# 题目

现有一个按 升序 排列的整数数组nums ，其中每个数字都 互不相同 。

给你一个整数k ，请你找出并返回从数组最左边开始的第k个缺失数字。

示例 1：

输入：nums = [4,7,9,10], k = 1

输出：5

解释：第一个缺失数字为 5 。

示例 2：

输入：nums = [4,7,9,10], k = 3

输出：8

解释：缺失数字有 [5,6,8,...]，因此第三个缺失数字为8。

示例 3：

输入：nums = [1,2,4], k = 3

输出：6

解释：缺失数字有[3,5,6,7,...]，因此第三个缺失数字为6。

提示：

1 <= nums.length <= 5 \* 104

1 <= nums[i] <= 107

nums 按 升序 排列，其中所有元素 互不相同 。

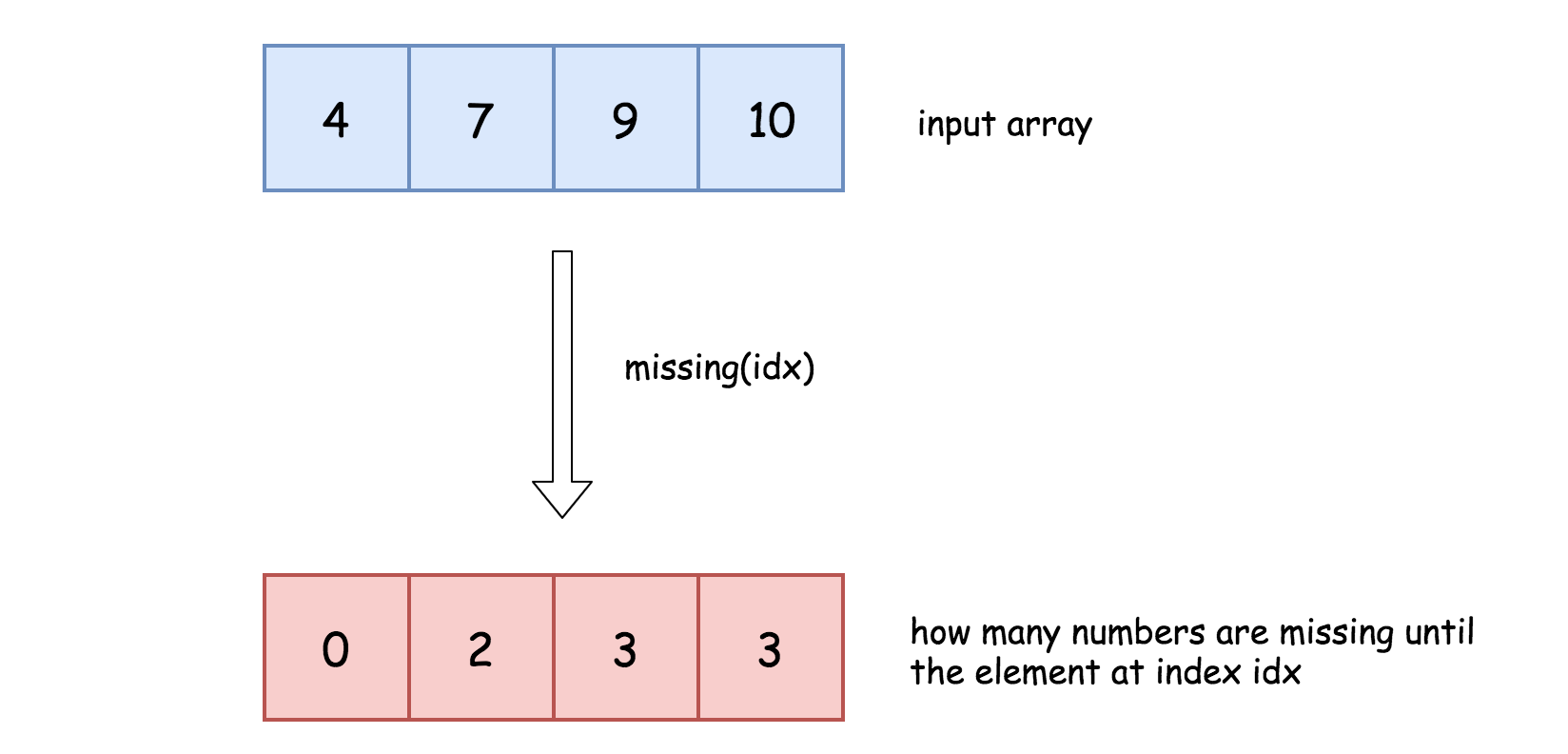
1 <= k <= 108

进阶：你可以设计一个对数时间复杂度（即，O(log(n))）的解决方案吗？

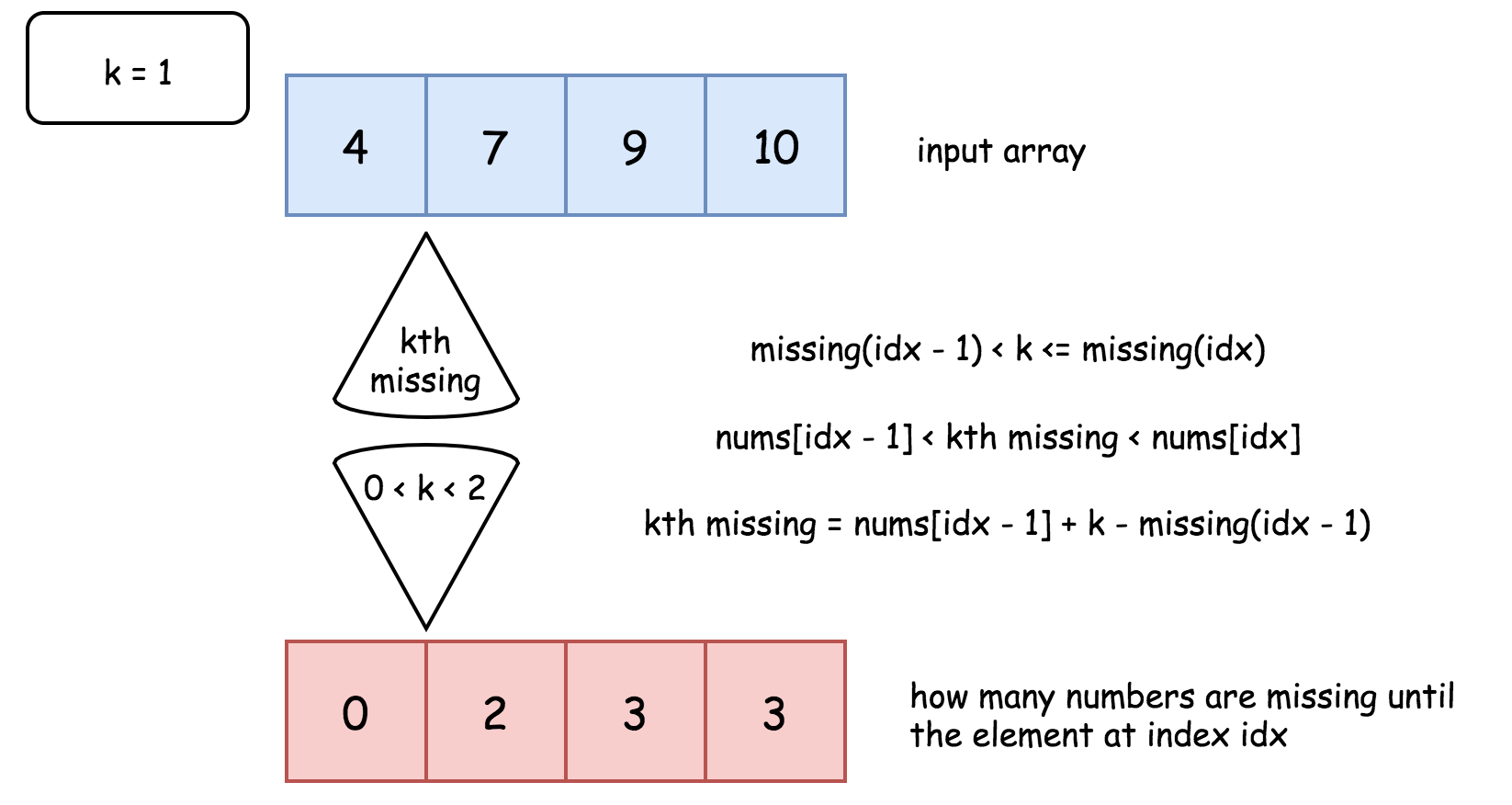
# 分析

## 方法一：线性扫描

我们用函数 missing(idx) 表示到 A[idx]­ 为止总共缺失的数字个数，下图给出了一个例子，数组为 [4, 7, 9, 10]，缺失的数字为 [5, 6, 8]，到数组中的数字 7, 9, 10 为止缺失的数字个数分别为 2, 3, 3 个。



如果我们能得到所有的missing(idx)，那么我们只要对其进行线性扫描就可以知道第K个缺失的数字了。具体来说，如果有missing(idx - 1) < K <= missing(idx)，那么就说明，第K个缺失的数字在A[idx - 1]和A[idx]之间，并且它比 A[idx - 1] 大K - missing(idx)。



而计算出所有missing(idx) 的方法也很简单，从A[0]到A[idx]应该有A[idx] - A[0] + 1个数字，而实际上根据下标，只有idx + 1个数字，因此缺失的数字个数为上面两个式子之差，即A[idx] - A[0] - idx。

class Solution {

// Return how many numbers are missing until nums[idx]

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

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

}

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

int n = nums.length;

// If kth missing number is larger than

// the last element of the array

if (k > missing(n - 1, nums))

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

int idx = 1;

// find idx such that

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

while (missing(idx, nums) < k) idx++;

// kth missing number is larger than nums[idx - 1]

// and smaller than nums[idx]

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

}

}

复杂度分析

时间复杂度：O(N)，最多只会遍历整个数组一次。

空间复杂度：O(1)。

C++：

class Solution1 {

public:

int missingElement(vector<int>& nums, int k) {

int N = nums.size();

assert(N >= 1);

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

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

if (missing >= k) {

return nums[i - 1] + k;

} else {

k -= missing;

}

}

return nums[N - 1] + k;

}

};

## 方法二：二分查找

分析

在方法一中，我们是通过线性扫描的方法找到missing(idx - 1) < K <= missing(idx)对应的idx的。事实上，由于missing(idx)是单调不减的，我们可以通过二分查找的方法找到满足条件的idx，并将时间复杂度降低到O(logN)。

class Solution {

// Return how many numbers are missing until nums[idx]

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

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

}

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

int n = nums.length;

// If kth missing number is larger than

// the last element of the array

if (k > missing(n - 1, nums))

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

int left = 0, right = n - 1, pivot;

// find left = right index such that

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

while (left != right) {

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

if (missing(pivot, nums) < k) left = pivot + 1;

else right = pivot;

}

// kth missing number is larger than nums[idx - 1]

// and smaller than nums[idx]

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

}

}

复杂度分析：

时间复杂度：O(logN)。

空间复杂度：O(1)。

C++：

class Solution {

public:

int missingElement(vector<int>& nums, int k) {

int N = nums.size();

int missingCnt = missingCountUntil(N - 1, nums);

if (k > missingCnt) {

return nums[N - 1] + k - missingCnt;

}

int left = 0;

int right = N - 1;

while (left <= right) {

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

auto missingCnt = missingCountUntil(mid, nums);

if (missingCnt < k) {

left = mid + 1;

} else {

right = mid - 1;

}

}

// left stop at the first element that has >= k missing elements

// therefore we need to calculate from left - 1 position.

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

}

private:

int missingCountUntil(int index, vector<int>& nums) {

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

}

};