【第二十周】专项面试题解析

958. 二叉树的完全性检验

1.假设知道完全二叉树的节点个数，接下来计算左右子树的节点数量，假设一共是12个

2.接下来递归左子树是不是7个节点的完全二叉树

3.右子树是不是4个节点的完全二叉树

```
 // 数一数二叉树里面有多少个节点
 var countNodeNum = function(root) {
     if(root == null) return 0;
    return countNodeNum(root.left) + countNodeNum(root.right) + 1;
};
var judge = function(root,n,m) {
    if(root == null) return n == 0;
    if(n == 0) return false;
    if(n == 1) return root.left == null && root.right == null;
    let k = Math.max(0,2 * m - 1);
    let l = Math.min(m ,n - k),r = n - k - l;
    return judge(root.left,(k - 1) / 2 + l, m / 2) && judge(root.right,(k - 1) /
2 + r,m / 2);

};

var isCompleteTree = function(root) {
    if(root == null) return true;
    let n = countNodeNum(root),m = 1,cnt = 1;
    while(cnt + 2 * m <= n){
        m *= 2;
        cnt += m;
    }
    return judge(root,n,m)
};
```

1367. 二叉树中的列表

1.先序遍历二叉树，寻找 root.val == head.val 的二叉树节点，与链表开头不一样的直接略过

2.每次找到这种节点后，递归的判断该子树能否和链表匹配上，见代码中的judge()函数

```
var isSubPath = function(head, root) {
    // 在一颗树上面找一条空链表肯定能找到
    if(head == null) return true;
    if(root == null) return false;
    // 从root开始捋着比较，是否能找到连续的符合题意的链表
    if(root.val == head.val  && judge(root,head))return true;
```

```
    //否则就递归地比较 用树中的每一个节点依次比较链表中的头节点
    return isSubPath(head,root.left) || isSubPath(head,root.right);
};
var judge = function(root,head){
    if(head == null) return true;
    if(root == null) return false;
    if(root.val != head.val) return false;
    // 这里证明root节点的值，等于head节点的值
    // 捋着向下比较左子树，向下比较右子树
    // 在左右子树中 找到任意一条路径 能够匹配到 链表剩余部分的节点 证明能够匹配成功
    return judge(root.left,head.next) || judge(root.right,head.next);
}
```

剑指 Offer 36. 二叉搜索树与双向链表

1.本题考察二叉搜索树的性质：左节点 < 当前节点 < 右节点。转换后的双向链表是有序的，这里采用中序递归遍历保证有序性。

2.题目要求循环双向链表，因此尾节点的 right 要指向首节点，首节点的 left 要指向尾节点。

3.结合中序遍历，递归处理二叉树。初始化一个代表上一个节点的 pre 变量。递归中要做的就是：pre 的 right 指针指向当前节点 node，node 的 left 指向 pre，并且将 pre 更新为 node。

4.要注意的是，当递归到最下面的左节点时，pre 为空，要保留节点作为循环链表的 head。并在中序遍历结束后，处理头节点和尾节点的指针关系。

```
 var in_order = function(root){
     if(root == null) return;
     //  搭建中序遍历的过程
     in_order(root.left);
     //  在中序遍历的过程中做操作
     if(pre == null){
         head = root;
     } else {
         pre.right = root;
     }
     root.left = pre;
     pre = root;
     in_order(root.right);
     return;
 }
var treeToDoublyList = function(root) {
    if(root == null) return null;
    head = pre = null;
    // 中序遍历，得到一条链表，head是链表的头，pre是链表的尾部，
    in_order(root);
    // 把链表连接起来，变成循环双端链表
    head.left = pre;
    pre.right = head;
    return head;
};
```

464. 我能赢吗

1. 首先我们把问题倒过来思考，不是累加到 100，而是从 100 开始取数，能否最后一个取完。

2.然后可以先排除两种情况

3.开局全部拿完，直接获胜

4.全部整数都使用过了还没拿完，无法获胜

5.接下来思考，怎么样能够获胜，有以下两种情况

6.很明显，只要 A 能让 B 输，A 就赢了

7.反过来说如果 A 无法让 B 输，那 A 就输了

8.当然，如果 A 可以直接把剩下的数一次性拿完，那 A 获胜

9.在A, B 角色是可以互换的，所以我们并不需要判断当前轮到 A 还是 B，只需要判断当前能否获胜

10.上面的可行分支，指的就是 A 还剩下哪些整数可以拿，最后就是处理状态了

11.无论如何，我们都需要记录哪些整数已经被拿过了

12.但是我们不需要记录总量还剩下多少，因为我们只要知道拿过哪些整数，就能算出剩下的总量

13.题目告诉我们 maxChoosableInteger <= 20，而一个 int 可以表示 32 位整数，因此我们可以使用一个 int 来表示当前的状态，每一位的 0 或 1 则代表这一位的整数是否已经拿过

```
var canIWin = function (maxChoosableInteger, desiredTotal) {
    // 直接获胜
    if (maxChoosableInteger >= desiredTotal) return true;

    // 全部拿完也无法到达
    var sum = maxChoosableInteger * (maxChoosableInteger + 1) / 2;
    if (desiredTotal > sum) return false;

    // 记忆化
    var dp = {};

    /**
     * @param {number} total 剩余的数量
     * @param {number} state 使用二进制位表示抽过的状态
     */
    function f(total, state) {
        // 有缓存
        if (dp[state] !== undefined) return dp[state];

        for (var i = 1; i <= maxChoosableInteger; i++) {
            var curr = 1 << i;
            // 已经抽过这个数
            if (curr & state) continue;
            // 直接获胜
            if (i >= total) return dp[state] = true;
            // 可以让对方输
            if (!f(total - i, state | curr)) return dp[state] = true;
        }
```

```
        // 没有任何让对方输的方法
        return dp[state] = false;
    }

    return f(desiredTotal, 0);
};
```

172. 阶乘后的零

1、开始寻找有趣的规律, 如果阶乘中的某一位数是5的整数倍, 则可以跟2生成一个0, 而2的整数倍即偶数的数量很显然比5的整数倍的数量多

2、要考虑 25 可以拆分成 5 * 5, 可以生成2个零, 125可以拆分成 5 * 5 * 5可以生成3个零, ...

3、再次寻找规律, 可以发现, 结果为n除以5的整数次幂向下取整之和； 4、 const res = Math.floor(n / 5) + Math.floor(n / (5 * 5)) + ...

```
var trailingZeroes = function (n) {
  let m = 5,cnt = 0;
  while (n / m) {
   cnt += Math.floor(n / m);
   m *= 5;
  }
  return cnt;
}
```

384. 打乱数组

考察封装思维的题，记录当前数组原始的状态

1.n个数产生的结果必须有n!种可能。遍历nums数组，每次取[i, n-1]闭区间的一个随机数nums[rand]，交换nums[i]和nums[rand]即可；

2.如nums = ['Solution', 'shuffle', 'reset', 'shuffle']时，执行for循环如下

3.第一轮，i = 0，rand 取值范围是 [0, 3]，有 4 个可能的取值 ['Solution', 'shuffle', 'reset', 'shuffle']

4.第二轮，i = 1，rand 取值范围是 [1, 3]，有 3 个可能的取值 ['shuffle', 'reset','shuffle']

5.第三轮，i = 2，rand 取值范围是 [2, 3]，有 2 个可能的取值 ['reset','shuffle']

6.第四轮，i = 3，rand 取值范围是 [3, 3]，有 1 个可能的取值 ['shuffle']

7.最后共有结果数为 每轮数组操作的数量相乘 = 24 = 4!

```
/**
 * @param {number[]} nums
```

```
 */
var Solution = function(nums) {
    this.nums = nums;
};

/**
 * 获取原数组
 * Resets the array to its original configuration and return it.
 * @return {number[]}
 */
Solution.prototype.reset = function() {
    return this.nums;
};

/**
 * 打乱数组
 * Returns a random shuffling of the array.
 * @return {number[]}
 */
Solution.prototype.shuffle = function() {
    const nums = this.nums.slice(0);
    let n = nums.length;

    // 产生的结果有 n! 种可能
    for (let i = 0; i < n; i++) {
        // 从 i 到 n-1 随机选一个
        const rand = randOne(i, n - 1);
        // 交换nums数组i和rand下标的两个元素
        [ nums[i], nums[rand] ] = [ nums[rand], nums[i] ];
    }

    return nums;
};

// 获取闭区间 [n, m] 内的一个随机整数
function randOne(n, m) {
    return Math.floor(Math.random() * (m - n + 1)) + n;
};

// const nums = ['Solution', 'shuffle', 'reset', 'shuffle'];
// var obj = new Solution(nums);
// var param_1 = obj.reset();
// var param_2 = obj.shuffle();
// console.log(param_1);
// console.log(param_2);
```

[437. 路径总和 III]

给了一个二叉树，给了一个路经总和，让我们看看这个二叉树中到底有多少条路径 等于给定我们的路径和值1.方法双递归：所谓双递归，其实就是内外两层递归实现暴力遍历所有节点

2.外层递归保证遍历到树的所有节点；内层递归才是真正寻找当前节点的递归 -- 以当前节点为根节点，路径和符合要求的情况

3.然后将所有情况加在一起就是最终的结果了。

4.时间复杂度分析：外层递归遍历所有节点，内层节点遍历子树节点，所以最终是 1+2+3+...+N,所以平均O(N^2)

5.空间复杂度：O(1)

```
var pathSum = function (root, targetSum) {
    const recursion = (root) => {
        // 从根节点起，符合的有要求的递归
        if (!root) return 0
        // 内层递归
        const dfs = (cRoot, leave) => {
            // 从当前节点为起点，符合条件的格式
            if (!cRoot) return 0
            const flag = (cRoot.val === leave) ? 1 : 0
            const cLeft = dfs(cRoot.left, leave-cRoot.val)
            const cRight = dfs(cRoot.right, leave-cRoot.val)
            return flag + cLeft + cRight
        }
        const page = dfs(root,targetSum) // 以当前节点为起点，满足条件的个数
        const left = recursion(root.left)
        const right = recursion(root.right)
        return page + left + right
    }
    return recursion(root)
};
```