四月份月度测试题

同学们！正课在视频的10分钟后开始！看回放请自动拖10分钟！

104. 二叉树的最大深度

先拿到左子树和右子树的最大深度 L和R，然后两者取最大值加1就是最大深度。

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function(root) {
    if(root === null)  return null;
    var Llen = maxDepth(root.left);
    var Rlen = maxDepth(root.right);
    return Llen >= Rlen ? Llen + 1 : Rlen + 1;
    return root;
};
```

面试题 04.05. 合法二叉搜索树

首先要知道合法搜索的条件：左子树的值小于根节点的值，根节点的值小于右子树的值；

这里我们用递归中序遍历。因为中序遍历出来的序列是一个升序，只需要我们在遍历的

时候判断，当前节点的值是否大于前一个中序遍历到的节点的值即可。

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function(root) {
```

```
    let stack = [];
    let inorder = -Infinity;
    while (stack.length || root !== null) {
        while (root !== null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        // 如果中序遍历得到的节点的值小于等于前一个 inorder，说明不是二叉搜索树
        if (root.val <= inorder) {
            return false;
        }
        inorder = root.val;
        root = root.right;
    }
    return true;
};
```

230. 二叉搜索树中第K小的元素

把二叉树按着从小到大进行排序，也是进行中序遍历，然后取出前k个。

方法一：递归

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
// 递归
 let kthSmallest = function(root, k) {
    let res = null
    let inOrderTraverseNode = function(node) {
        if(node !== null && k > 0) {
            // 先遍历左子树
            inOrderTraverseNode(node.left)
```

```
            // 然后根节点
            if(--k === 0) {
                res = node.val
                return
            }
            // 再遍历右子树
            inOrderTraverseNode(node.right)
        }
    }
    inOrderTraverseNode(root)
    return res
}
```

方法二：迭代

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
// 迭代
let kthSmallest = function(root, k) {
    let stack = []
    let node = root

    while(node || stack.length) {
        // 遍历左子树
        while(node) {
            stack.push(node)
            node = node.left
        }

        node = stack.pop()
        if(--k === 0) {
            return node.val
        }
```

```
        node = node.right
    }
    return null
}
```

199. 二叉树的右视图

遍历Map（map的遍历顺序由其key的添加顺序决定）取每个key的value。题意要求返回的是数组（Array），得出结果后用三点运算符进行解构成数组。

```
var rightSideView = function(root) {
 if(!root) return [];

 // Map存储，key是当前节点的高度，value是当前节点的值
 let depthMap = new Map();

 // 构造队列，并赋予队首元素的初始高度
 let queue = [[root, 0]];

 while(queue.length) {
   // 取出队首元素
   let [ {val, left, right}, depth ] = queue.shift();

   /*
   关键点
   更新Map中每个key所对应的val，
   因为是BFS，所以可以保证最终Map的key所对应的val是同一层节点中的最右边节点的val
   */
   depthMap.set(depth, val);

   // 高度递增
   depth += 1;

   // 仅将存在的节点推入队列中
   if(left) {
     queue.push([left, depth]);
   }

   // 仅将存在的节点推入队列中
   if(right) {
     queue.push([right, depth]);
   }
 }

 // Map.prototype.values()返回是可迭代对象，故需利用“展开语法”来将其转换为数组
 return [...depthMap.values()];
}
```

[100. 相同的树]

首先我们要知道什么是相同的二叉树:

当且仅当两个二叉树的结构完全相同，且所有对应节点的值相同的时候。因此，用递归来比较两棵树相同位置的当前节点是否相同，若不同返回false；如果相同就递归比较左右孩子节点。

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
// 方法：递归
var isSameTree = function(p, q) {
    if (p === null && q !== null) {
      return false
    } else if (p !== null && q === null) {
      return false
    } else if (p === null && q === null) {
      return true
    } else {
      return  p.val === q.val && isSameTree(p.left, q.left) &&
isSameTree(p.right, q.right)
    }
  return isSameTree(p, q)
}
```

101. 对称二叉树

方法一：递归; 首先我们要知道满足对称二叉树的条件：(1) 他们两个根节点具有相同的值 (2)
每个树的右子树都与另一个树的左子树镜像对称。这道题要写递归和迭代两种方法。

```
 var isSymmetric = function(root) {
    if(!root) return true
    return isMirror(root.left, root.right)
};
// 辅助函数
function isMirror(leftroot, rightroot) {
```

```
    // 左右子树均为空，说明相等
    if(!leftroot && !rightroot) return true;
    // 左右子树有一个不为空，说明不相等
    if(!leftroot) return false;
    if(!rightroot) return false;
    // 若两个节点值相同，且左子树的左子树等于右子树的右子树，左子树的右子树等于右子树的左子树
    if(leftroot.val == rightroot.val && isMirror(leftroot.left, rightroot.right)
&& isMirror(leftroot.right, rightroot.left)) return true;
    else return false;
}
```

方法二：迭代

```
var isSymmetric = function(root) {
    // 根节点为空，肯定对称
    if(!root) return true
    // 压入左子树和右子树，判断
    let stack = [root.left, root.right]
    // 循环加栈代替迭代
    while(stack.length) {
        let left = stack.pop()
        let right = stack.pop()
        // 都是空节点就跳出，继续循环
        if(!left && !right) continue
        // 有一个不为空就是不对称的
        if(!left) return false
        if(!right) return false
        // 左右节点值相等
        if(left.val == right.val) {
            // 继续判断左右
            stack.push(left.left)
            stack.push(right.right)
            stack.push(left.right)
            stack.push(right.left)
        } else {
            return false
        }
    }
    return true
};
```

235. 二叉搜索树的最近公共祖先

根据二叉搜索树性质：如果 p.val 和 q.val 都比 root.val 小，则 p、q 肯定在 root

的左子树，便递归左子树；否则递归右子树。如果一个小于根节点，一个大于根节点，那么根节点就是最近祖先。

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
var lowestCommonAncestor = function(root, p, q) {
    let ancestor = root;
    while (true) {
        if (p.val < ancestor.val && q.val < ancestor.val) {
            return lowestCommonAncestor(ancestor.left,p,q)
        }
        else if (p.val > ancestor.val && q.val > ancestor.val) {
                return lowestCommonAncestor(ancestor.right,p,q)
        }
        // 一个小一个大就证明 ancestor是最近祖先
        else {
            break;
        }
    }
    return ancestor;
};
```

124. 二叉树中的最大路径和

我们知道一个子树内部的最大路径和 = 左子树**提供**的最大路径和 + 根节点值 + 右子树**提供**的最大路径和。分别递归遍历左子树和右子树的最大路径和，在最大路径和和当前路径和之间取最大值并返回。

```
const maxPathSum = (root) => {
    let maxSum = Number.MIN_SAFE_INTEGER; // 最大路径和
    const dfs = (root) => {
        if (root == null) { // 遍历到null节点，收益0
            return 0;
```

```
        }
        const left = dfs(root.left);  // 左子树提供的最大路径和
        const right = dfs(root.right); // 右子树提供的最大路径和

        const innerMaxSum = left + root.val + right; // 当前子树内部的最大路径和
        maxSum = Math.max(maxSum, innerMaxSum);      // 挑战最大纪录

        const outputMaxSum = root.val + Math.max(left, right); // 当前子树对外提供的最大和

        // 对外提供的路径和为负，直接返回0。否则正常返回
        return outputMaxSum < 0 ? 0 : outputMaxSum;
    };

    dfs(root);  // 递归的入口
    return maxSum;
};
```

[347. 前 K 个高频元素]

map+数组；利用 map 记录每个元素出现的频率，利用数组来比较排序元素。类似第六课的前k个高频单词

```
let topKFrequent = function(nums, k) {
    let map = new Map(), arr = [...new Set(nums)]
    nums.map((num) => {
        if(map.has(num)) map.set(num, map.get(num)+1)
        else map.set(num, 1)
    })

    return arr.sort((a, b) => map.get(b) - map.get(a)).slice(0, k);
};
```

973. 最接近原点的 K 个点

堆的做法；这道题和我们已经做过的【剑指 Offer 40. 最小的k个数】一样的套路和逻辑

```
/**
* @param {number[][]} points
* @param {number} k
* @return {number[][]}
*/
var kClosest = function(points, K, p = new PriorityQueue()) {
   for (var i = 0, v; i < points.length; i++) {
       v = points[i][0] * points[i][0] + points[i][1] * points[i][1]
       if (i < K) {
           p.add(v, i) // 0 到 K - 1 项，放入 优先队列
       } else if (v < p.first()) {// < 优先队列第一项
```

```
            p.shift() // 优先队列中距离最大的节点弹出
            p.add(v, i) // 放入当前节点，上浮
        }
    }
    return p.q.map(v => points[v.second]) // 将优先队列 还原为 位置
};
class PriorityQueue {
    constructor(a) {
        this.q = []
        a && this._build(a)
    }
    add(v, second) { // 添加
        this.q.push({v, second}) // 添加 值（比较用） 和 第二参数（索引）
        this._up(this.q.length - 1) // 放入 二叉树的尾部，然后 上浮
    }
    shift() { // 弹出
        this.q.shift() // 弹出 根节点
        if (this.q.length) { // 如果还有节点
            this.q.unshift(this.q.pop()) // 将 最尾部的节点 放到根节点位置
            this._down(0) // 下沉 根节点
        }
    }
    first() { // 根节点的值
        return this.q[0].v
    }
    _build(a) { // 初始化
        this.q.push({v:a[0]}) // 先给空队列放入第0个元素
        for (var i = 1; i < a.length; i++) this.q.unshift({v:a[i]}),
this._down(0)
        // 从第1个元素起，把新元素放在根节点，然后下沉 根节点
    }
    _swap(l, r, t) { // 交换
        t = this.q[l], this.q[l] = this.q[r], this.q[r] = t // 交换两个节点
    }
    _down(i){ // 下沉
        var t = this.q.length - 2 >> 1, max, maxI // 叶子节点的根节点索引，下沉到 叶
子节点的根节点停止
        while(i <= t){
            var l = i * 2 + 1, r = l + 1 // 左子节点的索引 = 当前节点索引 * 2 + 1，右
子节点的索引 = 左子节点的索引 + 1
            if ((this.q[l] ? this.q[l].v : -Infinity) > (this.q[r] ? this.q[r].v
: -Infinity))
                max = this.q[l].v, maxI = l
            else max = this.q[r].v, maxI = r // 找到 左子节点 和 右子节点的 较大者
```

```
            if (this.q[i].v < max) this._swap(i, maxI), i = maxI // 当前节点的 左子节点 或 右子节点 比 它大，交换
            else break
        }
    }
    _up(i) { // 上浮
        while(i > 0){ // 不能超过根节点
            var t = i - 1 >> 1 // 当前节点的 根节点索引 = 当前节点索引 - 1 的一半
            if (this.q[i].v > this.q[t].v) this._swap(i, t), i = t // 当前节点值 比 它的根节点 大，交换
            else break
        }
    }
}
```

451. 根据字符出现频率排序

map + es6的解构数组；先遍历得到每个字符出现的频率，然后进行解构成数组降序排序，最后返回

```
/**
 * @param {string} s
 * @return {string}
 */
var frequencySort = function(s) {
    let map = new Map();
    let ans = '';
    for (let w of s) {
        map.set(w, (map.get(w) || 0) + 1);
    }
    map = new Map([...map].sort((v1, v2) => v2[1] - v1[1]));
    for(let [k, v] of map) {
        for(let i = 0; i < v; i++) {
            ans += k;
        }
    }
    return ans;
};
```