在开始之前

从本节开始，我们进入“大厂真题解读与训练”环节。在进入正题之前，笔者想要先帮大家捋清楚两件事情：

如何正确看待你已经做过的那些题

在2-23节的漫长的知识讲解过程中，我们学过的所有题目，都是**实打实的大厂真题**。不要因为是例题，就心不在焉。要知道，能够选入例题、作为“教具”出现的题目，一定都是经典中的经典，是需要反复咀嚼的。

如何正确看待你即将要做的这些题

“真题解读”!==“猜题”。

一些同学早期在各种营销号、培训机构广告的蛊惑下，潜意识里会觉得大公司总会有一套一成不变的面试套路，认为有类似于“面试题库”这样的稳定题源存在，因此对面试猜题这种性质的行为抱有强烈的幻想。

我们刷题之旅的第一步，就是要打破这种幻想——算法面试几乎没有什么因公司而异的套路，就算有（比如Google），它的更新频率也是非常高的。唯一的“套路”只能是你扎实的算法基本功和丰富的解题思路方面的积累（这也是小册从开篇到现在一直在引导大家做的事情）——这些东西是需要你真刀真枪地花时间和算法面对面搏斗才能沉淀下来的“内力”，唯有它能够以不变应万变。

本环节在整本小册中的作用，是对前述知识体系的补充，意在帮助同学们**扩展解题思路、强化做题手感**。所谓“大厂真题”，只不过是用来试炼学习效果、提升综合能力的“教具”，它们的任务是帮你快速建立起实战场景下的解题自信，而不是为了劝退或者炫技。

在接下来几节的学习过程中，最要紧的是保持住学习的平常心——不要被标题中高大上的公司 Title 给吓到了，要知道这些题对你来说终究会是小菜一碟。你需要做的仅仅是专注于题目和题目背后的思路，将题目对自己的价值最大化，扎扎实实地跑完这一场算法马拉松的最后一公里。

大家加油！

最长回文子串问题

> 题目描述：给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

> 示例 1：
> 输入: "babad"
> 输出: "bab"
> 注意: "aba" 也是一个有效答案。

> 示例 2：
> 输入: "cbbd"
> 输出: "bb"

命题关键字：字符串、动态规划

思路分析

这道题最直接的思路仍然是暴力解法：
定义两个指针 `i` 和 `j`，用这两个指针嵌套两层循环，尝试枚举出给定字符串对应的所有可能的子序列，判断每一个子序列是否回文：若回文且长度已经超越当前的长度最大值，则更新长度最大值，并记录该子串的两个端点。当所有的子串枚举结束时，我们也就得到了最大长度的回文子串。

枚举子串需要两层循环，对应的复杂度是 `O(n^2)`；判断是否回文，又额外需要 `O(n)` 的开销。因此，这个暴力解法的时间复杂度就是 `O(n^3)`。

由于这个复杂度过于辣鸡，我们看看就行了。下面抛弃本能，恢复理智，我们结合前面做题的经验，重新来看这道题。

题干中的“最长”二字，表明了这是一道“求最值”型问题。前面我们说过，看到最值，就要把动态规划调度进可用解题工具里。
继续往下分析，发现这道题中，较长回文子串中可能包含较短的回文子串（最优子结构）；若按照暴力解法来做，多次遍历的过程中不可避免地会涉及到对同一个回文子串的重复判断（重叠子问题），因此，这道题用动态规划求解是比较合理的。

这道题中，我们拿到的原始素材是一个字符串序列，符合“序列型”动态规划的特征。大家现在已经知道，对于序列型动态规划，我们总是需要以它的索引为线索去构造一维或二维的状态数组。对于这道题来说，由于定位任意子串需要的是两个索引，因此我们的状态数组应该是一个二维数组：

```
// 初始化一个二维数组
let dp = [];
const len = s.length
for (let i = 0; i < len; i ++) {
    dp[i] = [];
};
```

由于 `i` 和 `j` 分别表示子串的两个端点，只要我们明确了这两个值，就能间接地求出子串的长度。因此 `dp[i][j]` 不必额外记录长度这个状态，只需要记录该区间内的字符串是否回文。这里我们把回文记为 1（或 `true`），不回文记为0（或 `false`）。

按照这个思路走下去，我们需要关注到的无疑就是字符串的两个端点 `s[i]` 和 `s[j]` 了。当遍历到一对新的端点的时候，有以下两种可能的状态转移情况：

1. `s[i] === s[j]` 。这种情况下，只要以 `s[i+1]` 和 `s[j-1]` 为端点的字符串是回文字符串，那么 `dp[i][j] = 1` 就成立，否则 `dp[i][j] = 0` 。
2. `s[i] !== s[j]` 。这种情况下，一定有 `dp[i][j]=0` 。

到这里，我们也就明确到了这道题的状态转移方程，这里我用编码表达如下：

```
if(s[i] === s[j]) {
    dp[i][j] = dp[i+1][j-1]
} else {
    dp[i][j] = 0
}
```

找出了状态转移方程，现在来找边界值。这里大家需要注意的是：**如果在一个序列中，涉及到了 `i` 、 `j` 两个索引，那么一定要关注到 `i===j` 这种特殊情况**。在这道题中，由于 `i===j` 时， `dp[i][i]` 对应的是一个单独的字母，单独的字母必然回文（长度为1），因此 `dp[i][i] = 1` 就是这道题的边界值（或者说初始值）。

现在，明确了初始值，明确了状态转移方程，我们来写代码（注意看注释）：

编码实现

```
/**
 * @param {string} s
 * @return {string}
 */
const longestPalindrome = function(s) {
    const dp = [];
    // 缓存字符串长度
    const len = s.length
    // 初始化状态二维数组
    for (let i = 0; i < len; i ++) {
        dp[i] = [];
    };

    // 初始化最长回文子串的两个端点值
    let st = 0, end=0
    // 初始化最长回文子串的初始值为1
    for(let i=0;i<len;i++) {
```

```
            dp[i][i] = 1
        }
        // 这里为了降低题目的复杂度，我们预先对悬念比较小的 s[i][i+1] 也做了处理
        for(let i=0;i<len-1;i++){
            if(s[i]===s[i+1]) {
                dp[i][i+1] = 1
                st = i
                end = i+1
            }
        }

        // n 代表子串的长度，从3开始递增
        for(let n=3;n<=len;n++) {
            // 下面的两层循环，用来实现状态转移方程
            for(let i=0;i<=len-n;i++) {
                let j = i+n-1
                if(dp[i+1][j-1]) {
                    if(s[i]===s[j]){
                        // 若定位到更长的回文子串，则更新目标子串端点的索引值
                        dp[i][j] = 1
                        st = i
                        end = j
                    }
                }
            }
        }
        // 最后依据端点值把子串截取出来即可
        return s.substring(st,end+1);
    }
```

从前序（先序）与中序遍历序列构造二叉树

> 题目描述：根据一棵树的前序遍历与中序遍历构造二叉树。

> 注意: 你可以假设树中没有重复的元素。
> 例如，给出

> 前序遍历 preorder = [3,9,20,15,7]
> 中序遍历 inorder = [9,3,15,20,7]
> 返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

命题关键字：二叉树、前序、中序、遍历序列特征、递归

思路分析

这道题非常非常非常非常经典。
第一次见到它，没思路，是正常的；第二次见到它，写不顺，也是正常的——对于经典的题目，我们未必一定要完全靠自己的聪明才智去解决它（直接看答案一点也不丢人），但一定要追求一个**熟练度**（你得对答案有充分的理解和把握，能靠条件反射来做题）。

这道题解题的一个切入点，就是**前序遍历序列和中序遍历之间的关系**。

我们假设前序遍历序列中的元素分别为 p1、p2.....pn，中序遍历序列中的元素分别为 i1、i2.....in。那么两个序列之间就有以下关系：

```
          root   |<-左子树->|   |<- 右子树 ->|
           ↓
前序序列   p1     p2......pk    p(k+1)......pn

           |<- 左子树 ->|        root   |<- 右子树 ->|
                                 ↓
中序序列   i1 i2......i(k-1)     ik    i(k+1)......in
```

它们之间的关系蕴含着两个重要的规律：

1. 前序序列头部的元素 `p1` ，一定是当前二叉树的根结点（想一想，为什么？）。
2. 中序遍历序列中，以二叉树的根结点为界划分出的两个子序列，分别对应着二叉树的左子树和二叉树的右子树。

基于以上两个规律，我们不难明确这道题的解题思路：在中序序列中定位到根结点(`p1`)对应的坐标，然后基于这个坐标划分出左右子树对应的两个子序列，进而明确到左右子树各自在前序、中序遍历序列中对应的索引区间，由此构造左右子树。

以上面的示意简图为例，根结点(p1)在中序序列中的坐标索引为 k ，于是左子树的结点个数就可以通过计算得出：

```
numLeft = k - 1
```

这里为了确保逻辑的通用性，我们把前序序列当前范围的头部索引记为 preL ，尾部索引记为 preR ；把中序序列当前范围的头部索引记为 inL ，尾部索引记为 inR 。
那么左子树在前序序列中的索引区间就是 [preL+1,preL+numLeft] ，在中序序列中的索引区间是 [inL, k-1] ；右子树在前序序列的索引区间是 [preL+numLeft+1, preR] ，在中序序列中的索引区间是 [k+1, inR] 。

此时我们会发现，基于左子树和右子树各自对应的前序、中序子序列，我们完全可以直接重复执行上面的逻辑来定位到左右子树各自的根结点和子树的序列区间。通过反复 重复这套定位+构造 的逻辑，我们就能够完成整个二叉树的构建。

二叉树类题目中的重复逻辑，90%都是用递归来完成的。下面我就基于递归思想来完成这道题的编码示范（注意看注释里的解析）：

编码实现

```
/**
 * 预定义树的结点结构.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {number[]} preorder
 * @param {number[]} inorder
 * @return {TreeNode}
 */
const buildTree = function(preorder, inorder) {
    // 缓存结点总个数（遍历序列的长度）
    const len = preorder.length
    // 定义构造二叉树结点的递归函数
    function build(preL, preR, inL, inR) {
        // 处理越界情况
        if(preL > preR) {
```

```
            return null
        }
        // 初始化目标结点
        const root = new TreeNode()
        // 目标结点映射的是当前前序遍历序列的头部结点（也就是当前范围的根结点）
        root.val = preorder[preL]
        // 定位到根结点在中序遍历序列中的位置
        const k = inorder.indexOf(root.val)
        // 计算出左子树中结点的个数
        const numLeft = k - inL
        // 构造左子树
        root.left = build(preL+1, preL+numLeft, inL, k-1)
        // 构造右子树
        root.right = build(preL+numLeft+1, preR, k+1, inR)
        // 返回当前结点
        return root
    }
    // 递归构造二叉树
    return build(0, len-1, 0, len-1)
};
```

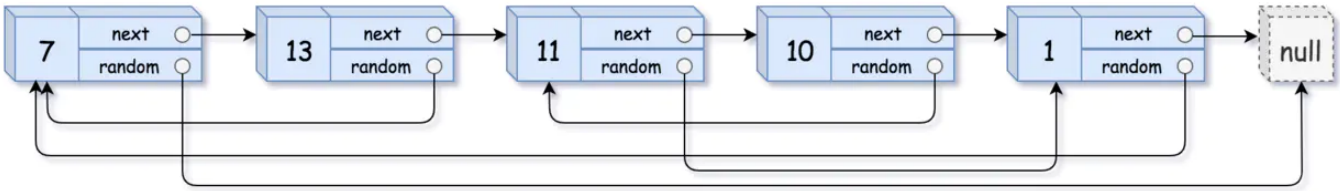
> 请思考：如果把题目中的“前序”改成“后序”，这道题应该怎么做？

> 提示：不妨先写出题示二叉树对应的后序遍历序列，然后比猫画虎，寻找它和中序遍历之间的关系。答案其实就藏在我们的题解中，相信你一定能挖掘出新的规律，加油呀~~

复制带随机指针的链表

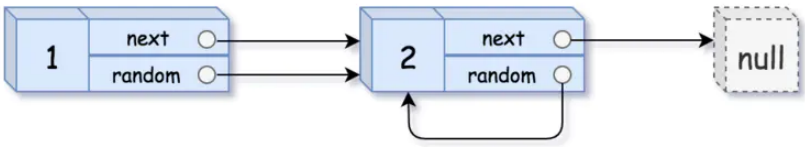
> 题目描述：给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。要求返回这个链表的 深拷贝。
> 我们用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：
> val：一个表示 Node.val 的整数。
> random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null 。

> 示例1：



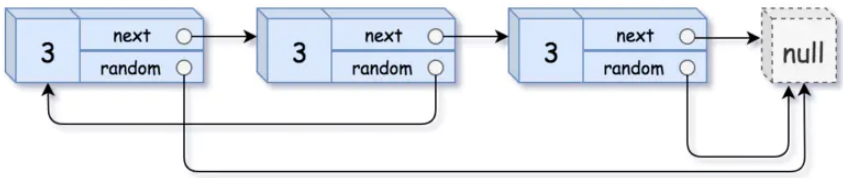

```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

> 示例2：




```
输入：head = [[1,1],[2,1]]
输出：[[1,1],[2,1]]
```

> 示例3：




```
输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]
```

> 示例4：

输入：head = []
输出：[]
解释：给定的链表为空（空指针），因此返回 null。

命题关键字：数据结构、链表、哈希表

思路分析

这道题考的是数据结构相关的基础知识和进阶操作。
关于基础知识，我们前面已经叨叨了不少了，这里不再赘述。这道题我着重要讲的是面向实战场景的几个解题突破口：

1. 啥是深拷贝：在这道题里，深拷贝是相对于引用拷贝来说的，对于 JS 中的对象 a 和对象 b ，假如我们单纯赋值：

```
a = b
```

那么 a 和 b 其实是指向了同一个引用，这就是引用拷贝。深拷贝的意思是说 a 和 b 的内容相同，但是占据两块不同的内存空间，也就是拥有两个不同的引用。对于链表中的 Node 对象（假设对象中的属性分别是数据域 val 和指针域 next ）来说，我们可以这样做：

```
// 先开辟一块新的内存空间
const copyNode = new Node()
// copy旧结点的值
copyNode.val = curr.val
// copy旧结点的next指针
copyNode.next = curr.next ? new Node() : null
```

2. 如何处理深拷贝过程中的结点关系：笔者在这里最推荐的一种做法是用 Map 结构：在这道题中，除了 next 指针还有 random 指针，结点关系相对复杂，这就意味着我们在处理结点关系的过程中必然会遇到“根据原结点定位它对应的copy结点”这样的需求。 Map 结构可以帮我们做到这一点。
3. next 指针和 random 指针各自应该如何处理：我们可以先走一遍普通链表（也就是没有 random 指针）的复制流程。在这个过程中，一方面是完成对结点的复制+存储工作，另一方面也用 next 指针把新链表串了起来。这一步做完之后，新链表和老链表之间唯一的区别就在于 random 指针了。此时我们只需要**同步遍历新旧两个链表**，把 random 的指向映射到新链表上去即可。

基于对以上三个问题的探讨，我们可以有以下编码（注意注释里的解析）：

编码实现

```
/**
 * // Definition for a Node.
 * function Node(val, next, random) {
 *    this.val = val;
 *    this.next = next;
 *    this.random = random;
 * };
 */

/**
 * @param {Node} head
 * @return {Node}
 */
const copyRandomList = (head) => {
    // 处理边界条件
    if (!head) return null
    // 初始化copy的头部结点
    let copyHead = new Node()
    // 初始化copy的游标结点
    let copyNode = copyHead
    // 初始化hashMap
    const hashMap = new Map()
    let curr = head
    // 首次循环，正常处理链表的复制
    while (curr) {
        copyNode.val = curr.val
        copyNode.next = curr.next ? new Node() : null
        hashMap.set(curr, copyNode)
        curr = curr.next
        copyNode = copyNode.next
    }
    // 将游标复位到head
    curr = head
    // 将copy链表的游标也复位到copyHead
    copyNode = copyHead
    // 再搞个循环，特殊处理random关系
```

```
    while (curr) {
        // 处理random的指向
        copyNode.random = curr.random ? hashMap.get(curr.random) : null
        // copyNode 和 curr 两个游标一起前进
        copyNode = copyNode.next
        curr = curr.next
    }

    // 注意这里返回的是copyHead而不是head
    return copyHead
};
```