# 一、剑指offer

2019/10/1

## 1、二维数组中的查找

在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

### （1）首答：左上角开始遍历

public class Solution {

public boolean Find(int target, int [][] array) {

if(array == null || array.length == 0 ||(array.length == 1 && array[0].length == 0)) return false;

int row = array.length;

int col = array[0].length;

boolean found = false;

for(int i=0; i<row && found != true; i++){

if(array[i][0] <= target){

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

if(array[i][j] == target){

found = true;

break;

}

if(array[i][j] > target) break;

}

}

else{

break;

}

}

return found;

}

}

运行时间：134ms

占用内存：16796k

时间复杂度：

空间复杂度：

### （2）思路二：左下角开始遍历

最佳答案：没有之一。思路：首先我们选择从左下角开始搜寻，(为什么不从左上角开始搜寻，左上角向右和向下都是递增，那么对于一个点，对于向右和向下会产生一个岔路；如果我们选择从左下脚开始搜寻的话，如果大于就向右，如果小于就向下)。

public class Solution {

    public boolean Find(int [][] array,int target) {

int len = array.length-1;

        int i = 0;

        while((len >= 0)&& (i < array[0].length)){

            if(array[len][i] > target){

                len--;

            }else if(array[len][i] < target){

                i++;

            }else{

                return true;

            }

        }

        return false;

    }

}

运行时间：143ms

占用内存：17424k

时间复杂度：

空间复杂度：

## 2、字符串替换空格

请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

### （1）首答：从前往后遍历，好写但是效率低

public class Solution {

public String replaceSpace(StringBuffer str) {

int n = str.length();

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

if(str.charAt(i) == ' '){

str.setCharAt(i, '%');

str.insert(i+1, 20);

i += 2;

n += 2;

}

}

return str.toString();

}

}

运行时间：19ms

占用内存：9308k

时间复杂度：

空间复杂度：

### （2）优化：从后往前遍历

小白：没看出来从后往前处理的方式有何用，跟前到后有啥区别？

小黑：从后往前，每个空格后面的字符只需要移动一次。从前往后，当遇到第一个空格时，要移动第一个空格后所有的字符一次；当遇到第二个空格时，要移动第二个空格后所有的字符一次；以此类推。所以总的移动次数会更多。

public class Solution {

public String replaceSpace(StringBuffer str) {

int n = str.length();

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

if(str.charAt(i) == ' '){

str.setCharAt(i, '%');

str.insert(i+1, 20);

}

}

return str.toString();

}

}

运行时间：18ms

占用内存：9676k

时间复杂度：

空间复杂度：

## 3、反转链表

### （1）递归

class Solution {

public ListNode reverseList(ListNode head) {

if(head == null || head.next == null) return head;

ListNode p = reverseList(head.next);

head.next.next = head;

head.next = null;

return p;

}

}

时间复杂度：

空间复杂度：

### （2）非递归

class Solution {

public ListNode reverseList(ListNode head) {

if(head == null) return head;

ListNode pre = null;

ListNode next = null;

while(head != null){

next = head.next;

head.next = pre;

pre = head;

head = next;

}

return pre;

}

}

时间复杂度：

空间复杂度：

## 4、从尾到头打印链表

输入一个链表，按链表从尾到头的顺序返回一个ArrayList。

### （1）首答：先反转链表，再遍历

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> list = new ArrayList<Integer>();

ListNode root = reverseList(listNode);

if(root == null) return list;

while(root != null){

list.add(root.val);

root = root.next;

}

return list;

}

public ListNode reverseList(ListNode head){

if(head == null || head.next == null) return head;

ListNode p = reverseList(head.next);

head.next.next = head;

head.next = null;

return p;

}

}

运行时间：18ms

占用内存：9452k

时间复杂度：，虽然需要遍历两遍，但复杂度不随n变化，仍为

空间复杂度：

### （2）优化：直接一次递归

这个直接递归需要在方法外定义一个ArrayList，这是一次遍历的关键，如果在方法内定义反而不好用递归，因为会涉及到递归定义的情况。

import java.util.ArrayList;

public class Solution {

ArrayList<Integer> arrayList = new ArrayList<Integer>();

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

if(listNode != null){

printListFromTailToHead(listNode.next);

arrayList.add(listNode.val);

}

return arrayList;

}

}

运行时间：18ms

占用内存：9392k

时间复杂度：，比起上个方法的优势在于只需遍历一遍。

空间复杂度：

### （3）利用栈实现反向

很容易想到可以用栈来实现反向打印，但是由于这种方式同样会进行两次遍历，但是占用的内存资源更多，所以是三种方式中最差的。

import java.util.ArrayList;

import java.util.Stack;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> arrayList = new ArrayList<Integer>();

if(listNode == null) return arrayList;

Stack<Integer> s = new Stack<Integer>();

while(listNode != null){

s.push(listNode.val);

listNode = listNode.next;

}

while(s.empty() != true){

arrayList.add(s.pop());

}

return arrayList;

}

}

运行时间：20ms

占用内存：9272k

时间复杂度：，比起上个方法的优势在于只需遍历一遍。

空间复杂度：

## 5、最小的k个数（topN问题）

2019/10/6 Special day!

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

### （1）法一：快速排序

可以将这n个数排序，最终得到排序后的前k个数即为所求 。

//法一：快速排序法

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) {

int length = input.length;

ArrayList<Integer> list = new ArrayList<Integer>();

if(input == null || length == 0 || k > length) return list;

quickSort(input, 0, length-1);

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

list.add(input[i]);

}

return list;

}

public void quickSort(int[] array, int start, int end){

if(start < end){

int key = array[start];

int i = start;

for(int j=start+1; j<=end; j++){

if(array[j] < key){

swap(array, j, ++i);

}

}

array[start] = array[i];

array[i] = key;

quickSort(array, start, i-1);

quickSort(array, i+1, end);

}

}

public void swap(int[] A, int a, int b){

if(a != b){

A[a] ^= A[b];

A[b] ^= A[a];

A[a] ^= A[b];

}

}

}

运行时间：21ms

占用内存：9692k

时间复杂度：

空间复杂度：

### （2）法二：利用patition函数

我们可以利用快速排序的思路解决这个问题。如果基于数组的第K个数字来调整，则使得比第K个数字小的所有数字 都位于数组的左边，比第K个数字大的都位于数组的右边。这样调整后，位于数组中左边K个数字就是最小的k个数字（这k个数字不一定是排序的）。这个思路的时间复杂度是O(n)，但是需要修改输入的数组。

//法二：使用Patition函数

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) {

int length = input.length;

ArrayList<Integer> list = new ArrayList<Integer>();

if(input == null || length == 0 || k > length) return list;

int start = 0;

int end = length - 1;

int index = patition(input, start, end);

while(index != k-1){

if(index > k-1){

end = index - 1;

index = patition(input, start, end);

}

else{

start = index + 1;

index = patition(input, start, end);

}

}

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

list.add(input[i]);

}

return list;

}

public int patition(int[] array, int start, int end){

int pivot = array[start];

while(start < end){

while(start < end && array[end] >= pivot){

--end;

}

swap(array, start, end);

while(start < end && array[start] <= pivot){

++start;

}

swap(array, start, end);

}

return start;

}

public void swap(int[] A, int a, int b){

if(a != b){

A[a] ^= A[b];

A[b] ^= A[a];

A[a] ^= A[b];

}

}

}

时间复杂度：

空间复杂度：

### （3）法三：建大顶堆

使用大小为k的容器，将数组前k个数先存放进去，然后遍历数组中的数和每次容器中的最大值比较，如果数组中的值小，则替换容器中的最大值   
\*   
\* 容器可以是最大堆，可以是ArrayList,也可以是红黑树   
\*   
\*/

详细思路：

我们可以先创建一个大小为k的数据容器来存储最小的k个数字，接下来每次从输入的n个整数中读入一个数。如果容器中已有的数字少于k个，则直接把这次读入的整数放入到容器中；如果容器中已经有k个数字了，此时需要替换已有的数字。找出这已有的k个数字中的最大值，然后拿这次待插入的整数和最大值进行比较，如果比当前的最大值小，则替换已有的最大值；如果大的话，可以抛弃这个整数。我们可以使用最大堆（堆排序），也可以使用红黑树来完成这个容器的创建。

这种思路的时间复杂度是O(nlogk)。这种解法虽然慢一些，但是有两个明显的优点：（1）没有修改输入的数据（2）该算法非常适合海量数据的输入。

关于海量数据处理

假设题目要求是从海量的数据中造出最小的k个数字，由于内存的大小是有限的，有可能不能把这些海量数据一次性全部载入内存。这个时候，我们可以从辅助存储空间（如硬盘）中每次读入一个数字，根据最小的k个数，判断这些数据是不是需要放入容器中，这种思路只要求内存能容下k个数即可。因此，它最适合的情形是n很大并且k较小的问题。

//法三：建大顶堆

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) {

int length = input.length;

ArrayList<Integer> list = new ArrayList<Integer>();

if(input == null || length == 0 || k > length) return list;

buildMaxHeap(input, k);

for(int j=k; j<length; j++){

int maxValue = input[0];

if(input[j] < maxValue){

swap(input, j, 0);

heapAdjust(input, 0, k);

}

}

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

list.add(input[i]);

}

return list;

}

public void buildMaxHeap(int[] array, int heapSize){

for(int i = (heapSize - 2) >> 1; i>=0; i--){

heapAdjust(array, i, heapSize);

}

}

public void heapAdjust(int[] array, int i, int heapSize){

int left = 2 \* i + 1;

int right = 2 \* i + 2;

int largest = i;

if(left < heapSize && array[left] > array[largest]){

largest = left;

}

if(right < heapSize && array[right] > array[largest]){

largest = right;

}

if(largest != i){

swap(array, i, largest);

heapAdjust(array, largest, heapSize);

}

}

public void swap(int[] A, int a, int b){

if(a != b){

A[a] ^= A[b];

A[b] ^= A[a];

A[a] ^= A[b];

}

}

}

运行时间：24ms

占用内存：9536k

时间复杂度：

空间复杂度：

## 6、求1+2+3+…+n

求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

### （1）法一：短路求值定理

解题思路：

1.需利用逻辑与的短路特性实现递归终止。 2.当n==0时，(n>0)&&((sum+=Sum\_Solution(n-1))>0)只执行前面的判断，为false，然后直接返回0；

3.当n>0时，执行sum+=Sum\_Solution(n-1)，实现递归计算Sum\_Solution(n)。

public class Solution {

public int Sum\_Solution(int n) {

int ans = n;

boolean t = ((ans!=0) && ((ans += Sum\_Solution(n-1))!=0));

return ans;

}

}

运行时间：14ms

占用内存：9304k

时间复杂度：

空间复杂度：

### （2）法二：利用sizeof（）函数

用公式是不可以的，公式里有乘法！！实现乘法可以用sizeof多维数组，两行代码就可以

class Solution {

public:

int Sum\_Solution(int n) {

bool a[n][n+1];

return sizeof(a)>>1;

}

};

时间复杂度：

空间复杂度：

## 7、滑动窗口最大值

2019/10/8

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

### （1）法一：用数组实现

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> maxInWindows(int [] num, int size)

{

int length = num.length;

ArrayList<Integer> list = new ArrayList<Integer>();

if(num == null || size > length || size <= 0){

return list;

}

int[] result = new int[2];

result = findMax(num, 0, size-1);

list.add(result[0]);

for(int j=1; j+size-1 < length; j++){

if(num[j+size-1] > result[0]){

result[0] = num[j+size-1];

result[1] = j+size-1;

}

else if(result[1] < j){

result = findMax(num, j, j+size-1);

}

list.add(result[0]);

}

return list;

}

public int[] findMax(int[] array, int start, int end){

int[] arr = new int[2];

int maxValue = array[start];

int maxIndex = start;

for(int i=start+1; i<=end; i++){

if(array[i] > maxValue){

maxValue = array[i];

maxIndex = i;

}

}

arr[0] = maxValue;

arr[1] = maxIndex;

return arr;

}

}

运行时间：17ms

占用内存：9252k

时间复杂度：

空间复杂度：

### （2）法二：用双端队列

注意点：

双端队列里保存的是下标，begin > q.peekFirst()是为了判断滑动一次后，是不是把最大值划出去了。

begin是用于保存当前窗口的第一个值在原始数组中的【下标】，这样写的目的是为了少写一些判断边界条件的代码。

import java.util.\*;

/\*\*

用一个双端队列，队列第一个位置保存当前窗口的最大值，当窗口滑动一次

1.判断当前最大值是否过期

2.新增加的值从队尾开始比较，把所有比他小的值丢掉

\*/

public class Solution {

public ArrayList<Integer> maxInWindows(int [] num, int size)

{

ArrayList<Integer> res = new ArrayList<>();

if(size == 0) return res;

int begin;

ArrayDeque<Integer> q = new ArrayDeque<>();

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

begin = i - size + 1;

if(q.isEmpty())

q.add(i);

else if(begin > q.peekFirst())

q.pollFirst();

while((!q.isEmpty()) && num[q.peekLast()] <= num[i])

q.pollLast();

q.add(i);

if(begin >= 0)

res.add(num[q.peekFirst()]);

}

return res;

}

}

运行时间：19ms

占用内存：9224k

时间复杂度：

空间复杂度：

## 8、数据流中的中位数

2019/10/8

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。

### （1）法一：PriorityQueue来设置一个小顶堆和大顶堆

解题思路就是用两个堆，一个大顶堆，一个小顶堆来过滤数据。

**解题思路**

* 先用java集合PriorityQueue来设置一个小顶堆和大顶堆
* 主要的思想是：因为要求的是中位数，那么这两个堆，**大顶堆用来存较小的数，从大到小排列**；
* *小顶堆存较大的数，从小到大的顺序排序\**，显然中位数就是大顶堆的根节点与小顶堆的根节点和的平均数。
* ⭐保证：小顶堆中的元素都大于等于大顶堆中的元素，所以每次塞值，并不是直接塞进去，而是从另一个堆中poll出一个最大（最小）的塞值
* ⭐当数目为偶数的时候，将这个值插入大顶堆中，再将大顶堆中根节点（即最大值）插入到小顶堆中；
* ⭐当数目为奇数的时候，将这个值插入小顶堆中，再讲小顶堆中根节点（即最小值）插入到大顶堆中；
* ⭐取中位数的时候，如果当前个数为偶数，显然是取小顶堆和大顶堆根结点的平均值；如果当前个数为奇数，显然是取小顶堆的根节点

理解了上面所述的主体思想，下面举个例子辅助验证一下。

例如，传入的数据为：[5,2,3,4,1,6,7,0,8],那么按照要求，输出是"5.00 3.50 3.00 3.50 3.00 3.50 4.00 3.50 4.00 "

那么整个程序的执行流程应该是（用min表示小顶堆，max表示大顶堆）：

* 5先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，此时min=[5],max=[无]，avg=[5.00]
* 2先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，此时min=[5],max=[2],avg=[(5+2)/2]=[3.50]
* 3先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，此时min=[3,5],max=[2],avg=[3.00]
* 4先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，此时min=[4,5],max=[3,2],avg=[(4+3)/2]=[3.50]
* 1先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，此时min=[3,4,5],max=[2,1]，avg=[3/00]
* 6先进入小顶堆，然后将小顶堆中最小值放入大顶堆中，此时min=[4,5,6],max=[3,2,1],avg=[(4+3)/2]=[3.50]
* 7先进入大顶堆，然后将大顶堆中最大值放入小顶堆中，此时min=[4,5,6,7],max=[3,2,1],avg=[4]=[4.00]
* 0先进入小顶堆，然后将小顶堆中最大值放入小顶堆中，此时min=[4,5,6,7],max=[3,2,1,0],avg=[(4+3)/2]=[3.50]
* 8先进入大顶堆，然后将大顶堆中最小值放入大顶堆中，此时min=[4,5,6,7,8],max=[3,2,1,0],avg=[4.00]

import java.util.PriorityQueue;

import java.util.Comparator;

public class Solution {

//小顶堆

private PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

//大顶堆

private PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(15, new Comparator<Integer>(){

@Override

public int compare(Integer o1, Integer o2){

return o2 - o1;

}

});

//记录偶数个还是奇数个

int count = 0;

//每次插入小顶堆的是当前大顶堆中最大的数

//每次插入大顶堆的是当前小顶堆中最小的数

//这样保证小顶堆中的数永远大于等于大顶堆中的数

//中位数就可以方便地从两者的根结点中获取了

public void Insert(Integer num) {

//总记录数为偶数的话，则先插入到大顶堆，然后将大顶堆中最大的数插入到小顶堆中

if(count % 2 == 0){

maxHeap.offer(num);

int max = maxHeap.poll();

minHeap.offer(max);

}

//总记录数为奇数的话，则先插入到小顶堆，然后将小顶堆中最小的数插入到大顶堆中

else{

minHeap.offer(num);

int min = minHeap.poll();

maxHeap.offer(min);

}

count++;

}

public Double GetMedian() {

//当前为偶数个，则取小顶堆和大顶堆的堆顶元素求平均

if(count % 2 == 0){

return new Double(minHeap.peek() + maxHeap.peek())/2;

}else{

//当前为奇数个，则直接从小顶堆中取元素即可

return new Double(minHeap.peek());

}

}

}

运行时间：22ms

占用内存：9668k

时间复杂度：

空间复杂度：

Q&A

为什么奇偶的时候要分别插入到大小顶堆中进行过滤？只插入到某一个顶堆中过滤不可以吗？  
只插入到一个顶堆中的话，如果快速地拿到中位数呢？因为输入长度也不确定，插入到顶堆再去遍历的话复杂度又高；这个方法的巧妙在于用了PriorityQueue，本质上是一个优先队列，可以分为最大优先队列和最小优先队列（这里就相当于排序了的大顶堆或小顶堆），优先队列的一个最大特性就是，当插入元素或者删除元素的时候，队列会自动进行调整，保证队首元素一定是优先权最大/最小。这个解法，保证最大优先队列取出的值是按照从大到小顺序排列，最小优先队列反之，那么我求中位数的时候，就特别简单了，偶数的时候直接取两个队列的头元素再一平均就好了，奇数的时候从其中一个队列中取头元素就好了。不知道有没有说清楚

## 9、连续子数组的最大和

2019/10/8

HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)。

### （1）思路一：分治法

解题思路：该题即求连续子段的最大和，采用分治法。

详见《算法分析与设计学习笔记》分治法一节。

public class Solution {

public int FindGreatestSumOfSubArray(int[] array) {

if(array == null || array.length == 0) return 0;

int length = array.length;

return maxSum(array, 0, length-1);

}

public int maxSum(int[] array, int left, int right){

if(left == right){

return array[left];

}

int mid = (left + right) / 2;

int sumLeft = maxSum(array, left, mid);

int sumRight = maxSum(array, mid+1, right);

int s1 = array[mid], lefts = array[mid];

for(int i=mid-1; i>=left; i--){

lefts += array[i];

if(lefts > s1){

s1 = lefts;

}

}

int s2 = array[mid+1], rights = array[mid+1];

for(int j=mid+2; j<=right; j++){

rights += array[j];

if(rights > s2){

s2 = rights;

}

}

int sum = s1 + s2;

if(sumLeft > sum){

sum = sumLeft;

}

if(sumRight > sum){

sum = sumRight;

}

return sum;

}

}

运行时间：13ms

占用内存：9360k

时间复杂度：

空间复杂度：

### （2）思路二：动态规划法

使用动态规划

F（i）：以array[i]为末尾元素的子数组的和的最大值，子数组的元素的相对位置不变

F（i）=max（F（i-1）+array[i] ， array[i]）

res：所有子数组的和的最大值

res=max（res，F（i））

如数组[6, -3, -2, 7, -15, 1, 2, 2]

初始状态：

    F（0）=6

    res=6

i=1：

    F（1）=max（F（0）-3，-3）=max（6-3，3）=3

    res=max（F（1），res）=max（3，6）=6

i=2：

    F（2）=max（F（1）-2，-2）=max（3-2，-2）=1

    res=max（F（2），res）=max（1，6）=6

i=3：

    F（3）=max（F（2）+7，7）=max（1+7，7）=8

    res=max（F（2），res）=max（8，6）=8

i=4：

    F（4）=max（F（3）-15，-15）=max（8-15，-15）=-7

    res=max（F（4），res）=max（-7，8）=8

以此类推

最终res的值为8

推导思路：

dp[i]表示以元素array[i]结尾的最大连续子数组和.

以[-2,-3,4,-1,-2,1,5,-3]为例

可以发现,

dp[0] = -2

dp[1] = -3

dp[2] = 4

dp[3] = 3

以此类推,会发现

dp[i] = max{dp[i-1]+array[i],array[i]}.

public class Solution {

public int FindGreatestSumOfSubArray(int[] array) {

int max = array[0]; //以array[i]为末尾元素的子数组的和的最大值

int res = array[0]; //当前所有子数组的和的最大值

for(int i=1; i<array.length; i++){

max = Math.max(max + array[i], array[i]);

res = Math.max(res, max);

}

return res;

}

}

运行时间：15ms

占用内存：9412k

时间复杂度：

空间复杂度：

## 10、重建二叉树

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

TreeNode\* reConstructBinaryTree(vector<int> pre, vector<int> vin) {

int inLen = vin.size();

if (inLen == 0) {

return NULL;

}

vector<int> left\_pre, right\_pre, left\_in, right\_in;

//创建根结点，根结点肯定是前序遍历的第一个数

TreeNode \*head = new TreeNode(pre[0]);

//找到中序遍历根结点所在位置，存放于变量gen中

int gen = 0;

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

if (vin[i] == pre[0]) {

gen = i;

break;

}

}

//对于中序遍历，根结点左边的结点位于根结点左边，根结点右边的结点位于根结点右边

//利用上述这点，对二叉树结点进行归并

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

left\_pre.push\_back(pre[i + 1]); //前序第一个为根结点

left\_in.push\_back(vin[i]);

}

for (int i = gen + 1; i<inLen; i++) {

right\_in.push\_back(vin[i]);

right\_pre.push\_back(pre[i]);

}

//递归取出前序和中序遍历根结点左边和右边的子树

head->left = reConstructBinaryTree(left\_pre, left\_in);

head->right = reConstructBinaryTree(right\_pre, right\_in);

return head;

}

};

运行时间：5ms

占用内存：480k

时间复杂度：

空间复杂度：

## 11、用两个栈实现队列

用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

class Solution

{

public:

void push(int node) {

while(!stack2.empty()){

stack1.push(stack2.top());

stack2.pop();

}

stack1.push(node);

}

int pop() {

while(!stack1.empty()){

stack2.push(stack1.top());

stack1.pop();

}

int temp = stack2.top();

stack2.pop();

return temp;

}

private:

stack<int> stack1;

stack<int> stack2;

};

运行时间：3ms

占用内存：476k

时间复杂度：

空间复杂度：

## 12、旋转数组的最小数字

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。  
输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。  
例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。  
NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

### （1）法一：从数组后面开始两两比较

这是比较容易想到的方法，由于旋转数组只是旋转了数组前面若干个元素，所以从数组尾部开始查找要比从数组头部查找需要的平均比较次数要少（一般情况下）。

class Solution {

public:

int minNumberInRotateArray(vector<int> rotateArray) {

if(rotateArray.empty()){

return 0;

}

int len = rotateArray.size();

for(int i=len-1; i>=1; i--){

if(rotateArray[i] < rotateArray[i-1]){

return rotateArray[i];

}

}

return rotateArray[0];

}

};

运行时间：27ms

占用内存：860k

时间复杂度：

空间复杂度：

### （2）法二：二分法

采用二分法解答这个问题，

mid = low + (high - low)/2

需要考虑三种情况：

(1)array[mid] > array[high]:

出现这种情况的array类似[3,4,5,6,0,1,2]，此时最小数字一定在mid的右边。

low = mid + 1

(2)array[mid] == array[high]:

出现这种情况的array类似 [1,0,1,1,1] 或者[1,1,1,0,1]，此时最小数字不好判断在mid左边

还是右边,这时只好一个一个试 ，

high = high - 1

(3)array[mid] < array[high]:

出现这种情况的array类似[2,2,3,4,5,6,6],此时最小数字一定就是array[mid]或者在mid的左

边。因为右边必然都是递增的。

high = mid

**注意这里有个坑：如果待查询的范围最后只剩两个数，那么mid** **一定会指向下标靠前的数字**

比如 array = [4,6]

array[low] = 4 ;array[mid] = 4 ; array[high] = 6 ;

如果high = mid - 1，就会产生错误， 因此high = mid

但情形(1)中low = mid + 1就不会错误

class Solution {

public:

int minNumberInRotateArray(vector<int> rotateArray) {

if(rotateArray.empty()){

return 0;

}

int low = 0, high = rotateArray.size() - 1;

while(low < high){

int mid = low + (high - low) / 2;

if(rotateArray[mid] > rotateArray[high]){

low = mid + 1;

}

else if(rotateArray[mid] == rotateArray[high]){

high = high - 1;

}

else{

high = mid;

}

}

return rotateArray[low];

}

};

运行时间：29ms

占用内存：596k

时间复杂度：

空间复杂度：

## 13、斐波那契数列

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。

n<=39

注意考虑n=0和n=1的情况即可。

class Solution {

public:

int Fibonacci(int n) {

if(n == 0){

return 0;

}

if(n == 1){

return 1;

}

int pre\_pre = 0;

int pre = 1;

int ret = 0;

while(n > 1){

ret = pre + pre\_pre;

pre\_pre = pre;

pre = ret;

n--;

}

return ret;

}

};

运行时间：4ms

占用内存：472k

时间复杂度：O(n)

空间复杂度：

# 二、Leetcode

## 1、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。

示例:

给定如下二叉树，以及目标和 sum = 22，

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

返回 true, 因为存在目标和为 22 的根节点到叶子节点的路径 5->4->11->2。

### （1）思路一：递归

最直接的方法就是利用递归，遍历整棵树：如果当前节点不是叶子，对它的所有孩子节点，递归调用 hasPathSum 函数，其中 sum 值减去当前节点的权值；如果当前节点是叶子，检查 sum 值是否为 0，也就是是否找到了给定的目标和。

版本一：（自己写的）

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

int totalSum = 0;

return findPath(root, sum, totalSum);

}

public boolean findPath(TreeNode root, int sum, int totalSum){

if(root == null){

return false;

}

totalSum += root.val;

boolean bool1 = findPath(root.left, sum, totalSum);

boolean bool2 = findPath(root.right, sum, totalSum);

boolean bool3 = (root.left == null && root.right == null && totalSum == sum)? true : false;

if(bool3 || bool1 || bool2){

return true;

}

else{

return false;

}

}

}

版本二：官方题解（比上个我自己写的要简练一些…）

class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if(root == null){

return false;

}

sum -= root.val;

if(root.left == null && root.right == null){

return (sum == 0);

}

return hasPathSum(root.left, sum) || hasPathSum(root.right, sum);

}

}

两个版本时间复杂度和空间复杂度均相同，分析如下：

执行用时 :1 ms, 在所有 Java 提交中击败了99.11%的用户

内存消耗 :37.2 MB, 在所有 Java 提交中击败了68.56%的用户

时间复杂度：

空间复杂度：最好情况，最坏情况

复杂度分析

时间复杂度：我们访问每个节点一次，时间复杂度为 ，其中 n 是节点个数。

空间复杂度：最坏情况下，整棵树是非平衡的，例如每个节点都只有一个孩子，递归会调用 n 次（树的高度），因此栈的空间开销是 。但在最好情况下，树是完全平衡的，高度只有，因此在这种情况下空间复杂度只有 。

### （2）思路二：非递归（迭代）

使用两个栈进行实现，其中一个栈用来放结点，一个栈放剩余目标和。

我们可以用栈将递归转成迭代的形式。深度优先搜索在除了最坏情况下都比广度优先搜索更快。最坏情况是指满足目标和的 root->leaf 路径是最后被考虑的，这种情况下深度优先搜索和广度优先搜索代价是相通的。

利用深度优先策略访问每个节点，同时更新剩余的目标和。

所以我们从包含根节点的栈开始模拟，剩余目标和为 sum - root.val。

然后开始迭代：弹出当前元素，如果当前剩余目标和为 0 并且在叶子节点上返回 True；如果剩余和不为零并且还处在非叶子节点上，将当前节点的所有孩子以及对应的剩余和压入栈中。

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if(root == null){

return false;

}

Stack<TreeNode> stackNode = new Stack<TreeNode>();

Stack<Integer> stackSum = new Stack<Integer>();

stackNode.push(root);

stackSum.push(sum - root.val);

while(!stackNode.empty()){

TreeNode node = (TreeNode)stackNode.pop();

int curSum = (Integer)stackSum.pop();

if(node.left == null && node.right == null && curSum == 0){

return true;

}

if(node.right != null){

stackNode.push(node.right);

stackSum.push(curSum - node.right.val);

}

if(node.left != null){

stackNode.push(node.left);

stackSum.push(curSum - node.left.val);

}

}

return false;

}

}

执行用时 :5 ms, 在所有 Java 提交中击败了7.77%的用户

内存消耗 :39.4 MB, 在所有 Java 提交中击败了18.84%的用户

时间复杂度：

空间复杂度：最好情况，最坏情况

复杂度分析

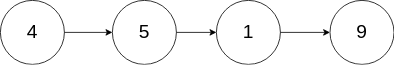
时间复杂度：和递归方法相同是 O(N)O(N)。

空间复杂度：当树不平衡的最坏情况下是 O(N)O(N) 。在最好情况（树是平衡的）下是 O(\log N)O(logN)。

小总结：比较以上两个递归和非递归的算法的运行时间，发现递归算法的运行时间居然更短一些，可见并不是非递归算法总是更快。

## 2、删除链表中的结点

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:

示例 1:

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.

示例 2:

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。

### （1）Java版本

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;;

}

}

执行用时 :0 ms, 在所有 Java 提交中击败了100.00%的用户

内存消耗 :36 MB, 在所有 Java 提交中击败了68.47%的用户

时间复杂度：

空间复杂度：

### （2）以二级指针为参数（C++版本）

void deleteLLNode(LLNode \*\*pNode)

{

if(!pNode) return;

LLNode \*temp = \*pNode;

\*pNode = temp->pNext;

delete temp;

}

## 3、移除链表元素

删除链表中等于给定值 ***val***的所有节点。

**示例:**

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

**输出:** 1->2->3->4->5

### （1）首答，很蠢

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeElements(ListNode head, int val) {

if(head == null) return head;

ListNode root = head;

while(head.val == val){

head = head.next;

root = head;

if(head == null) return head;

}

while(head.next != null){

if(head.next.val == val){

head.next = head.next.next;

}

else{

head = head.next;

}

}

return root;

}

}

执行用时 :1 ms, 在所有 Java 提交中击败了100.00%的用户

内存消耗 :39.3 MB, 在所有 Java 提交中击败了94.47%的用户

时间复杂度：

空间复杂度：

### （2）巧妙：手动添加一个头结点

前两行增加一个header，这样避免前n个node 数据为val的情况，很巧妙，学习了。

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode root = new ListNode(0);

root.next = head;

ListNode temp = root;

while(root.next != null){

if(root.next.val == val){

root.next = root.next.next;

}

else{

root = root.next;

}

}

return temp.next;

}

}

执行用时 :1 ms, 在所有 Java 提交中击败了100.00%的用户

内存消耗 :38.8 MB, 在所有 Java 提交中击败了95.02%的用户

时间复杂度：

空间复杂度：

## 4、删除排序链表中的重复元素 II

2019/10/10

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。

示例 1:

输入: 1->2->3->3->4->4->5

输出: 1->2->5

示例 2:

输入: 1->1->1->2->3

输出: 2->3

### （1）思路一：非递归

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if(head == null || head.next == null) return head;

ListNode root = new ListNode(0);

ListNode temp = root;

while(head != null){

boolean dup = false;

while(head != null && head.next != null && head.val == head.next.val){

head = head.next;

dup = true;

}

if(!dup){

root.next = head;

root = root.next;

}

head = head.next;

}

//如果链表最后以重复元素结尾

root.next = null;

return temp.next;

}

}

执行用时 :1 ms, 在所有 Java 提交中击败了99.49%的用户

内存消耗 :36.8 MB, 在所有 Java 提交中击败了60.18%的用户

时间复杂度：

空间复杂度：

### （2）思路二：递归写法

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if(head == null) return head;

if(head.next != null && head.val == head.next.val){

while(head != null && head.next != null && head.val == head.next.val){

head = head.next;

}

return deleteDuplicates(head.next);

}

else{

head.next = deleteDuplicates(head.next);

}

return head;

}

}

执行用时 :1 ms, 在所有 Java 提交中击败了99.49%的用户

内存消耗 :36.8 MB, 在所有 Java 提交中击败了60.18%的用户

时间复杂度：

空间复杂度：，栈的深度最大为列表的长度

## 5、二叉树中的最大路径和

给定一个非空二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径至少包含一个节点，且不一定经过根节点。

示例 1:

输入: [1,2,3]

1

/ \

2 3

输出: 6

示例 2:

输入: [-10,9,20,null,null,15,7]

  -10

   / \

  9  20

    /  \

   15   7

输出: 42

### （1）思路：递归

二叉树的问题一般都可以用递归做。

对于任意一个结点，如果最大和路径包含该结点，那么只可能是两种情况：

1、其左右子树中所构成的路径值较大的那个加上该结点的值后向父结点回溯构成最大路径。

2、左右子树都在最大路径中，加上该结点的值构成了最终的最大路径。

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

private int ret = Integer.MIN\_VALUE;

public int maxPathSum(TreeNode root) {

getMax(root);

return ret;

}

public int getMax(TreeNode root){

if(root == null) return 0;

//如果子树路径和为负则应当置0表示最大路径不包含子树

int left = Math.max(0, getMax(root.left));

int right = Math.max(0, getMax(root.right));

//判断在该结点包含左右子树的路径和（即情况2）是否大于当前最大路径和

ret = Math.max(ret, root.val + left + right);

return Math.max(root.val + left, root.val + right);

}

}

执行用时 :2 ms, 在所有 Java 提交中击败了92.75%的用户

内存消耗 :41.8 MB, 在所有 Java 提交中击败了80.93%的用户

时间复杂度：最好情况，最坏情况

空间复杂度：最好情况，最坏情况

注：这类题的复杂度往往与树的深度有关。

# 三、其他算法题

## 1、海量数据判断某个数据是否存在

1.布隆过滤器

白名单拦截

基于guava

需要二进制向量和一系列随机映射的hash函数

原理：

数据经过一系列hash算法，计算出具体位置，置为1

经典场景

字处理软件中，需要检查一个英语单词是否拼写正确（100w）

在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上（100w）

在网络爬虫里，一个网址是否被访问过（5亿个网站）

### 题目一

**题目描述：**

在2.5亿个整数中判断一个数是否存在，注意，内存不足以容纳2.5亿个整数。

**分析解答：**

方法一：分治法

对于大数据相关的算法题，分治法是一个非常好的方法。针对这一题来说，主要思路为：可以根据实际可用内存的情况，确定一个Hash函数，比如：hash(value)%1000，通过这个Hash函数可以把这2.5亿个数字划分到1000个文件中去（a1，a2……，a1000），然后再对待查找的数字使用同样的Hash函数求出Hash值，假设计算出的Hash值为i，如果这个数存在，那么它一定在文件ai中。通过这种方法就可以把题目转化为文件ai中是否存在这个数。那么接下来的求解过程中可以选用的思路计较多，有：

（1）由于划分后的文件比较小了，就可以直接装载到内存中去，可以把文件中所有的数字都保存到hash\_set中，然后判断待查找的数字是否存在。

（2）如果这个文件中的数字占用的空间还是太大，那么可以用1相同的方法把这个文件继续划分为更小的文件，然后确定待查找的数字可能存在的文件，然后在相应的文件中继续查找。

方法二：位图法

对于这一类判断数字是否存在、判断数字是否重复的问题，位图法是一种非常高效的方法。以32位整型为例，它可以表示数字的个数为2^32（约43亿）.可以申请一个位图，让每个整数对应的位图中的一个bit，这样2^32个数需要的位图的大小为512MB。

512MB是怎么算出的？

总共有2^32个数，每个是1bit，所以

2^32 bit = 2^29 bytes = 2^19 KB = 2^9 MB = 512 MB

具体实现的思路为：申请一个512MB的位图，并把所有的位都初始化为0；接着遍历所有的整数，对遍历到的数字，把相应的位置上的bit设置为1.最后判断待查找的数对应的位图上的值是多少，如果是0，那么表示这个数字不存在，如果是1，那么表示这个数字存在。

### 题目二

1、题目描述

给定100亿个网址，如何检测出重复的文件？这里所谓的“重复”是指两个URL完全相同。

2、思路

100亿个网址（URL）要占用多少空间呢？如果每个网址平均长度为100个字符，每个字符要占用4字节，则这份100亿个网址的列表将占用约4兆兆字节（4TB）。在内存中可能放不下那么多数据。

不过，不防假装一下，这些数据真的奇迹般的放进了内存，毕竟先求解简化的题目是很有用的做法。对于此题的简化版，只要创建一个散列表（HashMap），第一次扫描所有网址，将URL作为key，将URL出现的次数作为value，统计所有URL出现的次数。第二次扫描所有URL,出现次数大于1的URL，则为重复的URL。（另一种做法是对列表进行排序，找出重复项，这需要额外耗费一些时间，几无优点可言）。

至此，我们得到此题简化版的解法，那么，假设我们手上有4000GB的数据，而且无法全部放入内存，该怎么办？倒也好办，我们可以将部分数据存储至磁盘，或者将数据分拆到多台机器上。

解法1：存储至磁盘

若将所有数据存储在一台机器上，可以对数据进行两次扫描。第一次扫描是将网址列表拆分为4000组，每组1GB。简单的做法是将每个网址u存放在名为<x>.txt的文件中，其中x=hash(u)%4000。也就是说，我们会根据网址的散列值（除以分组数量取余数）分割这些网址。这样一来，所有散列值相同的网址都会位于同一文件。

第二次扫描时，我们其实是在实现前面简化版问题的解法：将每个文件载入内存，创建网址的散列表（HashMap），找出重复的。

解法2：多台机器

另一种解法的基本流程是一样的，只不过要使用多台机器。在这种解法中，我们会将网址发送到机器x上，而不是储存至文件<x>.txt。

使用多台机器有优点也有缺点。

主要优点是可以并行执行这些操作，同时处理4000个分组。对于海量数据，这么做就能迅速有效的解决问题。

缺点是现在必须依靠4000台不同的机器，同时要做到操作无误。这可能不太现实（特别是对于数据量更大、机器更多的情况），我们需要开始考虑如何处理机器故障。此外，涉及这么多机器，无疑大幅度增加了系统的复杂性。

注：如果利用Hadoop，可以在mapper中，以x作为key，相应的url作为value发送到reducer，相同key的url会来到同一个reducer，利用前面简化版问题的解法，找出重复网址。

### 题目三

腾讯面试题：给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？

这个题目已经有一段时间了，但是腾讯现在还在用来面试。腾讯第一次面试的时候我听面试官问其他人了。我不是故意的。。。

**方案1：**申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

**方案2：**  
因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；这里我们把40亿个数中的每一个用32位的二进制来表示。假设这40亿个数开始放在一个文件中。

    然后将这40亿个数分成两类:

      1.最高位为0  
      2.最高位为1  
并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；与要查找的数的最高位比较并接着进入相应的文件再查找

      再然后把这个文件为又分成两类:  
      1.次最高位为0  
      2.次最高位为1

并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）； 与要查找的数的次最高位比较并接着进入相应的文件再查找。  
    .......  
    以此类推，就可以找到了,而且时间复杂度为O(logn)。

注：这个类推一般不会推到底的，否则需要的文件就太多了，每个数对应一个文件的话，需要2^32个文件？…