1.编辑距离

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character  
b) Delete a character  
c) Replace a character

这道题让求从一个字符串转变到另一个字符串需要的变换步骤，共有三种变换方式，插入一个字符，删除一个字符，和替换一个字符。根据以往的经验，对于字符串相关的题目十有八九都是用动态规划Dynamic Programming来解，这道题也不例外。这道题我们需要维护一个二维的数组dp，其中dp[i][j]表示从word1的前i个字符转换到word2的前j个字符所需要的步骤。那我们可以先给这个二维数组dp的第一行第一列赋值，这个很简单，因为第一行和第一列对应的总有一个字符串是空串，于是转换步骤完全是另一个字符串的长度。跟以往的DP题目类似，难点还是在于找出递推式，我们可以举个例子来看，比如word1是“bbc"，word2是”abcd“，那么我们可以得到dp数组如下：

class Solution {

public:

int minDistance(string word1, string word2) {

int n1 = word1.size(), n2 = word2.size();

int dp[n1 + 1][n2 + 1];

for (int i = 0; i <= n1; ++i) dp[i][0] = i;

for (int i = 0; i <= n2; ++i) dp[0][i] = i;

for (int i = 1; i <= n1; ++i) {

for (int j = 1; j <= n2; ++j) {

if (word1[i ] == word2[j ]) {

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

} else {

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

}

}

}

return dp[n1][n2];

}

};

2.矩阵赋0

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

[click to show follow up.](https://leetcode.com/problems/set-matrix-zeroes/)

Follow up:

Did you use extra space?  
A straight forward solution using O(mn) space is probably a bad idea.  
A simple improvement uses O(m + n) space, but still not the best solution.  
Could you devise a constant space solution?

据说这题是CareerCup上的原题，我还没有刷CareerCup，所以不知道啦，不过这题也不算难，虽然我也是看了网上的解法照着写的，但是下次遇到绝对想的起来。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

代码如下:

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;

}

}

};

3.三色球使得相同的球挨在一起，并按一定颜色顺序排列

推广：将一个数组按小于10，等于10，大于10排列，也是这样设置两个指针然后一遍遍历

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:  
You are not suppose to use the library's sort function for this problem.

[click to show follow up.](https://leetcode.com/problems/sort-colors/)

Follow up:  
A rather straight forward solution is a two-pass algorithm using counting sort.  
First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

题目中还要让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。

class Solution {

public:

void sortColors(int A[], int n) {

int red = 0, blue = n - 1;

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

if (A[i] == 0) {

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

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

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

}

}

}

};

4.最小窗口子串

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,  
S = "ADOBECODEBANC"  
T = "ABC"

Minimum window is "BANC".

Note:  
If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

这道题的要求是要在O(n)的时间度里实现找到这个最小窗口字串，那么暴力搜索Brute Force肯定是不能用的，我们可以考虑哈希表，其中key是T中的字符，value是该字符出现的次数。

- 我们最开始先扫描一遍T，把对应的字符及其出现的次数存到哈希表中。

- 然后开始遍历S，遇到T中的字符，就把对应的哈希表中的value减一，直到包含了T中的所有的字符，纪录一个字串并更新最小字串值。

- 将子窗口的左边界向右移，略掉不在T中的字符，如果某个在T中的字符出现的次数大于哈希表中的value，则也可以跳过该字符。

class Solution {

public:

string minWindow(string S, string T) {

if (T.size() > S.size()) return "";

string res = "";

int left = 0, count = 0, minLen = S.size() + 1;

unordered\_map<char, int> m;

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

if (m.find(T[i]) != m.end()) ++m[T[i]];

else m[T[i]] = 1;

}

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

if (m.find(S[right]) != m.end()) {

--m[S[right]];

if (m[S[right]] >= 0) ++count;

while (count == T.size()) {

if (right - left + 1 < minLen) {

minLen = right - left + 1;

res = S.substr(left, minLen);

}

if (m.find(S[left]) != m.end()) {

++m[S[left]];

if (m[S[left]] > 0) --count;

}

++left;

}

}

}

return res;

}

};

5.组合项

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,  
If n = 4 and k = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

这道题让求1到n共n个数字里k个数的组合数的所有情况，还是要用深度优先搜索DFS来解，根据以往的经验，像这种要求出所有结果的集合，一般都是用DFS调用递归来解。那么我们建立一个保存最终结果的大集合res，还要定义一个保存每一个组合的小集合out，每次放一个数到out里，如果out里数个数到了k个，则把out保存到最终结果中，否则在下一层中继续调用递归。网友[u010500263的博客](http://blog.csdn.net/u010500263/article/details/18435495" \t "http://www.cnblogs.com/grandyang/p/_blank)里有一张图很好的说明了递归调用的顺序，请点击[这里](http://blog.csdn.net/u010500263/article/details/18435495" \t "http://www.cnblogs.com/grandyang/p/_blank)。根据上面分析，可写出代码如下：

class Solution {

public:

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

vector<vector<int> > res;

vector<int> out;

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

return res;

}

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

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

else {

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

out.push\_back(i);

combineDFS(n, k, i + 1, out, res);

out.pop\_back();

}

}

}

};

6.子集合

Given a set of distinct integers, S, return all possible subsets.

Note:

* Elements in a subset must be in non-descending order.
* The solution set must not contain duplicate subsets.

For example,  
If S = [1,2,3], a solution is:

[

[3],

[1],

[2],

[1,2,3],

[1,3],

[2,3],

[1,2],

[]

]

下面来看递归的解法，相当于一种深度优先搜索，参见网友[JustDoIt的博客](http://www.cnblogs.com/TenosDoIt/p/3451902.html" \t "http://www.cnblogs.com/grandyang/p/_blank)，由于原集合每一个数字只有两种状态，要么存在，要么不存在，那么在构造子集时就有选择和不选择两种情况，所以可以构造一棵二叉树，左子树表示选择该层处理的节点，右子树表示不选择，最终的叶节点就是所有子集合，树的结构如下：

[]

/ \

/ \

/ \

[1] []

/ \ / \

/ \ / \

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

/ \ / \ / \ / \

[1 2 3] [1 2] [1 3] [1] [2 3] [2] [3] []

// Recursion

class Solution {

public:

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

vector<vector<int> > res;

vector<int> out;

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

getSubsets(S, 0, out, res);

return res;

}

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

res.push\_back(out);

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

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

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

out.pop\_back();

}

}

};

7.Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

* Elements in a subset must be in non-descending order.
* The solution set must not contain duplicate subsets.

For example,  
If S = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

对于递归的解法，根据之前[Subsets 子集合](http://www.cnblogs.com/grandyang/p/4309345.html) 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下图所示：

[]

/ \

/ \

/ \

[1] []

/ \ / \

/ \ / \

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

/ \ / \ / \ / \

[1 2 2] [1 2] X [1] [2 2] [2] X []

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。代码如下：

// Recursion

class Solution {

public:

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

vector<vector<int> > res;

vector<int> out;

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

getSubsets(S, 0, out, res);

return res;

}

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

res.push\_back(out);

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

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

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

out.pop\_back();

while (S[i] == S[i + 1]) ++i;

}

}

};

8.Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,  
Given board =

[

["ABCE"],

["SFCS"],

["ADEE"]

]

word = "ABCCED", -> returns true,  
word = "SEE", -> returns true,  
word = "ABCB", -> returns false.

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，具体看代码实现如下：

class Solution {

public:

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

if (word.empty()) return true;

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

vector<vector<bool> > visited(board.size(), vector<bool>(board[0].size(), false));

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

for (int j = 0; j < board[i].size(); ++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;

if (i < 0 || j < 0 || i >= board.size() || j >= board[0].size() || 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;

}

};

9.Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

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了，代码如下：

class Solution {

public:

int search(int A[], int n, int target) {

if (n == 0) return -1;

int left = 0, right = n - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

} else {

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

else left = mid + 1;

}

}

return -1;

}

};

10.Follow up for "Search in Rotated Sorted Array":  
What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

这道是之前那道[Search in Rotated Sorted Array 在旋转有序数组中搜索](http://www.cnblogs.com/grandyang/p/4325648.html) 的延伸，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](http://www.cnblogs.com/grandyang/p/4325648.html) 中的方法，可以得到代码如下：

class Solution {

public:

bool search(int A[], int n, int target) {

if (n == 0) return false;

int left = 0, right = n - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

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

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

else left = mid + 1;

} else --right;

}

return false;

}

};

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

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

\* };

\*/

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

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

ListNode \*start = new ListNode(0);

start->next = head;

ListNode \*pre = start;

while (pre->next) {

ListNode \*cur = pre->next;

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

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

else pre = pre->next;

}

return start->next;

}

};

11.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，当当前值小于栈顶值时，取出栈顶元素，然后计算当前矩形面积，然后再对比当前值和新的栈顶值大小，若还是栈顶值大，则再取出栈顶，算此时共同矩形区域面积，照此类推，可得最大矩形。代码如下：

class Solution {

public:

int largestRectangleArea(vector<int> &height) {

int res = 0;

stack<int> s;

height.push\_back(0);

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

if (s.empty() || height[s.top()] < height[i]) s.push(i);

else {

int cur = s.top();

s.pop();

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

--i;

}

}

return res;

}

};

12.划分链表

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,  
Given 1->4->3->2->5->2 and x = 3,  
return 1->2->2->4->3->5.

这道题要求我们划分链表，把所有小于给定值的节点都移到前面，大于该值的节点顺序不变，相当于一个局部排序的问题。那么可以想到的一种解法是首先找到第一个大于或等于给定值的节点，用题目中给的例子来说就是先找到4，然后再找小于3的值，每找到一个就将其取出置于4之前即可，代码如下：

class Solution {

public:

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

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

dummy->next = head;

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

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

cur = pre;

while (cur->next) {

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

ListNode \*tmp = cur->next;

cur->next = tmp->next;

tmp->next = pre->next;

pre->next = tmp;

pre = pre->next;//始终指向小于x的链表的最后一个

} else {

cur = cur->next;

}

}

return dummy->next;

}

};

13.混合有序数组

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:  
You may assume that A has enough space (size that is greater or equal to m + n) to hold additional elements from B. The number of elements initialized in A and B are m andn respectively.

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

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

class Solution {

public:

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

int count = m + n - 1;

--m; --n;

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

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

}

};

14.Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:  
Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

public class Solution {

public List<String> restoreIpAddresses(String s) {

List<String> res = new ArrayList<String>();

helper(s, 0, "", res);

return res;

}

public void helper(String s, int n, String out, List<String> res) {

if (n == 4) {

if (s.isEmpty()) res.add(out);

return;

}

for (int k = 1; k < 4; ++k) {

if (s.length() < k) break;

int val = Integer.parseInt(s.substring(0, k));

if (val > 255 || k != String.valueOf(val).length()) continue;

helper(s.substring(k), n + 1, out + s.substring(0, k) + (n == 3 ? "" : "."), res);

}

}

}

15.共线点个数

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

这道题给了我们一堆二维点，然后让我们求最大的共线点的个数，根据初中数学我们知道，两点确定一条直线，而且可以写成y = ax + b的形式，所有共线的点都满足这个公式。所以这些给定点两两之间都可以算一个斜率，每个斜率代表一条直线，对每一条直线，带入所有的点看是否共线并计算个数，这是整体的思路。但是还有两点特殊情况需要考虑，二是当两个点重合时，无法确定一条直线，但这也是共线的情况，需要特殊处理。二是斜率不存在的情况，由于两个点(x1, y1)和(x2, y2)的斜率k表示为(y2 - y1) / (x2 - x1)，那么当x1 = x2时斜率不存在，这种共线情况需要特殊处理。我们需要用到哈希表来记录斜率和共线点个数之间的映射，其中第一种重合点的情况我们假定其斜率为INT\_MIN，第二种情况我们假定其斜率为INT\_MAX，这样都可以用map映射了。我们还需要顶一个变量duplicate来记录重合点的个数，最后只需和哈希表中的数字相加即为共线点的总数。代码如下：

class Solution {

public:

int maxPoints(vector<Point>& points) {

int res = 0;

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

unordered\_map<float, int> m;

int duplicate = 1;

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

if (points[i].x == points[j].x && points[i].y == points[j].y) {

++duplicate;

} else if (points[i].x == points[j].x) {

++m[INT\_MAX];

} else {

float slope = (float)(points[j].y - points[i].y) / (points[j].x - points[i].x);

++m[slope];

}

}

res = max(res, duplicate);

for (auto it = m.begin(); it != m.end(); ++it) {

res = max(res, it->second + duplicate);

}

}

return res;

}

};

16.Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

这道寻找旋转有序数组的最小值肯定不能通过直接遍历整个数组来寻找，这个方法过于简单粗暴，这样的话，旋不旋转就没有意义。应该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。代码如下：

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];

}

};

17.Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

The array may contain duplicates.

寻找旋转有序重复数组的最小值是对之前问题的延伸([http://www.cnblogs.com/grandyang/p/4032934.html](http://www.cnblogs.com/grandyang/p/4032934.html" \t "http://www.cnblogs.com/grandyang/p/_blank))，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)，参见代码如下：

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;

}

};

18.奇数位为奇数，偶数位为偶数

设定 odd even两个标志位为odd为1，even为0，然后从最后一位开始看，如果说最后一位为偶数，就和even位交换，even自增2，如果是奇数，则和odd位交换，odd自增2，一直到二者其中一个越界位置。   解法比较奇特。

1. #include<iostream>
2. #include <hash\_map>
3. #include <algorithm>
4. **using** **namespace** std;
5. //改变数组，令其奇数位置存储奇数，偶数位置存储偶数
6. //要求空间复杂度为O（1），时间复杂度为O（n）
7. **int** a[]={9,6,5,4,3,2,1,8};
8. **void** ChangeArray(**int** len)
9. {
10. **int** end=a[len-1];
11. **int** even=0;
12. **int** odd=1;
13. **while**(even<len&&odd<len)
14. {
15. **if**(a[len-1]%2==0)
16. {
17. std::swap(a[len-1],a[even]);
18. even+=2;
19. }
20. **else**
21. {
22. std::swap(a[len-1],a[odd]);
23. odd+=2;
24. }
25. }
27. }
29. **int** main()
30. {
31. ChangeArray(8);
32. **return** 0;
33. }

19.在一列数组中取出一个或多个不相邻数，使其和最大

分别维护两个变量a和b，然后按奇偶分别来更新a和b，这样就可以保证组成最大和的数字不相邻，代码如下：

class Solution {

public:

int rob(vector<int> &num) {

int a = 0, b = 0;

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

if (i % 2 == 0) {

a += num[i];

a = max(a, b);

} else {

b += num[i];

b = max(a, b);

}

}

return max(a, b);

}

};

20.旋转数组

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

其实还有种类似翻转字符的方法，思路是先把前n-k个数字翻转一下，再把后k个数字翻转一下，最后再把整个数组翻转一下：

1 2 3 4 5 6 7   
4 3 2 1 5 6 7   
4 3 2 1 7 6 5  
5 6 7 1 2 3 4

class Solution {

public:

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

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

int n = nums.size();

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

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

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

}

};

21.求最大间隔：

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

即：**一个无序的实数数组a[i]，要求求里面大小相邻的实数的差的最大值。比如 double a[]={1,5,4,0.2,100} 这个无序的数组，相邻的数的最大差值为100-5=95.**

遇到这类问题肯定先想到的是要给数组排序，但是题目要求是要线性的时间和空间，那么只能用桶排序或者基排序

桶排序Bucket Sort来做，首先找出数组的最大值和最小值，然后要确定每个桶的容量，即为(最大值 - 最小值) / 个数 + 1，在确定桶的个数，即为(最大值 - 最小值) / 桶的容量 + 1，然后需要在每个桶中找出局部最大值和最小值，而最大间距的两个数不会在同一个桶中，而是一个桶的最小值和另一个桶的最大值之间的间距

class Solution {

public:

int maximumGap(vector<int> &numss) {

if (numss.empty()) return 0;

int mx = INT\_MIN, mn = INT\_MAX, n = numss.size();

for (int d : numss) {

mx = max(mx, d);

mn = min(mn, d);

}

int size = (mx - mn) / n + 1;

int bucket\_nums = (mx - mn) / size + 1;

vector<int> bucket\_min(bucket\_nums, INT\_MAX);

vector<int> bucket\_max(bucket\_nums, INT\_MIN);

set<int> s;

for (int d : numss) {

int idx = (d - mn) / size;

bucket\_min[idx] = min(bucket\_min[idx], d);

bucket\_max[idx] = max(bucket\_max[idx], d);

s.insert(idx);

}

int pre = 0, res = 0;

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

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

res = max(res, bucket\_min[i] - bucket\_max[pre]);

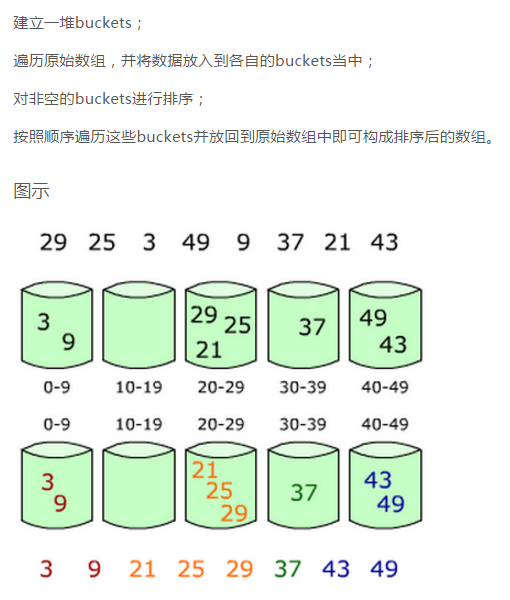
pre = i;

}

return res;

}

};



bucket采用链表结构以充分利用空间资源

N个待排数据，M个桶，桶****排序的平均时间复杂度为线性的O(N+C)，其中C=N\*(logN-logM)。如果相对于同样的N，桶数量M越大，其效率越高，最好的时间复杂度达到O(N)。 当然桶排序的空间复杂度 为O(N+M)，****如果输入数据非常庞大，而桶的数量也非常多，则空间代价无疑是昂贵的。此外，桶排序是稳定的。

****基数排序的思想就是将待排数据中的每组关键字依次进行桶分配****。比如下面的待排序列：

278、109、063、930、589、184、505、269、008、083

我们将每个数值的个位，十位，百位分成三个关键字： 278 -> k1(个位)=8 ，k2(十位)=7 ，k3=(百位)=2。

然后从最低位个位开始(从最次关键字开始)，对所有数据的k1关键字进行桶分配(因为，每个数字都是 0-9的，因此桶大小为10)，再依次输出桶中的数据得到下面的序列。

930、063、083、184、505、278、008、109、589、269

再对上面的序列接着进行针对k2的桶分配，输出序列为：

505、008、109、930、063、269、278、083、184、589

最后针对k3的桶分配，输出序列为：

008、063、083、109、184、269、278、505、589、930

****性能分析****

很明显，****基数排序的性能比桶排序要略差。每一次关键字的桶分配都需要O(N)的时间复杂度，而且分配之后得到新的关键字序列又需要O(N)的时间复杂度。假如待排数据可以分为d个关键字，则基数排序的时间复杂度将是O(d\*2N) ，当然d要远远小于N，因此基本上还是线性级别的。基数排序的空间复杂度为O(N+M)，其中M为桶的数量。一般来说N>>M，因此额外空间需要大概N个左右。****

最长回文子序列

最长回文子串

22.最长回文子序列

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:  
Input:

"bbbab"

Output:

4

One possible longest palindromic subsequence is "bbbb".

给了我们一个字符串，让我们求最大的回文子序列，子序列和子字符串不同，不需要连续

递归形式的解法，memo数组这里起到了一个缓存已经计算过了的结果，这样能提高运算效率，使其不会TLE，参见代码如下：

 class Solution {

public:

int longestPalindromeSubseq(string s) {

int n = s.size();

vector<vector<int>> memo(n, vector<int>(n, -1));

return helper(s, 0,n - 1, m emo);

}

int helper(string& s, int i, int j, vector<vector<int>>& memo) {

if (memo[i][j] != -1) return memo[i][j];

if (i > j) return 0;

if (i == j) return 1;

if (s[i] == s[j]) {

memo[i][j] = helper(s, i + 1, j - 1, memo) + 2;

} else {

memo[i][j] = max(helper(s, i + 1, j, memo), helper(s, i, j - 1, memo));

}

return memo[i][j];

}

};

23.最长回文子串

中心扩展就是把给定的字符串的每一个字母当做中心，向两边扩展，这样来找最长的子回文串。算法复杂度为O(N^2)。

但是要考虑两种情况：

1、像aba，这样长度为奇数。

2、想abba，这样长度为偶数。

1. string findLongestPalindrome(string &s)
2. {
3. **const** **int** length=s.size();
4. **int** maxlength=0;
5. **int** start;
7. **for**(**int** i=0;i<length;i++)//长度为奇数
8. {
9. **int** j=i-1,k=i+1;
10. **while**(j>=0&&k<length&&s.at(j)==s.at(k))
11. {
12. **if**(k-j+1>maxlength)
13. {
14. maxlength=k-j+1;
15. start=j;
16. }
17. j--;
18. k++;
19. }
20. }
22. **for**(**int** i=0;i<length;i++)//长度为偶数
23. {
24. **int** j=i,k=i+1;
25. **while**(j>=0&&k<length&&s.at(j)==s.at(k))
26. {
27. **if**(k-j+1>maxlength)
28. {
29. maxlength=k-j+1;
30. start=j;
31. }
32. j--;
33. k++;
34. }
35. }
36. **if**(maxlength>0)
37. **return** s.substr(start,maxlength);
38. **return** NULL;
39. }

24.找出树每行最大的节点

You need to find the largest value in each row of a binary tree.

Example:

Input:

1

/ \

3 2

/ \ \

5 3 9

Output: [1, 3, 9]

这道题让我们找二叉树每行的最大的结点值，那么实际上最直接的方法就是用层序遍历，然后在每一层中找到最大值，加入结果res中即可，参见代码如下：

class Solution {

public:

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

if (!root) return {};

vector<int> res;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

int n = q.size(), mx = INT\_MIN;

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

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

mx = max(mx, t->val);

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

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

}

res.push\_back(mx);

}

return res;

}

};

25.Given a binary tree, find the leftmost value in the last row of the tree.

Example 1:

Input:

2

/ \

1 3

Output:

1

Example 2:

Input:

1

/ \

2 3

/ / \

4 5 6

/

7

Output:

7

这道题用层序遍历更直接一些，因为层序遍历时遍历完当前行所有结点之后才去下一行，那么我们再遍历每行第一个结点时更新结果res即可参见代码如下：

class Solution {

public:

int findBottomLeftValue(TreeNode\* root) {

if (!root) return 0;

int res = 0;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

int n = q.size();

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

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

if (i == 0) res = t->val;

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

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

}

}

return res;

}

};

26.Given scores of N athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".

Example 1:

Input: [5, 4, 3, 2, 1]

Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]

Explanation: The first three athletes got the top three highest scores, so they got "Gold Medal", "Silver Medal" and "Bronze Medal".   
For the left two athletes, you just need to output their relative ranks according to their scores.

Note:

1. N is a positive integer and won't exceed 10,000.
2. All the scores of athletes are guaranteed to be unique.

这道题给了我们一组分数，让我们求相对排名，前三名分别是金银铜牌，后面的就是名次数，不是一道难题，我们可以利用堆来排序，建立一个优先队列，把分数和其坐标位置放入队列中，会自动按其分数高低排序，然后我们从顶端开始一个一个取出数据，由于保存了其在原数组的位置，我们可以直接将其存到结果res中正确的位置，用一个变量cnt来记录名词，前三名给奖牌，后面就是名次数，参见代码如下：

class Solution {

public:

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

int n = nums.size(), cnt = 1;

vector<string> res(n, "");

priority\_queue<pair<int, int>> q;

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

q.push({nums[i], i});

}

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

int idx = q.top().second; q.pop();

if (cnt == 1) res[idx] = "Gold Medal";

else if (cnt == 2) res[idx] = "Silver Medal";

else if (cnt == 3) res[idx] = "Bronze Medal";

else res[idx] = to\_string(cnt);

++cnt;

}

return res;

}

};

下面这种方法思路和上面一样，不过数据结构用的不同，这里利用map的自动排序的功能，不过map是升序排列的，所以我们遍历的时候就要从最后面开始遍历，最后一个是金牌，然后往前一次是银牌，铜牌，名次数等，参见代码如下：

class Solution {

public:

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

int n = nums.size(), cnt = 1;

vector<string> res(n, "");

map<int, int> m;

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

m[nums[i]] = i;

}

for (auto it = m.rbegin(); it != m.rend(); ++it) {

if (cnt == 1) res[it->second] = "Gold Medal";

else if (cnt == 2) res[it->second] = "Silver Medal";

else if (cnt == 3) res[it->second] = "Bronze Medal";

else res[it->second] = to\_string(cnt);

++cnt;

}

return res;

}

};

27.七进制

Given an integer, return its base 7 string representation.

Example 1:

Input: 100

Output: "202"

Example 2:

Input: -7

Output: "-10"

Note: The input will be in range of [-1e7, 1e7].

这道题给了我们一个数，让我们转为七进制的数，而且这个可正可负。那么我们想如果给一个十进制的100，怎么转为七进制。我会先用100除以49，商2余2。在除以7，商0余2，于是就得到七进制的202。其实我们还可以反过来算，先用100除以7，商14余2，然后用14除以7，商2余0，再用2除以7，商0余2，这样也可以得到202。这种方法更适合于代码实现，要注意的是，我们要处理好负数的情况，参见代码如下：

class Solution {

public:

string convertToBase7(int num) {

if (num == 0) return "0";

string res = "";

bool positive = num > 0;

while (num != 0) {

res = to\_string(abs(num % 7)) + res;

num /= 7;

}

return positive ? res : "-" + res;

}

};

28.下一个较大的元素

You are given two arrays (without duplicates) nums1 and nums2 where nums1’s elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.

The Next Greater Number of a number x in nums1 is the first greater number to its right in nums2. If it does not exist, output -1 for this number.

Example 1:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2].

Output: [-1,3,-1]

Explanation:

For number 4 in the first array, you cannot find the next greater number for it in the second array, so output -1.

For number 1 in the first array, the next greater number for it in the second array is 3.

For number 2 in the first array, there is no next greater number for it in the second array, so output -1.

Example 2:

Input: nums1 = [2,4], nums2 = [1,2,3,4].

Output: [3,-1]

Explanation:

For number 2 in the first array, the next greater number for it in the second array is 3.

For number 4 in the first array, there is no next greater number for it in the second array, so output -1.

Note:

1. All elements in nums1 and nums2 are unique.
2. The length of both nums1 and nums2 would not exceed 1000.

这道题给了我们一个数组，又给了该数组的一个子集合，让我们求集合中每个数字在原数组中右边第一个较大的数字。参考题目中给的例子，题意不难理解，既然这次难度标识为Easy，想必不是一道太难的题。二话不说，先上无脑暴力搜索，遍历子集合中的每一个数字，然后在原数组中找到这个数字，然后向右遍历，找到第一个大于该数字的数即可，参见代码如下：

class Solution {

public:

vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {

vector<int> res(findNums.size());

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

int j = 0, k = 0;

for (; j < nums.size(); ++j) {

if (nums[j] == findNums[i]) break;

}

for (k = j + 1; k < nums.size(); ++k) {

if (nums[k] > nums[j]) {

res[i] = nums[k];

break;

}

}

if (k == nums.size()) res[i] = -1;

}

return res;

}

};

我们来对上面的方法稍做优化，我们用哈希表先来建立每个数字和其坐标位置之间的映射，那么我们在遍历子集合中的数字时，就能直接定位到该数字在原数组中的位置，然后再往右边遍历寻找较大数即可，参见代码如下：

class Solution {

public:

vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {

vector<int> res(findNums.size());

unordered\_map<int, int> m;

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

m[nums[i]] = i;

}

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

res[i] = -1;

int start = m[findNums[i]];

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

if (nums[j] > findNums[i]) {

res[i] = nums[j];

break;

}

}

}

return res;

}

};

29.下一个较大的元素

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

Input: [1,2,1]

Output: [2,-1,2]

Explanation: The first 1's next greater number is 2;   
The number 2 can't find next greater number;   
The second 1's next greater number needs to search circularly, which is also 2.

Note: The length of given array won't exceed 10000.

这道题是之前那道[Next Greater Element I](http://www.cnblogs.com/grandyang/p/6399855.html" \t "http://www.cnblogs.com/grandyang/p/_blank)的拓展，不同的是，此时数组是一个循环数组，就是说某一个元素的下一个较大值可以在其前面，那么对于循环数组的遍历，为了使下标不超过数组的长度，我们需要对n取余，下面先来看暴力破解的方法，遍历每一个数字，然后对于每一个遍历到的数字，遍历所有其他数字，注意不是遍历到数组末尾，而是通过循环数组遍历其前一个数字，遇到较大值则存入结果res中，并break，再进行下一个数字的遍历，参见代码如下：

class Solution {

public:

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

int n = nums.size();

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

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

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

if (nums[j % n] > nums[i]) {

res[i] = nums[j % n];

break;

}

}

}

return res;

}

};

30.You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S.

**Example 1:**

**Input:** nums is [1, 1, 1, 1, 1], S is 3. **Output:** 5**Explanation:**

-1+1+1+1+1 = 3

+1-1+1+1+1 = 3

+1+1-1+1+1 = 3

+1+1+1-1+1 = 3

+1+1+1+1-1 = 3

There are 5 ways to assign symbols to make the sum of nums be target 3.

**Note:**

* 1. The length of the given array is positive and will not exceed 20.
  2. The sum of elements in the given array will not exceed 1000.
  3. Your output answer is guaranteed to be fitted in a 32-bit integer.

这道题给了我们一个数组，和一个目标值，让我们给数组中每个数字加上正号或负号，然后求和要和目标值相等，求有多少中不同的情况。那么对于这种求多种情况的问题，我最想到的方法使用递归来做。我们从第一个数字，调用递归函数，在递归函数中，分别对目标值进行加上当前数字调用递归，和减去当前数字调用递归，这样会涵盖所有情况，并且当所有数字遍历完成后，我们看若目标值为0了，则结果res自增1，参见代码如下：

class Solution {

public:

int findTargetSumWays(vector<int>& nums, int S) {

int res = 0;

helper(nums, S, 0, res);

return res;

}

void helper(vector<int>& nums, int S, int start, int& res) {

if (start >= nums.size()) {

if (S == 0) ++res;

return;

}

helper(nums, S - nums[start], start + 1, res);

helper(nums, S + nums[start], start + 1, res);

}

};

31.最大连续1的个数

Given a binary array, find the maximum number of consecutive 1s in this array.

**Example 1:**

**Input:** [1,1,0,1,1,1]**Output:** 3**Explanation:** The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

这道题让我们求最大连续1的个数，不是一道难题。我们可以遍历一遍数组，用一个计数器cnt来统计1的个数，方法是如果当前数字为0，那么cnt重置为0，如果不是0，cnt自增1，然后每次更新结果res即可，参见代码如下：

根据面积来求出矩形的长和宽，要求长和宽的差距尽量的小，那么就是说越接近正方形越好。那就从1开始，看能不能整除，循环的终止条件是看平方值是否小于等于面积，参见代码如下：

class Solution {

public:

vector<int> constructRectangle(int area) {

int r = 1;

for (int i = 1; i \* i <= area; ++i) {

if (area % i == 0) r = i;

}

return {area / r, r};

}

};

class Solution {

public:

int findMaxConsecutiveOnes(vector<int>& nums) {

int res = 0, cnt = 0;

for (int num : nums) {

cnt = (num == 0) ? 0 : cnt + 1;

res = max(res, cnt);

}

return res;

}

};

32.Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

Input: [1,0,1,1,0]

Output: 4

Explanation: Flip the first zero will get the the maximum number of consecutive 1s.

After flipping, the maximum number of consecutive 1s is 4.

Note:

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

Follow up:  
What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

这道题在之前那道题[Max Consecutive Ones](http://www.cnblogs.com/grandyang/p/6360942.html)的基础上加了一个条件，说我们有一次将0翻转成1的机会，问此时最大连续1的个数，再看看follow up中的说明，很明显是让我们只遍历一次数组，那我们想，肯定需要用一个变量cnt来记录连续1的个数吧，那么当遇到了0的时候怎么处理呢，因为我们有一次0变1的机会，所以我们遇到0了还是要累加cnt，然后我们此时需要用另外一个变量cur来保存当前cnt的值，然后cnt重置为0，以便于让cnt一直用来统计纯连续1的个数，然后我们每次都用用cnt+cur来更新结果res，参见代码如下：

解法一：

class Solution {

public:

int findMaxConsecutiveOnes(vector<int>& nums) {

int res = 0, cur = 0, cnt = 0;

for (int num : nums) {

++cnt;

if (num == 0) {

cur = cnt;

cnt = 0;

}

res = max(res, cnt + cur);

}

return res;

}

};

上面的方法有局限性，如果题目中说能翻转k次怎么办呢，我们最好用一个通解来处理这类问题。我们可以维护一个窗口[left,right]来容纳至少k个0。我们遇到了0，就累加zero的个数，然后判断如果此时0的个数大于k，那么我们我们右移左边界left，如果移除掉的nums[left]为0，那么我们zero自减1。如果不大于k，那么我们用窗口中数字的个数来更新res，参见代码如下：

class Solution {

public:

int findMaxConsecutiveOnes(vector<int>& nums) {

int res = 0, zero = 0, left = 0, k = 1;

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

if (nums[right] == 0) ++zero;

while (zero > k) {

if (nums[left++] == 0) --zero;

}

res = max(res, right - left + 1);

}

return res;

}

};

上面那种方法对于follow up中的情况无法使用，因为nums[left]需要访问之前的数字。我们可以将遇到的0的位置全都保存下来，这样我们需要移动left的时候就知道移到哪里了：

 class Solution {

public:

int findMaxConsecutiveOnes(vector<int>& nums) {

int res = 0, left = 0, k = 1;

queue<int> q;

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

if (nums[right] == 0) q.push(right);

if (q.size() > k) {

left = q.front() + 1; q.pop();

}

res = max(res, right - left + 1);

}

return res;

}

};