数据结构和算法绪论

前言

数据结构和算法可以让程序员脱胎换骨，刷算法题可以帮助我们通过面试和笔试，找到梦寐以求的工作，进入一线大厂或者拿高薪。怎么刷题呢？LeetCode 上有 2000 多道题目，难道要全部刷完？

国内的一线大厂在面试和笔试的时候会考察什么样的题目？Mark 老师抽取 8 个公司 Top20 的高频题目做了一个合并统计：

	字节	微软	美团	阿里巴巴	快手	腾讯	百度	京东
题号	3.无重复字符的最长子串	215.数组中的第K个最大元	206.反转链表	146.LRU缓存机制	206.反转链表	206.反转链表	206.反转链表	206.反转链表
	25.K个一组翻转链表	206.反转链表	补充题4.手撕快速排序	1.两数之和	88.合并两个有序数组	146.LRU缓存机制	215.数组中的第K个最大元	142.环形链表 II
	206.反转链表	236.二叉树的最近公共祖先	141.环形链表	3.无重复字符的最长子串	3.无重复字符的最长子串	补充题4.手撕快速排序	15.三数之和	补充题4.手撕快速排序
	215.数组中的第K个最大元	48.旋转图像	88.合并两个有序数组	206.反转链表	53.最大子序和	53.最大子序和	141.环形链表	4.寻找两个正序数组的中位
	146.LRU缓存机制	124.二叉树中的最大路径和	102.二叉树的层序遍历	215.数组中的第K个最大元	102.二叉树的层序遍历	21.合并两个有序链表	补充题4.手撕快速排序	102.二叉树的层序遍历
	103.二叉树的锯齿形层次遍	53.最大子序和	21.合并两个有序链表	15.三数之和	补充题4.手撕快速排序	704.二分查找	46.全排列	215.数组中的第K个最大元
	121.买卖股票的最佳时机	224.基本计算器	92.反转链表 II	102.二叉树的层序遍历	146.LRU缓存机制	415.字符串相加	20.有效的括号	94.二叉树的中序遍历
	15.三数之和	151.翻转字符串里的单词	142.环形链表 II	165.比较版本号	21.合并两个有序链表	3.无重复字符的最长子串	1.两数之和	55.跳跃游戏
	160.相交链表	543.二叉树的直径	3.无重复字符的最长子串	94.二叉树的中序遍历	144.二叉树的前序遍历	8.字符串转换整数(atoi)	94.二叉树的中序遍历	21.合并两个有序链表
	1.两数之和	297.二叉树的序列化与反序	20.有效的括号	53.最大子序和	141.环形链表	215.数组中的第K个最大元	142.环形链表 II	15.三数之和
	236.二叉树的最近公共祖先	91.解码方法	215.数组中的第K个最大元	20.有效的括号	160.相交链表	470.用Rand7()实现 Rand10	104.二叉树的最大深度	1.两数之和
	200.岛屿数量	39.组合总和	143.重排链表	5.最长回文子串	142.环形链表 II	234.回文链表	53.最大子序和	83.删除排序链表中的重复
	42.接雨水	207.课程表	124.二叉树中的最大路径和	141.环形链表	46.全排列	1.两数之和	3.无重复字符的最长子串	415.字符串相加
	33.搜索旋转排序数组	450.删除二叉搜索树中的	718.最长重复子数组	23.合并K个排序链表	215.数组中的第K个最大元	141.环形链表	4.寻找两个正序数组的中位	46.全排列
	53.最大子序和	146.LRU缓存机制	1.两数之和	169.多数元素	92.反转链表 II	153.寻找旋转排序数组中的	146.LRU缓存机制	704.二分查找
	54.螺旋矩阵	47.全排列 II	232.用栈实现队列	46.全排列	103.二叉树的锯齿形层次遍	20.有效的括号	102.二叉树的层序遍历	25.K个一组翻转链表
	21.合并两个有序链表	22.括号生成	704.二分查找	415.字符串相加	121.买卖股票的最佳时机	5.最长回文子串	19.删除链表的倒数第N个	剑指Offer 53- II.0~n-1中
	31.下一个排列	98.验证二叉搜索树	146.LRU缓存机制	236.二叉树的最近公共祖先	415.字符串相加	4.寻找两个正序数组的中位	199.二叉树的右视图	146.LRU缓存机制
	88.合并两个有序数组	141.环形链表	剑指Offer 22.链表中倒数第	144.二叉树的前序遍历	23.合并K个排序链表	160.相交链表	5.最长回文子串	876.链表的中间结点
	300.最长上升子序列	110.平衡二叉树	54.螺旋矩阵	121.买卖股票的最佳时机	20.有效的括号	补充题6.手撕堆排序	160.相交链表	剑指Offer 10- II.青蛙跳台
1	Y		Y	Y		Y	Y	Y
3	Y		Y	Y	Y	Y	Y	
4						Y	Y	Y
5				Y		Y	Y	
8						Y		
15	Y			Y			Y	Y
19							Y	
20			Y	Y	Y	Y	Y	
21	Y		Y		Y	Y		Y

发现存在着很大的重复性，有些题目甚至所有的公司都有考察，比如 LeetCode-206.反转链表。同时这些题目所属的类别包括了基础的数组、链表、排序，高阶的位运算、动态规划也有考察。从题目的难度来说，中等难度占了一半以上，困难的只有十分之一不到。

这就提示我们，刷题要刷，但不是盲目的刷，要有针对性，重点的去刷，所以 Mark 老师除了上面的大厂 Top20 的高频题，还从 LeetCode《LeetCode 热题 HOT 100》中精选了题目，一起对这些题目进行讲解。

但是请注意，Mark 老师在讲解这些题目时，默认大家已经具备了 Java 的基础知识，所以数组、ArrayList、链表 LinkedList、HashMap 等常见常用的数据结构不再解释它们的具体含义、区别和用法，如果需要会在解题过程中直接使用。

但是对于位运算、动态规划、并查集等不是特别常见的数据结构和算法，则会解释具体的概念和含义后，再进入题目的讲解。

但是在正式进入学习之前，我们首先要知道为什么我们要学习数据结构和算法，怎么正确的学习数据结构和算法，怎么正确的刷题。

为什么要学习数据结构和算法?

对数据结构和算法不重视，对日常工作会有很大的影响。

首先，程序员这个群体也是有金字塔结构的。如果连基本的算法和数据结构都不会，基本上就比较底层，底层就意味着低薪酬。付出同样时长的脑力劳动，赚得就会比别人少。

其次，作为团队里的一员，很多时候不光要做好自己的本职工作，也要和其他团队进行技术问题上的沟通，如果没有扎实的算法和数据结构基础，很难及时发现问题并提出独到的见解。

另外，技术栈本身每天都在变化，同时也会随着不同行业不同公司改变。能否快速适应新技术和新环境就显得尤为重要。这就要求你必须具有以不变应万变的的计算机思维、算法思维和逻辑思维能力。

所以数据结构与算法能力的考核在以 BAT 为代表的国内大厂，乃至硅谷大厂等硅谷高科技公司的面试里占了相当大的比重。总结起来，考察的原因有：

算法能力能够准确辨别一个程序员的技术功底是否扎实；

算法能力是发掘程序员的学习能力与成长潜力的关键手段；

算法能力能够协助判断程序员在面对新问题时，分析并解决问题的能力，通过算法问题能够看出候选人的解题思路，以及能将思路迅速地变成代码的能力；

算法能力是设计一个高性能系统、性能优化的必备基础。

怎么样学习数据结构和算法

学习

范围

数据结构和算法？很多人就会联想到“数学”，觉得算法会涉及到很多深奥的数学知识。数据结构和算法课程确实会涉及一些数学方面的推理、证明，尤其是在分析某个算法的时间、空间复杂度的时候，但是这个不需要担心。

如果你是算法工程师或者去学习《算法导论》，里面有非常复杂的数学证明和推理，确实需要很好的数学功底。但是对于我们大多数程序员来说，特别是学习本堂课的同学来说，只要有高中数学水平，就完全可以没有任何问题。

数据结构和算法包含的内容很多，很多高级的数据结构与算法，比如二分图、最大流等，这些在我们平常的开发中很少会用到，也就没有必要投入精力去学习。

不管是应付面试还是工作需要，只要集中精力逐一攻克最常用的、最基础数据结构与算法主要包括：

数据结构：数组、链表、栈、队列、散列表、二叉树、堆、跳表、图、Trie[traɪ]树；

算法：递归、排序、二分查找、搜索、哈希算法、贪心算法、分治算法、回溯算法、动态规划、字符串匹配算法。

掌握了这些基础的数据结构和算法，再学更加复杂的数据结构和算法，就会比较容易了。

虽然校招算法会考的比社招多，但是考来考去，基本不考这种例如 B 树，B+树，KMP 的具体实现的，对于这些，我们只需要知道它的原理和应用即可。但是红黑树之类，在某些大厂面试中是有可能出现。、

学习方式

知识沉淀

知识需要沉淀，不要想试图一下子掌握所有。在学习的过程中，一定会碰到“拦路虎”。如果哪个知识点没有怎么学懂，不要着急，这是正常的。因为，想听一遍、看一遍就把所有知识掌握，这肯定是不可能的。学习知识的过程是反复迭代、不断沉淀的过程。

多练习

“边学边练”这一招非常有用，也就是要多动手，常用的数据结构和算法，用代码实现一遍，也要在专门的刷题网站，比如 LeetCode 上进行刷题。这些都是打好基本功，想要达到顶尖的水平，基本功非常重要。不管什么运动，基本功是区别业余和职业选手的根本，所以一个算法题只写一遍是绝对不够的。

必须要刻意练习，怎么样刻意练习？

具体的题目求解

首先对某个具体的题目求解来说，可以使用下面的解法：

1、读题，分析看清楚题目到底要你做什么？

2、多解，一个题目尽可能的想到几种解法，然后从方方面面去分析这几种解法的优劣，从中选择一个较好的解法。

3、代码实现。

4、运行测试用例，测试用例的选择上，除了一些正常的数据，还要有意识的构造一些边界数据来检测程序是否运行正常。

刷题的目的

刷题不能为刷题而刷题，那是无效的努力，只会感动自己。刷题的目的是通过刷一个题搞清楚一类题目的解法并且建立正确的解题能力和技巧，每做一个题都要有自己的深入的思考和总结。否则就会陷入无效的“题海战术”而收效甚微。

面试

有了前面的刻意练习，面试中的算法题基本上就不是大问题了。为了提高算法面试通过率，还需要把自己的状态调整到最佳，包括和面试官交谈时的状态、是否能清晰地分析问题、如何把自己的思路完美地告诉给面试官，最后是书写代码的水平，也就是能否能流畅地写出可读性高的代码。

如果在面试过程中很紧张，一时半会想不到什么好解法好思路，怎么办？没有关系，我们不妨说出一种最差劲的解法。 要知道完全做不出来和做不出优秀的解法是不一样的。所以大家做题的过程不要忽略“暴力解法”，而且暴力解法通常是好解法好思路的起点。另外“暴力解法”可以给我们缓冲的时间，通过和面试官的沟通，向面试官展示你的思考方式，思维能力，沟通能力，并在沟通的过程中一步步优化原来的解法。

这是因为，第一，无法在规定时间内完美地写出代码并不是世界末日，只要求职者对问题分析正确，把握了正确的思路，代码能清晰地看到输入和输出的逻辑、结构，最重要的是能迅速地将思路转变成部分代码，就能让大部分面试官满意。

第二，在解决问题的过程中缺乏和面试官的交流，甚至对面试官的疑问不屑一顾，那又怎能让面试官放心跟他一起工作呢？所以面试者的沟通能力也是一个考察点。

基本概念和术语

复杂度分析

复杂度分析是在数据结构和算法我们经常听到的一个概念，因为数据结构和算法解决的是如何更省、更快地存储和处理数据的问题，因此，我们就需要一个考量效率和资源消耗的方法，这就是复杂度分析方法。

算法的复杂度

任何程序的执行，都需要空间，比如内存空间和时间，所以只要讲到数据结构与算法，就一定离不开时间、空间复杂度分析。而这两者之中，我们更关注时间复杂度，毕竟时间一去不回，而程序需要的执行空间可以靠增加内存等存储空间来解决。

事后统计法

把代码跑一遍，通过统计、监控，同样也可以得到算法执的时间和占用的内存大小。而且会更准确。这种评估算法执效率的方法叫事后统计法。但是，这种统计方法有非常大的局限性。

1、必须依据算法事先编制好程序，这通常需要花费大量的时间和精力。如果编制出来发现它根本是很糟糕的算法，不是竹篮打水一场空吗?

2. 测试结果非常依赖测试环境

测试环境中硬件的不同会对测试结果有很大的影响。比如，我们拿同样一段代码，分别用不同的 CPU 执行，速度肯定是不一样，甚至还可能出现两台机器完全相反的结果。

3. 测试结果受数据性质或者规模的影响很大。后面我们会讲到排序算法，对同一个排序算法，待排序数据的有序度不一样，排序的执时间就会有很大的差别。极端情况下，如果数据已经是有序的，那排序算法需要做任何操作，执时间就会非常短。

除此之外，如果测试数据规模太小，测试结果可能无法真实地反应算法的性能。比如 10 个数字的排序，不管用什么算法，差异几乎是零。而如果有一百万个随机数字排序，那不同算法的差异就非常大了。比如，JDK 中的排序，对于小规模的数据排序，Java 反而采用的是较慢的插入排序而不是一般意义上更快的快速排序。

所以，我们需要一个不用具体的测试数据来测试，就可以粗地估计算法的执效率的方法。这就是时间、空间复杂度分析。

因为时间复杂度的重要性相对空间复杂度更高，所以我们主要分析时间复杂度，在必要的时候会分析空间复杂度。

大 O 表示法

算法的执行效率，粗略地讲，就是算法代码执行的时间。但是，如何在不运行代码的情况下,估算一段代码的执行时间呢?

我们用 求 1,2,3…n 的累加和 这个例子(例 1)来说明。

```
int cal(int n) {
    int sum = 0;
     for (int i=1; i <= n; ++i) {
         sum = sum + i;
    }
    return sum;
}
```

上述的代码中，尽管每行代码长度、实际执行的时间都不一样，但是在算法估计中，我们总是认为每行代码执行的时间都一样，我们假设这个时间为 t。

这段代码的总执行时间是多少呢?

第 int sum = 0;、return sum;行代码分别需要 1 个 t 的执行时间，for 循环的两行代码都运行了 n 遍，所以需要 2n*t 的执行时间，所以这段代码总的执行时间就是(2n+2)*t。可以看出来，所有代码的执行时间 T(n)与每行代码的执行次数成正比。

按照这个分析思路，我们再来看这段代码(例 2)。

```
int cal2(int n) {
    int sum = 0;
    for (int i=1; i <= n; ++i) {
        for (int j=1; j <= n; ++j) {
            sum = sum + i * j;
        }
    }
}
```

int sum = 0 行，需要 1 个 t 的执行时间，第一个 for 循环执行了 n 遍，需要 n * t 的执行时间，第二个 for 循环两行代码在外层循环的每遍执行中都执行了 n 遍，所以总共执行了 n^2 遍，所以需要 n^2 *t 的执行时间。所以，整段代码总的执行时间 $T(n)=(n^2+n+1)*t$。

尽管我们不知道 t 的具体值，但是通过这两段代码执行时间的推导过程，我们可以得到一个非常重要的规律，那就是，所有代码的执行时间 T(n)与每行代码的执行次数 n 成正比。我们可以把这个规律总结成一个公式。

$$T(n)=O(f(n))$$

其中，T(n)表示代码执行的时间；n 表示数据规模的大小；f(n)表示每行代码执行的次数总和，因为它是 n 的某个函数，所以用 f(n)来表示。整个公式表示随问题规模 n 的增大，算法执行时间的增长率和 f(n)的增长率相同。

所以，第一个例子中的 T(n)= O(2n+2)，第二个例子中的 T(n)= O(n^2+n+1)。这就是**大 O 时间复杂度表示法**。大 O 时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，所以，也叫作渐进时间复杂度(asymptotic time complexity)，简称时间复杂度。

时间复杂度

时间复杂度分析

如何分析一段代码的时间复杂度？常用的方法主要有：

只关注循环执行次数最多的一段代码

因为大 O 这种复杂度表示方法只是表示一种变化趋势。我们通常会忽略掉公式中的常量、低阶、系数，只需要记录一个最大阶的量级就可以了。所以，我们在分析一个算法、一段代码的时间复杂度的时候，也只关注循环执行次数最多的那一段代码就可以了。这段核心代码执行次数的 n 的量级，就是整段要分析代码的时间复杂度。

比如例 1 中其中第 2、3 行代码都是常量级的执行时间，与 n 的大小无关，所以对于复杂度并没有影响。循环执行次数最多的是第 4、5 行代码，这两行代码被执行了 n 次，所以总的时间复杂度就是 O(n)。

总复杂度等于最高阶项的复杂度

看看下面这段代码：

```
int cal3(int n) {

    int sum_1 = 0;

    for (int p = 1; p < 100; ++p) {

        sum_1 = sum_1 + p;

    }

    int sum_2 = 0;
```

```
        for ( int q = 1; q < n; ++q) {
            sum_2 = sum_2 + q;
        }
        int sum_3 = 0;
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                sum_3 = sum_3 + i * j;
            }
        }
        return sum_1 + sum_2 + sum_3;
    }
```


这个代码分为三部分，分别是求 sum_1、sum_2、sum_3。我们可以分别分析每一部分的时间复杂度，然后把它们放到一块儿，再取一个量级最大的作为整段代码的复杂度。

sum_1 的时间复杂度是多少呢?这段代码循环执行了 100 次，所以是一个常量的执行时间，跟 n 的规模无关。

注意即便这段代码循环 10000 次、100000 次，只要是一个已知的数，跟 n 无关，照样也是常量级的执行时间。也就是说不管常量的执行时间多大，我们都可以忽略掉。因为它本身对增长趋势并没有影响。

那 sum_2 和 sum_3 的时间复杂度是多少呢?答案是 O(n)和 $O(n^2)$。

那么上面的代码总的时间复杂度是多少？很明显，就是 $O(n^2+n+100)$。

因为 n^2 增长趋势是远快于 n 的，所以在计算时间复杂度时，我们取其中最大的量级。所以，整段代码的时间复杂度就为 $O(n^2)$。也就是说:总的时间复杂度就等于量级最大的那段代码的时间复杂度。那我们将这个规律抽象成公式就是：

如果 T1(n)=O(fn))，T2(n)=O(g(n));

那么 T(n)=T1(n)+T2(n)=max(O(f(n)), O(g(n)))=O(max(f(n), g(n)).

嵌套代码的复杂度等于嵌套内外代码复杂度的乘积

我们看看下面的代码：

```
int complexCal(int n) {
    int ret = 0;
    for (int i = 1; i < n; ++i) {
        ret = ret + f(i);
    }
    return ret;
}
int f(int n) {
    int sum = 0;
    int i = 1;
    for (; i < n; ++i) {
        sum = sum + i;
    }
    return sum;
}
```


我们单独看 complexCal()方法。假设 f()只是一个普通的操作，那第 4~6 行的时间复杂度就是，T1(n)= O(n)。但 f()函数本身不是一个简单的操作，它的时间复杂度是 T2(n)= O(n)，所以，整个 cal()函数的时间复杂度就是，T(n)= T1(n)* T2(n)= O(n*n)= $O(n^2)$。

不过，你并不用刻意去记忆复杂度分析。实际上，复杂度分析这个东西关键在于“熟练”。你只要多看案例，多分析会发现这个分析其实并不难。

总的来说推导大 O 阶:

1.用常数 1 取代运行时间中的所有加法常数。

2.在修改后的运行次数函数中,只保留最高阶项。

3.如果最高阶项存在且不是 1，则去除与这个项相乘的常数。得到的结果就是大 O 阶。

常见时间复杂度

虽然代码千差万别，但是常见的复杂度量级并不多。大致包括：

O(1)	常数阶
O(n)	线性阶
$O(n^2)$	平方阶
O(logn)	对数阶
O(nlogn)	线性对数阶
$O(n^3)$	立方阶
$O(2^n)$	指数阶
O(n!)	阶乘阶

从小到大依次是：

$O(1)<O(logn)<O(n)< O(nlogn)< O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)$

在上述的列表中，有两点需要注意，第一；在我们一般的算法实践中，当算法的时间复杂度达到 $O(n^3)$时，这个算法在工程应用上基本上就没什么意义了，不大的 n 都会使结果变得不可接受，即使是 $O(n^2)$的时间复杂度，在运用上，我们都要再三考量，小心谨慎。

第二，$O(2^n)$和 O(n!)，我们一般称为非多项式量级。时间复杂度为非多项式量级的算法问题叫作 NP（Non-Deterministic Polynomial，非确定多项式）问题。

当数据规模 n 越来越大时，非多项式级算法的执行时间会急剧增加，求解问题的执行时间会无限增长。所以，非多项式时间复杂度的算法其实是非常低效的算法，在工程实践上一般也是不考虑使用的。

O(1)

首先要明确，O(1)只是常量级时间复杂度的一种表示方法，并不是指只执行了一行代码。比如这段代码，即便有 3 行，它的时间复杂度也是 O(1)，而不是 O(3)。

```
{

int i = 8;

int j = 6;

int sum = i + j;

}
```

只要代码的执行时间不随 n 的增大而增长，这样代码的时间复杂度我们都记作 O(1)。或者说，一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是 O(1)。

O(logn) 和 O(nlogn)

对数阶时间复杂度非常常见，看下面的例子：

```
i=1;
while (i <= n) {
    i = i * 2;
}
```

很明显第三行代码是循环执行次数最多的。所以，我们只要能计算出这行代码被执行了多少次，就能知道整段代码的时间复杂度。

从代码中可以看出，变量 i 的值从 1 开始取，每循环一次就乘以 2。当大于 n 时，循环结束。实际上，变量 i 的取值就是一个等比数列：

$2^0,2^1,2^2,2^3,2^4,……2^x=n$

所以，我们只要知道 x 值是多少，就知道这行代码执行的次数了。通过 $2^x=n$ 求解 x，$x=\log_2 n$，所以，这段代码的时间复杂度就是 $O(\log_2 n)$。

那么下面这个例子：

```
i=1;
while (i <= n) {
    i = i * 3;
}
```


很自然，时间复杂度就是 $O(\log_3 n)$，

实际上，不管是以 2 为底、以 3 为底，还是以 10 为底，我们可以把所有对数阶的时间复杂度都记为 O(logn)。

因为对数之间是可以互相转换的，$\log_3 n$ 就等于 $\log_3 2 * \log_2 n$，很明显 $\log_3 2$ 是一个常量。基于我们前面的一个理论：在采用大 O 标记复杂度的时候，可以忽略系数。所以从时间复杂度来说，$O(\log_3 n)$就等于 $O(\log_2 n)$。因此，在对数阶时间复杂度的表示方法里，我们忽略对数的“底”，统一表示为 O(logn)。

如果一段代码的时间复杂度是 O(logn)，我们循环执行 n 遍，时间复杂度就是 O(nlogn)了。而且，O(nlogn)也是一种非常常见的算法时间复杂度。

O(m+n)、O(m*n)

O(m+n)、O(m*n)表示代码的复杂度由两个数据的规模来决定。比如：

```
int mnCal(int m, int n) {
  int sum_1 = 0;
  int i = 1;
  for (; i < m; ++i) {
    sum_1 = sum_1 + i;
```

```
    }
    int sum_2 = 0;
    int j = 1;
    for (; j < n; ++j) {
      sum_2 = sum_2 + j;
    }
    return sum_1 + sum_2;
}
```

从代码中可以看出，m 和 n 是表示两个数据规模。我们无法事先评估 m 和 n 谁的量级大，所以我们在表示复杂度的时候，就不能简单地利用加法法则，省略掉其中一个。所以，上面代码的时间复杂度就是 O(m+n)，O(m*n)也是同理。

空间复杂度

理解了大O表示法和时间复杂度分析，空间复杂度就非常简单了。

时间复杂度的全称是渐进时间复杂度，表示算法的执行时间与数据规模之间的增长关系。同理，空间复杂度全称就是渐进空间复杂度(asymptotic space complexity)，表示算法的存储空间与数据规模之间的增长关系。比如：

```
void print(int n) {
    int i = 0;
    int[] a = new int[n];
    for (i; i <n; ++i) {
        a[i] = i * i;
    }
    for (i = n-1; i >= 0; --i) {
        print out a[i]
    }
}
```


在上面的例子中，我们申请了一个空间存储变量 i，但是它是常量阶的，跟数据规模 n 没有关系，所以我们可以忽略。第 3 行申请了一个大小为 n 的 int 类型数组，除此之外，剩下的代码都没有占用更多的空间，所以整段代码的空间复杂度就是 O(n)。

一般来说，如果算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为 O(1)。

我们常见的空间复杂度就是 O(1)、O(n)、O(n2)，像 O(logn)、O(nlogn)这样的对数阶复杂度平一般很少见，而且，空间复杂度分析比时间复杂度分析要简单很多。所以，我们重点要掌握的还是算法的时间复杂度的问题。

版权说明

从现在开始我们进入了正式的刷题过程，我们会遵循着按专题和难易度相结合的形式，由易到难，逐一攻破这些题目。在我们的课程中，出现的题目和题目相关属性，比如内含图片、文字等，版权归属于各自的网站或书籍作者或出版社，包括但不限于 LeetCode、《剑指 Offer》等。

力扣中文网地址：https://leetcode-cn.com/

力扣全球地址：https://leetcode.com/

递归

因为在我们后面的解题过程中，需要使用递归，所以有必要和大家一起来学习下递归。而且递归是一种应用非常广泛的算法（或者说编程技巧)。之后我们要讲的很多数据结构和算法的编码实现都要用到递归，比如 DFS 深度优先搜索、前中后序二叉树遍历等等。所以，搞懂递归非常重要，否则，后面复杂一些的数据结构和算法学起来就会比较吃力。

什么是递归呢？举个例子，有个长辈给了我们一个上锁的神秘宝箱，并告诉我们钥匙埋在院子里的大树下而且一定存在，我们兴高采烈地把它挖出来，结果是个木娃娃，摇一摇里面有东西，打开一看，还是一个木娃娃，我们继续努力，一层层的打开，直到打开第 6 个娃娃，最后终于发现了钥匙。

这个过程，如果我们用伪代码来描述如下：

```
void openBox(当前盒子){
    打开盒子;
    if(盒子里是钥匙){
        return  钥匙;
    }
    else{
        openBox(更小的盒子);
    }
}
```

可以看见，在 openBox 方法中，我们又调用了 openBox 方法自己，这种对方法或者函数的使用方式，就叫做递归。

但是由于递归函数调用自己，因此编写这样的函数时很容易出错，进而导致无限递归。比如上面的那个木娃娃如果是个魔法娃娃，一个套一个有无限个，那么即使我们开到天荒地老也是找不到钥匙的。于是我们给自己定个规矩，开到第 20 个娃娃还没看到钥匙，那我们也不继续下去了，所以我们的代码就要变成：

```
void openBox2(当前盒子，盒子的深度){
   if(盒子的深度>=20)
      return “我不干了”;
   if(盒子里是钥匙){
         return  钥匙;
    }
    else{
        盒子的深度加 1;
         openBox2(更小的盒子，盒子的深度);
    }
}
```

所以编写递归函数时，必须告诉它何时停止递归。正因为如此，每个递归函数都有两部分：基线条件（base case）或者说终止条件和递归条件（recursive case）。递归条件指的是什么条件下方法或者函数调用自己，而终止条件则指的是方法或者函数不再调用自己，从而避免形成无限递归。在我们上面的 openBox 例子中，

```
if(盒子里是钥匙){
    return  钥匙;
}
```

基线条件

```
else{
    openBox(更小的盒子);
```

递归条件

所以总的来说，如果一个问题满足：

1.一个问题的解可以分解为几个子问题的解。

何为子问题？子问题就是数据规模更小的问题。

2.这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样

3.存在基线/终止条件。

把问题分解为子问题，把子问题再分解为子子问题，一层一层分解下去，不能存在无限循环，这就需要有基线/终止条件。

满足上面三个条件，就可以考虑使用递归来解决问题。

但是递归有利有弊，利是递归代码的表达力很强，写起来非常简洁：而弊就是因为递归需要方法的反复调用，这就牵涉到方法在执行的过程中需要 JVM 对方法进行大量的压栈和出栈的操作，所以空间复杂度高、有堆栈溢出的风险、过多的方法调用会耗时较多、存在重复计算等问题。

所以，在开发过程中，我们要根据实际情况来选择是否需要用递归的方式来实现。而且从理论上来说，所有的递归代码都可以改为迭代循环的非递归写法。

现在我们已经了解了什么是递归，我们来看看如何用递归解决问题。

(LeetCode-70) 爬楼梯

热度

《LeetCode 热题 HOT 100》专题

题目

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

示例 2：

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶

分析和解答

方法 1

我们仔细想下，实际上，可以根据第一步的走法把所有走法分为两类，第一类是第一步走了 1 个台阶，另一类是第一步走了 2 个台阶。

所以 n 个台阶的走法的个数就等于先走 1 阶后剩下的 n-1 个台阶的走法个数再加上先走 2 阶后剩下的 n-2 个台阶的走法个数。用公式表示就是：

f(n) = f(n-1)+f(n-2)

这其实就是个递归公式，我们再来看下终止条件。当有一个台阶时，我们不需要再继续递归，就只有一种走法。所以 f(1)=1。但是这个递归终止条件不够。

n=2 时，f(2)=f(1)+f(0)。如果递归终止条件只有一个 f(1)=1，那 f(2)就无法求解了。所以除了 f(⑴)=1 这一个递归终止条件外，还要有 f(0)=1，表示走 0 个台阶有一种走法，不过这样子有点滑稽。所以，我们可以把 f(2)=2 单独作为一种终止条件，表示走 2 个台阶，有两种走法，一步走完或者分两步来走。

所以，递归终止条件就是 f(1)=1，f(2)=2。

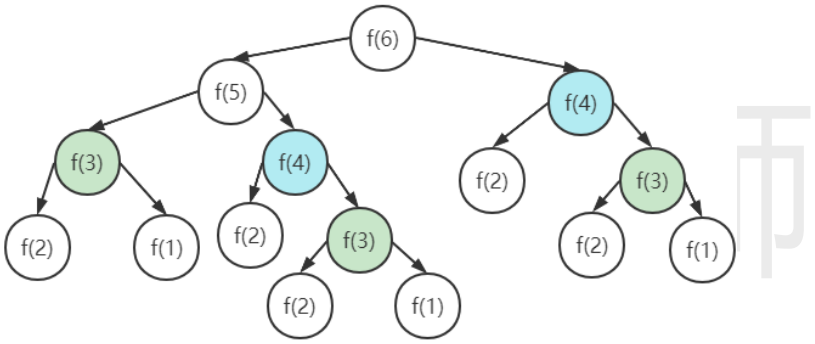
综合在一起就是这样的:

$$f(n)=\begin{cases}1 & n=1\\ 2 & n=2\\ f(n-1)+f(n-2) & n>3\end{cases}$$

这种方法的时间复杂度为 $O(n^2)$。

方法 2

仔细分析我们上面的实现，最大的问题是什么？存在着大量的重复计算，我们以 f(6)来分析一下：

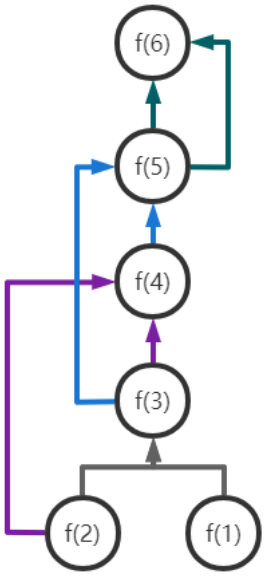


可以看到在 f(6)的求解过程中，f(3)和 f(4)都被求解了多次，这个其实是没必要的，我们可以通过一个 HashMap 来保存已经求解过的 f(k)。当递归调用到 f(k)时，先看下是否已经求解过了。如果是，则直接从散列表中取值返回，不需要重复计算，这样就能避免刚讲的问题了。

这种方法的时间复杂度为 O(n)。

方法 3

在这个题目中，递归的解法是自顶向下，由最开始的顶层数字一层层分解直到最底层的 f(1 和 f(2)，再将结果又一层层累加上来。循环的解法则可以直接由底向上，从最底层的 f(1 和 f(2)，直接累加上来即可，而且从图中可以看到，上一个循环中计算出来的值刚好可以在下一个循环中使用。



不过一般来说，循环取代递归的解法在代码上要复杂一些，也比较难以理解一点。

这种方法的时间复杂度为 O(n)。

代码

ClimbingStairs_70.java

(剑指 Offer 10) 斐波那契数列

热度

京东

题目

写一个函数，输入 n，求斐波那契（Fibonacci）数列的第 n 项。斐波那契数列的定义如下:

$$f(n)\begin{cases}0 & n=0\\1 & n=1\\f(n-2)+f(n-1) & n>1\end{cases}$$

分析和解答

可以看到，这个斐波那契数列的公式几乎和上面的“(LeetCode-70) 爬楼梯”一模一样，所以具体的解答和代码就作为作业留给大家自己思考和实现。

数组

(LeetCode-1)两数之和

热度

字节、美团、阿里巴巴、腾讯、百度、京东

《LeetCode 热题 HOT 100》专题

题目

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 1：

输入：nums = [2,7,11,15], target = 9

输出：[0,1]

解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。

示例 2：

输入：nums = [3,2,4], target = 6

输出：[1,2]

分析和解答

方法 1

暴力穷举，复杂度 $O(n^2)$。

对数组中每个元素，都去计算它和数组中其他元素的和 sum 是否等于目标值 target，如果是则返回结果，不是则继续循环，直到将所有元素检查一遍。

方法 2

这道题最优的做法时间复杂度是 O(n)。用一个哈希表，存储每个数对应的下标。具体做法是：顺序扫描数组，对每一个元素，在 map 中找能组合给定值的另一半数字，如果找到了，直接返回 2 个数字的下标即可。如果找不到，就把这个数字存入 map 中，等待扫到“另一半"数字的时候，再取出来返回结果。

当然，这个 hash 表中存入组合给定值的另一半数字也是可以的。

代码

TwoSum_1.java

(LeetCode-88) 合并两个有序数组

热度

字节、美团、快手

《LeetCode 热题 HOT 100》专题

题目

给你两个按 非递减顺序 排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n ，分别表示 nums1 和 nums2 中的元素数目。

请你 合并 nums2 到 nums1 中，使合并后的数组同样按 非递减顺序 排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0 ，应忽略。nums2 的长度为 n 。

示例 1：

输入：nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

输出：[1,2,2,3,5,6]

解释：需要合并 [1,2,3] 和 [2,5,6] 。

合并结果是 [1,2,2,3,5,6] ，其中斜体加粗标注的为 nums1 中的元素。

示例 2：

输入：nums1 = [1], m = 1, nums2 = [], n = 0

输出：[1]

解释：需要合并 [1] 和 [] 。

合并结果是 [1] 。

示例 3：

输入：nums1 = [0], m = 0, nums2 = [1], n = 1

输出：[1]

解释：需要合并的数组是 [] 和 [1] 。

合并结果是 [1] 。

注意，因为 m = 0 ，所以 nums1 中没有元素。nums1 中仅存的 0 仅仅是为了确保合并结果可以顺利存放到 nums1 中。

进阶：你可以设计实现一个时间复杂度为 O(m + n) 的算法解决此问题吗？

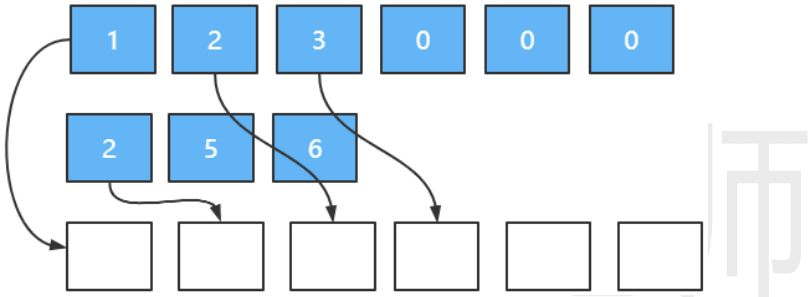
分析和解答

方法 1

最容易想到的办法，把两个数组合并在一起后，再进行排序。这里的排序我们可以直接使用 JDK 中的 sort 方法，JDK 中的排序在元素的数量小于 47 时，使用的插入排序，大于 47 时使用的快速排序，我们简单的以快速排序作为标准，这种方法的时间复杂度来为 O((m+n)log(m+n))，空间复杂度为：O(log(m+n))。当然现在我们还不知道快速排序是个什么东西，我们后面会学到的并且会带大家去手写实现一个快速排序。

方法 2

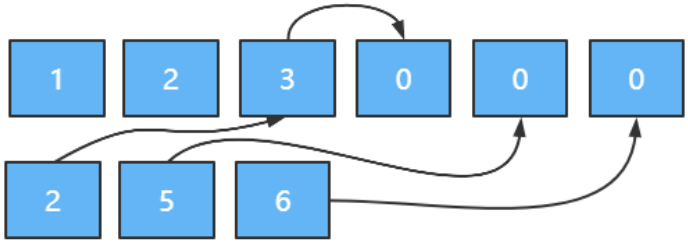
方法 1 的最大问题是什么，仔细观察题目我们其实可以发现，这个两个数组本身其实就是已经有序的了，这一点我们没有利用上，所以改进的办法就是利用“双指针”，每次从两个数组头部取出比较小的数字放到结果中。



这种方法的时间复杂度是多少呢？因为要把两个数组各循环一遍，所以时间复杂度是 O(m+n)，在处理上因为需要一个额外的空白数组来存放两个数组的值，所以空间复杂度也是 O(m+n)。

方法 3

方法 2 中还需要一个长度为 m+n 的临时数组作为中转，能不能不要这个数组呢？因为题目中的整数数组 nums1 的后面还有空位，完全可以利用上。所以改进方法就是，依然使用双指针，但是倒序处理。



这种情况下，时间复杂度是 O(m+n)，空间复杂度则变为 O(1)。

代码

MergeSortedArray_88.java

(LeetCode-283)移动零

热度

《LeetCode 热题 HOT 100》专题

题目

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

说明:

必须在原数组上操作，不能拷贝额外的数组。

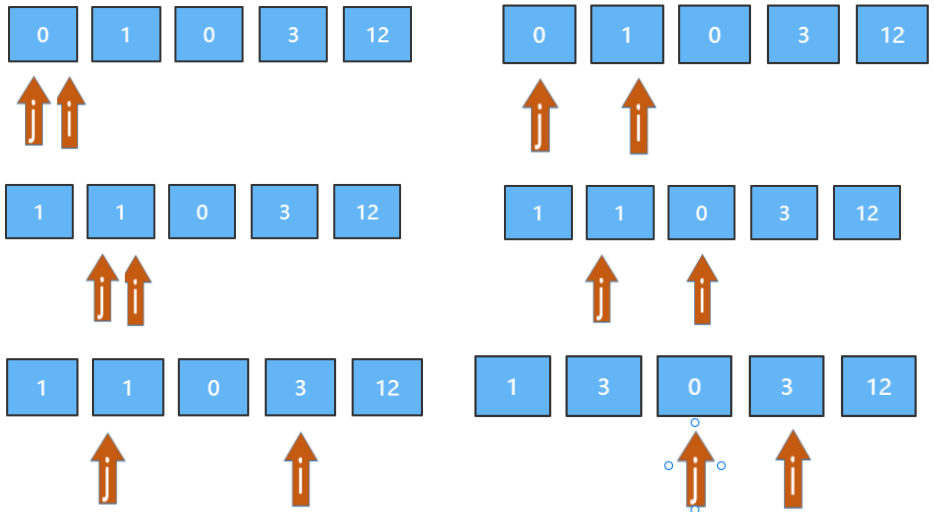
尽量减少操作次数。

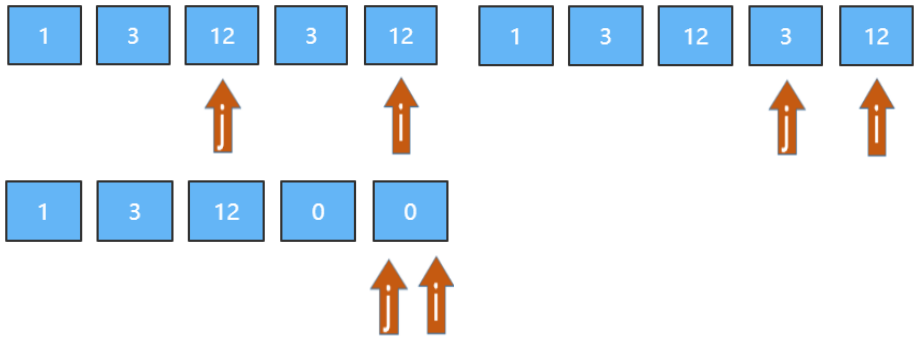
分析和解答

方法

依然可以用双指针的办法，两个指针 i 和 j。i 负责遍历整个数组，在遍历数组的时候，j 用来记录**当前**所有非 0 元素的个数。遍历的时候每遇到一个非 0 元素就将其往数组左边挪，挪动到 j 所在的位置，注意是挪动，不是交换位置，j 同时也移动一个位置。当第一次遍历完后，j 指针的下标就指向了已经排完了位置的最后一个非 0 元素下标。

进行第二次遍历的时候，起始位置就从 j 开始到结束，将剩下的这段区域内的元素全部置为 0 即可。





时间复杂度是 O(n)，空间复杂度则变为 O(1)。

方法 2

还有一种一次遍历的方式，参考了快速排序的思想，不过快速排序要在后面才讲到，所有我们这里就不细说了，等到我们学习完快速排序后，同学们可以回来自行思考下这个题目如何用快速排序的思想来解决。

代码

MoveZeroes_283.java

(LeetCode-448)找到所有数组中消失的数字

热度

《LeetCode 热题 HOT 100》专题

题目

给你一个含 n 个整数的数组 nums ，其中 nums[i] 在区间 [1, n] 内。请你找出所有在 [1, n] 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

示例 1：

输入：nums = [4,3,2,7,8,2,3,1]

输出：[5,6]

示例 2：

输入：nums = [1,1]

输出：[2]

提示：

n == nums.length

1 <= n <= 105

1 <= nums[i] <= n

进阶：你能在不使用额外空间且时间复杂度为 O(n) 的情况下解决这个问题吗？你可以假定返回的数组不算在额外空间内。

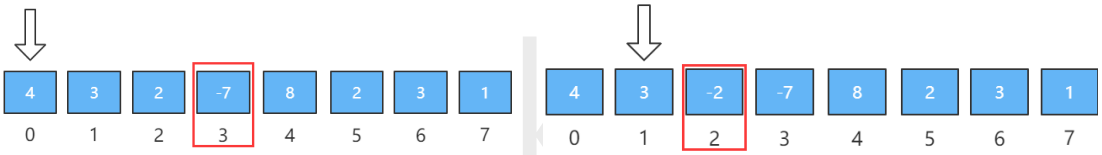
分析和解答

方法 1

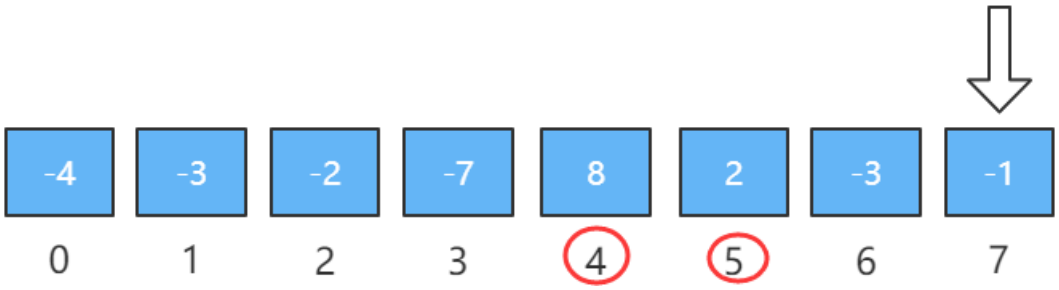
如果没有进阶的限制，我们可以用一个哈希表记录数组 nums 中的数字，由于数字范围均在 [1,n] 中，记录数字后我们再利用哈希表检查 [1,n] 中的每一个数是否出现，从而找到缺失的数字。但是很明显，这种情况下是不满足进阶的限制条件不使用额外空间，所以这种方法我们不考虑。

但是现在题目要求我们找到“数组中消失的数字”，那就意味着必须要有个地方记录哪些数字在数组中出现过，哪些没出现过，那么怎么处理呢？那就只能利用数组 nums 本身了。具体的做法是，数组的长度本身为 n，而数组 nums 的数字范围均在 [1,n] 中，所以我们可以利用这一范围之外的数字，来记录数字是否出现过，比如加一个数或者把数字变为负数。我们以负数为例进行说明。

比如第 0 个元素的数字是 4，则把第[4-1]个元素的数字改为-7，第 1 个元素的数字是 3，则把第[3-1]个元素的数字改为-2，依次类推。



最终数组会变为



可以看到，只有下标为 4 和 5 的元素的数字还是正数，说明数字 4+1=5 和数字 5+1=6 没有在数组中出现过。

加一个数的思路是一样的，但是要做注意，加以后的和不能在数组的元素中存在。这个加数我们可以设置为 n，数组 nums 的数字范围均在 [1,n]，所以加上 n 后，最小的数也是 n+1，可以避免我们上面所说的问题。

但是要注意，当我们遍历到某个位置时，其中的数可能已经变化过，因此还需要进行判断和还原。

这种方法的时间复杂度：O(n)，空间复杂度：O(1)。

代码

NumbersDisappeared_448.java

剑指 Offer 53 题目二：0～n-1 中缺失的数字

热度

京东

题目

0～n-1 中缺失的数字。

一个长度为 n-1 的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围 0～n-1 之内。在范围 0～n-1 内的 n 个数字中有且只有一个数字不在该数组中，请找出这个数字。

分析和解答

方法 1

这个题目咋看起来和“(LeetCode-448)找到所有数组中消失的数字”非常像，但其实有很大的差别。本题中数组是递增排序的，所有的数字都是唯一的，而且缺失的数字只有一个。这种不同导致解法上有很大的不同。

仔细观察我们发现

因为 0～n-1 这些数字在数组中是排序的，因此数组中开始的一些数字与它们的下标相同。也就是说，0 在下标为 0 的位置，1 在下标为 1 的位置,以此类推。如果不在数组中的那个数字记为 m，那么所有比 m 小的数字的下标都与它们的值相同。

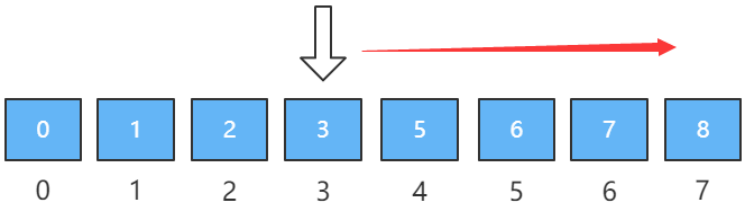
由于 m 不在数组中，那么 m+1 处在下标为 m 的位置，m+2 处在下标为 m+1 的位置，以此类推。

所以最直观的解法是扫描整个数组，当发现数组的下标和数组元素的值不一致时，下标的值就是缺失的数字，当然这个数字可能出现在数组的头部，中部，也可能出现在尾部。毫无疑问，这种方法的时间复杂度就是 O(n)。有没有更快的解法呢？

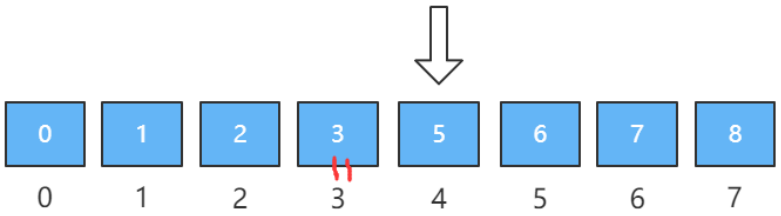
方法 2

在我们上面的解法里，没有充分的利用数组是递增排序的这一特点。我们没有必要从数组的头部开始扫描起，完全可以从数组的中间开始扫描，这就是二分查找的思想。根据扫描的结果分为三种情况：

1、如果中间元素的值和下标相等，那么下一轮查找只需要查找右半边；



2、如果中间元素的值和下标不相等，并且它前面一个元素和它的下标相等，这意味着这个中间的数字正好是第一个值和下标不相等的元素，它的下标就是在数组中不存在的数字；



3、如果中间元素的值和下标不相等，并且它前面一个元素和它的下标不相等，这意味着下一轮查找我们只需要在左半边查找即可。

代码

MissNumber_offer53II.java

链表

(LeetCode-21)合并两个有序链表

热度

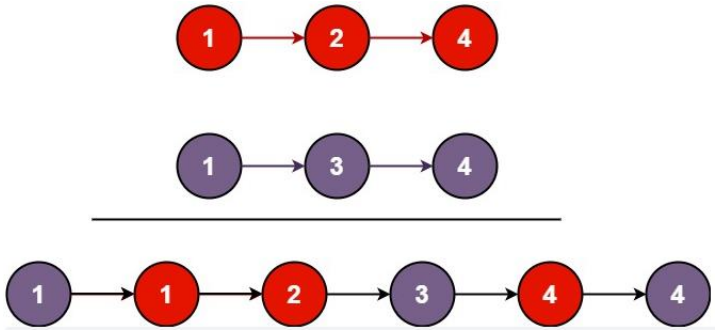
字节、美团、阿里巴巴、腾讯、百度、京东

《LeetCode 热题 HOT 100》专题

题目

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：



输入：l1 = [1,2,4], l2 = [1,3,4]

输出：[1,1,2,3,4,4]

示例 2：

输入：l1 = [], l2 = []

输出：[]

示例 3：

输入：l1 = [], l2 = [0]

输出：[0]

提示：

两个链表的节点数目范围是 [0, 50]

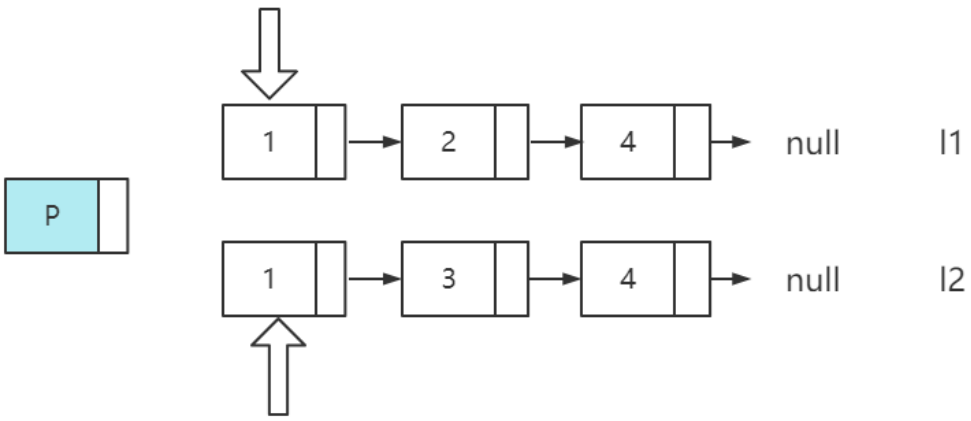
-100 <= Node.val <= 100

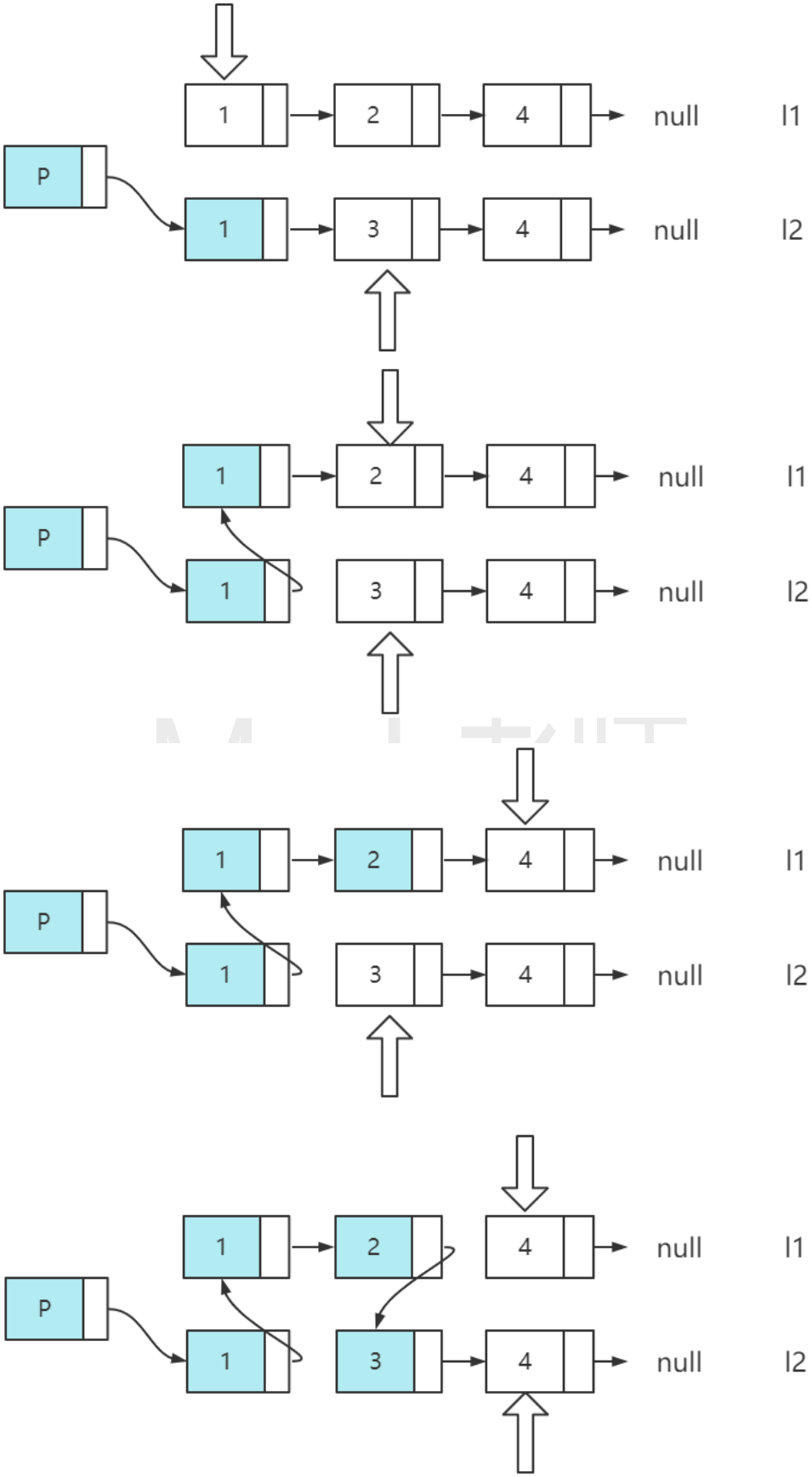
l1 和 l2 均按 非递减顺序 排列

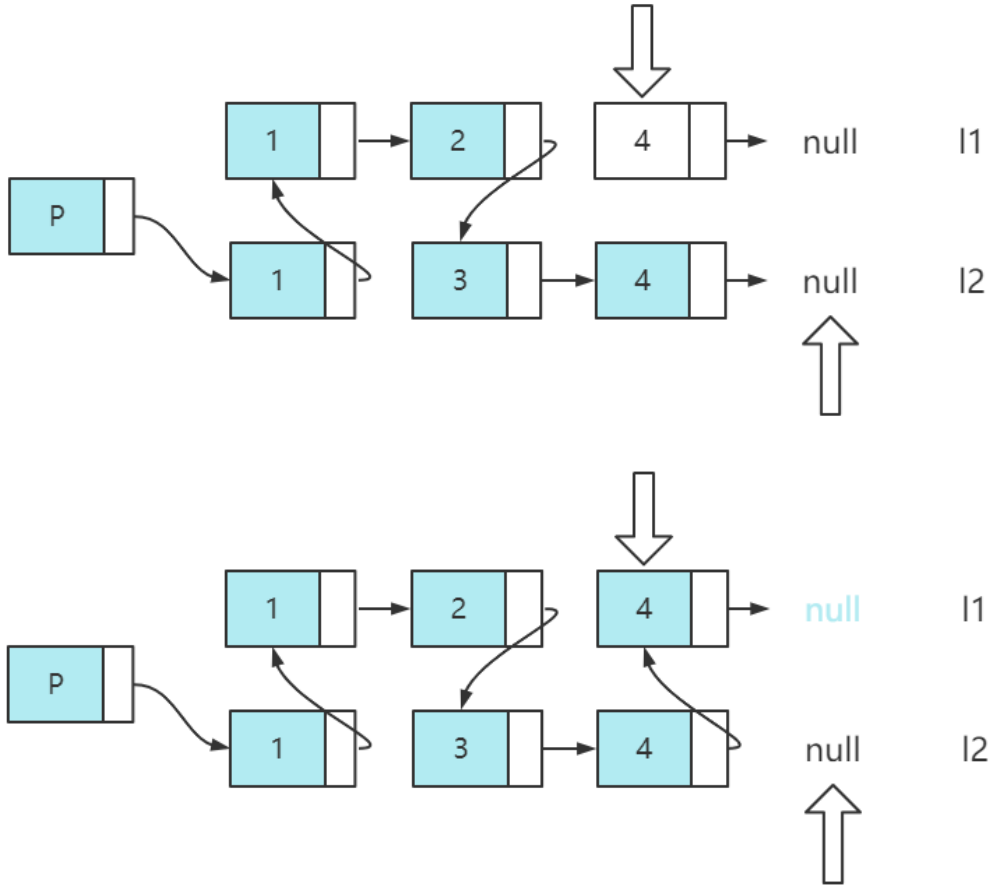
分析和解答

方法 1

其实这个问题和前面的“(LeetCode-88) 合并两个有序数组”有非常大的相似之处，同样是合并，同样是有序，所以这个题目中我们同样可以使用类似“双指针”来解决这个问题。每次从两个链表头部取出比较小的数字放到结果中。为此我们需要引入一个结果节点 resultNode ，这可以在最后让我们比较容易地返回合并后的链表。




1
2
4
null
l1
P
1
3
4
null
l2
1
2
4
null
l1
P
1
3
4
null
l2
1
2
4
null
l1
P
1
3
4
null
l2
1
2
4
null
l1
P
1
3
4
null
l2



这种方法的时间复杂度是多少呢？因为要把两个链表各循环一遍，所以时间复杂度是 O(m+n)，空间复杂度则是 O(1)。

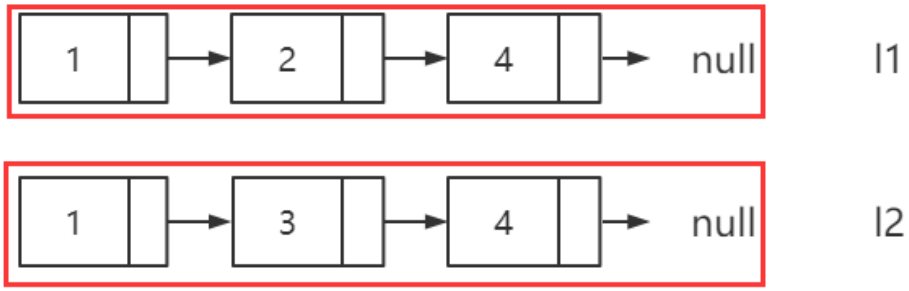
当然在实际编码要注意：

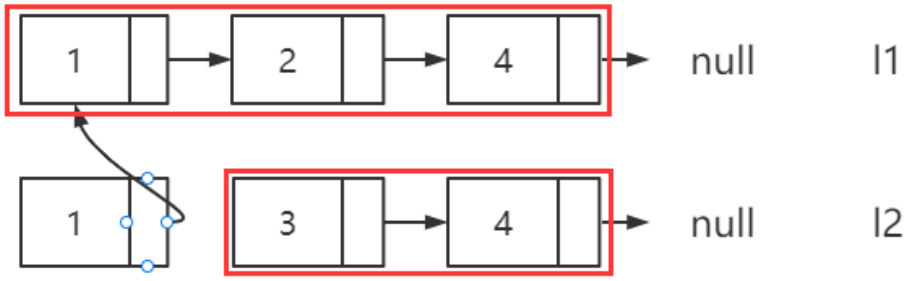
1、边界情况。如果 l1 或者 l2 一开始就是空链表 ，那么没有任何操作需要合并，所以我们只需要返回非空链表即可。

2、因为链表自带指针，所以我们没有必要再单独声明两个指针了。

方法 2

通过研究双指针的解法，我们可以发现，每当我们处理了链表中的一个节点，





这个问题就演变为，一个变小的链表和另外一个链表的合并问题，和我们本来的问题“合并两个有序链表”是同一性质的问题，只是规模更小点，这就完全可以利用递归来解决。

这种方法的时间复杂度是 O(m+n)，空间复杂度也是 O(m+n)。

代码

MergeTwoSortLists_21.java

(LeetCode-83) 删除排序链表中的重复元素

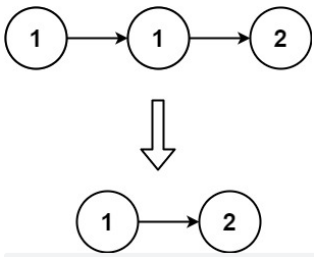
热度

京东

题目

存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除所有重复的元素，使每个元素 只出现一次 。返回同样按升序排列的结果链表。

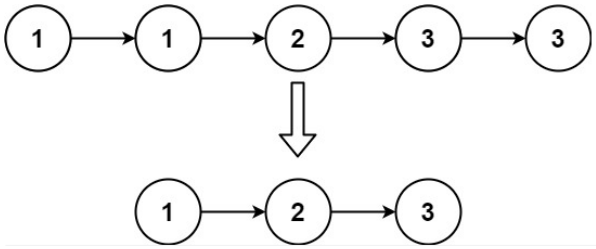
示例 1：



输入：head = [1,1,2]

输出：[1,2]

示例 2：



输入：head = [1,1,2,3,3]

输出：[1,2,3]

提示：

链表中节点数目在范围 [0, 300] 内

-100 <= Node.val <= 100

题目数据保证链表已经按升序排列

分析和解答

方法 1

分析题目我们可以看到，链表是排好序的，因此重复的元素在链表中出现的位置一定是连续的，所以我们可以这么做。

遍历链表，只要发现当前节点的值和下一个节点的值是一样的，那么下一个节点的值就是重复的，自然就可以从链表中去除。

自然这种方法的时间复杂度：O(n)，空间复杂度为 O(1)。

方法 2

还可以使用递归来实现，代码比方法 1 要简洁，当然时间复杂度上是一定会上升的。道理很简单，每处理完一个节点，就意味着需要处理的链表变小，而这个变小的链表在处理上和原链表的处理方式是一模一样的，这就完全可以利用递归来做。

代码

RemoveDuplicates_83.java

(LeetCode-141) 环形链表

热度

微软、美团、阿里巴巴、快手、腾讯、百度

《LeetCode 热题 HOT 100》专题

题目

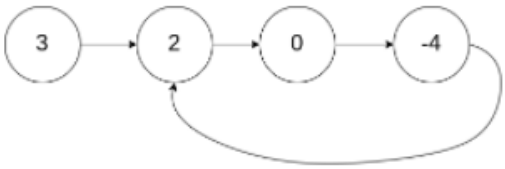
给定个链表，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true 。 否则，返回 false 。

进阶：你能用 O(1)内存解决此问题吗？

示例 1：

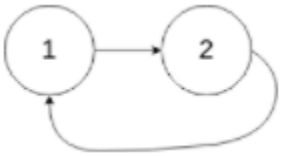


输入：head = [3,2,0,-4], pos = 1

输出：返回索引为 1 的链表节点

解释：链表中有一个环，其尾部连接到第二个节点。

示例 2：



输入：head = [1,2], pos = 0

输出：返回索引为 0 的链表节点

解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1

输出：返回 null

解释：链表中没有环。

提示：

链表中节点的数目范围在范围 [0, 10^4] 内

-10^5 <= Node.val <= 10^5

pos 的值为 -1 或者链表中的一个有效索引

分析和解答

方法 1

可以使用 Hash 表来解决：

1、从表头结点开始,逐个遍历链表中的每个结点。

2、对于每个结点，检查该结点的地址是否存在于散列表中。

3、如果存在，则表明当前访问的结点已经被访问过了。出现这种情况只能是因为给定链表中存在环。

4、如果散列表中没有当前结点的地址，那么把该地址插入散列表中。重复上述过程,直至到达表尾或者找到环。

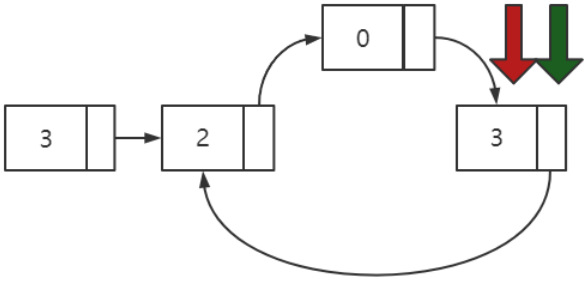
这个方法时间复杂度为 O(n)，用于扫描链表。空间复杂度为 O(n)，用于散列表的空间开销。

但是题目进阶要求是能用 O(1)内存解决此问题，所以这个方法我们不做具体实现，留给同学们自行实现。

方法 2

对于判断是否存在环形链表，其实存在着一种通用解法，该方法称为 Floyd 环判定算法。该方法使用两个在链表中具有不同移动速度的指针。一旦它们进入环，就成为一个环形追及问题，两者肯定相遇，只要相遇就表示存在环。

在工程实践中，一般两个指针每次分别移动 1 个结点和 2 个结点，其他的移动速度也能解决这个问题，但是会增加算法的复杂度。



代码

LinkedListCycle_141.java

(LeetCode-142) 环形链表 II

热度

美团、快手、百度、京东

《LeetCode 热题 HOT 100》专题

题目

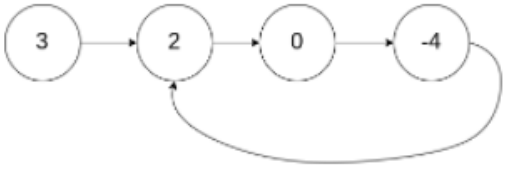
给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 null。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。注意，pos 仅仅是用于标识环的情况，并不会作为参数传递到函数中。

说明：不允许修改给定的链表。

进阶：你是否可以使用 O(1) 空间解决此题？

示例 1：

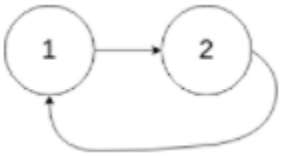


输入：head = [3,2,0,-4], pos = 1

输出：返回索引为 1 的链表节点

解释：链表中有一个环，其尾部连接到第二个节点。

示例 2：



输入：head = [1,2], pos = 0

输出：返回索引为 0 的链表节点

解释：链表中有一个环，其尾部连接到第一个节点。

提示：

链表中节点的数目范围在范围 [0, 10^4] 内

-10^5 <= Node.val <= 10^5

pos 的值为 -1 或者链表中的一个有效索引

分析和解答

方法 1

这个问题和《(LeetCode-141) 环形链表》有紧密的关系，在对 141 这个问题求解的基础上，再进行扩展。

在找到链表中的环后，初始化慢指针的值，使其指向链表的表头结点。然后，慢指针和快指针从各自的位置开始沿着链表移动，每次均移动一个结点。它们相遇的位置就是环的开始结点。

代码

LinkedListCycleII_142.java

(LeetCode-160) 相交链表

热度

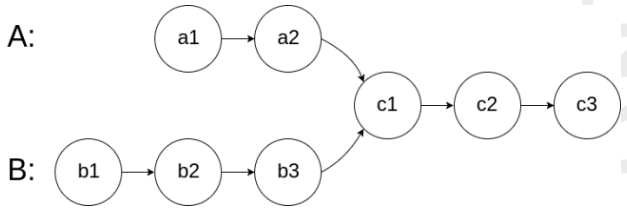
字节、快手、腾讯、百度

《LeetCode 热题 HOT 100》专题

题目

给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 null 。

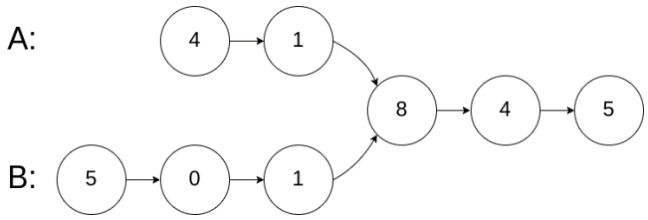
图示两个链表在节点 c1 开始相交：



题目数据保证整个链式结构中不存在环。

注意，函数返回结果后，链表必须保持其原始结构 。

示例 1：



输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

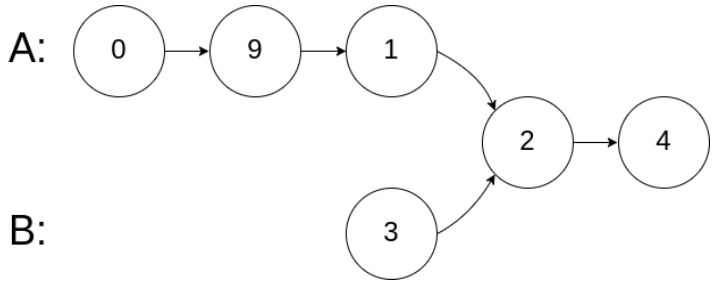
输出：Intersected at '8'

解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。

从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。

在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

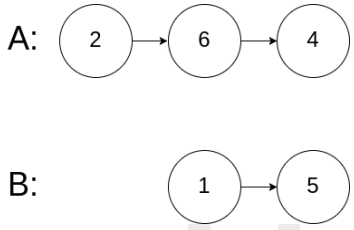
输出：Intersected at '2'

解释：相交节点的值为 2 （注意，如果两个链表相交则不能为 0）。

从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。

在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

输出：null

解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。

由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。

这两个链表不相交，因此返回 null 。

提示：

listA 中节点数目为 m

listB 中节点数目为 n

0 <= m, n <= 3 * 104

1 <= Node.val <= 105

0 <= skipA <= m

0 <= skipB <= n

如果 listA 和 listB 没有交点，intersectVal 为 0

如果 listA 和 listB 有交点，intersectVal == listA[skipA + 1] == listB[skipB + 1]

进阶：你能否设计一个时间复杂度 O(n) 、仅用 O(1) 内存的解决方案？

分析和解答

方法 1

可以用暴力法求解，把第一个链表中的每一个结点指针与第二个链表中的每一个结点指针比较，当出现相等的结点指针时，即为相交结点。但是，这种方法的时间复杂度较高，时间复杂度为 O(mn)，空间复杂度为 O(1)。具体代码请自行实现。

方法 2

还可以使用 Hash 表求解，具体方法是：

1、选择结点较少的链表(如果链表的长度是未知的，那么随便选择一个链表)，将其所有结点的指针值保存在 Hash 表中。

2、遍历另一个链表，对于该链表中的每一个结点，检查 Hash 表中是否已经保存了其结点指针。

3、如果两个链表存在合并点，那么必定会在 Hash 表中找到记录(结点指针)。

时间复杂度为:第一个链表创建散列表的时间开销加上扫描第二个链表的时间开销，等于 O(m)+O(n)(或 O(n)+O(m))，取决于选择哪个链表来建立散列表)。这两种情况具有的算法时间复杂度是相同的。空间复杂度为 O(n)或 O(m)。

但是这种方法不符合题目中进阶的要求。

方法 3

还可使用双指针的方法。

创建两个指针 pA 和 pB，初始时分别指向两个链表的头节点 ，然后将两个指针依次遍历两个链表的每个节点。具体做法如下：

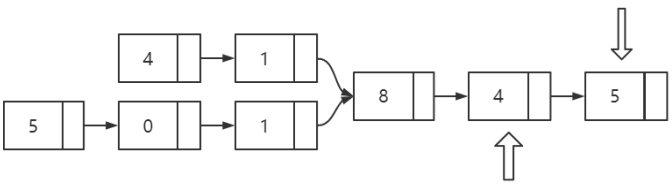
每步操作需要同时更新指针 pA 和 pB。

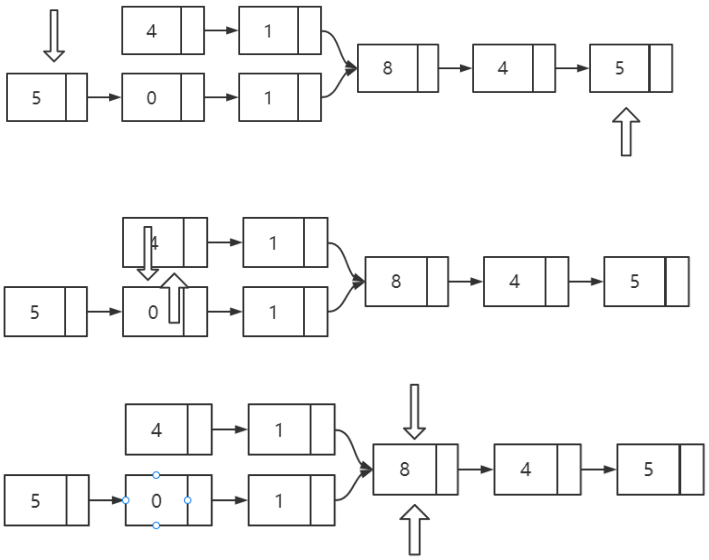
如果指针 pA 不为空，则将指针 pA 移到下一个节点；如果指针 pB 不为空，则将指针 pB 移到下一个节点。

如果指针 pA 为空，则将指针 pA 移到链表 headB 的头节点；如果指针 pB 为空，则将指针 pB 移到链表 headA 的头节点。

当指针 pA 和 pB 指向同一个节点或者都为空时，返回它们指向的节点或者 null。

这种方法的时间复杂度为 O(m+n)，空间复杂度为 O(1)。





方法 4

另外一种解决办法是：

1、获得两个链表(L1 和 L2)的长度

2、计算两个长度的差 d。

3、从较长链表的表头开始，移动 d 步。

4、在两个链表中开始同时移动，直至出现两个后继指针值相等的情况。

这种方法的时间复杂度为 O(m+n)，空间复杂度为 O(1)。

代码

IntersectionTwoLinkedLists_160.java

(LeetCode-206) 反转链表

热度

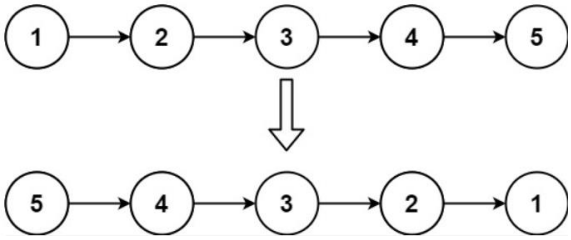
字节、微软、美团、阿里巴巴、快手、腾讯、百度、京东

《LeetCode 热题 HOT 100》专题

题目

给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

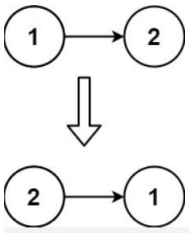
示例 1：



输入：head = [1,2,3,4,5]

输出：[5,4,3,2,1]

示例 2：



输入：head = [1,2]

输出：[2,1]

示例 3：

输入：head = []

输出：[]

提示：

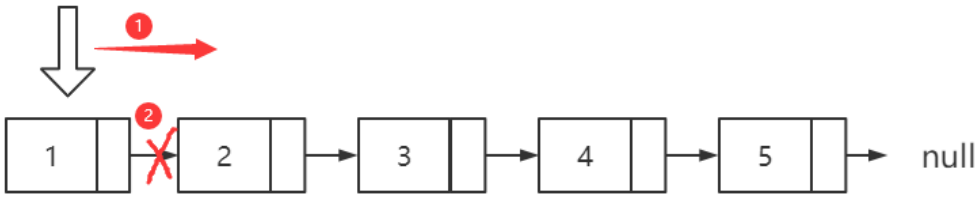
链表中节点的数目范围是 [0, 5000]

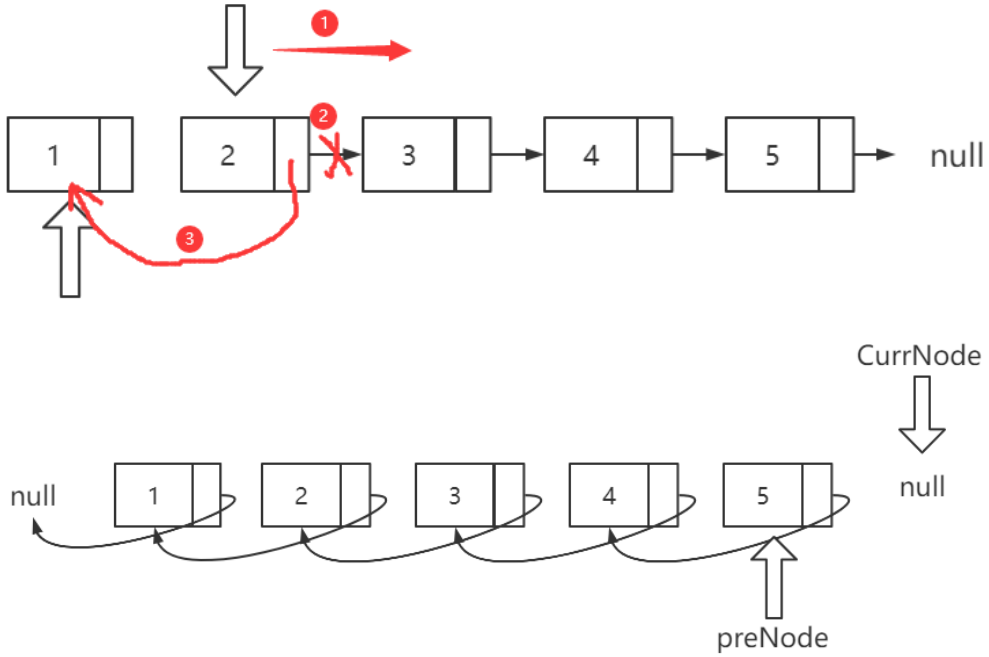
-5000 <= Node.val <= 5000

分析和解答

方法 1

这个题目本身没有什么特别难的地方，主要是要注意，在我们遍历链表的时候，在对每一个节点处理的过程中，因为要将当前结点的 next 指针改为指向前一个节点。但这是单链表，节点是没有引用其前一个节点的，因此我们必须事先存储其前一个节点。所以，这个本质上来说，还是一种双指针的运用。





代码

ReverseLinkedList_206.java

(LeetCode-234) 回文链表

热度

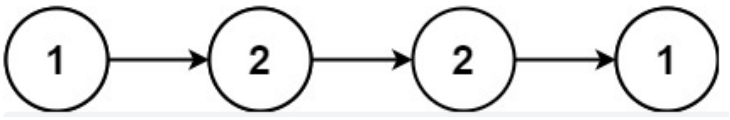
腾讯、

《LeetCode 热题 HOT 100》专题

题目

给你一个单链表的头节点 head ，请你判断该链表是否为回文链表。如果是，返回 true ；否则，返回 false 。

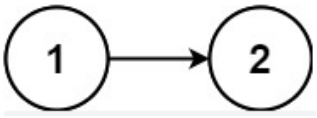
示例 1：



输入：head = [1,2,2,1]

输出：true

示例 2：



输入：head = [1,2]

输出：false

提示：

链表中节点数目在范围[1, 10^5] 内

0 <= Node.val <= 9

进阶：你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题？

分析和解答

方法 1

可以采用双指针法，将链表的值复制到数组列表中，再使用双指针法判断。

第一步，遍历链表将值复制到数组列表中。

第二步的使用双指针法来检查是否为回文。在起点放置一个指针，在结尾放置一个指针，每一次迭代判断两个指针指向的元素是否相同，若不同，返回 false；相同则将两个指针向内移动，并继续判断，直到两个指针相遇。

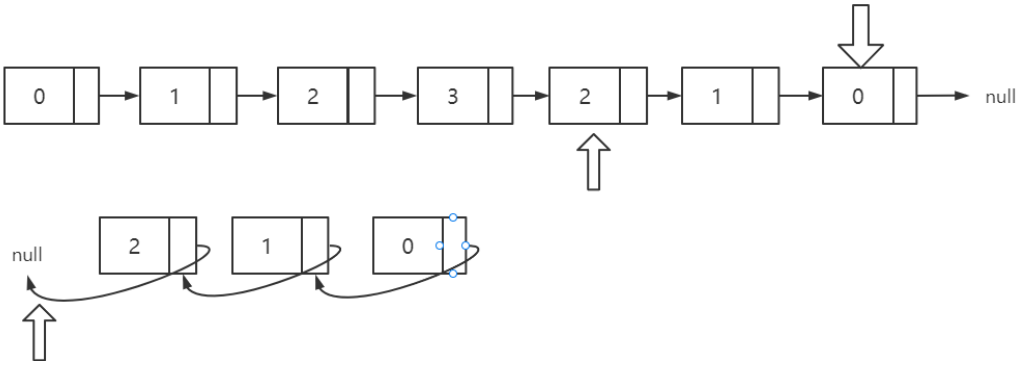
这种方法时间复杂度 O(n)，空间复杂度因为需要一个数组，所以为 O(n)。不太符合题目的进阶要求。

方法 2

还可以用双指针结合反转链表的方法来解决这个问题：将链表的后半部分反转（修改链表结构），然后将前半部分和后半部分进行比较。比较完成后我们应该将链表恢复原样。

具体做法是：

快慢指针同时出发。当快指针移动到链表的末尾时，慢指针恰好到链表的中间。通过慢指针将链表分为两部分。然后反转后半部分链表，判断是否回文。至于是否要恢复链表，则看自己喜欢。



这种方法的时间复杂度 O(n)，空间复杂度 O(1)。

代码

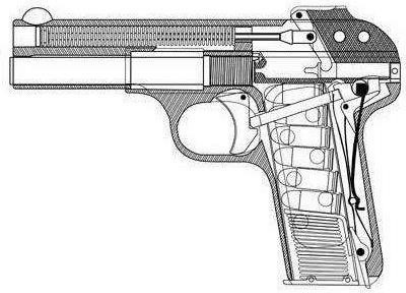
PalindromeLinkedList_234.java

栈与队列

概念解释

栈(堆栈)

枪都有弹夹，弹夹里都有弹簧，在开枪的时候，先压入弹夹的子弹反而是最后才能射出来的。

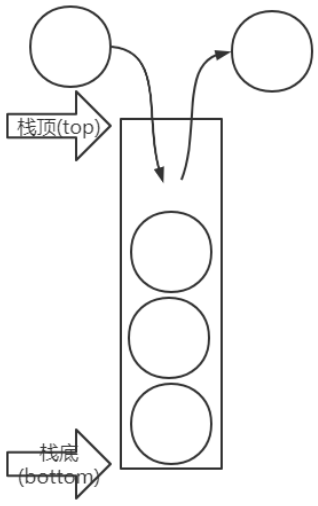


栈就是像弹夹一样的数据结构，先进去的元素，却要后出来，而后进的元素，反而可以先出来。栈有时也被称为堆栈，注意要和堆进行区分。什么是堆，等我们后面讲到树这种数据结构的时候，会讲到堆。

在我们软件应用中，栈这种后进先出数据结构的应用是非常普遍的。比如你用浏览器上网时，不管什么浏览器都有一个“后退”键，你点击后可以按访问顺序的逆序加载浏览过的网页。

很多办公软件，比如 Word、Photoshop 等文档或图像编辑软件中，都有撤销(undb）的操作，也是用栈这种方式来实现的。

我们把允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)，不含任何数据元素的栈称为空栈。栈又称为后进先出（Last In First Out)的结构，简称 LIFO 结构。

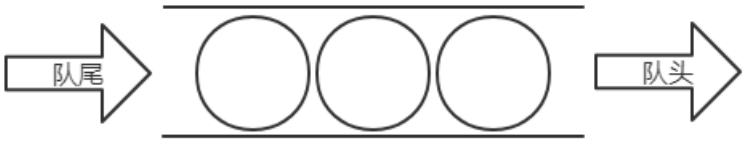


队列

我们去买东西，收银窗口很少，而付款的人很多，于是我们就在收银窗口前排起队来，先排在收银窗口的人自然就先付款先离开，后排队的人后付款后离开。

这就是队列的概念，队列(queue)是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

队列是一种先进先出 (First In First Out)的线性表，简称 FIFO。允许插入的一端称为队尾，允许删除的一端称为队头。



(LeetCode-232) 用栈实现队列

热度

美团

题目

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

void push(int x) 将元素 x 推到队列的末尾

int pop() 从队列的开头移除并返回元素

int peek() 返回队列开头的元素

boolean empty() 如果队列为空，返回 true ；否则，返回 false

说明：

你只能使用标准的栈操作 —— 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。

你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

进阶：

你能否实现每个操作均摊时间复杂度为 O(1) 的队列？换句话说，执行 n 个操作的总时间复杂度为 O(n) ，即使其中一个操作可能花费较长时间。

示例：

输入：

["MyQueue", "push", "push", "peek", "pop", "empty"]

[[], [1], [2], [], [], []]

输出：

[null, null, null, 1, 1, false]

解释：

MyQueue myQueue = new MyQueue();

myQueue.push(1); // queue is: [1]

myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

myQueue.peek(); // return 1

myQueue.pop(); // return 1, queue is [2]

myQueue.empty(); // return false

提示：

1 <= x <= 9

最多调用 100 次 push、pop、peek 和 empty

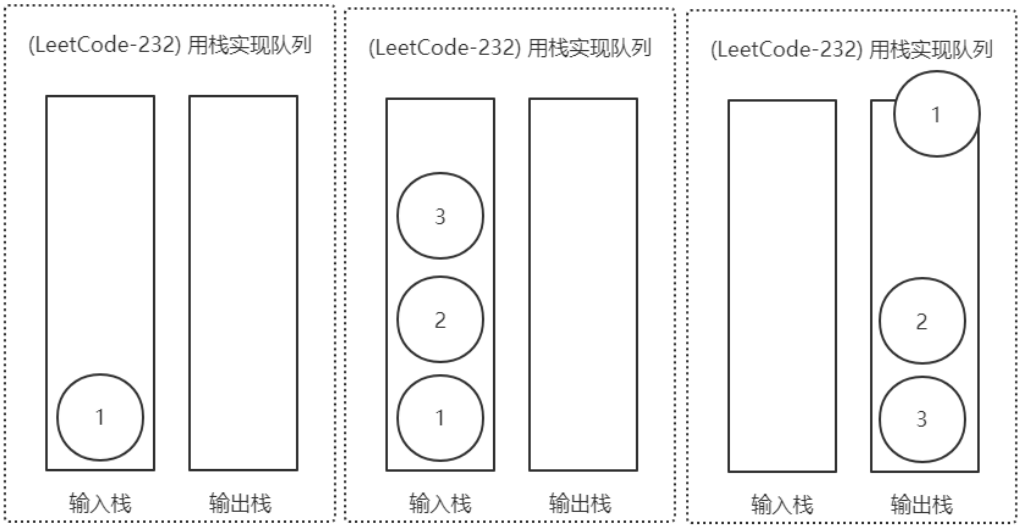
假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）。

分析和解答

方法 1

我们知道栈是后进先出的，为了保证最终形成的队列符合先进先出的的定义，我们需要使用两个栈来实现。将一个栈当作输入栈，用于压入 push 传入的数据；另一个栈当作输出栈，用于 pop 和 peek 操作。

每次 pop 或 peek 时，若输出栈为空则将输入栈的全部数据依次弹出并压入输出栈，这样输出栈从栈顶往栈底的顺序就是队列从队首往队尾的顺序。



代码

ImplQueueUsingStacks_232.java

(LeetCode-394) 字符串解码

热度

《LeetCode 热题 HOT 100》专题

题目

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

示例 1：

输入：s = "3[a]2[bc]"

输出："aaabcbc"

示例 2：

输入：s = "3[a2[c]]"

输出："accaccacc"

示例 3：

输入：s = "2[abc]3[cd]ef"

输出："abcabccdcdcdef"

示例 4：

输入：s = "abc3[cd]xyz"

输出："abccdcdcdxyz"

分析和解答

方法 1

这是我们遇到的第一个中等难度的题，不过不用担心，第一，在我们本阶段的题解中，中等难度的题大约只有这么一个；第二，中等难度的题也没有想象中的那么难，仔细分析题目和示例依然可以做出来的；第三，中等难度的题在一线大厂的面试题中几乎占据了 50%以上的分量，不管有多担心，中等难度的题依然是我们需要下力气去研究和熟练的题目。

我们观察示例，可以发现，其实输入字符串可以分为最小的单位，我们这里称为元组，元组的组成的共性是什么？

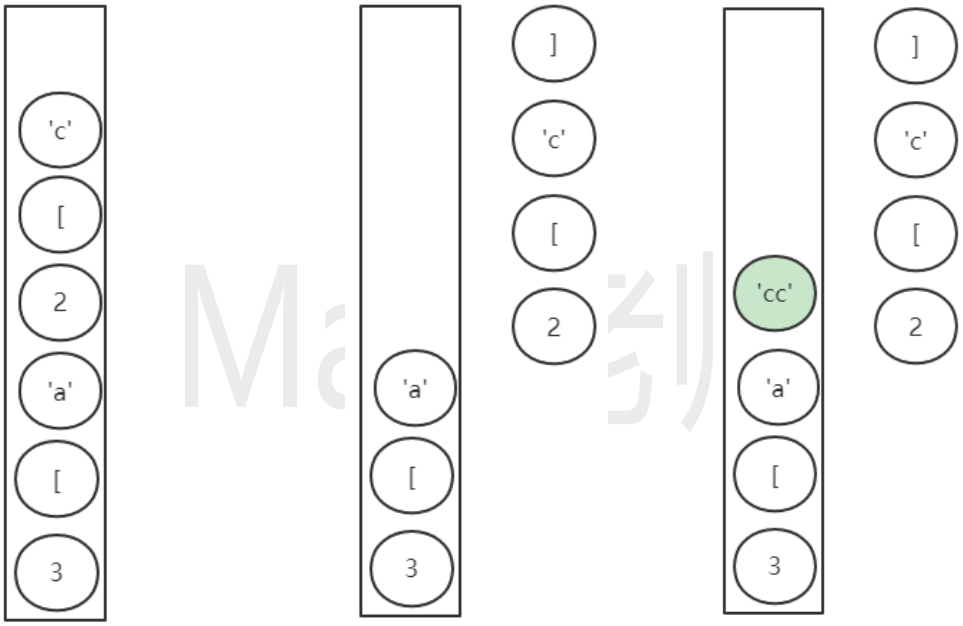
数字[字符串]，也就是：一个数字+左中括号“[”+字符串]+右中括号“]”

其中数字和“[”、“]”可以没有，这表示字符串无需重复，直接输出即可。而元组本身又可以成为其他元组的字符串，构成一个复合元组。

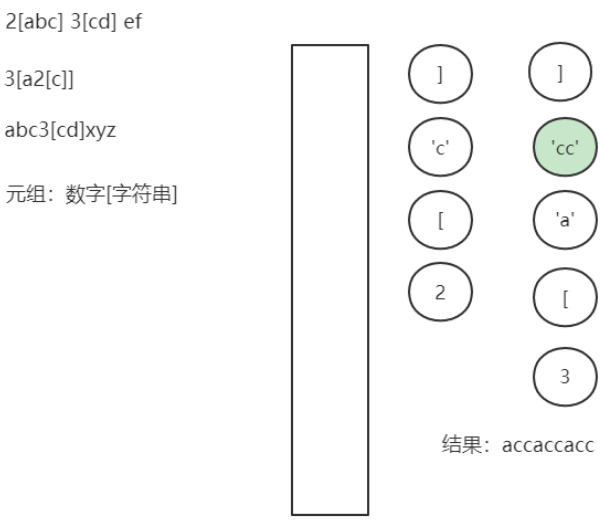
仔细分析，可以看到，每当遇到右中括号“]”，表示前面的字符串需要进行重复，重复的字符串到哪里截止呢？需要往前走，到最近的一个左中括号“[”为止。重复几遍呢？再往前走，离“[”最近的一个数字。

通过分析过程，我们是否可以得到什么提示？需要按顺序往前回溯曾经处理的数据，是不是有点先进后出的感觉。所以可以利用栈来处理这个题目。我们以实例 2 中的“3[a2[c]]”来演示这个过程。

首先按字符读取字符串，只要不是右中括号“]”，将数据逐一压栈；当遇到右中括号“]”，开始出栈，一直出栈到遇到的第一个左中括号“[”的前一个为止；根据出栈的元素拼凑出一个临时字符串，并将临时字符串压栈；



继续读取原字符串，重复上述的过程，遇到右中括号“]”之前压栈，遇到右中括号“]”开始出栈并拼凑重复的字符串，直到原字符串读取完毕以及栈为空为止。



但是在具体的代码实现上，需要注意很重要的一点，题目示例中的数字是个位数，但是实际上，数字完全有可能是个多位数，所以数字需要单独处理一下。同时在栈的选择上，可以使用 JDK 中的 Stack 或者 LinkedList 都可以。

代码

DecodeString_394_UseStack.java

DecodeString_394_UseList.java

树

概念解释

什么是树

我们前面所接触到的链表或者数组，都可以看成元素被一条线串起来了，元素之间是相邻的一对一的关系。然后对这条线上的元素的访问，就是顺着这条线移动，所以链表或者数组也被称为线性表。

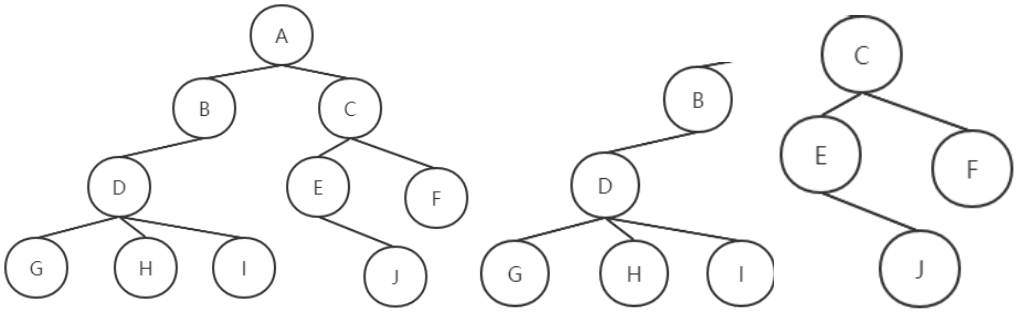
可现实中，还有很多一对多的情况需要处理，所以我们需要研究这种一对多的数据结构——“树”。

树的定义是：

树(Tree)是 n(n≥0）个结点的有限集。n=0 时称为空树。在任意一棵非空树中：

(1）有且仅有一个特定的称为根(Root)的结点;

(2)当 n>1 时,其余结点可分为 m(m>0）个互不相交的有限集 T1、T2、……、Tm,其中每一个集合本身又是一棵树,并且称为根的子树(SubTree)。

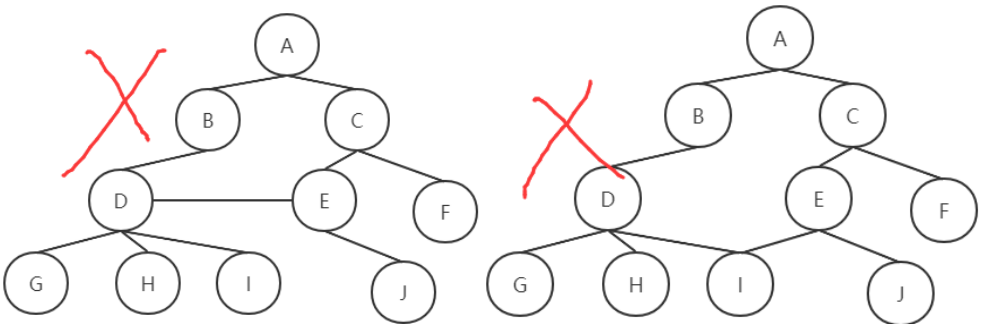


在上图中，A 结点就是根结点，子树 T1 和子树 T2 就是根结点 A 的子树。当然,D、G、H、I 组成的树又是 B 为根结点的子树，E、J 组成的树是 C 为根结点的子树。

但是要注意：

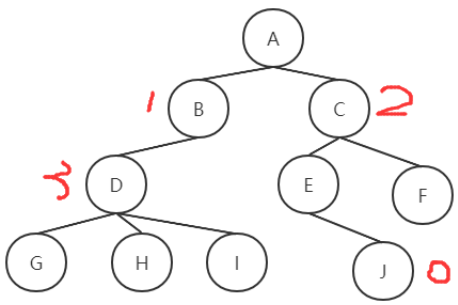
1、根结点是唯一的，不可能存在多个根结点，别和现实中的大树混在一起，现实中的树有很多根须,那是真实的树，数据结构中的树是只能有一个根结点。

2. 子树的个数没有限制但它们一定是互不相交的。下面的两个结构就不符合树的定义，因为它们都有相交的子树。



树的度

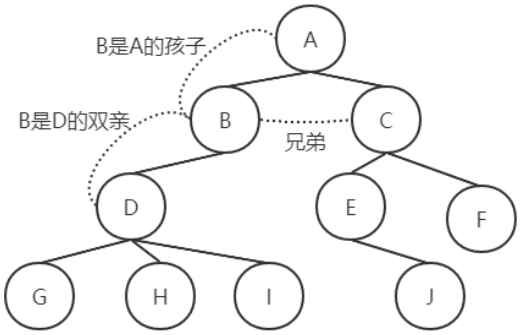
树的结点包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为结点的度(Degree)。度为 0 的结点称为叶结点(Leaf)或终端结点；度不为 0 的结点称为非终端结点或分支结点。除根结点之外，分支结点也称为内部结点。树的度是树内各结点的度的最大值。我们例子中的数结点的度的最大值是结点 D 的度，为 3，所以树的度也为 3。



结点间关系

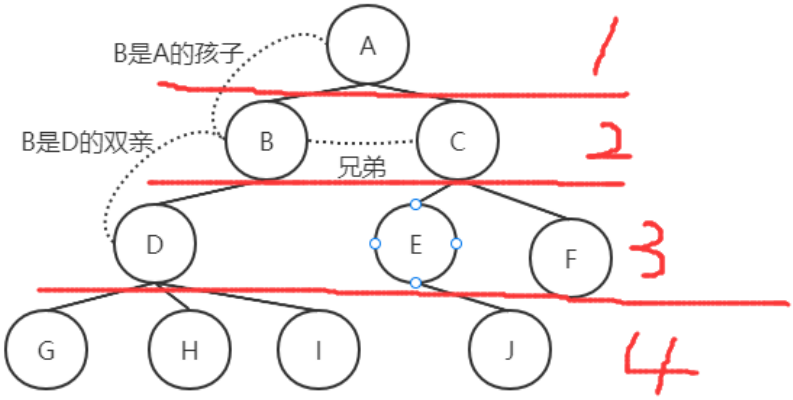
A 结点的子树的根称为 A 结点的孩子（Child)，相应地，A 结点称为孩子的双亲(Parent)。同一个双亲的孩子之间互称兄弟(Sibling)。

结点的祖先是从根到该结点所经分支上的所有结点。所以对于 H 来说，D、B、A 都是它的祖先。反之，以某结点为根的子树中的任一结点都称为该结点的子孙。B 的子孙有 D、G、H、I。



树的深度

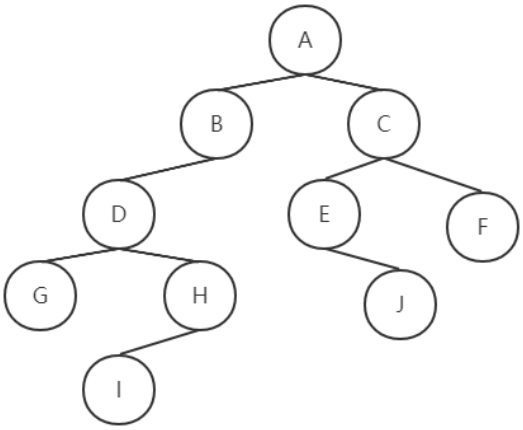
结点的层次（Level)从根开始定义起，根为第一层，根的孩子为第二层。若某结点在第 N 层，则其子树的根就在第 N+1 层。其双亲在同一层的结点互为堂兄弟。D、E、F 是堂兄弟，而 G、H、I、J 也是。树中结点的最大层次称为树的深度(Depth)或高度，当前树的深度为 4。



二叉树

二叉树属于一种特殊的树，定义如下：

二叉树(Binary Tree)是 n(n≥0）个结点的有限集合,该集合或者为空集(称为空二叉树)。或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

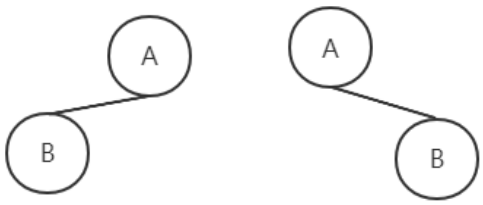


二叉树的特点有:

每个结点最多有两棵子树，所以二叉树中不存在度大于 2 的结点。注意不是只有两棵子树，而是最多有。没有子树或者有一棵子树都是可以的。

.左子树和右子树是有顺序的，次序不能任意颠倒。就像人是双手、双脚，但显然左手、左脚和右手、右脚是不一样的，右手戴左手套、右脚穿左鞋都会极其别扭和难受。

即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。如下图，都是二叉树，但它们却是不同的二叉树。



二叉树具有五种基本形态:

1. 空二叉树。
2. 只有一个根结点。
3. 根结点只有左子树。
4. 根结点只有右子树。
5. 根结点既有左子树又有右子树。

(LeetCode-94) 二叉树的中序遍历

热度

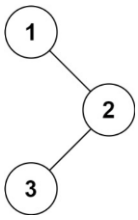
阿里巴巴、百度、京东

《LeetCode 热题 HOT 100》专题

题目

给定一个二叉树的根节点 root ，返回它的 中序 遍历。

示例 1：



输入：root = [1,null,2,3]

输出：[1,3,2]

示例 2：

输入：root = []

输出：[]

示例 3：

输入：root = [1]

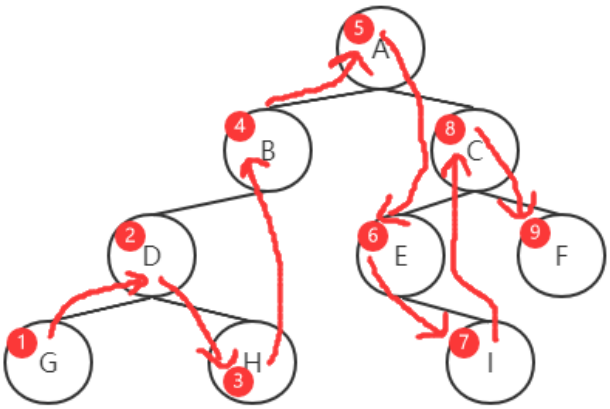
输出：[1]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

分析和解答

方法 1

规则是若树为空，则空操作返回，否则从根结点开始（注意并不是先访问根结点），中序遍历根结点的左子树,然后是访问根结点，最后中序遍历右子树。如图所示，遍历的顺序为: GDHBAEICF。



简单来说，对于一个二叉树而言，有左子树，则访问左子树的值，访问完左子树或者左子树为空，访问根结点的值，访问完根结点，再访问右子树的值。

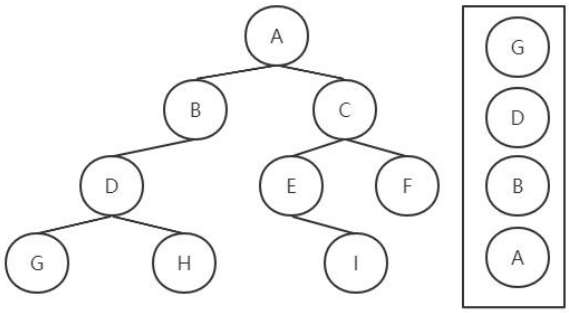
访问左子树或者右子树的时候我们按照同样的方式遍历，直到遍历完整棵树。因此遍历定义我们可以看到，这个是天然具有递归的性质的，所以我们可以直接用递归方法来实现。这种方法的时间复杂度：O(n)，空间复杂度：O(n)。

方法 2

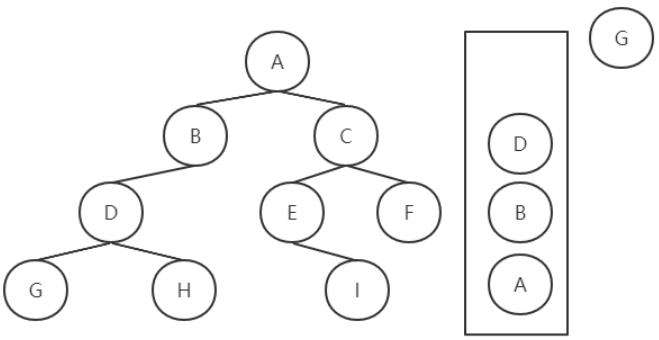
从方法 1 的代码可以看出，用递归来实现二叉树的遍历，代码非常简洁，但是在面试中，面试官不会这么轻易放过面试获选人，往往会要求用循环迭代的方法来实现树的遍历，怎么做呢？

这种实现，往往要借助栈这种数据结构，总的来说，就是每经过一个根结点，就需要把这个根结点放到栈中，访问完这个根结点的左子树后，再访问栈中的根结点。

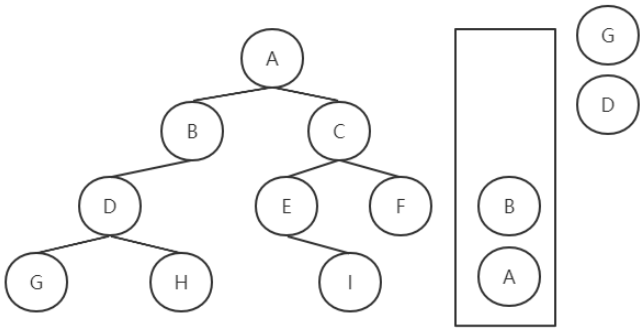
我们代码结合图示来理解整个过程：



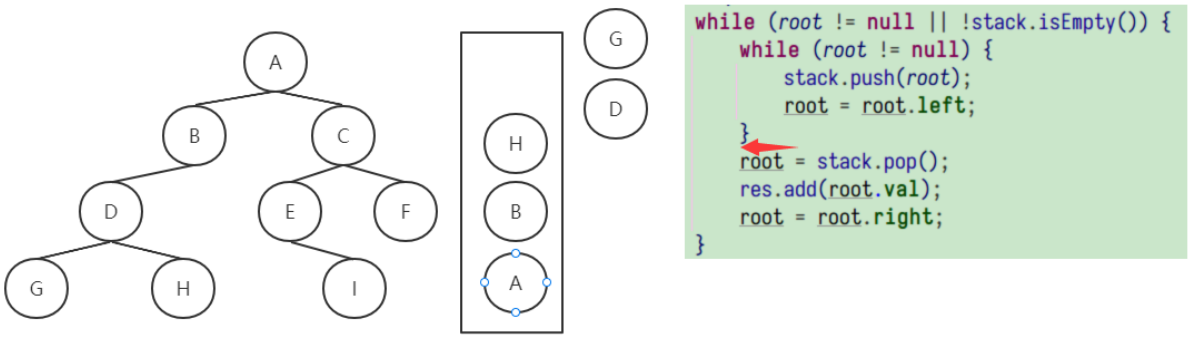

```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}
```



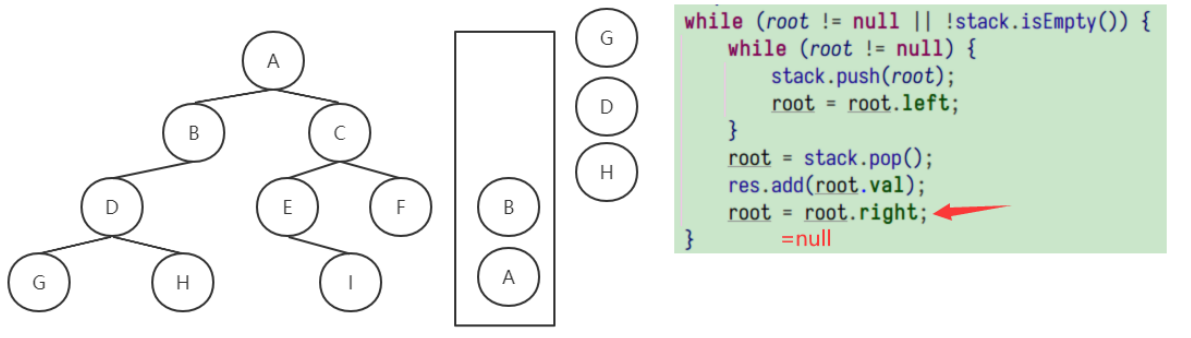

```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}        = null
```



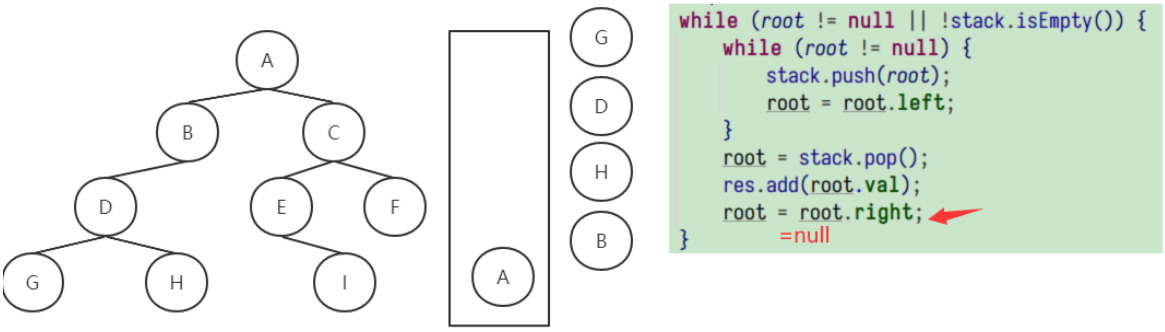

```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}        = H
```



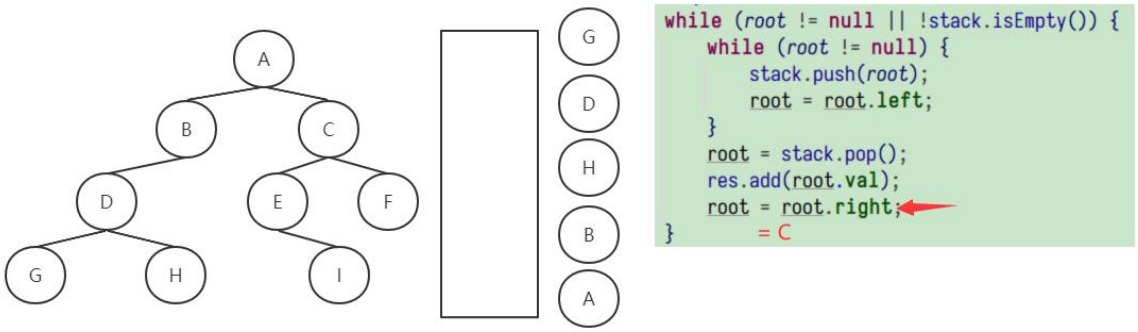

```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}
```




```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}       =null
```




```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}       =null
```




```
while (root != null || !stack.isEmpty()) {
    while (root != null) {
        stack.push(root);
        root = root.left;
    }
    root = stack.pop();
    res.add(root.val);
    root = root.right;
}       = C
```

A 的右子树的处理过程与上面类似，请大家自行思考。

代码

BinTreeInOrderTraversal_94.java

(LeetCode-144) 二叉树的前序遍历

热度

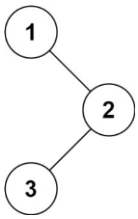
阿里巴巴、快手

《LeetCode 热题 HOT 100》专题

题目

给你二叉树的根节点 root ，返回它节点值的 前序 遍历。

示例 1：



输入：root = [1,null,2,3]

输出：[1,2,3]

示例 2：

输入：root = []

输出：[]

示例 3：

输入：root = [1]

输出：[1]

分析和解答

方法 1

前序遍历简单来说，对于一个二叉树而言，先访问根结点的值，有左子树，则访问左子树的值，访问完左子树，再访问右子树的值。

访问左子树或者右子树的时候我们按照同样的方式遍历，直到遍历完整棵树。同样可以使用递归来实现。在中序遍历的基础上，调整下语句的顺序即可。这种方法的时间复杂度：O(n)，空间复杂度：O(n)。

方法 2

前序遍历的迭代循环同样要借助栈，在具体的代码实现上，和中序遍历也高度相似，做非常小的调整即可。

代码

BinTreePreOrderTraversal_144.java

排序算法

排序就是将一组对象按照某种逻辑顺序重新排列的过程。比如，订单按照日期排序的——这种排序很可能使用了某种排序算法。在计算时代早期，大家普遍认为 30% 的计算周期都用在了排序上。如果今天这个比例降低了，可能的原因之一是如今的排序算法更加高效，而并非排序的重要性降低了。现在计算机的广泛使用使得数据无处不在，而整理数据的第一步通常就是进行排序。几乎所有的计算机系统都实现了各种排序算法以供系统和用户使用。

即使你只是使用标准库中的排序函数，学习排序算法仍然有三大实际意义：

- IT 从业人员必备技能，也是互联网公司面试的必考点；
- 类似的技术也能有效解决其他类型的问题；
- 排序算法常常是我们解决其他问题的第一步。

排序在商业数据处理和现代科学计算中有着重要的地位，它能够应用于事物处理、组合优化、天体物理学、分子动力学、语言学、基因组学、天气预报和很多其他领域。其中一种排序算法（快速排序）甚至被誉为 20 世纪科学和工程领域的十大算法之一。

数据结构和算法中，关于排序有十大算法，包括冒泡排序，简单选择排序，简单插入排序，归并排序，堆排序，快速排序、希尔排序、计数排序，基数排序，桶排序。

一般在面试中最常考的是快速排序和堆排序、归并排序，并且经常有面试官要求现场写出这 3 种排序的代码。对这 3 种排序的代码一定要信手拈来才行。对于其他排序可能会要求比较各自的优劣、各种算法的思想及其使用场景，还有要知道算法的时间和空间复杂度。

通常查找和排序算法的考察是面试的开始，如果这些问题回答不好，估计面试官都没有继续面试下去的兴趣都没了。所以想开个好头就要把常见的排序算法思想及其特点要熟练掌握，有必要时要熟练写出代码。我们将由易到难学习这十种算法。

排序算法总结

(LeetCode- 912) 排序数组

对于排序，LeetCode 上有个 912 号排序数组的题目，可以自行实现了代码后测试下性能。

	平均时间复杂度	最好情况	最坏情况	空间复杂度	稳定性	比较算法
冒泡排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定	是
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	是
插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定	是
希尔排序	O(nlogn)~$O(n^2)$	$O(n^{1.3})$	$O(n^2)$	O(1)	不稳定	是
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定	是
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	不稳定	是
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定	是
计数排序	O(n+k)	O(n+k)	O(n+k)	O(k)	稳定	否
桶排序	O(n+k)	O(n+k)	$O(n^2)$	O(n+k)	稳定	否
基数排序	O(n*k)	O(n*k)	O(n*k)	O(n+k)	稳定	否

计数排序中的 K 为整数的范围；基数排序时间复杂度为 O（N*M），其中 N 为数据个数，M 为数据位数；桶排序的时间复杂度为 O(n+c)，但是 c 比较复杂，c=n*(logn-logk)，其中 n 表示待排数据的个数，k 表示桶的个数。

查找算法

(LeetCode-704) 二分查找

热度

美团、腾讯、京东

题目

给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target ，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。

示例 1:

输入: nums = [-1,0,3,5,9,12], target = 9

输出: 4

解释: 9 出现在 nums 中并且下标为 4

示例 2:

输入: nums = [-1,0,3,5,9,12], target = 2

输出: -1

解释: 2 不存在 nums 中因此返回 -1

分析和解答

查找算法概论

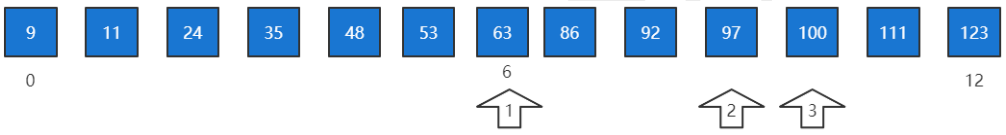
查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算。常见的查找算法有七种：

1. 顺序查找
2. 二分查找
3. 插值查找
4. 斐波那契查找
5. 树表查找
6. 分块查找
7. 哈希查找

二分查找

也成为折半查找，属于有序查找算法。用给定值 k 先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据 k 与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。折半查找的前提条件是需要有序表顺序存储。

比如下面的数组寻找 100 的过程



代码

BinSearch_704.java