目录

[1.Two Sum 两数之和 6](#_Toc519243016)

[2. Add Two Numbers两数相加 7](#_Toc519243017)

[7. Reverse Integer反转整数 11](#_Toc519243018)

[9. Palindrome Number回文数 13](#_Toc519243019)

[13. Roman to Integer罗马数字转整数 15](#_Toc519243020)

[14. Longest Common Prefix 最常公共前缀 18](#_Toc519243021)

[20. Valid Parentheses有效的括号 22](#_Toc519243022)

[26. Remove Duplicates from Sorted Array删除排序数组中的重复项 23](#_Toc519243023)

[27. Remove Element移除元素 24](#_Toc519243024)

[28. Implement strStr() 实现strStr() 24](#_Toc519243025)

[35. Search Insert Position搜索插入位置 25](#_Toc519243026)

[38. Count and Say报数 25](#_Toc519243027)

[53. Maximum Subarray最大子序和 27](#_Toc519243028)

[58. Length of Last Word最后一个单词的长度 28](#_Toc519243029)

[66. Plus One加一 29](#_Toc519243030)

[67. Add Binary二进制求和 30](#_Toc519243031)

[69. Sqrt(x) x 的平方根 31](#_Toc519243032)

[70. Simplify Path爬楼梯 32](#_Toc519243033)

[83. Remove Duplicates from Sorted List删除排序链表中的重复元素（区别26） 33](#_Toc519243034)

[88. Merge Sorted Array合并两个有序数组 34](#_Toc519243035)

[94. Binary Tree Inorder Traversal二叉树的中序遍历 35](#_Toc519243036)

[96. Unique Binary Search Trees不同的二叉搜索树 39](#_Toc519243037)

[95. Unique Binary Search Trees II不同的二叉搜索树 II 40](#_Toc519243038)

[98. Validate Binary Search Tree验证二叉搜索树 42](#_Toc519243039)

[99. Recover Binary Search Tree恢复二叉搜索树 43](#_Toc519243040)

[100. Same Tree相同的树 45](#_Toc519243041)

[101. Symmetric Tree对称二叉树 46](#_Toc519243042)

[102. Binary Tree Level Order Traversal二叉树的层次遍历 47](#_Toc519243043)

[103. Binary Tree Zigzag Level Order Traversal二叉树的锯齿形层次遍历 49](#_Toc519243044)

[104. Maximum Depth of Binary Tree二叉树的最大深度 49](#_Toc519243045)

[105. Construct Binary Tree from Preorder and Inorder Traversal 50](#_Toc519243046)

[从前序与中序遍历序列构造二叉树 50](#_Toc519243047)

[106. Construct Binary Tree from Inorder and Postorder Traversal 51](#_Toc519243048)

[从中序与后序遍历序列构造二叉树 51](#_Toc519243049)

[107. Binary Tree Level Order Traversal II二叉树的层次遍历 II 51](#_Toc519243050)

[108. Convert Sorted Array to Binary Search Tree将有序数组转换为二叉搜索树 52](#_Toc519243051)

[110. Balanced Binary Tree平衡二叉树 53](#_Toc519243052)

[111. Minimum Depth of Binary Tree二叉树的最小深度 54](#_Toc519243053)

[112. Path Sum路径总和 55](#_Toc519243054)

[118. Pascal's Triangle杨辉三角 56](#_Toc519243055)

[119. Pascal's Triangle II杨辉三角 II 57](#_Toc519243056)

[121. Best Time to Buy and Sell Stock买卖股票的最佳时机 58](#_Toc519243057)

[122. Best Time to Buy and Sell Stock II买卖股票的最佳时机II 58](#_Toc519243058)

[125. Valid Palindrome验证回文串 58](#_Toc519243059)

[136. Single Number只出现一次的数字 59](#_Toc519243060)

[141. Linked List Cycle环形链表 60](#_Toc519243061)

[155. Min Stack最小栈 61](#_Toc519243062)

[160. Intersection of Two Linked Lists相交链表 61](#_Toc519243063)

[167. Two Sum II - Input array is sorted两数之和 II - 输入有序数组 63](#_Toc519243064)

[168. Excel Sheet Column Title Excel表列名称 64](#_Toc519243065)

[171. Excel Sheet Column Number Excel表列序号 65](#_Toc519243066)

[169. Majority Element求众数 66](#_Toc519243067)

[172. Factorial Trailing Zeroes 阶乘后的零 68](#_Toc519243068)

[189. Rotate Array旋转数组 69](#_Toc519243069)

[190. Reverse Bits颠倒二进制位 71](#_Toc519243070)

[191. Number of 1 Bits位1的个数 73](#_Toc519243071)

[198. House Robber打家劫舍 73](#_Toc519243072)

[202. Happy Number快乐数 75](#_Toc519243073)

[203. Remove Linked List Elements删除链表中的节点 77](#_Toc519243074)

[204. Count Primes计数质数 78](#_Toc519243075)

[205. Isomorphic Strings同构字符串 80](#_Toc519243076)

[217. Contains Duplicate存在重复元素 81](#_Toc519243077)

[219. Contains Duplicate II存在重复元素 II 81](#_Toc519243078)

[226. Invert Binary Tree翻转二叉树 82](#_Toc519243079)

[231. Power of Two 2的幂 83](#_Toc519243080)

[234. Palindrome Linked List回文链表 84](#_Toc519243081)

[237. Delete Node in a Linked List删除链表中的节点 85](#_Toc519243082)

[206. Reverse Linked List反转链表 86](#_Toc519243083)

[21. Merge Two Sorted Lists合并两个有序链表 88](#_Toc519243084)

[23. Merge k Sorted Lists合并K个排序链表 89](#_Toc519243085)

[19. Remove Nth Node From End of List删除链表的倒数第N个节点 92](#_Toc519243086)

[24. Swap Nodes in Pairs两两交换链表中的节点 93](#_Toc519243087)

[25. Reverse Nodes in k-Group k个一组翻转链表 95](#_Toc519243088)

[61. Rotate List旋转链表 99](#_Toc519243089)

[82. Remove Duplicates from Sorted List II删除排序链表中的重复元素 II 100](#_Toc519243090)

[86. Partition List分隔链表 101](#_Toc519243091)

[92. Reverse Linked List II反转链表 II 102](#_Toc519243092)

[109. Convert Sorted List to Binary Search Tree有序链表转换二叉搜索树 104](#_Toc519243093)

[138. Copy List with Random Pointer复制带随机指针的链表 105](#_Toc519243094)

[142. Linked List Cycle II环形链表 II 108](#_Toc519243095)

[补.环链表的相关问题 109](#_Toc519243096)

[143. Reorder List重排链表 110](#_Toc519243097)

[147. Insertion Sort List对链表进行插入排序 111](#_Toc519243098)

[148. Sort List排序链表 112](#_Toc519243099)

[369. Plus One Linked List 链表加一运算 114](#_Toc519243100)

[445. Add Two Numbers II两数相加 II 116](#_Toc519243101)

[328. Odd Even Linked List奇偶链表 118](#_Toc519243102)

[78. Subsets子集 120](#_Toc519243103)

[137. Single Number II只出现一次的数字 II 122](#_Toc519243104)

[260. Single Number III只出现一次的数字 III 123](#_Toc519243105)

[187. Repeated DNA Sequences重复的DNA序列 124](#_Toc519243106)

[338. Counting BitsBit位计数 126](#_Toc519243107)

[342. Power of Four 4的幂 128](#_Toc519243108)

[371. Sum of Two Integers两整数之和 129](#_Toc519243109)

[389. Find the Difference找不同 129](#_Toc519243110)

[393. UTF-8 Validation UTF-8 编码验证 131](#_Toc519243111)

[201. Bitwise AND of Numbers Range数字范围按位与 134](#_Toc519243112)

[268. Missing Number缺失数字 136](#_Toc519243113)

[318. Maximum Product of Word Lengths最大单词长度乘积 137](#_Toc519243114)

[397. Integer Replacement整数替换 139](#_Toc519243115)

[401. Binary Watch二进制手表 141](#_Toc519243116)

[405. Convert a Number to Hexadecimal数字转换为十六进制数 143](#_Toc519243117)

[421. Maximum XOR of Two Numbers in an Array数组中两个数的最大异或值 144](#_Toc519243118)

[461. Hamming Distance汉明距离 148](#_Toc519243119)

[477. Total Hamming Distance汉明距离总和 150](#_Toc519243120)

[476. Number Complement数字的补数 151](#_Toc519243121)

[693. Binary Number with Alternating Bits交替位二进制数 153](#_Toc519243122)

[762. Prime Number of Set Bits in Binary Representation 154](#_Toc519243123)

[二进制表示中质数个计算置位 154](#_Toc519243124)

1.Two Sum 两数之和

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。

你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。

示例: 给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

【知识点】散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希(Hash）表，函数f(key)为哈希(Hash) 函数。

【知识点】map和unordered\_map

map底层采用的是红黑树的实现查询的时间复杂度为O(logn)。

unordered\_map的底层才用哈希表的实现，查询的时间复杂度为是O(1)。

虽然unordered\_map的查询从算法上分析比map快，但是它有一些其它的消耗，比如哈希函数的构造和分析，还有如果出现哈希冲突解决哈希冲突等等都有一定的消耗，因此unordered\_map的效率在很大的程度上由它的hash函数算法决定，而红黑树的效率是一个稳定值。

所以unordered\_map内部就是无序的，数据是按散列函数插入到槽里面去的，数据之间无顺序可言，如果我们不需要内部有序，这种实现是没有问题的。

【分析】HashMap是常数级的查找效率。我们在遍历数组的时候，用target减去遍历到的数字，就是另一个需要的数字了，直接在HashMap中查找其是否存在即可，注意要判断查找到的数字不是第一个数字，比如target是4，遍历到了一个2，那么另外一个2不能是之前那个2。

【实现】步骤为：先遍历一遍数组，建立HashMap映射，然后再遍历一遍，开始查找，找到则记录index。

2. Add Two Numbers两数相加

给定两个非空链表来表示两个非负整数。位数按照逆序方式存储，它们的每个节点只存储单个数字。将两数相加返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例： 输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)

输出：7 -> 0 -> 8

原因：342 + 465 = 807

【知识点】ListNode双向链表

typedef struct listNode {

struct listNode \*prev; // 前置节点

struct listNode \*next; // 后置节点

void \*value; // 节点的值

} listNode;

【知识点】三目运算符 （）？ ：

语法为：条件表达式？表达式1：表达式2。

说明：问号前面的位置是判断的条件，判断结果为bool型，为true时调用表达式1，为false时调用表达式2。

其逻辑为：“如果为条件成立或者满足则执行表达式1，否则执行第二个。”常用在设置默认值,例如某个值不一定存在,则判断这个值是否存在,不存在给默认值(表达式2)

【分析】算法很简单，链表的数据类型也不难。就是建立一个新链表，然后把输入的两个链表从头往后撸，每两个相加，添加一个新节点到新链表后面，就是要处理下进位问题。还有就是最高位的进位问题要最后特殊处理一下。

总结注意：

需要考虑大数情况，所以不能直接用int保存每个LinkList对应的值。

考虑 [5] [5] -> [0, 1] 这种单独进位的情况。

class Solution {

public:

ListNode \*addTwoNumbers(ListNode \*l1, ListNode \*l2) {

ListNode \*res = new ListNode(-1);

ListNode \*cur = res;

int carry = 0;

while (l1 || l2) {

int n1 = l1 ? l1->val : 0;

int n2 = l2 ? l2->val : 0;

int sum = n1 + n2 + carry;

carry = sum / 10;

cur->next = new ListNode(sum % 10);

cur = cur->next;

if (l1) l1 = l1->next;

if (l2) l2 = l2->next;

}

if (carry) cur->next = new ListNode(1);

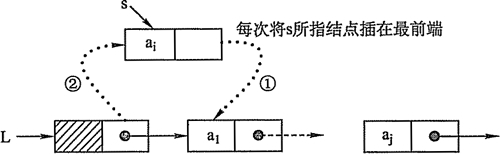
return res->next;

}

};

【知识点】单链表的建立：头插法和尾插法。

链表是线性表的一种表示形式，和线性表的另一种表达形式——数组不同。在链表中，没有固定长度的设定，表中节点是彼此相连的，表中的每一个节点链接下一个节点（即后继），链表头（Head）指向链表的第一个节点。所以，要想找到某一节点的位置，只需要确定它的上一结点的位置即可。这种性质使得链表的创建与插入变得相当灵活。

 1.头插法: 在链表的开头插入一个新的节点，也就是，必须使得链表头Head指向新节点，该新节点指向原来是表头的第一个节点。下图为头插法的具体情况：每个新节点都插在了原来第一个节点的前面。而有这种方法新建的链表是后来居前的，也就是链表是逆序的

插入步骤：

Node newNode; //生成新节点newNode

Node curr --> head.next ;

newNode.--> curr; //新节点指向原来的第一节点

head --> newNode;//头部指向新节点

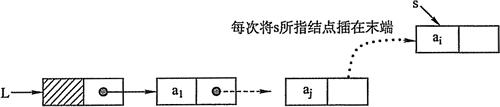
或： p->next = Head->next;

Head->next = p;

如果是不带头结点的链表那么对应是

p->next = Head;

Head = p;

2.尾插法：在链表的尾部插入一个节点。尾插法是比较容易理解并令大家习惯的插入形式，生成一个新节点后直接插入链表的尾端，也就是让原来最后一个节点指向该新节点。这也是链表长度增长的一种最基本的方式。后来居后，生成的链表是顺序的。

插入步骤：

Node newNode; // 生成新节点newNode

prev --> newNode; // 尾部指向新节点

或： for(t = Head; t->next; t=t->next); //结束时t指向尾节点

p->next = NULL; //进行插入

t->next = p;

【补充】第2题，定义ListNode

struct ListNode {

int val;

ListNode \*next;

ListNode(int x) : val(x), next(NULL) {}

};

# 7. Reverse Integer反转整数

给定一个 32 位有符号整数，将整数中的数字进行反转。

|  |  |  |
| --- | --- | --- |
| 示例 1:  输入: 123  输出: 321 | 示例 2:  输入: -123  输出: -321 | 示例 3:  输入: 120  输出: 21 |

注意:

假设我们的环境只能存储 32 位有符号整数，其数值范围是 [−231,  231− 1]。根据这个假设，如果反转后的整数溢出，则返回 0。

【分析】翻转数字问题需要注意的就是溢出问题，看了许多网上的解法，由于之前的OJ没有对溢出进行测试，所以网上很多人的解法没有处理溢出问题也能通过OJ。现在OJ更新了溢出测试，所以还是要考虑到。为什么会存在溢出问题呢，我们知道int型的数值范围是 -2147483648～2147483647， 那么如果我们要翻转 1000000009 这个在范围内的数得到 9000000001，而翻转后的数就超过了范围。

【知识点】INT\_MAX INT\_MIN包含在limits.h头文件中，在C/C++语言中，不能够直接使用-2147483648来代替最小负数，因为这不是一个数字，而是一个表达式。表达式的意思是对整数21473648取负，但是2147483648已经溢出了int的上限，所以定义为（-INT\_MAX -1）。

C中int类型是32位的，范围是-2147483648到2147483647 。

（1）最轻微的上溢是INT\_MAX + 1 :结果是 INT\_MIN;

（2）最严重的上溢是INT\_MAX + INT\_MAX :结果是-2;

（3）最轻微的下溢是INT\_MIN - 1:结果是是INT\_MAX;

（4）最严重的下溢是INT\_MIN + INT\_MIN:结果是0 。

【思考】在贴出答案的同时，OJ还提了一个问题 ：

To check for overflow/underflow, we could check if ret > 214748364 or ret < –214748364 before multiplying by 10. On the other hand, we do not need to check if ret == 214748364, why? （214748364 即为 INT\_MAX / 10）

为什么不用check是否等于214748364呢，因为输入的x也是一个整型数，所以x的范围也应该在 -2147483648～2147483647 之间，那么x的第一位只能是1或者2，翻转之后res的最后一位只能是1或2，所以res只能是 2147483641 或 2147483642 都在int的范围内。但是它们对应的x为 1463847412 和 2463847412，后者超出了数值范围。所以当过程中res等于 214748364 时， 输入的x只能为 1463847412， 翻转后的结果为 2147483641，都在正确的范围内，所以不用check。

【解答】

class Solution {

public:

int reverse(int x) {

int res = 0;

while (x != 0) {

if (abs(res) > INT\_MAX / 10) return 0;

res = res \* 10 + x % 10;

x /= 10;

}

return res;

}

};

# 9. Palindrome Number回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

|  |  |  |
| --- | --- | --- |
| 示例 1:  输入: 121  输出: true | 示例 2:  输入: -121  输出: false  解释: 从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。 | 示例 3:  输入: 10  输出: false  解释: 从右向左读, 为 01 。因此它不是一个回文数。 |

进阶:你能不将整数转为字符串来解决这个问题吗？

【分析】这道验证回文数字的题，如果将数字转为字符串，就变成了验证回文字符串的题，没啥难度了，直接来做进阶，不能转为字符串，而是直接对整数进行操作，我们可以利用取整和取余来获得我们想要的数字，比如 1221 这个数字，如果 计算 1221 / 1000， 则可得首位1， 如果 1221 % 10， 则可得到末尾1，进行比较，然后把中间的22取出继续比较。代码如下：

class Solution {

public:

bool isPalindrome(int x) {

if (x < 0) return false;

int div = 1;

while (x / div >= 10) div \*= 10;

while (x > 0) {

int left = x / div;

int right = x % 10;

if (left != right) return false;

x = (x % div) / 10;

div /= 100;

}

return true;

}

};

另一种很巧妙的解法，还是首先判断x是否为负数，trick，因为我们知道整数的最高位不能是0，所以回文数的最低位也不能为0，数字0除外，所以如果发现某个正数的末尾是0了，也直接返回false即可。解法，要验证回文数，那么就需要看前后半段是否对称，如果把后半段翻转一下，就看和前半段是否相等就行了。所以我们的做法就是取出后半段数字，进行翻转，具体做法是，每次通过对10取余，取出最低位的数字，然后加到取出数的末尾，就是将revertNum乘以10，再加上这个余数，这样我们的翻转也就同时完成了，每取一个最低位数字，x都要自除以10。这样当revertNum大于等于x的时候循环停止。由于回文数的位数可奇可偶，如果是偶数的话，那么revertNum就应该和x相等了；如果是奇数的话，那么最中间的数字就在revertNum的最高位上了，我们除以10以后应该和x是相等的，代码如下：

class Solution {

public:

bool isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0)) return false;

int revertNum = 0;

while (x > revertNum) {

revertNum = revertNum \* 10 + x % 10;

x /= 10;

}

return x == revertNum || x == revertNum / 10;

}

};

# 13. Roman to Integer罗马数字转整数

罗马数字包含以下七种字符：I， V， X， L，C，D 和 M

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 字符 | I | V | X | L | C | D | M |
| 数值 | 1 | 5 | 10 | 50 | 100 | 500 | 1000 |

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

* + - I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
    - X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
    - C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内

|  |  |  |
| --- | --- | --- |
| 输入: "III"  输出: 3 | 输入: "LVIII"  输出: 58  解释: C = 100,  L = 50,  XXX = 30,  III = 3. | 输入: "MCMXCIV"  输出: 1994  解释: M = 1000,  CM = 900,  XC = 90,  IV = 4. |
| 输入: "IV"  输出: 4 |
| 输入: "IX"  输出: 9 |

【分析】这道题好就好在没有让我们来验证输入字符串是不是罗马数字，这样省掉不少功夫。我们需要用到map数据结构，来将罗马数字的字母转化为对应的整数值，因为输入的一定是罗马数字，那么我们只要考虑两种情况即可：

第一，如果当前数字是最后一个数字，或者之后的数字比它小的话，则加上当前数字；第二，其他情况则减去这个数字。

另一种思路：我们也可以每次跟前面的数字比较，如果小于等于前面的数字，我们先加上当前的数字，如果大于的前面的数字，我们加上当前的数字减去二倍前面的数字，这样可以把在上一个循环多加数减掉

【知识点】map数据结构

Map是STL的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在map中出现一次，第二个可能称为该关键字的值）的数据处理能力，基于这个特性，它有可能在完成我们处理一对一数据的时候，在编程上提供快速通道。这里说下map内部数据的组织，map内部自建一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的。

自动建立Key－value的对应。key 和 value可以是任意你需要的类型。根据key值快速查找记录，查找的复杂度基本是Log(N)，如果有1000个记录，最多查找10次，1,000,000个记录，最多查找20次。

<1>建立并使用map

#include <map> //注意，STL头文件没有扩展名.h

map对象是模板类，需要关键字和存储对象两个模板参数：

std:map<int,string> personnel;

这样就定义了一个用int作为索引,并拥有相关联的指向string的指针.

<2> map添加数据；

map<int ,string> maplive;

1.maplive.insert(pair<int,string>(102,"aclive"));

2.maplive.insert(map<int,string>::value\_type(321,"hai"));

3, maplive[112]="April";//map中最简单最常用的插入添加！

<3>map的基本操作函数：

C++ Maps是一种关联式容器，包含“关键字/值”对

begin() 返回指向map头部的迭代器

clear(） 删除所有元素

count() 返回指定元素出现的次数

empty() 如果map为空则返回true

end() 返回指向map末尾的迭代器

equal\_range() 返回特殊条目的迭代器对

erase() 删除一个元素

find() 查找一个元素

get\_allocator() 返回map的配置器

insert() 插入元素

key\_comp() 返回比较元素key的函数

lower\_bound() 返回键值>=给定元素的第一个位置

max\_size() 返回可以容纳的最大元素个数

rbegin() 返回一个指向map尾部的逆向迭代器

rend() 返回一个指向map头部的逆向迭代器

size() 返回map中元素的个数

swap() 交换两个map

upper\_bound() 返回键值>给定元素的第一个位置

value\_comp() 返回比较元素value的函数

<4>map的sort问题：

Map中的元素是自动按key升序排序,所以不能对map用sort函数。

# 14. Longest Common Prefix 最常公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

|  |  |
| --- | --- |
| DEMO1 | 输入: ["flower","flow","flight"]  输出: "fl" |
| DEMO2 | 输入: ["dog","racecar","car"]  输出: ""  解释: 输入不存在公共前缀。 |

说明:所有输入只包含小写字母 a-z 。

【分析】我们定义两个变量i和j，其中i是遍历搜索字符串中的字符，j是遍历字符串集中的每个字符串。这里将单词上下排好，则相当于一个各行长度有可能不相等的二维数组，我们遍历顺序和一般的横向逐行遍历不同，而是采用纵向逐列遍历，在遍历的过程中，如果某一行没有了，说明其为最短的单词，因为共同前缀的长度不能长于最短单词，所以此时返回已经找出的共同前缀。我们每次取出第一个字符串的某一个位置的单词，然后遍历其他所有字符串的对应位置看是否相等，如果有不满足的直接返回res，如果都相同，则将当前字符存入结果，继续检查下一个位置的字符。

如果我们发现当前某个字符和下一行对应位置的字符不相等，说明不会再有更长的共同前缀了，我们直接通过用substr的方法直接取出共同前缀的子字符串。如果遍历结束前没有返回结果的话，说明第一个单词就是公共前缀，返回为结果即可。

【知识点】substring和subsrt都是获取指定位数 字符串的方法；

语法： substr(start,end)//第二个参数是截取字符串的长度

substring(start,end)//第二个参数是截取字符串最终的下标

substring(one); substr(one);//两者可以互换。

示例：

var a="abcdefghiklmnopqrstuvwxyz";

var b=a.substr(3,5);

var c=a.substring(3,5);

打印输出的结果是：

defgh

de

注意最后5下标是不会取到的意思是只能截取a字符串的3,4下标，

截取的时候是不会截取到最后一个。

【CODE】

class Solution {

public:

string longestCommonPrefix(vector<string>& strs) {

if (strs.empty()) return "";

for (int j = 0; j < strs[0].size(); ++j) { //思考和j++的区别

for (int i = 0; i < strs.size() - 1; ++i) {

if (j >= strs[i].size() || j >= strs[i + 1].size() || strs[i][j] != strs[i + 1][j]) {

return strs[i].substr(0, j);

}

}

}

return strs[0];

}

};

【思考】两者不同的地方在于++i 占用的空间比 i++ 要小，i++是多定义一个变量的。运算符重载： 前缀T& operator++(); 后缀T& operator++(int);

可以知道后缀的占用空间比前缀的要大，即用i++用的空间比++i的空间要大。

【知识点】vector

vector是一个能够存放任意类型的动态数组，能够增加和压缩数据。vector在C++标准模板库中的部分内容，它是一个多功能的，能够操作多种数据结构和算法的模板类和函数库。

创建vector：为了使用vector，必须包含头文件<vector>。另，vector属于std命名空间，因此需要通过命名限定，可以有如下三种方式，后两种方式更好，因为未引入的无关的内容。

using namespace std;

using namespace std::vector;

std::vector<int> vec;

1、如果你要表示的向量长度较长（需要为向量内部保存很多数），容易导致内存泄漏，而且效率会很低；

2、Vector作为函数的参数或者返回值时，需要注意它的写法：

double Distance(vector<int>&a, vector<int>&b) 其中的“&”绝对不能少！！！

实例：

vector<int>test;//建立一个vector，int为数组元素的数据类型，test为动态数组名

简单的使用方法如下：

vector<int>test;//建立一个vector

test.push\_back(1);

test.push\_back(2);//把1和2压入vector，这样test[0]就是1,test[1]就是2

现在以 std::vector<int> vec为例，描述相关函数的功能！

vec.begin()//指向迭代器中第一个元素。

vec.end()//指向迭代器中末端元素的下一个，指向一个不存在元素。

vec.push\_back(elem) //在尾部加入一个数据。

vec.pop\_back() //删除最后一个数据。

vec.capacity() //vector可用空间的大小。

vec.size()//返回容器中数据个数。

vec.empty() //判断容器是否为空。

vec.front() //传回第一个数据。

vec.back() //传回最后一个数据，不检查这个数据是否存在。

vec.at(index) //传回索引idx所指的数据，如果idx越界，抛出out\_of\_range。

vec.clear() //移除容器中所有数据。

vec.erase(iterator) //删除pos位置的数据，传回下一个数据的位置。

vec.erase(begin,end) //删除[beg,end)区间的数据，传回下一个数据的位置。注意：begin和end为iterator

vec.insert(position,elem) //在pos位置插入一个elem拷贝，传回新数据位置。

vec.insert(position,n,elem) //在pos位置插入n个elem数据，无返回值。

vec.insert(position,begin,end) //在pos位置插入在[beg,end)区间的数据，无返回值。

assign函数原型及功能：

void assign(const\_iterator first,const\_iterator last); //功能：将区间[first,last)的元素赋值到当前的vector中，当前vector会清除掉容器中之前的内容。

void assign(size\_type n,const T& x = T()); //功能：赋n个值为x的元素到当前vector中，当前vector会清除掉容器中之前的内容。

vec.resize(num)//重新指定vector的长度。

vec.resize(num,value)//重新指定vector的长度。并设定新增的元素的值

# 20. Valid Parentheses有效的括号

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

【分析】验证输入的字符串是否为括号字符串，包括大括号，中括号和小括号。这里我们需要用一个栈，我们开始遍历输入字符串，如果当前字符为左半边括号时，则将其压入栈中，如果遇到右半边括号时，若此时栈为空，则直接返回false，如不为空，则取出栈顶元素，若为对应的左半边括号，则继续循环，反之返回false。

【CODE】class Solution {

public:

bool isValid(string s) {

stack<char> parentheses;

for (int i = 0; i < s.size(); ++i) {

if (s[i] == '(' || s[i] == '[' || s[i] == '{') parentheses.push(s[i]);

else {

if (parentheses.empty()) return false;

if (s[i] == ')' && parentheses.top() != '(') return false;

if (s[i] == ']' && parentheses.top() != '[') return false;

if (s[i] == '}' && parentheses.top() != '{') return false;

parentheses.pop();

}

}

return parentheses.empty();

}

};

# 26. Remove Duplicates from Sorted Array删除排序数组中的重复项

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

【分析】快慢指针思想。使用快慢指针来记录遍历的坐标，最开始时两个指针都指向第一个数字，如果两个指针指的数字相同，则快指针向前走一步，如果不同，则两个指针都向前走一步，这样当快指针走完整个数组后，慢指针当前的坐标加1就是数组中不同数字的个数，并且数组依托慢指针修改为新数组。

【CODE】

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

if (nums.empty()) return 0;

int pre = 0, cur = 0, n = nums.size();

while (cur < n) {

if (nums[pre] == nums[cur]) ++cur;

else nums[++pre] = nums[cur++];

}

return pre + 1;

}

};

# 27. Remove Element移除元素

给定一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

【分析】只需要一个变量用来计数，然后遍历原数组，如果当前的值和给定值不同，我们就把当前值覆盖计数变量的位置，并将计数变量加1。

# 28. Implement strStr() 实现strStr()

实现 strStr() 函数。给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

说明:当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。对于本题而言，当 needle 是空字符串时我们应当返回 0 。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

DEMO1 输入: haystack = "hello", needle = "ll"

输出: 2

DEMO2 输入: haystack = "aaaaa", needle = "bba"

输出: -1

【分析】在一个字符串中找另一个字符串第一次出现的位置，首先要做一些判断，如果子字符串为空，则返回0，如果子字符串长度大于母字符串长度，则返回-1。然后我们开始遍历母字符串，我们并不需要遍历整个母字符串，而是遍历到剩下的长度和子字符串相等的位置即可，这样可以提高运算效率。然后对于每一个字符，我们都遍历一遍子字符串，一个一个字符的对应比较，如果对应位置有不等的，则跳出循环，如果一直都没有跳出循环，则说明子字符串出现了，则返回起始位置即可。

# 35. Search Insert Position搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。你可以假设数组中无重复元素。

DEMO 输入: [1,3,5,6], 5

输出: 2 输入: [1,3,5,6], 2

输出: 1 输入: [1,3,5,6], 7

输出: 4 输入: [1,3,5,6], 0

输出: 0

【分析】遍历一遍原数组，若当前数字大于或等于目标值，则返回当前坐标，如果遍历结束了，说明目标值比数组中任何一个数都要大，则返回数组长度n即可。

还可以用二分搜索法来优化我们的时间复杂度。

# 38. Count and Say报数

报数序列是指一个整数序列，按照其中的整数的顺序进行报数，得到下一个数。其前五项如下：

1. 1

2. 11

3. 21

4. 1211

5. 111221

1 被读作 "one 1" ("一个一") , 即 11。

11 被读作 "two 1s" ("两个一"）, 即 21。

21 被读作 "one 2", "one 1" （"一个二" , "一个一") , 即 1211。

给定一个正整数 n ，输出报数序列的第 n 项。

注意：整数顺序将表示为一个字符串。

【知识点】

1. to\_string( ) 数字转换成字符串

2. C++里的string 的+运算符重载过了的，数值类型会被转成char 然后加进去，当然有时会丢精度的。string 可以和数值、string、char\* 进行 + 运算。

【CODE】

class Solution {

public:

string countAndSay(int n) {

if (n <= 0) return "";

string res = "1";

while (--n) {

string cur = "";

for (int i = 0; i < res.size(); ++i) {

int cnt = 1;

while (i + 1 < res.size() && res[i] == res[i + 1]) {

++cnt;

++i;

}

cur += to\_string(cnt) + res[i];

}

res = cur;

}

return res;

}

};

# 53. Maximum Subarray最大子序和

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例: 输入: [-2,1,-3,4,-1,2,1,-5,4],

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

进阶: 如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解

【分析】用两种方法来解，分别是O(n)的解法，还有用分治法Divide and Conquer Approach，这个解法的时间复杂度是O(nlgn)，那我们就先来看O(n)的解法，定义两个变量res和curSum，其中res保存最终要返回的结果，即最大的子数组之和，curSum初始值为0，每遍历一个数字num，比较curSum + num和num中的较大值存入curSum，然后再把res和curSum中的较大值存入res，以此类推直到遍历完整个数组，可得到最大子数组的值存在res中。

分治法的思想就类似于二分搜索法，我们需要把数组一分为二，分别找出左边和右边的最大子数组之和，然后还要从中间开始向左右分别扫描，求出的最大值分别和左右两边得出的最大值相比较取最大的那一个。

【知识点】foreach语句

foreach语句是java5的新特征之一，在遍历数组、集合方面，foreach为开发人员提供了极大的方便。是for语句的特殊简化版本，但是foreach语句并不能完全取代for语句，然而，任何的foreach语句都可以改写为for语句版本。foreach并不是一个关键字，习惯上将这种特殊的for语句格式称之为“foreach”语句。从英文字面意思理解foreach也就是“for 每一个”的意思。实际上也就是这个意思。

foreach的语句格式：

for(元素类型t 元素变量x : 遍历对象obj){

引用了x的语句;

}

for (int num : nums) {} //等同于for(int i =0;i<nums.length;i++){}

# 58. Length of Last Word最后一个单词的长度

给定一个仅包含大小写字母和空格 ' ' 的字符串，返回其最后一个单词的长度。

如果不存在最后一个单词，请返回 0 。

说明：一个单词是指由字母组成，但不包含任何空格的字符串

示例: 输入: "Hello World" 输出: 5

【分析】由于我们只关于最后一个单词的长度，所以开头有多少个空格起始我们并不在意，我们从字符串末尾开始，先将末尾的空格都去掉，然后开始找非空格的字符的长度即可。。

【知识点】读入外挂 std::ios::sync\_with\_stdio(false);

取消cin于stdin的同步，可以将cin和scanf的效率相匹敌。

在ACM里，经常出现 数据集超大造成 cin TLE的情况。这时候大部分人（包括原来我也是）认为这是cin的效率不及scanf的错，甚至还上升到C语言和C++语言的执行效率层面的无聊争论。其实像上文所说，这只是C++为了兼容而采取的保守措施。我们可以在IO之前将stdio解除绑定，这样做了之后要注意不要同时混用cout和printf 之类。

在默认的情况下cin绑定的是cout，每次执行 << 操作符的时候都要调用flush，这样会增加IO负担。可以通过tie(0)（0表示NULL）来解除cin与cout的绑定，进一步加快执行效率。

|  |  |
| --- | --- |
| #include <iostream>  int main() {  std::ios::sync\_with\_stdio(false);  std::cin.tie(0);  // IO  } | static auto x = [](){  std::ios::sync\_with\_stdio(false);  std::cin.tie(0);  return 0;  }(); |

# 66. Plus One加一

给定一个非负整数组成的非空数组，在该数的基础上加一，返回一个新的数组。

最高位数字存放在数组的首位， 数组中每个元素只存储一个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

|  |  |  |
| --- | --- | --- |
| DEMO1 | 输入: [1,2,3]输出: [1,2,4] | 解释: 输入数组表示数字 123。 |
| DEMO2 | 输入: [4,3,2,1]输出: [4,3,2,2] | 解释: 输入数组表示数字 4321。 |

【分析】将一个数字的每个位上的数字分别存到一个一维向量中，最高位在最开头，我们需要给这个数字加一，即在末尾数字加一，如果末尾数字是9，那么则会有进位问题，而如果前面位上的数字仍为9，则需要继续向前进位。具体算法如下：首先判断最后一位是否为9，若不是，直接加一返回，若是，则该位赋0，再继续查前一位，同样的方法，知道查完第一位。如果第一位原本为9，加一后会产生新的一位，那么最后要做的是，查运算完的第一位是否为0，如果是，则在最前头加一个

【CODE】class Solution {

public:

vector<int> plusOne(vector<int>& digits) {

int n = digits.size();

for ( int i = n - 1 ; i >= 0 ; --i){

if (digits[i] == 9 ) digits[i]=0;

else{

digits[i] += 1;

return digits ;

}

}

if (digits.front() == 0) digits.insert(digits.begin() , 1);

return digits;

}

};

# 67. Add Binary二进制求和

给定两个二进制字符串，返回他们的和（用二进制表示）。

输入为非空字符串且只包含数字 1 和 0。

|  |  |
| --- | --- |
| DEMO1 | 输入: a = "11", b = "1"输出: "100" |
| DEMO2 | 输入: a = "1010", b = "1011"输出: "10101" |

【分析】巧妙又简洁的写法，用了两个指针分别指向a和b的末尾，然后每次取出一个字符，转为数字，若无法取出字符则按0处理，然后定义进位carry，初始化为0，将三者加起来，对2取余即为当前位的数字，对2取商即为当前进位的值，记得最后还要判断下carry，如果为1的话，要在结果最前面加上一个1。

【CODE】

class Solution {

public:

string addBinary(string a, string b) {

string res = "";

int m = a.size() - 1, n = b.size() - 1, carry = 0;

while (m >= 0 || n >= 0) {

int p = m >= 0 ? a[m--] - '0' : 0;

int q = n >= 0 ? b[n--] - '0' : 0;

int sum = p + q + carry;

res = to\_string(sum % 2) + res;

carry = sum / 2;

}

return carry == 1 ? "1" + res : res;

}

};

# 69. Sqrt(x) x 的平方根

实现 int sqrt(int x) 函数。计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

【分析】算一个候选值的平方，然后和x比较大小，为了缩短查找时间，我们采用二分搜索法来找平方根，找最后一个不小于目标值的数。

法二：利用牛顿迭代法，是用逼近法求方程根的神器，在这里也可以借用一下，因为要求x2 = n的解，令f(x)=x2-n，相当于求解f(x)=0的解，可以求出递推式如下：

xi+1=xi - (xi2 - n) / (2xi) = xi - xi / 2 + n / (2xi) = xi / 2 + n / 2xi = (xi + n/xi) / 2

【CODE】

class Solution {

public:

int mySqrt(int x) {

long res = x;

while (res \* res > x) {

res = (res + x / res) / 2;

}

return res;

}

};

# 70. Simplify Path爬楼梯

假设你正在爬楼梯。需要 n 步你才能到达楼顶。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。

【分析】由于每次最多爬2阶，楼梯的第i阶，只有可能从楼梯的第i-1阶与第i-2阶到达。故到达第i阶的爬法，只与第i-1阶和第i-2阶的爬法数量直接相关。

|  |  |
| --- | --- |
|  | 【动态规划原理】  1.设置递推数组dp[0…n]//确定原问题与子问题  2.dp[i]代表到达第i阶有多少走法//确定状态  3.dp[0]=0,dp[1]=1,dp[2]=2//确定边界状态的值  4. **Dp[i] = dp[i-1] + dp[i-2]**  **//**确定状态状态转换方程 |

【DEMO】

class Solution {

public:

int climbStairs(int n) {

std::vector<int> dp(n + 3, 0);

dp[1] = 1;

dp[2] = 2;

for (int i = 3; i <= n; i++){

dp[i] = dp[i-1] + dp[i-2];

}

return dp[n];

}

};

# 83. Remove Duplicates from Sorted List删除排序链表中的重复元素（区别26）

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

【分析】移除有序链表中的重复项需要定义个指针指向该链表的第一个元素，然后第一个元素和第二个元素比较，如果重复了，则删掉第二个元素，如果不重复，指针指向第二个元素。这样遍历完整个链表，则剩下的元素没有重复项。

【知识点】|| 与 &&

//表达式1||表达式2

//如果表达式1是true或者表达式2是true,整条式子的结果就是true,否则是false

//表达式1&&表达式2

//这个操作符要求两边的表达式都是true整条表达式才是true

【CODE】

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (!head || !head->next) return head;

//!表达式1||!表达式 2 表达式1,2中有一个为false，整个条件为true

ListNode \*start = head;

while (start && start->next) {

if (start->val == start->next->val) {

ListNode \*tmp = start->next;

start->next = start->next->next;

delete tmp;

} else start = start->next;

}

return head;

}

};

# 88. Merge Sorted Array合并两个有序数组

给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使得 num1 成为一个有序数组。

说明:初始化 nums1 和 nums2 的元素数量分别为 m 和 n。

你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

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

nums2 = [2,5,6], n = 3

【分析】方法一：混合插入有序数组，由于两个数组都是有序的，所有只要按顺序比较大小即可。最先想到的方法是建立一个m+n大小的新数组，然后逐个从A和B数组中取出元素比较，把较小的加入新数组，然后在考虑A数组有剩余和B数组有剩余的两种情况，最后在把新数组的元素重新赋值到A数组中即可。

方法二：更简洁的方法，而且不用申请新变量。算法思想是：由于合并后A数组的大小必定是m+n，所以从最后面开始往前赋值，先比较A和B中最后一个元素的大小，把较大的那个插入到m+n-1的位置上，再依次向前推。如果A中所有的元素都比B小，那么前m个还是A原来的内容，没有改变。如果A中的数组比B大的，当A循环完了，B中还有元素没加入A，直接用个循环把B中所有的元素覆盖到A剩下的位置。

【CODE】class Solution {

public:

void merge(int A[], int m, int B[], int n) {

int count = m + n - 1;

--m; --n;

while (m >= 0 && n >= 0) A[count--] = A[m] > B[n] ? A[m--] : B[n--];

while (n >= 0) A[count--] = B[n--];

}

};

# 94. Binary Tree Inorder Traversal二叉树的中序遍历

给定一个二叉树，返回它的中序遍历。

示例:输入: [1,null,2,3] 输出: [1,3,2]

【分析】二叉树的中序遍历顺序为左-根-右，可以有递归和非递归来解，其中非递归解法又分为两种，一种是使用栈来接，另一种不需要使用栈。我们先来看递归方法，十分直接，对左子结点调用递归函数，根节点访问值，右子节点再调用递归函数。

【CODE】// Recursion

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

inorder(root, res);

return res;

}

void inorder(TreeNode \*root, vector<int> &res) {

if (!root) return;

if (root->left) inorder(root->left, res);

res.push\_back(root->val);

if (root->right) inorder(root->right, res);

}

};

【分析】非递归使用栈的解法，需要用栈来做，思路是从根节点开始，先将根节点压入栈，然后再将其所有左子结点压入栈，然后取出栈顶节点，保存节点值，再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中。这样就保证了访问顺序为左-根-右。

【CODE】// Non-recursion

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

stack<TreeNode\*> s;

TreeNode \*p = root;

while (p || !s.empty()) {

while (p) {

s.push(p);

p = p->left;

}

p = s.top();

s.pop();

res.push\_back(p->val);

p = p->right;

}

return res;

}

};

【分析】另一种很巧妙的解法，这种方法不需要使用栈，所以空间复杂度为常量，这种非递归不用栈的遍历方法有个专门的名字，叫Morris Traversal，在介绍这种方法之前，我们先来引入一种新型树，叫 Threaded binary tree，螺纹二叉树，维基百科上关于它的英文定义：

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

就是说螺纹二叉树实际上是把所有原本为空的右子节点指向了中序遍历顺序之后的那个节点，把所有原本为空的左子节点都指向了中序遍历之前的那个节点。由于我们既不能用递归，又不能用栈，那我们如何保证访问顺序是中序遍历的左-根-右呢。原来我们需要构建一个螺纹二叉树，我们需要将所有为空的右子节点指向中序遍历的下一个节点，这样我们中序遍历完左子结点后，就能顺利的回到其根节点继续遍历了。

【算法】

1. 初始化指针cur指向root

2. 当cur不为空时

　 - 如果cur没有左子结点

　 a) 打印出cur的值

　　 b) 将cur指针指向其右子节点

　 - 反之

　 将pre指针指向cur的左子树中的最右子节点

　　　 \* 若pre不存在右子节点

　　　 a) 将其右子节点指回cur

　　　　 b) cur指向其左子节点

　　　 \* 反之

　　　　　 a) 将pre的右子节点置空

　　　　　 b) 打印cur的值

　　　　　 c) 将cur指针指向其右子节点

【CODE】// Non-recursion and no stack

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

if (!root) return res;

TreeNode \*cur, \*pre;

cur = root;

while (cur) {

if (!cur->left) {

res.push\_back(cur->val);

cur = cur->right;

} else {

pre = cur->left;

while (pre->right && pre->right != cur) pre = pre->right;

if (!pre->right) {

pre->right = cur;

cur = cur->left;

} else {

pre->right = NULL;

res.push\_back(cur->val);

cur = cur->right;

}

}

}

return res;

}

};

Morris遍历不仅仅对中序遍历有用，对先序和后序同样有用。所以对二叉树的三种常见遍历顺序(先序，中序，后序)就有三种解法(递归，非递归，Morris遍历)，总共有九段代码呀，熟练掌握这九种写法才算初步掌握了树的遍历~~ 至于二叉树的层序遍历也有递归和非递归解法，至于有没有Morris遍历的解法还有待大神们的解答~~

# 96. Unique Binary Search Trees不同的二叉搜索树

给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种？

【知识点】卡特兰数（Catalan number）

【分析】我们把n = 0 时赋为1，因为空树也算一种二叉搜索树，那么n = 1时的情况可以看做是其左子树个数乘以右子树的个数，左右字数都是空树，所以1乘1还是1。那么n = 2时，由于1和2都可以为跟，分别算出来，再把它们加起来即可。n = 2的情况可由下面式子算出：

dp[2] = dp[0] \* dp[1]　　　(1为根的情况)

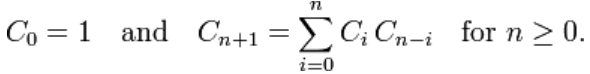
　　　　+ dp[1] \* dp[0]　　 (2为根的情况)

同理可写出 n = 3 的计算方法：

dp[3] = dp[0] \* dp[2]　　　(1为根的情况)

　　　　+ dp[1] \* dp[1]　　 (2为根的情况)

　　　 + dp[2] \* dp[0]　　 (3为根的情况)

由此可以得出卡塔兰数列的递推式为：

【CODE】

class Solution {

public:

int numTrees(int n) {

vector<int> dp(n + 1, 0);

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; ++i) {

for (int j = 0; j < i; ++j) {

dp[i] += dp[j] \* dp[i - j - 1];

}

}

return dp[n];

}

};

# 95. Unique Binary Search Trees II不同的二叉搜索树 II

给定一个整数 n，生成所有由 1 ... n 为节点所组成的二叉搜索树。

|  |  |
| --- | --- |
| 输入: 3  输出:[  [1,null,3,2],  [3,2,null,1],  [3,1,null,null,2],  [2,1,3],  [1,null,2,null,3]  ] | 解释:  以上的输出对应以下 5 种不同结构的二叉搜索树：  1 3 3 2 1  \ / / / \ \  3 2 1 1 3 2  / / \ \  2 1 2 3 |

【分析】之前那个只要求算出所有不同的二叉搜索树的个数，这道题让把那些二叉树都建立出来。这种建树问题一般来说都是用**递归**来解，这道题也不例外，划分左右子树，递归构造。至于递归函数中为什么都用的是指针，若不用指针，全部实例化的话会存在大量的对象拷贝，要调用拷贝构造函数。

【CODE】

class Solution {

public:

vector<TreeNode \*> generateTrees(int n) {

if (n == 0) return {};

return \*generateTreesDFS(1, n);

}

vector<TreeNode\*> \*generateTreesDFS(int start, int end) {

vector<TreeNode\*> \*subTree = new vector<TreeNode\*>();

if (start > end) subTree->push\_back(NULL);

else {

for (int i = start; i <= end; ++i) {

vector<TreeNode\*> \*leftSubTree = generateTreesDFS(start, i - 1);

vector<TreeNode\*> \*rightSubTree = generateTreesDFS(i + 1, end);

for (int j = 0; j < leftSubTree->size(); ++j) {

for (int k = 0; k < rightSubTree->size(); ++k) {

TreeNode \*node = new TreeNode(i);

node->left = (\*leftSubTree)[j];

node->right = (\*rightSubTree)[k];

subTree->push\_back(node);

}

}

}

}

return subTree;

}

};

# 98. Validate Binary Search Tree验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

【分析】这道验证二叉搜索树有很多种解法，可以利用它本身的性质来做，即左<根<右，也可以通过利用中序遍历结果为有序数列来做，下面我们先来看最简单的一种，就是利用其本身性质来做，初始化时带入系统最大值和最小值，在递归过程中换成它们自己的节点值，用long代替int就是为了包括int的边界条件。

【CODE】

// Recursion without inorder traversal

class Solution {

public:

bool isValidBST(TreeNode \*root) {

return isValidBST(root, LONG\_MIN, LONG\_MAX);

}

bool isValidBST(TreeNode \*root, long mn, long mx) {

if (!root) return true;

if (root->val <= mn || root->val >= mx) return false;

return isValidBST(root->left, mn, root->val) && isValidBST(root->right, root->val, mx);

}

};

【分析】一般的二叉搜索树是左<=根<右，而这道题设定为左<根<右，那么就可以用中序遍历来做。因为如果不去掉左=根这个条件的话，那么下边两个数用中序遍历无法区分：

20 20

/ \

20 20

它们的中序遍历结果都一样，但是左边的是BST，右边的不是BST。去掉等号的条件则相当于去掉了这种限制条件。下面我们来看使用中序遍历来做，这种方法思路很直接，通过中序遍历将所有的节点值存到一个数组里，然后再来判断这个数组是不是有序的。或者，不将遍历结果存入一个数组遍历完成再比较，而是每当遍历到一个新节点时和其上一个节点比较，如果不大于上一个节点那么则返回false，全部遍历完成后返回true。

也可以用非递归来做，需要用到栈，因为中序遍历可以非递归来实现。最后还有一种方法，由于中序遍历还有非递归且无栈的实现方法，称之为Morris遍历，可以参考Binary Tree Inorder Traversal，这种实现方法虽然写起来比递归版本要复杂的多，但是好处在于是O(1)空间复杂度。

# 99. Recover Binary Search Tree恢复二叉搜索树

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树。

进阶:使用 O(n) 空间复杂度的解法很容易实现。

你能想出一个只使用常数空间的解决方案吗？

【分析】传统的中序遍历递归，不过再应该输出节点值的地方，换成了判断pre和当前节点值的大小，如果pre的大，若first为空，则将first指向pre指的节点，把second指向当前节点。这样中序遍历完整个树，若first和second都存在，则交换它们的节点值即可。这个算法的空间复杂度仍为O(n)。

常数空间解决方案：用的Morris遍历，这是一种非递归且不使用栈，空间复杂度为O(1)的遍历方法，可参见之前Binary Tree Inorder Traversal 二叉树的中序遍历，在其基础上做些修改，加入first, second和parent指针，来比较当前节点值和中序遍历的前一节点值的大小，跟递归算法的思路相似。

【CODE】// Now O(1) space complexity

class Solution {

public:

void recoverTree(TreeNode \*root) {

TreeNode \*first = NULL, \*second = NULL, \*parent = NULL;

TreeNode \*cur, \*pre;

cur = root;

while (cur) {

if (!cur->left) {

if (parent && parent->val > cur->val) {

if (!first) first = parent;

second = cur;

}

parent = cur;

cur = cur->right;

} else {

pre = cur->left;

while (pre->right && pre->right != cur) pre = pre->right;

if (!pre->right) {

pre->right = cur;

cur = cur->left;

} else {

pre->right = NULL;

if (parent->val > cur->val) {

if (!first) first = parent;

second = cur;

}

parent = cur;

cur = cur->right;

}

}

}

if (first && second) swap(first->val, second->val);

}

};

# 100. Same Tree相同的树

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

【分析】判断两棵树是否相同和之前的判断两棵树是否对称都是一样的原理，利用深度优先搜索DFS来递归。

【知识点】DFS深度优先搜索；二叉树的四种遍历(层序，先序，中序，后序)

【CODE】class Solution {

public:

bool isSameTree(TreeNode \*p, TreeNode \*q) {

if (!p && !q) return true;

if ((p && !q) || (!p && q) || (p->val != q->val)) return false;

return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

}

};

# 101. Symmetric Tree对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

|  |  |
| --- | --- |
| 二叉树 [1,2,2,3,4,4,3] 是对称的。  1  / \  2 2  / \ / \  3 4 4 3 | [1,2,2,null,3,null,3] 则不是镜像对称的:  1  / \  2 2  \ \  3 3 |

【分析】判断二叉树是否是平衡树，比如有两个节点n1, n2，我们需要比较n1的左子节点的值和n2的右子节点的值是否相等，同时还要比较n1的右子节点的值和n2的左子结点的值是否相等，以此类推比较完所有的左右两个节点。我们可以用递归和迭代两种方法来实现，写法不同，但是算法核心都一样。

【CODE】class Solution {

public:

bool isSymmetric(TreeNode \*left, TreeNode \*right) {

if (!left && !right) return true;

if (left && !right || !left && right || left->val != right->val) return false;

return isSymmetric(left->left, right->right) && isSymmetric(left->right, right->left);

}

bool isSymmetric(TreeNode \*root) {

if (!root) return true;

return isSymmetric(root->left, root->right);

}

};

# 102. Binary Tree Level Order Traversal二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。

（即逐层地，从左到右访问所有节点）。

【分析】层序遍历二叉树是典型的广度优先搜索BFS的应用，但是这里稍微复杂一点的是，我们要把各个层的数分开，存到一个二维向量里面，大体思路还是基本相同的，建立一个queue，然后先把根节点放进去，这时候找根节点的左右两个子节点，这时候去掉根节点，此时queue里的元素就是下一层的所有节点，用一个for循环遍历它们，然后存到一个一维向量里，遍历完之后再把这个一维向量存到二维向量里，以此类推，可以完成层序遍历。

【CODE】// Iterative

class Solution {

public:

vector<vector<int> > levelOrder(TreeNode \*root) {

vector<vector<int> > res;

if (root == NULL) return res;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

vector<int> oneLevel;

int size = q.size();

for (int i = 0; i < size; ++i) {

TreeNode \*node = q.front();

q.pop();

oneLevel.push\_back(node->val);

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

res.push\_back(oneLevel);

}

return res;

}

};

【分析】递归的写法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字

【CODE】// Recursive

class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int> > res;

levelorder(root, 0, res);

return res;

}

void levelorder(TreeNode \*root, int level, vector<vector<int> > &res) {

if (!root) return;

if (res.size() == level) res.push\_back({});

res[level].push\_back(root->val);

if (root->left) levelorder(root->left, level + 1, res);

if (root->right) levelorder(root->right, level + 1, res);

}

};

# 103. Binary Tree Zigzag Level Order Traversal二叉树的锯齿形层次遍历

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

|  |  |
| --- | --- |
| 给定二叉树 [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7 | 返回锯齿形层次遍历如下：  [  [3],  [20,9],  [15,7]  ] |

【分析】这道二叉树的之字形层序遍历是之前那道[LeetCode] Binary Tree Level Order Traversal 二叉树层序遍历的变形，不同之处在于一行是从左到右遍历，下一行是从右往左遍历，交叉往返的之字形的层序遍历。根据其特点我们用到栈的后进先出的特点，这道题我们维护两个栈，相邻两行分别存到两个栈中，进栈的顺序也不相同，一个栈是先进左子结点然后右子节点，另一个栈是先进右子节点然后左子结点，这样出栈的顺序就是我们想要的之字形。

# 104. Maximum Depth of Binary Tree二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

【分析】求二叉树的最大深度问题用到深度优先搜索DFS，递归的完美应用，跟求二叉树的最小深度问题原理相同。

【CODE】class Solution {

public:

int maxDepth(TreeNode\* root) {

if (!root) return 0;

return 1 + max(maxDepth(root->left), maxDepth(root->right));

}

};

# 105. Construct Binary Tree from Preorder and Inorder Traversal

# 从前序与中序遍历序列构造二叉树

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。

【分析】先序的顺序的第一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。

【思考】怎么没有由先序和后序遍历建立二叉树呢？这是因为先序和后序遍历不能唯一的确定一个二叉树。对于先序遍历都为1 2 3的五棵二叉树，它们的中序遍历都不相同，而它们的后序遍历却有相同的，所以只有和中序遍历一起才能唯一的确定一棵二叉树。

|  |  |  |  |
| --- | --- | --- | --- |
| 1  / \  2 3 | preorder: 1 2 3  inorder: 2 1 3  postorder: 2 3 1 |  | preorder: 1 2 3  inorder: 3 2 1  postorder: 2 3 1 |
|  | preorder: 1 2 3  inorder: 2 3 1  postorder: 2 3 1 |  | preorder: 1 2 3  inorder: 1 3 2  postorder: 2 3 1 |
|  | preorder: 1 2 3  inorder: 1 2 3  postorder: 2 3 1 |  |  |

# 106. Construct Binary Tree from Inorder and Postorder Traversal

# 从中序与后序遍历序列构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:你可以假设树中没有重复的元素。

【分析】要求从中序和后序遍历的结果来重建原二叉树，我们知道中序的遍历顺序是左-根-右，后序的顺序是左-右-根，对于这种树的重建一般都是采用递归来做，可参见Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树。针对这道题，由于后序的顺序的最后一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。

# 107. Binary Tree Level Order Traversal II二叉树的层次遍历 II

给定一个二叉树，返回其节点值自底向上的层次遍历。

（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

|  |  |
| --- | --- |
| 给定二叉树 [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7 | 返回其自底向上的层次遍历为：  [  [15,7],  [9,20],  [3]  ] |

【分析】从底部层序遍历其实还是从顶部开始遍历，只不过最后存储的方式有所改变。另一种递归的解法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字。

# 108. Convert Sorted Array to Binary Search Tree将有序数组转换为二叉搜索树

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

【分析】将有序数组转为二叉搜索树，所谓二叉搜索树，是一种始终满足左<根<右的特性，如果将二叉搜索树按中序遍历的话，得到的就是一个有序数组了。那么反过来，我们可以得知，根节点应该是有序数组的中间点，从中间点分开为左右两个有序数组，在分别找出其中间点作为原中间点的左右两个子节点，这正是二分查找法的核心思想。

【CODE】

class Solution {

public:

TreeNode \*sortedArrayToBST(vector<int> &num) {

return sortedArrayToBST(num, 0 , num.size() - 1);

}

TreeNode \*sortedArrayToBST(vector<int> &num, int left, int right) {

if (left > right) return NULL;

int mid = (left + right) / 2;

TreeNode \*cur = new TreeNode(num[mid]);

cur->left = sortedArrayToBST(num, left, mid - 1);

cur->right = sortedArrayToBST(num, mid + 1, right);

return cur;

}

};

# 110. Balanced Binary Tree平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

【分析】求二叉树是否平衡，根据题目中的定义，高度平衡二叉树是每一个节点的两个字数的深度差不能超过1，那么我们肯定需要一个求各个点深度的函数，然后对每个节点的两个子树来比较深度差，时间复杂度为O(NlgN)。

上法正确但不是很高效，因为每一个点都会被上面的点计算深度时访问一次，我们可以进行优化。方法是如果我们发现子树不平衡，则不计算具体的深度，而是直接返回-1。那么优化后的方法为：对于每一个节点，我们通过checkDepth方法递归获得左右子树的深度，如果子树是平衡的，则返回真实的深度，若不平衡，直接返回-1，此方法时间复杂度O(N)，空间复杂度O(H)。

【CODE】class Solution {

public:

bool isBalanced(TreeNode \*root) {

if (checkDepth(root) == -1) return false;

else return true;

}

int checkDepth(TreeNode \*root) {

if (!root) return 0;

int left = checkDepth(root->left);

if (left == -1) return -1;

int right = checkDepth(root->right);

if (right == -1) return -1;

int diff = abs(left - right);

if (diff > 1) return -1;

else return 1 + max(left, right);

}

};

# 111. Minimum Depth of Binary Tree二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

【分析】二叉树的经典问题之最小深度问题就是就最短路径的节点个数，还是用深度优先搜索DFS来完成，万能的递归。

【CODE】

class Solution {

public:

int minDepth(TreeNode \*root) {

if (root == NULL) return 0;

if (root->left == NULL && root->right == NULL) return 1;

if (root->left == NULL) return minDepth(root->right) + 1;

else if (root->right == NULL) return minDepth(root->left) + 1;

else return 1 + min(minDepth(root->left), minDepth(root->right));

}

};

# 112. Path Sum路径总和

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明: 叶子节点是指没有子节点的节点。

【分析】这道求二叉树的路径需要用深度优先算法DFS的思想来遍历每一条完整的路径，也就是利用递归不停地寻找子节点的左右子节点，而调用递归函数的参数只有当前节点和sum值。首先，如果输入的是一个空节点，则直接返回false，如果输入的只有一个根节点，则比较当前根节点的值和参数sum值是否相同，若相同，返回true，否则false。 这个条件也是递归的终止条件。下面我们就要开始递归了，由于函数的返回值是True/False，我们可以同时两个方向一起递归，中间用或||连接，只要有一个是True，整个结果就是True。递归左右节点时，这时候的sum值应该是原sum值减去当前节点的值。

【CODE】

class Solution {

public:

bool hasPathSum(TreeNode \*root, int sum) {

if (root == NULL) return false;

if (root->left == NULL && root->right == NULL && root->val == sum ) return true;

return hasPathSum(root->left, sum - root->val) || hasPathSum(root->right, sum - root->val);

}

};

# 118. Pascal's Triangle杨辉三角

给定一个非负整数 numRows，生成杨辉三角的前 numRows 行。

【分析】具体生成算是：每一行的首个和结尾一个数字都是1，

从第三行开始，中间的每个数字都是上一行的左右两个数字之和。

【CODE】

class Solution {

public:

vector<vector<int> > generate(int numRows) {

vector<vector<int> > res;

if (numRows <= 0) return res;

res.assign(numRows, vector<int>(1));

for (int i = 0; i < numRows; ++i) {

res[i][0] = 1;

if (i == 0) continue;

for (int j = 1; j < i; ++j) {

res[i].push\_back(res[i-1][j] + res[i-1][j-1]);

}

res[i].push\_back(1);

}

return res;

}

};

【知识点】assign函数

# 119. Pascal's Triangle II杨辉三角 II

给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。

【知识点】杨辉三角是二项式系数的一种写法。杨辉三角形第n层（顶层称第0层，第1行，第n层即第n+1行，此处n为包含0在内的自然数）正好对应于二项式\left(a+b\right)^{n}展开的系数。例如第二层1 2 1是幂指数为2的二项式\left(a+b\right)^{2}展开形式a^{2}+2ab+b^{2}的系数。

杨辉三角主要有下列五条性质：

1. 杨辉三角以正整数构成，数字左右对称，每行由1开始逐渐变大，然后变小，回到1。
2. 第n行的数字个数为n个。
3. 第n行的第k个数字为组合数C_{n-1}^{k-1}。
4. 第n行数字和为2^{n-1}。
5. 除每行最左侧与最右侧的数字以外，每个数字等于它的左上方与右上方两个数字之和（也就是说，第n行第k个数字等于第n-1行的第k-1个数字与第k个数字的和）。这是因为有组合恒等式：C_{n}^{i}=C_{n-1}^{i-1}+C_{n-1}^{i}。可用此性质写出整个杨辉三角形。

【分析】题目有额外限制条件，程序只能使用O(k)的额外空间，那么这样就不能把每行都算出来，而是要用其他的方法, 我最先考虑用的是第三条性质，算出每个组合数来生成第n行系数。利用第五条性质，除了第一个和最后一个数字之外，其他的数字都是上一行左右两个值之和。那么我们只需要两个for循环，除了第一个数为1之外，后面的数都是上一次循环的数值加上它前面位置的数值之和，不停地更新每一个位置的值，便可以得到第n行的数字。

# 121. Best Time to Buy and Sell Stock买卖股票的最佳时机

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。注意：你不能在买入股票前卖出股票。

【分析】只需要遍历一次数组，用一个变量记录遍历过数中的最小值，然后每次计算当前值和这个最小值之间的差值最为利润，然后每次选较大的利润来更新。当遍历完成后当前利润即为所求。

# 122. Best Time to Buy and Sell Stock II买卖股票的最佳时机II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

【分析】这道题可以无限次买入和卖出。我们都知道炒股想挣钱当然是低价买入高价抛出，那么这里我们只需要从第二天开始，如果当前价格比之前价格高，则把差值加入利润中，因为我们可以昨天买入，今日卖出，若明日价更高的话，还可以今日买入，明日再抛出。以此类推，遍历完整个数组后即可求得最大利润。

# 125. Valid Palindrome验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。说明：本题中，我们将空字符串定义为有效的回文串。

示例 1:输入: "A man, a plan, a canal: Panama" 输出: true

示例 2输入: "race a car" 输出: false

【分析】验证回文字符串是比较常见的问题，所谓回文，就是一个正读和反读都一样的字符串，比如“level”或者“noon”等等就是回文串。但是这里，加入了空格和非字母数字的字符，增加了些难度，但其实原理还是很简单：只需要建立两个指针，left和right, 分别从字符的开头和结尾处开始遍历整个字符串，如果遇到非字母数字的字符就跳过，继续往下找，直到找到下一个字母数字或者结束遍历，如果遇到大写字母，就将其转为小写。等左右指针都找到字母数字时，比较这两个字符，若相等，则继续比较下面两个分别找到的字母数字，若不相等，直接返回false. 时间复杂度为O(n)。也可以用系统自带的判断是否是数母字符的判断函数isalnum。

【知识点】无差别判断大小写字母 (s + 32 - 'a') %32 //取余

# 136. Single Number只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度（O(n)）。 你可以不使用额外空间来实现吗（空间复杂度O(1)）？

【分析】时间复杂度必须是O(n)，并且空间复杂度为O(1)，使得不能用排序方法，也不能使用map数据结构。那么只能另辟蹊径，需要用位操作Bit Operation来解此题，即逻辑异或。逻辑异或的真值表为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | F | F | T | T |
| B | F | T | F | T |
| ⊕ | F | T | T | F |

由于数字在计算机是以二进制存储的，每位上都是0或1，如果我们把两个相同的数字异或，0与0异或是0,1与1异或也是0，那么我们会得到0。根据这个特点，我们把数组中所有的数字都异或起来，则每对相同的数字都会得0，然后最后剩下来的数字就是那个只有1次的数字。

【CODE】class Solution {

public:

int singleNumber(vector<int>& nums) {

int res = 0;

for (auto num : nums) res ^= num;

return res;

}

};

# 141. Linked List Cycle环形链表

给定一个链表，判断链表中是否有环。

进阶：你能否不使用额外空间解决此题？

【分析】这道题是快慢指针的经典应用。只需要设两个指针，一个每次走一步的慢指针和一个每次走两步的快指针，如果链表里有环的话，两个指针最终肯定会相遇。

【CODE】class Solution {

public:

bool hasCycle(ListNode \*head) {

ListNode \*slow = head, \*fast = head;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) return true;

}

return false;

}

};

# 155. Min Stack最小栈

设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) -- 将元素 x 推入栈中。

pop() -- 删除栈顶的元素。

top() -- 获取栈顶元素。

getMin() -- 检索栈中的最小元素。

【分析】这道最小栈跟原来的栈相比就是多了一个功能，可以返回该栈的最小值。使用两个栈来实现，一个栈来按顺序存储push进来的数据，另一个用来存出现过的最小值。

# 160. Intersection of Two Linked Lists相交链表

编写一个程序，找到两个单链表相交的起始节点。

注意： 如果两个链表没有交点，返回 null.

在返回结果后，两个链表仍须保持原有的结构。

可假定整个链表结构中没有循环。

程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

【分析】这道求两个链表的交点题要求执行时间为O(n)，则不能利用类似冒泡法原理去暴力查找相同点，事实证明如果链表很长的话，那样的方法效率很低。如果两个链长度相同的话，那么对应的一个个比下去就能找到，所以只需要把长链表变短即可。具体算法为：分别遍历两个链表，得到分别对应的长度。然后求长度的差值，把较长的那个链表向后移动这个差值的个数，然后一一比较即可。

一种特别巧妙的方法，虽然题目中强调了链表中不存在环，但是我们可以用环的思想来做，我们让两条链表分别从各自的开头开始往后遍历，当其中一条遍历到末尾时，我们跳到另一个条链表的开头继续遍历。两个指针最终会相等，而且只有两种情况，一种情况是在交点处相遇，另一种情况是在各自的末尾的空节点处相等。为什么一定会相等呢，因为两个指针走过的路程相同，是两个链表的长度之和，所以一定会相等。这个思路真的很巧妙，而且更重要的是代码写起来特别的简洁。

【CODE】

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

if (!headA || !headB) return NULL;

ListNode \*a = headA, \*b = headB;

while (a != b) {

a = a ? a->next : headB;

b = b ? b->next : headA;

}

return a;

}

};

# 167. Two Sum II - Input array is sorted两数之和 II - 输入有序数组

给定一个已按照升序排列的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:返回的下标值（index1 和 index2）不是从零开始的。

你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

【分析】一道Two Sum的衍生题，这道题其实应该更容易一些，因为给定的数组是有序的，而且题目中限定了一定会有解，我最开始想到的方法是二分法来搜索，因为一定有解，而且数组是有序的，那么第一个数字肯定要小于目标值target，那么我们每次用二分法来搜索target - numbers[i]即可。

这种方法并不效率，时间复杂度是O(nlgn)，我们再来看一种O(n)的解法，我们只需要两个指针，一个指向开头，一个指向末尾，然后向中间遍历，如果指向的两个数相加正好等于target的话，直接返回两个指针的位置即可，若小于target，左指针右移一位，若大于target，右指针左移一位，以此类推直至两个指针相遇停止。

【CODE】// O(n)

class Solution {

public:

vector<int> twoSum(vector<int>& numbers, int target) {

int l = 0, r = numbers.size() - 1;

while (l < r) {

int sum = numbers[l] + numbers[r];

if (sum == target) return {l + 1, r + 1};

else if (sum < target) ++l;

else --r;

}

return {};

}

};

# 168. Excel Sheet Column Title Excel表列名称

给定一个正整数，返回它在 Excel 表中相对应的列名称。

|  |  |
| --- | --- |
| 1 -> A  2 -> B  3 -> C  ...  26 -> Z  27 -> AA  28 -> AB  ... | 输入: 1  输出: "A" |
| 输入: 28  输出: "AB" |
| 输入: 701  输出: "ZY" |

【分析】此题和Excel Sheet Column Number 求Excel表列序号是一起的，起始原理都一样，就是一位一位的求，此题从低位往高位求，每进一位，则把原数缩小26倍，再对26取余，之后减去余数，再缩小26倍，以此类推，可以求出各个位置上的字母。最后只需将整个字符串翻转一下即可。

【CODE】//感受简洁代码的魅力：

// Non-recursion

class Solution {

public:

string convertToTitle(int n) {

string res;

while (n) {

res += --n % 26 + 'A';

n /= 26;

}

return string(res.rbegin(), res.rend());

}

};

//使用递归解法

class Solution {

public:

string convertToTitle(int n) {

return n == 0 ? "" : convertToTitle(n / 26) + (char)(--n % 26 + 'A');

}

};

# 171. Excel Sheet Column Number Excel表列序号

给定一个Excel表格中的列名称，返回其相应的列序号。

【分析】这题实际上相当于一种二十六进制转十进制的问题，并不难，只要一位一位的转换即可。

# 169. Majority Element求众数

给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。你可以假设数组是非空的，并且给定的数组总是存在众数。

【分析】这是道求众数的问题，有很多种解法，比较好的有两种，一种是用哈希表，这种方法需要O(n)的时间和空间，另一种是用一种叫摩尔投票法 Moore Voting，需要O(n)的时间和O(1)的空间，比前一种方法更好。

摩尔投票法先将第一个数字假设为众数，然后把计数器设为1，比较下一个数和此数是否相等，若相等则计数器加一，反之减一。然后看此时计数器的值，若为零，则将当前值设为候选众数。以此类推直到遍历完整个数组，当前候选众数即为该数组的众数。弄懂摩尔投票法的精髓，首先要明确的是这个很厉害的方法是有前提的，就是数组中一定要有众数的存在才能使用，下面我们来看本算法的思路，这是一种先假设候选者，然后再进行验证的算法。我们现将数组中的第一个数假设为众数，然后进行统计其出现的次数，如果遇到同样的数，则计数器自增1，否则计数器自减1，如果计数器减到了0，则更换当前数字为候选者。这是一个很巧妙的设定，也是本算法的精髓所在，为啥遇到不同的要计数器减1呢，为啥减到0了又要更换候选者呢？首先是有那个强大的前提存在，一定会有一个出现超过半数的数字存在，那么如果计数器减到0了话，说明目前不是候选者数字的个数已经跟候选者的出现个数相同了，那么这个候选者已经很weak，不一定能出现超过半数，我们选择更换当前的候选者。那有可能你会有疑问，那万一后面又大量的出现了之前的候选者怎么办，不需要担心，如果之前的候选者在后面大量出现的话，其又会重新变为候选者，直到最终验证成为正确的众数。

【CODE】

class Solution {

public:

int majorityElement(vector<int>& nums) {

int res = 0, cnt = 0;

for (int num : nums) {

if (cnt == 0) {res = num; ++cnt;}

else (num == res) ? ++cnt : --cnt;

}

return res;

}

};

【分析】下面这种解法利用到了位操作Bit Manipulation来解，将中位数按位来建立，从0到31位，每次统计下数组中该位上0和1的个数，如果1多，那么我们将结果res中该位变为1，最后累加出来的res就是中位数了，相当赞的方法，这种思路尤其在这道题的延伸Majority Element II中有重要的应用。

【CODE】

class Solution {

public:

int majorityElement(vector<int>& nums) {

int res = 0, n = nums.size();

for (int i = 0; i < 32; ++i) {

int ones = 0, zeros = 0;

for (int num : nums) {

if (ones > n / 2 || zeros > n / 2) break;

if ((num & (1 << i)) != 0) ++ones;

else ++zeros;

}

if (ones > zeros) res |= (1 << i);

}

return res;

}

};

# 172. Factorial Trailing Zeroes 阶乘后的零

给定一个整数 n，返回 n! 结果尾数中零的数量。

说明: 你算法的时间复杂度应为 O(log n) 。

【分析】求一个数的阶乘末尾0的个数，也就是要找乘数中10的个数，而10可分解为2和5，而我们可知2的数量又远大于5的数量，那么此题即便为找出5的个数。仍需注意的一点就是，像25,125，这样的不只含有一个5的数字需要考虑进去。

【CODE】

class Solution {

public:

int trailingZeroes(int n) {

int res = 0;

while (n) {

res += n / 5;

n /= 5;

}

return res;

}

};

【分析】递归的解法，思路和上面完全一样，写法更简洁了，一行搞定。

【CODE】

class Solution {

public:

int trailingZeroes(int n) {

return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);

}

};

# 189. Rotate Array旋转数组

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

说明:尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。

要求使用空间复杂度为 O(1) 的原地算法。

|  |  |
| --- | --- |
| 输入: [1,2,3,4,5,6,7] 和 k = 3  输出: [5,6,7,1,2,3,4]  解释:  向右旋转 1 步: [7,1,2,3,4,5,6]  向右旋转 2 步: [6,7,1,2,3,4,5]  向右旋转 3 步: [5,6,7,1,2,3,4] | 输入: [-1,-100,3,99] 和 k = 2  输出: [3,99,-1,-100]  解释:  向右旋转 1 步: [99,-1,-100,3]  向右旋转 2 步: [3,99,-1,-100] |

【分析】有一种O(n)的空间复杂度的方法，我们复制一个和nums一样的数组，然后利用映射关系i -> (i+k)%n来交换数字。//超时！

【CODE】

class Solution {

public:

void rotate(vector<int>& nums, int k) {

vector<int> t = nums;

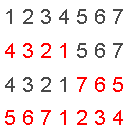
for (int i = 0; i < nums.size(); ++i) {

nums[(i + k) % nums.size()] = t[i];

}

}

};

【分析】类似翻转字符的方法，思路是先把前n-k个数字翻转一下，再把后k个数字翻转一下，最后再把整个数组翻转一下。//超时！

【CODE】

class Solution {

public:

void rotate(vector<int>& nums, int k) {

if (nums.empty() || (k %= nums.size()) == 0) return;

int n = nums.size();

reverse(nums.begin(), nums.begin() + n - k);

reverse(nums.begin() + n - k, nums.end());

reverse(nums.begin(), nums.end());

}

};

【分析】旋转数组的操作也可以看做从数组的末尾取k个数组放入数组的开头，所以我们用STL的push\_back和erase可以很容易的实现这些操作。

【CODE】

class Solution {

public:

void rotate(vector<int>& nums, int k) {

if (nums.empty() || (k %= nums.size()) == 0) return;

int n = nums.size();

for (int i = 0; i < n - k; ++i) {

nums.push\_back(nums[0]);

nums.erase(nums.begin());

}

}

};

# 190. Reverse Bits颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

输入: 43261596

输出: 964176192

解释: 43261596 的二进制表示形式为 00000010100101000001111010011100 ，

返回 964176192，其二进制表示形式为 00111001011110000010100101000000 。

进阶:如果多次调用这个函数，你将如何优化你的算法？

【分析】对于这道题，我们只需要把要翻转的数从右向左一位位的取出来，如果取出来的是1，我们将结果res左移一位并且加上1；如果取出来的是0，我们将结果res左移一位，然后将n右移一位即可。

【CODE】class Solution {

public:

uint32\_t reverseBits(uint32\_t n) {

uint32\_t ret = 0;

/\*

for(int i=0;i<32;i++){

ret=ret\*2+n%2;

n = n/2;

}

\*/

for(int i=0;i<32;i++){

ret=(ret<<1)^(n&1);

n>>=1;

}

return ret;

}

};

【知识点】二进制操作

左移运算符 << 右移运算符 >>

按位与&、按位与或|、按位异或^运算符

* & 按位“与”运算符 (&) 会将第一操作数的每一位与第二操作数的相应位进行比较。如果两个位均为 1，则对应的结果位将设置为 1。否则，将对应的结果位设置为 0。按位“与”运算符的两个操作数必须为整型。
* | 按位或运算符则是将两个数字的二进制值的每一位进行或运算, 或运算也就是二进制的每一位分别进行运算，如果两个都是 0 那么此位为 0，至少有一个数该位为 1，或运算结果就是 1。可以看得出来，任意数与自身进行或运算还是这个数。
* ^ 按位“异或”运算符 (^) 将第一操作数的每个位与第二操作数的相应位进行比较。如果一个位是 0，另一个位是 1，则相应的结果位将设置为 1。否则，将对应的结果位设置为 0。

【Genius】

class Solution {

public:

uint32\_t reverseBits(uint32\_t n) {

n = (n >> 16) | (n << 16);

n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);

n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);

n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);

n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);

return n;

}

};

for 8 bit binary number abcdefgh, the process is as follow:

abcdefgh -> efghabcd -> ghefcdab -> hgfedcba

# 191. Number of 1 Bits位1的个数

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 ‘1’ 的个数（也被称为汉明重量）。

【CODE】//熟悉使用位操作

class Solution {

public:

int hammingWeight(uint32\_t n) {

int res = 0;

for (int i = 0; i < 32; ++i) {

res += (n & 1);

n = n >> 1;

}

return res;

}

};

# 198. House Robber打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

|  |
| --- |
| 输入: [1,2,3,1] 输出: 4  解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。  偷窃到的最高金额 = 1 + 3 = 4 。 |
| 输入: [2,7,9,3,1]输出: 12  解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。偷窃到的最高金额 = 2 + 9 + 1 = 12 。 |

【分析】这道题的本质相当于在一列数组中取出一个或多个不相邻数，使其和最大。那么我们对于这类求极值的问题首先考虑动态规划Dynamic Programming来解，我们维护一个一位数组dp，其中dp[i]表示到i位置时不相邻数能形成的最大和，那么递推公式怎么写呢，我们先拿一个简单的例子来分析一下，比如说nums为{3, 2, 1, 5}，那么我们来看我们的dp数组应该是什么样的，首先dp[0]=3没啥疑问，再看dp[1]是多少呢，由于3比2大，所以我们抢第一个房子的3，当前房子的2不抢，所以dp[1]=3，那么再来看dp[2]，由于不能抢相邻的，所以我们可以用再前面的一个的dp值加上当前的房间值，和当前房间的前面一个dp值比较，取较大值当做当前dp值，所以我们可以得到递推公式dp[i] = max(num[i] + dp[i - 2], dp[i - 1]), 由此看出我们需要初始化dp[0]和dp[1]，其中dp[0]即为num[0]，dp[1]此时应该为max(num[0], num[1])。

【CODE】// DP

class Solution {

public:

int rob(vector<int> &num) {

if (num.size() <= 1) return num.empty() ? 0 : num[0];

vector<int> dp = {num[0], max(num[0], num[1])};

for (int i = 2; i < num.size(); ++i) {

dp.push\_back(max(num[i] + dp[i - 2], dp[i - 1]));

}

return dp.back();

}

};

还有一种解法，核心思想还是用DP，分别维护两个变量a和b，然后按奇偶分别来更新a和b，这样就可以保证组成最大和的数字不相邻。

【CODE】class Solution {

public:

int rob(vector<int> &nums) {

int a = 0, b = 0;

for (int i = 0; i < nums.size(); ++i) {

int m = a, n = b;

a = n + nums[i];

b = max(m, n);

}

return max(a, b);

}

};

# 202. Happy Number快乐数

编写一个算法来判断一个数是不是“快乐数”。

一个“快乐数”定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是无限循环但始终变不到 1。如果可以变为 1，那么这个数就是快乐数。

|  |  |
| --- | --- |
| 输入: 19  输出: true  解释:  12 + 92 = 82  82 + 22 = 68  62 + 82 = 100  12 + 02 + 02 = 1 | 输入:11 输出:false  1^2 + 1^2 = 2  2^2 = 4  4^2 = 16  1^2 + 6^2 = 37  3^2 + 7^2 = 58  5^2 + 8^2 = 89  8^2 + 9^2 = 145  1^2 + 4^2 + 5^2 = 42  4^2 + 2^2 = 20  2^2 + 0^2 = 4 |

【分析】题目定义了一种快乐数，并且给出了快乐数的演示。但是问题的解决还需要找到发现不快乐数的方法，那么我们来看一个不是快乐数的情况，比如数字11有如右的计算过程，可以看到4这个数字重复出现，之后的数字又都会重复之前两个4之间的顺序，这个循环中不包含1，那么数字11不是一个快乐数。

考虑怎么用代码来实现，我们可以用set来记录所有出现过的数字，然后每出现一个新数字，在set中查找看是否存在，若不存在则加入表中，若存在则跳出循环，并且判断此数是否为1，若为1返回true，不为1返回false。

这道题也可以不用set来做，关于不快乐数有个特点，循环的数字中必定会有4，所有不快乐数的数位平方和计算，最後都会进入 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4 的循环中。，就是利用这个性质，就可以不用set。

【CODE】class Solution {

public:

bool isHappy(int n) {

while (n != 1 && n != 4) {

int t = 0;

while (n) {

t += (n % 10) \* (n % 10);

n /= 10;

}

n = t;

}

return n == 1;

}

};

# 203. Remove Linked List Elements删除链表中的节点

删除链表中等于给定值 val 的所有节点。

Example

Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6

Return: 1 --> 2 --> 3 --> 4 --> 5

【分析】移除链表元素是链表的基本操作之一，考察了基本的链表遍历和设置指针的知识点，我们只需定义几个辅助指针，然后遍历原链表，遇到与给定值相同的元素，将该元素的前后连个节点连接起来，然后删除该元素即可，要注意的是还是需要在链表开头加上一个dummy node。

【CODE】class Solution {

public:

ListNode\* removeElements(ListNode\* head, int val) {

ListNode \*dummy = new ListNode(-1), \*pre = dummy;

dummy->next = head;

while (pre->next) {

if (pre->next->val == val) {

ListNode \*t = pre->next;

pre->next = t->next;

t->next = NULL;

delete t;

} else {

pre = pre->next;

}

}

return dummy->next;

}

};

也可以用递归来解，写法很简洁，通过递归调用到链表末尾，然后回来，需要要删的元素，将链表next指针指向下一个元素即可。

【CODE】

class Solution {

public:

ListNode\* removeElements(ListNode\* head, int val) {

if (!head) return NULL;

head->next = removeElements(head->next, val);

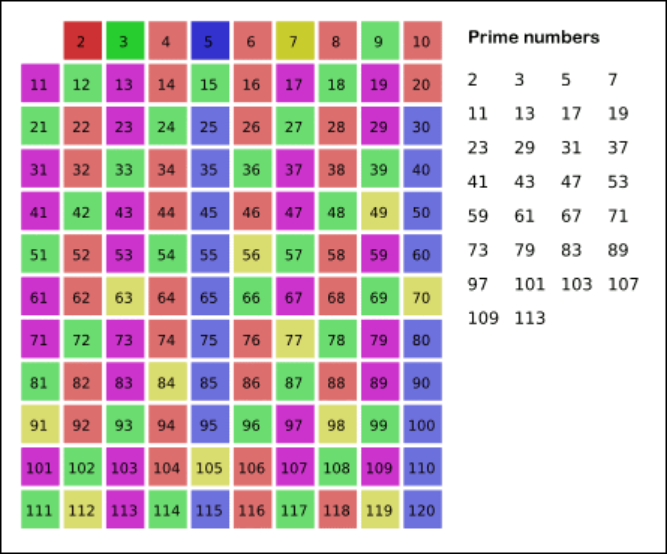
return head->val == val ? head->next : head;

}

};

# 204. Count Primes计数质数

统计所有小于非负整数 n 的质数的数量。

【知识点】埃拉托斯特尼筛法Sieve of Eratosthenes

我们从2开始遍历到根号n，先找到第一个质数2，然后将其所有的倍数全部标记出来，然后到下一个质数3，标记其所有倍数，一次类推，直到根号n，此时数组中未被标记的数字就是质数。

【分析】这道题给定一个非负数n，让我们求小于n的质数的个数，我们需要一个n-1长度的bool型数组来记录每个数字是否被标记，长度为n-1的原因是题目说是小于n的质数个数，并不包括n。 然后我们用两个for循环来实现埃拉托斯特尼筛法。

【CODE】

class Solution {

public:

int countPrimes(int n) {

vector<bool> num(n - 1, true);

num[0] = false;

int res = 0, limit = sqrt(n);

for (int i = 2; i <= limit; ++i) {

if (num[i - 1]) {

for (int j = i \* i; j < n; j += i) {

num[j - 1] = false;

}

}

}

for (int j = 0; j < n - 1; ++j) {

if (num[j]) ++res;

}

return res;

}

};

# 205. Isomorphic Strings同构字符串

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以被替换得到 t ，那么这两个字符串是同构的。

所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。**两个字符不能映射到同一个字符上，但字符可以映射自己本身。**（可以假设 s 和 t 具有相同的长度。）

|  |  |  |
| --- | --- | --- |
| 输入: s = "egg", t = "add"  输出: true | 输入: s = "foo", t = "bar"  输出: false | 输入: s = "paper", t = "title"  输出: true |

【分析】求同构字符串，相同的字符一定要被同一个字符替代，且一个字符不能被多个字符替代，即不能出现一对多的映射。

根据一对一映射的特点，我们需要用两个哈希表分别来记录原字符串和目标字符串中字符出现情况，由于ASCII码只有256个字符，所以我们可以用一个256大小的数组来代替哈希表，并初始化为0，我们遍历原字符串，分别从源字符串和目标字符串取出一个字符，然后分别在两个哈希表中查找其值，若不相等，则返回false，若相等，将其值更新为i + 1，因为默认的值是0，所以我们更新值为i + 1，这样当 i=0 时，则映射为1，如果不加1的话，那么就无法区分是否更新了。**//理解理解！！！**

【CODE】class Solution {

public:

bool isIsomorphic(string s, string t) {

int m1[256] = {0}, m2[256] = {0}, n = s.size();

for (int i = 0; i < n; ++i) {

if (m1[s[i]] != m2[t[i]]) return false;

m1[s[i]] = i + 1;

m2[t[i]] = i + 1;

}

return true;

}

};

# 217. Contains Duplicate存在重复元素

给定一个整数数组，判断是否存在重复元素。

如果任何值在数组中出现至少两次，函数返回 true。如果数组中每个元素都不相同，则返回 false。

【分析】使用一个哈希表，遍历整个数组，如果哈希表里存在，返回false，如果不存在，则将其放入哈希表中。

另一种解法，就是先将数组排个序，然后再比较相邻两个数字是否相等，时间复杂度取决于排序方法。

【CODE】//hash

class Solution {

public:

bool containsDuplicate(vector<int>& nums) {

unordered\_map<int, int> m;

for (int i = 0; i < nums.size(); ++i) {

if (m.find(nums[i]) != m.end()) return true;

++m[nums[i]];

}

return false;

}

};

# 219. Contains Duplicate II存在重复元素 II

给定一个整数数组和一个整数 k，判断数组中是否存在两个不同的索引 i 和 j，使得 nums [i] = nums [j]，并且 i 和 j 的差的绝对值最大为 k。

【分析】需要一个哈希表，来记录每个数字和其坐标的映射，然后我们需要一个变量d来记录第一次出现重复数字的坐标差。

# 226. Invert Binary Tree翻转二叉树

翻转一棵二叉树。

【分析】翻转二叉树，是树的基本操作之一，不算难题。可以用递归和非递归两种方法来解。先来看递归的方法，写法非常简洁，五行代码搞定，交换当前左右节点，并直接调用递归即可。

【CODE】// Recursion

class Solution {

public:

TreeNode\* invertTree(TreeNode\* root) {

if (!root) return NULL;

TreeNode \*tmp = root->left;

root->left = invertTree(root->right);

root->right = invertTree(tmp);

return root;

}

};

非递归的方法也不复杂，跟二叉树的层序遍历一样，需要用queue来辅助，先把根节点排入队列中，然后从队中取出来，交换其左右节点，如果存在则分别将左右节点在排入队列中，以此类推直到队列中木有节点了停止循环，返回root即可。

【CODE】// Non-Recursion

class Solution {

public:

TreeNode\* invertTree(TreeNode\* root) {

if (!root) return NULL;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode \*node = q.front(); q.pop();

TreeNode \*tmp = node->left;

node->left = node->right;

node->right = tmp;

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

return root;

}

};

# 231. Power of Two 2的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

【分析】判断一个数是否为2的次方数，而且要求时间和空间复杂度都为常数，那么对于这种玩数字的题，我们应该首先考虑位操作 Bit Operation。那么我们来观察下2的次方数的二进制写法的特点：

1 2 4 8 16 　　....

1 10 100 1000 10000　....

那么我们很容易看出来2的次方数都只有一个1，剩下的都是0，所以解题思路就有了，我们只要每次判断最低位是否为1，然后向右移位，最后统计1的个数即可判断是否是2的次方数。技巧，如果一个数是2的次方数的话，那么它的二进数必然是最高位为1，其它都为0，那么如果此时我们减1的话，则最高位会降一位，其余为0的位现在都为变为1，那么我们把两数相与，就会得到0，用这个性质也能来解题，而且只需一行代码就可以搞定。

【CODE】return (n > 0) && (!(n & (n - 1)));

# 234. Palindrome Linked List回文链表

判断一个链表是否为回文链表。能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决？

【分析】链表比字符串难的地方就在于不能通过坐标来直接访问，而只能从头开始遍历到某个位置。那么根据回文串的特点，我们需要比较对应位置的值是否相等，那么我们首先需要找到链表的中点，这个可以用快慢指针来实现。

我们还需要用栈，每次慢指针走一步，都把值存入栈中，等到达中点时，链表的前半段都存入栈中了，由于栈的后进先出的性质，就可以和后半段链表按照回文对应的顺序比较了。但是栈结构不满足O(1)的空间复杂度。

那么在找到中点后，将后半段的链表翻转一下，这样就可以按照回文的顺序比较。

【CODE】

class Solution {

public:

bool isPalindrome(ListNode\* head) {

if (!head || !head->next) return true;

ListNode \*slow = head, \*fast = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

ListNode \*last = slow->next, \*pre = head;

while (last->next) {

ListNode \*tmp = last->next;

last->next = tmp->next;

tmp->next = slow->next;

slow->next = tmp;

}

# 237. Delete Node in a Linked List删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为: 4 -> 5 -> 1 -> 9

|  |
| --- |
| 输入: head = [4,5,1,9], node = 5  输出: [4,1,9]  解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9. |
| 输入: head = [4,5,1,9], node = 1  输出: [4,5,9]  解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9. |

说明：链表至少包含两个节点。链表中所有节点的值都是唯一的。给定的节点为非末尾节点并且一定是链表中的一个有效节点。不要从你的函数中返回任何结果。

【分析】删除链表的一个节点，没有给我们链表的起点，只给我们了一个要删的节点，我们之前要删除一个节点的方法是要有其前一个节点的位置，然后将其前一个节点的next连向要删节点的下一个，然后delete掉要删的节点即可。这道题的处理方法是先把当前节点的值用下一个节点的值覆盖了，然后我们删除下一个节点即可。

【CODE】class Solution {

public:

void deleteNode(ListNode\* node) {

node->val = node->next->val;

ListNode \*tmp = node->next;

node->next = tmp->next;

delete tmp;

}

};

# 206. Reverse Linked List反转链表

反转一个单链表。

输入: 1->2->3->4->5->NULL 输出: 5->4->3->2->1->NULL

【分析】迭代的解法，思路是在原链表之前建立一个dummy node，因为首节点会变，然后从head开始，将之后的一个节点移到dummy node之后，重复此操作知道head成为末节点为止。

【CODE】// Iterative

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (!head) return head;

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*cur = head;

while (cur->next) {

ListNode \*tmp = cur->next;

cur->next = tmp->next;

tmp->next = dummy->next;

dummy->next = tmp;

}

return dummy->next;

}

};

class Solution { //简洁写法

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* pre = NULL;

ListNode\* cur = head;

while (cur != NULL) {

ListNode\* next = cur->next; // forward

cur->next = pre; // reverse

pre = cur; // forward

cur = next; // forward

}

return pre; // cur == NULL;

}

};

【分析】递归解法，代码量更少。思路是不断的进入递归函数，直到head指向最后一个节点，p指向之前一个节点，然后调换head和p的位置，再返回上一层递归函数，再交换p和head的位置，每次交换后，head节点后面都是交换好的顺序，直到p为首节点，然后再交换，首节点就成了为节点，此时整个链表也完成了翻转。

【CODE】// Recursive

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*p = head;

head = reverseList(p->next);

p->next->next = p;

p->next = NULL;

return head;

}

};

# 21. Merge Two Sorted Lists合并两个有序链表

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

输入：1->2->4, 1->3->4 输出：1->1->2->3->4->4

【分析】具体思想是新建一个链表，然后比较两个链表中的元素值，把较小的那个链到新链表中，由于两个输入链表的长度可能不同，所以最终会有一个链表先完成插入所有元素，则直接另一个未完成的链表直接链入新链表的末尾。

【CODE】

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

ListNode \*dummy = new ListNode(-1), \*cur = dummy;

while (l1 && l2) {

if (l1->val < l2->val) {

cur->next = l1;

l1 = l1->next;

} else {

cur->next = l2;

l2 = l2->next;

}

cur = cur->next;

}

cur->next = l1 ? l1 : l2;

return dummy->next;

}

};

【分析】递归写法，当某个链表为空了，就返回另一个。然后核心还是比较当前两个节点值大小，如果l1的小，那么对于l1的下一个节点和l2调用递归函数，将返回值赋值给l1.next，然后返回l1；否则就对于l2的下一个节点和l1调用递归函数，将返回值赋值给l2.next，然后返回l2。

【CODE】

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

if (!l1 || (l2 && l1->val > l2->val)) swap(l1, l2);

if (l1) l1->next = mergeTwoLists(l1->next, l2);

return l1;

}

};

# 23. Merge k Sorted Lists合并K个排序链表

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

【分析】不管合并几个，基本还是要两两合并。那么我们首先考虑的方法是能不能利用上题的解法来解答此题。答案是肯定的，但是需要修改，怎么修改，最先想到的就是两两合并，就是前两个先合并，合并好了再跟第三个，然后第四个直到第k个。这样的思路是对的，但是效率不高，没法通过OJ，所以我们只能换一种思路，这里就需要用到分治法 Divide and Conquer Approach。简单来说就是不停的对半划分，比如k个链表先划分为合并两个k/2个链表的任务，再不停的往下划分，直到划分成只有一个或两个链表的任务，开始合并。举个例子来说比如合并6个链表，那么按照分治法，我们首先分别合并1和4,2和5,3和6。这样下一次只需合并3个链表，我们再合并1和3，最后和2合并就可以了。

【CODE】class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

if (lists.size() == 0) return NULL;

int n = lists.size();

while (n > 1) {

int k = (n + 1) / 2;

for (int i = 0; i < n / 2; ++i) {

lists[i] = mergeTwoLists(lists[i], lists[i + k]);

}

n = k;

}

return lists[0];

}

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode \*head = new ListNode(-1);

ListNode \*cur = head;

while (l1 && l2) {

if (l1->val < l2->val) {

cur->next = l1;

l1 = l1->next;

} else {

cur->next = l2;

l2 = l2->next;

}

cur = cur->next;

}

if (l1) cur->next = l1;

if (l2) cur->next = l2;

return head->next;

}

};

【分析】另一种解法，这种解法利用了最小堆这种数据结构，我们首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后我们每次取出最小的那个元素加入我们最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可。

【CODE】struct cmp {

bool operator () (ListNode \*a, ListNode \*b) {

return a->val > b->val;

}

};

class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

priority\_queue<ListNode\*, vector<ListNode\*>, cmp> q;

for (int i = 0; i < lists.size(); ++i) {

if (lists[i]) q.push(lists[i]);

}

ListNode \*head = NULL, \*pre = NULL, \*tmp = NULL;

while (!q.empty()) {

tmp = q.top();

q.pop();

if (!pre) head = tmp;

else pre->next = tmp;

pre = tmp;

if (tmp->next) q.push(tmp->next);

}

return head;

}

};

# 19. Remove Nth Node From End of List删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

说明：给定的 n 保证是有效的。

进阶：你能尝试使用一趟扫描实现吗？

【分析】移除链表倒数第N个节点，限定n一定是有效的，即n不会大于链表中的元素总数。还要求一次遍历解决问题，那么就得想些比较巧妙的方法了。比如我们首先要考虑的时，如何找到倒数第N个节点，由于只允许一次遍历，所以我们不能用一次完整的遍历来统计链表中元素的个数，而是遍历到对应位置就应该移除了。那么我们需要用两个指针来帮助我们解题，pre和cur指针。首先cur指针先向前走N步，如果此时cur指向空，说明N为链表的长度，则需要移除的为首元素，那么此时我们返回head->next即可，如果cur存在，我们再继续往下走，此时pre指针也跟着走，直到cur为最后一个元素时停止，此时pre指向要移除元素的前一个元素，我们再修改指针跳过需要移除的元素即可。

【CODE】

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

if (!head->next) return NULL;

ListNode \*pre = head, \*cur = head;

for (int i = 0; i < n; ++i) cur = cur->next;

if (!cur) return head->next;

while (cur->next) {

cur = cur->next;

pre = pre->next;

}

pre->next = pre->next->next;

return head;

}

};

# 24. Swap Nodes in Pairs两两交换链表中的节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

示例:给定 1->2->3->4, 你应该返回 2->1->4->3.

说明:你的算法只能使用常数的额外空间。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】基本的链表操作题，我们可以分别用递归和迭代来实现。对于迭代实现，还是需要建立dummy节点，注意在连接节点的时候，最好画个图，以免把自己搞晕。

【CODE】

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

ListNode \*dummy = new ListNode(-1), \*pre = dummy;

dummy->next = head;

while (pre->next && pre->next->next) {

ListNode \*t = pre->next->next;

pre->next->next = t->next;

t->next = pre->next;

pre->next = t;

pre = t->next;

}

return dummy->next;

}

};

【分析】递归的写法更简洁，实际上利用了回溯的思想，递归遍历到链表末尾，然后先交换末尾两个，然后依次往前交换。

【CODE】

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*t = head->next;

head->next = swapPairs(head->next->next);

t->next = head;

return t;

}

};

# 25. Reverse Nodes in k-Group k个一组翻转链表

给出一个链表，每 k 个节点一组进行翻转，并返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么将最后剩余节点保持原有顺序。

给定这个链表：1->2->3->4->5

当 k = 2 时，应当返回: 2->1->4->3->5

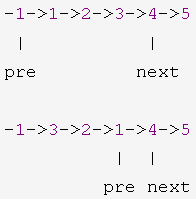
当 k = 3 时，应当返回: 3->2->1->4->5

说明 :你的算法只能使用常数的额外空间。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】以每k个为一组来翻转链表，实际上是把原链表分成若干小段，然后分别对其进行翻转，那么肯定总共需要两个函数，一个是用来分段的，一个是用来翻转的，我们就以题目中给的例子来看，对于给定链表1->2->3->4->5，一般在处理链表问题时，我们大多时候都会在开头再加一个dummy node，因为翻转链表时头结点可能会变化，为了记录当前最新的头结点的位置而引入的dummy node，那么我们加入dummy node后的链表变为-1->1->2->3->4->5，如果k为3的话，我们的目标是将1,2,3翻转一下，那么我们需要一些指针，pre和next分别指向要翻转的链表的前后的位置，然后翻转后pre的位置更新到如下新的位置。以此类推，只要next走过k个节点，就可以调用翻转函数来进行局部翻转了。

【CODE】

class Solution {

public:

ListNode \*reverseKGroup(ListNode \*head, int k) {

if (!head || k == 1) return head;

ListNode \*dummy = new ListNode(-1);

ListNode \*pre = dummy, \*cur = head;

dummy->next = head;

int i = 0;

while (cur) {

++i;

if (i % k == 0) {

pre = reverseOneGroup(pre, cur->next);

cur = pre->next;

} else {

cur = cur->next;

}

}

return dummy->next;

}

ListNode \*reverseOneGroup(ListNode \*pre, ListNode \*next) {

ListNode \*last = pre->next;

ListNode \*cur = last->next;

while(cur != next) {

last->next = cur->next;

cur->next = pre->next;

pre->next = cur;

cur = last->next;

}

return last;

}

};

【分析】也可以在一个函数中完成，我们首先遍历整个链表，统计出链表的长度，然后如果长度大于等于k，我们开始交换节点，当k=2时，每段我们只需要交换一次，当k=3时，每段需要交换2此，所以i从1开始循环，注意交换一段后更新pre指针，然后num自减k，直到num<k时循环结束。

【CODE】class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode \*dummy = new ListNode(-1), \*pre = dummy, \*cur = pre;

dummy->next = head;

int num = 0;

while (cur = cur->next) ++num;

while (num >= k) {

cur = pre->next;

for (int i = 1; i < k; ++i) {

ListNode \*t = cur->next;

cur->next = t->next;

t->next = pre->next;

pre->next = t;

}

pre = cur;

num -= k;

}

return dummy->next;

}

};

【分析】使用递归来做，我们用head记录每段的开始位置，cur记录结束位置的下一个节点，然后我们调用reverse函数来将这段翻转，然后得到一个new\_head，原来的head就变成了末尾，这时候后面接上递归调用下一段得到的新节点，返回new\_head即可。

【CODE】class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode \*cur = head;

for (int i = 0; i < k; ++i) {

if (!cur) return head;

cur = cur->next;

}

ListNode \*new\_head = reverse(head, cur);

head->next = reverseKGroup(cur, k);

return new\_head;

}

ListNode\* reverse(ListNode\* head, ListNode\* tail) {

ListNode \*pre = tail;

while (head != tail) {

ListNode \*t = head->next;

head->next = pre;

pre = head;

head = t;

}

return pre;

}

};

# 61. Rotate List旋转链表

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

|  |  |
| --- | --- |
| 输入: 1->2->3->4->5->NULL, k = 2  输出: 4->5->1->2->3->NULL  解释:  向右旋转 1 步: 5->1->2->3->4->NULL  向右旋转 2 步: 4->5->1->2->3->NULL | 输入: 0->1->2->NULL, k = 4  输出: 2->0->1->NULL  解释:  向右旋转 1 步: 2->0->1->NULL  向右旋转 2 步: 1->2->0->NULL  向右旋转 3 步: 0->1->2->NULL  向右旋转 4 步: 2->0->1->NULL |

【分析】先遍历整个链表获得链表长度n，然后此时把链表头和尾链接起来，在往后走n - k % n个节点就到达新链表的头结点前一个点，这时断开链表即可。（还有一种利用快慢指针的做法，需要考虑空链表和k>n的情况。）

【CODE】class Solution {

public:

ListNode \*rotateRight(ListNode \*head, int k) {

if (!head) return NULL;

int n = 1;

ListNode \*cur = head;

while (cur->next) {

++n;

cur = cur->next;

}

cur->next = head;

int m = n - k % n;

for (int i = 0; i < m; ++i) {

cur = cur->next;

}

ListNode \*newhead = cur->next;

cur->next = NULL;

return newhead;

}

};

# 82. Remove Duplicates from Sorted List II删除排序链表中的重复元素 II

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 *没有重复出现* 的数字。

|  |  |
| --- | --- |
| 输入: 1->2->3->3->4->4->5  输出: 1->2->5 | 输入: 1->1->1->2->3  输出: 2->3 |

【分析】要删掉所有的重复项，由于链表开头可能会有重复项，被删掉的话头指针会改变，而最终却还需要返回链表的头指针。所以需要定义一个新的节点，然后链上原链表，然后定义一个前驱指针和一个现指针，每当前驱指针指向新建的节点，现指针从下一个位置开始往下遍历，遇到相同的则继续往下，直到遇到不同项时，把前驱指针的next指向下面那个不同的元素。如果现指针遍历的第一个元素就不相同，则把前驱指针向下移一位。

【CODE】

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (!head || !head->next) return head;

ListNode \*start = new ListNode(0);

start->next = head;

ListNode \*pre = start;

while (pre->next) {

ListNode \*cur = pre->next;

while (cur->next && cur->next->val == cur->val) cur = cur->next;

if (cur != pre->next) pre->next = cur->next;

else pre = pre->next;

}

return start->next;

}

};

# 86. Partition List分隔链表

给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。你应当保留两个分区中每个节点的初始相对位置。

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

【分析】首先找到第一个大于或等于给定值的节点，用题目中给的例子来说就是先找到4，然后再找小于3的值，每找到一个就将其取出置于4之前即可。

还有一种解法，就是将所有小于给定值的节点取出组成一个新的链表，此时原链表中剩余的节点的值都大于或等于给定值，只要将原链表直接接在新链表后即可。

【CODE】方法1

class Solution {

public:

ListNode \*partition(ListNode \*head, int x) {

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*pre = dummy, \*cur = head;;

while (pre->next && pre->next->val < x) pre = pre->next;

cur = pre;

while (cur->next) {

if (cur->next->val < x) {

ListNode \*tmp = cur->next;

cur->next = tmp->next;

tmp->next = pre->next;

pre->next = tmp;

pre = pre->next;

} else {

cur = cur->next;

}

}

return dummy->next;

}

};

# 92. Reverse Linked List II反转链表 II

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

说明:1 ≤ m ≤ n ≤ 链表长度。

输入: 1->2->3->4->5->NULL, m = 2, n = 4

输出: 1->4->3->2->5->NULL

【分析】对于链表的问题，根据以往的经验一般都是要建一个dummy node，连上原链表的头结点，这样的话就算头结点变动了，我们还可以通过dummy->next来获得新链表的头结点。这道题的要求是只通过一次遍历完成，就拿题目中的例子来说，变换的是2,3,4这三个点，那么我们可以先取出2，用front指针指向2，然后当取出3的时候，我们把3加到2的前面，把front指针前移到3，依次类推，到4后停止，这样我们得到一个新链表4->3->2, front指针指向4。对于原链表连说，有两个点的位置很重要，需要用指针记录下来，分别是1和5，因为当2,3,4被取走时，原链表就变成了1->5->NULL，要把新链表插入的时候需要这两个点的位置。1的位置很好找，因为知道m的值，我们用pre指针记录1的位置，5的位置最后才能记录，当4结点被取走后，5的位置需要记下来，这样我们就可以把倒置后的那一小段链表加入到原链表中。

【CODE】class Solution {

public:

ListNode \*reverseBetween(ListNode \*head, int m, int n) {

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*cur = dummy;

ListNode \*pre, \*front, \*last;

for (int i = 1; i <= m - 1; ++i) cur = cur->next;

pre = cur;

last = cur->next;

for (int i = m; i <= n; ++i) {

cur = pre->next;

pre->next = cur->next;

cur->next = front;

front = cur;

}

cur = pre->next;

pre->next = front;

last->next = cur;

return dummy->next;

}

};

# 109. Convert Sorted List to Binary Search Tree有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

【分析】数组方便就方便在可以通过index直接访问任意一个元素，而链表不行。由于二分查找法每次需要找到中点，而链表的查找中间点可以通过快慢指针来操作，可参见之前的Reorder List 链表重排序和Linked List Cycle II 单链表中的环二有关快慢指针的应用。找到中点后，要以中点的值建立一个数的根节点，然后需要把原链表断开，分为前后两个链表，都不能包含原中节点，然后再分别对这两个链表递归调用原函数，分别连上左右子节点即可。

【CODE】

class Solution {

public:

TreeNode \*sortedListToBST(ListNode \*head) {

if (!head) return NULL;

if (!head->next) return new TreeNode(head->val);

ListNode \*slow = head;

ListNode \*fast = head;

ListNode \*last = slow;

while (fast->next && fast->next->next) {

last = slow;

slow = slow->next;

fast = fast->next->next;

}

fast = slow->next;

last->next = NULL;

TreeNode \*cur = new TreeNode(slow->val);

if (head != slow) cur->left = sortedListToBST(head);

cur->right = sortedListToBST(fast);

return cur;

}

};

# 138. Copy List with Random Pointer复制带随机指针的链表

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度拷贝。

【分析】链表的深度拷贝题的难点就在于如何处理随机指针的问题，由于每一个节点都有一个随机指针，这个指针可以为空，也可以指向链表的任意一个节点，如果我们在每生成一个新节点给其随机指针赋值时，都要去遍历原链表的话，OJ上肯定会超时，所以我们可以考虑用Hash map来缩短查找时间，第一遍遍历生成所有新节点时同时建立一个原节点和新节点的哈希表，第二遍给随机指针赋值时，查找时间是常数级。

【CODE】

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if (!head) return NULL;

RandomListNode \*res = new RandomListNode(head->label);

RandomListNode \*node = res;

RandomListNode \*cur = head->next;

map<RandomListNode\*, RandomListNode\*> m;

m[head] = res;

while (cur) {

RandomListNode \*tmp = new RandomListNode(cur->label);

node->next = tmp;

m[cur] = tmp;

node = node->next;

cur = cur->next;

}

node = res;

cur = head;

while (node) {

node->random = m[cur->random];

node = node->next;

cur = cur->next;

}

return res;

}

};

【分析】如果使用哈希表占用额外的空间，如果这道题限制了空间的话，就要考虑别的方法。下面这个方法很巧妙分以下三个步骤：

1. 在原链表的每个节点后面拷贝出一个新的节点

2. 依次给新的节点的随机指针赋值，

而且这个赋值非常容易 cur->next->random = cur->random->next

3. 断开链表可得到深度拷贝后的新链表

【CODE】

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if (!head) return NULL;

RandomListNode \*cur = head;

while (cur) {

RandomListNode \*node = new RandomListNode(cur->label);

node->next = cur->next;

cur->next = node;

cur = node->next;

}

cur = head;

while (cur) {

if (cur->random) {

cur->next->random = cur->random->next;

}

cur = cur->next->next;

}

cur = head;

RandomListNode \*res = head->next;

while (cur) {

RandomListNode \*tmp = cur->next;

cur->next = tmp->next;

if(tmp->next) tmp->next = tmp->next->next;

cur = cur->next;

}

return res;

}

};

# 142. Linked List Cycle II环形链表 II

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。

说明：不允许修改给定的链表。

进阶：你是否可以不用额外空间解决此题？

【分析】设快慢指针，不过这次要记录两个指针相遇的位置，当两个指针相遇了后，让其一指针从链表头开始，一步两步，此时再相遇的位置就是链表中环的起始位置。

【CODE】class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

ListNode \*slow = head, \*fast = head;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) break;

}

if (!fast || !fast->next) return NULL;

slow = head;

while (slow != fast) {

slow = slow->next;

fast = fast->next;

}

return fast;

}

};

# 补.环链表的相关问题

1. 环的长度是多少？

2. 如何找到环中第一个节点（即Linked List Cycle II）？

3. 如何将有环的链表变成单链表（解除环）？

4. 如何判断两个单链表是否有交点？如何找到第一个相交的节点？

设：链表头是X，环的第一个节点是Y，slow和fast第一次的交点是Z。各段的长度分别是a,b,c，如图所示。环的长度是L。slow和fast的速度分别是qs,qf。

1. 方法一（网上都是这个答案）：

第一次相遇后，让slow,fast继续走，记录到下次相遇时循环了几次。因为当fast第二次到达Z点时，fast走了一圈，slow走了半圈，而当fast第三次到达Z点时，fast走了两圈，slow走了一圈，正好还在Z点相遇。

方法二：

第一次相遇后，让fast停着不走了，slow继续走，记录到下次相遇时循环了几次。

方法三（最简单）：

第一次相遇时slow走过的距离：a+b，fast走过的距离：a+b+c+b。

因为fast的速度是slow的两倍，所以fast走的距离是slow的两倍，有 2(a+b) = a+b+c+b，可以得到a=c（这个结论很重要！）。

L=b+c=a+b，也就是说，从一开始到二者第一次相遇，循环的次数就等于环的长度。

2. 我们已经得到了结论a=c，那么让两个指针分别从X和Z开始走，每次走一步，那么正好会在Y相遇！也就是环的第一个节点。

3. 在上一个问题的最后，将c段中Y点之前的那个节点与Y的链接切断即可。

4. 如何判断两个单链表是否有交点？先判断两个链表是否有环，如果一个有环一个没环，肯定不相交；如果两个都没有环，判断两个列表的尾部是否相等；如果两个都有环，判断一个链表上的Z点是否在另一个链表上。

如何找到第一个相交的节点？求出两个链表的长度L1,L2（如果有环，则将Y点当做尾节点来算），假设L1<L2，用两个指针分别从两个链表的头部开始走，长度为L2的链表先走(L2-L1)步，然后两个一起走，直到二者相遇。

# 143. Reorder List重排链表

给定一个单链表 L：L0→L1→…→Ln-1→Ln ，

将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】这道链表重排序问题可以拆分为以下三个小问题：

1. 使用快慢指针来找到链表的中点，并将链表从中点处断开，形成两个独立的链表。

2. 将第二个链翻转。

3. 将第二个链表的元素间隔地插入第一个链表中。

【CODE】

class Solution {

public:

void reorderList(ListNode \*head) {

if (!head || !head->next || !head->next->next) return;

ListNode \*fast = head;

ListNode \*slow = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

ListNode \*mid = slow->next;

slow->next = NULL;

ListNode \*last = mid;

ListNode \*pre = NULL;

while (last) {

ListNode \*next = last->next;

last->next = pre;

pre = last;

last = next;

}

while (head && pre) {

ListNode \*next = head->next;

head->next = pre;

pre = pre->next;

head->next->next = next;

head = next;

}

}

};

# 147. Insertion Sort List对链表进行插入排序

插入排序算法：

1.插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。

2.每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。

3.重复直到所有输入数据插入完为止。

【分析】链表的插入排序实现原理很简单，就是一个元素一个元素的从原链表中取出来，然后按顺序插入到新链表中，时间复杂度为O(n2)，是一种效率并不是很高的算法，但是空间复杂度为O(1)，以高时间复杂度换取了低空间复杂度

【CODE】

class Solution {

public:

ListNode\* insertionSortList(ListNode\* head) {

ListNode \*dummy = new ListNode(-1), \*cur = dummy;

while (head) {

ListNode \*t = head->next;

cur = dummy;

while (cur->next && cur->next->val <= head->val) {

cur = cur->next;

}

head->next = cur->next;

cur->next = head;

head = t;

}

return dummy->next;

}

};

# 148. Sort List排序链表

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

【分析】常见排序方法有很多，插入排序，选择排序，堆排序，快速排序，冒泡排序，归并排序，桶排序等等。。它们的时间复杂度不尽相同，而这里题目限定了时间必须为O(nlgn)，符合要求只有快速排序，归并排序，堆排序，而根据单链表的特点，最适于用归并排序。

【CODE】

class Solution {

public:

ListNode\* sortList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*slow = head, \*fast = head, \*pre = head;

while (fast && fast->next) {

pre = slow;

slow = slow->next;

fast = fast->next->next;

}

pre->next = NULL;

return merge(sortList(head), sortList(slow));

}

ListNode\* merge(ListNode\* l1, ListNode\* l2) {

if (!l1) return l2;

if (!l2) return l1;

if (l1->val < l2->val) {

l1->next = merge(l1->next, l2);

return l1;

} else {

l2->next = merge(l1, l2->next);

return l2;

}

}

};

# 369. Plus One Linked List 链表加一运算

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example: Input:1->2->3 Output:1->2->4

【分析】给了我们一个链表，用来模拟一个三位数，表头是高位，现在让我们进行加1运算，这道题的难点在于链表无法通过坐标来访问元素，只能通过遍历的方式进行，而这题刚好让我们从链尾开始操作，从后往前，遇到进位也要正确的处理，最后还有可能要在开头补上一位。那么我们反过来想，如果链尾是高位，那么进行加1运算就方便多了，直接就可以边遍历边进行运算处理，那么我们可以做的就是先把链表翻转一下，然后现在就是链尾是高位了，我们进行加1处理运算结束后，再把链表翻转回来即可。

【CODE】

class Solution {

public:

ListNode\* plusOne(ListNode\* head) {

if (!head) return head;

ListNode \*rev\_head = reverse(head), \*cur = rev\_head, \*pre = cur;

int carry = 1;

while (cur) {

pre = cur;

int t = cur->val + carry;

cur->val = t % 10;

carry = t / 10;

if (carry == 0) break;

cur = cur->next;

}

if (carry) pre->next = new ListNode(1);

return reverse(rev\_head);

}

ListNode\* reverse(ListNode \*head) {

if (!head) return head;

ListNode \*dummy = new ListNode(-1), \*cur = head;

dummy->next = head;

while (cur->next) {

ListNode \*t = cur->next;

cur->next = t->next;

t->next = dummy->next;

dummy->next = t;

}

return dummy->next;

}

};

【分析】可以通过递归来实现，这样我们就不用翻转链表了，通过递归一层一层的调用，最先处理的是链尾元素，我们将其加1，然后看是否有进位，返回进位，然后回溯到表头，加完进位，如果发现又产生了新的进位，那么我们在最开头加上一个新节点即可。

【CODE】class Solution {

public:

ListNode\* plusOne(ListNode\* head) {

if (!head) return head;

int carry = helper(head);

if (carry == 1) {

ListNode \*res = new ListNode(1);

res->next = head;

return res;

}

return head;

}

int helper(ListNode \*node) {

if (!node) return 1;

int carry = helper(node->next);

int sum = node->val + carry;

node->val = sum % 10;

return sum / 10;

}

};

# 445. Add Two Numbers II两数相加 II

给定两个非空链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储单个数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶:如果输入链表不能修改该如何处理？即你不能对列表中的节点进行翻转。

输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4) 输出: 7 -> 8 -> 0 -> 7

【分析】我们可以看到这道题的最高位在链表首位置，如果我们给链表翻转一下的话就跟之前的题目一样了，这里我们来看不修改链表顺序的方法。由于加法需要从最低位开始运算，而最低位在链表末尾，链表只能从前往后遍历，没法取到前面的元素，那怎么办呢？我们可以利用栈来保存所有的元素，然后利用栈的后进先出的特点就可以从后往前取数字了，我们首先遍历两个链表，将所有数字分别压入两个栈s1和s2中，我们建立一个值为0的res节点，然后开始循环，如果栈不为空，则将栈顶数字加入sum中，然后将res节点值赋为sum%10，然后新建一个进位节点head，赋值为sum/10，如果没有进位，那么就是0，然后我们head后面连上res，将res指向head，这样循环退出后，我们只要看res的值是否为0，为0返回res->next，不为0则返回res即可。

【CODE】

class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

stack<int> s1, s2;

while (l1) {

s1.push(l1->val);

l1 = l1->next;

}

while (l2) {

s2.push(l2->val);

l2 = l2->next;

}

int sum = 0;

ListNode \*res = new ListNode(0);

while (!s1.empty() || !s2.empty()) {

if (!s1.empty()) {sum += s1.top(); s1.pop();}

if (!s2.empty()) {sum += s2.top(); s2.pop();}

res->val = sum % 10;

ListNode \*head = new ListNode(sum / 10);

head->next = res;

res = head;

sum /= 10;

}

return res->val == 0 ? res->next : res;

}

};

# 328. Odd Even Linked List奇偶链表

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

|  |  |
| --- | --- |
| 输入: 1->2->3->4->5->NULL  输出: 1->3->5->2->4->NULL | 输入: 2->1->3->5->6->4->7->NULL  输出: 2->3->6->7->1->5->4->NULL |

【分析】pre指向奇节点，cur指向偶节点，然后把偶节点cur后面的那个奇节点提前到pre的后面，然后pre和cur各自前进一步，此时cur又指向偶节点，pre指向当前奇节点的末尾，以此类推直至把所有的偶节点都提前了即可。

【CODE】比较好

class Solution {

public:

ListNode\* oddEvenList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*pre = head, \*cur = head->next;

while (cur && cur->next) {

ListNode \*tmp = pre->next;

pre->next = cur->next;

cur->next = cur->next->next;

pre->next->next = tmp;

cur = cur->next;

pre = pre->next;

}

return head;

}

};

【分析】用两个奇偶指针分别指向奇偶节点的起始位置，另外需要一个单独的指针even\_head来保存偶节点的起点位置，然后把奇节点的指向偶节点的下一个(一定是奇节点)，此奇节点后移一步，再把偶节点指向下一个奇节点的下一个(一定是偶节点)，此偶节点后移一步，以此类推直至末尾，此时把分开的偶节点的链表连在奇节点的链表后即可。

【CODE】

class Solution {

public:

ListNode\* oddEvenList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*odd = head, \*even = head->next, \*even\_head = even;

while (even && even->next) {

odd = odd->next = even->next;

even = even->next = odd->next;

}

odd->next = even\_head;

return head;

}

};

# 78. Subsets子集

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

【分析】把数组中所有的数分配一个状态，true表示这个数在子集中出现，false表示在子集中不出现，那么对于一个长度为n的数组，每个数字都有出现与不出现两种情况，所以共有2n中情况，那么我们把每种情况都转换出来就是子集了，用题目中[1 2 3]数组为例，数组共有8个子集，每个子集的序号的二进制表示，把是1的位对应原数组中的数字取出来就是一个子集，八种情况都取出来就是所有的子集了。

【CODE】class Solution {

public:

vector<vector<int> > subsets(vector<int> &S) {

vector<vector<int> > res;

sort(S.begin(), S.end());

int max = 1 << S.size();

for (int k = 0; k < max; ++k) {

vector<int> out = convertIntToSet(S, k);

res.push\_back(out);

}

return res;

}

vector<int> convertIntToSet(vector<int> &S, int k) {

vector<int> sub;

int idx = 0;

for (int i = k; i > 0; i >>= 1) {

if ((i & 1) == 1) {

sub.push\_back(S[idx]);

}

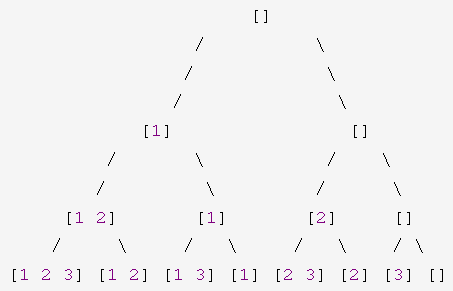
++idx;

}

return sub;

}

};

【分析】递归的解法，相当于一种深度优先搜索，由于原集合每一个数字只有两种状态，要么存在，要么不存在，那么在构造子集时就有选择和不选择两种情况，所以可以构造一棵二叉树，左子树表示选择该层处理的节点，右子树表示不选择，最终的叶节点就是所有子集合，树的结构如图：

【CODE】class Solution {

public:

vector<vector<int> > subsets(vector<int> &S) {

vector<vector<int> > res;

vector<int> out;

sort(S.begin(), S.end());

getSubsets(S, 0, out, res);

return res;

}

void getSubsets(vector<int> &S, int pos, vector<int> &out, vector<vector<int> > &res) {

res.push\_back(out);

for (int i = pos; i < S.size(); ++i) {

out.push\_back(S[i]);

getSubsets(S, i + 1, out, res);

out.pop\_back();

}

}

};

# 137. Single Number II只出现一次的数字 II

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

【分析】利用计算机按位储存数字的特性来做，这道题就是除了一个单独的数字之外，数组中其他的数字都出现了三次，那么还是要利用位操作 Bit Operation 来解此题。我们可以建立一个**32位的数字**，来统计每一位上1出现的个数，我们知道如果某一位上为1的话，那么如果该整数出现了三次，对3去余为0，我们把每个数的对应位都加起来对3取余，最终剩下来的那个数就是单独的数字。

*但是一定要这么来记录数字出现的次数吗？*我们把数组中数字的每一位累加起来对3取余，剩下的结果就是那个单独数组该位上的数字，由于我们累加的过程都要对3取余，那么每一位上累加的过程就是0->1->2->0，换成二进制的表示为00->01->10->00，那么我们可以写出对应关系：

00 (+) 1 = 01

01 (+) 1 = 10

10 (+) 1 = 00 ( mod 3)

那么我们用ab来表示开始的状态，对于加1操作后，得到的新状态的ab的算法如下：

b = b xor r & ~a;

a = a xor r & ~b;

我们这里的ab就是上面的三种状态00，01，10的十位和各位，刚开始的时候，a和b都是0，当此时遇到数字1的时候，b更新为1，a更新为0，就是01的状态；再次遇到1的时候，b更新为0，a更新为1，就是10的状态；再次遇到1的时候，b更新为0，a更新为0，就是00的状态，相当于重置了；最后的结果保存在b中。

【CODE】class Solution {

public:

int singleNumber(vector<int>& nums) {

int a = 0, b = 0;

for (int i = 0; i < nums.size(); ++i) {

b = (b ^ nums[i]) & ~a;

a = (a ^ nums[i]) & ~b;

}

return b;

}

};

# 260. Single Number III只出现一次的数字 III

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

注意： 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现？

【分析】这道题其实是很巧妙的利用了Single Number 单独的数字的解法，因为那道解法是可以准确的找出只出现了一次的数字，但前提是其他数字必须出现两次才行。而这题有两个数字都只出现了一次，那么我们如果能想办法把原数组分为两个小数组，不相同的两个数字分别在两个小数组中，这样分别调用Single Number 单独的数字的解法就可以得到答案。那么如何实现呢，首先我们先把原数组全部异或起来，那么我们会得到一个数字，这个数字是两个不相同的数字异或的结果，我们取出其中任意一位为‘1’的位，为了方便起见，我们用 a &= -a 来取出最右端为‘1’的位，然后和原数组中的数字挨个相与，那么我们要求的两个不同的数字就被分到了两个小组中，分别将两个小组中的数字都异或起来，就可以得到最终结果。

【CODE】class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

int diff = accumulate(nums.begin(), nums.end(), 0, bit\_xor<int>());

diff &= -diff;

vector<int> res(2, 0);

for (auto &a : nums) {

if (a & diff) res[0] ^= a;

else res[1] ^= a;

}

return res;

}

};

# 187. Repeated DNA Sequences重复的DNA序列

所有 DNA 由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如：“ACGAATTCCG”。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找 DNA 分子中所有出现超多一次的10个字母长的序列（子串）。

【分析】用两位来表示一个字符，00表示A，01表示C，10表示G，11表示T，那么总共需要20位就可以表示10个字符流。为了提取出后20位，我们还需要用个mask，取值为0x3ffff，用此mask可取出后18位，再向左平移2位即可。算法的思想是，当取出第十个字符时，将其存在哈希表里，和该字符串出现频率映射，之后每向左移2位替换一个字符，查找新字符串在哈希表里出现次数，如果之前刚好出现过一次，则将当前字符串存入返回值的数组并将其出现次数加一，如果从未出现过，则将其映射到1。为了能更清楚的阐述整个过程，我们用题目中给的例子来分析整个过程：

首先我们取出前九个字符AAAAACCCC，根据上面的分析，我们用2位来表示一个字符，所以这九个字符可以用二进制表示为000000000001010101，然后我们继续遍历字符串，下一个进来的是C，则当前字符为AAAAACCCCC，二进制表示为00000000000101010101，然后我们将其存入哈希表中，用二进制的好处是可以用一个int变量来表示任意十个字符序列，比起直接存入字符串大大的节省了内存空间，然后再读入下一个字符C，则此时字符串为AAAACCCCCA，我们还是存入其二进制的表示形式，以此类推，当某个序列之前已经出现过了，我们将其存入结果res中即可.

【CODE】class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

unordered\_set<string> res;

unordered\_set<int> st;

unordered\_map<int, int> m{{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};

int cur = 0, i = 0;

while (i < 9) cur = cur << 2 | m[s[i++]];

while (i < s.size()) {

cur = ((cur & 0x3ffff) << 2) | (m[s[i++]]);

if (st.count(cur)) res.insert(s.substr(i - 10, 10));

else st.insert(cur);

}

return vector<string>(res.begin(), res.end());

}

};

【分析】如果我们不需要考虑节省内存空间，那我们可以直接将10个字符组成字符串存入set中，那么也就不需要mask了。

【CODE】class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

set<string> res, st;

for (int i = 0; i + 9 < s.size(); ++i) {

string t = s.substr(i, 10);

if (st.count(t)) res.insert(t);

else st.insert(t);

}

return vector<string>{res.begin(), res.end()};

}

};

# 338. Counting BitsBit位计数

给定一个非负整数 num。 对于范围 0 ≤ i ≤ num 中的每个数字 i ，计算其二进制数中的1的数目并将它们作为数组返回。

示例：比如给定 num = 5 ，应该返回 [0,1,1,2,1,2].

进阶：给出时间复杂度为O(n \* sizeof(integer)) 的解答非常容易。 但是你可以在线性时间O(n)内用一次遍历做到吗？

要求算法的空间复杂度为O(n)。

你能进一步完善解法吗？ 在c ++或任何其他语言中不使用任何内置函数（如c++里的 \_\_builtin\_popcount）来执行此操作。

【分析】列写二进制数，找出规律。可以找到一个规律是，从1开始，遇到偶数时，其1的个数和该偶数除以2得到的数字的1的个数相同，遇到奇数时，其1的个数等于该奇数除以2得到的数字的1的个数再加1。

【CODE】

class Solution {

public:

vector<int> countBits(int num) {

vector<int> res{0};

for (int i = 1; i <= num; ++i) {

if (i % 2 == 0) res.push\_back(res[i / 2]);

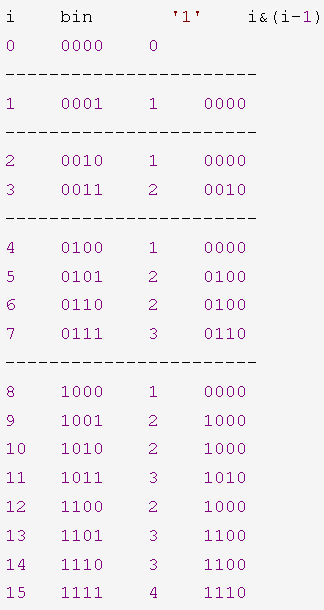
else res.push\_back(res[i / 2] + 1);

}

return res;

}

};

【分析】一种巧妙的规律，利用了i&(i - 1)， 这个本来是用来判断一个数是否是2的指数的快捷方法，比如8，二进制位1000, 那么8&(8-1)为0，只要为0就是2的指数, 那么我们观察一下0到15的数字和其对应的i&(i - 1)值，可以发现每个i值都是i&(i-1)对应的值加1。

【CODE】

class Solution {

public:

vector<int> countBits(int num) {

vector<int> res(num + 1, 0);

for (int i = 1; i <= num; ++i) {

res[i] = res[i & (i - 1)] + 1;

}

return res;

}

};

# 342. Power of Four 4的幂

给定一个整数 (32位有符整数型)，请写出一个函数来检验它是否是4的幂。

问题进阶：你能***不使用循环/递归***来解决这个问题吗？

【分析】判断一个数是否为4的次方数，那么最直接的方法就是不停的除以4，看最终结果是否为1。这就会利用到循环。

使用换底公式来做：

return num > 0 && int(log10(num) / log10(4)) - log10(num) / log10(4) == 0;

知道num & (num - 1)可以用来判断一个数是否为2的次方数，更进一步说，就是二进制表示下，只有最高位是1，那么由于是2的次方数，不一定是4的次方数，比如8，所以我们还要其他的限定条件，我们仔细观察可以发现，4的次方数的最高位的1都是计数位，那么我们只需与上一个数(0x55555555) <==> 1010101010101010101010101010101，如果得到的数还是其本身，则可以肯定其为4的次方数。

【CODE】class Solution {

public:

bool isPowerOfFour(int num) {

return num > 0 && !(num & (num - 1)) && (num & 0x55555555) == num;

}

};

【分析】或者我们在确定其是2的次方数了之后，发现只要是4的次方数，减1之后可以被3整除，所以可以写出代码。

【CODE】class Solution {

public:

bool isPowerOfFour(int num) {

return num > 0 && !(num & (num - 1)) && (num - 1) % 3 == 0;

}

};

# 371. Sum of Two Integers两整数之和

不使用运算符 + 和-，计算两整数a 、b之和。

【分析】用异或算不带进位的和，用与并左移1位来算进位，再把两者加起来即可。

【CODE】return b == 0 ? a : getSum(a ^ b, (a & b) << 1);

# 389. Find the Difference找不同

给定两个字符串 s 和 t，它们只包含小写字母。

字符串 t 由字符串 s 随机重排，然后在随机位置添加一个字母。

请找出在 t 中被添加的字母。

【分析】给我们两个字符串s和t，t是在s的任意一个地方加上了一个字符，让我们找出新加上的那个字符。首先第一反应的方法就是用哈希表来建立字符和个数之间的映射，如果在遍历t的时候某个映射值小于0了，那么返回该字符即可。

也可以使用位操作Bit Manipulation来做，利用异或的性质，相同位返回0，这样相同的字符都抵消了，剩下的就是后加的那个字符。

也可以直接用加和减，相同的字符一减一加也抵消了，剩下的就是后加的那个字符。

【CODE】

class Solution {

public:

char findTheDifference(string s, string t) {

char res = 0;

for (char c : s) res ^= c; // for (char c : s) res -= c;

for (char c : t) res ^= c; // for (char c : t) res += c;

return res;

}

};

【分析】利用了STL的accumulate函数，实际上是上面解法二的改写，一行就写完。

【知识点】STL的accumulate函数（#include <numeric>）

用来计算特定范围内（包括连续的部分和初始值）所有元素的和，除此之外，还可以用指定的二进制操作来计算特定范围内的元素结果。其头文件在numeric中。

accumulate原函数声明定义如下：

template<class InputIterator, class Type>

Type accumulate(

InputIterator \_First,

InputIterator \_Last,

Type \_Val

);

template<class InputIterator, class Type, class Fn2>

Type **accumulate**(

**InputIterator \_First**,

**InputIterator \_Last**,

**Type \_Val**,

**BinaryOperation \_Binary\_op**

);

参数说明：

|  |  |
| --- | --- |
| **\_First** | 指定范围内第一个迭代的值或者结合操作选项使用。 |
| **InputIterator \_Last** | 指定范围内最后一个迭代值或者结合操作项使用。 |
| **\_Val** | 要计算的初始值。 |
| **\_Binary\_op** | 运用于指定范围内所有元素和前面计算得到结果的参数。 |

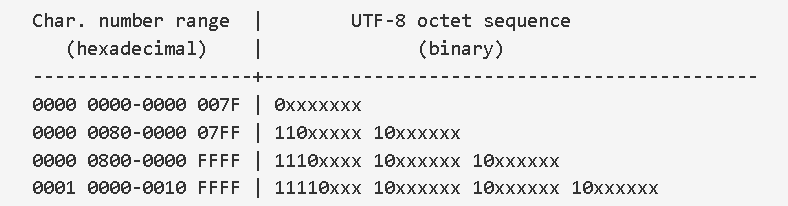
【CODE】 return accumulate(**begin(s)**, **end(s += t)**, **0**, **bit\_xor<int>()**);

# 393. UTF-8 Validation UTF-8 编码验证

UTF-8 中的一个字符可能的长度为 1 到 4 字节，遵循以下的规则：

对于 1 字节的字符，字节的第一位设为0，后面7位为这个符号的unicode码。

对于 n 字节的字符 (n > 1)，第一个字节的前 n 位都设为1，第 n+1 位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

这是 UTF-8 编码的工作方式：

给定一个表示数据的整数数组，返回它是否为有效的 utf-8 编码。

注意:输入是整数数组。只有每个整数的最低 8 个有效位用来存储数据。这意味着每个整数只表示 1 字节的数据。

【分析】UTF-8编码，这种互联网所采用的通用的编码格式的产生是为了解决ASCII只能表示英文字符的局限性，和统一Unicode的实现方式。摘自维基UTF-8编码：

对于UTF-8编码中的任意字节B，如果B的第一位为0，则B独立的表示一个字符(ASCII码)； (非ASCII字符)

如果B的第一位为1，第二位为0，则B为一个多字节字符中的一个字节；

如果B的前两位为1，第三位为0，则B为两个字节表示的字符中的第一个字节；

如果B的前三位为1，第四位为0，则B为三个字节表示的字符中的第一个字节；

如果B的前四位为1，第五位为0，则B为四个字节表示的字符中的第一个字节；

因此，对UTF-8编码中的任意字节，根据第一位，可判断是否为ASCII字符；根据前二位，可判断该字节是否为一个字符编码的第一个字节；根据前四位（如果前两位均为1），可确定该字节为字符编码的第一个字节，并且可判断对应的字符由几个字节表示；根据前五位（如果前四位为1），可判断编码是否有错误或数据传输过程中是否有错误。

那么根据上面的描述，我们可以先来判断第一位，如果是0的话，则说明是ASCII码，我们直接跳过，判断方法是只要比二进制数10000000小的数第一位肯定是0，然后我们来处理第一位是1的情况，由于第一位的1只是个标识符，后面连续跟的1的个数才是表示后面的字节的个数，我们可以统一从第一位开始连续1的个数，然后减去1就是后面的字节的个数，我想的办法是如果该数字大于等于128，则表示第一位是1，然后减去128，如果得到的数大于等于64，则表示第二位是1，依次类推就可以得到连续的个数，我们要注意10000000这个数是不合法的，遇到了直接返回false。我们得到了cnt的个数，只要验证后面的字节是否是以10开头的数即可，验证方法也很简单，只要这个数在10000000 ~ 10111111范围之间，则一定是10开头的。

【CODE】

class Solution {

public:

bool validUtf8(vector<int>& data) {

for (int i = 0; i < data.size(); ++i) {

if (data[i] < 0b10000000) {

continue;

} else {

int cnt = 0, val = data[i];

for (int j = 7; j >= 1; --j) {

if (val >= pow(2, j)) ++cnt;

else break;

val -= pow(2, j);

}

if (cnt == 1) return false;

for (int j = i + 1; j < i + cnt; ++j) {

if (data[j] > 0b10111111 || data[j] < 0b10000000) return false;

}

i += cnt - 1;

}

}

return true;

}

};

【分析】一种非常简洁的方法，也是要记连续1的个数，如果是标识字节，先将其向右平移五位，如果得到110，则说明后面跟了一个字节，否则向右平移四位，如果得到1110，则说明后面跟了两个字节，否则向右平移三位，如果得到11110，则说明后面跟了三个字节，否则向右平移七位，如果为1的话，说明是10000000这种情况，不能当标识字节，直接返回false。在非标识字节中，向右平移六位，如果得到的不是10，则说明不是以10开头的，直接返回false，否则cnt自减1，成功完成遍历返回true。

【CODE】

class Solution {

public:

bool validUtf8(vector<int>& data) {

int cnt = 0;

for (int d : data) {

if (cnt == 0) {

if ((d >> 5) == 0b110) cnt = 1;

else if ((d >> 4) == 0b1110) cnt = 2;

else if ((d >> 3) == 0b11110) cnt = 3;

else if (d >> 7) return false;

} else {

if ((d >> 6) != 0b10) return false;

--cnt;

}

}

return cnt == 0;

}

};

# 201. Bitwise AND of Numbers Range数字范围按位与

给定范围 [m, n]，其中 0 <= m <= n <= 2147483647，返回此范围内所有数字的按位与（包含 m, n 两端点）。

【分析】先从题目中给的例子来分析，[5, 7]里共有三个数字，分别写出它们的二进制为：101　　110　　111

相与后的结果为100，仔细观察我们可以得出，最后的数是该数字范围内所有的数的左边共同的部分，如果上面那个例子不太明显，我们再来看一个范围[26, 30]，它们的二进制如下：

11010　　11011　　11100　　11101　　11110

发现了规律后，我们只要写代码找到左边公共的部分即可，我们可以从建立一个32位都是1的mask，然后每次向左移一位，比较m和n是否相同，不同再继续左移一位，直至相同，然后把m和mask相与就是最终结果。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

int d = INT\_MAX;

while ((m & d) != (n & d)) {

d <<= 1;

}

return m & d;

}

};

【分析】另一种解法，不需要用mask，直接平移m和n，每次向右移一位，直到m和n相等，记录下所有平移的次数i，然后再把m左移i位即为最终结果。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

int i = 0;

while (m != n) {

m >>= 1;

n >>= 1;

++i;

}

return (m << i);

}

};

【分析】递归来做的，如果n大于m，那么就对m和n分别除以2，并且调用递归函数，对结果再乘以2，一定要乘回来，不然就不对了，就举一个最简单的例子，m = 10, n = 11，注意这里是二进制表示的，然后各自除以2，都变成了1，调用递归返回1，这时候要乘以2，才能变回10。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

return (n > m) ? (rangeBitwiseAnd(m / 2, n / 2) << 1) : m;

}

};

【分析】如果m小于n就进行循环，n与上n-1，举个简单的例子，110与上(110-1)，得到100，这不就相当于去掉最低位的1么，n就这样每次去掉最低位的1，如果小于等于m了，返回此时的n即可。

【CODE】

class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

while (m < n) n &= (n - 1);

return n;

}

};

# 268. Missing Number缺失数字

给定一个包含 0, 1, 2, ..., n 中 n 个数的序列，找出 0 .. n 中没有出现在序列中的那个数。说明:你的算法应具有线性时间复杂度。你能否仅使用额外常数空间来实现?

【分析】使用位操作Bit Manipulation来解，用到了异或操作的特性。思路是既然0到n之间少了一个数，我们将这个少了一个数的数组合0到n之间完整的数组异或一下，那么相同的数字都变为0了，剩下的就是少了的那个数字了。

【CODE】class Solution {

public:

int missingNumber(vector<int>& nums) {

int res = 0;

for (int i = 0; i < nums.size(); ++i) {

res ^= (i + 1) ^ nums[i];

}

return res;

}

};

# 318. Maximum Product of Word Lengths最大单词长度乘积

给定一个字符串数组 words，找到 length(word[i]) \* length(word[j]) 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

|  |
| --- |
| 输入: ["abcw","baz","foo","bar","xtfn","abcdef"]  输出: 16  解释: 这两个单词为 "abcw", "xtfn"。 |
| 输入: ["a","ab","abc","d","cd","bcd","abcd"]  输出: 4  解释: 这两个单词为 "ab", "cd"。 |
| 输入: ["a","aa","aaa","aaaa"]  输出: 0  解释: 不存在这样的两个单词。 |

【分析】给我们了一个单词数组，让我们求两个没有相同字母的单词的长度之积的最大值。我开始想的方法是每两个单词先比较，如果没有相同字母，则计算其长度之积，然后每次更新结果就能找到最大值。但是开始想的两个单词比较的方法是利用哈希表先将一个单词的所有出现的字母存入哈希表，然后检查另一个单词的各个字母是否在哈希表出现过，若都没出现过，则说明两个单词没有相同字母，则计算两个单词长度之积并更新结果。但是这种判断方法无法通过OJ的***大数据集***，大神们的解法，都是用了mask，因为题目中说都是**小写字母，那么只有26位，一个整型数int有32位，我们可以用后26位来对应26个字母，若为1，说明该对应位置的字母出现过，那么每个单词的都可由一个int数字表示**，**两个单词没有共同字母的条件是这两个int数想与为0**，用这个判断方法可以通过OJ。

【CODE】class Solution {

public:

int maxProduct(vector<string>& words) {

int res = 0;

vector<int> mask(words.size(), 0);

for (int i = 0; i < words.size(); ++i) {

for (char c : words[i]) {

mask[i] |= 1 << (c - 'a');

}

for (int j = 0; j < i; ++j) {

if (!(mask[i] & mask[j])) {

res = max(res, int(words[i].size() \* words[j].size()));

}

}

}

return res;

}

};

【分析】还有一种写法，借助哈希表，映射每个mask的值和其单词的长度，每算出一个单词的mask，遍历哈希表里的值，如果和其中的mask值相与为0，则将当前单词的长度和哈希表中存的单词长度相乘并更新结果。

【CODE】

class Solution {

public:

int maxProduct(vector<string>& words) {

int res = 0;

unordered\_map<int, int> m;

for (string word : words) {

int mask = 0;

for (char c : word) {

mask |= 1 << (c - 'a');

}

m[mask] = max(m[mask], int(word.size()));

for (auto a : m) {

if (!(mask & a.first)) {

res = max(res, (int)word.size() \* a.second);

}

}

}

return res;

}

};

# 397. Integer Replacement整数替换

给定一个正整数 n，你可以做如下操作：

1. 如果 n 是偶数，则用 n / 2替换 n。

2. 如果 n 是奇数，则可以用 n + 1或n - 1替换 n。

n 变为 1 所需的最小替换次数是多少？

【分析】看道题的要求，就会感觉用递归很合适，我们直接按照规则写出递归即可，注意由于有n+1的操作，所以当n为INT\_MAX的时候，就有可能**溢出**，所以我们可以先将n转为长整型，然后再进行运算。

【CODE】

class Solution {

public:

int integerReplacement(int n) {

if (n == 1) return 0;

if (n % 2 == 0) return 1 + integerReplacement(n / 2);

else {

long long t = n;

return 2 + min(integerReplacement((t + 1) / 2), integerReplacement((t - 1) / 2));

}

}

};

【分析】也可以使用迭代的解法，那么这里就有小技巧了，当n为奇数的时候，我们什么时候应该加1，什么时候应该减1呢，通过观察来说，除了3和7以外，所有加1就变成4的倍数的奇数，适合加1运算，比如15:

15 -> 16 -> 8 -> 4 -> 2 -> 1

15 -> 14 -> 7 -> 6 -> 3 -> 2 -> 1

对于7来说，加1和减1的结果相同，我们可以不用管，对于3来说，减1的步骤小，所以我们需要去掉这种情况。那么我们如何知道某个数字加1后是否是4的倍数呢，我们可以用个小技巧，由于我们之前判定其是奇数了，那么最右边一位肯定是1，如果其右边第二位也是1的话，那么进行加1运算，进位后右边肯定会出现两个0，则一定是4的倍数，搞定。如果之前判定是偶数，那么除以2即可。

【CODE】

class Solution {

public:

int integerReplacement(int n) {

long long t = n;

int cnt = 0;

while (t > 1) {

++cnt;

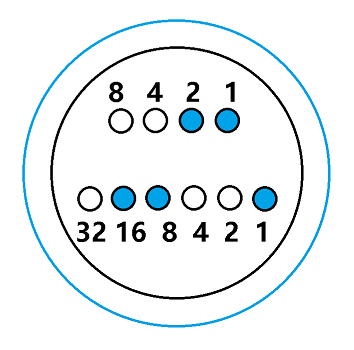
if (t & 1) {

if ((t & 2) && (t != 3)) ++t;

else --t;

} else {

t >>= 1;

 }

}

return cnt;

}

};

# 401. Binary Watch二进制手表

二进制手表顶部有 4 个 LED 代表小时（0-11），底部的 6 个 LED 代表分钟（0-59）。

每个 LED 代表一个 0 或 1，最低位在右侧。（上图二进制手表读取 “3:25”）

给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

输入: n = 1

返回: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

注意事项：输出的顺序没有要求。

小时不会以零开头，比如 “01:00” 是不允许的，应为 “1:00”。

分钟必须由两位数组成，可能以零开头，比如 “10:2” 是无效的，应为 “10:02”。

【分析】第一种是遍历，用到的是bitset这个C++ STL模板。

按照数值上分析，最多只有12×60=720种组合，所以只要遍历这720种组合，通过bitset的count函数判断各个组合的1的个数是否等于num，是则添加对应的时间字符串到结果数组中，最后返回结果数组即可。

【CODE】

class Solution {

public:

vector<string> readBinaryWatch(int num) {

vector<string> times;

for (int i = 0; i < 12; i++) {

bitset<4> h((size\_t) i);

for (int j = 0; j < 60; j++) {

bitset<6> m((size\_t) j);

if (h.count() + m.count() == num)

times.push\_back(to\_string(i) + (j < 10? ":0": ":") + to\_string(j));

}

}

return times;

}

};

【分析】除了按数值，也可以从二进制表示上遍历，由于小时有4位，分钟有6位，所以总共10位就可以表示所有的情况，最大的数值是0x2FF (1011 | 111111)，所以遍历从0到0x2FF之间的数值，判断1的个数是否等于num，是则截取出小时和分钟，并拼接成字符串添加到结果数组。

【CODE】class Solution {

public:

vector<string> readBinaryWatch(int num) {

vector<string> times;

if (num < 0 || num > 8)

return times;

for (int i = 0; i < 0x2FF; i++) {

if (bitset<10>(i).count() == num)

add(i, times);

}

return times;

}

void add(int i, vector<string>& t) {

string s;

int m = i & 0x3F;

if (m > 59)

return;

int h = i >> 6 & 0xF;

if (h > 11)

return;

t.push\_back(to\_string(h) + (m < 10? ":0" : ":") + to\_string(m));

}

};

# 405. Convert a Number to Hexadecimal数字转换为十六进制数

给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用 ***补码运算*** 方法。

注意:十六进制中所有字母(a-f)都必须是小写。

十六进制字符串中不能包含多余的前导零。如果要转化的数为0，那么以单个字符'0'来表示；对于其他情况，十六进制字符串中的第一个字符将不会是0字符。

给定的数确保在32位有符号整数范围内。

**不能使用任何由库提供的将数字直接转换或格式化为十六进制的方法。**

【分析】采取位操作的思路，每次取出最右边四位，如果其大于等于10，找到对应的字母加入结果，反之则将对应的数字加入结果，然后num像右平移四位，循环停止的条件是num为0，或者是已经循环了7次。

【CODE】class Solution {

public:

string toHex(int num) {

string res = "";

for (int i = 0; num && i < 8; ++i) {

int t = num & 0xf;

if (t >= 10) res = char('a' + t - 10) + res;

else res = char('0' + t) + res;

num >>= 4;

}

/\* string res = "", str = "0123456789abcdef";

\* int cnt = 0;

\* while (num != 0 && cnt++ < 8) {

\* res = str[(num & 0xf)] + res;

\* num >>= 4;

}\*/

return res.empty() ? "0" : res;

}

};

【分析】把要转换的十六进制的数字字母都放在一个字符串中，按位置直接取亦可。

# 421. Maximum XOR of Two Numbers in an Array数组中两个数的最大异或值

给定一个非空数组，数组中元素为 a0, a1, a2, … , an-1，其中 0 ≤ ai < 231 。

找到 ai 和aj 最大的异或 (XOR) 运算结果，其中0 ≤ i, j < n 。

你能在O(n)的时间解决这个问题吗？

【分析】这道题是一道典型的位操作Bit Manipulation的题目，开始以为异或值最大的两个数一定包括数组的最大值，但是OJ给了另一个例子{10,23,20,18,28}，这个数组的异或最大值是10和20异或，得到30。那么只能另辟蹊径，正确的做法是按位遍历，题目中给定了数字的返回不会超过231,那么最多只能有32位，我们用一个从左往右的mask，用来提取数字的前缀，然后将其都存入HashSet中，我们用一个变量t，用来验证当前位为1再或上之前结果res，看结果和HashSet中的前缀异或之后在不在HashSet中，这里用到了一个性质，若a^b=c，那么a=b^c，因为t是我们要验证的当前最大值，所以我们遍历HashSet中的数时，和t异或后的结果仍在HashSet中，说明两个前缀可以异或出t的值，所以我们更新res为t，继续遍历，如果上述讲解不容易理解，那么建议自己带个例子一步一步试试，并把每次循环中HashSet中所有的数字都打印出来，基本应该就能理解

3 10 5 25 2 8

11 1010 101 11001 10 1000

我们观察这些数字最大的为25，其二进制最高位在 i=4 时为1，那么我们的循环[31, 5]之间是取不到任何数字的，所以不会对结果res有任何影响。

当 i=4 时，我们此时mask为前28位为‘1’的二进制数，跟除25以外的任何数相‘与’，都会得到0。 然后跟25的二进制数10101相‘与’，得到二进制数10000，存入HashSet中，那么此时HashSet中就有0和16两个数字。此时我们的t为结果res（此时为0）‘或’上二进制数10000，得到二进制数10000。然后我们遍历HashSet，由于HashSet是无序的，所以我们会取出0和16中的其中一个，如果prefix取出的是0，那么t=16‘异或’上0，还等于16，而16是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。实际上prefix先取16的话也一样，那么t=16‘异或’上16，等于0，而0是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=3 时，我们此时mask为前29位为‘1’的二进制数，如上所示，跟数字3，5，2中任何一个相‘与’，都会得到0。然后跟10的二进制数1010，或跟8的二进制数1000相‘与’，都会得到二进制数1000，即8。跟25的二进制数11001相‘与’，会得到二进数11000，即24，存入HashSet中，那么此时HashSet中就有0，8，和24三个数字。此时我们的t为结果res（此时为16）‘或’上二进制数1000，得到二进制数11000，即24。此时遍历HashSet中的数，当prefix取出0，那么t=24‘异或’上0，还等于24，而24是在HashSet中存在的，所以此时结果res更新为24，然后break掉遍历HashSet的循环。大家可以尝试其他的数，当prefix取出24，其实也可以更新结果res为24的。但是8就不行啦，因为HashSet中没有16。不过无所谓了，我们只要有一个能更新结果res就可以了。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=2 时，我们此时mask为前30位为‘1’的二进制数，如上所示，跟3的二进制数11相‘与’，会得到二进制数0，即0。然后跟10的二进制数1010相‘与’，会得到二进制数1000，即8。然后跟5的二进制数101相‘与’，会得到二进制数100，即4。然后跟25的二进制数11001相‘与’，会得到二进制数11000，即24。跟数字2和8相‘与’，分别会得到0和8，跟前面重复了。所以最终HashSet中就有0，4，8，和24这四个数字。此时我们的t为结果res（此时为24）‘或’上二进制数100，得到二进制数11100，即28。那么就要验证结果res能否取到28。我们遍历HashSet，当prefix取出0，那么t=28‘异或’上0，还等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么t=28‘异或’上二进制数100，等于24，在HashSet中存在，Bingo！结果res更新为28。其他的数可以不用试了。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=1 时，我们此时mask为前31位为‘1’的二进制数，如上所示，每个数与mask相‘与’后，我们HashSet中会有2，4，8，10，24这五个数。此时我们的t为结果res（此时为28）‘或’上二进制数10，得到二进制数11110，即30。那么就要验证结果res能否取到30。我们遍历HashSet，当prefix取出2，那么t=30‘异或’上2，等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么t=30‘异或’上4，等于26，但是HashSet中没有26，所以不行。当prefix取出8，那么t=30‘异或’上8，等于22，但是HashSet中没有22，所以不行。当prefix取出10，那么t=30‘异或’上10，等于20，但是HashSet中没有20，所以不行。当prefix取出24，那么t=30‘异或’上24，等于6，但是HashSet中没有6，所以不行。遍历完了HashSet所有的数，结果res没有被更新，还是28。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=0 时，我们此时mask为前32位为‘1’的二进制数，如上所示，每个数与mask相‘与’后，我们HashSet中会有2，3，5，8，10，25这六个数。此时我们的t为结果res（此时为28）‘或’上二进制数1，得到二进制数11101，即29。那么就要验证结果res能否取到29。取出HashSet中每一个数字来验证，跟上面的验证方法相同，这里博主偷懒就不写了，最终可以发现，结果res无法被更新，还是28，所以最终的结果就是28。

综上所述，我们来分析一下这道题的核心。我们希望用二进制来拼出结果的数，最终结果28的二进制数为11100，里面有三个‘1’，我们来找一下都是谁贡献了这三个‘1’？在 i=4 时，数字25贡献了最高位的‘1’，在 i=3 时，数字25贡献了次高位的‘1’，在 i=2 时，数字5贡献了第三位的‘1’。而一旦某个数贡献了‘1’，那么之后在需要贡献‘1’的时候，此数就可以再继续贡献‘1’。而一旦有两个数贡献了‘1’后，那么之后的‘1’就基本上只跟这两个数有关了，其他数字有‘1’也贡献不出来。验证方法里使用了前面提到的性质，a ^ b = t，如果t是所求结果话，我们可以先假定一个t，然后验证，如果a ^ t = b成立，说明该t可以通过a和b‘异或’得到。

【CODE】class Solution {

public:

int findMaximumXOR(vector<int>& nums) {

int res = 0, mask = 0;

for (int i = 31; i >= 0; --i) {

mask |= (1 << i);

unordered\_set<int> s;

for (int num : nums) {

s.insert(num & mask);

}

int t = res | (1 << i);

for (int prefix : s) {

if (s.count(t ^ prefix)) {

res = t;

break;

}

}

}

return res;

}

};

# 461. Hamming Distance汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。注意：0 ≤ x, y < 231.

【分析】两个数字之间的汉明距离就是其二进制数对应位不同的个数，那么最直接了当的做法就是按位分别取出两个数对应位上的数并异或，我们知道异或的性质上相同的为0，不同的为1，我们只要把为1的情况累加起来就是汉明距离。

【CODE】class Solution {

public:

int hammingDistance(int x, int y) {

int res = 0;

for (int i = 0; i < 32; ++i) {

if ((x & (1 << i)) ^ (y & (1 << i))) {

++res;

}

}

return res;

}

};

【分析】对上面的代码进行优化，我们可以一开始直接将两个数字异或起来，然后我们遍历异或结果的每一位，统计为1的个数，也能达到同样的效果。

【CODE】

class Solution {

public:

int hammingDistance(int x, int y) {

int res = 0, exc = x ^ y;

for (int i = 0; i < 32; ++i) {

res += (exc >> i) & 1;

}

/\*while (exc) {

\* ++res;

\* exc &= (exc - 1);

}\*/

return res;

}

};

【分析】上面的遍历每一位的方法并不高效，还可以进一步优化，假如数为num, num & (num - 1)可以快速地移除最右边的bit 1， 一直循环到num为0, 总的循环数就是num中bit 1的个数。

【分析】递归的写法，非常的简洁，递归终止的条件是当两个数异或为0时，表明此时两个数完全相同，我们返回0，否则我们返回异或和对2取余加上对x/2和y/2调用递归的结果。异或和对2取余相当于检查最低位是否相同，而对x/2和y/2调用递归相当于将x和y分别向右移动一位，这样每一位都可以比较到，也能得到正确结果。

【CODE】

class Solution {

public:

int hammingDistance(int x, int y) {

if ((x ^ y) == 0) return 0;

return (x ^ y) % 2 + hammingDistance(x / 2, y / 2);

}

};

# 477. Total Hamming Distance汉明距离总和

两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。

计算一个数组中，任意两个数之间汉明距离的总和。

【分析】需要用异或来求每个位上的情况，那么我们需要来找出某种规律来，比如我们看下面这个例子，4，14，2和1：

4: 0 1 0 0

14: 1 1 1 0

2: 0 0 1 0

1: 0 0 0 1

我们先看最后一列，有三个0和一个1，那么它们之间相互的汉明距离就是3，即1和其他三个0分别的距离累加，然后在看第三列，累加汉明距离为4，因为每个1都会跟两个0产生两个汉明距离，同理第二列也是4，第一列是3。我们仔细观察累计汉明距离和0跟1的个数，我们可以发现其实就是0的个数乘以1的个数，发现了这个重要的规律，那么整道题就迎刃而解了，只要统计出每一位的1的个数即可。

【CODE】

class Solution {

public:

int totalHammingDistance(vector<int>& nums) {

int res = 0, n = nums.size();

for (int i = 0; i < 32; ++i) {

int cnt = 0;

for (int num : nums) {

if (num & (1 << i)) ++cnt;

}

res += cnt \* (n - cnt);

}

return res;

}

};

# 476. Number Complement数字的补数

给定一个正整数，输出它的补数。补数是对该数的二进制表示取反。

注意:给定的整数保证在32位带符号整数的范围内。

你可以假定二进制数不包含前导零位。

输入: 5 输出: 2

解释: 5的二进制表示为101（没有前导零位），其补数为010。所以你需要输出2。

输入: 1 输出: 0

解释: 1的二进制表示为1（没有前导零位），其补数为0。所以你需要输出0。

【分析】位操作里面的取反符号～本身就可以翻转位，但是如果直接对num取反的话就是每一位都翻转了，而最高位1之前的0是不能翻转的，所以我们只要用一个mask来标记最高位1前面的所有0的位置，然后对mask取反后，与上对num取反的结果即可。

【CODE】

class Solution {

public:

int findComplement(int num) {

int mask = INT\_MAX;

while (mask & num) mask <<= 1;

return ~mask & ~num;

}

};

【分析】迭代的写法，一行搞定，思路就是每次都右移一位，并根据最低位的值先进行翻转，如果当前值小于等于1了，就不用再调用递归函数。

【CODE】

class Solution {

public:

int findComplement(int num) {

return (1 - num % 2) + 2 \* (num <= 1 ? 0 : findComplement(num / 2));

}

};

# 693. Binary Number with Alternating Bits交替位二进制数

给定一个正整数，检查他是否为交替位二进制数：换句话说，就是他的二进制数相邻的两个位数永不相等。

【分析】利用了0和1的交替的特性，进行错位相加，从而组成全1的二进制数，然后再用一个检测全1的二进制数的trick，就是‘与’上加1后的数，因为全1的二进制数加1，就会进一位，并且除了最高位，其余位都是0，跟原数相‘与’就会得0，所以我们可以这样判断。比如n是10101，那么n>>1就是1010，二者相加就是11111，再加1就是100000，二者相‘与’就是0。

【CODE】

class Solution {

public:

bool hasAlternatingBits(int n) {

return ((n + (n >> 1) + 1) & (n + (n >> 1))) == 0;

}

};

【分析】将n右移两位，再和原来的n亦或，得到的新n其实就是除了最高位，其余都是0的数，然后再和自身减1的数相‘与’，如果是0就返回true，反之false。比如n是10101，那么n/4是101，二者相‘亦或’，得到10000，此时再减1，为1111，二者相‘与’得0。

【CODE】

class Solution {

public:

bool hasAlternatingBits(int n) {

return ((n ^= n / 4) & (n - 1)) == 0;

}

};

# 762. Prime Number of Set Bits in Binary Representation二进制表示中质数个计算置位

给定两个整数 L 和 R ，找到闭区间 [L, R] 范围内，计算置位位数为质数的整数个数。（注意，计算置位代表二进制表示中1的个数。例如 21 的二进制表示 10101 有 3 个计算置位。还有，1 不是质数。）

|  |  |
| --- | --- |
| 输入: L = 6, R = 10  输出: 4  解释:  6 -> 110 (2 个计算置位，2 是质数)  7 -> 111 (3 个计算置位，3 是质数)  9 -> 1001 (2 个计算置位，2 是质数)  10-> 1010 (2 个计算置位，2 是质数) | 输入: L = 10, R = 15  输出: 5  解释:  10 -> 1010 (2 个计算置位, 2 是质数)  11 -> 1011 (3 个计算置位, 3 是质数)  12 -> 1100 (2 个计算置位, 2 是质数)  13 -> 1101 (3 个计算置位, 3 是质数)  14 -> 1110 (3 个计算置位, 3 是质数)  15 -> 1111 (4 个计算置位, 4 不是质数) |

【分析】C++的内置函数\_\_builtin\_popcount来快速的求出非零位的个数cnt，然后又利用到了20以内的数，只要不能被2和3的一定是质数，又可以快速判断了质数。

【CODE】

class Solution {

public:

int countPrimeSetBits(int L, int R) {

int res = 0;

for (int i = L; i <= R; ++i) {

int cnt = \_\_builtin\_popcount(i);

res += cnt < 4 ? cnt > 1 : (cnt % 2 && cnt % 3);

}

return res;

}

};