<https://leetcode.com/problems/set-mismatch/description/>

You have a set of integers s, which originally contains all the numbers from 1 to n. Unfortunately, due to some error, one of the numbers in s got duplicated to another number in the set, which results in **repetition of one** number and **loss of another** number.

You are given an integer array nums representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return *them in the form of an array*.

**Example 1:**

Input: nums = [1,2,2,4]

Output: [2,3]

**Example 2:**

Input: nums = [1,1]

Output: [1,2]

**Constraints:**

* 2 <= nums.length <= 104
* 1 <= nums[i] <= 104

**Attempt 1: 2023-09-15**

**Solution 1: Efficient way to flip original element to negative  (360 min)**

class Solution {

public int[] findErrorNums(int[] nums) {

int[] result = new int[2];

int n = nums.length;

// 遍历每个数字，然后将其应该出现的位置上的数字变为其相反数，这样

// 如果我们再变为其相反数之前已经成负数了，说明该数字是重复数

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

// 举例 1：nums = [1,2,2,4]

// 第一个数1应该出现的位置index是Math.abs(nums[0]) - 1 = 1 - 1 = 0

// 第二个数2应该出现的位置index是Math.abs(nums[1]) - 1 = 2 - 1 = 1

// 第三个数2应该出现的位置index是Math.abs(nums[2]) - 1 = 2 - 1 = 1

// 第四个数4应该出现的位置index是Math.abs(nums[3]) - 1 = 4 - 1 = 3

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 举例 2：nums = [2,3,2]

// 第一个数2应该出现的位置index是Math.abs(nums[0]) - 1 = 2 - 1 = 1

// 第二个数3应该出现的位置index是Math.abs(nums[1]) - 1 = 3 - 1 = 2

// 第三个数2应该出现的位置index是Math.abs(nums[2]) - 1 = 2 - 1 = 1

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

// 举例 1：nums = [1,2,2,4]

// 第一个数1应该出现的位置index = 0，将其应该出现的位置上的

// 数字nums[index] = nums[0] = 1变为其相反数-1成为负数，

// nums = [-1,2,2,4]

// 第二个数2应该出现的位置index = 1，将其应该出现的位置上的

// 数字nums[index] = nums[1] = 2变为其相反数-2成为负数,

// nums = [-1,-2,2,4]

// 第三个数2应该出现的位置index = 1，此时遇到了该位置上的数字

// 2在将其变为其相反数-2之前已经成负数-2了（来自前一轮），说明

// 该数字2是重复数，注意此时i = 2的nums[i] = nums[2]已经被前

// 一轮翻转为-2了，需要用Math.abs(nums[i])取得原来的正值2，

// 另一方面，此时的index = Math.abs(nums[2]) - 1 = 1，而

// Math.abs(nums[index]) = Math.abs(nums[1])也等于2，在该

// 例子输入为nums = [1,2,2,4]看不出问题，可以混用，但输入换成

// nums = [2,3,2]时就会发现重复数必须用Math.abs(nums[i])获取，

// 而非Math.abs(nums[index])，意义上可以理解为i代表重复数在原

// 数组中的真实位置，而index代表重复数在原数组中的理论位置，理论

// 上原数组中如果没有重复数的情况下，一个数对应一个位置，也就是说

// 一个i或者说nums[i]对应一个理论位置index，不应该出现两个或者

// 多个i或者说nums[i]对应一个理论位置index的情况，理论位置index

// 应该具备唯一性，但凡出现多个i或者说nums[i]对应一个理论位置index

// 的情况意味着i或者说nums[i]重复了之前某个位置为j（j != i）的数

// 的值，所以Math.abs(nums[i])就是重复的数，将Math.abs(nums[i])

// 加入res[0]中即可

// 第四个数4应该出现的位置index = 3，将其应该出现的位置上的

// 数字nums[index] = nums[3] = 4变为其相反数-4成为负数,

// nums = [-1,-2,2,-4]

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 举例 2：nums = [2,3,2]

// 第一个数2应该出现的位置index = 1，将其应该出现的位置上的

// 数字nums[index] = nums[1] = 3变为其相反数-3成为负数，

// nums = [2,-3,2]

// 第二个数3应该出现的位置index = 2，将其应该出现的位置上的

// 数字nums[index] = nums[2] = 2变为其相反数-2成为负数,

// nums = [2,-3,-2]

// 第三个数2应该出现的位置index = 1，此时遇到了该位置上的数字

// 3在将其变为其相反数-3之前已经成负数-3了（来自前一轮），说明

// 该数字2是重复数，注意此时i = 2的nums[i] = nums[2]已经被前

// 一轮翻转为-2了，需要用Math.abs(nums[i])取得原来的正值2，

// 另一方面，此时的index = Math.abs(nums[2]) - 1 = 1，而

// Math.abs(nums[index]) = Math.abs(nums[1])等于3，而此时

// Math.abs(nums[i]) = Math.abs(nums[2])等于2，这与举例 1就

// 不同了，Math.abs(nums[index])和Math.abs(nums[i])只有一个

// 能代表重复数，根据下面的理论，当然Math.abs(nums[i])才是重复

// 数的代表，意义上可以理解为i代表重复数在原数组中的真实位置，

// 而index代表重复数在原数组中的理论位置，理论上原数组中如果没有

// 重复数的情况下，一个数对应一个位置，也就是说一个i或者说nums[i]

// 对应一个理论位置index，不应该出现两个或者多个i或者说nums[i]

// 对应一个理论位置index的情况，理论位置index应该具备唯一性，

// 但凡出现多个i或者说nums[i]对应一个理论位置index的情况意味着

// i或者说nums[i]重复了之前某个位置为j（j != i）的数的值，所以

// Math.abs(nums[i])就是重复的数，将Math.abs(nums[i])加入

// res[0]中即可

if(nums[index] < 0) {

//result[0] = Math.abs(nums[index]); --> Wrong !!

result[0] = Math.abs(nums[i]); // Same as result[0] = (index + 1)

} else {

nums[index] = -nums[index];

}

}

// 然后再遍历原数组，如果某个位置上的数字为正数，说明该位置对应的数字

// 没有出现过，加入res[1]中即可

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

if(nums[i] > 0) {

result[1] = i + 1;

}

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/set-mismatch/editorial/>

### **Approach 6: Using Constant Space**

#### **Algorithm**

We can save the space used in the last approach, if we can somehow, include the information regarding the duplicity of an element or absence of an element in the nums array. Let's see how this can be done.

We know that all the elements in the given nums array are positive, and lie in the range 111to n only. Thus, we can pick up each element i from nums. For every number i picked up, we can invert the element at the index ∣i∣. By doing so, if one of the elements j occurs twice, when this number is encountered the second time, the element nums[∣i∣] will be found to be negative.

Thus, while doing the inversions, we can check if a number found is already negative, to find the duplicate number.

After the inversions have been done, if all the elements in nums are present correctly, the resultant nums array will have all the elements as negative now. But, if one of the numbers, j is missing, the element at the jth index will be positive. This can be used to determine the missing number.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

int dup = -1, missing = 1;

for (int n: nums) {

if (nums[Math.abs(n) - 1] < 0)

dup = Math.abs(n);

else

nums[Math.abs(n) - 1] \*= -1;

}

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

if (nums[i] > 0)

missing = i + 1;

}

return new int[]{dup, missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(n). Two traversals over the nums array of size n are done.
* Space complexity : O(1). Constant extra space is used.

**Refer to**

<https://grandyang.com/leetcode/645/>

**我们来看一种更省空间的解法，这种解法思路相当巧妙，遍历每个数字，然后将其应该出现的位置上的数字变为其相反数，这样如果我们再变为其相反数之前已经成负数了，说明该数字是重复数，将其将入结果res中，然后再遍历原数组，如果某个位置上的数字为正数，说明该位置对应的数字没有出现过，加入res中即可，参见代码如下：**

class Solution {

public:

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

vector<int> res(2, -1);

for (int i : nums) {

if (nums[abs(i) - 1] < 0) res[0] = abs(i);

else nums[abs(i) - 1] \*= -1;

}

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

if (nums[i] > 0) res[1] = i + 1;

}

return res;

}

};

**Solution 2: Brute Force (10 min)**

class Solution {

public int[] findErrorNums(int[] nums) {

int n = nums.length;

int[] result = new int[2];

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

int count = 0;

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

if(nums[j] == i) {

count++;

}

}

if(count == 2) {

result[0] = i;

}

if(count == 0) {

result[1] = i;

}

}

return result;

}

}

Time complexity : O(n^2). We traverse over the nums array of size n for each of the numbers from 1 to n.

Space complexity : O(1). Constant extra space is used.

**Refer to**

<https://leetcode.com/problems/set-mismatch/>

### **Approach 1: Brute Force**

The most naive solution is to consider each number from 111to n, and traverse over the whole nums array to check if the current number occurs twice in nums or doesn't occur at all. We need to set the duplicate number, dup and the missing number, missing, appropriately in such cases respectively.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

int dup = -1, missing = -1;

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

int count = 0;

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

if (nums[j] == i)

count++;

}

if (count == 2)

dup = i;

else if (count == 0)

missing = i;

}

return new int[] {dup, missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(n^2). We traverse over the nums array of size n for each of the numbers from 1 to n.
* Space complexity : O(1). Constant extra space is used.

**Solution 3: Better Brute Force (10 min)**

class Solution {

public int[] findErrorNums(int[] nums) {

int n = nums.length;

int[] result = new int[2];

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

int count = 0;

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

if(nums[j] == i) {

count++;

}

}

if(count == 2) {

result[0] = i;

}

if(count == 0) {

result[1] = i;

}

if(result[0] > 0 && result[1] > 0) {

break;

}

}

return result;

}

}

Time complexity : O(n^2). We traverse over the nums array of size n for each of the numbers from 1 to n, in the worst case.

Space complexity : O(1). Constant extra space is used.

**Refer to**

<https://leetcode.com/problems/set-mismatch/>

### **Approach 2: Better Brute Force**

In the last approach, we continued the search process, even when we've already found the duplicate and the missing number. But, as per the problem statement, we know that only one number will be repeated and only one number will be missing. Thus, we can optimize the last approach to some extent, by stopping the search process as soon as we find these two required numbers.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

int dup = -1, missing = -1;;

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

int count = 0;

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

if (nums[j] == i)

count++;

}

if (count == 2)

dup = i;

else if (count == 0)

missing = i;

if (dup > 0 && missing > 0)

break;

}

return new int[] {dup, missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(n^2). We traverse over the nums array of size n for each of the numbers from 1 to n, in the worst case.
* Space complexity : O(1). Constant extra space is used.

**Solution 4: Sorting (60 min, difficult on logic of getting the missing one -> result[1])**

Consider corner case e.g [2,2] in the begining

class Solution {

public int[] findErrorNums(int[] nums) {

Arrays.sort(nums);

int[] result = new int[2];

// We can consider this corner case at beginning

// e.g nums = [2,2]

// for i = 1, nums[1](=2) < nums[0] + 1(=3)

// then the missing one not nums[i - 1] + 1 or n = 2, but 1

result[1] = 1;

int n = nums.length;

// The gap between nums[i - 1] and nums[i] means

// the missing element exists

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

// for i = 3, nums[3](=4) > nums[2] + 1(=3)

// then the missing one is (nums[2] + 1 as 3)

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

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

result[0] = nums[i];

} else if(nums[i] > nums[i - 1] + 1) {

result[1] = nums[i - 1] + 1;

}

}

// e.g nums = [1,1]

// for i = 1, nums[1](=1) < nums[0] + 1(=2)

// then the missing one not nums[i - 1] + 1 but n = 2

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

result[1] = n;

}

return result;

}

}

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

Consider corner case e.g [2,2] in the end

class Solution {

public int[] findErrorNums(int[] nums) {

Arrays.sort(nums);

int[] result = new int[2];

// We can consider this corner case here all put in the end

// e.g nums = [2,2]

// for i = 1, nums[1](=2) < nums[0] + 1(=3)

// then the missing one not nums[i - 1] + 1 or n = 2, but 1

//result[1] = 1;

int n = nums.length;

// The gap between nums[i - 1] and nums[i] means

// the missing element exists

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

// for i = 3, nums[3](=4) > nums[2] + 1(=3)

// then the missing one is (nums[2] + 1 as 3)

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

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

result[0] = nums[i];

} else if(nums[i] > nums[i - 1] + 1) {

result[1] = nums[i - 1] + 1;

}

}

// e.g nums = [1,1]

// for i = 1, nums[1](=1) < nums[0] + 1(=2)

// then the missing one not nums[i - 1] + 1 but n = 2

// e.g nums = [2,2]

// for i = 1, nums[1](=2) < nums[0] + 1(=3)

// then the missing one not nums[i - 1] + 1 or n = 2, but 1

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

result[1] = n;

} else if(nums[0] != 1) {

result[1] = 1;

}

return result;

}

}

Time complexity : O(nlog⁡n). Sorting takes O(nlog⁡n) time.

Space complexity : O(log⁡n). Sorting takes O(log⁡n)space.

**Refer to**

<https://leetcode.com/problems/set-mismatch/editorial/>

### **Approach 3: Using Sorting**

#### **Algorithm**

One way to further optimize the last approach is to sort the given nums array. This way, the numbers which are equal will always lie together.

Further, we can easily identify the missing number by checking if every two consecutive elements in the sorted nums array are just one count apart or not.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

Arrays.sort(nums);

int dup = -1, missing = 1;

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

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

dup = nums[i];

else if (nums[i] > nums[i - 1] + 1)

missing = nums[i - 1] + 1;

}

return new int[] {dup, nums[nums.length - 1] != nums.length ? nums.length : missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(nlog⁡n). Sorting takes O(nlog⁡n) time.
* Space complexity : O(log⁡n). Sorting takes O(log⁡n)space.

**Solution 5: Harsh Table (10 min)**

class Solution {

public int[] findErrorNums(int[] nums) {

int[] result = new int[2];

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

int n = nums.length;

// 在freq map的key set中预制好1到n很关键，这样就可以基于缺失数对应的频次

// 保持为0没有变化的原理，轻易找出缺失的那个数

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

freq.put(i, 0);

}

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

freq.put(nums[i], freq.get(nums[i]) + 1);

}

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

if(entry.getValue() == 2) {

result[0] = entry.getKey();

}

if(entry.getValue() == 0) {

result[1] = entry.getKey();

}

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/set-mismatch/editorial/>

### **Approach 4: Using Map**

#### **Algorithm**

The given problem can also be solved easily if we can somehow keep a track of the number of times each element of the nums array occurs. One way to do so is to make an entry for each element of nums in a HashMap map. This map stores the entries in the form (num i, count i). Here, num refers to the ith element in nums and count i refers to the number of times this element occurs in nums. Whenever, the same element occurs again, we can increment the count corresponding to the same.

After this, we can consider every number from 1 to n, and check for its presence in map. If it isn't present, we can update the missing variable appropriately. But, if the count corresponding to the current number is 2, we can update the dup variable with the current number.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

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

int dup = -1, missing = 1;

for (int n: nums) {

map.put(n, map.getOrDefault(n, 0) + 1);

}

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

if (map.containsKey(i)) {

if (map.get(i) == 2)

dup = i;

} else

missing = i;

}

return new int[]{dup, missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(n). Traversing over nums of size n takes O(n) time. Considering each number from 1 to n also takes O(n)time.
* Space complexity : O(n). map can contain at most n entries for each of the numbers from 1 to n.

**Solution 6: Frequency array (10 min)**

class Solution {

public int[] findErrorNums(int[] nums) {

int n = nums.length;

int[] result = new int[2];

int[] freq = new int[n + 1];

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

freq[nums[i]]++;

}

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

if(freq[i] == 2) {

result[0] = i;

}

if(freq[i] == 0) {

result[1] = i;

}

// Early terminate as we found what we want

if(result[0] > 0 && result[1] > 0) {

break;

}

}

return result;

}

}

Time complexity : O(n). Traversing over nums of size n takes O(n) time. Considering each number from 1 to n also takes O(n) time.

Space complexity : O(n). arr can contain at most n elements for each of the numbers from 1 to n.

**Refer to**

<https://leetcode.com/problems/set-mismatch/editorial/>

### **Approach 5: Using Extra Array**

#### **Algorithm**

In the last approach, we make use of a map to store the elements of nums along with their corresponding counts. But, we can note, that each entry in map requires two entries. Thus, putting up n entries requires 2n2n2*n*space actually. We can reduce this space required to n by making use of an array, arr instead.

Now, the indices of arr can be used instead of storing the elements again. Thus, we make use of arr in such a way that, arr[i] is used to store the number of occurrences of the element i+1. The rest of the process remains the same as in the last approach.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

int[] arr = new int[nums.length + 1];

int dup = -1, missing = 1;

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

arr[nums[i]] += 1;

}

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

if (arr[i] == 0)

missing = i;

else if (arr[i] == 2)

dup = i;

}

return new int[]{dup, missing};

}

}

#### **Complexity Analysis**

* Time complexity : O(n). Traversing over nums of size n takes O(n) time. Considering each number from 1 to n also takes O(n) time.
* Space complexity : O(n). arr can contain at most n elements for each of the numbers from 1 to n.

**Solution 7: XOR (??? min)**

**Refer to**

<https://leetcode.com/problems/set-mismatch/editorial/>

### **Approach 7: Using XOR**

#### **Algorithm**

Before we dive into the solution to this problem, let's consider a simple problem. Consider an array with n−1 elements containing numbers from 1 to n with one number missing out of them. Now, how do we find out this missing element. One of the solutions is to take the XOR of all the elements of this array with all the numbers from 1 to n. By doing so, we get the required missing number. This works because XORing a number with itself results in a 0 result. Thus, only the number which is missing can't get cancelled with this XORing.

Now, using this idea as the base, let's take it a step forward and use it for the current problem. By taking the XOR of all the elements of the given nums array with all the numbers from 1 to n, we will get a result, xor, as x⊕y. Here, x and y refer to the repeated and the missing term in the given nums array. This happens on the same grounds as in the first problem discussed above.

Now, in the resultant xor, we'll get a 1 in the binary representation only at those bit positions which have a 1 in one out of the numbers x and y, and a 0 at the same bit position in the other one. In the current solution, we consider the rightmost bit which is 1 in the xor, although any bit would work. Let's say, this position is called the rightmostbit.

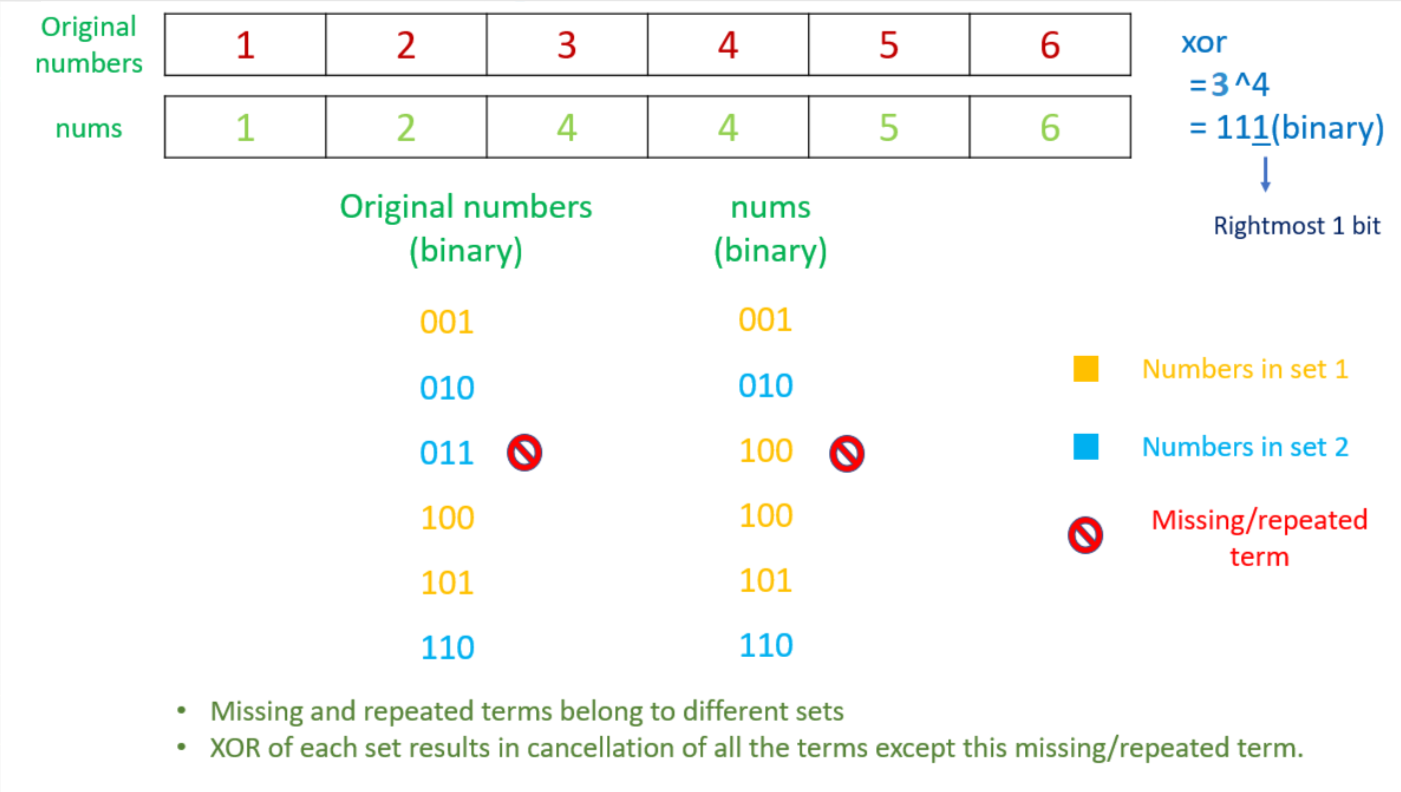
If we divide the elements of the given nums array into two parts such that the first set contains the elements which have a 1 at the rightmostbit position and the second set contains the elements having a 0 at the same position, we'll get one out of x or y in one set and the other one in the second set. Now, our problem has reduced somewhat to the simple problem discussed above.

To solve this reduced problem, we can find out the elements from 1 to n and consider them as a part of the previous sets only, with the allocation of the set depending on a 1 or 0 at the righmostbit position.

Now, if we do the XOR of all the elements of the first set, all the elements will result in an XOR of 0, due to cancellation of the similar terms in both nums and the numbers (1:n)(1:n)(1:*n*), except one term, which is either x or y.

For the other term, we can do the XOR of all the elements in the second set as well.

Consider the example [1 2 4 4 5 6]



One more traversal over the nums can be used to identify the missing and the repeated number out of the two numbers found.

#### **Implementation**

public class Solution {

public int[] findErrorNums(int[] nums) {

int xor = 0, xor0 = 0, xor1 = 0;

for (int n: nums)

xor ^= n;

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

xor ^= i;

int rightmostbit = xor & ~(xor - 1);

for (int n: nums) {

if ((n & rightmostbit) != 0)

xor1 ^= n;

else

xor0 ^= n;

}

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

if ((i & rightmostbit) != 0)

xor1 ^= i;

else

xor0 ^= i;

}

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

if (nums[i] == xor0)

return new int[]{xor0, xor1};

}

return new int[]{xor1, xor0};

}

}

#### **Complexity Analysis**

* Time complexity : O(n). We iterate over n elements five times.
* Space complexity : O(1). Constant extra space is used.