRight(preorder, left_inorder)
        node.left = self.buildTree(left_preorder, left_inorder)
        node.right = self.buildTree(right_preorder, right_inorder)
        return node

    def getLeftRight(self, preorder, left_inorder):
        left_preorder = []
        right_preorder = []
        left_inorder_dict = {n:1 for n in left_inorder}
        for o in preorder:
            if o in left_inorder_dict:
                left_preorder.append(o)
            else:
                right_preorder.append(o)
        return left_preorder, right_preorder


def testCase():
    import json
    import time
    line_list = []
    with open("./test_case/105.txt") as f:
        for line in f.readlines():
            n_list = json.loads(line.strip())
            line_list.append([int(n) for n in n_list])
    preorder = line_list[0]
    inorder = line_list[1]

    # print(len(preorder))
```

```
    # print(len(inorder))
    t1 = time.time()
    sol = Solution()
    sol.buildTree(preorder, inorder)
    t2 = time.time()
    print((t2-t1)*1000)

if __name__ == "__main__":
    testCase()
```

106. 从中序与后序遍历序列构造二叉树

```
# Definition for a binary tree node.
from typing import List

# 106. 从中序与后序遍历序列构造二叉树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        if len(postorder) == 0:
            return None
        val = postorder.pop()
        node = TreeNode(val)
        inorder_idx = inorder.index(val)
        left_inorder = inorder[:inorder_idx]
        right_inorder = inorder[inorder_idx+1:]
        left_postorder, right_postorder = self.getPostOrderLeftRight(postorder,
left_inorder)
        node.left = self.buildTree(left_inorder, left_postorder)
        node.right = self.buildTree(right_inorder, right_postorder)
        return node

    def getPostOrderLeftRight(self, postorder, left_inorder):
        left_inorder_dict = {i:1 for i in left_inorder}
        left_postorder, right_postorder = [] , []
        for o in postorder:
            if o in left_inorder_dict:
                left_postorder.append(o)
            else:
                right_postorder.append(o)
        return left_postorder, right_postorder
```

654. 最大二叉树

```python
# Definition for a binary tree node.

# 654. 最大二叉树
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> TreeNode:
        if len(nums) == 0:
            return None
        max_val, max_idx = self.getMaxIdx(nums)
        print(max_val)
        left_nums = nums[:max_idx]
        right_nums = nums[max_idx+1:]
        node = TreeNode(max_val)
        node.left = self.constructMaximumBinaryTree(left_nums)
        node.right = self.constructMaximumBinaryTree(right_nums)
        return node


    def getMaxIdx(self, nums):
        max_val, max_idx = nums[0], 0
        for i in range(1, len(nums)):
            if max_val < nums[i]:
                max_val = nums[i]
                max_idx = i
        return max_val, max_idx

def testCase():
    nums = [3,2,1,6,0,5]
    sol = Solution()
    node = sol.constructMaximumBinaryTree(nums)
    print_node([node])

if __name__ == "__main__":
    testCase()
```

107. 二叉树的层序遍历 II

```python
# Definition for a binary tree node.
from typing import List

# 107. 二叉树的层序遍历 II
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
```

```
        self.left = left
        self.right = right

class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []
        node_list = [root]
        result = []
        self.recur_lines(node_list, result)
        return result

    def recur_lines(self, node_list, res):
        if len(node_list) == 0:
            return
        new_nodes = []
        tmp = []
        for node in node_list:
            if node.left is not None:
                new_nodes.append(node.left)
            if node.right is not None:
                new_nodes.append(node.right)
            tmp.append(node.val)
        self.recur_lines(new_nodes, res)
        res.append(tmp)
        return
```

111. 二叉树的最小深度

```
# Definition for a binary tree node.

# 111. 二叉树的最小深度
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def minDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        return self.getDepth(root)

    def getDepth(self, root):
        if root.left is None and root.right is None:
            return 1
        elif root.left is None and root.right is not None:
            return self.getDepth(root.right) + 1
        elif root.left is not None and root.right is None:
            return self.getDepth(root.left) + 1
```

```
        elif root.left is not None and root.right is not None:
            return min(self.getDepth(root.left), self.getDepth(root.right)) + 1
```

114. 二叉树展开为链表

```
# Definition for a binary tree node.

# 114. 二叉树展开为链表
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        if root is None:
            return []

        node_list = [root]
        while len(node_list) > 0:
            node = node_list.pop(0)
            tmp_nodes = []
            node_left, node_right = node.left , node.right
            node.left = None
            if node_left is not None:
                node.right = node_left
                if node_right is not None:
                    tmp_nodes = [node_left, node_right]
                else:
                    tmp_nodes = [node_left]
                tmp_nodes.extend(node_list)
                node_list = tmp_nodes
                continue
            elif node_right  is not None:
                tmp_nodes = [node_right]
                tmp_nodes.extend(node_list)
                node_list = tmp_nodes
                continue
            elif node_right is None:
                if len(node_list) > 0:
                    node.right = node_list[0]
                else:
                    break
        return root
```

116. 填充每个节点的下一个右侧节点指针

```
"""
# Definition for a Node.

"""
# 116. 填充每个节点的下一个右侧节点指针
from typing import Optional

class Node:
    def __init__(self, val: int = 0, left: 'Node' = None, right: 'Node' = None,
next: 'Node' = None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next

class Solution:
    def connect(self, root: Optional[Node]) -> Optional[Node]:
        if root is None:
            return None
        node_list = [root]
        while len(node_list) > 0:
            nodes = []
            for i in range(len(node_list)):
                if i + 1 < len(node_list):
                    next_node = node_list[i+1]
                else:
                    next_node = None
                node_list[i].next = next_node
                if node_list[i].left is not None:
                    nodes.append(node_list[i].left)
                if node_list[i].right is not None:
                    nodes.append(node_list[i].right)
            node_list = nodes
        return root
```

226. 翻转二叉树

```
# Definition for a binary tree node.
# 226. 翻转二叉树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if root is None:
            return root
        node_list = [root]
```

```
        self.invertTreeHelp(node_list)
        return root

    def invertTreeHelp(self, node_list):
        if len(node_list) == 0:
            return
        tmp = []
        print(node_list)
        for node in node_list:
            left_node, right_node = node.left, node.right
            node.left, node.right = right_node, left_node
            if left_node is not None:
                tmp.append(left_node)
            if right_node is not None:
                tmp.append(right_node)
        node_list = tmp
        self.invertTreeHelp(node_list)
```

144.二叉树的前序遍历

```
# Definition for a binary tree node.
from typing import List, Optional

# 144.二叉树的前序遍历
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# 递归版本
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        result = []
        self.preorder(root, result)
        return result

    def preorder(self, root, result):
        if root is None:
            return
        result.append(root.val)
        self.preorder(root.left, result)
        self.preorder(root.right, result)

# 非递归版本
class Solution1:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        node_list = [root]
        result = []
```

```
        while len(node_list) > 0:
            node = node_list.pop(0)
            tmp = []
            result.append(node.val)
            if node.left is not None:
                tmp.append(node.left)
            if node.right is not None:
                tmp.append(node.right)
            node_list = tmp + node_list
        return result
```

145.二叉树的后序遍历

```
# Definition for a binary tree node.
from typing import List, Optional

# 145.二叉树的后序遍历
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# 递归版本
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        result = []
        self.postOrder(root, result)
        return result

    def postOrder(self, root, result):
        if root is None:
            return []
        self.postOrder(root.left, result)
        self.postOrder(root.right, result)
        result.append(root.val)

# 非递归版本
class Solution1:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        result = []
        node_list = [root]
        while len(node_list) > 0:
            print(node_list)
            node = node_list.pop(0)
            if node.left is None and node.right is None:
                result.append(node.val)
            else:
```

```
                tmp = []
                if node.left is not None:
                    tmp.append(node.left)
                if node.right is not None:
                    tmp.append(node.right)
                node.left, node.right = None, None
                tmp.append(node)
                node_list = tmp + node_list
        return  result
```

222. 完全二叉树的节点个数

```
# Definition for a binary tree node.

# 222. 完全二叉树的节点个数
# 等比数列公式 Sn = a1 * (1-q**n)/(1-q), q为比值
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def countNodes(self, root: TreeNode) -> int:
        left, right = root, root
        hight_left, hight_right = 0, 0
        while left is not None:
            left = left.left
            hight_left += 1
        while right is not None:
            right = right.right
            hight_right += 1
        if hight_left == hight_right:
            return 2 ** hight_left - 1
        return 1 + self.countNodes(root.left) + self.countNodes(root.right)

class Solution1:
    def countNodes(self, root: TreeNode) -> int:
        if root is None:
            return 0
        if root.left is None and root.right is None:
            return 1
        elif root.left is None and root.right is not None:
            return 1 + self.countNodes(root.right)
        elif root.left is not None and root.right is None:
            return 1 + self.countNodes(root.left)
        elif root.left is not None and root.right is not None:
            return 1 + self.countNodes(root.left) + self.countNodes(root.right)
```

236. 二叉树的最近公共祖先

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

# 236. 二叉树的最近公共祖先
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        if root is None:
            return None
        # 这里处理p=5 q=4的情况
        if root.val == p.val or root.val == q.val:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if left is not None and right is not None:
            return root
        if left is None and right is None:
            return None
        if left is None:
            return right
        else:
            return left

# 会超时 因为递归在反复计算
class Solution1:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        if root is None:
            return root
        if (root.val == p.val and self.is_exists(root, q)) or (root.val == q.val
and self.is_exists(root, p)):
            return root
        left_p, right_p = self.is_exists(root.left, p), self.is_exists(root.righ,
p)
        left_q, right_q = self.is_exists(root.left, q), self.is_exists(root.right,
q)

        if (left_p and right_q) or (left_q and right_p):
            return root
        elif left_p and left_q:
            return self.lowestCommonAncestor(root.left, p, q)
        elif right_p and right_q:
            return self.lowestCommonAncestor(root.right, p, q)
        else:
            return None

    def is_exists(self, root, node):
        if root is None:
```

```
            return False
        if root.val == node.val:
            return True
        return self.is_exists(root.left, node) or self.is_exists(root.right, node)
```

297. 二叉树的序列化与反序列化

```
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
# 297. 二叉树的序列化与反序列化
class Codec:
    def __init__(self):
        self.sep=','
        self.null="#"

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        result_list = []
        self.serHelp(root, result_list)
        return self.sep.join(result_list)

    def serHelp(self, root, result_list):
        if root is None:
            result_list.append(self.null)
            return
        result_list.append(str(root.val))
        self.serHelp(root.left, result_list)
        self.serHelp(root.right, result_list)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        data_list = data.split(self.sep)
        return self.desHelp(data_list)

    def desHelp(self, data_list):
        if len(data_list) == 0:
            return None
        first = data_list.pop(0)
        if first == self.null:
```

```
            return None
        node = TreeNode(int(first))
        node.left = self.desHelp(data_list)
        node.right = self.desHelp(data_list)
        return node

# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))
```

501. 二叉搜索树中的众数

```
# Definition for a binary tree node.
# 501. 二叉搜索树中的众数
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def __init__(self):
        self.pre_node = None
        self.cur_count = 0
        self.max_count = 0
        self.mode_list = []

    def findMode(self, root: TreeNode) -> List[int]:
        self.traverse(root)
        return self.mode_list

    def traverse(self, root):
        if root is None:
            return
        self.traverse(root.left)
        # 开头
        if self.pre_node is None:
            self.cur_count = 1
            self.max_count = 1
            self.mode_list.append(root.val)
        else:
            if root.val == self.pre_node.val:
                self.cur_count += 1
                if self.cur_count == self.max_count:
                    self.mode_list.append(root.val)
                elif self.cur_count > self.max_count:
                    self.mode_list.clear()
                    self.max_count = self.cur_count
```

```
                    self.mode_list.append(root.val)
            elif root.val != self.pre_node.val:
                self.cur_count = 1
                if self.cur_count == self.max_count:
                    self.mode_list.append(root.val)
        self.pre_node = root
        self.traverse(root.right)
```

543. 二叉树的直径

```
# Definition for a binary tree node.
# 543. 二叉树的直径
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.max_len = 0
        self.getDepth(root)
        return self.max_len

    def getDepth(self, root):
        if root is None:
            return 0
        left_depth, right_depth = self.getDepth(root.left),
self.getDepth(root.right)
        self.max_len = max(left_depth + right_depth, self.max_len)
        return max(left_depth, right_depth) + 1
```

559. N 叉树的最大深度

```
"""
# Definition for a Node.
"""
# 559. N 叉树的最大深度
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class Solution:
    def maxDepth(self, root: 'Node') -> int:
        if root is None:
            return 0
        depth = 0
        for node in root.children:
```

```
            depth =  max(self.maxDepth(node), depth)
        return depth + 1
```

589. N 叉树的前序遍历

```
from typing import List

# 589. N 叉树的前序遍历
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children

class Solution:
    def preorder(self, root: 'Node') -> List[int]:
        if root is None:
            return []
        self.result = []
        self.preoderHelp(root)
        return self.result

    def preoderHelp(self, root):
        if root is None:
            return
        self.result.append(root.val)
        if root.children is not None:
            for node in root.children:
                self.preoderHelp(node)

# 非递归遍历
class Solution1:
    def preorder(self, root: 'Node') -> List[int]:
        if root is None:
            return []
        node_list = [root]
        result = []
        while len(node_list) > 0:
            node = node_list.pop(0)
            result.append(node.val)
            if node.children is not None:
                node_list = node.children + node_list
        return result
```

652. 寻找重复的子树

```
# Definition for a binary tree node.
# 652. 寻找重复的子树
from typing import Optional
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def findDuplicateSubtrees(self, root: Optional[TreeNode]) ->
List[Optional[TreeNode]]:
        if root is None:
            return []
        self.subtree_dict = dict()
        self.result = []
        self.traverse(root)
        return self.result

    def traverse(self, root):
        if root is None:
            return "#"
        left, right = self.traverse(root.left), self.traverse(root.right)
        subtree = left + "," + right + "," + str(root.val)
        if self.subtree_dict.get(subtree) is None:
            self.subtree_dict[subtree] = 1
        else:
            self.subtree_dict[subtree] += 1
        if self.subtree_dict.get(subtree) == 2:
            self.result.append(root)
        return subtree
```

654. 最大二叉树

```
# Definition for a binary tree node.
# 654. 最大二叉树
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> TreeNode:
        if len(nums) == 0:
            return None
        max_val, max_idx = self.getMaxIdx(nums)
        print(max_val)
        left_nums = nums[:max_idx]
        right_nums = nums[max_idx+1:]
        node = TreeNode(max_val)
        node.left = self.constructMaximumBinaryTree(left_nums)
```

```
        node.right = self.constructMaximumBinaryTree(right_nums)
        return node

    def getMaxIdx(self, nums):
        max_val, max_idx = nums[0], 0
        for i in range(1, len(nums)):
            if max_val < nums[i]:
                max_val = nums[i]
                max_idx = i
        return max_val, max_idx
```

965. 单值二叉树

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# 965. 单值二叉树
class Solution:
    def isUnivalTree(self, root: TreeNode) -> bool:
        if root is None:
            return False
        val = root.val
        return self.univalTree(root, val)

    def univalTree(self, root, val):
        if root is None:
            return True
        if root.val == val:
            return self.univalTree(root.left, val) and self.univalTree(root.right,
val)
        else:
            return False
```

2.2.二叉搜索树

95. 不同的二叉搜索树 II

```
# Definition for a binary tree node.
# 95. 不同的二叉搜索树 II
from typing import List

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
```

```
        self.right = right
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if n == 0:
            return []
        return self.build(1, n)

    def build(self, left, right):
        res = []
        if left > right:
            res.append(None)
            return res
        for i in range(left, right+1):
            # 自顶向下
            left_tree_list = self.build(left, i-1)
            right_tree_list = self.build(i+1, right)
            for left_tree in left_tree_list:
                for right_tree in right_tree_list:
                    node = TreeNode(i)
                    node.left = left_tree
                    node.right = right_tree
                    res.append(node)
        # 自底向上
        return res
```

96. 不同的二叉搜索树

```
# 96. 不同的二叉搜索树
class Solution:
    def numTrees(self, n: int) -> int:
        if n == 0:
            return 0
        self.val_list = [[0 for _ in range(n+2)] for _ in range(n+2)]
        return self.countTree(1, n+1)

    def countTree(self, left, right):
        if left >= right:
            return 1
        res = 0
        if self.val_list[left][right] != 0:
            return self.val_list[left][right]
        for i in range(left, right):
            left_count = self.countTree(left, i)
            right_count = self.countTree(i+1, right)
            res += left_count * right_count
        self.val_list[left][right] = res
        return res
```

98. 验证二叉搜索树

```python
# Definition for a binary tree node.
# 98. 验证二叉搜索树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        if root is None:
            return True
        elif root.left is None and root.right is not None:
            if root.val < self.getLeftNode(root.right).val:
                return self.isValidBST(root.right)
            else:
                return False
        elif root.left is not None and root.right is None:
            if self.getRigthNode(root.left).val < root.val:
                return self.isValidBST(root.left)
            else:
                return False
        elif root.left is not None and root.right is not None:
            if root.val > self.getRigthNode(root.left).val and root.val <
self.getLeftNode(root.right).val:
                return self.isValidBST(root.left) and self.isValidBST(root.right)
            else:
                return False
        else:
            return True

    def getLeftNode(self, node):
        if node.left is None:
            return node
        return self.getLeftNode(node.left)

    def getRigthNode(self, node):
        if node.right is None:
            return node
        return self.getRigthNode(node.right)
```

450. 删除二叉搜索树中的节点

```python
# Definition for a binary tree node.
from typing import Optional

# 450. 删除二叉搜索树中的节点
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
```

```
        self.left = left
        self.right = right

class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) ->
Optional[TreeNode]:
        if  root is None:
            return None
        if root.val > key:
            root.left = self.deleteNode(root.left, key)
        elif root.val < key:
            root.right = self.deleteNode(root.right, key)
        else:
            if root.left is None or root.right is None:
                root = root.left if root.left is not None else root.right
            else:
                cur = root.right
                while cur.left is not None:
                    cur = cur.left
                root.val = cur.val
                root.right = self.deleteNode(root.right, cur.val)
        return root
```

700. 二叉搜索树中的搜索

```
#700. 二叉搜索树中的搜索

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return None
        if root.val > val:
            return self.searchBST(root.left, val)
        elif root.val < val:
            return self.searchBST(root.right, val)
        else:
            return root

# 非递归版本
class Solution1:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return None
        val_node = None
```

```
        while root is not None:
            if root.val > val:
                root = root.left
            elif root.val < val:
                root = root.right
            else:
                val_node = root
                break
        return val_node
```

701. 二叉搜索树中的插入操作

```
# Definition for a binary tree node.
# 701. 二叉搜索树中的插入操作
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def insertIntoBST(self, root: TreeNode, val: int) -> TreeNode:
        if root is None:
            return TreeNode(val)
        if root.val > val:
            root.left = self.insertIntoBST(root.left, val)
        elif root.val < val:
            root.right = self.insertIntoBST(root.right, val)
        return root
```

230.二叉搜索树中第K小的元素

```
# Definition for a binary tree node.
from typing import Optional

# 230.二叉搜索树中第K小的元素
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        k, target = self.getK(root, k, None)
        return target

    def getK(self, root, k, target):
        if root is None:
```

```
            return k, target
        k, target = self.getK(root.left, k, target)
        k -= 1
        if k == 0:
            target = root.val
            return k, target
        k, target = self.getK(root.right, k, target)
        return k, target
```

538. 把二叉搜索树转换为累加树

```
# Definition for a binary tree node.
from typing import Optional

# 538. 把二叉搜索树转换为累加树
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if root is None:
            return None
        self.cum_val = 0
        result = 0
        self.CumSumTree(root, result)
        return root

    def CumSumTree(self, root, result):
        if root is None:
            return result
        result = self.CumSumTree(root.right, result)
        result += root.val
        root.val = result
        result = self.CumSumTree(root.left, result)
        return result
```

530. 二叉搜索树的最小绝对差

```
# Definition for a binary tree node.
#530. 二叉搜索树的最小绝对差
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
    def getMinimumDifference(self, root: TreeNode) -> int:
        pre_node = None
        min_val = 0
        if root is None:
            return min_val
        else:
            min_val = (root.val - root.left.val) if root.left is not None else
(root.right.val - root.val)
        min_val, _ = self.getMinDelta(root, min_val, pre_node)
        return min_val

    def getMinDelta(self, root, min_val, pre_node):
        if root is None:
            return min_val, pre_node
        min_val, pre_node = self.getMinDelta(root.left, min_val, pre_node)
        if pre_node is not None:
            min_val = min(root.val - pre_node.val, min_val)
        pre_node = root
        min_val , pre_node = self.getMinDelta(root.right, min_val, pre_node)
        return min_val, pre_node
```

783. 二叉搜索树节点最小距离

```
# Definition for a binary tree node.
# 783. 二叉搜索树节点最小距离
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
import sys
class Solution:
    def minDiffInBST(self, root: TreeNode) -> int:
        if root is None:
            return 0
        pre_node = None
        min_val = sys.maxsize
        min_val, pre_node = self.getMin(root, pre_node, min_val)
        return min_val


    def getMin(self, root, pre_node, min_val):
        if root is None:
            return min_val, pre_node
        min_val, pre_node = self.getMin(root.left, pre_node, min_val)
        if pre_node is not None:
            min_val = min(root.val - pre_node.val, min_val)
        pre_node = root
        min_val, pre_node = self.getMin(root.right, pre_node, min_val)
        return min_val, pre_node
```

1373. 二叉搜索子树的最大键值和

```python
# Definition for a binary tree node.
# 1373. 二叉搜索子树的最大键值和

from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class Solution:
    def maxSumBST(self, root: Optional[TreeNode]) -> int:
        if root.val is None:
            return 0
        self.max_val = 0
        status, val = self.searchBST(root, True, 0)
        return self.max_val

    def searchBST(self, root, status, val):
        if root is None:
            return True and status, 0
        left_status, left_val = self.searchBST(root.left, status, val)
        right_status, right_val = self.searchBST(root.right, status, val)
        status = left_status and right_status
        if root.left is not None:
            left_node = self.getRightNode(root.left)
            if left_node.val < root.val:
                status = status and True
            else:
                status = status and False
        if root.right is not None:
            right_node = self.getLeftNode(root.right)
            if right_node.val > root.val:
                status = status and True
            else:
                status = status and False
        if status:
            cum_val = root.val + left_val + right_val
            self.max_val = max(cum_val, self.max_val)
        else:
            cum_val = 0
        return status, cum_val

    def getLeftNode(self, root):
        if root is None:
            return None
        while root.left is not None:
            root = root.left
        return root
```

```
    def getRightNode(self, root):
        if root is None:
            return None
        while root.right is not None:
            root = root.right
        return root
```

2.3.图论算法

797. 所有可能的路径

```
# 797. 所有可能的路径
from typing import List

class Solution:
    def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:
        pre_list = [0]
        path_list = []
        self.getPath(graph[0], graph, path_list, pre_list)
        return path_list

    def getPath(self, n_list, graph, path_list, pre_list):
        for i in n_list:
            if i == len(graph)-1:
                tmp = pre_list[::]
                tmp.append(i)
                path_list.append(tmp)
            else:
                self.getPath(graph[i], graph, path_list, pre_list + [i])
```

785.判断二分图(二分图)

```
from typing import List

# 785.判断二分图
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n = len(graph)
        self.ok = True
        self.color = [False for _ in range(n)]
        self.visited = [False for _ in range(n)]
        for v in range(n):
            if self.visited[v] is False:
                self.traverse(graph, v)
        return self.ok

    def traverse(self, graph, v):
        if self.ok is False:
```

```
            return
        self.visited[v] = True
        for w in graph[v]:
            if self.visited[w] is False:
                self.color[w]  = not self.color[v]
                self.traverse(graph, w)
            else:
                if self.color[w] == self.color[v]:
                    self.ok = False
```

886. 可能的二分法(二分图)

```
# 886. 可能的二分法

class Solution:
    def possibleBipartition(self, n: int, dislikes: List[List[int]]) -> bool:
        graph = self.buildGraph(n, dislikes)

        self.visited = [False for _ in range(len(graph))]
        self.color = [False for _ in range(len(graph))]
        self.ok = True
        for v in range(1, n+1):
            if self.visited[v] is False:
                self.trans(graph, v)
        return self.ok

    def buildGraph(self, n, dislikes):
        graph = [[] for _ in range(n+1)]
        for edge in dislikes:
            v, w = edge[1], edge[0]
            if graph[v] is None:
                graph[v] = []
            graph[v].append(w)
            if graph[w] is None:
                graph[w] = []
            graph[w].append(v)
        return graph

    def trans(self, graph, v):
        if self.ok is False:
            return
        self.visited[v] = True
        for w in graph[v]:
            if self.visited[w] is False:
                self.color[w] = not self.color[v]
                self.trans(graph, w)
            else:
                if self.color[w] == self.color[v]:
                    self.ok = False
```

207. 课程表 (拓扑排序)

```python
# 207. 课程表
from typing import List

class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        # 递归的点，防止递归的loop
        self.on_path = [False for _ in range(numCourses+1)]
        # 遍历的点，防止遍历的loop
        self.visited = [False for _ in range(numCourses+1)]
        self.has_clcle = False
        graph = self.buildGraph(numCourses, prerequisites)
        for i in range(numCourses):
            self.traverse(graph, i)
        return not self.has_clcle

    def buildGraph(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]
        for edge in prerequisites:
            froms = edge[1]
            tos = edge[0]
            graph[froms].append(tos)
        return graph

    def traverse(self, graph, i):
        if self.on_path[i] is True:
            self.has_clcle = True
        if self.visited[i] or self.has_clcle:
            return
        self.visited[i] = True
        self.on_path[i] = True
        for t in graph[i]:
            self.traverse(graph, t)
        self.on_path[i] = False
```

210. 课程表 II (拓扑排序)

```python
# 210. 课程表 II
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        self.postorder = []
        self.has_clcle = False
        self.visited, self.on_path = [False for _ in range(numCourses+1)], [False
for _ in range(numCourses+1)]
        graph = self.buildGraph(numCourses, prerequisites)
        # 遍历图
        for i in range(numCourses):
            self.traverse(graph, i)
```

```
        # 有环就直接返回
        if self.has_clcle:
            return []
        self.postorder = self.postorder[::-1]
        return self.postorder

    def traverse(self, graph, s):
        if self.on_path[s]:
            self.has_clcle = True
        if self.visited[s] or self.has_clcle:
            return
        # 前序遍历
        self.on_path[s] = True
        self.visited[s] = True
        for t in graph[s]:
            self.traverse(graph, t)
        self.on_path[s] = False
        # 后序遍历 这里整个图遍历的时候不会有问题吗？
        self.postorder.append(s)
        return


    def buildGraph(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses+1)]
        for edge in prerequisites:
            # 单向
            froms, tos = edge[1], edge[0]
            graph[froms].append(tos)
        return graph
```

130. 被围绕的区域

```
# 130. 被围绕的区域
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        if board is None or len(board) == 0:
            return
        row, col = len(board), len(board[0])
        for i in range(row):
            self.dfs(board, i, 0)
            self.dfs(board, i, col - 1)
        for j in range(col):
            self.dfs(board, 0, j)
            self.dfs(board, row-1, j)
        for i in range(row):
            for j in range(col):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
```

```
                if board[i][j] == '-':
                    board[i][j] = 'O'

    def dfs(self, board, i, j):
        if (i< 0 or j < 0 or i >= len(board) or j >= len(board[0]) or board[i][j]
!= "O"):
            return
        board[i][j] = '-'
        self.dfs(board, i-1, j)
        self.dfs(board, i+1, j)
        self.dfs(board, i, j-1)
        self.dfs(board, i, j+1)
        return
```

990. 等式方程的可满足性

```
# 990. 等式方程的可满足性
from typing import List

class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
```

```
        return self.count

class Solution:
    def equationsPossible(self, equations: List[str]) -> bool:
        uf = UF(26)
        a_id = ord('a')
        for eq in equations:
            if eq[1] == '=':
                x = eq[0]
                y = eq[3]
                uf.union(ord(x) - a_id, ord(y) - a_id)
        for eq in equations:
            if eq[1] == '!':
                x = eq[0]
                y = eq[3]
                if uf.connected(ord(x) - a_id, ord(y) - a_id):
                    return False
        return True
```

261.以图判树(最小生成树)

```
# 261.以图判树
from msilib.schema import PublishComponent
from typing import List


class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ
```

```
    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

class Solution:
    def validTree(n: int, edges:List[List[int]]):
        uf = UF(n)
        for edge in edges:
            u, v = edge[0], edge[1]
            if uf.connected(u, v):
                return False
            # 这条边不会产生环，可以是树的一部分
            uf.union(u, v)
        # 保证最后只有一个连通分量
        return uf.count() == 1
```

1135.最低成本连通所有城市(最小生成树)

```
# 1135.最低成本连通所有城市
from typing import List

class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1

    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ
```

```
    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

# 1135.最低成本连通所有城市
class Solution:
    def minimumCost(n:int, connections: List[List[int]]):
        uf = UF(n+1)
        # 最小生成树，按照weight升序排序
        connections = sorted(connections, key=lambda x: x[2])
        mst = 0.0
        for edge in connections:
            u, v, weight = edge[0], edge[1], edge[2]
            # 成环就跳过
            if uf.connected(u, v):
                continue
            mst += weight
            uf.union(u, v)
        # 保证所有节点连通，因为0没有被使用，额外占有一个，因此总共是2
        return mst if uf.count() == 2 else -1
```

1584.连接所有点的最小费用(最小生成树)

```
# 1584.连接所有点的最小费用
from typing import List

class UF():
    def __init__(self, n):
        self.count = n
        self.parent = [i for i in range(n)]
        self.size = [1 for _ in range(n)]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        if rootP == rootQ:
            return None
        # 小树挂大树下面
        if self.size[rootP] > self.size[rootQ]:
            self.parent[rootQ] = rootP
            self.size[rootP] += self.size[rootQ]
        else:
            self.parent[rootP] = rootQ
            self.size[rootQ] += self.size[rootP]
        self.count -= 1
```

```
    def connected(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)
        return rootP == rootQ

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def count(self):
        return self.count

# 1584.连接所有点的最小费用
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n = len(points)
        edges = []
        for i in  range(n):
            for j in range(i+1, n):
                xi, yi = points[i][0], points[i][1]
                xj, yj = points[j][0], points[j][1]
                edges.append([i, j, abs(xi - xj) + abs(yi-yj)])
        edges = sorted(edges, key=lambda x: x[2])
        mst = 0.0
        uf = UF(n)
        for edge in edges:
            u, v, weight = edge[0], edge[1], edge[2]
            if uf.connected(u, v):
                continue
            mst += weight
            uf.union(u, v)

        return mst
```

743. 网络延迟时间(最短路径)

```
# 743. 网络延迟时间
from typing import List
import sys

class State:
    def __init__(self, id, distFromStart):
        self.id = id
        self.dist_from_start = distFromStart

class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        # 节点从1开始，所以需要一个n+1的领接表
        graph = [[] for _ in range(n+1)]
```

```
        # 构造图
        for edge in times:
            froms, tos, weight = edge[0], edge[1], edge[2]
            graph[froms].append([tos, weight])
        print(graph)
        distTo = self.dijkstra(k, graph)
        print(distTo)
        res = 0
        for dist in distTo[1:]:
            if dist == sys.maxsize:
                return -1
            res = max(res, dist)
        return res

    def dijkstra(self, start:int, graph:List[List[List[int]]]):
        # 用于记录节点的最小距离
        distTo = [sys.maxsize for _ in range(len(graph))]
        distTo[start] = 0

        pq = []
        # 从start节点开始进行BFS
        pq.append(State(start, 0))

        while len(pq) > 0:
            cur_state = pq.pop(0)
            cur_node_id = cur_state.id
            cur_dist_from_start = cur_state.dist_from_start

            if cur_dist_from_start > distTo[cur_node_id]:
                continue
            # 将cur_node 的相邻节点装入队列
            for neighbor in graph[cur_node_id]:
                next_node_id = neighbor[0]
                dist_to_next_node = distTo[cur_node_id] + neighbor[1]
                if distTo[next_node_id] > dist_to_next_node:
                    distTo[next_node_id] = dist_to_next_node
                    pq.append(State(next_node_id, dist_to_next_node))
        return distTo
```

1514. 概率最大的路径(最短路径)

```
# 1514. 概率最大的路径

from typing import List
import sys

class State:
    def __init__(self,cur_id, cur_prob):
        self.cur_id = cur_id
        self.cur_prob = cur_prob
```

```
class Solution:
    def maxProbability(self, n: int, edges: List[List[int]], succProb:
List[float], start: int, end: int) -> float:
        graph = self.buildGraph(n, edges, succProb)
        probTo = [0 for _ in range(n)]
        pq = []
        pq.append(State(start, 1))
        probTo[start] = 1
        # 广度优先遍历
        while len(pq) > 0:
            cur_state = pq.pop(0)
            cur_id = cur_state.cur_id
            cur_prob = cur_state.cur_prob

            if probTo[cur_id] > cur_prob:
                continue
            for neighbor in graph[cur_id]:
                next_id, prob = neighbor[0], neighbor[1]
                next_prob = cur_prob * prob
                if probTo[next_id] < next_prob:
                    probTo[next_id] = next_prob
                    pq.append(State(next_id, next_prob))
        return probTo[end]

    def buildGraph(self, n: int, edges:List[List[int]], succProb: List[float]) ->
List[List[int]]:
        graph = [[] for _ in range(n)]
        for edge, prob in zip(edges, succProb):
            u, v = edge[0], edge[1]
            graph[u].append([v, prob])
            graph[v].append([u, prob])
        return graph
```

1631. 最小体力消耗路径(最短路径)

```
import sys
from typing import List
import sys

# 1631. 最小体力消耗路径
class State:
    def __init__(self, x, y, delta):
        self.x = x
        self.y = y
        self.delta = delta

class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        if len(heights) == 1 and len(heights[0]) == 1:
            return 0
        row, col = len(heights), len(heights[0])
```

```
        graph = self.buildGraph(heights)
        pq = []
        delta_list = [[sys.maxsize for _ in range(col)] for _ in range(row)]

        delta_list[0][0] = 0
        pq.append(State(0, 0, 0))
        while len(pq) > 0:
            cur_state = pq.pop(0)
            if delta_list[cur_state.x][cur_state.y] < cur_state.delta:
                continue
            for neighbor in graph[cur_state.x][cur_state.y]:
                next_x, next_y, height = neighbor[0], neighbor[1], neighbor[2]
                val = max(height, delta_list[cur_state.x][cur_state.y])
                if delta_list[next_x][next_y] > val:
                    delta_list[next_x][next_y] = val
                    state = State(next_x, next_y, val)
                    pq.append(state)
        return delta_list[-1][-1]

    def buildGraph(self, heights):
        graph = [[[] for _ in range(len(heights[0]))] for _ in
range(len(heights))]
        for i in range(len(heights)):
            for j in range(len(heights[0])):
                val = heights[i][j]
                left, right, top, bot = None, None, None, None
                if j - 1 >= 0:
                    left = heights[i][j-1]
                    delta = abs(left - val)
                    graph[i][j].append([i, j-1, delta])
                if j + 1 < len(heights[0]):
                    right = heights[i][j+1]
                    delta = abs(right - val)
                    graph[i][j].append([i, j+1, delta])
                if i -1 >= 0:
                    top = heights[i-1][j]
                    delta = abs(top - val)
                    graph[i][j].append([i-1, j, delta])
                if i + 1 < len(heights):
                    bot = heights[i+1][j]
                    delta = abs(bot - val)
                    graph[i][j].append([i+1, j, delta])
        return graph
```