**HashMap 、HashTable**

325. Maximum Size Subarray Sum Equals k

Given an array nums and a target value k, find the maximum length of a subarray that sums to k. If there isn't one, return 0 instead.

Given nums = [1, -1, 5, -2, 3], k = 3,

return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)

Given nums = [-2, -1, 2, 1], k = 1,

return 2. (because the subarray [-1, 2] sums to 1 and is the longest)

**题解：**

因为需要找的是长度最长，且满足sum = k的值，所以需要确定到nums[i]的sum的值，以及相应的index，

用map来做最合适

// map: <sum of all elements before index i, i>

// 向后走时，

public int maxSubArrayLen(int[] nums, int k) {

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

int sum = 0, max = 0;

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

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

sum += nums[i]; *// sum of all elements for [0, i]*

if (sum == k) max = i + 1;

if (map.containsKey(sum - k)) {

max = Math.max(max, i - map.get(sum - k));

}

if (!map.containsKey(sum)) { // 只在不contains的时候加入，确保保留的始终是最前的index

map.put(sum, i);

}

}

return max;

}

128. Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

**S1. HashMap,** time O(n)

用一个map存每个数字所涉及的LCS, <number, length of consecutive sequence it belongs to>

每次拿到一个新数字时，如果目前数字不在map中，可以尝试向左向右找相连的数字，

并且更新该数字和两侧相邻数字对应的length值。

public int longestConsecutive(int[] nums) {

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

int max = 1;

Map<Integer, Integer> map = new HashMap<>(); *// <number, length>*

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

if (map.containsKey(nums[i])) continue;

int cur = nums[i];

int leftLen = map.getOrDefault(cur - 1, 0);

int rightLen = map.getOrDefault(cur + 1, 0);

int totalLen = leftLen + 1 + rightLen;

max = Math.max(max, totalLen);

map.put(cur, totalLen);

map.put(cur - leftLen, totalLen);

map.put(cur + rightLen, totalLen);

}

return max;

}

500. Keyboard Row

Given a List of words, return the words that can be typed using letters of alphabet on only one row's of American keyboard like the image below.





题解：

用java 8的新特性stream和regex来做，一行就可以

public String[] findWords(String[] words) {

return Stream.of(words).filter(s -> s.toLowerCase().matches("[qwertyuiop]\*|[asdfghjkl]\*|[zxcvbnm]\*")).toArray(String[]::new);

}

347. Top K Frequent Elements

Given a non-empty array of integers, return the k most frequent elements.

For example, Given [1,1,1,2,2,3] and k = 2, return [1,2].

Note: You may assume k is always valid, 1 ≤ k ≤ number of unique elements.

Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

**S1.HashMap + bucket Sort**

1. 首先，计算出每个数字的freq，存在map里
2. 用List[i] 表示freq = i 的数字
3. 从freqList倒序，取得k个。为什么需要在最后用subList，是因为有可能k=10，到其中某个循环k = 9, 然后下个freq有多个num满足条件，addAll之后会> 10

public List<Integer> topKFrequent(int[] nums, int k) {

Map<Integer, Integer> freqMap = new HashMap<>(); // <num， freq>

List<Integer>[] freqList = new ArrayList[nums.length + 1]; // <freq, list of nums>

for (int num:nums) {

freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

}

for (int key:freqMap.keySet()) {

int freq = freqMap.get(key);

if (freqList[freq] == null) {

freqList[freq] = new ArrayList<>();

}

freqList[freq].add(key);

}

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

for (int i = nums.length;i >= 0 && res.size() < k;i--) {

if (freqList[i] != null) res.addAll(freqList[i]);

}

return res.subList(0, k);

}

451. Sort Characters By Frequency

Given a string, sort it in decreasing order based on the frequency of characters.

Input: "Aabb" Output: "bbAa"

Explanation:

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

**S1. HashMap + bucket Sort,**

和上面的做法完全一样，只是不是top k, 而是return全部。纯粹用来练手。

public String frequencySort(String s) {

Map<Character, Integer> freqMap = new HashMap<>();

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

freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);

}

int n = s.length();

List<Character>[] freqList = new ArrayList[n + 1];

for (Map.Entry<Character, Integer> entry:freqMap.entrySet()) {

int freq = entry.getValue();

if (freqList[freq] == null) freqList[freq] = new ArrayList<>();

freqList[freq].add(entry.getKey());

}

StringBuilder sb = new StringBuilder();

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

if (freqList[i] != null) {

for (char c:freqList[i]) {

for (int j = 0;j < i;j++) sb.append(c);

}

}

}

return sb.toString();

}

205. Isomorphic Strings

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example, Given "egg", "add", return true. Given "foo", "bar", return false.

**S HashMap**

一个char只能和一个char对应，必须是一一对应关系。

public boolean isIsomorphic(String s, String t) {

if(s == null || t == null) return false;

if(s.length() != t.length()) return false;

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

int n = s.length();

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

char c1 = s.charAt(i);

char c2 = t.charAt(i);

if(map.containsKey(c1)){

*// not consistant with previous*

if(map.get(c1) != c2) return false;

} else {

*// c2 already been mapped*

if(map.containsValue(c2)) return false; // 要check所有的value

map.put(c1, c2);

}

}

return true;

}

290. Word Pattern

Given a pattern and a string str, find if str follows the same pattern

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.

Examples:

pattern = "abba", str = "dog cat cat dog" should return true.

pattern = "abba", str = "dog cat cat fish" should return false.

You may assume pattern contains only lowercase letters, and str contains lowercase letters separated by a single space.

**S HashMap**

和上面几乎一样，只是换成了<char, String> 的map

public boolean wordPattern(String pattern, String str) {

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

Map<Character, String> map = new HashMap<>();

String[] strs = str.split(" ");

char[] patts = pattern.toCharArray();

if (strs.length != patts.length) return false;

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

if (map.containsKey(patts[i])) {

if (!map.get(patts[i]).equals(strs[i])) return false;

} else if (map.containsValue(strs[i])) {

return false;

} else {

map.put(patts[i], strs[i]);

}

}

return true;

}

**Array**

31. Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

**S1**

这个题目考察的其实就是一个数组，怎么把它转换成刚好大一点的值。

从后往前找，找第一个下降的拐点，比如4 5 3 2 1, 就是找到4，假设index为i，然后4需要和某个数字swap

不一定是4后面的那个值，事实上是需要和最靠后的，比它大的值交换，

所以应该再从最后check一遍，找到第一个比它大的值，两者swap，

swap之后，从index = i+1到最后，其实数值是不断减小的，所以需要全部reverse，才能找到next permutation.

public void nextPermutation(int[] nums) {

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

int n = nums.length;

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

if (nums[i] < nums[i + 1]) {

*// find the number just > nums[i]*

int j = n-1;

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

swap(nums, i, j);

*// now numbers from i+1 to end are in decrease order*

*// to get to next permutation, need to reverse them.*

reverse(nums, i + 1, n - 1);

return;

}

}

reverse(nums, 0, n-1);

}

private void reverse(int[] nums, int start, int end) {

while (start < end) {

swap(nums, start++, end--);

}

}

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

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

41. First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

**S Array**

这里重点是这样一个分析：integer array的first missing positive number, 只能是[1, n+1]范围内的取值，n是array size。

了解这个就好办了，尝试把1-n+1的数字分别放到下标为0~n的位置上。第一个不对应的就是first missing，如果全部对应，就是n+1。

时间复杂度的计算：worst condition：所有的数字分别是1-n, 这样需要swap的次数最多。

但是每次swap, 会保证其中一个数字归位，以后不需要再换，所以所有的数字需要的时间是O(n), 遍历的时间也是O(n)，所以总时间是O(n)。

public int firstMissingPositive(int[] nums) {

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

int n = nums.length;

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

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

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

}

}

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

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

}

return n + 1;

}

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

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example, Given [1,3],[2,6],[8,10],[15,18], return [1,6],[8,10],[15,18].

**S 依次往后移动，并且分情况加interval即可**

public List<Interval> merge(List<Interval> intervals) {

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

if (intervals == null || intervals.size() == 0) return res;

Collections.sort(intervals, new IntervalComparator());

int prevStart = intervals.get(0).start;

int prevEnd = intervals.get(0).end;

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

Interval cur = intervals.get(i);

if (cur.start > prevEnd) { *// no overlap*

res.add(new Interval(prevStart, prevEnd));

prevStart = cur.start;

prevEnd = cur.end;

} else {

prevEnd = Math.max(prevEnd, cur.end);

}

}

*// last one still not added*

res.add(new Interval(prevStart, prevEnd));

return res;

}

private class IntervalComparator implements Comparator<Interval> {

public int compare(Interval i1, Interval i2) {

return i1.start - i2.start;

}

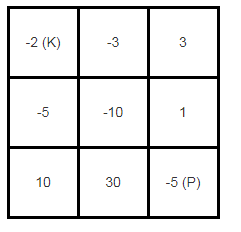
}

**Greedy**

174. Dungeon Game

题意是，出发点在左上角，终点在右下角，中间每个格的数字代表着经过这个格子生命值会发生的变化，比如会-5或者+3，如果走到其中某个点，生命值 <= 0，则立刻失败。

要求的是，要成功从左上角走到右下角，初始生命值最少是多少？



**题解：**

这道题其实就是要找从左上角到右下角失去生命值最少的路径，以及过程中失去的最少的生命值。

在这里反着写，从右下角往左上角来找：

*// hps[i][j]: min hp needed at (i, j)*

public static int calculateMinimumHP(int[][] dungeon) {

if (dungeon == null) return 0;

int m = dungeon.length, n = dungeon[0].length;

int[][] hp = new int[m][n]; *// hp needed to get here.*

hp[m - 1][n - 1] = Math.max(1, 1 - dungeon[m - 1][n - 1]);

*// last column*

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

hp[i][n - 1] = Math.max(hp[i + 1][n - 1] - dungeon[i][n - 1], 1);

}

*// first row*

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

hp[m - 1][j] = Math.max(hp[m - 1][j + 1] - dungeon[m - 1][j], 1);

}

*// other ordinary*

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

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

int need = Math.min(hp[i][j + 1], hp[i + 1][j]) - dungeon[i][j];

*// if need <= 0, then set to be 1*

hp[i][j] = Math.max(1, need);

}

}

return hp[0][0];

}

55. Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example: A = [2,3,1,1,4], return true. A = [3,2,1,0,4], return false.

**S1. General DP**

用dp[i]来表示能否到达i，用两个for loop

time: O(n^2), space O(n)

public boolean canJump(int[] nums) {

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

int n = nums.length;

boolean[] dp = new boolean[n];

dp[0] = true;

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

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

if (dp[j] && nums[j] + j >= i) {

dp[i] = true;

break;

}

}

}

return dp[n-1];

}

**S2. keep pace of max step**

这个方法更机智，time O(n), space O(1)

每一步通过判断max 与 i的关系，判断能否走到当前这步。如果可以就继续往下走。

任意一步不行就return false, 如果一直没有return 就相当于成功了。

public boolean canJump(int[] nums) {

int max = 0, n = nums.length;

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

if (i > max) return false;

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

//if (max >= n) return true; // 这里其实可以加判断早return true

}

return true;

}

45. Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example: Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

**S1. DP**

重点是，只有在迫不得已的时候jump才++，time O(n), space O(1)

public int jump(int[] nums) {

if (nums == null || nums.length < 2) return 0;

int step\_count = 0; // minimum steps for reaching end.

int last\_jump\_max = 0; // 上一步最远到达的距离

int current\_jump\_max = 0; // 从当前这里开始jump，最远能到达的index

for (int i = 0;i < nums.length - 1;i++) {

current\_jump\_max = Math.max(current\_jump\_max, nums[i] + i);

if (i == last\_jump\_max) { // 走到last\_jump最远到达的点，不得不jump

step\_count++;

last\_jump\_max = current\_jump\_max;

}

}

return step\_count;

}

330. Patching Array

Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

nums = [1, 3], n = 6, Return 1.

Combinations of nums are [1], [3], [1,3], which form possible sums of: 1, 3, 4.

Now if we add/patch 2 to nums, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3].

Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].

So we only need 1 patch.

nums = [1, 5, 10], n = 20, Return 2. The two patches can be [2, 4].

**题解：**

假设miss是[0, n]中最小的missng number, 意味着[1, miss-1]的所有数字都可以通过sum得到；

然后如果发现有一个number missing: num <= miss, 就把它加到smaller sums来build all sums in [0, miss+num]

If we don't, then we must add such a number to the array, and it's best to add miss itself, to maximize the reach.

corner case: when n = Integer.MAX\_VALUE, while循环的miss <= n会始终成立。

所以用long来表示miss

public int minPatches(int[] nums, int n) {

long miss = 1;

int count = 0;

int i = 0;

int len = nums.length;

while (miss <= n) {

if (i < len && nums[i] <= miss) {

miss += nums[i];

i++;

} else {

miss += miss;

count++;

}

}

return count;

}

406. Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.

Input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output: [[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

**题解：**

分为sort和insert两步，这里的重点是，高个子不会被矮个子挡到，也就是在插入矮个子的时候，如果有高个子需要在它前面，那就需要已经在里面插入指定位置才会正确，所以在insert的时候需要先insert个子高的人， 来保证后面insert个子矮的人时index是对的。

所以在sort的步骤，把高个子的人放在前面。

其实h相同时，k的顺序并不重要。但是要考虑到一开始res为空的时候，加入直接插入pos = 1，很明显会exception，所以k从小到大来insert.

public int[][] reconstructQueue(int[][] people) {

*// if h different: sort by h descending,*

*// if h same: sort by k ascending*

Arrays.sort(people, new Comparator<int[]>() {

@Override

public int compare(int[] o1, int[] o2) {

return o1[0] == o2[0] ? (o1[1] - o2[1]) : o2[0] - o1[0];

}

});

List<int[]> res = new LinkedList<>();

for (int[] cur : people) {

res.add(cur[1], cur); *// pos, content*

}

return res.toArray(new int[people.length][]);

}

321. Create Maximum Number

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.

Example 1:

nums1 = [3, 4, 6, 5]，nums2 = [9, 1, 2, 5, 8, 3]，k = 5

return [9, 8, 6, 5, 3]

Example 2:

nums1 = [6, 7]，nums2 = [6, 0, 4]，k = 5

return [6, 7, 6, 0, 4]

题解：

public int[] maxNumber(int[] nums1, int[] nums2, int k) {

int n = nums1.length;

int m = nums2.length;

int[] ans = new int[k];

for (int i = Math.max(0, k - m); i <= k && i <= n; ++i) {

int[] candidate = merge(maxArray(nums1, i), maxArray(nums2, k - i), k);

if (greater(candidate, 0, ans, 0)) ans = candidate;

}

return ans;

}

private int[] merge(int[] nums1, int[] nums2, int k) {

int[] ans = new int[k];

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

ans[r] = greater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];

}

return ans;

}

*// compare nums1[i, end] with nums2[j, end]*

*// if nums1 is greater, return true.*

public boolean greater(int[] nums1, int i, int[] nums2, int j) {

int m = nums1.length, n = nums2.length;

while (i < m && j < n && nums1[i] == nums2[j]) {

i++;

j++;

}

return j == n || (i < m && nums1[i] > nums2[j]);

}

*// get the max array of size k from nums.*

public int[] maxArray(int[] nums, int k) {

int n = nums.length;

int[] ans = new int[k];

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

while (n - i + j > k && j > 0 && ans[j - 1] < nums[i]) j--;

if (j < k) ans[j++] = nums[i];

}

return ans;

}

452. Minimum Number of Arrows to Burst Balloons

描述很复杂，这道题基本是这么回事：

输入是int 2d array, 其中2d array由多个[x1, x2]组成，对应一个气球。一旦击中x1, x2之间(inclusive)的坐标，就可以看作气球被击中。

问击中所有气球最少需要多少个arrow.

重点是思路，代码很好写。

按照x2来排序，然后鉴于需要戳中每个气球，所以我们选择的坐标，一定要包含第一个气球[x1, x2]的坐标，

此外，我们还希望这个坐标尽可能多的存在在其他气球的有效范围内。所以在[x1, x2]的范围中，选择x2。

然后往下走，直到走到下一个x2不在范围内的气球，再选x2.

public int findMinArrowShots(int[][] points) {

if (points == null || points.length == 0 || points[0].length != 2)

return 0;

int count = 1, n = points.length;

*// 1. sort by end:*

Arrays.sort(points, (a, b)->(a[1] - b[1]));

int shoot = points[0][1];

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

if (points[i][0] > shoot || points[i][1] < shoot) {

shoot = points[i][1];

count++;

}

}

return count;

}

253. Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

For example, Given [[0, 30],[5, 10],[15, 20]], return 2.

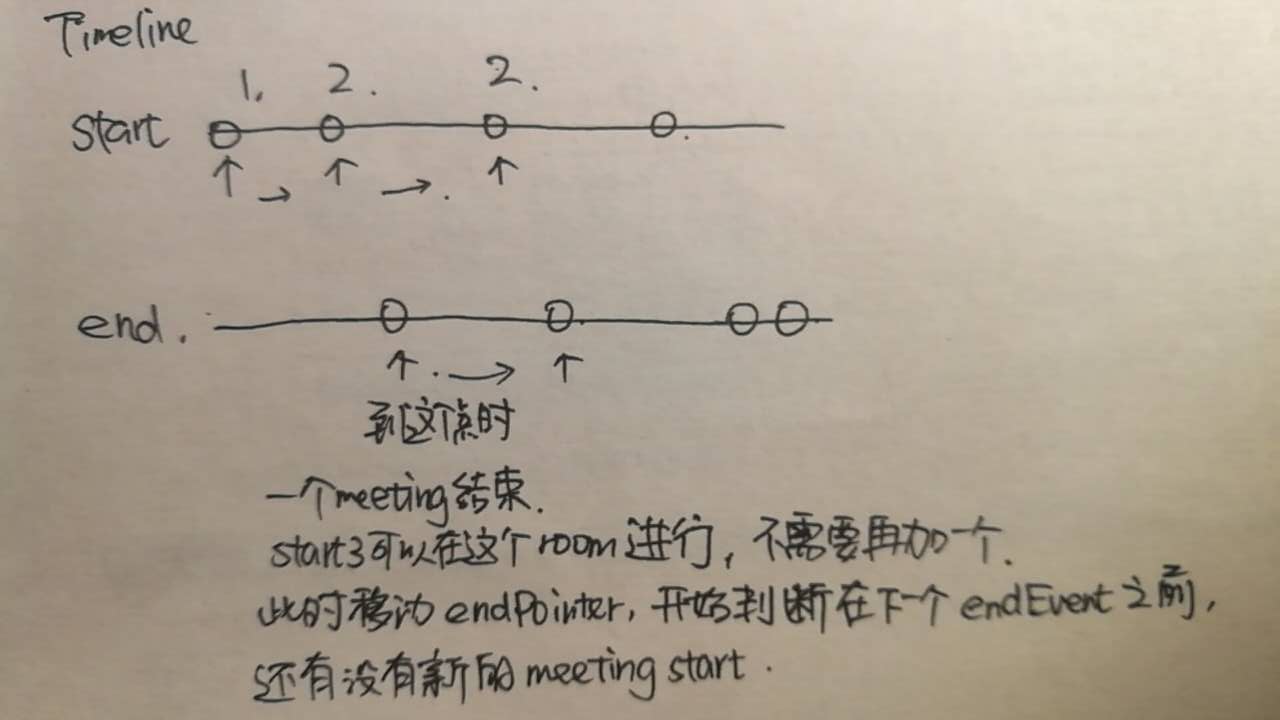
**S1 Greedy ?? 其实不知道怎么归类算法**

首先分别sort start time, end time 便于下一步进行遍历；

核心就是判断：

在某个meeting end之前，有多少个meeting开始

新开始的meeting能不能放到之前结束的room里，不占用新的room



public int minMeetingRooms(Interval[] intervals) {

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

int n = intervals.length;

int[] start = new int[n];

int[] end = new int[n];

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

start[i] = intervals[i].start;

end[i] = intervals[i].end;

}

Arrays.sort(start);

Arrays.sort(end);

int count = 0, endPointer = 0;

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

if (start[i] < end[endPointer]) count++;

else endPointer++;

}

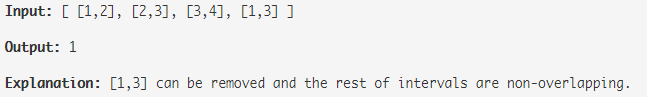
return count;

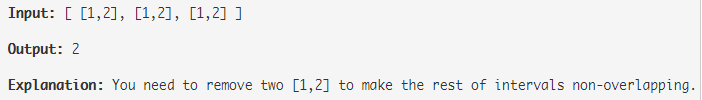
}

435. Non-overlapping Intervals

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

1. You may assume the interval's end point is always bigger than its start point.
2. Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.





题解：

有思路之后一遍过，机智, time O(nlogn), space O(1)

先按照start排序，然后遍历，如果某个区间的start > 上一个有效区间的end, 则出现重叠，需要删掉一个。

那么删掉哪一个呢？

* end较大的那一个。因为end较大的更容易和后面的区间发生overlap

public int eraseOverlapIntervals(Interval[] intervals) {

if (intervals == null || intervals.length < 2) return 0;

Arrays.sort(intervals, (a, b)->(a.start - b.start));

int end = intervals[0].end;

int n = intervals.length;

int count = 0;

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

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

end = Math.min(end, intervals[i].end);

count++;

} else {

end = intervals[i].end;

}

}

return count;

}

RPG game:

You’re playing your favorite RPG, and your character has just found a room full of treasure. You have n inventory slots. Luckily, objects of the same type stack together, with the maximum size of the stack depending on the type (e.g. coins might stack to 10, diamonds to 5, armor to 1, etc.). Each stack (or partial stack) takes up 1 inventory slot. Each item has a selling value (e.g. a single diamond might be worth 10, so a stack of 5 diamonds would be worth 50). You want to maximize the total selling value of the items in your inventory.

Write a function to find the set of things to bring home that maximizes the total value.

Input:

n: The number of inventory slots.

items: Array of item types, one for each item in the room

item\_infos: Array of structs, one for each unique item type

struct ItemInfo {.

String name;

int value;

int maximum\_stack\_size;

}

Example input

n: 3

items: [“diamond”, “ruby”, “armor”, “diamond”, “diamond”, “ruby”, “diamond”, “diamond”, “diamond”, “diamond”, “diamond” “armor”]. 1point3acres.com/bbs

item\_infos:

{

name=”diamond”

value=10

maximum\_stack\_size=5},

{

name=”ruby”.

value=5

maximum\_stack\_size=5.

},

{

name=”armor”

value=25

maximum\_stack\_size=1

}

]

Example output

105 (1 stack of 5 diamonds worth 50, 1 partial stack of 3 diamonds worth 30, 1 stack of 1 armor worth 25)

S1. 用greedy 以及priorityQueue做

时间复杂度，宝石的种类数是k, 那么需要

public static int maxValue(int n, String[] stuff, Item[] items) {

if (n <= 0 || stuff == null || stuff.length == 0

|| items == null || items.length == 0) return 0;

Map<String, Integer> count = new HashMap<>();

*// get the # of each*

for (String s:stuff) {

count.put(s, count.getOrDefault(s, 0) + 1);

}

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

*// map of <item name:item>, can easily get information.*

Map<String, Item> itemMap = new HashMap<>();

for (Item i:items) {

itemMap.put(i.name, i);

}

*// for each type, diamond, xx, yyy...:*

for (String key:count.keySet()) {

int amount = count.get(key);

int fullAmount = itemMap.get(key).maxSize;

int value = itemMap.get(key).value;

while (amount > 0) {

if (amount >= fullAmount) {

maxHeap.add(value \* fullAmount);

amount -= fullAmount;

} else {

maxHeap.add(value \* amount);

break;

}

}

}

int res = 0;

while (n-- > 0) {

res += maxHeap.poll();

}

return res;

}

**Recursion / Stack**

341. Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1: Given the list [[1,1],2,[1,1]],

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

**S. Recursion, beats 89%**

如果是number就加入list，否则就继续调用

public class NestedIterator implements Iterator<Integer> {

List<Integer> list;

int index;

public NestedIterator(List<NestedInteger> nestedList) {

if (nestedList == null) return;

index = 0;

list = new ArrayList<>();

helper(nestedList);

}

void helper(List<NestedInteger> nestedList) {

for (NestedInteger i:nestedList) {

if (i.isInteger()) {

list.add(i.getInteger());

} else {

helper(i.getList());

}

}

}

@Override

public Integer next() {

if (!hasNext()) return null;

return list.get(index++);

}

@Override

public boolean hasNext() {

return index < list.size();

}

}

**S2 Stack beats 20+%**

一开始从后往前全部放入stack，这样能保证取的时候是从前往后。

在hasNext时，判断顶端的是不是数字，如果是则返回，不是则取出，把list中的从后往前再放进去。用while loop，直到取到数字为止。

public class NestedIterator implements Iterator<Integer> {

Stack<NestedInteger> stack = new Stack<>();

public NestedIterator(List<NestedInteger> nestedList) {

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

stack.push(nestedList.get(i));

}

}

@Override

public Integer next() {

return stack.pop().getInteger();

}

@Override

public boolean hasNext() {

while(!stack.isEmpty()) {

NestedInteger curr = stack.peek();

if(curr.isInteger()) {

return true;

}

stack.pop();

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

stack.push(curr.getList().get(i));

}

}

return false;

}

}

251. Flatten 2D Vector

Implement an iterator to flatten a 2d vector.

For example, Given 2d vector

[ [1,2], [3], [4,5,6]]

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

**S Iterator + Recursion**

用Iterator做其实比用两个指针还要容易。

public class Vector2D implements Iterator<Integer> {

Iterator<List<Integer>> listIter;

Iterator<Integer> curIter;

Integer peek;

public Vector2D(List<List<Integer>> vec2d) {

this.listIter = vec2d.iterator();

peek = internalNext();

}

private Integer internalNext() {

if (curIter != null && curIter.hasNext()) { // current list还没走完

return curIter.next();

} else if (listIter.hasNext()) { // current list走完了，但还有别的list可以走

curIter = listIter.next().iterator();

return internalNext();

} else { // 全部走完

return null;

}

}

@Override

public Integer next() {

Integer res = peek;

peek = internalNext();

return res;

}

@Override

public boolean hasNext() {

return peek != null;

}

}

71. Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,

path = "/home/", => "/home"

path = "/a/./b/../../c/", => "/c"

corner case: /../../../..: /

may contain multiple ‘/’ like ‘/home//foo’ == ‘/home/foo’

**S Stack**

首先按照”/”split，之后需要考虑的几种情况:

/a: store in stack

/.: overlook

/..: pop if stack not empty

public String simplifyPath(String path) {

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

String[] names = path.split("/");

for (String n:names) {

if (n.equals("..")) {

if (!s.isEmpty()) s.pop();

} else if (n.equals(".")) {

continue;

} else if (n.length() > 0){

s.push("/" + n);

}

}

String out = "";

while (!s.isEmpty()) {

out = s.pop() + out;

}

if (out.length() == 0) return "/";

return out;

}

224. Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, non-negative integers and empty spaces .

You may assume that the given expression is always valid.

227. Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

**S1 Stack**

最开始就是用stack解的，这个解法的核心就在于，查看的是上一次的sign，

public int calculate(String s) {

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

char sign = '+';

int num = 0;

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

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

char c = s.charAt(i);

if (Character.isDigit(c)) {

num = num\*10 + s.charAt(i) - '0';

}

if (c == '+' || c == '-' || c == '\*' || c == '/' || i == s.length() - 1) {

switch(sign) { *// 查看的是上一次的*

case '+': stack.push(num);break;

case '-': stack.push(-num);break;

case '\*': stack.push(stack.pop() \* num);break;

case '/': stack.push(stack.pop() / num);break;

}

sign = c;

num = 0;

}

}

int out = 0;

for (int i:stack) {

out += i;

}

return out;

}

**S2. 空间优化 space O(1)**

后来才知道原来也有Space O(1)的解法，重点就是包含\* / 这样操作的，不更新result, 只有+- 更新result。

\*/ 涉及这两个操作的数值计算存在prev里，等到下一个+-操作时加进result

用md作为标志位，表明数字的前一个符号是哪一个，用sign做标志位，表明正负，

遍历每一位：

* 如果是数字：就用while继续往后找，直到碰到不是数字的 – 记得把index-1，然后看前一个非数字位是什么，来更新prev的值
* 如果是+ / -: 把之前的计算结果prev \* sign 存入result, 重新设置sign=1 or -1
* 如果是\* / /: 更新prev,

public int calculate(String s) {

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

int result = 0, prev = 0;

int sign = 1;

int md = 0; *// 1: \*, -1: /*

int n = s.length();

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

char c = s.charAt(i);

if (Character.isDigit(c)) {

int num = c - '0';

while (++i < n && Character.isDigit(s.charAt(i))) {

num = 10\*num + s.charAt(i) - '0';

}

i--; // 这一位不是数字，没有被处理，需要减回来

if (md != 0) {

prev = md == 1? prev\*num : prev/num;

md = 0;

} else {

prev = num;

}

} else if (c == '+' || c == '-') {

result += sign\*prev;

sign = c == '+' ? 1:-1;

} else if (c == '\*' || c == '/') {

md = c == '\*' ? 1 : -1;

}

}

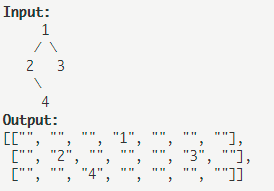
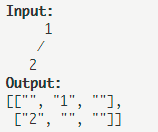
return result + sign \* prev;

}

655. Print Binary Tree

Print a binary tree in an **m\*n** 2D string array following these rules:

1. The row number m should be equal to the height of the given binary tree.
2. The column number n should always be an odd number.
3. The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. **Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree**. **However, if two subtrees are none, then you don't need to leave space for both of them.**
4. Each unused space should contain an empty string "".
5. Print the subtrees following the same rules.

**S1. Recursion**

一开始先建立好res, 把所有的都设成default，也就是””.

然后再从root开始，计算每一个TreeNode的位置坐标，并放置相应的val。

recursion的截止条件是row == rows, 意味着height已经超过树原有的深度。

public List<List<String>> printTree(TreeNode root) {

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

int height = root == null ? 1 : getHeight(root);

int rows = height, cols = (int) Math.pow(2, height) - 1;

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

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

row.add("");

}

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

res.add(new ArrayList<>(row));

}

helper(res, root, 0, rows, 0, cols - 1);

return res;

}

private void helper(List<List<String>> res, TreeNode root, int row, int rows, int i, int j) {

if (row == rows || root == null) return;

int mid = (i + j) / 2;

res.get(row).set(mid, Integer.toString(root.val));

helper(res, root.left, row + 1, rows, i, mid - 1);

helper(res, root.right, row + 1, rows, mid + 1, j);

}

public int getHeight(TreeNode root) {

if (root == null) return 0;

return 1 + Math.max(getHeight(root.left), getHeight(root.right));

}

394. Decode String

Given an encoded string, return it's decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

s = "3[a]2[bc]", return "aaabcbc".

s = "3[a2[c]]", return "accaccacc".

s = "2[abc]3[cd]ef", return "abcabccdcdcdef".

**S1. Stack**

用两个stack, 分别存repeat number和需要repeat的string，

1. number, push into number-stack, \*\*\* 数字可能不是一位，需要拿到所有的数字
2. '[', 把之前的string存到string stack中，开始用一个新的””开始存新的，需要repeat的string
3. ']', 从string-stack中pop出一个string出来，再从number-stack中pop出一个number(代表这个string要重复多少遍)
4. append to当前的string

public String decodeString(String s) {

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

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

Stack<String> strStack = new Stack<>();

String res = "";

int i = 0, n = s.length();

while (i < n) {

if (Character.isDigit(s.charAt(i))) {

int number = 0;

while (Character.isDigit(s.charAt(i))) {

number = number\*10 + s.charAt(i) - '0';

i++;

}

numStack.push(number);

} else if (s.charAt(i) == '[') {

strStack.push(res); *// string before '[' be stored*

res = ""; *// would store the string inside '[]'*

i++;

} else if (s.charAt(i) == ']') {

String prev = strStack.pop();

int number = numStack.pop();

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

prev += res;

}

res = prev;

i++;

} else {

res += s.charAt(i++);

}

}

return res;

}