<https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/>

Given an array nums of n integers where nums[i] is in the range [1, n], return *an array of all the integers in the range* [1, n] *that do not appear in* nums.

**Example 1:**

Input: nums = [4,3,2,7,8,2,3,1]

Output: [5,6]

**Example 2:**

Input: nums = [1,1]

Output: [2]

**Constraints:**

* n == nums.length
* 1 <= n <= 105
* 1 <= nums[i] <= n

**Follow up:** Could you do it without extra space and in O(n) runtime? You may assume the returned list does not count as extra space.

**Solution 1: Efficient way to flip original element to negative (30 min)**

class Solution {

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

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

int n = nums.length;

// e.g nums = [4,3,2,7,8,2,3,1]

// i = 0 -> index = 3 -> nums = [4,3,2,-7,8,2,3,1]

// i = 1 -> index = 2 -> nums = [4,3,-2,-7,8,2,3,1]

// i = 2 -> index = 1 -> nums = [4,-3,-2,-7,8,2,3,1]

// i = 3 -> index = 6 -> nums = [4,-3,-2,-7,8,2,-3,1]

// i = 4 -> index = 7 -> nums = [4,-3,-2,-7,8,2,-3,-1]

// i = 5 -> index = 1 -> nums = [4,-3,-2,-7,8,2,-3,-1]

// i = 6 -> index = 2 -> nums = [4,-3,-2,-7,8,2,-3,-1]

// i = 7 -> index = 0 -> nums = [-4,-3,-2,-7,8,2,-3,-1]

// 注意观察，在i = 2和i = 5的情况下，index两次出现了1，但是在第二次

// 出现1的时候我们依然保留了第一次出现1的情况下翻转nums[1]=3所取得的

// -3的结果，为什么要保留-3呢？为什么只对剩余的正数做翻转而翻转完后

// 就不动了你？因为我们的目标是找到消失的数，也就是没有机会被第一次翻转

// 为负数的数，而某些数被第二次翻转相当于占用了这些没有机会被第一次翻转

// 的数的翻转机会，那么只要一个数被翻转了，无论多少次我们都记录为负数，

// 而未能得到哪怕一次翻转机会的数就被从始至终保留为正数，在第二次扫描中

// 可以轻易通过保留下来的正数坐标找到消失的数，关系为坐标+1

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

int index = Math.abs(nums[i]) - 1;

if(nums[index] > 0) {

nums[index] = -nums[index];

}

}

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

if(nums[i] > 0) {

result.add(i + 1);

}

}

return result;

}

}

Time Complexcity: O(N)

Space Complexcity: O(1)

**Refer to**

<https://grandyang.com/leetcode/448/>

这道题让我们找出数组中所有消失的数，**跟之前那道[Find All Duplicates in an Array](http://www.cnblogs.com/grandyang/p/6209746.html)极其类似，那道题让找出所有重复的数字，这道题让找不存在的数，这类问题的一个重要条件就是1 ≤ a[i] ≤ n (n = size of array)，不然很难在O(1)空间和O(n)时间内完成。**三种解法也跟之前题目的解法极其类似。首先来看第一种解法，这种解法的思路路是，**对于每个数字nums[i]，如果其对应的nums[nums[i] - 1]是正数，我们就赋值为其相反数，如果已经是负数了，就不变了，那么最后我们只要把留下的整数对应的位置加入结果res中即可**，参见代码如下：

class Solution {

public:

vector<int> findDisappearedNumbers(vector<int>& nums) {

vector<int> res;

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

int idx = abs(nums[i]) - 1;

nums[idx] = (nums[idx] > 0) ? -nums[idx] : nums[idx];

}

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

if (nums[i] > 0) {

res.push\_back(i + 1);

}

}

return res;

}

};

**Solution 2: Efficient way to keep swapping if nums[i] != nums[nums[i] - 1] (30 min)**

class Solution {

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

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

int n = nums.length;

// e.g nums = [4,3,2,7,8,2,3,1]

// -> nums = [1,2,3,4,3,2,7,8]

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

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

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

i--;

}

}

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

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

result.add(i + 1);

}

}

return result;

}

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

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

}

Time Complexcity: O(N)

Space Complexcity: O(1)

**Refer to**

<https://grandyang.com/leetcode/448/>

第二种方法是将nums[i]置换到其对应的位置nums[nums[i]-1]上去，比如对于没有缺失项的正确的顺序应该是[1, 2, 3, 4, 5, 6, 7, 8]，而我们现在却是[4,3,2,7,8,2,3,1]，我们需要把数字移动到正确的位置上去，比如第一个4就应该和7先交换个位置，以此类推，最后得到的顺序应该是[1, 2, 3, 4, 3, 2, 7, 8]，我们最后在对应位置检验，如果nums[i]和i+1不等，那么我们将i+1存入结果res中即可，参见代码如下：

class Solution {

public:

vector<int> findDisappearedNumbers(vector<int>& nums) {

vector<int> res;

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

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

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

--i;

}

}

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

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

res.push\_back(i + 1);

}

}

return res;

}

};

**Solution 3: Efficient way to find number over boundary after +n (30 min)**

class Solution {

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

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

int n = nums.length;

// e.g nums = [4,3,2,7,8,2,3,1]

// -> nums = [12,19,18,15,8,2,11,9]

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

nums[(nums[i] - 1) % n] += n;

}

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

if(nums[i] <= n) {

result.add(i + 1);

}

}

return result;

}

}

**Refer to**

<https://grandyang.com/leetcode/448/>

下面这种方法是在nums[nums[i]-1]位置累加数组长度n，注意nums[i]-1有可能越界，所以我们需要对n取余，最后要找出缺失的数只需要看nums[i]的值是否小于等于n即可，最后遍历完nums[i]数组为[12, 19, 18, 15, 8, 2, 11, 9]，我们发现有两个数字8和2小于等于n，那么就可以通过i+1来得到正确的结果5和6了，参见代码如下：

class Solution {

public:

vector<int> findDisappearedNumbers(vector<int>& nums) {

vector<int> res;

int n = nums.size();

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

nums[(nums[i] - 1) % n] += n;

}

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

if (nums[i] <= n) {

res.push\_back(i + 1);

}

}

return res;

}

};