## 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。

示例 1:

输入: [1,2,3,4,5,6,7] 和 k = 3

输出: [5,6,7,1,2,3,4]

示例 2:

输入: [-1,-100,3,99] 和 k = 2

输出: [3,99,-1,-100]

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。

解法1：

class Solution{

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

{

int n = nums.size();

if(nums.empty() || (k %= n) == 0)

return;

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

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

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

}

};

## 移动零

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

示例:

输入: [0,1,0,3,12]

输出: [1,3,12,0,0]

说明:

必须在原数组上操作，不能拷贝额外的数组。

尽量减少操作次数。

解法1：

class Solution{

void moveZeroes(vector<int>& nums)

{

int iLen = nums.size();

if(0 == iLen)

return;

int iIndex = 0;

//把不为0的值从第一位开始存

for(int i = 0; i < iLen; ++i)

if(nums[i] != 0)

nums[iIndex++] = nums[i];

//后面补0

for(int i = iIndex; i < iLen; ++i)

nums[i] = 0;

}

}

## 旋转图像

给定一个 n × n 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明：

你必须在**原地**旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

示例 1:

给定 matrix =

[

[1,2,3],

[4,5,6],

[7,8,9]

],

原地旋转输入矩阵，使其变为:

[

[7,4,1],

[8,5,2],

[9,6,3]

]

示例 2:

给定 matrix =

[

[ 5, 1, 9,11],

[ 2, 4, 8,10],

[13, 3, 6, 7],

[15,14,12,16]

],

原地旋转输入矩阵，使其变为:

[

[15,13, 2, 5],

[14, 3, 4, 1],

[12, 6, 8, 9],

[16, 7,10,11]

]

解法1：先取其转置矩阵，在逐行逆序

[1,2,3], [1,4,7] [7,4,1]

[4,5,6] -> [2,5,8] -> [8,5,2]

[7,8,9] [3,6,9] [9,6,3]

class Solution {

public:

void rotate(vector<vector<int>>& matrix) {

if(matrix.size() == 0 || matrix.size() == 1)

return;

//求matrix的转置矩阵

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

for(int j = i; j < matrix.size(); ++j)

if(i != j)

{

int tmp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = tmp;

}

//在将每一行反转

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

reverse(matrix[i].begin(),matrix[i].end());

}

};

解法2：以对角线为轴进行翻转，在以X轴中线上下翻转（蓝色表示翻转轴）

[1,2,3] [9,6,3] [7,4,1]

[4,5,6]-> [8,5,2]-> [8,5,2]

[7,8,9] [7,4,1] [9,6,3]

class Solution {

public:

void rotate(vector<vector<int>>& matrix) {

if(matrix.size() == 0 || matrix.size() == 1)

return;

int n = matrix.size();

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

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

swap(matrix[i][j],matrix[n-1-j][n-1-i]);

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

}

};

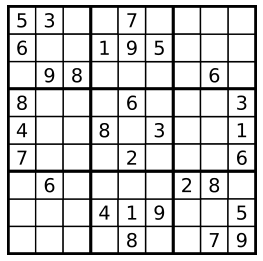
## 有效的数独

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

数字 1-9 在每一行只能出现一次。

数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。



上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例 1:

输入:

[

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

输出: true

示例 2:

输入:

[

["8","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

输出: false

解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。

但由于位于左上角的 3x3 宫内有两个 8 存在, 因此这个数独是无效的。

说明:

一个有效的数独（部分已被填充）不一定是可解的。

只需要根据以上规则，验证已经填入的数字是否有效即可。

给定数独序列只包含数字 1-9 和字符 '.' 。

给定数独永远是 9x9 形式的。

解法1：

class Solution {

public:

bool isValidSudoku(vector<vector<char>>& board) {

if(board.size() != 9)

return false;

set<char> rowset;

set<char> colset;

set<char> blockset;

for(int i = 0; i < 9; ++i)

{

rowset.clear();

colset.clear();

for(int j = 0; j < 9; ++j)

{

//检查行

if(board[i][j] != '.')

{

if(rowset.find(board[i][j]) != rowset.end())

return false;

rowset.insert(board[i][j]);

}

//检查列

if(board[j][i] != '.')

{

if(colset.find(board[j][i]) != colset.end())

return false;

colset.insert(board[j][i]);

}

//检查块

if((i % 3 == 0) && (j % 3 == 0))

{

blockset.clear();

for(int x = i; x < i+3; ++x)

for(int y = j; y < j+3; ++y)

if(board[x][y] != '.')

{

if(blockset.find(board[x][y]) != blockset.end())

return false;

blockset.insert(board[x][y]);

}

}

}

}

return true;

}

};

## 字符串转整数 (atoi)

实现 atoi，将字符串转为整数。

在找到第一个非空字符之前，需要移除掉字符串中的空格字符。如果第一个非空字符是正号或负号，选取该符号，并将其与后面尽可能多的连续的数字组合起来，这部分字符即为整数的值。如果第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

字符串可以在形成整数的字符后面包括多余的字符，这些字符可以被忽略，它们对于函数没有影响。

当字符串中的第一个非空字符序列不是个有效的整数；或字符串为空；或字符串仅包含空白字符时，则不进行转换。

若函数不能执行有效的转换，返回 0。

说明：

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]。如果数值超过可表示的范围，则返回 INT\_MAX (231 − 1) 或 INT\_MIN (−231) 。

示例 1:

输入: "42"

输出: 42

示例 2:

输入: " -42"

输出: -42

解释: 第一个非空白字符为 '-', 它是一个负号。

我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

示例 3:

输入: "4193 with words"

输出: 4193

解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。

示例 4:

输入: "words and 987"

输出: 0

解释: 第一个非空字符是 'w', 但它不是数字或正、负号。

因此无法执行有效的转换。

示例 5:

输入: "-91283472332"

输出: -2147483648

解释: 数字 "-91283472332" 超过 32 位有符号整数范围。

因此返回 INT\_MIN (−231) 。

class Solution {

public:

int myAtoi(string str) {

if(str.size() == 0)

return 0;

int res = 0;

int sign = 1;

int iIndex = 0;

//找到第一个非空字符的位置

while(iIndex < str.size() && str[iIndex] == ' ')

iIndex++;

//判断第一个非空字符是否是正负号

if(str[iIndex] == '+' || str[iIndex] == '-')

sign = str[iIndex++] == '+' ? 1 : -1;

//从第一个数字字符开始计算

while(iIndex < str.size() && isdigit(str[iIndex]))

{

**// that statement is used to test if the num is bigger than INT\_MAX after the str[i] is handled, if base > INT\_MAX/10,**

**// then base10+ str[i] -‘0’> base10>INT\_MAX, or when base== INT\_MAX/10, that means all the places are the same as INT\_MAX except the ones place, so str[i]>‘7’ is needed.**

**// 上面这段是LeetCode国外站对下面代码的解释。**

**// 简单来说就是**

**// 如果`base > MAX\_VALUE/10`，那么`base\*10 + new\_value` > `base\*10` > ` MAX\_VALUE`。这个应该很容易理解，这种情况下就会发生溢出。**

**// 若`base == INT\_MAX/10`，而且`new\_value = str.charAt(i++) - '0'`大于`7`，也会发生溢出。因为`MAX\_VALUE = 2147483647`**

if((res > INT\_MAX / 10) || (res == INT\_MAX / 10 && str[iIndex] - '0' > 7))//溢出

return sign == 1 ? INT\_MAX : INT\_MIN;

res = res \* 10 + str[iIndex++] - '0';

}

return res\*sign;

}

};

## 字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

案例:

s = "leetcode"

返回 0.

s = "loveleetcode",

返回 2.

注意事项：您可以假定该字符串只包含小写字母。

解法1：

class Solution {

public:

int firstUniqChar(string s) {

if(s.size() == 0)

return -1;

int tmp[26] = {0};

//遍历每个字符出现的次数

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

tmp[s[i]-'a']++;

//找到第一个出现次数为1的字符

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

if(tmp[s[i]-'a'] == 1)

return i;

return -1;

}

};

## 数数并说

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。

示例 1:

输入: 1

输出: "1"

示例 2:

输入: 4

输出: "1211"

解法1：

class Solution {

public:

string countAndSay(int n) {

if(n <= 0)

return "-1";

string res = "1";

for(int i = 1; i < n; ++i)

{

string str;

int iIndex = 0;

while(iIndex < res.size())

{

//记录值

char tmp = res[iIndex];

//记录数连续出现的次数

int iCount = 0;

//查找数连续出现的次数

while(iIndex < res.size() && res[iIndex] == tmp)

{

++iCount;

++iIndex;

}

//将数连续出现的次数和数值存入string中

str.push\_back(iCount+'0');

str.push\_back(tmp);

}

res = str;

}

return res;

}

};

## 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。

示例：

输入：1->2->4, 1->3->4

输出：1->1->2->3->4->4

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

方法一：递归

class Solution {

public:

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

if(l1 == NULL || l2 == NULL)

{

if(!l1)

return l2;

else if(!l2)

return l1;

else

return NULL;

}

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

{

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

return l1;

}

else

{

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

return l2;

}

}

};

static const auto io\_speed\_up=[]()

{

std::ios::sync\_with\_stdio(false);

cin.tie(nullptr);

return 0;

}();

方法2：

class Solution {

public:

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

if(l1 == NULL) return l2;

if(l2 == NULL) return l1;

ListNode\* new\_list = NULL;

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

{

new\_list = l1;

l1 = l1->next;

}

else

{

new\_list = l2;

l2 = l2->next;

}

ListNode \* pre = new\_list;

while( l1 != NULL && l2 != NULL ){

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

pre->next = l1;

l1 = l1->next;

}

else{

pre->next = l2;

l2 = l2->next;

}

pre = pre->next;

}

if( l1 ) pre->next = l1;

else pre->next = l2;

return new\_list;

}

};

方法3：

前面加一个无关的节点，不用去比较两个链表第一个的大小

class Solution {

public:

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

if(!l1 && !l2) return NULL;

ListNode \* new\_list = new ListNode(0);

ListNode \* pre = new\_list;

while( l1 != NULL && l2 != NULL ){

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

pre->next = l1;

l1 = l1->next;

}

else{

pre->next = l2;

l2 = l2->next;

}

pre = pre->next;

}

if( l1 ) pre->next = l1;

else pre->next = l2;

return new\_list->next;

}

};

## 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。

示例：

给定一个链表: 1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.

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？

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

class Solution {

public:

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

//如果没有说明n一定有效，需判断n是否超长

ListNode \*node = head;

unsigned int uiLen = 0;

while(node)

{

++uiLen;

node = node->next;

}

if(n > uiLen)

return head;

ListNode \*preNode = head;

ListNode \*curNode = head;

int i = 0;

//将前指针向后移动n个位置

while(preNode && i < n)

{

preNode = preNode->next;

++i;

}

//已移动到最后一个节点的下个位置（即空节点），说明待删除的是第一个节点

if(!preNode)

return curNode->next;

//前后指针同时向后移动，直到前指针移到最后一个节点位置

while(preNode->next)

{

preNode = preNode->next;

curNode = curNode->next;

}

//curNode指向节点的下一个节点为待删除节点，去掉待删除节点

curNode->next = curNode->next->next;

return head;

}

};

## 反转链表

解法1：一个最简单的办法就是借助栈的后进先出功能，先扫描一遍链表保存每个节点的值，然后再从头到尾遍历，将栈中元素值一一赋给链表节点。时空复杂度都是O(n)。

代码略。

解法2：可以做到in-place的反转。链表反转后，实际上只是中间节点的指针反转，并且反转后原来链表的头结点的下一个节点应该为NULL，而反转后链表的头结点为原来链表的尾节点。我们可以从头结点开始，每次处理两个节点之间的一个指针，将其反转过来。然后再处理接下来两个节点之间的指针……直至遇到尾节点，设置为新链表的头结点即可。

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* rHead = NULL; // 反转后的头节点

ListNode\* curr = head; // 当前处理节点

ListNode\* pTail = NULL; // 反转后尾节点

while(curr != NULL) {

ListNode\* pNext = curr->next;

if(pNext == NULL)

rHead = curr;

curr->next = pTail;

pTail = curr;

curr = pNext;

}

return rHead;

}

};

解法3：递归的方式

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (head == NULL || head->next == NULL) return head;

ListNode\* rHead = reverseList(head->next); // 反转得到新链表的头节点

head->next->next = head; // 当前节点的下一个节点的next指针反转过来

head->next = NULL; // 设置新链表的尾节点

return rHead;

}

};

## 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？

解法1：

双指针解法。设置两个指针（快慢指针），慢指针每次走一步，快指针每次走两步，如果链表中有环，两个指针最终肯定会相遇。

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

class Solution {

public:

bool hasCycle(ListNode \*head) {

ListNode \*fast = head;

ListNode \*slow = head;

while(fast && fast->next)

{

fast = fast->next->next;

slow = slow->next;

if(fast == slow)

return true;

}

return false;

}

};

## 二叉树的前序遍历

给定一个二叉树，返回它的前序遍历。

**进阶:** 递归算法很简单，你可以通过迭代算法完成吗？

### 解法1：递归方法

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

1.1 使用局部变量数组返回结果

class Solution {

public:

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

vector<int> res;

if(!root)

return res;

//根节点

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

//左子树

vector<int> tmp1 = preorderTraversal(root->left);

if(!tmp1.empty())

for(int i = 0; i < tmp1.size(); ++i)

res.push\_back(tmp1[i]);

//右子树

vector<int> tmp2 = preorderTraversal(root->right);

if(!tmp2.empty())

for(int i = 0; i < tmp2.size(); ++i)

res.push\_back(tmp2[i]);

return res;

}

};

1.2 使用全局变量数组返回结果

class Solution{

private:

vector<int> preorderRes;

public:

vector<int> preorderTraversal(TreeNode \* root)

{

if(!root)

return preorderRes;

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

preorderRes = preorderTraversal(root->left);

preorderRes = preorderTraversal(root->right);

return preorderRes;

}

};

### 解法2：迭代方法

2.1

class Solution{

vector<int> preorderTraversal(TreeNode \*root)

{

vector<int> res;

if(!root)

return res;

TreeNode \*cur = root;

stack<TreeNode\*> s;

while(curr || !s.empty())

{

if(curr)

{

s.push(curr);//当前节点入栈

res.push\_back(curr->val);//先序，先记录根节点的值

curr = curr->left;//走左边

}

else

{

curr = s.top()->right;//走右边

s.pop();

}

}

return res;

}

};

2.2 LeetCode示例代码

class Solution{

public:

vector<int> preorderTraversal(TreeNode \*root)

{

vector<int> preorderRes;

if(!root)

return preorderRes;

stack<TreeNode \*> s;

s.push(root);

while(!s.empty())

{

TreeNode \*curr = s.top();

preorderRes.push\_back(curr->val);

s.pop();

if(curr->right)

s.push(curr->right);

if(curr->left)

s.push(curr->left);

}

return preorderRes;

}

};

## 二叉树的中序遍历

给定一个二叉树，返回它的中序遍历。

struct TreeNode{

int val;

struct TreeNode \*left;

struct TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

### 解法一：递归方法

1.1 使用局部变量数组返回

class Solution{

public:

vector<int> InorderTraversal(TreeNode \*root)

{

vector<int> res;

if(!root)

return res;

//左子树

vector<int> left;

left = InorderTraversal(root->left)

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

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

//根节点

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

//右子树

vector<int> right;

right = InorderTraversal(root->right);

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

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

return res;

}

}

1.2 使用全局变量数组

class Solution{

private:

vector<int> inorderRes;

public:

vector<int> inorderTraversal(TreeNode \*root)

{

if(!root)

return inorderRes;

inorderRes = inorderTraversal(root->left);

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

inorderRes = inorderTraversal(root->right);

return inorderRes;

}

};

### 解法二：迭代方法

class Solution{

vector<int> InorderTraversal(TreeNode \*root)

{

vector<int> res;

if(!root)

return res;

TreeNode \*curr = root;

stack<TreeNode \*> s;

while(curr || !s.empty())

{

if(curr)

{

s.push(curr);//节点入栈

curr = curr->left;//走左边

}

else

{

res.push\_back(s.top()->val);

curr = s.top()->right;//走右边

s.pop();

}

}

return res;

}

};

## 二叉树的后序遍历

给定一个二叉树，返回它的后序遍历。

struct TreeNode{

int val;

struct TreeNode \*left;

struct TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

### 解法一：递归方法

1.1 使用局部数组变量返回

class Solution {

public:

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

vector<int> res;

if(!root)

return res;

//左子树

vector<int> left;

left = postorderTraversal(root->left);

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

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

//右子树

vector<int> right;

right = postorderTraversal(root->right);

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

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

//根节点

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

return res;

}

};

1.2 使用全局数组变量

class Solution{

private:

vector<int> postorderRes;

public:

vector<int> postorderTraversal(TreeNode \*root)

{

if(!root)

return postorderRes;

postorderRes = postorderTraversal(root->left);

postorderRes = postorderTraversal(root->right);

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

return postorderRes;

}

}

### 解法二：迭代方法

class Solution {

public:

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

vector<int> res;

if(!root)

return res;

stack<TreeNode \*> s;

s.push(root);

while(!s.empty())

{

TreeNode \*node = s.top();

res.push(node->val);

s.pop();

if(node->left)

s.push(node->left);

if(node->right)

s.push(node->right);

}

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

return res;

}

};

## 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）。

struct TreeNode{

int val;

struct TreeNode \*left;

struct TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

class Solution{

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

{

vector<vector<int>> res;

if(!root)

return res;

queue<TreeNode \*> q;

q.push(root);

while(!q.empty())

{

int iNodeCount = q.size();//每层节点的个数

vector<int> levelVal;

//将每层节点的值存入res,清空队列，再存入下一层的全部节点

***//for(int i = 0; i < q.size();++i)循环中q在弹出，大小是变化的，所以不能用来跟循环变量i比较，应先将大小值记录下来。纠结了好久。***

for(int i = 0; i < iNodeCount;++i)

{

TreeNode \*node = q.front();

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

q.pop();

if(node->left)

q.push(node->left);

if(node->right)

q.push(node->right);

}

res.push\_back(levelVal);

}

return res;

}

};

## 二叉树的深度遍历

找出从二叉树的根节点到每个叶子节点的路径

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution{

private:

vector<int> onePath;

public:

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

{

if(!root)

return;

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

if(!root->left && !root->right)

res.push\_back(onePath);

if(root->left)

{

DFSTraversal(root->left, res);

onePath.pop\_back();

}

if(root->right)

{

DFSTraversal(root->right, res);

onePath.pop\_back();

}

}

}

## 二叉树的最大深度

解法1：

递归方法

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(root == NULL)

return 0;

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

}

};

解法2：

使用层序遍历二叉树，然后计算总层数，即为二叉树的最大深度

Class Solution{

public:

int maxDepth(TreeNode \*root){

if(root == NULL)

return 0;

int res = 0;

queue<TreeNode\*> q;

q.push(root);

while(!q.empty())

{

++res;

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

{

TreeNode \*node = q.front();

q.pop();

//将每层的节点存入queue中

if(node->left)

q.push(node->left);

if(node->right)

q.push(node->right);

}

}

return res;

}

}

## 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。

struct TreeNode{

int val;

TreeNode \*left;

TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

解法1：采用递归调用的形式，每次从sum中减去根节点的值，然后判断当前节点是否是叶子节点了。

class Solution{

public:

bool hasPathSum(TreeNode \*root, int sum)

{

if(!root)

return false;

int t = sum - root->val;

if(!root->left && !root->right)

return t == 0 ? true : false;

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

}

};

## 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。

例如，给出

中序遍历 inorder = [9,3,15,20,7]

后序遍历 postorder = [9,15,7,20,3]

struct TreeNode {

int val;

TreeNode \*left;

TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

解法1：比较容易看懂的

class Solution{

TreeNode \*buildTree(vector<int>& inorder, vector<int>& postorder)

{

int iPostSize = postorder.size();

if(0 == iPostSize)

return NULL;

TreeNode \*root = new TreeNode(postorder[iPostSize-1]);

if(1 == iPostSize)//是叶子节点

return root;

//分别找左右子树的中、后序遍历

vector<int> leftInorder;

vector<int> rightInorder;

vector<int> leftPostorder;

vector<int> rightPostorder;

//找到根节点在中序遍历中的位置

int iPos = 0;

for(; iPos < inorder.size(); ++iPos)

if(inorder[iPos] == postorder[iPostSize-1])

break;

//左子树中序

for(int i = 0; i < iPos; ++i)

leftInorder.push\_back(inorder[i]);

//右子树中序

for(int i = iPos+1; i < inorder.size(); ++i)

rightInorder.push\_back(inorder[i]);

//在后序遍历中找左子树的后序

for(int i = 0; i < iPos; ++i)

leftPostorder.push\_back(postorder[i]);

//在后序遍历中找右子树的后序

for(int i = leftPostorder.size(); i < postorder.size()-1; ++i)

rightPostorder.push\_back(postorder[i]);

//递归调用

root->left = buildTree(leftInorder,leftPostorder);

root->right = buildTree(rightInorder,rightPostorder);

return root;

}

};

其他解法参见LeetCode提交的代码，不太看的懂，没注释

## 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。

例如，给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

struct TreeNode {

int val;

TreeNode \*left;

TreeNode \*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

}

解法1：

class Solution {

public:

TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {

if(preorder.size() == 0)

return NULL;

TreeNode \*root = new TreeNode(preorder[0]);

if(preorder.size() == 1)

return root;

//去中序遍历中找根节点位置

int iPos = 0;

for(; iPos < inorder.size(); ++iPos)

if(inorder[iPos] == preorder[0])

break;

vector<int> leftPreorder;

vector<int> leftInorder;

vector<int> rightPreorder;

vector<int> rightInorder;

//左子树的中序遍历

for(int i = 0; i < iPos; ++i)

leftInorder.push\_back(inorder[i]);

//右子树的中序遍历

for(int i = iPos+1; i < inorder.size(); ++i)

rightInorder.push\_back(inorder[i]);

//左子树的前序遍历

for(int i = 1; i < iPos+1; ++i)

leftPreorder.push\_back(preorder[i]);

//右子树的前序遍历

for(int i = 1+leftPreorder.size(); i < preorder.size(); ++i)

rightPreorder.push\_back(preorder[i]);

root->left = buildTree(leftPreorder,leftInorder);

root->right = buildTree(rightPreorder,rightInorder);

return root;

}

};

解法2：LeetCode上运行最快的示例代码，有点看不懂

class Solution {

public:

TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {

stack<TreeNode\*> s;

TreeNode \*root = NULL,\*node;

int i = 1,j = 0, flag = 0;

if(preorder.size() == 0)

return root;

root = new TreeNode(preorder[0]);

s.push(root);

node = root;

int size =preorder.size();

while(i<size){

if(!s.empty()&&s.top()->val == inorder[j]){

node = s.top();

s.pop();

flag = 1;

j++;

}

else{

TreeNode\* temp = new TreeNode(preorder[i]);

i++;

if(flag == 0){

node->left =temp;

node = node->left;

s.push(node);

}

else{

node->right =temp;

node = node->right;

s.push(node);

flag = 0;

}

}

}

return root;

}

};

## 每个节点的右向指针

给定一个二叉树

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

说明:

你只能使用**额外常数空间**。

使用**递归**解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

你可以**假设它是一个完美二叉树**（即所有叶子节点都在同一层，每个父节点都有两个子节点）。

示例:

给定完美二叉树，

1

/ \

2 3

/ \ / \

4 5 6 7

调用你的函数后，该完美二叉树变为：

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

struct TreeLinkNode

{

int val;

TreeLinkNode \*left, \*right,\*next;

TreeLinkNode(int x):x(val),left(NULL),right(NULL),next(NULL){}

}

解法1：（非递归，但不是O(1)的空间复杂度）

class Solution

{

public:

void connect(TreeLinkNode\* root)

{

if(!root)

return;

queue<TreeLinkNode\*> q;

q.push(root);

while(!q.empty())

{

int iSize = q.size();

for(int i = 0; i < iSize; ++i)

{

TreeLinkNode \*pNode = q.front();

q.pop();

if(i < iSize - 1)

pNode->next = q.front();

if(pNode->left)

q.push(pNode->left);

if(pNode->right)

q.push(pNode->right);

}

}

}

};

解法2：递归

class Solution{

public:

void connect(TreeLinkNode \*root)

{

if(!root)

return;

if(root->left)

root->left->next = root->right;

if(root->right)

root->right->next = root->next ? root->next->left : NULL;

connect(root->left);

connect(root->right);

}

}

解法3：

class Solution{

public:

void connect(TreeLinkNode \*root)

{

if(!root)

return;

TreeLinkNode \*start = root,\*cur = NULL;

while(start->left)

{

cur = start;

while(cur)

{

cur->left->next = cur->right;

if(cur->next)

{

cur->right->next = cur->next->left;

cur = cur->next;

}

}

start = start->left;

}

}

}

## 二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

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

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

\_\_\_\_\_\_\_3\_\_\_\_\_\_

/ \

\_\_\_5\_\_ \_\_\_1\_\_

/ \ / \

6 \_2 0 8

/ \

7 4

示例 1:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

说明:

所有节点的值都是唯一的。

p、q 为不同节点且均存在于给定的二叉树中。

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

解法1：使用dfs得到根结点到某个结点的路径，然后比对路径就可以得到最近的公共祖先

class Solution {

public:

vector<TreeNode\*> fa1;

vector<TreeNode\*> fa2;

void dfs(TreeNode\* root, TreeNode \*p, TreeNode \*q, vector<TreeNode\*> &fa)

{

if(!root)

return;

if(root == p)

fa1 = fa;

if(root == q)

fa2 = fa;

if(root->left)

{

fa.push\_back(root->left);

dfs(root->left,p,q,fa);

fa.pop\_back();

}

if(root->right)

{

fa.push\_back(root->right);

dfs(root->right,p,q,fa);

fa.pop\_back();

}

}

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if(!root)

return NULL;

vector<TreeNode\*> fa;

fa.push\_back(root);

dfs(root,p,q,fa);

int i = 0;

TreeNode \*res = NULL;

while(i < fa1.size() && i < fa2.size())

{

if(fa1[i] == fa2[i])

res = fa1[i++];

else

break;

}

return res;

}

};

## 二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例:

你可以将以下二叉树：

1

/ \

2 3

/ \

4 5

序列化为 "[1,2,3,null,null,4,5]"

提示: 这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明: 不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

解法1：

class Codec {

public:

void serialize(TreeNode\* root, stringstream& ss)

{

if(root==NULL)

{

ss<<"\*";

return ;

}

ss<<"n"<<root->val;

serialize(root->left, ss);

serialize(root->right, ss);

}

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

stringstream ss;

serialize(root, ss);

return ss.str();

}

TreeNode\* build(stringstream &ss)

{

char ch;

TreeNode \*r = NULL;

ss>>ch;

if(ch=='\*')

{

return NULL;

}

else if(ch=='n')

{

int x;

ss>>x;

r = new TreeNode(x);

r->left = build(ss);

r->right= build(ss);

}

return r;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

stringstream ss(data);

return build(ss);

}

};

// Your Codec object will be instantiated and called as such:

// Codec codec;

// codec.deserialize(codec.serialize(root));

解法2：

class Codec {

public:

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

if (root == nullptr) return "[]";

queue<TreeNode\*> q;

string ans="[" + to\_string(root->val) ;

q.push(root);

while(!q.empty())

{

TreeNode\* temp = q.front();

q.pop();

string val = "null";

if(temp->left != nullptr)

{

q.push(temp->left);

val = to\_string(temp->left->val);

}

ans += "," + val;

val = "null";

if(temp->right != nullptr)

{

q.push(temp->right);

val = to\_string(temp->right->val);

}

ans += "," + val;

}

ans += "]";

return ans;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

if(data.length()<=2) return nullptr;

TreeNode\* head = nullptr;

bool flag = false;

queue<TreeNode\*> q;

string temp;

int count = 0;

TreeNode\* ptr = nullptr;

for(int i=1;i<data.length()-2;i++)

{

if(data[i] != ',')

{

temp += data[i];

}

else

{

count++;

if(count %2 == 0)

{

ptr = q.front();

q.pop();

count = 0;

}

if(temp == "null")

{

if(count%2 == 0)

{

ptr->left = nullptr;

}

else

{

ptr->right = nullptr;

}

}

else

{

int val = atoi(temp.c\_str());

TreeNode\* node = new TreeNode(val);

if(!flag)

{

head = node;

flag = true;

}

q.push(node);

if(ptr != nullptr){

if(count%2 == 0)

{

ptr->left = node;

}

else

{

ptr->right = node;

}

}

}

temp = "";

}

}

return head;

}

};

解法3：

class Codec {

public:

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

vector<TreeNode\*> list;

list.push\_back(root);

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

if(list[i]){

list.push\_back(list[i]->left);

list.push\_back(list[i]->right);

}

}

string res = "";

int length = list.size();

if(length<1){

return res;

}

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

if(list[i]){

res+=to\_string(list[i]->val)+",";

}

else{

res+="n,";

}

}

res.pop\_back();

return res;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

char str[30];

vector<TreeNode\*> list;

int length = data.size();

if(length<1){

return nullptr;

}

int j = 0;

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

if(data[i] == ','){

str[j] = '\0';

j = 0;

if(str[0] == 'n'){

list.push\_back(nullptr);

}

else{

list.push\_back(new TreeNode(atoi(str)));

}

}

else{

str[j] = data[i];

j++;

}

}

str[j] = '\0';

if(str[0] == 'n'){

list.push\_back(nullptr);

}

else{

list.push\_back(new TreeNode(atoi(str)));

}

int lengthl = list.size();

int index = 1;

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

while(list[i] == nullptr&&i<lengthl){

i++;

}

if(i>=lengthl){

break;

}

list[i]->left = list[index];

list[i]->right = list[index+1];

index+=2;

}

return list[0];

}

};

## 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。

解法1：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool isValidBST(TreeNode\* root) {

if(!root)

return true;

//1、左子树上的所有节点的值都不大于根节点

//2、右子树上的所有节点的值都大于根节点

if(IsLess(root->left,root->val) && IsGreater(root->right,root->val))

//递归判断左右子树是否都是二叉搜索树

return isValidBST(root->left)&&isValidBST(root->right);

else

return false;

}

//判断以node为根节点的二叉树上的所有节点是否都小于等于根节点

bool IsLess(TreeNode \*node,int val)

{

if(!node)

return true;

if(node->val >= val)

return false;

//递归判断左右子树上的所有节点值是否都小于等于根节点

return IsLess(node->left,val)&&IsLess(node->right,val);

}

//判断以node为根节点的二叉树上的所有节点是否都大于根节点

bool IsGreater(TreeNode \*node,int val)

{

if(!node)

return true;

if(node->val <= val)

return false;

//递归判断左右子树上的所有节点值是否都大于根节点

return IsGreater(node->left,val)&&IsGreater(node->right,val);

}

};

解法2：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（**没太看懂呢？？**）

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);

}

};

解法3：因为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。

class Solution {

public:

bool isValidBST(TreeNode \*root) {

if (!root) return true;

vector<int> vals;

inorder(root, vals);

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

if (vals[i] >= vals[i + 1]) return false;

}

return true;

}

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

if (!root) return;

inorder(root->left, vals);

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

inorder(root->right, vals);

}

};

解法4：LeetCode上运行较快的示例代码

class Solution {

public:

bool isValidBST(TreeNode\* root) {

if(root == NULL)

return true;

return visitTree(root);

}

bool visitTree(TreeNode\* root){

if(root->left)

if(!visitTree(root->left)){

return false;

}

if(lastNum >= (long long)root->val){

return false;

}

lastNum = root->val;

if(root->right)

if(!visitTree(root->right)){

return false;

}

return true;

}

private:

long long lastNum = (long long)INT\_MIN - 1;

};

## 树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

struct TreeNode

{

int val;

TreeNode \*left,\*right;

TreeNode(int x):val(x),left(NULL),right(NULL){}

};

bool HasSubtree(TreeNode \*root1, TreeNode \*root2)

{

//只有当两个数均不为空时才进行比较

if(!root2 || !root1)

return false;

bool res = false;

//如果找到tree2根节点对应的点

if(root1->val == root2->val)

res = IsSubtree(root1,root2);//以这个点为起点判断是否包含tree2

//如果还没找到，就以该节点的左孩子节点为起点，判断是否包含tree2

if(!res)

res = HasSubtree(root->left,root2);

//如果还没找到，就以该节点的右孩子节点为起点，判断是否包含tree2

if(!res)

res = HasSubtree(root->right,root2);

return res;

}

bool IsSubtree(TreeNode \*root1,TreeNode \*root2)

{

//tree2比较完了都能对应上，返回true

if(!root2)

return true;

//tree2还没比较完，但tree1已经结束了，返回false

if(!root1)

return false;

//如果其中一个节点没对应上，返回false

if(root1->val != root2->val)

return false;

//如果根节点对应上了，分别去子节点里去比较

return IsSubtree(root1->left, root2->left) && IsSubtree(root1->right,root2->right);

}

## 二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。

/\*

struct TreeNode {

int val;

struct TreeNode \*left;

struct TreeNode \*right;

TreeNode(int x) :

val(x), left(NULL), right(NULL) {

}

};\*/

解法1：先序遍历这棵树的每个结点，如果遍历到的结点有子结点，则交换它的两个子结点。当交换完所有非叶子结点的左右子结点之后，就得到了树的镜像。

class Solution {

public:

void Mirror(TreeNode \*pRoot) {

if(!pRoot)

return;

if(!pRoot->left && !pRoot->right)

return;

swap(&pRoot->left,&pRoot->right);

Mirror(pRoot->left);

Mirror(pRoot->right);

}

void swap(TreeNode \*\*node1, TreeNode \*\*node2)

{

TreeNode \*t = \*node1;

\*node1 = \*node2;

\*node2 = t;

}

};

## 计数质数

统计所有小于非负整数 n 的质数的数量。

示例:

输入: 10

输出: 4

解释: 小于 10 的质数一共有 4 个, 它们是 2, 3, 5, 7 。

解法1：第一次筛选2的倍数的数字，将其都筛选出去，第二轮筛选3的倍数的数字，筛选后，剩下的第一个数字就是5（因为4在第一次筛选的时候作为2的倍数已经筛出去）第三轮则筛选5倍数的数字，第四轮7倍数，第五轮11倍数……依次筛选下去，筛n轮。

class Solution {

public:

int countPrimes(int n) {

if(n-1 < 2)

return 0;

int iCount = 0;

int \*flags = new int[n];

memset(flags,0,sizeof(int)\*n);

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

{

if(flags[i] == 0)

{

++iCount;

for(int j = 1; j \* i < n; ++j)

flags[j\*i] = 1;

}

}

delete [] flags;

flags = NULL;

return iCount;

}

};

解法2：

class Solution {

public:

int countPrimes(int n) {

int sum = 0;

if(n==0 || n==1) return 0;

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

if(isPrime(i)) sum++;

}

return sum;

}

bool isPrime(int num)

{

if(num == 2|| num == 3 )

return true ;

//不在6的倍数两侧的一定不是质数

if(num %6 != 1 && num %6 != 5)

return false ;

int tmp =sqrt( num);

//在6的倍数两侧的也可能不是质数

for(int i = 5;i <= tmp; i += 6 )

if(num %i == 0||num %(i+ 2) == 0 )

return false ;

//排除所有，剩余的是质数

return true ;

}

};

## 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。

示例 1:

输入: "()" 输入: "()[]{}" 输入: "(]" 输入: "([)]" 输入: "{[]}"

输出: true 输出: true 输出: false 输出: false 输出: true

class Solution{

public:

bool isValidParentheses(string s)

{

If(s.size() == 0)

return true;

stack<char> tmp;

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

{

If(tmp.empty())

tmp.push(s[i]);

else if((tmp.top()==’(’ && s[i]==’)’) ||

(tmp.top()==’[’ && s[i]==’]’) ||

(tmp.top()==’{’ && s[i]==’}’) )

tmp.pop();

else

tmp.push(s[i]);

}

return tmp.empty();

}

};

## 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）。

示例 :

输入: 11

输出: 3

解释: 整数 11 的二进制表示为 00000000000000000000000000001011

示例 2:

输入: 128

输出: 1

解释: 整数 128 的二进制表示为 00000000000000000000000010000000

class Solution {

public:

int hammingWeight(uint32\_t n) {

int iCount = 0;

while(n > 0)

{

++iCount;

n = n & (n-1);

}

return iCount;

}

};

## 颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

示例:

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 。

class Solution {

public:

uint32\_t reverseBits(uint32\_t n) {

uint32\_t res = 0;

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

{

/\*这个更好理解\*/

res << 1;

if(1 == (n&1))

++res;

n >>= 1;

/\*res每次左移一位再异或上n的最低位

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

n >>= 1;//去掉最低位\*/

}

return res;

}

};

## LeetCode编译报错原因分析

**LeetCode编译报错 -- reference binding to null pointer of type 'value\_type'**

**原因1**:当函数返回值为vector<vector<DataType>>类型时，eg: vector<vector<int>> generate(int numRows)。对于这个返回值vector表示的二维数组要先申请大小：

vector<vector<int>> res(**numsRows**)

原因2：下标越界

## 长度最小的子数组

给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 ≥ s 的长度最小的连续子数组。如果不存在符合条件的连续子数组，返回 0。

示例:

输入: s = 7, nums = [2,3,1,2,4,3]

输出: 2

解释: 子数组 [4,3] 是该条件下的长度最小的连续子数组。

进阶:

如果你已经完成了O(n) 时间复杂度的解法, 请尝试 O(n log n) 时间复杂度的解法。

class Solution {

public:

int minSubArrayLen(int s, vector<int>& nums) {

if(nums.size() == 0)

return 0;

int iMinLen = nums.size() + 1;

int i= 0, j = 0, sum = 0;

while(i < nums.size())

{

if(j < nums.size() && sum < s)

sum += nums[j++];

else

sum -= nums[i++];

if(sum >= s)

iMinLen = min(iMinLen, j - i);

}

if(iMinLen == nums.size() + 1)

return 0;

return iMinLen;

}

};

## 最大连续1的个数

给定一个二进制数组， 计算其中最大连续1的个数。

示例 1:

输入: [1,1,0,1,1,1]

输出: 3

解释: 开头的两位和最后的三位都是连续1，所以最大连续1的个数是 3.

注意：

输入的数组只包含 0 和1。

输入数组的长度是正整数，且不超过 10,000。

class Solution {

public:

int findMaxConsecutiveOnes(vector<int>& nums) {

if(nums.size() == 0)

return 0;

int iMaxSize = 0,iCount = 0;

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

{

if(nums[i] == 0)

{

if(iCount > iMaxSize)

iMaxSize = iCount;

iCount = 0;

}

else

++iCount;

}

return max(iMaxSize, iCount);

}

};

## 链表中的双指针解法

### 环形链表，并找到环的起点

定义两个指针，分别为快指针fast和慢指针slow，快指针每次移动两个节点，慢指针每次移动一个节点，如果链表中存在环，则两个节点必定会相遇（fast == slow）;此时将快指针指回到首节点位置，然后同时移动快慢指针（此时以相同的速度移动），直到两个指针相遇（fast == slow），该位置节点即为环的起点。

### 相交链表（两个链表的第一个公共节点）

1）首先应该明确，对于单链表，如果相交（存在第一个公共节点），那么两个链表相交应该呈Y型（如下图所示），而不是X型。

A: a1 → a2

↘

c1 → c2 → c3

↗

B: b1 → b2 → b3

根据两个链表长度的差值a，将较长的链表头指针先移动距离a，然后以相同速度同时移动两个指针，一旦两个指针相等，即为两个链表的第一个公共节点。

**2）不需要算长度的算法：**

class Solution {

public:

    ListNode\* FindFirstCommonNode( ListNode \*pHead1, ListNode \*pHead2) {

        ListNode \*p1 = pHead1;

        ListNode \*p2 = pHead2;

        while(p1!=p2){

            p1 = (p1==NULL ? pHead2 : p1->next);

            p2 = (p2==NULL ? pHead1 : p2->next);

        }

        return p1;

    }

};

### 删除链表中倒数第N个节点（1<=N<=链表长）

定义两个指针（暂且叫做前、后指针，初始都指向首节点），先将前指针向前移动N，如果此时前指针为空，则待删除节点为首节点，结束；否则继续同时移动前、后指针，直到前指针移动到最后一个节点位置，此时后指针所指节点的下一个节点即为待删除节点。

## 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。

/\*\*

\* Definition for singly-linked list with a random pointer.

\* struct RandomListNode {

\* int label;

\* RandomListNode \*next, \*random;

\* RandomListNode(int x) : label(x), next(NULL), random(NULL) {}

\* };

\*/

解法1：空间复杂度：O（1）

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if(!head)

return NULL;

RandomListNode \*curNode = head;

//在原链表的每个节点后面拷贝一个新的节点

while(curNode)

{

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

tmp->next = curNode->next;

curNode->next = tmp;

curNode = tmp->next;

}

//给新的节点的random赋值

curNode = head;

while(curNode)

{

if(curNode->random)

curNode->next->random = curNode->random->next;

curNode = curNode->next->next;

}

//断开新旧节点，取出新链表

curNode = head;

RandomListNode \*newHead = curNode->next;

while(curNode)

{

RandomListNode \*tmp = curNode->next;

curNode->next = tmp->next;

if(tmp->next)

tmp->next = tmp->next->next;

curNode = curNode->next;

}

return newHead;

}

};

解法2：

class Solution{

public:

RandomListNode\* copyRandomList(RandomListNode\* head)

{

if(!head)

return;

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

RandomListNode \*newCurNode = newHead, \*curNode = head->next;

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

m[head] = newHead;

while(curNode)

{

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

newCurNode->next = tmp;

m[curNode] = tmp;

curNode = curNode->next;

newCurNode = newCurNode->next;

}

curNode = head;

newCurNode = newHead;

while(newCurNode)

{

newCurNode->random = m[curNode->random];

newCurNode = newCurNode->next;

curNode = curNode->next;

}

}

}

## 3的幂/4的幂

给定一个整数，写一个函数来判断它是否是 3/4 的幂次方。

解法1：

bool isPowerOfThree(int n) {

if (n <= 0)

return false;

double tmp = log10(n) / log10(3);

if (tmp - int(tmp) == 0)

return true;

return false;

}

注：4的幂只要将log10(3)改为log10(4)

## 有效的完全平方数

给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 True，否则返回 False。

解法1：二分查找

bool isPerfectSquare(int num) {

if(num <= 0)

return false;

long long left = 1,right = num;

while(left <= right)

{

long long mid = left + (right-left)/2;

long long t = mid\*mid;

if(t == num)

return true;

else if(t > num)

right = mid - 1;

else

left = mid + 1;

}

return false;

}

解法2：任意完全平方数都可以表示成连续的奇数和

bool isPerfectSquare(int num) {

int i = 1;

while(num>0)

{

num -= i;

i += 2;

}

return num == 0;

}

## 两整数之和

不使用运算符 + 和 - ​​​​​​​，计算两整数 ​​​​​​​a 、b ​​​​​​​之和。

int getSum(int a, int b) {

if(b == 0)

return a;

int sum = a ^ b;//相加但不进位

int carry = (a & b) << 1;//进位但不相加

return getSum(sum,carry);

}