**一．Slicing Winsow to find a optimal condition**

567. Permutation in String

Given two strings s1 and s2, write a function to return true if s2 contains the permutation of s1. In other words, one of the first string's permutations is the substring of the second string.

Example 1: Input:s1 = "ab" s2 = "eidbaooo"，Output: True

Explanation: s2 contains one permutation of s1 ("ba").

**S1. slicing window**

window size是s1的长度，用int[] count来保存 **s2在当前位置时每个char出现的frequency,**

每移动一个位置就update并判断一次。

*// use a slicing window samelength s1*

*// count the freq of chars appear in s1.*

public boolean checkInclusion(String s1, String s2) {

if (s1 == null || s2 == null || s1.length() > s2.length())

return false;

int[] count = new int[26];

int len1 = s1.length(), len2 = s2.length();

*// 1. init:*

for (int i = 0;i < len1;i++) {

count[s1.charAt(i) - 'a']++;

count[s2.charAt(i) - 'a']--;

}

if (allZero(count)) return true;

*// 2. keep moving and keep the window*

for (int i = len1;i < len2;i++) {

count[s2.charAt(i) - 'a']--;

count[s2.charAt(i - len1) - 'a']++;

if (allZero(count)) return true;

}

return false;

}

private boolean allZero(int[] nums) {

for (int num:nums) {

if (num != 0) return false;

}

return true;

}

3. Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "pw**wke**w", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

**S1 Two pointers**

用boolean[] 来判断某个char在window中是否出现过，window的范围是[i, j]

每次遇到不满足的情况就break, 更新一次maxLen.

public int lengthOfLongestSubstring(String s) {

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

boolean[] hash = new boolean[256];

int maxLen = 1;

int j = 0;

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

while (j < s.length()) {

if (!hash[s.charAt(j)]) {

hash[s.charAt(j)] = true;

j++;

} else break;

}

maxLen = Math.max(maxLen, j - i);

hash[s.charAt(i)] = false; // 最前面的char不包括在内

}

return maxLen;

}

159. Longest Substring with At Most Two Distinct Characters

Given a string, find the length of the longest substring T that contains at most 2 distinct characters.

For example, Given s = "eceba",

T is "ece" which its length is 3.

**S1: slicing window**

维护一个slicing window，两个字母c1, c2以及它们最后一次出现的坐标p1, p2,

当出现一个新字母时，把slicing window的头移到Math.min(p1, p2) + 1的位置。

public int lengthOfLongestSubstringTwoDistinct(String s) {

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

char c1 = '.', c2 = '.';

int p1 = -1, p2 = -1;

int start = 0, maxLen = 0;

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

char c = s.charAt(i);

if (c == c1) p1 = i;

else if (c == c2) p2 = i;

else {

if (p1 < p2) { // 判断p1,p2哪个靠前，替换靠前的char

start = p1 + 1;

c1 = c;

p1 = i;

} else {

start = p2 + 1;

c2 = c;

p2 = i;

}

}

maxLen = Math.max(maxLen, i - start + 1);

}

return maxLen;

}

28. Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack

**S1 利用java的函数**

public int strStr(String haystack, String needle) {

return haystack.indexOf(needle);

}

**S2 for 循环**

public int strStr(String haystack, String needle) {

if (haystack == null || needle == null) return -1;

int m = haystack.length(), n = needle.length();

if (m < n) return -1;

if (n == 0) return 0;

int threthold = m - n;

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

if (haystack.substring(i, i + n).equals(needle)) return i;

}

return -1;

}

524. Longest Word in Dictionary through Deleting

Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string. If there are more than one possible results, return the longest word with the smallest lexicographical order. If there is no possible result, return “”

**S1.Two pointers, moving pointers to 2 string**

查看s1能不能由s2 delete实现( s1长度<=s2)：用两个指针，分别指向s1, s2的对应位置，遍历s2的char，check是否和s1对应位置的char相同，相同则指针都++，不同仅s2++。

lexicographical order: s1.compareTo(s2)

time O(k) \* O(n), k is # of words in list, n is # of chars in s.

public String findLongestWord(String s, List<String> d) {

String longest = "";

for (String dict:d) {

int dictLen = dict.length();

int idx = 0; *// number of chars that s and dict match*

for (char c:s.toCharArray()) {

if (idx < dictLen && dict.charAt(idx) == c) idx++;

}

*// when find s after deleting:*

if (idx == dictLen) {

if (dictLen > longest.length()

|| (dictLen == longest.length() && dict.compareTo(longest) < 0))

longest = dict;

}

}

return longest;

}

209. Minimum Size Subarray Sum

Given an array of n positive integers and a positive integer s, find the **minimal length of a contiguous subarray of which the sum ≥ s**. If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and s = 7,

the subarray [4,3] has the minimal length under the problem constraint.

**S1. Two pointers.**

先移动right, 直到sum >= s, 然后再移动left, 在满足sum >= s的条件时更新min

public int minSubArrayLen(int s, int[] nums) {

int left = 0, right = 0;

int sum = 0, n = nums.length;

int min = Integer.MAX\_VALUE;

while (right < n || sum >= s) {

if (sum < s) {

sum += nums[right++]; *// right not included.*

} else {

min = Math.min(min, right - left);

sum -= nums[left++];

}

}

return min == Integer.MAX\_VALUE ? 0 : min;

}

264. Ugly Number II

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number, and n does not exceed 1690.

**题解：**

保存对2,3,5这三个倍数的pointer即可，每次加入其中的最小值。

需要注意的是，nums里面不能有重复。2\*3,3\*2这样的情况只能加入一个6.

所以需要用if 进行处理，而不能用else, 因为可能会有多个pointer在某一步同时后移。

public int nthUglyNumber(int n) {

int[] nums = new int[n];

nums[0] = 1;

int p2 = 0, p3 = 0,p5 = 0;

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

int num2 = nums[p2]\*2, num3 = nums[p3]\*3, num5 = nums[p5]\*5;

nums[i] = Math.min(num2, Math.min(num3, num5));

if (nums[i] == num2) p2++;

if (nums[i] == num3) p3++;

if (nums[i] == num5) p5++;

}

return nums[n-1];

}

IMC - find max length of valid password

A valid password must at least contains one uppercase character and should not contain any numbers.

Given a string, return the maximum length of a valid password in it.

大概有DP的做法？

**S1. Two pointers**

这是一个naive solution， 对于每个是大写字符的位置，向两边延伸，直到遇到number。

并更新global max. worst time O(n^2)

public static int maxPassword(String s) {

if (s == null || s.length() == 0) return -1;

int n = s.length();

int max = -1;

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

char cur = s.charAt(i);

if (!Character.isUpperCase(cur)) continue;

max = Math.max(max, getLength(s, i));

}

return max;

}

private static int getLength(String s, int i) {

int left = i - 1, right = i + 1, n = s.length();

while (left >= 0 && !Character.isDigit(s.charAt(left))) {

left--;

}

while (right < n && !Character.isDigit(s.charAt(right))) {

right++;

}

return right - left - 1;

}

438. Find All Anagrams in a String

Given a string s and a non-empty string p, find all the start indices of p's anagrams in s.

Strings consists of lowercase English letters only and the length of both strings s and p will not be larger than 20,100.

The order of output does not matter.

Input: s: "**cba**eba**bac**d" p: "abc" Output: [0, 6]

Explanation:

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**S1 Two pointers + hash array**

用两个指针，维护一个fixed length window, 在string s上向右移动，并检查满足条件的count个数。

只遍历一次，time O(n)

public List<Integer> findAnagrams(String s, String p) {

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

if (s == null || p == null || s.length() == 0

|| p.length() == 0 || s.length() < p.length())

return res;

int hash[] = new int[256];

for (char c:p.toCharArray()) {

hash[c]++;

}

int left = 0, right = 0, count = 0;

int slen = s.length(), plen = p.length();

while (right < slen) {

if (hash[s.charAt(right)] >= 1) {

count++; // count: 已经match上的字符数量。

}

hash[s.charAt(right)]--;

right++;

if (count == plen) res.add(left); // 代表全部match上，加入返回的res

if (right - left == plen) { // need to move left pointer

if (hash[s.charAt(left)] >= 0) count--; // left指向的字符本来能match上，要移开了

hash[s.charAt(left)]++;

left++;

}

}

return res;

}

Sum类

1. Two Sum

Given an array of integers, **return indices of the two numbers** such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example: Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

Follow up: 返回所有解

public int[] twoSum(int[] nums, int target) {

// <number, index of number>

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

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

int match = target - nums[i];

if (map.containsKey(match)) {

return new int[]{map.get(match), i};

}

map.put(nums[i], i);

}

return null;

}

// 如果需要返回所有解：

Map: <number, List<index of this number>>

并且发现match后加入result，不return，继续check下一个。

15. 3Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0?

**Find all unique triplets** in the array which gives the sum of zero.

Note:

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.

**S1: 2 pointer**

先sort（是为了后续跳过重复数字），固定最左侧的数字i，在[i+1, len-1]的范围内用 2 sum的做法

跳过重复的数字，此处的target为0，事实适用于所有target

time O(n^2)

public List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> out = new ArrayList<>();

Arrays.sort(nums);

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

int front = i + 1;

int back = nums.length - 1;

while (front < back) {

int sum = nums[front] + nums[i] + nums[back];

if (sum < 0) front++;

else if (sum > 0) back--;

else {

out.add(new ArrayList<>(Arrays.asList(nums[front], nums[i], nums[back])));

// 需要注意这里，因为是++front的格式，需要保证front范围valid

while (front < nums.length - 1

&& nums[++front] == nums[front - 1]);

// 同理需要保证back的范围valid

while (back > 0

&& nums[--back] == nums[back + 1]);

}

}

// 跳过重复的数字

while (i < nums.length - 1 && nums[++i] == nums[i - 1]);

}

return out;

}

18. 4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

Note: The solution set must not contain duplicate quadruplets.

**S1.Pointers**

在3 Sum的基础上来解决，time O(n^3),

其实有一个方法可以解决所有k Sum的问题，依次把kSum化成k – 1阶，再降成k-2阶，直到降到3，2来解决。降阶可以通过backtracking实现

public List<List<Integer>> fourSum(int[] nums, int target) {

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

Arrays.sort(nums);

kSum\_Trim(nums, target, 4, 0, res, new ArrayList<>());

return res;

}

void kSum\_Trim(int[] a, int target, int k, int start, List<List<Integer>> result, List<Integer> path) {

int max = a[a.length - 1];

if (a[start] \* k > target || max \* k < target) return;

if (k == 2) { *// 2 Sum*

int left = start;

int right = a.length - 1;

while (left < right) {

if (a[left] + a[right] < target) left++;

else if (a[left] + a[right] > target) right--;

else {

result.add(new ArrayList<>(path));

result.get(result.size() - 1).addAll(Arrays.asList(a[left], a[right]));

left++; right--;

while (left < right && a[left] == a[left - 1]) left++;

while (left < right && a[right] == a[right + 1]) right--;

}

}

} else { *// k Sum*

for (int i = start; i < a.length - k + 1; i++) {

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

if (a[i] + max \* (k - 1) < target) continue;

if (a[i] \* k > target) break;

if (a[i] \* k == target) {

if (a[i + k - 1] == a[i]) {

result.add(new ArrayList<>(path));

List<Integer> temp = new ArrayList<>();

for (int x = 0; x < k; x++) temp.add(a[i]);

result.get(result.size() - 1).addAll(temp); *// Add result immediately.*

}

break;

}

path.add(a[i]);

kSum\_Trim(a, target - a[i], k - 1, i + 1, result, path);

path.remove(path.size() - 1); *// Backtracking*

}

}

}

**Merging, Use PriorityQueue**

88. Merge Sorted Array

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.

题解：

从后往前放就可以。

public void merge(int[] nums1, int m, int[] nums2, int n) {

int k = m + n - 1;

int i = m - 1,j = n - 1;

for (;i >= 0 && j >= 0;k--) {

if (nums1[i] >= nums2[j]) {

nums1[k] = nums1[i--];

} else {

nums1[k] = nums2[j--];

}

}

while (j >= 0) {

nums1[k--] = nums2[j--];

}

}

Move Elements

283. Move Zeroes

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].

**S1. 直接置位**

public void moveZeroes(int[] nums) {

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

int idx = 0, n = nums.length;

for (int num: nums) {

if (num != 0) nums[idx++] = num;

}

while (idx < n) {

nums[idx++] = 0;

}

}

**S2. Swap**

两个指针往后走，fast负责找非0的值，slow始终指向最前面为0的值，fast每找到一个，两者就交换

27. Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example: Given input array nums = [3,2,2,3], val = 3

Your function should return length = 2, with the first two elements of nums being 2.

**S1. 也是直接置位即可，和上一道题的方法类似**

不要看着觉得简单 – 写的时候想不到，总是想要swap

public int removeElement(int[] nums, int val) {

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

int idx = 0, n = nums.length;

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

if (nums[i] != val) nums[idx++] = nums[i];

}

return idx;

}

26. Reverse Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

You must do this in place with constant memory.

For example, given input array nums = [1,1,2], your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

**S1. read-write pointers**

有一点要注意：read pointer总是比write pointer走得快，所以不会覆盖，在for loop里面可以每次都写，不需要做判断。

*// 0 1 1 1 2 3 5*

public int removeDuplicates(int[] nums) {

if (nums == null || nums.length < 2) return nums.length;

int count = 1;

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

if (nums[i] != nums[i - 1]) {

nums[count++] = nums[i];

}

}

return count;

}

**3 way partition – k way partition问题**

75. Sort Colors

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.

**S1.counting**

数有多少个0, 1, 2，然后在数组里填写相应个数的0,1,2

**S2.3-way partition**

三种颜色：0， 1， 2，通过left, right来分界，left指向的位置的左侧全部为0，right的右侧全部为2, 不含left, right.

需要注意的是，和右侧的元素swap后，i的位置上是一个没有处理过的元素，所以此时不能i++, 仍然需要对i进行处理。

*// here set pivot as 1*

public void sortColors(int[] nums) {

int left = 0;

int right = nums.length - 1;

int i = 0;

while (i <= right) {

if (nums[i] < 1) {

swap(nums, left++, i++);

} else if (nums[i] > 1) {

swap(nums, right--, i); *// attention! 这里i不能++*

} else {

i++;

}

}

}

void swap(int[] nums, int i, int j) {

int tmp = nums[i];

nums[i] = nums[j];

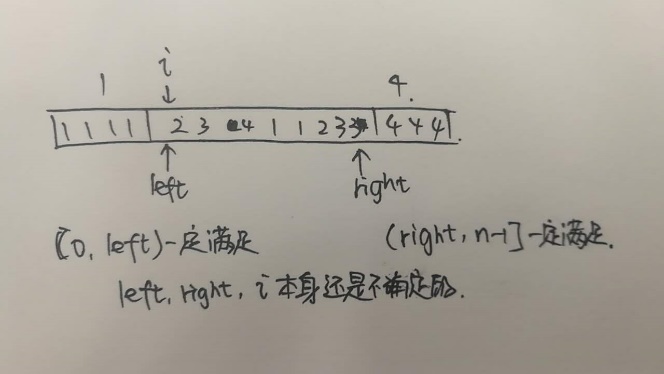
nums[j] = tmp;

}

Lintcode - Sort Colors II

Given an array of n objects with k different colors (numbered from 1 to k), sort them so that objects of the same color are adjacent, with the colors in the order 1, 2, ... k.

是上面问题的进阶版，k-way partition



**S1. Two Pointers – 荷兰旗**

lowColor, highColor原来是确定的1,3，现在也需要进行循环，每次while循环可以确定最小和最大的两个颜色，比如第一次确定1和10，第二次确定2和9...

内循环一次的时间复杂度是O(n), 一次内循环可以确定两个元素，所以需要k/2个内循环：time: O(n \* k/2)

public void sortColors2(int[] colors, int k) {

int low = 1, high = k;

int left = 0, right = colors.length - 1;

while (low < high) { // 每个循环确定low, high两种颜色，

int cur = left; // 并且low/high指定的颜色分别在left左边，right右边

while (cur <= right) {

if (colors[cur] == low) {

swap(colors, left++, cur++);

} else if (colors[cur] == high) {

swap(colors, right--, cur);

} else {

cur++;

}

}

low++;

high--;

}

}

private void swap(int[] nums, int i, int j) {

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

360. Sort Transformed Array

Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form f(x) = ax2 + bx + c to each element x in the array.

The returned array must be in sorted order.

Expected time complexity: O(n)

Example:

nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5, Result: [3, 9, 15, 33]

nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5 Result: [-23, -5, 1, 7]

**S1.merge sort**

二次函数有一个bitonic结构，转折点turning point在 –b/2a, 我们把这一点取作pivot点，

a = 0: 在b >= 0 时无需重新排序，b < 0需要reverse

a != 0: 二次函数有两种形态，先上升到pivot再下降，或者先下降到pivot再上升。

相当于merge两个list，取front, end, 根据两者对应的数值哪个距离pivot近，确定顺序。

public int[] sortTransformedArray(int[] nums, int a, int b, int c) {

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

return a == 0?sortLinear(nums, b, c, b > 0):sortBitonic(nums, a, b, c);

}

*// if b < 0, reverse, b > 0 same order*

int[] sortLinear(int[] nums, int b, int c, boolean increase) {

int n = nums.length;

int[] res = new int[n];

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

res[increase ? i : n-i-1] = b\*nums[i] + c;

}

return res;

}

int[] sortBitonic(int[] nums, int a, int b, int c) {

double pivot = -b/2.0/a;

int n = nums.length, front = 0, back = n - 1;

boolean increase = a < 0;

int[] res = new int[n];

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

int index = Math.abs(nums[front] - pivot) > Math.abs(nums[back] - pivot) ? front++ : back--;

res[increase ? i : n-i-1] = cal(nums[index], a, b, c);

}

return res;

}

int cal(int x, int a, int b, int c) {

return a\*x\*x + b\*x + c;

}

485. Max Consecutive Ones

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

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

Output: 3

The first two digits or the last three digits are consecutive 1s.The maximum number of consecutive 1s is 3.

**S1. DP**

time O(n), space O(n)

public int findMaxConsecutiveOnes(int[] nums) {

int n = nums.length;

int[] dp = new int[n];

dp[0] = nums[0];

int max = dp[0];

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

dp[i] = nums[i] == 0 ? 0 : dp[i-1]+1;

max = Math.max(max, dp[i]);

}

return max;

}

优化空间到O(1)

public int findMaxConsecutiveOnes(int[] nums) {

int n = nums.length;

int cur = nums[0];

int max = cur;

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

cur = nums[i] == 0 ? 0 : cur + 1;

max = Math.max(max, cur);

}

return max;

}

125. Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is not a palindrome.

**S1. Two pointers**

没什么，就是两个指针，一个从前往后，一个从后往前跑，只需要遍历一遍

public boolean isPalindrome(String s) {

if (s == null) return false;

if (s.length() < 2) return true;

s = s.toLowerCase();

int start = 0, end = s.length() - 1;

while (start < end) {

// 分别确定下一个为字母/数字的start/end char

while (start < end && !Character.isLetterOrDigit(s.charAt(start)))

start++;

while (start < end && !Character.isLetterOrDigit(s.charAt(end)))

end--;

if (s.charAt(start++) != s.charAt(end--)) return false;

}

return true;

}

349. Intersection of Two Arrays

Given two arrays, write a function to compute their intersection.

Example: Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].

Each element in the result must be unique. The result can be in any order.

**S1. Set**

如果可以用额外空间，那么直接用一个set存nums1，再用另一个set存intersection即可。

**S2. Two Pointers**

先sort，然后往后走，把相同的加进去。time O(nlogn)

**S3. Binary Search**

把nums1中出现过的值挨个在nums2中sort一遍

350. Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.

Example: Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].

Note:

Each element in the result should appear as many times as it shows in both arrays.

The result can be in any order.

**Follow up:**

What if the given array is already sorted? How would you optimize your algorithm?

What if nums1's size is small compared to nums2's size? Which algorithm is better?

What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

**S1. Set**

和上一道题相似，只用一个set存nums1的数值即可；

**S2. Two Pointers**

也可以用

与面积极值相关

11. Container with Most Water

Given *n* non-negative integers *a1* , *a2* , ..., *an* , where each represents a point at coordinate ( *i* , *ai* ). *n* vertical lines are drawn such that the two endpoints of line *i* is at ( *i* , *ai* ) and ( *i* , 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

**S1: 2 pointer**

两个指针分别指向nums的首尾，确定初始area，然后根据nums[front], nums[back]的大小关系决定移动哪一个。Time O(n).

public int maxArea(int[] height) {

if (height == null || height.length < 0) return 0;

int front = 0, back = height.length - 1;

int max = (back - front) \* Math.min(height[back], height[front]);

while (front < back) {

if (height[front] < height[back]) front++;

else back--;

max = Math.max(max, (back - front)

\* Math.min(height[back], height[front]));

}

return max;

}

42. Trapping rain water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

**S1. DP，从左到右，从右到左扫两遍**

先从左到右扫，确定每个位置的maxLeft 左边最大值，再从右往左扫，针对每个i确定右边最大值，如果maxRight > height[i] && left[i] > height[i] 则加进amount。time O(n)

public int trap(int[] height) {

if (height == null || height.length < 3) return 0;

int[] left = new int[height.length];

left[0] = height[0];

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

left[i] = Math.max(left[i - 1], height[i]);

}

int amount = 0, maxRight = height[height.length - 1];

for (int i = height.length - 2;i > 0;i--) {

if (maxRight > height[i] && left[i] > height[i]) {

amount += Math.min(left[i], maxRight) - height[i];

} else maxRight = height[i];

}

return amount;

}

**S2. 2 pointers 优化，只走一遍**

和上一个原理相同，只是同时从两边往中心处理，保留lmax, rmax, 可以确定水位为min(lmax, rmax),

如果lmax < rmax, 则从左边scan，直到找到> lmax的墙，此时水位上升到新的lmax，从两边向中间。

public int trap(int[] height) {

if (height == null || height.length < 3) return 0;

int amount = 0, l = 0, r = height.length - 1;

int lmax = height[l], rmax = height[r];

while (l < r) {

if (height[l] < height[r]) { // 左边低，左边向右移

lmax = Math.max(lmax, height[l]); // 如果是height[l] >= lmax, 那么下面amount += 0

amount += lmax - height[l++];

} else {

rmax = Math.max(rmax, height[r]);

amount += rmax - height[r--];

}

}

return amount;

}

287. Find the Duplicate Number – 在其他高频中也有记录

Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive), prove that at

least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

You must not modify the array (assume the array is read only).

You must use only constant, O(1) extra space. Your runtime complexity should be less than O(n 2 ).

There is only one duplicate number in the array, but it could be repeated more than once.

**S1. binary search**

这个纠结了很久，time O(nlogn)

nums中本来有n + 1个数字，需要考虑1 - n的数字是否重复，则min = 1, max = n, mid取中值，并计算在

[1, mid], (mid, n] 两个范围内的分别数量，因为前半区间的取值包括mid，所以min取mid + 1, 后半区间之前不包括mid，所以后续还需要考虑，所以取max = mid;

public int findDuplicate(int[] nums) {

if (nums == null || nums.length < 2) return -1;

int min = 1, max = nums.length - 1;

while(min < max){

int mid = min + (max - min) / 2;

int cnt = 0;

*// 计算总数组中有多少个数小于等于中间数*

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

if(nums[i] <= mid) cnt++;

}

*// 如果小于等于中间数的数量小于中间数，说明后半部分必有重复*

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

*// 否则前半部分必有重复*

else max = mid;

}

return min;

}

**S2. Find Cycle 神来之笔**

假设数组中没有重复，则可以把数组的下标0-n-1, 和1-n每个数找到一一映射，比如2-1-3，可以直接映射为0-2-3, 1-1，也可以把每个数字减一，成为0-1-2的映射，

但如果有重复，就会产生一对多的映射，比如2131, 映射为0-2-3, (1, 3)-1, 下标序列0-2-3-1-1-1-1-1.....会形成环。

再比如1321, 0-1-3-1, 2-2, (0, 3)-1, 2-2, 会形成0-1-3-1-3-1-3-1...的环。

相当于是找环的起点，就和Linked List Cycle II 做法相同。

public int findDuplicate(int[] nums) {

if (nums == null || nums.length < 2) return -1;

int slow = nums[0], fast = nums[nums[0]];

while (slow != fast) {

slow = nums[slow];

fast = nums[nums[fast]];

}

slow = 0;

while (slow != fast) {

slow = nums[slow];

fast = nums[fast];

}

return slow;

}

523. Continuous Subarray Sum

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of **k**, that is, sums up to n\*k where n is also an **integer**.

Input: [23, 2, 4, 6, 7], k=6 Output: True

Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

**S1 数组保存sum of [0, i] - time O(n^2), space O(n)**

第一遍遍历求sum[i]，表示[0, i]的和，

第二遍遍历对于每个i找sum[i + 2] ... n-1, 看是否有符合，k = 0, != 0分开考虑。

需要handle input = [0, 0]的corner case, 做法是sum多建一位，sum[0] = 0保存，在第二遍遍历时i = 0就可以判断是否成立。

public boolean checkSubarraySum(int[] nums, int k) {

if (nums == null || nums.length == 0) return false;

int[] sum = new int[nums.length + 1]; *// corner case [0,0] handled by nums.length + 1.*

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

sum[i] = sum[i-1] + nums[i-1];

}

for (int i = 0; i < nums.length; i++) { *// i start from 0 here, handle [0, 0]*

for (int j = i + 2; j <= nums.length; j++) {

if (k == 0) {

if (sum[j] - sum[i] == 0) return true;

} else if ((sum[j] - sum[i]) % k == 0) {

return true;

}

}

}

return false;

}

**S2 Map time O(n), space O(k)难做到的是space**

遍历数组，keep track of sumMod, 是每次加上nums[i]然后取 mod k的值，所以sumMod的范围是[0, k-1]，

map中的数值含义是<sumMod, corresponding index in array>, 遍历时如果发现sumMod在map中已经存在，比如对应value为m，说明(m, i]的sum满足要求，是k的倍数，要求至少两个数字，所以需要i – prev > 1。

还需要考虑一个corner case，input [0, 0], k = 0，此时是满足条件的，但是现有程序判定符合条件是左开右闭，所以需要把index = -1的条件加上，也就是在一开始加上 map.put(0, -1);

public boolean checkSubarraySum(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

map.put(0, -1);

int sumMod = 0;

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

sumMod += nums[i];

if (k != 0) sumMod %= k;

Integer prev = map.get(sumMod);

if (prev != null) {

if (i - prev > 1) return true;

} else map.put(sumMod, i);

}

return false;

}