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.

思路：第一次掃描數組，把所有元素放入Hashset

第二次掃描數組，把掃到的元素remove出hashset。如果該元素N在hashset就分別向左掃N-1 和向右掃N+1。得出包括元素N的連續序列。記錄該序列長度。

如果hashset中沒有某元素N，說明N已經包含在之前找到的連續數組中，那就需重複處理。

public class Solution {

public int longestConsecutive(int[] nums) {

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

HashSet<Integer> set=new HashSet<Integer>();

for(int n:nums)set.add(n);

int max=1;

for(int n:nums)

{

if(set.remove(n))

{

int val=n;

int sum=1;

while(set.remove(val-1))

{

val--;

sum++;

}

val=n;

while(set.remove(val+1))

{

val++;

sum++;

}

max=Math.max(max,sum);

}

}

return max;

}

}

287. Find the Duplicate Number

給你一個數組，長度為n+1，每個元素在[1,n]，只有一個重複元素。找出這個重複的元素。

思路：linked list中找環的起點思路一樣。

|  |
| --- |
| class Solution(object): |
|  | def findDuplicate(self, nums): |
|  | """ |
|  | :type nums: List[int] |
|  | :rtype: int |
|  | """ |
|  | # Treat each (key, value) pair of the array as the (pointer, next) node of the linked list, |
|  | # thus the duplicated number will be the begin of the cycle in the linked list. |
|  | # Besides, there is always a cycle in the linked list which |
|  | # starts from the first element of the array. |
|  | slow = nums[0] |
|  | fast = nums[nums[0]] |
|  | while slow != fast: |
|  | slow = nums[slow] |
|  | fast = nums[nums[fast]] |
|  |  |
|  | fast = 0 |
|  | while slow != fast: |
|  | slow = nums[slow] |
|  | fast = nums[fast] |
|  | return slow |

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.

For example,   
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.



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.

思路：int[] leftMaxHeight記錄當前i位置左邊最大的高度, int[] rightMaxHeight記錄當前位置i右邊的最大高度。

對於位置i，minheight=MIN( leftMaxHeight[i], rightMaxHeight[i]) 如果minheight<當前位置的height，那就可以裝水，可以裝minheight-height的水

**class** **Solution** {

**public:**

**int** trap(**int** A[], **int** n) {

**if**(n<**3**) **return** **0**;

vector<**int**> leftHeight(n,**0**);

vector<**int**> rightHeight(n,**0**);

**int** water = **0**;

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

leftHeight[i] = max(leftHeight[i-**1**], A[i-**1**]);

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

rightHeight[i] = max(rightHeight[i+**1**], A[i+**1**]);

**int** minHeight = min(leftHeight[i], rightHeight[i]);

**if**(minHeight>A[i]) water += (minHeight - A[i]);

}

**return** water;

}

};

45. Jump Game II

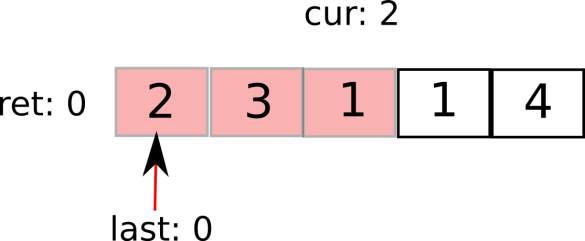
給你一個數組，數組每個元素代表最多跳多遠，一開始在第一個位置，找出跳到最後一個元素的最少步數。

思路：

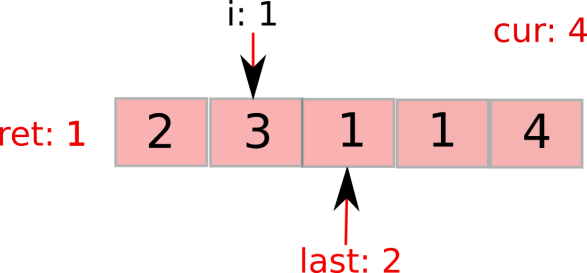
比如就是我们题目中的[2,3,1,1,4]。初始状态是这样的：cur表示最远能覆盖到的地方，用红色表示。last表示已经覆盖的地方，用箭头表示。它们都指在第一个元素上。



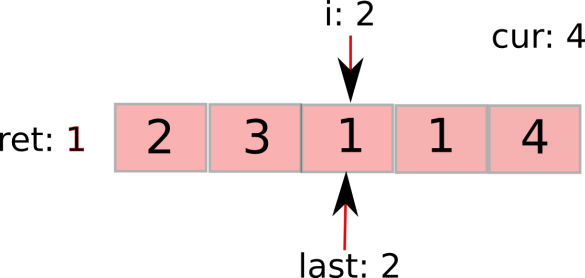
接下来，第一元素告诉cur，最远咱可以走2步。于是：



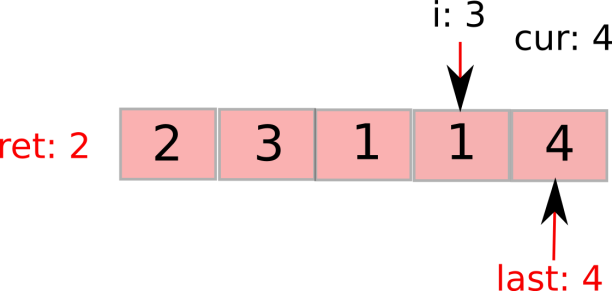
下一循环中，i指向1（图中的元素3），发现，哦，i小于last能到的范围，于是更新last（相当于说，进入了新的势力范围），步数ret加1.同时要更新cur。因为最远距离发现了。



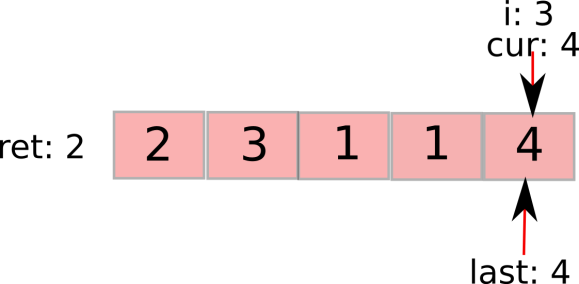
接下来，i继续前进，发现i在当前的势力范围内，无需更新last和步数ret。更新cur。



i继续前进，接下来发现**超过**当前势力范围，更新last和步数。cur已然最大了。



最后，i到最后一个元素。依然在势力范围内，遍历完成，返回ret。



class Solution {

public:

int jump(int A[], int n) {

int ret = 0;

int last = 0;

int curr = 0;

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

if (i > last) {

last = curr;

++ret;

}

curr = max(curr, i+A[i]);

}

return ret;

}

};

41. First Missing Positive

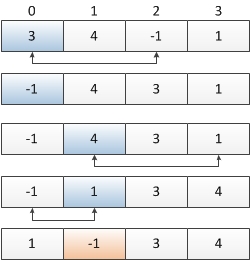
Given an unsorted integer array, find the first missing positive integer.

For example,

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

**思路**：交换数组元素，使得数组中第i位存放数值(i+1)。最后遍历数组，寻找第一个不符合此要求的元素，返回其下标。整个过程需要遍历两次数组，复杂度为O(n)。

下图以题目中给出的第二个例子为例，讲解操作过程。



public class Solution {

public int firstMissingPositive(int[] A) {

int i = 0;

while(i < A.length){

if(A[i] == i+1 || A[i] <= 0 || A[i] > A.length) i++;

else if(A[A[i]-1] != A[i]) swap(A, i, A[i]-1);

else i++;

}

i = 0;

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

return i+1;

}

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

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

73. Set Matrix Zeroes

給你一個矩陣arr[][], 如果arr[i][j]==0,將該行該列全部元素置0。要求不用額外空間。

思路：將第一行和第一列作為標誌

第一次掃描整個數組，如果發現arr[i][j]==0，該行的第一位和該列第一位置0。

第二次掃描除了第一行和第一列外的整個數組，完成置0操作

回頭再處理第一行第一列

1. **public** **void** setZeroes(**int**[][] matrix) {
2. **if**(matrix==**null** || matrix.length==0 || matrix[0].length==0)
3. **return**;
4. **boolean** rowFlag = **false**;
5. **boolean** colFlag = **false**;
6. **for**(**int** i=0;i<matrix.length;i++)
7. {
8. **if**(matrix[i][0]==0)
9. {
10. colFlag = **true**;
11. **break**;
12. }
13. }
14. **for**(**int** i=0;i<matrix[0].length;i++)
15. {
16. **if**(matrix[0][i]==0)
17. {
18. rowFlag = **true**;
19. **break**;
20. }
21. }
22. **for**(**int** i=1;i<matrix.length;i++)
23. {
24. **for**(**int** j=1;j<matrix[0].length;j++)
25. {
26. **if**(matrix[i][j]==0)
27. {
28. matrix[i][0] = 0;
29. matrix[0][j] = 0;
30. }
31. }
32. }
33. **for**(**int** i=1;i<matrix.length;i++)
34. {
35. **for**(**int** j=1;j<matrix[0].length;j++)
36. {
37. **if**(matrix[i][0]==0 || matrix[0][j]==0)
38. matrix[i][j] = 0;
39. }
40. }
41. **if**(colFlag)
42. {
43. **for**(**int** i=0;i<matrix.length;i++)
44. {
45. matrix[i][0] = 0;
46. }
47. }
48. **if**(rowFlag)
49. {
50. **for**(**int** i=0;i<matrix[0].length;i++)
51. {
52. matrix[0][i] = 0;
53. }
54. }
55. }

4 Sums

給一個數組你，找出所有4個數的組合，和為target

思路：

數組從小到大排序

4sum 調用3 sum， 3sum調用2sum

挑選每一位數時，如果當前的數與前一位一樣的，跳過。就可以消除重複的組合。例如：挑第4位最高位時，如果當前位跟之前一樣就跳過，之後3sum在當前位後面的數組範圍內進行

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

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

int len = nums.length;

if (nums == null || len < 4)

return res;

Arrays.sort(nums);

int max = nums[len - 1];

if (4 \* nums[0] > target || 4 \* max < target)

return res;

int i, z;

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

z = nums[i];

if (i > 0 && z == nums[i - 1])// avoid duplicate

continue;

if (z + 3 \* max < target) // z is too small

continue;

if (4 \* z > target) // z is too large

break;

if (4 \* z == target) { // z is the boundary

if (i + 3 < len && nums[i + 3] == z)

res.add(Arrays.asList(z, z, z, z));

break;

}

threeSumForFourSum(nums, target - z, i + 1, len - 1, res, z);

}

return res;

}

/\*

\* Find all possible distinguished three numbers adding up to the target

\* in sorted array nums[] between indices low and high. If there are,

\* add all of them into the ArrayList fourSumList, using

\* fourSumList.add(Arrays.asList(z1, the three numbers))

\*/

public void threeSumForFourSum(int[] nums, int target, int low, int high, ArrayList<List<Integer>> fourSumList,

int z1) {

if (low + 1 >= high)

return;

int max = nums[high];

if (3 \* nums[low] > target || 3 \* max < target)

return;

int i, z;

for (i = low; i < high - 1; i++) {

z = nums[i];

if (i > low && z == nums[i - 1]) // avoid duplicate

continue;

if (z + 2 \* max < target) // z is too small

continue;

if (3 \* z > target) // z is too large

break;

if (3 \* z == target) { // z is the boundary

if (i + 1 < high && nums[i + 2] == z)

fourSumList.add(Arrays.asList(z1, z, z, z));

break;

}

twoSumForFourSum(nums, target - z, i + 1, high, fourSumList, z1, z);

}

}

/\*

\* Find all possible distinguished two numbers adding up to the target

\* in sorted array nums[] between indices low and high. If there are,

\* add all of them into the ArrayList fourSumList, using

\* fourSumList.add(Arrays.asList(z1, z2, the two numbers))

\*/

public void twoSumForFourSum(int[] nums, int target, int low, int high, ArrayList<List<Integer>> fourSumList,

int z1, int z2) {

if (low >= high)

return;

if (2 \* nums[low] > target || 2 \* nums[high] < target)

return;

int i = low, j = high, sum, x;

while (i < j) {

sum = nums[i] + nums[j];

if (sum == target) {

fourSumList.add(Arrays.asList(z1, z2, nums[i], nums[j]));

x = nums[i];

while (++i < j && x == nums[i]) // avoid duplicate

;

x = nums[j];

while (i < --j && x == nums[j]) // avoid duplicate

;

}

if (sum < target)

i++;

if (sum > target)

j--;

}

return;

}

**325**[Maximum Size Subarray Sum Equals k 最大子数组之和为k](http://www.cnblogs.com/grandyang/p/5336668.html)

**subarray默認需要連續的子數組**

**用hashmap記錄從數組開始到中間元素N的求和，以及N的位置**

**每搜索到下一個元素，就算一個sum[n]放入hashmap。**

**并在hashmap中找sum[n]-hashmap=k,即hashmap=sum[n]-k是否存在，存在就說明有子數組和為k,接著就計算最大長度。**

**注意：數組求和可能會出現多個位置的和都是一樣。所以hashmap值記錄同一個和時候，最左邊，即最長的索引值位置**

**311 稀疏矩陣相乘**

**I x J 矩陣與Jx K矩陣相乘 得到的矩陣肯定是I x K**

C[i][j]=A[i][0]\*B[0][j] + A[i][1]\*B[1][j] + ... + A[i][k]\*B[k][j]

**先遍歷A數組的元素如果A[i][k]不為0，則可以相乘。**

**res[I,j]+=A[I,k]\*B[k,J]**

class Solution {

public:

vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {

vector<vector<int>> res(A.size(), vector<int>(B[0].size()));

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

for (int k = 0; k < A[0].size(); ++k) {

if (A[i][k] != 0) {

for (int j = 0; j < B[0].size(); ++j) {

if (B[k][j] != 0) res[i][j] += A[i][k] \* B[k][j];

}

}

}

}

return res;

}

};

# 253Meeting 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.

**思路：先排序，我們逐一找，先將第一個會議放入第一個房間，如果第二個會議有衝突，就放入第二個房間，依次類推。**

**每次第n個會議，我們會找出所有已經安排好的最早結束的會議，如果不衝突，就不需增加房間，否則需要增加房間。如果用list來記錄慢，所以使用prioriyqueue。**

****

215. Kth Largest Element in an Array

思路：quicksort類似

把問題變成找出第nums.length-k+1 小的元素

使用quicksort類似的partition算法，如果中間pivot的元素位置剛好長度是k，就找到。

如果左邊部分長過k，就到左邊找。左邊部分短過k，在右邊找

public class Solution {

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

return partition(nums,0,nums.length-1,nums.length-k+1);

}

//find the kth smallest element

private int partition(int[] nums, int start , int end, int k){

int key=nums[start];

int index=start;

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

if(nums[i]<key){

int temp=nums[i];

nums[i]=nums[index+1];

nums[index+1]=temp;

index++;

}

}

int temp=nums[start];

nums[start]=nums[index];

nums[index]=temp;

if(index-start+1==k)return nums[index];

else if(index-start+1>k)return partition(nums,start,index-1,k);

else return partition(nums,index+1,end,k-(index-start+1));

}

}

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

思路： 從後面往前掃，找到第一個不是遞減的數字。接著將這個數字num【i】 替換成 右邊比這個數字大的當中找一個最小的，兩個替換位置。最後，把num【i+1】到最後的字符串翻轉

public void nextPermutation(int[] A) {

if(A == null || A.length <= 1) return;

int i = A.length - 2;

while(i >= 0 && A[i] >= A[i + 1]) i--; // Find 1st id i that breaks descending order

if(i >= 0) { // If not entirely descending

int j = A.length - 1; // Start from the end

while(A[j] <= A[i]) j--; // Find rightmost first larger id j

swap(A, i, j); // Switch i and j

}

reverse(A, i + 1, A.length - 1); // Reverse the descending sequence

}

public void swap(int[] A, int i, int j) {

int tmp = A[i];

A[i] = A[j];

A[j] = tmp;

}

public void reverse(int[] A, int i, int j) {

while(i < j) swap(A, i++, j--);

}

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

思路：從數組後面掃到前面，找到第一個不是遞減的點 index。

在index後面，從數組最後掃上來，找到最右一個比index大的（這個數相當於在大於index中挑最小的），將index和這個數換位

最後index+1 到數組最後元素為止，整個區間翻轉

public void nextPermutation(int[] A) {

if(A == null || A.length <= 1) return;

int i = A.length - 2;

while(i >= 0 && A[i] >= A[i + 1]) i--; // Find 1st id i that breaks descending order

if(i >= 0) { // If not entirely descending

int j = A.length - 1; // Start from the end

while(A[j] <= A[i]) j--; // Find rightmost first larger id j

swap(A, i, j); // Switch i and j

}

reverse(A, i + 1, A.length - 1); // Reverse the descending sequence

}

public void swap(int[] A, int i, int j) {

int tmp = A[i];

A[i] = A[j];

A[j] = tmp;

}

public void reverse(int[] A, int i, int j) {

while(i < j) swap(A, i++, j--);

}

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.

思路：盡量是數組的位置i放入i+1.如果當前掃描到的是負數或者超出數組index範圍的不管。注意，swap之後i不能加1。因為可能swap后i位置的數不是i+1，需要繼續swap這個位置。如果num[index] = num[num[index] – 1] swap依然會死循環，別管他。

public class Solution {

public int firstMissingPositive(int[] nums) {

if(nums.length == 0)

return 1;

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

if(nums[i] > 0 && nums[i] <= nums.length && nums[i] != nums[nums[i] - 1]){

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

}else{

i++;

}

}

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

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

return i + 1;

}

}

return nums.length + 1;

}

public void swap(int[] nums, int i1, int i2){

int temp = nums[i1];

nums[i1] = nums[i2];

nums[i2] = temp;

}

}