# 数组Array

## 4. Median of Two Sorted Arrays 两个排序数组的中位数

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2 。

请找出这两个有序数组的中位数。要求算法的时间复杂度为 O(log (m+n)) 。

【知识点】二分查找法，二分搜索法

【分析】限制了时间复杂度为O(log (m+n))，看到这个时间复杂度，自然的想到了应该使用二分查找法来求解。但是这道题难在要在两个未合并的有序数组之间使用二分法，这里我们需要定义一个函数来找到第K个元素，由于两个数组长度之和的奇偶不确定，因此需要分情况来讨论，对于奇数的情况，直接找到最中间的数即可，偶数的话需要求最中间两个数的平均值。下面重点来看如何实现找到第K个元素，首先我们需要让数组1的长度小于或等于数组2的长度，那么我们只需判断如果数组1的长度大于数组2的长度的话，交换两个数组即可，然后我们要判断小的数组是否为空，为空的话，直接在另一个数组找第K个即可。还有一种情况是当K=1时，表示我们要找第一个元素，只要比较两个数组的第一个元素，返回较小的即可。

【CODE】

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

int m = nums1.size(), n = nums2.size();

return (findKth(nums1, nums2, (m + n + 1) / 2) + findKth(nums1, nums2, (m + n + 2) / 2)) / 2.0;

}

int findKth(vector<int> nums1, vector<int> nums2, int k) {

int m = nums1.size(), n = nums2.size();

if (m > n) return findKth(nums2, nums1, k);

if (m == 0) return nums2[k - 1];

if (k == 1) return min(nums1[0], nums2[0]);

int i = min(m, k / 2), j = min(n, k / 2);

if (nums1[i - 1] > nums2[j - 1]) {

return findKth(nums1, vector<int>(nums2.begin() + j, nums2.end()), k - j);

} else {

return findKth(vector<int>(nums1.begin() + i, nums1.end()), nums2, k - i);

}

return 0;

}

};

【CODE】

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

int m = nums1.size(), n = nums2.size();

if (m < n) return findMedianSortedArrays(nums2, nums1);

if (n == 0) return ((double)nums1[(m - 1) / 2] + (double)nums1[m / 2]) / 2.0;

int left = 0, right = n \* 2;

while (left <= right) {

int mid2 = (left + right) / 2;

int mid1 = m + n - mid2;

double L1 = mid1 == 0 ? INT\_MIN : nums1[(mid1 - 1) / 2];

double L2 = mid2 == 0 ? INT\_MIN : nums2[(mid2 - 1) / 2];

double R1 = mid1 == m \* 2 ? INT\_MAX : nums1[mid1 / 2];

double R2 = mid2 == n \* 2 ? INT\_MAX : nums2[mid2 / 2];

if (L1 > R2) left = mid2 + 1;

else if (L2 > R1) right = mid2 - 1;

else return (max(L1, L2) + min(R1, R2)) / 2;

}

return -1;

}

};

## 11. Container With Most Water盛最多水的容器

给定 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。画 n 条垂直线，使得垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0)。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

注意：你不能倾斜容器，n 至少是2。

【分析】定义i和j两个指针分别指向数组的左右两端，然后两个指针向中间搜索，每移动一次算一个值和结果比较取较大的，容器装水量的算法是找出左右两个边缘中较小的那个乘以两边缘的距离。对于相同的高度们直接移动i和j就行了，不再进行计算容量了。

【CODE】

class Solution {

public:

int maxArea(vector<int>& height) {

int res = 0, i = 0, j = height.size() - 1;

while (i < j) {

int h = min(height[i], height[j]);

res = max(res, h \* (j - i));

while (i < j && h == height[i]) ++i;

while (i < j && h == height[j]) --j;

}

return res;

}

};

## 15. 3Sum三数之和

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

【分析】要我们找出三个数且和为0，那么除了三个数全是0的情况之外，肯定会有负数和正数，我们还是要先fix一个数，然后去找另外两个数，我们只要找到两个数且和为第一个fix数的相反数就行了，既然另外两个数不能使用Two Sum的那种解法来找，如果能更有效的定位呢？我们肯定不希望遍历所有两个数的组合吧，所以如果数组是有序的，那么我们就可以用双指针以线性时间复杂度来遍历所有满足题意的两个数组合。

我们对原数组进行排序，然后开始遍历排序后的数组，这里注意不是遍历到最后一个停止，而是到倒数第三个就可以了。这里我们可以先做个剪枝优化，就是当遍历到正数的时候就break，为什么，因为我们的数组现在是有序的了，如果第一个要fix的数就是正数了，那么后面的数字就都是正数，就永远不会出现和为0的情况了。然后我们还要加上重复就跳过的处理，处理方法是从第二个数开始，如果和前面的数字相等，就跳过，因为我们不想把相同的数字fix两次。对于遍历到的数，用0减去这个fix的数得到一个target，然后只需要再之后找到两个数之和等于target即可。我们用两个指针分别指向fix数字之后开始的数组首尾两个数，如果两个数和正好为target，则将这两个数和fix的数一起存入结果中。然后就是跳过重复数字的步骤了，两个指针都需要检测重复数字。如果两数之和小于target，则我们将左边那个指针i右移一位，使得指向的数字增大一些。同理，如果两数之和大于target，则我们将右边那个指针j左移一位，使得指向的数字减小一些。

【CODE】

class Solution {

public:

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

vector<vector<int>> res;

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

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

if (nums[k] > 0) break;

if (k > 0 && nums[k] == nums[k - 1]) continue;

int target = 0 - nums[k];

int i = k + 1, j = nums.size() - 1;

while (i < j) {

if (nums[i] + nums[j] == target) {

res.push\_back({nums[k], nums[i], nums[j]});

while (i < j && nums[i] == nums[i + 1]) ++i;

while (i < j && nums[j] == nums[j - 1]) --j;

++i; --j;

} else if (nums[i] + nums[j] < target) ++i;

else --j;

}

}

return res;

}

};

## 16. 3Sum Closest最接近的三数之和

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

例如，给定数组 nums = [-1，2，1，-4], 和 target = 1.

与 target 最接近的三个数的和为 2. (-1 + 2 + 1 = 2).

【分析】那么这道题让我们返回这个最接近于给定值的值，即我们要保证当前三数和跟给定值之间的差的绝对值最小，所以我们需要定义一个变量diff用来记录差的绝对值，然后我们还是要先将数组排个序，然后开始遍历数组，思路跟那道三数之和很相似，都是先确定一个数，然后用两个指针left和right来滑动寻找另外两个数，每确定两个数，我们求出此三数之和，然后算和给定值的差的绝对值存在newDiff中，然后和diff比较并更新diff和结果closest即可。

【CODE】

class Solution {

public:

int threeSumClosest(vector<int>& nums, int target) {

int closest = nums[0] + nums[1] + nums[2];

int diff = abs(closest - target);

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

for (int i = 0; i < nums.size() - 2; ++i) {

int left = i + 1, right = nums.size() - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

int newDiff = abs(sum - target);

if (diff > newDiff) {

diff = newDiff;

closest = sum;

}

if (sum < target) ++left;

else --right;

}

}

return closest;

}

};

## 18. 4Sum四数之和

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：答案中不可以包含重复的四元组。

【分析】为了避免重复项，我们使用了STL中的set，其特点是不能有重复，如果新加入的数在set中原本就存在的话，插入操作就会失败，这样能很好的避免的重复项的存在。此题的O(n^3)解法的思路跟3Sum 三数之和基本没啥区别，就是多加了一层for循环，其他的都一样。

【CODE】

class Solution {

public:

vector<vector<int>> fourSum(vector<int> &nums, int target) {

set<vector<int>> res;

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

for (int i = 0; i < int(nums.size() - 3); ++i) {

for (int j = i + 1; j < int(nums.size() - 2); ++j) {

if (j > i + 1 && nums[j] == nums[j - 1]) continue;

int left = j + 1, right = nums.size() - 1;

while (left < right) {

int sum = nums[i] + nums[j] + nums[left] + nums[right];

if (sum == target) {

vector<int> out{nums[i], nums[j], nums[left], nums[right]};

res.insert(out);

++left; --right;

} else if (sum < target) ++left;

else --right;

}

}

}

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

}

};

## 31. Next Permutation下一个排列

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

【分析】看下面一个例子，有如下的一个数组

1　　2　　7　　4　　3　　1

下一个排列为：

1　　3　　1　　2　　4　　7

那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后我们再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

1　　2　　7　　4　　3　　1

1　　2　　7　　4　　3　　1

1　　3　　7　　4　　2　　1

1　　3　　1　　2　　4　　7

【CODE】

class Solution {

public:

void nextPermutation(vector<int>& nums) {int n = nums.size(), i = n - 2, j = n - 1;

while (i >= 0 && nums[i] >= nums[i + 1]) --i;

if (i >= 0) {

while (nums[j] <= nums[i]) --j;

swap(nums[i], nums[j]);

}

reverse(nums.begin() + i + 1, nums.end());

}

};

## 33. Search in Rotated Sorted Array 搜索旋转排序数组

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 O(log n) 级别。

【分析】这道题让在旋转数组中搜索一个给定值，若存在返回坐标，若不存在返回-1。我们还是考虑二分搜索法，但是这道题的难点在于我们不知道原数组在哪旋转了，我们还是用题目中给的例子来分析，对于数组[0 1 2 4 5 6 7] 共有下列七种旋转方法：

二分搜索法的关键在于获得了中间数后，判断下面要搜索左半段还是右半段，我们观察上面红色的数字都是升序的，由此我们可以观察出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边。

【CODE】

class Solution {

public:

int search(vector<int>& nums, int target) {

if (nums.size() == 0) return -1;

int left = 0, right = nums.size() - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (nums[mid] == target) return mid;

else if (nums[mid] < nums[right]) {

if (nums[mid] < target && nums[right] >= target) left = mid + 1;

else right = mid - 1;

} else {

if (nums[left] <= target && nums[mid] > target) right = mid - 1;

else left = mid + 1;

}

}

return -1;

}

};

## 34. Search for a Range在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log n) 级别。如果数组中不存在目标值，返回 [-1, -1]。

【分析】在一个有序整数数组中寻找相同目标值的起始和结束位置，而且限定了时间复杂度为O(logn)，这是典型的二分查找法的时间复杂度，所以这道题我们也需要用此方法，我们的思路是首先对原数组使用二分查找法，找出其中一个目标值的位置，然后向两边搜索找出起始和结束的位置。O(logn)的算法，使用两次二分查找法，第一次找到左边界，第二次调用找到右边界。

【CODE】

class Solution {

public:

vector<int> searchRange(vector<int>& nums, int target) {

int idx = search(nums, 0, nums.size() - 1, target);

if (idx == -1) return {-1, -1};

int left = idx, right = idx;

while (left > 0 && nums[left - 1] == nums[idx]) --left;

while (right < nums.size() - 1 && nums[right + 1] == nums[idx]) ++right;

return {left, right};

}

int search(vector<int>& nums, int left, int right, int target) {

if (left > right) return -1;

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

if (nums[mid] == target) return mid;

else if (nums[mid] < target) return search(nums, mid + 1, right, target);

else return search(nums, left, mid - 1, target);

}

};

## 39. Combination Sum组合总和

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：所有数字（包括 target）都是正整数；解集不能包含重复的组合。

【分析】像这种结果要求返回所有符合要求解的题十有八九都是要利用到递归，而且解题的思路都大同小异，相类似的题目有Path Sum II 二叉树路径之和之二，Subsets II 子集合之二，Permutations 全排列，Permutations II 全排列之二，Combinations 组合项等等，如果仔细研究这些题目发现都是一个套路，都是需要另写一个递归函数，这里我们新加入三个变量，start记录当前的递归到的下标，out为一个解，res保存所有已经得到的解，每次调用新的递归函数时，此时的target要减去当前数组的的数。

【CODE】

class Solution {

public:

vector<vector<int> > combinationSum(vector<int> &candidates, int target) {

vector<vector<int> > res;

vector<int> out;

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

combinationSumDFS(candidates, target, 0, out, res);

return res;

}

void combinationSumDFS(vector<int> &candidates, int target, int start, vector<int> &out, vector<vector<int> > &res) {

if (target < 0) return;

else if (target == 0) res.push\_back(out);

else {

for (int i = start; i < candidates.size(); ++i) {

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

combinationSumDFS(candidates, target - candidates[i], i, out, res);

out.pop\_back();

}

}

}

};

## 40. Combination Sum II组合总和 II

给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的每个数字在每个组合中只能使用一次。

【分析】上题给定数组中的数字可以重复使用，而这道题不能重复使用，只需要在之前的基础上修改两个地方即可，首先在递归的for循环里加上if (i > start && num[i] == num[i - 1]) continue; 这样可以防止res中出现重复项，然后就在递归调用combinationSum2DFS里面的参数换成i+1，这样就不会重复使用数组中的数字。

【CODE】

class Solution {

public:

vector<vector<int> > combinationSum2(vector<int> &num, int target) {

vector<vector<int> > res;

vector<int> out;

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

combinationSum2DFS(num, target, 0, out, res);

return res;

}

void combinationSum2DFS(vector<int> &num, int target, int start, vector<int> &out, vector<vector<int> > &res) {

if (target < 0) return;

else if (target == 0) res.push\_back(out);

else {

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

if (i > start && num[i] == num[i - 1]) continue;

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

combinationSum2DFS(num, target - num[i], i + 1, out, res);

out.pop\_back();

}

}

}

};

## 41. First Missing Positive缺失的第一个正数

给定一个未排序的整数数组，找出其中没有出现的最小的正整数。

说明:你的算法的时间复杂度应为O(n)，并且只能使用常数级别的空间。

【分析】让我们找缺失的首个正数，由于限定了O(n)的时间，所以一般的排序方法都不能用。不能建立新的数组，那么我们只能覆盖原有数组，我们的思路是把1放在数组第一个位置nums[0]，2放在第二个位置nums[1]，即需要把nums[i]放在nums[nums[i] - 1]上，那么我们遍历整个数组，如果nums[i] != i + 1, 而nums[i]为整数且不大于n，另外nums[i]不等于nums[nums[i] - 1]的话，我们将两者位置调换，如果不满足上述条件直接跳过，最后我们再遍历一遍数组，如果对应位置上的数不正确则返回正确的数。

【CODE】

class Solution {

public:

int firstMissingPositive(vector<int>& nums) {

int n = nums.size();

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

while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i]) {

swap(nums[i], nums[nums[i] - 1]);

}

}

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

if (nums[i] != i + 1) return i + 1;

}

return n + 1;

}

};

## 42. Trapping Rain Water接雨水

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

【分析】基于动态规划Dynamic Programming的方法，我们维护一个一维的dp数组，这个DP算法需要遍历两遍数组，第一遍遍历dp[i]中存入i位置左边的最大值，然后开始第二遍遍历数组，第二次遍历时找右边最大值，然后和左边最大值比较取其中的较小值，然后跟当前值A[i]相比，如果大于当前值，则将差值存入结果。

【CODE】

class Solution {

public:

int trap(vector<int>& height) {

int res = 0, mx = 0, n = height.size();

vector<int> dp(n, 0);

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

dp[i] = mx;

mx = max(mx, height[i]);

}

mx = 0;

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

dp[i] = min(dp[i], mx);

mx = max(mx, height[i]);

if (dp[i] > height[i]) res += dp[i] - height[i];

}

return res;

}

};

【分析】一种只需要遍历一次即可的解法，这个算法需要left和right两个指针分别指向数组的首尾位置，从两边向中间扫描，在当前两指针确定的范围内，先比较两头找出较小值，如果较小值是left指向的值，则从左向右扫描，如果较小值是right指向的值，则从右向左扫描，若遇到的值比当较小值小，则将差值存入结果，如遇到的值大，则重新确定新的窗口范围，以此类推直至left和right指针重合。

【CODE】class Solution {

public:

int trap(vector<int>& height) {

int l = 0, r = height.size() - 1, level = 0, res = 0;

while (l < r) {

int lower = height[(height[l] < height[r]) ? l++ : r--];

level = max(level, lower);

res += level - lower;

}

return res;

}

};

【分析】stack来实现。用stack的方法更容易理解，我们的做法是，遍历高度，如果此时栈为空，或者当前高度小于等于栈顶高度，则把当前高度的坐标压入栈，注意我们不直接把高度压入栈，而是把坐标压入栈，这样方便我们在后来算水平距离。当我们遇到比栈顶高度大的时候，就说明有可能会有坑存在，可以装雨水。此时我们栈里至少有一个高度，如果只有一个的话，那么不能形成坑，我们直接跳过，如果多余一个的话，那么此时把栈顶元素取出来当作坑，新的栈顶元素就是左边界，当前高度是右边界，只要取二者较小的，减去坑的高度，长度就是右边界坐标减去左边界坐标再减1，二者相乘就是盛水量。

【CODE】

class Solution {

public:

int trap(vector<int>& height) {

stack<int> st;

int i = 0, res = 0, n = height.size();

while (i < n) {

if (st.empty() || height[i] <= height[st.top()]) {

st.push(i++);

} else {

int t = st.top(); st.pop();

if (st.empty()) continue;

res += (min(height[i], height[st.top()]) - height[t]) \* (i - st.top() - 1);

}

}

return res;

}

};

## 45. Jump Game II跳跃游戏 II

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

【分析】此题的核心方法是利用贪婪算法Greedy的思想来解，为了较快的跳到末尾，我们想知道每一步能跳的范围，这里贪婪并不是要在能跳的范围中选跳力最远的那个位置，因为这样选下来不一定是最优解，这么一说感觉又有点不像贪婪算法了。我们这里贪的是一个能到达的最远范围，我们遍历当前跳跃能到的所有位置，然后根据该位置上的跳力来预测下一步能跳到的最远距离，贪出一个最远的范围，一旦当这个范围到达末尾时，当前所用的步数一定是最小步数。我们需要两个变量cur和pre分别来保存当前的能到达的最远位置和之前能到达的最远位置，只要cur未达到最后一个位置则循环继续，pre先赋值为cur的值，表示上一次循环后能到达的最远位置，如果当前位置i小于等于pre，说明还是在上一跳能到达的范围内，我们根据当前位置加跳力来更新cur，更新cur的方法是比较当前的cur和i + A[i]之中的较大值。如果题目中未说明是否能到达末尾，我们还可以判断此时pre和cur是否相等，如果相等说明cur没有更新，即无法到达末尾位置，返回-1。

【CODE】

class Solution {

public:

int jump(vector<int>& nums) {

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

while (cur < n - 1) {

++res;

int pre = cur;

for (; i <= pre; ++i) {

cur = max(cur, i + nums[i]);

}

if (pre == cur) return -1; // May not need this

}

return res;

}

};

## 48. Rotate Image旋转图像

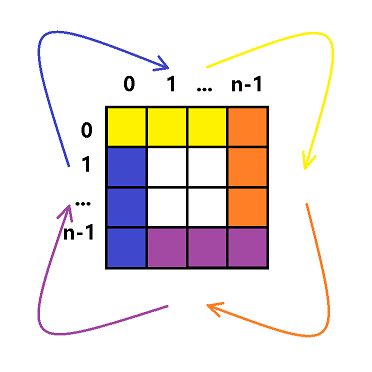
给定一个 n × n 的二维矩阵表示一个图像。

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。

【分析】仔细观察，二维数组我们可以看成是一圈，一圈的数据，如n=3，n=4时，都有两个圈，这里就叫做1环和2环吧，如果n更多，环会更多些。

对于每一环，旋转90度的操作，等于这个数字在环上移动（n-1）步，但是我们如何计算这个环上走的位置呢？这时候我们用四个变量来表示环的位置，左边（列），右边（列），上边（行），下边（行）的位置。

我们从左上角开始，按照左上这一行，循环移动n-1次数据，每次都能正好移动完成（上边的行-》右边-》下边-》左边），这两做n-1次流程就好了。

【CODE】

class Solution {

public:

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

int n=matrix.size();

if(n<2)return ;

int m=n/2;

int edgeTop=0,edgeBottom=n-1,edgeLeft=0,edgeRight=n-1;

while(m-->0){

int tmp=0,last=0;

for(int i=edgeLeft;i<=edgeRight-1;i++){

tmp=matrix[i][edgeRight];

matrix[i][edgeRight]=matrix[edgeTop][i];

last=tmp;

tmp=matrix[edgeBottom][edgeRight-(i-edgeLeft)];

matrix[edgeBottom][edgeRight-(i-edgeLeft)]=last;

last=tmp;

tmp=matrix[edgeBottom-(i-edgeLeft)][edgeLeft];

matrix[edgeBottom-(i-edgeLeft)][edgeLeft]=last;

matrix[edgeTop][i]=tmp;

}

edgeTop++;

edgeBottom--;

edgeLeft++;

edgeRight--;

}

}

};

## 54. Spiral Matrix螺旋矩阵

给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

【分析】将一个矩阵按照螺旋顺序打印出来，我们只能一条边一条边的打印，首先我们要从给定的mxn的矩阵中算出按螺旋顺序有几个环，注意最中间的环可以是一个数字，也可以是一行或者一列。

环数的计算公式是 min(m, n) / 2，知道了环数，我们可以对每个环的边按顺序打印，比如对于题目中给的那个例子，个边生成的顺序是(用颜色标记了数字) Red -> Green -> Blue -> Yellow -> Black

我们定义p，q为当前环的高度和宽度，当p或者q为1时，表示最后一个环只有一行或者一列，可以跳出循环。此题的难点在于下标的转换，如何正确的转换下标是解此题的关键，我们可以对照着上面的3x3的例子来完成下标的填写。

【CODE】

class Solution {

public:

vector<int> spiralOrder(vector<vector<int> > &matrix) {

vector<int> res;

if (matrix.empty() || matrix[0].empty()) return res;

int m = matrix.size(), n = matrix[0].size();

int c = m > n ? (n + 1) / 2 : (m + 1) / 2;

int p = m, q = n;

for (int i = 0; i < c; ++i, p -= 2, q -= 2) {

for (int col = i; col < i + q; ++col)

res.push\_back(matrix[i][col]);

for (int row = i + 1; row < i + p; ++row)

res.push\_back(matrix[row][i + q - 1]);

if (p == 1 || q == 1) break;

for (int col = i + q - 2; col >= i; --col)

res.push\_back(matrix[i + p - 1][col]);

for (int row = i + p - 2; row > i; --row)

res.push\_back(matrix[row][i]);

}

return res;

}

};

## 55. Jump Game跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

【分析】用动态规划Dynamic Programming来解，我们维护一个一位数组dp，其中dp[i]表示达到i位置时剩余的步数，那么难点就是推导状态转移方程。到达当前位置的剩余步数跟什么有关，其实是跟上一个位置的剩余步数和上一个位置的跳力有关，这里的跳力就是原数组中每个位置的数字，因为其代表了以当前位置为起点能到达的最远位置。所以当前位置的剩余步数（dp值）和当前位置的跳力中的较大那个数决定了当前能到的最远距离，而下一个位置的剩余步数（dp值）就等于当前的这个较大值减去1，因为需要花一个跳力到达下一个位置，所以我们就有状态转移方程了：dp[i] = max(dp[i - 1], nums[i - 1]) - 1，如果当某一个时刻dp数组的值为负了，说明无法抵达当前位置，则直接返回false，最后我们判断dp数组最后一位是否为非负数即可知道是否能抵达该位置。

【CODE】

class Solution {

public:

bool canJump(vector<int>& nums) {

vector<int> dp(nums.size(), 0);

for (int i = 1; i < nums.size(); ++i) {

dp[i] = max(dp[i - 1], nums[i - 1]) - 1;

if (dp[i] < 0) return false;

}

return dp.back() >= 0;

}

};

【分析】贪婪算法Greedy Algorithm，我们不关心每一个位置上的剩余步数，我们只希望知道能否到达末尾，也就是说我们只对最远能到达的位置感兴趣，所以我们维护一个变量reach，表示最远能到达的位置，初始化为0。遍历数组中每一个数字，如果当前坐标大于reach或者reach已经抵达最后一个位置则跳出循环，否则就更新reach的值为其和i + nums[i]中的较大值，其中i + nums[i]表示当前位置能到达的最大位置。

【CODE】

class Solution {

public:

bool canJump(vector<int>& nums) {

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

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

if (i > reach || reach >= n - 1) break;

reach = max(reach, i + nums[i]);

}

return reach >= n - 1;

}

};

## 56. Merge Intervals 合并区间

给出一个区间的集合，请合并所有重叠的区间。

【分析】要求我们合并区间，之前那题明确了输入区间集是有序的，而这题没有，所以我们首先要做的就是给区间集排序，由于我们要排序的是个结构体，所以我们要定义自己的comparator，才能用sort来排序，我们以start的值从小到大来排序，排完序我们就可以开始合并了，首先把第一个区间存入结果中，然后从第二个开始遍历区间集，如果结果中最后一个区间和遍历的当前区间无重叠，直接将当前区间存入结果中，如果有重叠，将结果中最后一个区间的end值更新为结果中最后一个区间的end和当前end值之中的较大值，然后继续遍历区间集，以此类推可以得到最终结果。

【CODE】

class Solution {

public:

vector<Interval> merge(vector<Interval>& intervals) {

if (intervals.empty()) return {};

sort(intervals.begin(), intervals.end(), [](Interval &a, Interval &b) {return a.start < b.start;});

vector<Interval> res{intervals[0]};

for (int i = 1; i < intervals.size(); ++i) {

if (res.back().end < intervals[i].start) {

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

} else {

res.back().end = max(res.back().end, intervals[i].end);

}

}

return res;

}

};

【分析】下面这种解法将起始位置和结束位置分别存到了两个不同的数组starts和ends中，然后分别进行排序，之后用两个指针i和j，初始化时分别指向starts和ends数组的首位置，然后如果i指向starts数组中的最后一个位置，或者当starts数组上i+1位置上的数字大于ends数组的i位置上的数时，此时说明区间已经不连续了，我们来看题目中的例子，排序后的starts和ends为：

starts: 1 2 8 15

ends: 3 6 10 18

红色为i的位置，蓝色为j的位置，那么此时starts[i+1]为8，ends[i]为6，8大于6，所以此时不连续了，将区间[starts[j], ends[i]]，即 [1, 6] 加入结果res中，然后j赋值为i+1继续循环。

【CODE】

class Solution {

public:

vector<Interval> merge(vector<Interval>& intervals) {

int n = intervals.size();

vector<Interval> res;

vector<int> starts, ends;

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

starts.push\_back(intervals[i].start);

ends.push\_back(intervals[i].end);

}

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

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

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

if (i == n - 1 || starts[i + 1] > ends[i]) {

res.push\_back(Interval(starts[j], ends[i]));

j = i + 1;

}

}

return res;

}

};

## 57. Insert Interval插入区间

给出一个无重叠的 ，按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

输入: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

输出: [[1,2],[3,10],[12,16]]

解释: 这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。

【分析】让我们在一系列非重叠的区间中插入一个新的区间，可能还需要和原有的区间合并，那么我们需要对给区间集一个一个的遍历比较，那么会有两种情况，重叠或是不重叠，不重叠的情况最好，直接将新区间插入到对应的位置即可，重叠的情况比较复杂，有时候会有多个重叠，我们需要更新新区间的范围以便包含所有重叠，之后将新区间加入结果res，最后将后面的区间再加入结果res即可。具体思路是，我们用一个变量cur来遍历区间，如果当前cur区间的结束位置小于要插入的区间的起始位置的话，说明没有重叠，则将cur区间加入结果res中，然后cur自增1。直到有cur越界或有重叠while循环退出，然后再用一个while循环处理所有重叠的区间，每次用取两个区间起始位置的较小值，和结束位置的较大值来更新要插入的区间，然后cur自增1。直到cur越界或者没有重叠时while循环退出。之后将更新好的新区间加入结果res，然后将cur之后的区间再加入结果res中即可。

【CODE】class Solution {

public:

vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {

vector<Interval> res;

int n = intervals.size(), cur = 0;

while (cur < n && intervals[cur].end < newInterval.start) {

res.push\_back(intervals[cur++]);

}

while (cur < n && intervals[cur].start <= newInterval.end) {

newInterval.start = min(newInterval.start, intervals[cur].start);

newInterval.end = max(newInterval.end, intervals[cur].end);

++cur;

}

res.push\_back(newInterval);

while (cur < n) {

res.push\_back(intervals[cur++]);

}

return res;

}

};

## 59. Spiral Matrix II螺旋矩阵 II

给定一个正整数 n，生成一个包含 1 到 n2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

【分析】此题跟之前那道 Spiral Matrix 螺旋矩阵 本质上没什么区别，就相当于个类似逆运算的过程，这道题是要按螺旋的顺序来填数，由于给定矩形是个正方形，我们计算环数时用n / 2来计算，若n为奇数时，此时最中间的那个点没有被算在环数里，所以最后需要单独赋值，还是下标转换问题是难点，参考之前 Spiral Matrix 螺旋矩阵 的讲解来转换下标。

【CODE】

class Solution {

public:

vector<vector<int> > generateMatrix(int n) {

vector<vector<int> > res(n, vector<int>(n, 1));

int val = 1, p = n;

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

for (int col = i; col < i + p; ++col)

res[i][col] = val++;

for (int row = i + 1; row < i + p; ++row)

res[row][i + p - 1] = val++;

for (int col = i + p - 2; col >= i; --col)

res[i + p - 1][col] = val++;

for (int row = i + p - 2; row > i; --row)

res[row][i] = val++;

}

if (n % 2 != 0) res[n / 2][n / 2] = val;

return res;

}

};

## 62. Unique Paths不同路径

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？（说明：m 和 n 的值均不超过 100。）

【分析】跟之前那道 Climbing Stairs 爬梯子问题类似，那道题是说可以每次能爬一格或两格，问到达顶部的所有不同爬法的个数。而这道题是每次可以向下走或者向右走，求到达最右下角的所有不同走法的个数。那么跟爬梯子问题一样，我们需要用动态规划Dynamic Programming来解，我们可以维护一个二维数组dp，其中dp[i][j]表示到当前位置不同的走法的个数，然后可以得到递推式为: dp[i][j] = dp[i - 1][j] + dp[i][j - 1]，这里为了节省空间，我们使用一维数组dp，一行一行的刷新也可以。

【CODE】// DP

class Solution {

public:

int uniquePaths(int m, int n) {

vector<int> dp(n, 1);

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

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

dp[j] += dp[j - 1];

}

}

return dp[n - 1];

}

};

【分析】机器人总共走了m + n - 2步，其中m - 1步向下走，n - 1步向右走，那么总共不同的方法个数就相当于在步数里面m - 1和n - 1中较小的那个数的取法，实际上是一道组合数的问题。

【CODE】

class Solution {

public:

int uniquePaths(int m, int n) {

double num = 1, denom = 1;

int small = m > n ? n : m;

for (int i = 1; i <= small - 1; ++i) {

num \*= m + n - 1 - i;

denom \*= i;

}

return (int)(num / denom);

}

};

## 63. Unique Paths II不同路径 II

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？

输入:

[

[0,0,0],

[0,1,0],

[0,0,0]

]

输出: 2

解释:

3x3 网格的正中间有一个障碍物。

从左上角到右下角一共有 2 条不同的路径：

1. 向右 -> 向右 -> 向下 -> 向下

2. 向下 -> 向下 -> 向右 -> 向右

【分析】在路径中加了一些障碍物，还是用动态规划Dynamic Programming来解，不同的是当遇到为1的点，将该位置的dp数组中的值清零，其余和之前那道题并没有什么区别。

【CODE】// DP

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

if (obstacleGrid.empty() || obstacleGrid[0].empty()) return 0;

int m = obstacleGrid.size(), n = obstacleGrid[0].size();

if (obstacleGrid[0][0] == 1) return 0;

vector<int> dp(n, 0);

dp[0] = 1;

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

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

if (obstacleGrid[i][j] == 1) dp[j] = 0;

else if (j > 0) dp[j] += dp[j - 1];

}

}

return dp[n - 1];

}

};

## 64. Minimum Path Sum 最小路径和

给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

【分析】动态规划Dynamic Programming来做，我们维护一个二维的dp数组，其中dp[i][j]表示当前位置的最小路径和，递推式也容易写出来 dp[i][j] = grid[i][j] + min(dp[i - 1][j],dp[i][j - 1]).

【CODE】

class Solution {

public:

int minPathSum(vector<vector<int> > &grid) {

int m = grid.size(), n = grid[0].size();

int dp[m][n];

dp[0][0] = grid[0][0];

for (int i = 1; i < m; ++i) dp[i][0] = grid[i][0] + dp[i - 1][0];

for (int i = 1; i < n; ++i) dp[0][i] = grid[0][i] + dp[0][i - 1];

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

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

dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);

}

}

return dp[m - 1][n - 1];

}

};

## 73. Set Matrix Zeroes矩阵置零

给定一个 m x n 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用原地算法。

【分析】这道题中说的空间复杂度为O(mn)的解法自不用多说，直接新建一个和matrix等大小的矩阵，然后一行一行的扫，只要有0，就将新建的矩阵的对应行全赋0，行扫完再扫列，然后把更新完的矩阵赋给matrix即可，这个算法的空间复杂度太高。将其优化到O(m+n)的方法是，用一个长度为m的一维数组记录各行中是否有0，用一个长度为n的一维数组记录各列中是否有0，最后直接更新matrix数组即可。这道题的要求是用O(1)的空间，那么我们就不能新建数组，我们考虑就用原数组的第一行第一列来记录各行各列是否有0.

- 先扫描第一行第一列，如果有0，则将各自的flag设置为true

- 然后扫描除去第一行第一列的整个数组，如果有0，则将对应的第一行和第一列的数字赋0

- 再次遍历除去第一行第一列的整个数组，如果对应的第一行和第一列的数字有一个为0，则将当前值赋0

- 最后根据第一行第一列的flag来更新第一行第一列

【CODE】class Solution {

public:

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

if (matrix.empty() || matrix[0].empty()) return;

int m = matrix.size(), n = matrix[0].size();

bool rowZero = false, colZero = false;

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

if (matrix[i][0] == 0) colZero = true;

}

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

if (matrix[0][i] == 0) rowZero = true;

}

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

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

if (matrix[i][j] == 0) {

matrix[0][j] = 0;

matrix[i][0] = 0;

}

}

}

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

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

if (matrix[0][j] == 0 || matrix[i][0] == 0) {

matrix[i][j] = 0;

}

}

}

if (rowZero) {

for (int i = 0; i < n; ++i) matrix[0][i] = 0;

}

if (colZero) {

for (int i = 0; i < m; ++i) matrix[i][0] = 0;

}

}

};

## 74. Search a 2D Matrix 搜索二维矩阵

编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

每行中的整数从左到右按升序排列。

每行的第一个整数大于前一行的最后一个整数。

【分析】使用一次二分查找法，如果我们按S型遍历该二维数组，可以得到一个有序的一维数组，那么我们只需要用一次二分查找法，而关键就在于坐标的转换，如何把二维坐标和一维坐标转换是关键点，把一个长度为n的一维数组转化为m\*n的二维数组(m\*n = n)后，那么原一维数组中下标为i的元素将出现在二维数组中的[i/n][i%n]的位置。

【CODE】// One binary search

class Solution {

public:

bool searchMatrix(vector<vector<int> > &matrix, int target) {

if (matrix.empty() || matrix[0].empty()) return false;

if (target < matrix[0][0] || target > matrix.back().back()) return false;

int m = matrix.size(), n = matrix[0].size();

int left = 0, right = m \* n - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (matrix[mid / n][mid % n] == target) return true;

else if (matrix[mid / n][mid % n] < target) left = mid + 1;

else right = mid - 1;

}

return false;

}

};

## 75. Sort Colors分类颜色

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意:不能使用代码库中的排序函数来解决这道题。

示例:输入: [2,0,2,1,1,0]

输出: [0,0,1,1,2,2]

进阶：一个直观的解决方案是使用计数排序的两趟扫描算法。

首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。

你能想出一个仅使用常数空间的一趟扫描算法吗？

【分析】只遍历一次数组来求解，那么我需要用双指针来做，分别从原数组的首尾往中心移动。

- 定义red指针指向开头位置，blue指针指向末尾位置

- 从头开始遍历原数组，如果遇到0，则交换该值和red指针指向的值，并将red指针后移一位。若遇到2，则交换该值和blue指针指向的值，并将blue指针前移一位。若遇到1，则继续遍历。

【CODE】

class Solution {

public:

void sortColors(vector<int>& nums){

int n = nums.size();

int red = 0, blue = n - 1;

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

if (nums[i] == 0) {

swap(nums[i], nums[red++]);

} else if (nums[i] == 2) {

swap(nums[i--], nums[blue--]);

}

}

}

};

## 79. Word Search单词搜索

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

【分析】这道题是典型的深度优先遍历DFS的应用，原二维数组就像是一个迷宫，可以上下左右四个方向行走，我们以二维数组中每一个数都作为起点和给定字符串做匹配，我们还需要一个和原数组等大小的visited数组，是bool型的，用来记录当前位置是否已经被访问过，因为题目要求一个cell只能被访问一次。如果二维数组board的当前字符和目标字符串word对应的字符相等，则对其上下左右四个邻字符分别调用DFS的递归函数，只要有一个返回true，那么就表示可以找到对应的字符串，否则就不能找到。

【CODE】

class Solution {

public:

bool exist(vector<vector<char>>& board, string word) {

if (board.empty() || board[0].empty()) return false;

int m = board.size(), n = board[0].size();

vector<vector<bool>> visited(m, vector<bool>(n, false));

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

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

if (search(board, word, 0, i, j, visited)) return true;

}

}

return false;

}

bool search(vector<vector<char>>& board, string word, int idx, int i, int j, vector<vector<bool>>& visited) {

if (idx == word.size()) return true;

int m = board.size(), n = board[0].size();

if (i < 0 || j < 0 || i >= m || j >= n || visited[i][j] || board[i][j] != word[idx]) return false;

visited[i][j] = true;

bool res = search(board, word, idx + 1, i - 1, j, visited)

|| search(board, word, idx + 1, i + 1, j, visited)

|| search(board, word, idx + 1, i, j - 1, visited)

|| search(board, word, idx + 1, i, j + 1, visited);

visited[i][j] = false;

return res;

}

};

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

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现两次，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。

说明:为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

【分析】用一个变量count来记录还允许有几次重复，count初始化为1，如果出现过一次重复，则count递减1，那么下次再出现重复，快指针直接前进一步，如果这时候不是重复的，则count恢复1，由于整个数组是有序的，所以一旦出现不重复的数，则一定比这个数大，此数之后不会再有重复项。

【CODE】

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

int n = nums.size();

if (n <= 2) return n;

int pre = 0, cur = 1, count = 1;

while (cur < n) {

if (nums[pre] == nums[cur] && count == 0) ++cur;

else {

if (nums[pre] == nums[cur]) --count;

else count = 1;

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

}

}

return pre + 1;

}

};

## 81. Search in Rotated Sorted Array II搜索旋转排序数组 II

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,0,1,2,2,5,6] 可能变为 [2,5,6,0,0,1,2] )。

编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 true，否则返回 false。

进阶:这是 搜索旋转排序数组 的延伸题目，本题中的 nums 可能包含重复元素。

这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？

【分析】现在数组中允许出现重复数字，这个也会影响我们选择哪半边继续搜索，由于之前那道题不存在相同值，我们在比较中间值和最右值时就完全符合之前所说的规律：如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的。而如果可以有重复值，就会出现来面两种情况，[3 1 1] 和 [1 1 3 1]，对于这两种情况中间值等于最右值时，目标值3既可以在左边又可以在右边，那怎么办么，对于这种情况其实处理非常简单，只要把最右值向左一位即可继续循环，如果还相同则继续移，直到移到不同值为止，然后其他部分还采用 Search in Rotated Sorted Array 在旋转有序数组中搜索中的方法。

【CODE】

class Solution {

public:

bool search(vector<int>& nums, int target) {

int n = nums.size();

if (n == 0) return false;

int left = 0, right = n - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (nums[mid] == target) return true;

else if (nums[mid] < nums[right]) {

if (nums[mid] < target && nums[right] >= target) left = mid + 1;

else right = mid - 1;

} else if (nums[mid] > nums[right]){

if (nums[left] <= target && nums[mid] > target) right = mid - 1;

else left = mid + 1;

} else --right;

}

return false;

}

};

## 84. Largest Rectangle in Histogram柱状图中最大的矩形

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。

以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。

图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

【分析】遍历数组，每找到一个局部峰值，然后向前遍历所有的值，算出共同的矩形面积，每次对比保留最大值。

【CODE】// Pruning optimize

class Solution {

public:

int largestRectangleArea(vector<int> &height) {

int res = 0;

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

if (i + 1 < height.size() && height[i] <= height[i + 1]) {

continue;

}

int minH = height[i];

for (int j = i; j >= 0; --j) {

minH = min(minH, height[j]);

int area = minH \* (i - j + 1);

res = max(res, area);

}

}

return res;

}

};

【知识点】单调栈

【分析】维护一个栈，用来保存递增序列，相当于上面那种方法的找局部峰值。我们可以看到，直方图矩形面积要最大的话，需要尽可能的使得连续的矩形多，并且最低一块的高度要高。有点像木桶原理一样，总是最低的那块板子决定桶的装水量。那么既然需要用单调栈来做，首先要考虑到底用递增栈，还是用递减栈来做。我们想啊，递增栈是维护递增的顺序，当遇到小于栈顶元素的数就开始处理，而递减栈正好相反，维护递减的顺序，当遇到大于栈顶元素的数开始处理。那么根据这道题的特点，我们需要按从高板子到低板子的顺序处理，先处理最高的板子，宽度为1，然后再处理旁边矮一些的板子，此时长度为2，因为之前的高板子可组成矮板子的矩形 ，因此我们需要一个递增栈，当遇到大的数字直接进栈，而当遇到小于栈顶元素的数字时，就要取出栈顶元素进行处理了，那取出的顺序就是从高板子到矮板子了，于是乎遇到的较小的数字只是一个触发，表示现在需要开始计算矩形面积了，为了使得最后一块板子也被处理，这里用了个小trick，在高度数组最后面加上一个0，这样原先的最后一个板子也可以被处理了。由于栈顶元素是矩形的高度，那么关键就是求出来宽度，那么跟之前那道Trapping Rain Water一样，单调栈中不能放高度，而是需要放坐标。由于我们先取出栈中最高的板子，那么就可以先算出长度为1的矩形面积了，然后再取下一个板子，此时根据矮板子的高度算长度为2的矩形面积，以此类推，知道数字大于栈顶元素为止，再次进栈。

【CODE】

class Solution {

public:

int largestRectangleArea(vector<int>& heights) {

int res = 0;

stack<int> st;

heights.push\_back(0);

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

while (!st.empty() && heights[st.top()] >= heights[i]) {

int cur = st.top(); st.pop();

res = max(res, heights[cur] \* (st.empty() ? i : (i - st.top() - 1)));

}

st.push(i);

}

return res;

}

};

## 85. Maximal Rectangle最大矩形

给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

输入:

[

["1","0","1","0","0"],

["1","0","1","1","1"],

["1","1","1","1","1"],

["1","0","0","1","0"]

]

输出: 6

【分析】此题是之前那道的 Largest Rectangle in Histogram 直方图中最大的矩形 的扩展，这道题的二维矩阵每一层向上都可以看做一个直方图，输入矩阵有多少行，就可以形成多少个直方图，对每个直方图都调用 Largest Rectangle in Histogram 直方图中最大的矩形 中的方法，就可以得到最大的矩形面积。那么这道题唯一要做的就是将每一层构成直方图，由于题目限定了输入矩阵的字符只有 '0' 和 '1' 两种，所以处理起来也相对简单。方法是，对于每一个点，如果是‘0’，则赋0，如果是 ‘1’，就赋 之前的height值加上1。

【CODE】

class Solution {

public:

int maximalRectangle(vector<vector<char>>& matrix) {

if (matrix.empty() || matrix[0].empty()) return 0;

int res = 0, m = matrix.size(), n = matrix[0].size();

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

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

stack<int> s;

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

if (j < n) {

height[j] = matrix[i][j] == '1' ? height[j] + 1 : 0;

}

while (!s.empty() && height[s.top()] >= height[j]) {

int cur = s.top(); s.pop();

res = max(res, height[cur] \* (s.empty() ? j : (j - s.top() - 1)));

}

s.push(j);

}

}

return res;

}

};

## 90. Subsets II 子集 II

给定一个可能包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）。

说明：解集不能包含重复的子集。

【分析】子集合之二是之前那道 Subsets 子集合 的延伸，这次输入数组允许有重复项，其他条件都不变，只需要在之前那道题解法的基础上稍加改动便可以做出来，我们先来看非递归解法，拿题目中的例子[1 2 2]来分析，根据之前 Subsets 子集合 里的分析可知，当处理到第一个2时，此时的子集合为[], [1], [2], [1, 2]，而这时再处理第二个2时，如果在[]和[1]后直接加2会产生重复，所以只能在上一个循环生成的后两个子集合后面加2，发现了这一点，题目就可以做了，我们用last来记录上一个处理的数字，然后判定当前的数字和上面的是否相同，若不同，则循环还是从0到当前子集的个数，若相同，则新子集个数减去之前循环时子集的个数当做起点来循环，这样就不会产生重复。

【CODE】

class Solution {

public:

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

if (S.empty()) return {};

vector<vector<int>> res(1);

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

int size = 1, last = S[0];

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

if (last != S[i]) {

last = S[i];

size = res.size();

}

int newSize = res.size();

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

res.push\_back(res[j]);

res.back().push\_back(S[i]);

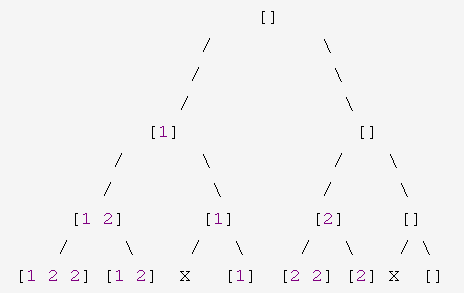
}

}

return res;

}

};

【分析】对于递归的解法，根据之前 Subsets 子集合 里的构建树的方法，在处理到第二个2时，由于前面已经处理了一次2，这次我们只在添加过2的[2] 和 [1 2]后面添加2，其他的都不添加，那么这样构成的二叉树如图：

代码只需在原有的基础上增加一句话，while (S[i] == S[i + 1]) ++i; 这句话的作用是跳过树中为X的叶节点，因为它们是重复的子集，应被抛弃。

【CODE】class Solution {

public:

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

if (S.empty()) return {};

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();

while (i + 1 < S.size() && S[i] == S[i + 1]) ++i;

}

}

};

## 120. Triangle三角形最小路径和

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

给定三角形：

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。

说明：如果你可以只使用 O(n) 的额外空间（n 为三角形的总行数）来解决这个问题，那么你的算法会很加分。

【分析】DP方法，复制三角形最后一行，作为用来更新的一位数组。然后逐个遍历这个DP数组，对于每个数字，和它之后的元素比较选择较小的再加上上面一行相邻位置的元素做为新的元素，然后一层一层的向上扫描，整个过程和冒泡排序的原理差不多，最后最小的元素都冒到前面，第一个元素即为所求。

【CODE】

class Solution {

public:

int minimumTotal(vector<vector<int> > &triangle) {

int n = triangle.size();

vector<int> dp(triangle.back());

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

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

dp[j] = min(dp[j], dp[j + 1]) + triangle[i][j];

}

}

return dp[0];

}

};

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

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。

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】这道是要求最多交易两次，找到最大利润，还是需要用动态规划Dynamic Programming来解，而这里我们需要两个递推公式来分别更新两个变量local和global，我们其实可以求至少k次交易的最大利润，找到通解后可以设定 k = 2，即为本题的解答。我们定义local[i][j]为在到达第i天时最多可进行j次交易并且最后一次交易在最后一天卖出的最大利润，此为局部最优。然后我们定义global[i][j]为在到达第i天时最多可进行j次交易的最大利润，此为全局最优。

递推式为：

local[i][j] = max(global[i - 1][j - 1] + max(diff, 0), local[i - 1][j] + diff)

global[i][j] = max(local[i][j], global[i - 1][j])

其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值中取较大值，而全局最优比较局部最优和前一天的全局最优。

【CODE】

class Solution {

public:

int maxProfit(vector<int> &prices) {

if (prices.empty()) return 0;

int n = prices.size(), g[n][3] = {0}, l[n][3] = {0};

for (int i = 1; i < prices.size(); ++i) {

int diff = prices[i] - prices[i - 1];

for (int j = 1; j <= 2; ++j) {

l[i][j] = max(g[i - 1][j - 1] + max(diff, 0), l[i - 1][j] + diff);

g[i][j] = max(l[i][j], g[i - 1][j]);

}

}

return g[n - 1][2];

}

};

## 126. Word Ladder II单词接龙 II

给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。转换需遵循如下规则：

每次转换只能改变一个字母。

转换过程中的中间单词必须是字典中的单词。

说明:

如果不存在这样的转换序列，返回一个空列表。

所有单词具有相同的长度。

所有单词只由小写字母组成。

字典中不存在重复的单词。

你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

输入:

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

输出:

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

【分析】这道题是相当有难度的一道题，它比之前那道Word Ladder 词语阶梯要复杂很多，全场第四低的通过率12.9%正说明了这道题的难度，下面这种解法的核心思想是BFS，大概思路如下：我们的目的是找出所有的路径，我们建立一个路径集paths，用以保存所有路径，然后是起始路径p，在p中先把起始单词放进去。然后定义两个整型变量level，和minLevel，其中level是记录循环中当前路径的长度，minLevel是记录最短路径的长度，这样的好处是，如果某条路径的长度超过了已有的最短路径的长度，那么舍弃，这样会提高运行速度，相当于一种剪枝。还要定义一个set变量words，用来记录已经循环过的路径中的词，然后就是BFS的核心了，循环路径集paths里的内容，取出队首路径，如果该路径长度大于level，说明字典中的有些词已经存入路径了，如果在路径中重复出现，则肯定不是最短路径，所以我们需要在字典中将这些词删去，然后将words清空，对循环对剪枝处理。然后我们取出当前路径的最后一个词，对每个字母进行替换并在字典中查找是否存在替换后的新词，这个过程在之前那道Word Ladder 词语阶梯里面也有。如果替换后的新词在字典中存在，将其加入words中，并在原有路径的基础上加上这个新词生成一条新路径，如果这个新词就是结束词，则此新路径为一条完整的路径，加入结果中，并更新minLevel，若不是结束词，解将新路径加入路径集中继续循环。

【CODE】

class Solution {

public:

vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

vector<vector<string>> res;

unordered\_set<string> dict(wordList.begin(), wordList.end());

vector<string> p{beginWord};

queue<vector<string>> paths;

paths.push(p);

int level = 1, minLevel = INT\_MAX;

unordered\_set<string> words;

while (!paths.empty()) {

auto t = paths.front(); paths.pop();

if (t.size() > level) {

for (string w : words) dict.erase(w);

words.clear();

level = t.size();

if (level > minLevel) break;

}

string last = t.back();

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

string newLast = last;

for (char ch = 'a'; ch <= 'z'; ++ch) {

newLast[i] = ch;

if (!dict.count(newLast)) continue;

words.insert(newLast);

vector<string> nextPath = t;

nextPath.push\_back(newLast);

if (newLast == endWord) {

res.push\_back(nextPath);

minLevel = level;

} else paths.push(nextPath);

}

}

}

return res;

}

};

## 128. Longest Consecutive Sequence最长连续序列

给定一个未排序的整数数组，找出最长连续序列的长度。

要求算法的时间复杂度为 O(n)。

【分析】要求求最长连续序列，并给定了O(n)复杂度限制，我们的思路是，使用一个集合set存入所有的数字，然后遍历数组中的每个数字，如果其在集合中存在，那么将其移除，然后分别用两个变量pre和next算出其前一个数跟后一个数，然后在集合中循环查找，如果pre在集合中，那么将pre移除集合，然后pre再自减1，直至pre不在集合之中，对next采用同样的方法，那么next-pre-1就是当前数字的最长连续序列，更新res即可。

【CODE】

class Solution {

public:

int longestConsecutive(vector<int>& nums) {

int res = 0;

unordered\_set<int> s(nums.begin(), nums.end());

for (int val : nums) {

if (!s.count(val)) continue;

s.erase(val);

int pre = val - 1, next = val + 1;

while (s.count(pre)) s.erase(pre--);

while (s.count(next)) s.erase(next++);

res = max(res, next - pre - 1);

}

return res;

}

};

【分析】采用哈希表来做，刚开始哈希表为空，然后遍历所有数字，如果该数字不在哈希表中，那么我们分别看其左右两个数字是否在哈希表中，如果在，则返回其哈希表中映射值，若不在，则返回0，然后我们将left+right+1作为当前数字的映射，并更新res结果，然后更新d-left和d-right的映射值。

【CODE】

class Solution {

public:

int longestConsecutive(vector<int>& nums) {

int res = 0;

unordered\_map<int, int> m;

for (int d : nums) {

if (!m.count(d)) {

int left = m.count(d - 1) ? m[d - 1] : 0;

int right = m.count(d + 1) ? m[d + 1] : 0;

int sum = left + right + 1;

m[d] = sum;

res = max(res, sum);

m[d - left] = sum;

m[d + right] = sum;

}

}

return res;

}

};

## 152. Maximum Product Subarray乘积最大子序列

给定一个整数数组 nums ，找出一个序列中乘积最大的连续子序列（该序列至少包含一个数）。

【分析】求最大子数组乘积问题是由最大子数组之和问题演变而来，但是却比求最大子数组之和要复杂，因为在求和的时候，遇到0，不会改变最大值，遇到负数，也只是会减小最大值而已。而在求最大子数组乘积的问题中，遇到0会使整个乘积为0，而遇到负数，则会使最大乘积变成最小乘积，正因为有负数和0的存在，使问题变得复杂了不少。。

比如，我们现在有一个数组[2, 3, -2, 4]，我们可以很容易的找出所有的连续子数组，[2], [3], [-2], [4], [2, 3], [3, -2], [-2, 4], [2, 3, -2], [3, -2, 4], [2, 3, -2, 4], 然后可以很轻松的算出最大的子数组乘积为6，来自子数组[2, 3].

那么我们如何写代码来实现自动找出最大子数组乘积呢，我最先想到的方比较简单粗暴，就是找出所有的子数组，然后算出每一个子数组的乘积，然后比较找出最大的一个，需要两个for循环，第一个for遍历整个数组，第二个for遍历含有当前数字的子数组，就是按以下顺序找出子数组: [2], [2, 3], [2, 3, -2], [2, 3, -2, 4], [3], [3, -2], [3, -2, 4], [-2], [-2, 4], [4], 我在本地测试的一些数组全部通过，于是兴高采烈的拿到OJ上测试，结果丧心病狂的OJ用一个有15000个数字的数组来测试，然后说我程序的运行时间超过了要求值，我一看我的代码，果然如此，时间复杂度O(n2), 得想办法只用一次循环搞定。我想来想去想不出好方法，于是到网上搜各位大神的解决方法。其实这道题最直接的方法就是用DP来做，而且要用两个dp数组，其中f[i]表示子数组[0, i]范围内的最大子数组乘积，g[i]表示子数组[0, i]范围内的最小子数组乘积，初始化时f[0]和g[0]都初始化为nums[0]，其余都初始化为0。那么从数组的第二个数字开始遍历，那么此时的最大值和最小值只会在这三个数字之间产生，即f[i-1]\*nums[i]，g[i-1]\*nums[i]，和nums[i]。所以我们用三者中的最大值来更新f[i]，用最小值来更新g[i]，然后用f[i]来更新结果res即可。

【CODE】

class Solution {

public:

int maxProduct(vector<int>& nums) {

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

vector<int> f(n, 0), g(n, 0);

f[0] = nums[0];

g[0] = nums[0];

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

f[i] = max(max(f[i - 1] \* nums[i], g[i - 1] \* nums[i]), nums[i]);

g[i] = min(min(f[i - 1] \* nums[i], g[i - 1] \* nums[i]), nums[i]);

res = max(res, f[i]);

}

return res;

}

};

## 153. Find Minimum in Rotated Sorted Array寻找旋转排序数组中的最小值

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

【分析】考虑将时间复杂度从简单粗暴的O(n)缩小到O(lgn)，这时候二分查找法就浮现在脑海。首先要判断这个有序数组是否旋转了，通过比较第一个和最后一个数的大小，如果第一个数小，则没有旋转，直接返回这个数。如果第一个数大，就要进一步搜索。我们定义left和right两个指针分别指向开头和结尾，还要找到中间那个数，然后和left指的数比较，如果中间的数大，则继续二分查找右半段数组，反之查找左半段。终止条件是当左右两个指针相邻，返回小的那个。

【CODE】

class Solution {

public:

int findMin(vector<int> &num) {

int left = 0, right = num.size() - 1;

if (num[left] > num[right]) {

while (left != (right - 1)) {

int mid = (left + right) / 2;

if (num[left] < num[mid]) left = mid;

else right = mid;

}

return min(num[left], num[right]);

}

return num[0];

}

};

## 154. Find Minimum in Rotated Sorted Array II寻找旋转排序数组中的最小值 II

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

【分析】当数组中存在大量的重复数字时，就会破坏二分查找法的机制，我们无法取得O(lgn)的时间复杂度，又将会回到简单粗暴的O(n)，比如如下两种情况：

{2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} 和 {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}， 我们发现，当第一个数字和最后一个数字，还有中间那个数字全部相等的时候，二分查找法就崩溃了，因为它无法判断到底该去左半边还是右半边。这种情况下，我们将左指针右移一位，略过一个相同数字，这对结果不会产生影响，因为我们只是去掉了一个相同的，然后对剩余的部分继续用二分查找法，在最坏的情况下，比如数组所有元素都相同，时间复杂度会升到O(n)。

【CODE】

class Solution {

public:

int findMin(vector<int> &nums) {

if (nums.empty()) return 0;

int left = 0, right = nums.size() - 1, res = nums[0];

while (left < right - 1) {

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

if (nums[left] < nums[mid]) {

res = min(res, nums[left]);

left = mid + 1;

} else if (nums[left] > nums[mid]) {

res = min(res, nums[right]);

right = mid;

} else ++left;

}

res = min(res, nums[left]);

res = min(res, nums[right]);

return res;

}

};

## 209. Minimum Size Subarray Sum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 的长度最小的子数组。如果不存在符合条件的子数组，返回 0。

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。

【分析】看O(nlgn)的解法，这个解法要用到二分查找法，思路是，我们建立一个比原数组长一位的sums数组，其中sums[i]表示nums数组中[0, i - 1]的和，然后我们对于sums中每一个值sums[i]，用二分查找法找到子数组的右边界位置，使该子数组之和大于sums[i] + s，然后我们更新最短长度的距离即可。

【CODE】// O(nlgn)

class Solution {

public:

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

int len = nums.size(), sums[len + 1] = {0}, res = len + 1;

for (int i = 1; i < len + 1; ++i) sums[i] = sums[i - 1] + nums[i - 1];

for (int i = 0; i < len + 1; ++i) {

int right = searchRight(i + 1, len, sums[i] + s, sums);

if (right == len + 1) break;

if (res > right - i) res = right - i;

}

return res == len + 1 ? 0 : res;

}

int searchRight(int left, int right, int key, int sums[]) {

while (left <= right) {

int mid = (left + right) / 2;

if (sums[mid] >= key) right = mid - 1;

else left = mid + 1;

}

return left;

}

};

## 216. Combination Sum III组合总和 III

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

说明:所有数字都是正整数。

解集不能包含重复的组合。

【分析】n是k个数字之和，如果n小于0，则直接返回，如果n正好等于0，而且此时out中数字的个数正好为k，说明此时是一个正确解，将其存入结果res中。

【CODE】

class Solution {

public:

vector<vector<int> > combinationSum3(int k, int n) {

vector<vector<int> > res;

vector<int> out;

combinationSum3DFS(k, n, 1, out, res);

return res;

}

void combinationSum3DFS(int k, int n, int level, vector<int> &out, vector<vector<int> > &res) {

if (n < 0) return;

if (n == 0 && out.size() == k) res.push\_back(out);

for (int i = level; i <= 9; ++i) {

out.push\_back(i);

combinationSum3DFS(k, n - i, i + 1, out, res);

out.pop\_back();

}

}

};

## 162. Find Peak Element 寻找峰值

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 nums，其中 nums[i] ≠ nums[i+1]，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 nums[-1] = nums[n] = -∞。

说明:你的解法应该是 O(logN) 时间复杂度的。

【分析】说明了局部峰值一定存在，那么实际上可以从第二个数字开始往后遍历，如果第二个数字比第一个数字小，说明此时第一个数字就是一个局部峰值；否则就往后继续遍历，现在是个递增趋势，如果此时某个数字小于前面那个数字，说明前面数字就是一个局部峰值，返回位置即可。如果循环结束了，说明原数组是个递增数组，返回最后一个位置即可。

【CODE】

class Solution {

public:

int findPeakElement(vector<int>& nums) {

for (int i = 1; i < nums.size(); ++i) {

if (nums[i] < nums[i - 1]) return i - 1;

}

return nums.size() - 1;

}

};

## 228. Summary Ranges汇总区间

给定一个无重复元素的有序整数数组，返回数组区间范围的汇总。

输入: [0,1,2,4,5,7]

输出: ["0->2","4->5","7"]

解释: 0,1,2 可组成一个连续的区间; 4,5 可组成一个连续的区间。

输入: [0,2,3,4,6,8,9]

输出: ["0","2->4","6","8->9"]

解释: 2,3,4 可组成一个连续的区间; 8,9 可组成一个连续的区间。

【分析】具体来说就是让我们找出连续的序列，然后首尾两个数字之间用个“->"来连接，那么我只需遍历一遍数组即可，每次检查下一个数是不是递增的，如果是，则继续往下遍历，如果不是了，我们还要判断此时是一个数还是一个序列，一个数直接存入结果，序列的话要存入首尾数字和箭头“->"。我们需要两个变量i和j，其中i是连续序列起始数字的位置，j是连续数列的长度，当j为1时，说明只有一个数字，若大于1，则是一个连续序列。

【CODE】

class Solution {

public:

vector<string> summaryRanges(vector<int>& nums) {

vector<string> res;

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

while (i < n) {

int j = 1;

while (i + j < n && nums[i + j] - nums[i] == j) ++j;

res.push\_back(j <= 1 ? to\_string(nums[i]) : to\_string(nums[i]) + "->" + to\_string(nums[i + j - 1]));

i += j;

}

return res;

}

};

## 229. Majority Element II求众数 II

给定一个大小为 n 的数组，找出其中所有出现超过 ⌊ n/3 ⌋ 次的元素。

说明: 要求算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

【知识点】摩尔投票法Moore Voting

是一种在线性时间O(n)和空间复杂度的情况下，在一个元素序列中查找包含最多的元素。

算法在局部变量中定义一个序列元素(m)和一个计数器(i)，初始化的情况下计数器为0. 算法依次扫描序列中的元素，当处理元素x的时候，如果计数器为0，那么将x赋值给m，然后将计数器(i)设置为1，如果计数器不为0，那么将序列元素m和x比较，如果相等，那么计数器加1，如果不等，那么计数器减1。处理之后，最后存储的序列元素(m)，就是这个序列中最多的元素。

如果不确定是否存储的元素m是最多的元素，还可以进行第二遍扫描判断是否为最多的元素。

【分析】让我们求出现次数大于n/3的众数，而且限定了时间和空间复杂度，那么就不能排序，也不能使用哈希表，这么苛刻的限制条件只有一种方法能解了，那就是摩尔投票法 Moore Voting，这种方法在之前那道题Majority Element 求众数中也使用了。题目中给了一条很重要的提示，让我们先考虑可能会有多少个众数，经过举了很多例子分析得出，任意一个数组出现次数大于n/3的众数最多有两个，具体的证明我就不会了，我也不是数学专业的。那么有了这个信息，我们使用投票法的核心是找出两个候选众数进行投票，需要两遍遍历，第一遍历找出两个候选众数，第二遍遍历重新投票验证这两个候选众数是否为众数即可，选候选众数方法和前面那篇Majority Element 求众数一样，由于之前那题题目中限定了一定会有众数存在，故而省略了验证候选众数的步骤，这道题却没有这种限定，即满足要求的众数可能不存在，所以要有验证。

【CODE】

class Solution {

public:

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

vector<int> res;

int m = 0, n = 0, cm = 0, cn = 0;

for (auto &a : nums) {

if (a == m) ++cm;

else if (a ==n) ++cn;

else if (cm == 0) m = a, cm = 1;

else if (cn == 0) n = a, cn = 1;

else --cm, --cn;

}

cm = cn = 0;

for (auto &a : nums) {

if (a == m) ++cm;

else if (a == n) ++cn;

}

if (cm > nums.size() / 3) res.push\_back(m);

if (cn > nums.size() / 3) res.push\_back(n);

return res;

}

};

## 238. Product of Array Except Self除自身以外数组的乘积

给定长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output ，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

说明: 请不要使用除法，且在 O(n) 时间复杂度内完成此题。

进阶：你可以在常数空间复杂度内完成这个题目吗？（ 出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

【分析】对于某一个数字，如果我们知道其前面所有数字的乘积，同时也知道后面所有的数乘积，那么二者相乘就是我们要的结果，所以我们只要分别创建出这两个数组即可，分别从数组的两个方向遍历就可以分别创建出乘积累积数组。

【CODE】

class Solution {

public:

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

int n = nums.size();

vector<int> fwd(n, 1), bwd(n, 1), res(n);

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

fwd[i + 1] = fwd[i] \* nums[i];

}

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

bwd[i - 1] = bwd[i] \* nums[i];

}

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

res[i] = fwd[i] \* bwd[i];

}

return res;

}

};

【分析】可以对上面的方法进行空间上的优化，由于最终的结果都是要乘到结果res中，所以我们可以不用单独的数组来保存乘积，而是直接累积到res中，我们先从前面遍历一遍，将乘积的累积存入res中，然后从后面开始遍历，用到一个临时变量right，初始化为1，然后每次不断累积，最终得到正确结果。

【CODE】

class Solution {

public:

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

vector<int> res(nums.size(), 1);

for (int i = 1; i < nums.size(); ++i) {

res[i] = res[i - 1] \* nums[i - 1];

}

int right = 1;

for (int i = nums.size() - 1; i >= 0; --i) {

res[i] \*= right;

right \*= nums[i];

}

return res;

}

};

## 283. Move Zeroes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。

说明:必须在原数组上操作，不能拷贝额外的数组。

尽量减少操作次数。

【分析】替换法in-place来做，需要用两个指针，一个不停的向后扫，找到非零位置，然后和前面那个指针交换位置即可。

【CODE】

class Solution {

public:

void moveZeroes(vector<int>& nums) {

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

if (nums[i]) {

swap(nums[i], nums[j++]);

}

}

}

};

## 287. Find the Duplicate Number寻找重复数

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。  
说明：

不能更改原数组（假设数组是只读的）。

只能使用额外的 O(1) 的空间。

时间复杂度小于 O(n2) 。

数组中只有一个重复的数字，但它可能不止重复出现一次。

【分析】这道题给了我们n+1个数，所有的数都在[1, n]区域内，首先让我们证明必定会有一个重复数，这不禁让我想起了小学华罗庚奥数中的抽屉原理(又叫鸽巢原理), 即如果有十个苹果放到九个抽屉里，如果苹果全在抽屉里，则至少有一个抽屉里有两个苹果，这里就不证明了，直接来做题吧。题目要求我们不能改变原数组，即不能给原数组排序，又不能用多余空间，那么哈希表神马的也就不用考虑了，又说时间小于O(n2)，也就不能用brute force的方法，那我们也就只能考虑用二分搜索法了，我们在区别[1, n]中搜索，首先求出中点mid，然后遍历整个数组，统计所有小于等于mid的数的个数，如果个数大于mid，则说明重复值在[mid+1, n]之间，反之，重复值应在[1, mid-1]之间，然后依次类推，直到搜索完成，此时的low就是我们要求的重复值。

【CODE】class Solution {

public:

int findDuplicate(vector<int>& nums) {

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

while (low < high) {

int mid = low + (high - low) \* 0.5;

int cnt = 0;

for (auto a : nums) {

if (a <= mid) ++cnt;

}

if (cnt <= mid) low = mid + 1;

else high = mid;

}

return low;

}

};

【分析】题目限定了区间[1,n]，所以可以巧妙的利用坐标和数值之间相互转换，而由于重复数字的存在，那么一定会形成环，我们用快慢指针可以找到环并确定环的起始位置。

【CODE】

class Solution {

public:

int findDuplicate(vector<int>& nums) {

int slow = 0, fast = 0, t = 0;

while (true) {

slow = nums[slow];

fast = nums[nums[fast]];

if (slow == fast) break;

}

while (true) {

slow = nums[slow];

t = nums[t];

if (slow == t) break;

}

return slow;

}

};

## 289. Game of Life生命游戏

生命游戏，简称为生命，是英国数学家约翰·何顿·康威在1970年发明的细胞自动机。

给定一个包含 m × n 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞具有一个初始状态 live（1）即为活细胞， 或 dead（0）即为死细胞。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；

如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；

如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；

如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

根据当前状态，写一个函数来计算面板上细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

|  |  |
| --- | --- |
| 输入:  [  [0,1,0],  [0,0,1],  [1,1,1],  [0,0,0]  ] | 输出:  [  [0,0,0],  [1,0,1],  [0,1,1],  [0,1,0]  ] |

进阶:

你可以使用原地算法解决本题吗？请注意，面板上所有格子需要同时被更新：你不能先更新某些格子，然后使用它们的更新后的值再更新其他格子。

本题中，我们使用二维数组来表示面板。原则上，面板是无限的，但当活细胞侵占了面板边界时会造成问题。你将如何解决这些问题？

【分析】这道题是有名的康威生命游戏, 这是一种细胞自动机，每一个位置有两种状态，1为活细胞，0为死细胞，对于每个位置都满足如下的条件：

1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡

2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活

3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡

4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活

由于题目中要求我们用置换方法in-place来解题，所以我们就不能新建一个相同大小的数组，那么我们只能更新原有数组，但是题目中要求所有的位置必须被同时更新，但是在循环程序中我们还是一个位置一个位置更新的，那么当一个位置更新了，这个位置成为其他位置的neighbor时，我们怎么知道其未更新的状态呢，我们可以使用状态机转换：

状态0： 死细胞转为死细胞

状态1： 活细胞转为活细胞

状态2： 活细胞转为死细胞

状态3： 死细胞转为活细胞

最后我们对所有状态对2取余，那么状态0和2就变成死细胞，状态1和3就是活细胞，达成目的。我们先对原数组进行逐个扫描，对于每一个位置，扫描其周围八个位置，如果遇到状态1或2，就计数器累加1，扫完8个邻居，如果少于两个活细胞或者大于三个活细胞，而且当前位置是活细胞的话，标记状态2，如果正好有三个活细胞且当前是死细胞的话，标记状态3。完成一遍扫描后再对数据扫描一遍，对2取余变成我们想要的结果。

【CODE】

class Solution {

public:

void gameOfLife(vector<vector<int> >& board) {

int m = board.size(), n = m ? board[0].size() : 0;

int dx[] = {-1, -1, -1, 0, 1, 1, 1, 0};

int dy[] = {-1, 0, 1, 1, 1, 0, -1, -1};

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

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

int cnt = 0;

for (int k = 0; k < 8; ++k) {

int x = i + dx[k], y = j + dy[k];

if (x >= 0 && x < m && y >= 0 && y < n && (board[x][y] == 1 || board[x][y] == 2)) {

++cnt;

}

}

if (board[i][j] && (cnt < 2 || cnt > 3)) board[i][j] = 2;

else if (!board[i][j] && cnt == 3) board[i][j] = 3;

}

}

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

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

board[i][j] %= 2;

}

}

}

};

## 380. Insert Delete GetRandom O(1)常数时间插入、删除和获取随机元素

设计一个支持在平均 时间复杂度 O(1) 下，执行以下操作的数据结构。

insert(val)：当元素 val 不存在时，向集合中插入该项。

remove(val)：元素 val 存在时，从集合中移除该项。

getRandom：随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

【分析】在常数时间范围内实现插入删除和获得随机数操作，如果这道题没有常数时间的限制，那么将会是一道非常简单的题，我们直接用一个set就可以搞定所有的操作。但是由于时间的限制，我们无法在常数时间内实现获取随机数，所以只能另辟蹊径。此题的正确解法是利用到了一个一维数组和一个哈希表，其中数组用来保存数字，哈希表用来建立每个数字和其在数组中的位置之间的映射，对于插入操作，我们先看这个数字是否已经在哈希表中存在，如果存在的话直接返回false，不存在的话，我们将其插入到数组的末尾，然后建立数字和其位置的映射。删除操作是比较巧妙的，我们还是要先判断其是否在哈希表里，如果没有，直接返回false。由于哈希表的删除是常数时间的，而数组并不是，为了使数组删除也能常数级，我们实际上将要删除的数字和数组的最后一个数字调换个位置，然后修改对应的哈希表中的值，这样我们只需要删除数组的最后一个元素即可，保证了常数时间内的删除。而返回随机数对于数组来说就很简单了，我们只要随机生成一个位置，返回该位置上的数字即可。

【CODE】

class RandomizedSet {

public:

/\*\* Initialize your data structure here. \*/

RandomizedSet() {}

/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

bool insert(int val) {

if (m.count(val)) return false;

nums.push\_back(val);

m[val] = nums.size() - 1;

return true;

}

/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

bool remove(int val) {

if (!m.count(val)) return false;

int last = nums.back();

m[last] = m[val];

nums[m[val]] = last;

nums.pop\_back();

m.erase(val);

return true;

}

/\*\* Get a random element from the set. \*/

int getRandom() {

return nums[rand() % nums.size()];

}

private:

vector<int> nums;

unordered\_map<int, int> m;

};