<https://leetcode.ca/all/1060.html>

Given a sorted array A of unique numbers, find the *K*-th missing number starting from the leftmost number of the array.

Example 1:

Input: A = [4,7,9,10], K = 1

Output: 5

Explanation:

The first missing number is 5.

Example 2:

Input: A = [4,7,9,10], K = 3

Output: 8

Explanation:

The missing numbers are [5,6,8,...], hence the third missing number is 8.

Example 3:

Input: A = [1,2,4], K = 3

Output: 6

Explanation:

The missing numbers are [3,5,6,7,...], hence the third missing number is 6.

Note:

* 1 <= A.length <= 50000
* 1 <= A[i] <= 1e7
* 1 <= K <= 1e8

**The only difference between L1060 and L1539 is L1060 start from leftmost element and L1539 start from 1 as first positive integer**

**Attempt 1: 2023-09-18**

**Solution 1:  Hash Table (10 min)**

class Solution {

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

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

for(int num : nums) {

set.add(num);

}

int i = nums[0];

int result = 0;

while(k > 0) {

if(!set.contains(i)) {

k--;

result = i;

}

i++;

}

return result;

}

}

========================================================

No need extra variable 'result'

class Solution {

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

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

for(int num : nums) {

set.add(num);

}

int i = nums[0];

//int result = 0;

while(k > 0) {

if(!set.contains(i)) {

k--;

//result = i;

}

i++;

}

//return result;

return i - 1;

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 2:  nums[i] and (i + 1) relation (30 min, difficult to think about)**

class Solution {

// e.g

// nums = [4,7,9,10], k = 3, expected = 8

// i = 1

// -> missing = nums[1] - nums[0] - (1 - (1 - 1)) = 7 - 4 - 1 = 2

// -> k(=3) > missing(=2)

// -> k -= missing -> 3 - 2 = 1

// i = 2

// -> missing = nums[2] - nums[1] - (2 - (2 - 1)) = 9 - 7 - 1 = 1

// -> k(=1) <= missing(=1)

// -> return nums[i - 1] + k = 7 + 1 = 8

// ==========================================

// e.g

// nums = [1,2,4], k = 3, expected = 6

// i = 1

// -> missing = nums[1] - nums[0] - (1 - (1 - 1)) = 2 - 1 - 1 = 0

// -> k(=3) > missing(=0)

// -> k -= missing -> 3 - 3 = 0

// i = 2

// -> missing = nums[2] - nums[1] - (2 - (2 - 1)) = 4 - 2 - 1 = 1

// -> k(=3) < missing(=1)

// -> k -= missing -> 3 - 1 = 2

// for loop end -> return nums[n - 1] + k = 4 + 2 = 6

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

int n = nums.length;

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

// missing number 的个数 = nums[end] - nums[start] - (end - start)

int missing = nums[i] - nums[i - 1] - (i - (i - 1));

if(k <= diff) {

return nums[i - 1] + k;

}

k -= missing;

}

return nums[n - 1] + k;

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://www.cnblogs.com/cnoodle/p/13193731.html>

有序数组中的缺失元素。

给你一个 **严格升序排列** 的正整数数组 arr 和一个整数 k 。请你找到这个数组里第 k 个缺失的正整数。

这道题有两种思路，一种线性，一种二分法。需要注意一个 corner case，如果 K 大于数组的长度了，那么第 K 个缺失的元素是最后一个元素 + K。同时，**missing number 的个数 = nums[end] - nums[start] - (end - start)**。举例，4和7之间缺失2个数字 = (7 - 4) - (1 - 0)。

A = [4,7,9,10], K = 3

首先是线性的做法。因为数组是有序的且递增的，所以每次只要计算一下每两个数字之间的差值 - 1就知道是否到 K 了。

一开始 7 - 4 = 3，虽然两个数字的差值是 3，但是实际缺失的数字只有 2 个，所以要减一。如果每两个数字之间的差值小于 K，则需要将 K 减去两数字之间的差值之后，再带入下两个数字之间去比较。反之如果某两个数字之间的差值小于当前的 K，那么第 K 个缺失的数字 = nums[i - 1] + K。

时间O(n)

空间O(1)

class Solution {

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

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

if (nums[i] - nums[i - 1] - 1 >= k) {

return nums[i - 1] + k;

}

k -= nums[i] - nums[i - 1] - 1;

}

return nums[nums.length - 1] + k;

}

}

**Solution 3:  Binary Search (120 min, difficult on getting this idea)**

**使用L704.Binary Search的找下界模版 (Find the first number able to produce k as difference between its value and its index)**

**Style 1: The Binary Search auxiliary method for calculating missing numbers determine based on range [0, curr\_index], the range start index always keep as 0, only change end index, the missing number count = nums[end] - nums[0] - (end - 0), and we compare this count with k by using L704.Binary Search template "Find Lower Boundary"**

class Solution {

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

int n = nums.length;

int lo = 0;

int hi = n - 1;

int total\_missing = missing(nums, hi);

// When k even larger than total missing number

if(total\_missing < k) {

return nums[hi] + k - total\_missing;

}

while(lo <= hi) {

int mid = lo + (hi - lo) / 2;

// Find missing number count between nums[0] to nums[mid]

int miss = missing(nums, mid);

// 'miss >= k', the '=' here means even we find a candidate

// 'hi' index satisfy condition, for getting first missing

// number we will continue to try by shrinking 'hi' to 'mid - 1'

if(miss >= k) {

hi = mid - 1;

// 'miss < k' means no need to try index <= mid anymore, as

// all of index <= k only make 'miss < k', not match 'miss == k'

// bottom line, so just increase 'lo' to 'mid + 1' to begin

// next iteration

} else {

lo = mid + 1;

}

}

// 'k - missing(nums, lo - 1)' means for total k missing

// numbers, the remaining number count after removing total

// missing number count between index [0, lo - 1], which

// needs to be added onto nums[lo - 1] to get exactly kth

// missing number

return nums[lo - 1] + k - missing(nums, lo - 1);

}

// The Binary Search auxiliary method for calculating missing

// numbers determine based on range [0, curr\_index], the range

// start index always keep as 0, only change end index, the

// missing number count = nums[end] - nums[0] - (end - 0)

private int missing(int[] nums, int end) {

return nums[end] - nums[0] - end;

}

}

Time Complexity: O(logN)

Space Complexity: O(1)

**Refer to**

<https://www.zsbeike.com/technology/026c49c7ea.html>

fun getKMissingNumber(nums: IntArray, k: Int): Int {

//Solution 2:

val n = nums.size

val missing = nums[n - 1] - nums[0] - (n - 1 - 0)

//If the missing numbers count of the whole array < k, then missing number must be after nums[n-1].

//then: res = nums[n-1] + missingCount.

if (missing < k) {

println("result:${nums[n - 1] + k - missCount(nums, n - 1)}")

return nums[n - 1] + k - missCount(nums, n - 1)

}

var left = 0

var right = n - 1

while (left < right) {

val mid = left + (right - left) / 2

if (missCount(nums, mid) < k) {

//fall in right side

left = mid + 1

} else {

right = mid - 1

}

}

println("result2:${nums[left - 1] + k - missCount(nums, left - 1)}")

return nums[left - 1] + k - missCount(nums, left - 1)

}

//for example: 4,(5,6),7 -- > 7 - 4 - (1 - 0) = 3 - 1 = 2;

private fun missCount(nums: IntArray, index: Int): Int {

return nums[index] - nums[0] - index

}

**Refer to**

<https://blog.csdn.net/qq_29051413/article/details/108679642>

解题思路

已知教组A是有序的，且每一个教字都是独一无二的，那么，如果 A1没有任何缺失，两个相邻元素差的绝对值应该为1.

比如: A[] = {5,6,7,8} 在中间就不缺任何数字，要缺也是从最后一个元素开始算起

如果我们要计算，截止到某个数字为止，总共缺了多少个数字，应该这样计算:

第 0个元素到第 n 个元素应该加了 n 次 1，第 n 位数字应该是第 0 位数字加上 n;

比如: 假设现在有数组 A[] = {4,7,9,10}，要计算截至到第 3 位总共缺失是数字，可以这样计算:

(A[3] - A[0]) - (3-0) = 10 - 4 - 3 = 4

A[3] - A[0] 表示第 3 位从第 0 位开始，实际加了多少次 1

3-0 表示第 3 位从第 0 位开始，如果不缺，应该加了多少次 1:

两者的差值，就是丢失的数字的个数。

公式合并简写为: missing(idx) = Alidx] - A[0] - idx。表示截至到第 idx 位为止，总共缺了多少个数字.

1. 定义一个idx = 0;

2. 判断 missing(idx)和 K 的大小;

3. 如果 missing(idx) < K，idx++，回到第2步;

4. 当出现 missing(idx)>= K，说明截至到第 idx 为止，缺失的数大于等于 K，那么缺失的数字就在idx-1 到 idx 之间

5. K - missing(idx-1) + Alidx-1] 即为答案。

class Solution {

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

int length = nums.length;

if(k > missing(length - 1, nums)) {

return nums[length - 1] + k - missing(length - 1, nums);

}

int left = 0, right = length - 1, mid;

while(left < right) {

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

if(missing(mid, nums) < k) {

left = mid + 1;

} else {

right = mid;

}

}

return nums[left - 1] + k - missing(left - 1, nums);

}

// 到 A[idx] 为止总共缺失的数字个数

private int missing(int idx, int[] nums) {

// 应该有的数量 = 实际有的数量 - 缺失的数量

return nums[idx] - nums[0] - idx;

}

}

**Style 2: The Binary Search auxiliary method for calculating missing numbers determine based on range [start\_index, curr\_index], the range start index always keep as 0, only change end index, the missing number count = nums[end] - nums[start] - (end - start), and we compare this count with k by using L704.Binary Search template "Find Lower Boundary"**

**Wrong solution:**

class Solution {

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

int n = nums.length;

int lo = 0;

int hi = n - 1;

int total\_missing = missing(nums, lo, hi);

// When k even larger than total missing number

if(total\_missing < k) {

return nums[hi] + k - total\_missing;

}

while(lo <= hi) {

int mid = lo + (hi - lo) / 2;

int miss = missing(nums, lo, mid);

if(miss >= k) {

hi = mid - 1;

} else {

k -= miss;

lo = mid + 1;

}

}

return nums[lo - 1] + k;

}

private int missing(int[] nums, int start, int end) {

return nums[end] - nums[start] - (end - start);

}

}

**Correction Solution:**

class Solution {

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

int n = nums.length;

int lo = 0;

int hi = n - 1;

int total\_missing = missing(nums, lo, hi);

// When k even larger than total missing number

if(total\_missing < k) {

return nums[hi] + k - total\_missing;

}

// 循环终止条件lo + 1 >= hi，因为基础条件改变，不再适合使用Style 1中的标准

// Find Lower Boundary模版（lo <= hi的终止条件式lo > hi，终止条件对应区

// 间为空），这里的终止条件对应的区间lo和hi之间必须有一个数的间隔，意味着lo不会

// 超越或等于hi，为什么会这样呢？因为在Style 1中我们固定了lo总是为0，唯一变动

// 的是hi，可以理解为数轴的左端不动，只向右不停的移动右端来寻找满足条件的左右两

// 端的中值mid，而在涉及到missing number count的计算时，Style 1中采用的

// (nums[mid] - nums[0] - mid)公式只涉及到mid，完全不涉及与lo的相对关系，

// 就算采用模版中lo = mid + 1导致lo大于mid也丝毫不影响公式的计算，然而在本

// 解法Style 2中，我们会发现(nums[mid] - nums[lo] - (mid - lo))的公式

// 会涉及到mid与lo的相对关系，那么在涉及到公式有效性的限制时，必须保证lo不会大于

// mid，这不仅需要终止条件改变保证lo不会超越甚至等于hi，而且需要在循环迭代lo

// 的值的时候不能出现lo = mid + 1导致lo大于mid的情况，所以配合之下lo = mid，

// 同样，hi = mid

while(lo + 1 < hi) {

int mid = lo + (hi - lo) / 2;

int miss = missing(nums, lo, mid);

if(miss >= k) {

//hi = mid - 1;

hi = mid;

} else {

// 别忘了每当miss < k需要提高lo到mid进行下一轮搜索的时候，

// 不同于Style 1是整体性区间[0, mid]的关系，这里的Style 2

// 是当前区间[lo, mid]的关系，我们需要在下一轮（下一个区间）

// 之前将之前一轮（前一个区间）的missing number count从

// k中去除，让k同步刷新到下一轮的状态

k -= miss;

//lo = mid + 1;

lo = mid;

}

}

return nums[lo] + k;

}

}

Time Complexity: O(logN)

Space Complexity: O(1)

**Refer to**

<https://www.cnblogs.com/cnoodle/p/13193731.html>

再来是二分法。**以后看到 input 是有序的，就要想想能不能往二分法上靠，以减小时间复杂度。**

因为缺失数字的个数 missing = nums[end] - nums[start] - (end - start)，所以如果 K > missing，也就是说如果数组中缺失的数字个数小于 K 的话，结果是 nums[end] + (k - missing)。如果是一般的情形，即缺失的数字夹在数组中间的话，那么我们就用二分法去找。这里的二分，看的是数组中间那个元素 nums[mid] 跟 nums[start] 元素之间有多少个缺失的元素，如果这个差值小于 K，说明要找的数字在数组的右半边，start = mid；反之如果 nums[mid] 跟 nums[start] 之间缺失元素的个数大于 K，则说明第 K 个缺失的数字在数组的左半边。

时间O(logn)

空间O(1)

class Solution {

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

int start = 0;

int end = nums.length - 1;

int missing = nums[end] - nums[start] - (end - start);

// corner case

if (k > missing) {

return nums[end] + (k - missing);

}

// normal case

while (start + 1 < end) {

int mid = start + (end - start) / 2;

missing = nums[mid] - nums[start] - (mid - start);

if (missing < k) {

k -= missing;

start = mid;

} else {

end = mid;

}

}

return nums[start] + k;

}

}