本节的命题风格是“大杂烩”：文中涉及到的题目本身并不难，但题目与题目之间的知识点跨度会比较大，目的是考验大家对知识点的熟练度和整合知识点的能力。

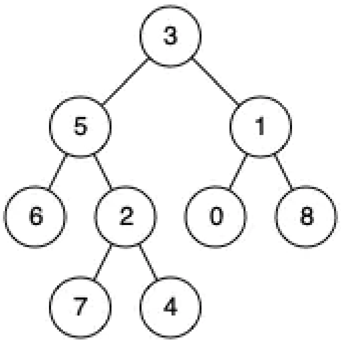
> 注：此处的“命题风格”仅出于笔者个人对课程设计的考虑，并非对腾讯公司命题思路的预测/总结。准备背题目的同学都醒醒。

寻找二叉树的最近公共祖先

> 题目描述： 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



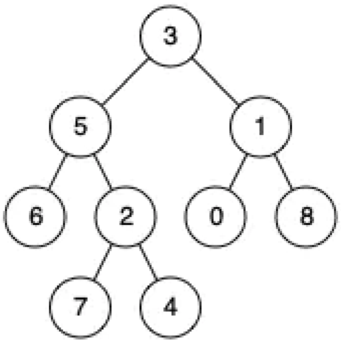
> 示例 1:
> 输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
> 输出: 3
> 解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

> 示例 2:
> 输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
> 输出: 5
> 解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

命题关键字：二叉树、递归

思路分析

这道题非常经典。很多人（包括我）第一次读完题目的时候，脑子里都是一片空白——确实，这道题的题干并不能够给我们提供什么有效的启发性信息。不过不要慌，当题干都是屁话时，我们不妨试试从“示例”中寻找答案：



题干中一直在强调“祖先结点”、“树的深度”等概念，这可能会误导一部分同学情不自禁地代入“爹找儿子”这种思维模式，然后陷入僵局。如果你不幸中招，别忘了：

> 虽然编码的时候我们实现的确实是“爹找儿子”，但是在规则摸索阶段，“儿子找爹”这种思维模式会更加人性化。

不管是爹找儿子，还是儿子找爹，我们都必须首先明确儿子和爹之间的关系有哪些，从而尝试去将不同的关系和“公共祖先”这个概念建立关联。这些信息，我们都可以从题目的示例中挖掘出来。
现在我按照“儿子向爹汇报”这个思路，一层层往上溯源，尝试枚举不同的父子关系形式。

> 注：下文所提及的“有效汇报”指的就是“爸爸我这里有p或者q”这样式儿的汇报哈

假如说我要寻找的是 `6` 和 `2` 的最近公共祖先，那么这中间出现的儿子和爹之间的关系就有以下几种：

1. 对于 `5` 这个结点来说，它的左边和右边各有一个目标儿子给他作有效汇报， `5` 也确实就是这俩目标儿子的最近公共祖先。

2. 对于 `3` 这个结点来说，由于 `6` 和 `2` 只存在于它的左孩子上，所以它得到的有效汇报只有1个。同时 `3` 本身又并不等同于 `6` 或者 `2` ，因此 `3` 不是最近公共祖先。
   这里我强调了“不等同”，那么相应地一定会有“等同”的情况——假如我们要寻找的目标结点是 `5` 和 `6` ，那么对于 `5` 来说，即使只有一侧的孩子结点给它作了有效的汇报，也不影响它作为两个结点的最近公共祖先而存在（因为它自己既是儿子也是爸爸）。

3. 对于 `1` 这个结点来说，它的左孩子和右孩子上都没有目标结点，这意味着它拿到的所有“汇报”就都是无效的，因此 `1` 不是最近公共祖先。

分析至此，我们发现了一个明显的规律：最近公共祖先和有效汇报个数之间，有着非常强烈的关联。
那么“有效汇报个数”就成了我们做题的抓手。由于一个结点最多有两个孩子，它拿到的有效汇报个数也无非只有0、1、2这三种可能性，我们逐个来看：

1. 若有效汇报个数为0，则 `p` 和 `q` 完全不存在与当前结点的后代中，当前结点一定不是最近公共祖先（对应示例二叉树中 `p=6，q=2` 时， `6、2、1` 之间的关系）。

2. 若有效汇报个数为2，则意味着 p 和 q 所在的两个分支刚好在当前结点交错了，当前结点就是 p 和 q 的最近公共祖先（对应示例二叉树中 p=6, q=2 时， 6、2、5 之间的关系）。

3. 若有效汇报个数为1，这里面蕴含着三种情况：
   a. 当前结点的左子树/右子树中，**包含了 p 或者 q 中的一个**。此时我们需要将 p 或者 q 所在的那棵子树的根结点作为有效结点上报，继续向上去寻找 p 和 q 所在分支的交错点。

   b. 当前结点的左子树/右子树中，**同时包含了 p 和 q** 。在有效汇报数为1的前提下，这种假设只可能对应一种情况，**那就是 p 和 q 之间互为父子关系**。此时我们仍然是需要将 p 和 q 所在的那个子树的根结点（其实就是 p 或者 q 中作为爸爸存在那个）作为有效结点给上报上去。

结合上面三种情况，我们可以进一步分析出以下结论：

1. 若有效汇报个数为2，直接返回当前结点
2. 若有效汇报个数为1，返回1所在的子树的根结点
3. 若有效汇报个数为0，则返回空（空就是无效汇报）

我们把这个判定规则，揉进二叉树递归的层层上报的逻辑里去，就得到了这道题的答案：

编码实现

```
/**
 * 二叉树结点的结构定义如下
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
const lowestCommonAncestor = function(root, p, q) {
    // 编写 dfs 逻辑
    function dfs(root) {
        // 若当前结点不存在（意味着无效）或者等于p/q（意味着找到目标），则直接返回
        if(!root || root === p || root === q) {
            return root
        }
```

```
        // 向左子树去寻找p和q
        const leftNode = dfs(root.left)
        // 向右子树去寻找p和q
        const rightNode = dfs(root.right)
        // 如果左子树和右子树同时包含了p和q，那么这个结点一定是最近公共祖先
        if(leftNode && rightNode) {
            return root
        }
        // 如果左子树和右子树其中一个包含了p或者q，则把对应的有效子树汇报上去，等待
        return leftNode || rightNode
    }

    // 调用 dfs 方法
    return dfs(root)
};
```

寻找两个正序数组的中位数

> 题目描述：给定两个大小为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。
> 请你找出这两个正序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。
> 你可以假设 nums1 和 nums2 不会同时为空。

> 示例 1: nums1 = [1, 3]
> nums2 = [2]
> 则中位数是 2.0

> 示例 2:
> nums1 = [1, 2]
> nums2 = [3, 4]
> 则中位数是 (2 + 3)/2 = 2.5

命题关键字：二分思想、数学问题

思路分析

在做这道题之前，大家先记住一个规律：

> 题目中若要求 `log` 级别的时间复杂度，则优先使用二分法解题

回到这道题上来，既然题目要求 `log` 级别的时间复杂度，我们首要的解题思路就不应该再是“遍历”，而应该是“切割”。

理解中位数的取值思路

接下来就需要思考切割的手法了。大家想想，如果只允许你用切割的方式来定位两个正序数组的中位数，你会怎么办？是不是应该首先想到从**元素的数量**上入手？

具体来说，假如我这里需要求解的是这样两个数组：

```
nums1 = [1, 3, 5, 7, 9]
nums2 = [2, 4, 6, 8, 10]
```

我要求解的中位数的范围是10个数，那么假如我在某个合适的位置分别切割了 `nums1` 和 `nums2` ：

```
[1, 3, 5,| 7, 9]
|<- s1 ->|

[2, 4, |6, 8, 10]
|<-s2->|
```

使得 `s1+s2` ，刚好就是10个数里面按正序排布的前5个数。这样我其实只需要关心切割边界的这些值就可以了：

```
       L1  R1
[1, 3, 5,| 7, 9]
|<- s1 ->|

    L2  R2
[2, 4, |6, 8, 10]
|<-s2->|
```

这个例子中，数组总长度是10，10是偶数。偶数个数字的中位数，按照定义需要取中间两个元素的平均值。而“中间两个元素”，一定分别是 `L1` 和 `L2` 中的较大值，以及 `R1` 和 `R2` 中的最小值（这个结论无需多言，你品品就出来了）：

```
// 取 L1 和 L2 中的较大值
const L = L1 > L2 ? L1 : L2
// 取 R1 和 R2 中的较小值
const R = R1 < R2 ? R1 : R2
// 计算平均值
return  (L + R)/2
```

此时假如给其中一个数组增加一个元素，让两个数组的长度和变为奇数：

```
       L1  R1
[1, 3, 5,| 7, 9, 11]
|<- s1 ->|

    L2  R2
[2, 4, |6, 8, 10]
|<-s2->|
```

那么中位数的取值就更简单了，我们只需要取 `R1` 和 `R2` 中的较小值即可：

```
const median = (R1 < R2) ? R1 : R2
```

到此为止，大家就对“切割法”下的中位数取值思路有了基本的了解。
以上我们所有的讨论，都是建立在 `nums1` 和 `nums2` 的分割点已知的前提下。实际上，对这道题来说，分割点的计算才是它真正的难点。

要解决这个问题，就需要请出二分思想了。

二分思想确定分割点

我们回头看这个数组

```
nums1 = [1, 3, 5, 7, 9]
nums2 = [2, 4, 6, 8, 10]
```

在不口算的情况下，没有人会知道 `R1` 、 `R2` 到底取在哪个位置是比较合理的，你只知道一件事——我需要让 `nums1切割后左侧的元素个数+nums2切割后左侧元素的个数===两个数组长度和的一半` 。
我们先用编码语言来表达一下这个关系：

```
// slice1和slice2分别表示R1的索引和R2的索引
slice1 + slice2 === Math.floor((nums1.length + nums2.length)/2)
```

nums1、nums2 的长度是已知的，这也就意味着只要求出 slice1 和 slice2 中的一个，另一个值就能求出来了。

因此我们的大方向先明确如下：

> 用二分法定位出其中一个数组的slice1，然后通过做减法求出另一个数组的slice2

“其中一个数组”到底以 nums1 为准还是以 nums2 为准？答案是以长度较短的数组为准，这样做可以减小二分计算的范围，从而提高我们算法的效率，所以我们代码开局就是要校验两个数组的长度大小关系：

```
const findMedianSortedArrays = function(nums1, nums2) {
    const len1 = nums1.length
    const len2 = nums2.length
    // 确保直接处理的数组（第一个数组）总是较短的数组
    if(len1 > len2) {
        return findMedianSortedArrays(nums2, nums1)
    }
    ...
}
```

从而确保较短的数组始终占据 nums1 的位置，后续我们就拿 nums1 开刀做二分。

这里我们假设 nums1 和 nums2 分别是以下两个数组：

```
nums1 = [5, 6, 7]
nums2 = [1, 2, 4, 12]
```

用二分法做题，首先需要明确二分的两个端点。在没有任何多余线索的情况下，我们只能把二分的端点定义为 nums1 的起点和终点：

```
// 初始化第一个数组二分范围的左端点
let slice1L = 0
// 初始化第一个数组二分范围的右端点
let slice1R = len1
```

基于此去计算 `slice1` 的值：

```
slice1 = Math.floor((slice1R - slice1L)/2) + slice1L
```

然后通过做减法求出 `slice2` ：

```
slice2 = Math.floor(len/2) - slice1
```

第一次二分，两个数组分别被分割为如下形状：

```
        L1   R1
nums1 = [5, |6, 7]

            L2  R2
nums2 = [1, 2, |4, 12]
```

如何确认你的二分是否合理？标准只有一个——**分割后，需要确保左侧的元素都比右侧的元素小**，也就是说你的两个分割线要间接地把两个数组按照正序分为两半。这个标准用变量关系可以表示如下：

```
L1 <= R1
L1 <= R2
L2 <= R1
L2 <= R2
```

由于数组本身是正序的，所以 `L1 <= R1` 、 `L2 <= R2` 是必然的，我们需要判断的是剩下两个不等关系：

若发现 `L1 > R2` ，则说明 `slice1` 取大了，需要用二分法将 `slice1` 适当左移；若发现 `L2 > R1` ，则说明 `slice1` 取小了，需要用二分法将 `slice1` 适当右移：

```
// 处理L1>R2的错误情况
 if(L1 > R2) {
    // 将slice1R左移，进而使slice1对应的值变小
    slice1R = slice1 - 1
} else if(L2 > R1) {
    // 反之将slice1L右移，进而使slice1对应的值变大
```

```
    slice1L = slice1 + 1
}
```

只有当以上两种偏差情况都不发生时，我们的分割线才算定位得恰到好处，此时就可以执行取中位数的逻辑了：

```
// len表示两个数组的总长度
if(len % 2 === 0) {
    // 偶数长度对应逻辑（取平均值）
    const L = L1 > L2 ? L1 : L2
    const R = R1 < R2 ? R1 : R2
    return  (L + R)/2
} else {
    // 奇数长度对应逻辑（取中间值）
    const median = (R1 < R2) ? R1 : R2
    return median
}
```

我们把以上的整个分析用代码串起来，就有了这道题的答案：

编码实现

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
const findMedianSortedArrays = function(nums1, nums2) {
    const len1 = nums1.length
    const len2 = nums2.length
    // 确保直接处理的数组（第一个数组）总是较短的数组
    if(len1 > len2) {
        return findMedianSortedArrays(nums2, nums1)
    }
    // 计算两个数组的总长度
    const len = len1 + len2
```

```
// 初始化第一个数组“下刀”的位置
let slice1 = 0
// 初始化第二个数组“下刀”的位置
let slice2 = 0
// 初始化第一个数组二分范围的左端点
let slice1L = 0
// 初始化第一个数组二分范围的右端点
let slice1R = len1
let L1, L2, R1, R2
// 当slice1没有越界时
while(slice1 <= len1) {
    // 以二分原则更新slice1
    slice1 = Math.floor((slice1R - slice1L)/2) + slice1L
    // 用总长度的1/2减去slice1，确定slice2
    slice2 = Math.floor(len/2) - slice1 // 计算L1、L2、R1、R2
    const L1 = (slice1===0)? -Infinity : nums1[slice1-1]
    const L2 = (slice2===0)? -Infinity : nums2[slice2-1]
    const R1 = (slice1===len1)? Infinity : nums1[slice1]
    const R2 = (slice2===len2)? Infinity: nums2[slice2]

    // 处理L1>R2的错误情况
    if(L1 > R2) {
        // 将slice1R左移，进而使slice1对应的值变小
        slice1R = slice1 - 1
    } else if(L2 > R1) {
        // 反之将slice1L右移，进而使slice1对应的值变大
        slice1L = slice1 + 1
    } else {
        // 如果已经符合取中位数的条件（L1<R2&&L2<R1)，则直接取中位数
        if(len % 2 === 0) {
            const L = L1 > L2 ? L1 : L2
            const R = R1 < R2 ? R1 : R2
            return  (L + R)/2
        } else {
            const median = (R1 < R2) ? R1 : R2
            return median
        }
```

```
            }

        }
        return -1
};
```

拓展

假如把题目中的 `O(log(m+n))` 改为 `O(m+n)` ，你会怎样做?

“粉刷房子”问题

> 题目描述: 假如有一排房子，共 n 个，每个房子可以被粉刷成红色、蓝色或者绿色这三种颜色中的一种，你需要粉刷所有的房子并且使其相邻的两个房子颜色不能相同。
> 当然，因为市场上不同颜色油漆的价格不同，所以房子粉刷成不同颜色的花费成本也是不同的。每个房子粉刷成不同颜色的花费是以一个 n x 3 的矩阵来表示的。
> 例如，costs[0][0] 表示第 0 号房子粉刷成红色的成本花费；costs[1][2] 表示第 1 号房子粉刷成绿色的花费，以此类推。请你计算出粉刷完所有房子最少的花费成本。

> 注意： 所有花费均为正整数。

> 示例： 输入: [[17,2,17],[16,16,5],[14,3,19]]
> 输出: 10
> 解释: 将 0 号房子粉刷成蓝色，1 号房子粉刷成绿色，2 号房子粉刷成蓝色。
> 最少花费: 2 + 5 + 3 = 10。

命题关键字：动态规划、滚动数组

思路分析

这道题的特征非常肤浅，从概念的角度来说，动态规划的两个特征全部命中（如果你不知道我在说啥，建议复习小册第22、23节）；从技巧的角度来说，“求最值”这个信号也在疯狂暗示你用动态规划来解决它。

对于最值型动态规划，我们最常用的思路仍然是动态规划专题中首推的“倒推”法。由于这个方法笔者已经重复地讲过太多次了，我们就不再在真题训练环节予以过多的表述（这道题的重点也不在这里）。结合“倒推”法，我们可以得出题目对应的状态转移方程是：

```
f[i][x] = Math.min(f[i-1][x以外的索引1号], f[i-1][x以外的索引2号]) + costs[:
```

其中 `f[i][x]` 对应的是当粉刷到第 `i` 个房子时，使用第 `x` （x=0、1、2）号油漆对应的总花费成本的最小值。

状态的初始值，就是当 `i=0` 时对应的三个值：

```
f[0][0] = costs[0][0]
f[0][1] = costs[0][1]
f[0][2] = costs[0][2]
```

`f[0][0]` 、 `f[0][1]` 、 `f[0][2]` 分别表示当粉刷到第0个房子时，对它使用0号、1号、2号油漆对应的总花费成本。此时由于只粉刷了一个房子，所以总花费成本就等于房子本身的花费成本。
基于以上两个结论，我们可以有如下的初步编码：

编码实现-基础版

```
/**
 * @param {number[][]} costs
 * @return {number}
 */
const minCost = function(costs) {
    // 处理边界情况
    if(!costs || !costs.length) return 0
    // 缓存房子的个数
    const len = costs.length
    // 初始化状态数组（二维）
    const f = new Array(len)
    for(let i=0;i<len;i++) {
        f[i] = new Array(3)
    }
    // 初始化状态值
    f[0][0] = costs[0][0]
    f[0][1] = costs[0][1]
    f[0][2] = costs[0][2]
    // 开始更新刷到每一个房子时的状态值
    for(let i=1;i<len;i++) {
      // 更新刷到当前房子时，给当前房子选用第0种油漆对应的最小总价
      f[i][0] = Math.min(f[i-1][1], f[i-1][2]) + costs[i][0]
```

```
        // 更新刷到当前房子时，给当前房子选用第1种油漆对应的最小总价
        f[i][1] = Math.min(f[i-1][2], f[i-1][0]) + costs[i][1]
        // 更新刷到当前房子时，给当前房子选用第2种油漆对应的最小总价
        f[i][2] = Math.min(f[i-1][1], f[i-1][0]) + costs[i][2]
    }
    // 返回刷到最后一个房子时，所有可能出现的总价中的最小值
    return Math.min(f[len-1][0], f[len-1][1], f[len-1][2])
};
```

如果你写出了以上答案，而你的面试官又是一个在算法方面稍有见识的人，他就会问你：这道题的空间复杂度能否进一步优化？
此时，没有读过算法小册的同学，他以为自己做完了整道题，其实好戏才刚刚开始。
而认真研读过小册第23节的同学，他认为这样的追问合情合理，甚至在一开始准备好了思路，就等面试官把舞台交给自己。只见他三下五除二，就变出了一个叫"滚动数组"的东西，把这道题的空间复杂度碾了个稀碎：

编码实现-优化版

```
/**
 * @param {number[][]} costs
 * @return {number}
 */
const minCost = function(costs) {
    // 处理边界情况
    if(!costs || !costs.length) return 0
    // 缓存房子的个数
    const len = costs.length
    // 开始更新状态
    for(let i=1;i<len;i++) {
        // now表示粉刷到当前房子时对应的价格状态
        const now = costs[i]
        // prev表示粉刷到上一个房子时的价格状态
        const prev = costs[i-1]
        // 更新当前状态下，刷三种油漆对应的三种最优价格
        now[0] += Math.min(prev[1], prev[2])
        now[1] += Math.min(prev[0], prev[2])
        now[2] += Math.min(prev[1], prev[0])
    }
```

```
    // 返回粉刷到最后一个房子时，总价格的最小值
    return Math.min(costs[len-1][0], costs[len-1][1], costs[len-1][2])
};
```

倘若对“基础版”代码稍作分析，你就会发现，其实我们每次更新 `f[i]` 时，需要的仅仅是 `f[i-1]` 对应的状态而已，因此我们只需要确保一个数组中总是能保持着有效的 `f[i-1]` 即可。
这样的特征，符合“滚动数组”的使用场景。在这道题中，我们直接滚动了题目中原有的 `costs` 变量，将空间复杂度缩减了一个量级。

“滚动数组”是什么、怎么用？如果你对此心怀疑惑，请耐下心来，复习一下小册的第23节吧~^_^