算法基础作业 2

LYZS 数据科学与大数据技术

1 Leetcode 题目

伪代码部分选择 $a \leftarrow b, a = b$ 表示将 b 的值赋值给 a, ▷ 表示注释

1.1 第二次作业

1.1.1 缺失的第一个正数

给你一个未排序的整数数组 $nums$ ，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 $O(n)$ 并且只使用常数级别额外空间的解决方案。

示例 1:

输入: $nums = [1, 2, 0]$

输出: 3

解释: 范围 [1,2] 中的数字都在数组中。

示例 2:

输入: $nums = [3, 4, -1, 1]$

输出: 2

解释: 1 在数组中，但 2 没有。

示例 3:

输入: $nums = [7, 8, 9, 11, 12]$

输出: 1

解释: 最小的正数 1 没有出现。

Algorithm 1 缺失的第一个正数

输入: 一个整数数组 $nums$

输出: 其中没有出现的最小的正整数

```
n ← length of nums                                                    ▷
for i from 0 to n − 1 do
    while nums[i] ≤ 0 or nums[i] > n do
        nums[i] ← n + 1
    end while
end for                                                               ▷
for i from 0 to n − 1 do
    while 1 ≤ nums[i] ≤ n and nums[i] ≠ nums[nums[i] − 1] do
        swap(nums[i], nums[nums[i] − 1])        ▷ 用交换将数组中的每个正整数放到它该在的位置
    end while
end for
for i from 0 to n − 1 do
    if nums[i] ≠ i + 1 then
        return i + 1
    end if
end for
return n + 1
```

首先遍历数组 $nums$，将所有非正整数替换为 $n+1$，如果 $nums[i]$ 在 1 到 n 范围内，并且 $nums[i]$ 不等于 $nums[nums[i]-1]$（即 nums[i] 不在正确的位置上），则交换 $nums[i]$ 和 $nums[nums[i]-1]$，尽量让每个索引 i 都对应 $i+1$ 的值，

遍历数组 $nums$，找出第一个 $nums[i]$ 不等于 $i+1$ 的索引 i。这意味着在索引 i 处的数字不是 $i+1$，因此 $i+1$ 是缺失的第一个正整数

如果所有的 $nums[i]$ 都等于 $i+1$，则说明数组中已经包含了从 1 到 n 的所有正整数，因此返回 $n+1$，

期望时间复杂度：$O(n)$

最好情况：如果数组已经部分排序，那么算法的时间复杂度接近 $O(n)$, 因为每个元素至多被置换一次。

最坏情况：如果数组完全未排序，且包含从 1 到 n 的所有整数，那么每个元素可能都需要移动到它的最终位置，这会导致时间复杂度接近 $O(n^2)$。

1.1.2 有序数组的单一元素

给你一个仅由整数组成的有序数组，其中每个元素都会出现两次，唯有一个数只会出现一次。

请你找出并返回只出现一次的那个数。

你设计的解决方案必须满足 $O(logn)$ 时间复杂度和 $O(1)$ 空间复杂度。

示例 1:

输入: $nums = [1,1,2,3,3,4,4,8,8]$

输出: 2

示例 2:

输入: $nums = [3,3,7,7,10,11,11]$

输出: 10

Algorithm 2 有序数组的单一元素

输入: 一个整数有序数组 $nums$

输出: 只出现一次的整数

```
low ← 0
high ← ((length of nums) - 1)
while low < high do
    mid ← ⌊(low+high)/2⌋                                  ▷ 考虑二分查找，向下取整
    if mid mod 2 = 0  and  nums[mid] = nums[mid + 1] then
        low ← mid + 2                                     ▷ 非重复元素在 mid 的右侧
    else if mid mod 2 = 1  and  nums[mid] = nums[mid − 1] then
        low ← mid + 1                                     ▷ 非重复元素在 mid 的右侧
    else
        high ← mid                                        ▷ 非重复元素在 mid 的左侧
    end if
end while
return nums[low]
```

时间复杂度：$O(logn)$

其中 n 是数组的长度。这是因为算法通过二分查找减少了搜索范围，每次迭代都将搜索区间减半。这里的数组从 $nums[0]$ 开始。

1.1.3 下一个更大元素 II

给定一个循环数组 $nums$ $nums[nums.length-1]$ 的下一个元素是 $nums[0]$），返回 $nums$ 中每个元素的下一个更大元素。

数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1 。

示例 1:

输入: $nums = [1, 2, 1]$

输出: $[2, -1, 2]$

解释: 第一个 1 的下一个更大的数是 2；数字 2 找不到下一个更大的数；第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

示例 2:

输入: $nums = [1, 2, 3, 4, 3]$

输出: $[2, 3, 4, -1, 4]$

Algorithm 3 下一个更大元素 II

输入: 一个循环数组 $nums$

输出: 一个存放 $nums$ 中每个元素的下一个更大元素的数组

```
n ← length of nums
stack ← empty stack                                   ▷ 建立一个空栈
res ← array of −1s with length n
for i in 1 to 2n do                                   ▷ 循环遍历两次数组
    index ← (i mod n)                                 ▷ 对齐索引
    while stack is not empty and nums[index] > nums[top(stack)] do
        temp ← pop(stack)
        res[temp] ← nums[index]
    end while
    if i < n then
        push(stack, i)                                ▷ 在第一次遍历时将索引入栈
    end if
end for
return res
```

时间复杂度：$O(2n) = O(n)$，其中 n 是数组的长度, 循环遍历了两次数组.

1.1.4 最大异或值

给你一个整数数组 $nums$ ，返回 $nums[i]XORnums[j]$ 的最大运算结果，其中 $0\ i\ j < n$ 。

示例 1：

输入：$nums = [3, 10, 5, 25, 2, 8]$

输出：28

解释：最大运算结果是 $5XOR25 = 28$

示例 2：

输入：$nums = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]$

输出：127

Algorithm 4 最大异或值

输入: 一个整数数组 $nums$

输出: $nums[i]XORnums[j]$ 的最大运算结果

```
max_xor ← 0
for i in 1 to n do
    for j in i to n do
        current_xor ← nums[i] ⊕ nums[j]
        max_xor ← max(max_xor, current_xor)
    end for
end for
return max_xor
```

采用暴力枚举的方法，对于每一对元素 $(nums[i], nums[j])$，计算它们的异或值, 更新 max_xor 为当前计算的异或值和已有最大值中较大的那个。

时间复杂度：$O(n^2)$ 其中 n 是数组 $nums$ 的长度, 该算法需要两层嵌套循环来遍历数组中所有可能的数对。

显然时间复杂度过高不能成功提交,

首先考虑利用数的二进制表示，从高位到低位构建一棵树（因为只有 0 和 1 两个值，所以是一棵二叉树），每个从根节点到叶子节点的路径都表示一个数, 即为前缀树 Trie。

然后遍历数组中的数字，将每一个二进制位，在对应的层中找到一个异或的最大值，也就是：如果是 1，找 0 的那条路径，如果是 0，找 1 的那条路径。这样搜索下来的路径就是这个数字和整个数组异或的最大值。

Algorithm 5 最大异或值

输入: 一个整数数组 $nums$

输出: $nums[i] XOR nums[j]$ 的最大运算结果

```
max_xor ← 0
trie ← new TrieNode()                                  ▷ 创建 Trie 树的根节点
for each num in nums do                                ▷ 第一次遍历数组，构建 Trie 树
    node ← trie                                        ▷ node 为 Trie 树中的当前节点
    for i from 31 down to 0 do                         ▷ 由题目中 0 <= nums[i] <= 2^(31) − 1
        bit ← the i-th bit of num
        if node.bit is None then
            node.bit ← new TrieNode()
        end if
        node ← node.bit
    end for
end for
for each num in nums do                                ▷ 第二次遍历数组，寻找最大异或值
    node ← trie
    current ← 0
    for i from 31 down to 0 do
        bit ← the i-th bit of num                      ▷ bit 为当前数字的第 i 位二进制值
        toggled_bit ← 1 − bit                          ▷ toggled_bit 为 bit 的相反值
        if node.toggled_bit is not None then
            current ← current ≫ 1|1                    ▷ 将 current 右移一位，并在最低位设置 1
            node ← node.toggled_bit
        else
            node ← node.bit                            ▷ 如果相反位的子节点不存在，移动到当前位的子节点
        end if
    end for
    max_xor ← max(max_xor, current)
end for
return max_xor
```

时间复杂度：O(n m)，

其中 n 是数组 $nums$ 的长度，m 是数字的位数（通常是 32 位）。每个数字插入 $Trie$ 树需要 $O(m)$ 的时间，查找最大异或值也需要 $O(m)$ 的时间