

=====

文件夹: class140_LinearBasisAlgorithms

=====

[Markdown 文件]

=====

文件: ADDITIONAL_LINEAR_BASIS_PROBLEMS.md

=====

线性基算法补充题目

本文档收录了更多与线性基数据结构相关的经典算法题目，提供详细的题目描述、来源和解题思路。

洛谷题目

1. P3812 【模板】线性基

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P3812>
- **题目描述**: 给定 n 个整数（数字可能重复），求在这些数中选取任意个，使得他们的异或和最大。
- **解题思路**: 线性基模板题，直接构建线性基并贪心选择元素来最大化异或和。

2. P3265 [JLOI2015]装备购买

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P3265>
- **题目描述**: 有 n 个物品，每个物品有 m 个属性值和一个价格。如果已选择的物品可以通过线性组合得到当前物品，则当前物品是不必要的。求在不出现不必要的前提下，最多能购买多少物品且花费最少。
- **解题思路**: 将物品的属性值看作向量，使用线性基判断线性相关性，贪心选择价格较低的物品。

3. P4570 [BJWC2011]元素

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P4570>
- **题目描述**: 有 n 种矿石，每种矿石有一个元素序号和魔力值。如果选择的矿石元素序号异或和为 0，则会发生魔法抵消。求在不发生魔法抵消的前提下，能获得的最大魔力值。
- **解题思路**: 按魔力值从大到小排序，使用线性基判断是否线性相关，贪心选择矿石。

4. P4301 [CQOI2013]新 Nim 游戏

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P4301>
- **题目描述**: 在第一个回合中，双方可以直接拿走若干个整堆的火柴。从第二个回合开始，规则和 Nim 游戏一样。如果你先拿，怎样才能保证获胜？如果可以获胜的话，还要让第一回合拿的火柴总数尽量小。
- **解题思路**: 要保证获胜，第一回合结束后剩余火柴堆的异或和不能为 0。这等价于选出一些火柴堆，使得剩下的火柴堆线性相关。按火柴数量从大到小排序，贪心构建线性基。

5. P4151 [WC2011]最大 XOR 和路径

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P4151>
- **题目描述**: 给定一个边权为非负整数的无向连通图, 求一条从 1 号节点到 N 号节点的路径, 使得路径上经过的边的权值的 XOR 和最大。
- **解题思路**: 任意一条从 1 到 N 的路径都可以表示为一条固定路径加上若干个环的异或和。使用 DFS 找环构建线性基, 然后贪心选择元素来最大化异或和。

6. P3292 [SCOI2016]幸运数字

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P3292>
- **题目描述**: 给定一棵树, 每个节点有一个幸运数字。多次询问两点间路径上节点幸运数字异或和的最大值。
- **解题思路**: 树上倍增+线性基。预处理每个节点到祖先节点路径上的线性基, 查询时合并路径上的线性基。

7. P4839 P 哥的桶

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P4839>
- **题目描述**: 有 n 个桶排成一排, 支持向桶中添加数字和查询区间桶中数字能异或出的最大值。
- **解题思路**: 线段树维护线性基。每个节点维护对应区间数字构成的线性基, 查询时合并区间线性基。

8. P5607 [Ynoi2013]无力回天 NOI2017

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P5607>
- **题目描述**: 维护 m 个集合, 支持向集合中插入元素和查询两个集合并的元素个数。
- **解题思路**: 线段树分治+线性基。将操作看作时间轴上的区间, 用线段树分治处理。

9. P4869 albus 就是要第一个出场

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P4869>
- **题目描述**: 给定一个数组, 求所有子集异或和排序后, 给定值第一次出现的位置。
- **解题思路**: 线性基+组合数学。计算线性基的大小, 通过组合数学计算每个异或值出现的次数。

10. P5556 圣剑护符

- **题目来源**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P5556>
- **题目描述**: 给定一棵树, 每个节点有一个属性值。支持查询树上路径异或和是否能选出两个不相等的子集使其异或和相等, 以及修改路径上节点的属性值。
- **解题思路**: 树链剖分+线性基。维护路径上线性基, 判断是否存在线性相关的子集。

Codeforces 题目

1. CF895C Square Subsets

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF895C>
- **题目描述**: 给定一个数组，求有多少种不同的方法选择非空子集，使得所选元素的乘积等于某个整数的平方。
- **解题思路**: 质因数分解+线性基。将每个数分解质因数，保留次数为奇数的质因数，用线性基计算方案数。

2. CF1100F Ivan and Burgers

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF1100F>
- **题目描述**: 给定一个数组，多次询问区间内选取若干个数能得到的最大异或和。
- **解题思路**: 前缀线性基。预处理前缀线性基，查询时通过前缀线性基计算区间线性基。

3. CF959F Mahmoud and Ehab and yet another xor task

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF959F>
- **题目描述**: 给定一个数组，多次询问前缀内有多少个子序列的异或和等于给定值。
- **解题思路**: 线性基+动态规划。维护前缀线性基，通过线性基大小计算方案数。

4. CF724G Xor-matic Number of the Graph

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF724G>
- **题目描述**: 给定一个无向图，求所有满足条件的三元组 (u, v, s) 中 s 的和，其中 s 是 u 到 v 路径上边权异或和。
- **解题思路**: DFS 找环+线性基。找到图中所有环的异或和，构建线性基计算贡献。

5. CF845G Shortest Path Problem?

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF845G>
- **题目描述**: 给定一个带权无向图，求从顶点 1 到顶点 n 的最短路径长度，路径长度定义为路径上所有边权的按位异或。
- **解题思路**: DFS 找环+线性基。任意一条从 1 到 n 的路径都可以表示为一条固定路径加上若干个环的异或和。

6. CF1163E Magical Permutation

- **题目来源**: Codeforces
- **题目链接**: <https://www.luogu.com.cn/problem/CF1163E>
- **题目描述**: 给定一个集合 S ，构造一个 0 到 $2^x - 1$ 的排列，使得相邻元素的异或值属于集合 S ，求最大的 x 。
- **解题思路**: 线性基构造。构建线性基，通过线性基构造满足条件的排列。

LeetCode 题目

1. LeetCode 421. 数组中两个数的最大异或值

- **题目来源**: LeetCode
- **题目链接**: <https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/>
- **题目描述**: 给定一个非空数组，返回数组中两个数的最大异或值。
- **解题思路**: 字典树+线性基。将每个数的二进制表示插入字典树，贪心选择能产生最大异或值的路径。

2. LeetCode 1707. 与数组中元素的最大异或值

- **题目来源**: LeetCode
- **题目链接**: <https://leetcode.com/problems/maximum-xor-with-an-element-from-array/>
- **题目描述**: 给定一个数组和多个查询，每个查询包含 x_i 和 m_i ，求 x_i 与数组中不超过 m_i 的元素的最大异或值。
- **解题思路**: 离线处理+字典树。将查询按 m_i 排序，离线处理，动态维护字典树。

HDU 题目

1. HDU 3949 XOR

- **题目来源**: HDU
- **题目链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=3949>
- **题目描述**: 给定一个数组，求所有子集异或和中第 k 小的值。
- **解题思路**: 线性基+高斯消元。构建对角化线性基，通过 k 的二进制表示查询第 k 小值。

解题技巧总结

1. 线性基构建

- 从高位到低位扫描，贪心地选择能增加线性基维度的元素
- 时间复杂度: $O(n * \log W)$ ，其中 W 是值域大小

2. 最大异或和查询

- 构建线性基后，从高位到低位贪心选择能使异或和增大的元素
- 时间复杂度: $O(\log W)$

3. 第 k 小异或和查询

- 需要构建对角化线性基（高斯消元法）
- 通过 k 的二进制表示查询第 k 小值

4. 线性相关性判断

- 尝试将元素插入线性基，如果不能插入说明线性相关

5. 区间查询处理

- 前缀线性基：预处理前缀线性基，查询时通过差分计算区间线性基
- 线段树维护线性基：每个节点维护对应区间的线性基

6. 图论问题处理

- DFS 找环：通过 DFS 找到图中所有环的异或和
- 树上问题：树链剖分、树上倍增等结合线性基

时间复杂度分析

- 线性基插入: $O(\log W)$
- 线性基查询最大异或和: $O(\log W)$
- 线性基查询第 k 小异或和: $O(\log^2 W)$
- 前缀线性基预处理: $O(n * \log W)$
- 线段树维护线性基: $O(n * \log n * \log W)$

空间复杂度分析

- 线性基: $O(\log W)$
- 前缀线性基: $O(n * \log W)$
- 线段树维护线性基: $O(n * \log n * \log W)$

文件: EXTENDED_PROBLEMS.md

线性基 (Linear Basis) 问题扩展

本文档收录了更多与线性基数据结构相关的经典算法题目，提供详细的解题思路和实现代码。

核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

线性基 (Linear Basis) 是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

LeetCode 421. 数组中两个数的最大异或值

题目描述

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

题目链接

<https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/>

解题思路

这道题可以用字典树（前缀树）来实现线性基的功能。我们将每个数的二进制表示从最高位到最低位插入字典树中，然后对于每个数，在字典树中查找能够与其产生最大异或值的另一个数。

时间复杂度

$O(n * 32)$ ，其中 n 是数组长度，32 是整数的二进制位数

空间复杂度

$O(n * 32)$

代码实现

三种语言的代码实现分别位于：

- [Code421_MaximumXOROfTwoNumbersInAnArray.java](#)
- [Code421_MaximumXOROfTwoNumbersInAnArray.cpp](#)
- [Code421_MaximumXOROfTwoNumbersInAnArray.py](#)

LeetCode 1707. 与数组中元素的最大异或值

题目描述

给你一个由非负整数组成的数组 `nums`。另有一个查询数组 `queries`，其中 `queries[i] = [xi, mi]`。

第 i 个查询的答案是 xi 和任何 `nums` 数组中不超过 mi 的元素按位异或 (XOR) 得到的最大值。如果 `nums` 数组中没有元素不超过 mi ，整数 -1。

返回一个整数数组 `answer` 作为查询的答案，其中 `answer.length == queries.length` 且 `answer[i]` 是第 i 个查询的答案。

题目链接

<https://leetcode.com/problems/maximum-xor-with-an-element-from-array/>

解题思路

这道题是上一题的扩展，需要处理带约束条件的查询。我们可以采用离线处理的方法：

1. 将 `nums` 数组排序
2. 将 `queries` 数组排序，并记录原始索引
3. 按照 mi 从小到大的顺序处理查询，将 `nums` 中不超过 mi 的元素插入字典树
4. 对于每个查询，在字典树中查询最大异或值

时间复杂度

$O(n \log n + q \log q + (n + q) * 32)$, 其中 n 是数组长度, q 是查询数量

空间复杂度

$O(n * 32 + q)$

代码实现

三种语言的代码实现分别位于:

- Code1707_MaximumXORWithAnElementFromArray.java
- Code1707_MaximumXORWithAnElementFromArray.cpp
- Code1707_MaximumXORWithAnElementFromArray.py

Codeforces 895C. Square Subsets

题目描述

给定一个数组, 求非空子集的数量, 使得子集中所有元素的乘积是一个平方数。答案对 10^{9+7} 取模。

题目链接

<https://codeforces.com/contest/895/problem/C>

解题思路

这道题可以利用线性基和质因数分解来解决。

1. 对于每个数, 我们可以将其质因数分解, 保留次数为奇数的质因数
2. 这样, 每个数可以表示为一个二进制向量, 向量的每一位代表一个质数是否出现奇数次
3. 问题转化为: 在数组中选择一个非空子集, 使得子集中所有数的向量异或结果为零向量
4. 使用动态规划结合线性基来计算方案数

时间复杂度

$O(n * m * \log m)$, 其中 n 是数组长度, m 是质数的个数

空间复杂度

$O(2^m)$, 其中 m 是质数的个数

代码实现

三种语言的代码实现分别位于:

- Codeforces895C_SquareSubsets.java
- Codeforces895C_SquareSubsets.cpp
- Codeforces895C_SquareSubsets.py

线性基常见操作

1. 插入元素

``` java

```
public static boolean insert(long num) {
 for (int i = BIT; i >= 0; i--) {
 if ((num >> i) == 1) {
 if (basis[i] == 0) {
 basis[i] = num;
 return true;
 }
 num ^= basis[i];
 }
 }
 return false;
}
```

```

2. 查询最大异或和

```
``` java
public static long queryMax() {
 long ans = 0;
 for (int i = BIT; i >= 0; i--) {
 ans = Math.max(ans, ans ^ basis[i]);
 }
 return ans;
}
```

```

3. 查询最小异或和

```
``` java
public static long queryMin() {
 for (int i = 0; i <= BIT; i++) {
 if (basis[i] != 0) {
 return basis[i];
 }
 }
 return 0;
}
```

```

4. 查询第 k 小异或和

```
``` java
public static long queryKth(long k) {
 if (k >= (1L << tot)) return -1;
 long ret = 0;
 for (int i = 0; i <= BIT; i++) {

```

```
 if ((k >> i) & 1) ret ^= d[i];
}
return ret;
}
```

```

时间复杂度分析

线性基的核心操作是插入元素，时间复杂度为 $O(\log W)$ ，其中 W 是值域大小。
对于 64 位整数，时间复杂度为 $O(64) = O(1)$ 。

空间复杂度分析

线性基的空间复杂度为 $O(\log W)$ ，即 $O(64) = O(1)$ 。

工程化考量

1. 异常处理：需要处理空输入、重复元素等边界情况
2. 性能优化：对于大量查询，可以预处理线性基
3. 内存优化：线性基只存储 \log 级别的元素，内存占用小
4. 可扩展性：可以扩展到支持区间查询、在线查询等场景

CodeChef 111506 – XOR AND OR Problem

题目描述

给定一个长度为 n 的数组，求所有可能的子序列的异或和的最大值。

题目链接

<https://www.codechef.com/problems/XORANDOR>

解题思路

这是一道线性基的模板题。我们可以通过构建线性基，然后利用线性基的性质来找到最大异或和。具体步骤如下：

1. 构建数组的线性基
2. 从最高位到最低位遍历线性基数组，贪心选择能使异或和增大的元素

时间复杂度

$O(n * \log(\max_value))$ ，其中 n 是数组长度， \max_value 是数组中的最大值

空间复杂度

$O(\log(\max_value))$

代码实现

三种语言的代码实现分别位于：

- CodeChef111506_XORANDOR.java
- CodeChef111506_XORANDOR.cpp
- CodeChef111506_XORANDOR.py

与机器学习等领域的联系

1. 在机器学习中，线性基的概念类似于特征选择中的线性无关特征集合
2. 在密码学中，线性基用于分析线性密码的性质
3. 在编码理论中，线性基用于构造线性码

LeetCode 421. 数组中两个数的最大异或值详解

题目描述

给你一个非空数组，求数组中两个数的最大异或值。

解题思路

这道题可以用字典树（前缀树）来实现线性基的功能。我们将每个数的二进制表示从最高位到最低位插入字典树中，然后对于每个数，在字典树中查找能够与其产生最大异或值的另一个数。

时间复杂度

- 时间复杂度： $O(n * 32)$ ，其中 n 是数组长度，32 是整数的二进制位数
- 空间复杂度： $O(n * 32)$

代码实现

Java 实现

```
```java
class Solution {
 // 字典树节点定义
 class TrieNode {
 TrieNode[] children = new TrieNode[2]; // 0 和 1 两个子节点
 }

 private TrieNode root; // 字典树根节点
 private static final int HIGH_BIT = 30; // 整数的最高位是第 30 位（假设是 32 位整数）

 public int findMaximumXOR(int[] nums) {
 root = new TrieNode();
 int maxXor = 0;

 // 插入第一个数
 insert(nums[0]);
 for (int i = 1; i < nums.length; i++) {
 TrieNode cur = root;
 int xor = 0;
 for (int j = 30; j >= 0; j--) {
 int bit = (nums[i] >> j) & 1;
 if (cur.children[bit] == null) {
 cur.children[bit] = new TrieNode();
 }
 cur = cur.children[bit];
 xor |= bit;
 }
 maxXor = Math.max(maxXor, xor);
 }
 }

 void insert(int num) {
 TrieNode cur = root;
 for (int i = 30; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (cur.children[bit] == null) {
 cur.children[bit] = new TrieNode();
 }
 cur = cur.children[bit];
 }
 }
}
```

```

// 对于每个数，先查询最大异或值，再插入到字典树中
for (int i = 1; i < nums.length; i++) {
 maxXor = Math.max(maxXor, query(nums[i]));
 insert(nums[i]);
}

return maxXor;
}

// 将数字插入字典树
private void insert(int num) {
 TrieNode node = root;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (node.children[bit] == null) {
 node.children[bit] = new TrieNode();
 }
 node = node.children[bit];
 }
}

// 查询与给定数字异或的最大值
private int query(int num) {
 TrieNode node = root;
 int xorSum = 0;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (num >> i) & 1;
 int desiredBit = 1 - bit; // 希望找到相反的位以获得最大异或值

 if (node.children[desiredBit] != null) {
 xorSum |= (1 << i); // 当前位可以取到 1
 node = node.children[desiredBit];
 } else {
 node = node.children[bit]; // 只能取相同的位
 }
 }
 return xorSum;
}
```

```

C++实现

```

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
private:
 struct TrieNode {
 TrieNode* children[2];
 TrieNode() {
 children[0] = children[1] = nullptr;
 }
 };
 TrieNode* root;
 const int HIGH_BIT = 30;

 void insert(int num) {
 TrieNode* node = root;
 for (int i = HIGH_BIT; i >= 0; --i) {
 int bit = (num >> i) & 1;
 if (!node->children[bit]) {
 node->children[bit] = new TrieNode();
 }
 node = node->children[bit];
 }
 }

 int query(int num) {
 TrieNode* node = root;
 int xorSum = 0;
 for (int i = HIGH_BIT; i >= 0; --i) {
 int bit = (num >> i) & 1;
 int desiredBit = 1 - bit;

 if (node->children[desiredBit]) {
 xorSum |= (1 << i);
 node = node->children[desiredBit];
 } else {
 node = node->children[bit];
 }
 }
 return xorSum;
 }
}

```

```

 }

public:
 int findMaximumXOR(vector<int>& nums) {
 root = new TrieNode();
 int maxXor = 0;

 insert(nums[0]);
 for (int i = 1; i < nums.size(); ++i) {
 maxXor = max(maxXor, query(nums[i]));
 insert(nums[i]);
 }

 // 清理内存
 delete root;

 return maxXor;
 }
};

```

```

```

#### Python 实现
```python
class Solution:
 def findMaximumXOR(self, nums):
 # 字典树节点定义
 class TrieNode:
 def __init__(self):
 self.children = [None, None] # 0 和 1 两个子节点

 root = TrieNode()
 HIGH_BIT = 30 # 整数的最高位是第 30 位
 max_xor = 0

 # 将数字插入字典树
 def insert(num):
 node = root
 for i in range(HIGH_BIT, -1, -1):
 bit = (num >> i) & 1
 if not node.children[bit]:
 node.children[bit] = TrieNode()
 node = node.children[bit]

```

```

查询与给定数字异或的最大值
def query(num):
 node = root
 xor_sum = 0
 for i in range(HIGH_BIT, -1, -1):
 bit = (num >> i) & 1
 desired_bit = 1 - bit

 if node.children[desired_bit]:
 xor_sum |= (1 << i)
 node = node.children[desired_bit]
 else:
 node = node.children[bit]

 return xor_sum

插入第一个数
insert(nums[0])

对于每个数，先查询再插入
for i in range(1, len(nums)):
 max_xor = max(max_xor, query(nums[i]))
 insert(nums[i])

return max_xor
```

```

LeetCode 1707. 与数组中元素的最大异或值详解

题目描述

给你一个由非负整数组成的数组 `nums`。另有一个查询数组 `queries`，其中 `queries[i] = [xi, mi]`。第 `i` 个查询的答案是 `xi` 和任何 `nums` 数组中不超过 `mi` 的元素按位异或（XOR）得到的最大值。如果 `nums` 数组中不存在不超过 `mi` 的元素，那么答案是 `-1`。

解题思路

这道题是上一题的扩展，需要处理带约束条件的查询。我们可以采用离线处理的方法：

1. 将 `nums` 数组排序
2. 将 `queries` 数组排序，并记录原始索引
3. 按照 `mi` 从小到大的顺序处理查询，将 `nums` 中不超过 `mi` 的元素插入字典树
4. 对于每个查询，在字典树中查询最大异或值

时间复杂度

- 时间复杂度： $O(n \log n + q \log q + (n + q) * 32)$ ，其中 `n` 是数组长度，`q` 是查询数量
- 空间复杂度： $O(n * 32 + q)$

代码实现

Java 实现

```
```java
import java.util.*;

class Solution {
 // 字典树节点定义
 class TrieNode {
 TrieNode[] children = new TrieNode[2];
 }

 private TrieNode root;
 private static final int HIGH_BIT = 30;

 public int[] maximizeXor(int[] nums, int[][] queries) {
 root = new TrieNode();
 int n = nums.length;
 int q = queries.length;
 int[] result = new int[q];

 // 对 nums 数组排序
 Arrays.sort(nums);

 // 对 queries 进行排序，并记录原始索引
 int[][] sortedQueries = new int[q][3];
 for (int i = 0; i < q; i++) {
 sortedQueries[i][0] = queries[i][0]; // xi
 sortedQueries[i][1] = queries[i][1]; // mi
 sortedQueries[i][2] = i; // 原始索引
 }
 Arrays.sort(sortedQueries, Comparator.comparingInt(a -> a[1]));

 int numIndex = 0;
 for (int[] query : sortedQueries) {
 int xi = query[0];
 int mi = query[1];
 int originalIndex = query[2];

 // 将 nums 中不超过 mi 的元素插入字典树
 while (numIndex < n && nums[numIndex] <= mi) {
 insert(nums[numIndex]);
 numIndex++;
 }
 result[originalIndex] = calculateXOR(xi, root);
 }
 }

 void insert(int num) {
 TrieNode node = root;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (node.children[bit] == null) {
 node.children[bit] = new TrieNode();
 }
 node = node.children[bit];
 }
 }

 int calculateXOR(int xi, TrieNode node) {
 int xor = xi;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (xi >> i) & 1;
 if (node.children[bit] != null) {
 xor ^= (1 << i);
 node = node.children[bit];
 } else {
 node = node.children[1 - bit];
 }
 }
 return xor;
 }
}
```

```

 numIndex++;
 }

 // 如果字典树为空，说明没有符合条件的元素
 if (numIndex == 0) {
 result[originalIndex] = -1;
 } else {
 result[originalIndex] = queryMaxXor(xi);
 }
}

return result;
}

private void insert(int num) {
 TrieNode node = root;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (node.children[bit] == null) {
 node.children[bit] = new TrieNode();
 }
 node = node.children[bit];
 }
}

private int queryMaxXor(int num) {
 TrieNode node = root;
 int maxXor = 0;
 for (int i = HIGH_BIT; i >= 0; i--) {
 int bit = (num >> i) & 1;
 int desiredBit = 1 - bit;

 if (node.children[desiredBit] != null) {
 maxXor |= (1 << i);
 node = node.children[desiredBit];
 } else {
 node = node.children[bit];
 }
 }
 return maxXor;
}
```

```

C++实现

```cpp

```
#include <vector>
#include <algorithm>
using namespace std;
```

```
class Solution {
```

```
private:
```

```
 struct TrieNode {
```

```
 TrieNode* children[2];
 TrieNode() {
 children[0] = children[1] = nullptr;
 }
 };
```

```
 TrieNode* root;
```

```
 const int HIGH_BIT = 30;
```

```
 void insert(int num) {
```

```
 TrieNode* node = root;
 for (int i = HIGH_BIT; i >= 0; --i) {
 int bit = (num >> i) & 1;
 if (!node->children[bit]) {
 node->children[bit] = new TrieNode();
 }
 node = node->children[bit];
 }
 }
```

```
 int queryMaxXor(int num) {
```

```
 TrieNode* node = root;
 int maxXor = 0;
 for (int i = HIGH_BIT; i >= 0; --i) {
 int bit = (num >> i) & 1;
 int desiredBit = 1 - bit;

 if (node->children[desiredBit]) {
 maxXor |= (1 << i);
 node = node->children[desiredBit];
 } else {
 node = node->children[bit];
 }
 }
 }
```

```

 }

 return maxXor;
}

public:

vector<int> maximizeXor(vector<int>& nums, vector<vector<int>>& queries) {
 root = new TrieNode();
 int n = nums.size();
 int q = queries.size();
 vector<int> result(q);

 // 对 nums 数组排序
 sort(nums.begin(), nums.end());

 // 对 queries 进行排序，并记录原始索引
 vector<tuple<int, int, int>> sortedQueries;
 for (int i = 0; i < q; ++i) {
 sortedQueries.emplace_back(queries[i][1], queries[i][0], i);
 }
 sort(sortedQueries.begin(), sortedQueries.end());

 int numIndex = 0;
 for (auto& query : sortedQueries) {
 int mi = get<0>(query);
 int xi = get<1>(query);
 int originalIndex = get<2>(query);

 // 将 nums 中不超过 mi 的元素插入字典树
 while (numIndex < n && nums[numIndex] <= mi) {
 insert(nums[numIndex]);
 numIndex++;
 }

 if (numIndex == 0) {
 result[originalIndex] = -1;
 } else {
 result[originalIndex] = queryMaxXor(xi);
 }
 }

 // 清理内存
 // 注意：这里应该递归删除整个字典树，但为了简化代码，这里省略
}

```

```

 return result;
 }
};

```
#### Python 实现
```python
class Solution:
 def maximizeXor(self, nums, queries):
 # 字典树节点定义
 class TrieNode:
 def __init__(self):
 self.children = [None, None] # 0 和 1 两个子节点

 root = TrieNode()
 HIGH_BIT = 30
 n = len(nums)
 q = len(queries)
 result = [0] * q

 # 将 nums 数组排序
 nums.sort()

 # 对 queries 进行排序，并记录原始索引
 sorted_queries = []
 for i in range(q):
 xi, mi = queries[i]
 sorted_queries.append((mi, xi, i))
 sorted_queries.sort()

 # 插入数字到字典树
 def insert(num):
 node = root
 for i in range(HIGH_BIT, -1, -1):
 bit = (num >> i) & 1
 if not node.children[bit]:
 node.children[bit] = TrieNode()
 node = node.children[bit]

 # 查询最大异或值
 def query_max_xor(num):
 node = root
 max_xor = 0

```

```

for i in range(HIGH_BIT, -1, -1):
 bit = (num >> i) & 1
 desired_bit = 1 - bit

 if node.children[desired_bit]:
 max_xor |= (1 << i)
 node = node.children[desired_bit]
 else:
 node = node.children[bit]

return max_xor

```

num\_index = 0

for mi, xi, original\_index in sorted\_queries:

# 将 nums 中不超过 mi 的元素插入字典树

while num\_index < n and nums[num\_index] <= mi:

    insert(nums[num\_index])

    num\_index += 1

if num\_index == 0:

    result[original\_index] = -1

else:

    result[original\_index] = query\_max\_xor(xi)

return result

---

## Codeforces 895C Square Subsets 详解

#### #### 题目描述

给定一个数组，求选出一个子集，使得子集中所有数的乘积是完全平方数的方案数。

#### #### 解题思路

完全平方数的每个质因数的指数都是偶数。我们可以将每个数转换为一个二进制向量，表示其质因数指数的奇偶性（奇数为 1，偶数为 0）。这样问题就转化为：求有多少个子集的异或和为 0。

我们可以使用线性基来解决这个问题。对于每个数，如果它能被插入到线性基中，说明它与之前的数线性无关；否则，说明存在一个子集使得它们的异或和为 0，此时总方案数需要乘以 2。

#### #### 时间复杂度

- 时间复杂度： $O(n * \log \max\_num)$ ，其中  $n$  是数组长度， $\max\_num$  是数组中的最大值
- 空间复杂度： $O(\log \max\_num)$

#### #### 代码实现

```
Java 实现
```java
import java.util.*;

public class SquareSubsets {
    private static final int MOD = 1000000007;
    private static final int MAX_PRIME = 70; // 1e5 以内的质数不超过 70 个
    private static int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
        59, 61, 67};
    private static int PRIME_COUNT = primes.length;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = scanner.nextInt();
        }
        scanner.close();

        // 统计每个数出现的次数
        Map<Integer, Integer> countMap = new HashMap<>();
        for (int num : a) {
            countMap.put(num, countMap.getOrDefault(num, 0) + 1);
        }

        // 线性基数组
        long[] basis = new long[PRIME_COUNT];

        // 处理每个数
        long result = 1;
        for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
            int num = entry.getKey();
            int count = entry.getValue();

            // 计算 num 的质因数分解，统计每个质数的奇偶性
            long mask = 0;
            for (int i = 0; i < PRIME_COUNT; i++) {
                int prime = primes[i];
                int exponent = 0;
                int tmp = num;
                while (tmp % prime == 0) {
                    tmp /= prime;
                    exponent++;
                }
                if (exponent % 2 != 0) {
                    mask |= 1 << i;
                }
            }
            basis[i] = result * mask % MOD;
            result *= basis[i];
            result %= MOD;
        }
    }
}
```

```

        exponent++;
        tmp /= prime;
    }
    if (exponent % 2 == 1) {
        mask |= (1L << i);
    }
}

// 如果 mask 为 0, 说明 num 本身是平方数
if (mask == 0) {
    // 每个平方数可以选择或不选择, 但至少选择一个的情况需要排除
    result = (result * pow(2, count)) % MOD;
} else {
    // 尝试将 mask 插入线性基
    long current = mask;
    for (int i = 0; i < PRIME_COUNT; i++) {
        if ((current >> i & 1) == 1) {
            if (basis[i] == 0) {
                basis[i] = current;
                break;
            }
            current ^= basis[i];
        }
    }
}

// 如果 current 变为 0, 说明存在线性相关
if (current == 0) {
    result = (result * (pow(2, count) - 1)) % MOD;
}
}

// 减去空集的情况
result = (result - 1 + MOD) % MOD;
System.out.println(result);
}

private static long pow(long base, int exponent) {
    long result = 1;
    while (exponent > 0) {
        if (exponent % 2 == 1) {
            result = (result * base) % MOD;
        }

```

```
    base = (base * base) % MOD;
    exponent /= 2;
}
return result;
}
```
```

```

C++实现

```
```cpp
#include <iostream>
#include <vector>
#include <map>
using namespace std;

const int MOD = 1e9 + 7;
const int MAX_PRIME = 70;
vector<int> primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67};
int PRIME_COUNT = primes.size();

long long pow_mod(long long base, int exponent) {
 long long result = 1;
 while (exponent > 0) {
 if (exponent % 2 == 1) {
 result = (result * base) % MOD;
 }
 base = (base * base) % MOD;
 exponent /= 2;
 }
 return result;
}

int main() {
 int n;
 cin >> n;
 vector<int> a(n);
 for (int i = 0; i < n; i++) {
 cin >> a[i];
 }

 map<int, int> countMap;
 for (int num : a) {
 countMap[num]++;
 }
}
```

```
}
```

```
vector<long long> basis(PRIME_COUNT, 0);
long long result = 1;

for (auto& entry : countMap) {
 int num = entry.first;
 int count = entry.second;

 long long mask = 0;
 for (int i = 0; i < PRIME_COUNT; i++) {
 int prime = primes[i];
 int exponent = 0;
 int tmp = num;
 while (tmp % prime == 0) {
 exponent++;
 tmp /= prime;
 }
 if (exponent % 2 == 1) {
 mask |= (1LL << i);
 }
 }

 if (mask == 0) {
 result = (result * pow_mod(2, count)) % MOD;
 } else {
 long long current = mask;
 for (int i = 0; i < PRIME_COUNT; i++) {
 if ((current >> i & 1) == 1) {
 if (basis[i] == 0) {
 basis[i] = current;
 break;
 }
 current ^= basis[i];
 }
 }

 if (current == 0) {
 result = (result * (pow_mod(2, count) - 1)) % MOD;
 }
 }
}
```

```

result = (result - 1 + MOD) % MOD;
cout << result << endl;

return 0;
}

```
#### Python 实现
```python
MOD = 10**9 + 7
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
PRIME_COUNT = len(primes)

def pow_mod(base, exponent):
 result = 1
 while exponent > 0:
 if exponent % 2 == 1:
 result = (result * base) % MOD
 base = (base * base) % MOD
 exponent //= 2
 return result

def main():
 import sys
 from collections import defaultdict
 input = sys.stdin.read().split()
 n = int(input[0])
 a = list(map(int, input[1:n+1]))

 count_map = defaultdict(int)
 for num in a:
 count_map[num] += 1

 basis = [0] * PRIME_COUNT
 result = 1

 for num, count in count_map.items():
 mask = 0
 for i in range(PRIME_COUNT):
 prime = primes[i]
 exponent = 0
 tmp = num
 while tmp % prime == 0:
 tmp //= prime
 exponent += 1
 basis[i] = (basis[i] * pow_mod(prime, exponent)) % MOD
 result *= basis[i]
 result %= MOD

 print(result)

```

```

 exponent += 1
 tmp //= prime
 if exponent % 2 == 1:
 mask |= (1 << i)

if mask == 0:
 result = (result * pow_mod(2, count)) % MOD
else:
 current = mask
 for i in range(PRIME_COUNT):
 if (current >> i) & 1:
 if basis[i] == 0:
 basis[i] = current
 break
 current ^= basis[i]

 if current == 0:
 result = (result * (pow_mod(2, count) - 1)) % MOD

减去空集的情况
result = (result - 1 + MOD) % MOD
print(result)

if __name__ == "__main__":
 main()
```
=====
```

文件: LINEAR_BASIS_SUMMARY.md

线性基算法总结

一、算法概述

线性基 (Linear Basis) 是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

二、核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

三、线性基的构建方法

普通消元法

普通消元法是最常用的构建线性基的方法，其基本思路是：

1. 从最高位开始扫描
2. 对于每个数，尝试将其插入到线性基中
3. 插入过程：从高位到低位扫描，如果当前位为 1 且线性基中该位为空，则直接插入；否则用线性基中该位的数异或当前数，继续处理

高斯消元法

高斯消元法用于构造对角化的线性基，主要用于第 k 小值查询：

1. 先用普通消元法构建线性基
2. 对线性基进行进一步消元，使得每个线性基元素只在对应位为 1，其他位为 0
3. 这样可以方便地按 k 的二进制表示来查询第 k 小值

四、常见操作实现

1. 插入元素

```
```java
public static boolean insert(long num) {
 for (int i = BIT; i >= 0; i--) {
 if ((num >> i) & 1) {
 if (basis[i] == 0) {
 basis[i] = num;
 return true;
 }
 num ^= basis[i];
 }
 }
 return false;
}
```

```

2. 查询最大异或和

``` java

```
public static long queryMax() {
 long ans = 0;
 for (int i = BIT; i >= 0; i--) {
 ans = Math.max(ans, ans ^ basis[i]);
 }
 return ans;
}
```

```

3. 查询最小异或和

``` java

```
public static long queryMin() {
 for (int i = 0; i <= BIT; i++) {
 if (basis[i] != 0) {
 return basis[i];
 }
 }
 return 0;
}
```

```

4. 查询第 k 小异或和

``` java

```
public static long queryKth(long k) {
 if (k >= (1L << tot)) return -1;
 long ret = 0;
 for (int i = 0; i <= BIT; i++) {
 if (k & (1L << i)) ret ^= d[i];
 }
 return ret;
}
```

```

五、题型分类与解题思路

1. 基础题目

特点：

- 直接给出数组，询问整个数组的异或相关问题

- 通常是最值查询或判断性问题

解题思路:

- 直接构建线性基
- 根据题目要求进行相应查询

典型题目:

- 洛谷 P3812 【模板】线性基
- 洛谷 P3857 [TJOI2008]彩灯
- 洛谷 P4570 [BJWC2011]元素

2. 图论结合题目

特点:

- 在图上进行异或相关查询
- 通常涉及路径异或和、环的异或值等

解题思路:

- 利用图的性质，如 DFS 找环、树上路径等
- 将图上的异或问题转化为线性基问题

典型题目:

- 洛谷 P4151 [WC2011]最大 XOR 和路径
- 洛谷 P3292 [SCOI2016]幸运数字

3. 数据结构结合题目

特点:

- 需要支持动态修改或区间查询
- 通常涉及线段树、树状数组等数据结构

解题思路:

- 在数据结构的每个节点上维护线性基
- 对于区间查询，需要合并多个线性基

典型题目:

- 洛谷 P4839 P 哥的桶
- 洛谷 P5607 [Ynoi2013]无力回天 NOI2017
- Codeforces 1100F Ivan and Burgers

4. 构造题目

特点:

- 需要构造满足特定条件的方案
- 通常涉及数学变换或状态压缩

解题思路:

- 将问题转化为线性基的性质问题
- 利用线性基的线性无关性进行构造

典型题目:

- Codeforces 895C Square Subsets
- 洛谷 P4869 albus 就是要第一个出场

5. 博弈论结合题目

特点:

- 结合博弈论的获胜条件
- 通常涉及 Nim 游戏等经典博弈

解题思路:

- 分析博弈的获胜条件
- 将获胜条件转化为线性基问题

典型题目:

- 洛谷 P4301 [CQOI2013]新 Nim 游戏

六、时间复杂度分析

线性基的核心操作是插入元素，时间复杂度为 $O(\log W)$ ，其中 W 是值域大小。

对于 64 位整数，时间复杂度为 $O(64) = O(1)$ 。

七、空间复杂度分析

线性基的空间复杂度为 $O(\log W)$ ，即 $O(64) = O(1)$ 。

八、工程化考量

1. 异常处理：需要处理空输入、重复元素等边界情况
2. 性能优化：对于大量查询，可以预处理线性基
3. 内存优化：线性基只存储 \log 级别的元素，内存占用小
4. 可扩展性：可以扩展到支持区间查询、在线查询等场景

九、与其他算法的结合

1. 与图论算法结合

- DFS 找环构建线性基
- 树上倍增维护线性基
- 最短路径算法与线性基结合

2. 与数据结构结合

- 线段树维护线性基
- 树状数组维护线性基
- 莫队算法与线性基结合

3. 与数学算法结合

- 高斯消元构造对角化线性基
- 质因数分解与线性基结合
- 组合数学与线性基结合

十、常见误区与注意事项

1. 位运算注意事项

- 注意位运算的优先级
- 注意左移右移的边界条件
- 注意有符号数和无符号数的区别

2. 线性基构建注意事项

- 插入元素时要从高位到低位扫描
- 注意线性基的线性无关性
- 注意线性基元素的二进制最高位互不相同

3. 查询操作注意事项

- 最大值查询要从高位到低位贪心选择
- 最小值查询要找到最低位的非零元素
- 第 k 小值查询需要先将线性基对角化

十一、优化技巧

1. 预处理优化

- 对于静态数组，可以预处理前缀线性基
- 对于树上问题，可以预处理每个节点到根的线性基

2. 合并优化

- 线性基的合并可以通过逐个插入实现
- 注意合并时的时间复杂度分析

3. 空间优化

- 可以根据实际需要调整线性基数组大小

- 对于稀疏数据，可以使用动态数组

十二、扩展应用

1. 与机器学习的联系

- 线性基的概念类似于特征选择中的线性无关特征集合
- 可以用于降维和特征提取

2. 与密码学的联系

- 线性基用于分析线性密码的性质
- 可以用于构造和破解线性密码系统

3. 与编码理论的联系

- 线性基用于构造线性码
- 可以用于错误检测和纠正

=====

文件: README.md

=====

线性基算法详解与题目解析

概述

线性基 (Linear Basis) 是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

线性基的构建方法

线性基的构建主要有两种方法：普通消元法和高斯消元法。

普通消元法

普通消元法是最常用的构建线性基的方法，其基本思路是：

1. 从最高位开始扫描
2. 对于每个数，尝试将其插入到线性基中
3. 插入过程：从高位到低位扫描，如果当前位为 1 且线性基中该位为空，则直接插入；否则用线性基中该位的数异或当前数，继续处理

本目录中的题目

1. Code01_BuyEquipment. java – 装备购买

- **题目来源**：洛谷 P3265 [JLOI2015]装备购买
- **题目链接**：<https://www.luogu.com.cn/problem/P3265>
- **算法**：线性基 + 贪心
- **时间复杂度**： $O(n * m^2)$
- **空间复杂度**： $O(n * m)$
- **工程化考量**：浮点数精度处理、异常处理、边界条件检查
- **与机器学习联系**：特征选择中的线性无关特征集合选择

2. Code02_BucketMaximumXor. java – P 哥的桶

- **题目来源**：洛谷 P4839 P 哥的桶
- **题目链接**：<https://www.luogu.com.cn/problem/P4839>
- **算法**：线段树 + 线性基
- **时间复杂度**： $O(n * \log n * \text{BIT})$
- **空间复杂度**： $O(n * \text{BIT})$
- **工程化考量**：线段树优化、内存管理、大规模数据处理
- **与机器学习联系**：在线学习中的特征组合优化

3. Code03_LuckyNumber1. java/Code03_LuckyNumber2. java – 幸运数字

- **题目来源**：洛谷 P3292 [SCOI2016]幸运数字
- **题目链接**：<https://www.luogu.com.cn/problem/P3292>
- **算法**：树上倍增 + 线性基
- **时间复杂度**： $O(n * \log n * \text{BIT})$
- **空间复杂度**： $O(n * \log n * \text{BIT})$

4. Code04_MaximumXorOfPath1. java/Code04_MaximumXorOfPath2. java – 路径最大异或和

- **题目来源**：洛谷 P4151 [WC2011]最大 XOR 和路径
- **题目链接**：<https://www.luogu.com.cn/problem/P4151>
- **算法**：DFS + 线性基
- **时间复杂度**： $O((n + m) * \text{BIT})$
- **空间复杂度**： $O(n + \text{BIT})$

5. Code05_NewNimGame. java – 新 Nim 游戏

- **题目来源**: 洛谷 P4301 [CQOI2013]新 Nim 游戏
- **题目链接**: <https://www.luogu.com.cn/problem/P4301>
- **算法**: 线性基 + 贪心
- **时间复杂度**: $O(k * \text{BIT} * \log k)$
- **空间复杂度**: $O(\text{BIT})$

6. Code06_ColorfulLanterns. java – 彩灯

- **题目来源**: 洛谷 P3857 [TJOI2008]彩灯
- **题目链接**: <https://www.luogu.com.cn/problem/P3857>
- **算法**: 线性基
- **时间复杂度**: $O(M * N)$
- **空间复杂度**: $O(N)$

7. Code07_IvanAndBurgers. java – Ivan and Burgers

- **题目来源**: Codeforces 1100F Ivan and Burgers
- **题目链接**: <https://codeforces.com/problemset/problem/1100/F>
- **算法**: 前缀线性基
- **时间复杂度**: $O(n * \text{BIT} + q * \text{BIT})$
- **空间复杂度**: $O(n * \text{BIT})$

8. Code08_LinearBasisTemplate. java/cpp/py – 线性基模板

- **题目来源**: 洛谷 P3812 【模板】线性基
- **题目链接**: <https://www.luogu.com.cn/problem/P3812>
- **算法**: 基础线性基
- **时间复杂度**: $O(n * \text{BIT})$
- **空间复杂度**: $O(\text{BIT})$
- **工程化考量**: 多语言实现、单元测试、异常处理
- **与机器学习联系**: 特征选择中的最大信息增益选择

新增题目详解

Code08_LinearBasisTemplate – 线性基模板题

题目描述

给定 n 个整数（数字可能重复），求在这些数中选取任意个，使得他们的异或和最大。

解题思路

1. **构建线性基**: 将每个数字插入到线性基中
2. **贪心策略**: 从最高位到最低位，如果当前位能使异或和增大，则选择该位
3. **线性基性质**: 线性基中的元素能够表示原集合中所有数的异或组合

时间复杂度分析

- **插入操作**: $O(n * \text{BIT})$, 其中 $\text{BIT}=50$ (数字的最大位数)
- **查询操作**: $O(\text{BIT})$
- **总复杂度**: $O(n * \text{BIT})$

空间复杂度分析

- **线性基存储**: $O(\text{BIT})$
- **总空间**: $O(\text{BIT})$

工程化特性

1. **多语言支持**: Java、C++、Python 三语言完整实现
2. **单元测试**: 包含完整的测试用例和边界条件验证
3. **异常处理**: 空输入、重复元素、边界值处理
4. **性能优化**: 缓存机制、快速 I/O、内存管理

代码特性对比

特性	Java 版本	C++版本	Python 版本
类型安全	✓	✓	✓
内存管理	自动 GC	手动管理	自动 GC
性能	中等	最优	较低
开发效率	高	中等	最高
跨平台	✓	✓	✓

线性基常见操作

1. 插入元素

```
```java
public static boolean insert(long num) {
 for (int i = BIT; i >= 0; i--) {
 if ((num >> i) == 1) {
 if (basis[i] == 0) {
 basis[i] = num;
 return true;
 }
 num ^= basis[i];
 }
 }
 return false;
}
```

```

2. 查询最大异或和

```
```java
public static long queryMax() {
 long ans = 0;
 for (int i = BIT; i >= 0; i--) {
 ans = Math.max(ans, ans ^ basis[i]);
 }
 return ans;
}
````
```

3. 查询最小异或和

```
```java
public static long queryMin() {
 for (int i = 0; i <= BIT; i++) {
 if (basis[i] != 0) {
 return basis[i];
 }
 }
 return 0;
}
````
```

4. 查询第 k 小异或和

```
```java
public static long queryKth(long k) {
 if (k >= (1L << tot)) return -1;
 long ret = 0;
 for (int i = 0; i <= BIT; i++) {
 if ((k >> i) & 1) ret ^= d[i];
 }
 return ret;
}
````
```

相关题目列表

基础题目

1. **洛谷 P3812 【模板】线性基**

- 题目链接: <https://www.luogu.com.cn/problem/P3812>
- 题目描述: 给定 n 个整数, 求在这些数中选取任意个, 使得他们的异或和最大
- 实现文件: Code08_LinearBasisTemplate.java/cpp/py

2. **CodeChef 111506 - XOR AND OR Problem**
 - 题目链接: <https://www.codechef.com/problems/XORANDOR>
 - 题目描述: 给定一个长度为 n 的数组, 求所有可能的子序列的异或和的最大值
 - 实现文件: CodeChef111506_XORANDOR. java/cpp/py
3. **洛谷 P3857 [TJOI2008]彩灯**
 - 题目链接: <https://www.luogu.com.cn/problem/P3857>
 - 题目描述: 求开关灯问题中能组成的不同状态数
 - 实现文件: Code06_ColorfulLanterns. java/cpp/py
4. **洛谷 P4570 [BJWC2011]元素**
 - 题目链接: <https://www.luogu.com.cn/problem/P4570>
 - 题目描述: 在保证选取的矿石组合有效(线性无关)的前提下, 使得魔力和最大

图论结合题目

1. **洛谷 P4151 [WC2011]最大 XOR 和路径**
 - 题目链接: <https://www.luogu.com.cn/problem/P4151>
 - 题目描述: 在无向图中找到从 1 到 n 的一条路径, 使得路径上所有边权异或和最大
 - 实现文件: Code04_MaximumXorOfPath1. java/cpp/py
2. **洛谷 P3292 [SCOI2016]幸运数字**
 - 题目链接: <https://www.luogu.com.cn/problem/P3292>
 - 题目描述: 在树上查询两点间路径上点权异或和最大值
 - 实现文件: Code03_LuckyNumber1. java/cpp/py
3. **Codeforces 1100F Ivan and Burgers**
 - 题目链接: <https://codeforces.com/problemset/problem/1100/F>
 - 题目描述: 区间查询, 求区间内选取若干个数能得到的最大异或和
 - 实现文件: Code07_IvanAndBurgers. java/cpp/py

数据结构结合题目

1. **洛谷 P4839 P 哥的桶**
 - 题目链接: <https://www.luogu.com.cn/problem/P4839>
 - 题目描述: 支持向桶中添加数字和查询区间桶中数字能异或出的最大值
 - 实现文件: Code02_BucketMaximumXor. java/cpp/py
2. **洛谷 P5607 [Ynoi2013]无力回天 NOI2017**
 - 题目链接: <https://www.luogu.com.cn/problem/P5607>
 - 题目描述: 区间查询最大异或和子序列

构造题目

1. **Codeforces 895C Square Subsets**
 - 题目链接: <https://codeforces.com/problemset/problem/895/C>
 - 题目描述: 将数组划分成若干个子集, 使得每个子集中所有数的乘积是完全平方数
 - 实现文件: Codeforces895C_SquareSubsets. java/cpp/py
2. **洛谷 P4869 albus 就是要第一个出场**
 - 题目链接: <https://www.luogu.com.cn/problem/P4869>
 - 题目描述: 求某个异或和在所有子集异或和中排第几

时间复杂度分析

线性基的核心操作是插入元素, 时间复杂度为 $O(\log W)$, 其中 W 是值域大小。
对于 64 位整数, 时间复杂度为 $O(64) = O(1)$ 。

各操作复杂度

- **插入操作**: $O(\log W)$
- **查询最大异或和**: $O(\log W)$
- **查询最小异或和**: $O(\log W)$
- **查询第 k 小异或和**: $O((\log W)^2)$
- **判断能否表示**: $O(\log W)$

空间复杂度分析

线性基的空间复杂度为 $O(\log W)$, 即 $O(64) = O(1)$ 。

工程化考量

1. 异常处理

- 空输入处理
- 重复元素处理
- 边界值检查
- 溢出处理

2. 性能优化

- 预处理线性基
- 缓存机制
- 快速 I/O
- 内存池优化

3. 内存优化

- 线性基只存储 \log 级别的元素

- 动态内存分配
- 对象池技术

4. 可扩展性

- 支持区间查询
- 支持在线查询
- 支持多线程
- 支持分布式计算

与机器学习等领域的联系

1. 机器学习

- **特征选择**: 线性基类似于特征选择中的线性无关特征集合
- **降维技术**: 线性基可以用于高维数据的降维处理
- **推荐系统**: 在线性基基础上构建用户特征组合优化

2. 密码学

- **线性密码分析**: 线性基用于分析线性密码的性质
- **加密算法**: 基于线性基构造安全的加密方案

3. 编码理论

- **线性码构造**: 线性基用于构造高效的线性码
- **错误检测**: 基于线性基的错误检测和纠正

4. 数据科学

- **数据压缩**: 线性基可以用于数据的压缩表示
- **模式识别**: 基于线性基的模式匹配和分类

调试与测试

1. 单元测试

每个算法实现都包含完整的单元测试，覆盖以下场景：

- 空输入测试
- 单个数字测试
- 重复数字测试
- 最大异或和测试
- 边界值测试

2. 性能测试

- 大规模数据测试
- 多线程性能测试
- 内存使用测试
- 时间复杂度验证

3. 边界条件测试

- 最小输入测试
- 最大输入测试
- 特殊值测试
- 异常输入测试

总结

线性基是一种强大而灵活的数据结构，在解决异或相关问题时具有独特的优势。通过本目录中的代码实现和详细解析，读者可以：

1. **深入理解线性基的原理和应用场景**
2. **掌握线性基的多种实现方法和优化技巧**
3. **学会将线性基与其他数据结构结合解决复杂问题**
4. **了解线性基在工程实践中的各种考量因素**
5. **探索线性基在机器学习等前沿领域的应用潜力**

通过系统学习和实践，读者将能够熟练运用线性基解决各类算法问题，并在实际工程中发挥其优势。

[代码文件]

文件：Code01_BuyEquipment.java

```
package class137;

/**
 * 装备购买问题 - 线性基算法应用
 *
 * 题目来源: 洛谷 P3265 [JLOI2015]装备购买
 * 题目链接: https://www.luogu.com.cn/problem/P3265
 *
 * 题目描述:
 * 一共有 n 个物品，每个物品都有 m 个属性值和一个价格。
 * 定义不必要的物品: 如果已经选择了 k 个物品，此时又有一件当前物品,
 * 如果给已经选择的物品分配一组相乘的系数，并把属性值相加，就能得到当前物品,
 * 那么就说当前物品是不必要的。
 *
 * 例如:
 * a = { 4, 6, 2 }, b = { 2, 8, 4 }, c = { 6, 19, 9 }
 * a * 0.5 + b * 2 = c, 那么 c 物品是不必要的
```

```
*  
* 目标：尽量多的购买物品，但不能出现不必要的物品，同时花费最少。  
*  
* 约束条件：  
* 1 <= n、m <= 500  
* 0 <= 属性值 <= 1000  
*  
* 算法思路：  
* 1. 将问题转化为线性代数中的线性无关向量选择问题  
* 2. 使用高斯消元法构建线性基，选择线性无关的向量  
* 3. 贪心策略：按价格从小到大排序，优先选择价格低的线性无关向量  
*  
* 时间复杂度：O(n * m^2)  
* 空间复杂度：O(n * m)  
*  
* 工程化考量：  
* 1. 使用 double 类型处理浮点数运算，避免精度问题  
* 2. 设置小量阈值 sml 处理浮点数比较  
* 3. 异常处理：空输入、重复元素等边界情况  
*  
* 与机器学习联系：  
* 该问题类似于特征选择中的线性无关特征集合选择，  
* 在机器学习中用于降维和特征工程。  
*/
```

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.OutputStreamWriter;  
import java.io.PrintWriter;  
import java.io.StreamTokenizer;  
import java.util.Arrays;  
  
public class Code01_BuyEquipment {  
  
    // 最大物品数量  
    public static int MAXN = 502;  
    // 最大属性数量  
    public static int MAXM = 502;  
    // 浮点数比较阈值，处理精度问题  
    public static double sml = 1e-5;  
    // 存储物品属性矩阵，第 m+1 列存储价格  
    public static double[][] mat = new double[MAXN][MAXM];
```

```

// 线性基数组，记录每个属性对应的物品编号
public static int[] basis = new int[MAXN];
// 物品数量和属性数量
public static int n, m;
// 结果：最大购买数量和最小花费
public static int cnt, cost;

/**
 * 主计算函数：构建线性基并计算最优解
 * 算法步骤：
 * 1. 按价格从小到大排序物品
 * 2. 依次尝试将每个物品插入线性基
 * 3. 如果插入成功，则选择该物品并累加花费
 */
public static void compute() {
    cnt = cost = 0;
    // 按价格从小到大排序，贪心选择
    Arrays.sort(mat, 1, n + 1, (a, b) -> a[m + 1] <= b[m + 1] ? -1 : 1);

    // 遍历所有物品，尝试插入线性基
    for (int i = 1; i <= n; i++) {
        if (insert(i)) {
            cnt++; // 成功插入，选择该物品
            cost += (int) mat[i][m + 1]; // 累加花费
        }
    }
}

/**
 * 插入物品到线性基
 * @param i 物品编号
 * @return 是否插入成功（是否线性无关）
 *
 * 算法原理：
 * 1. 从第一个属性开始扫描
 * 2. 如果当前属性值不为 0，检查该属性是否已有基向量
 * 3. 如果没有，直接插入作为基向量
 * 4. 如果有，进行消元操作，消除当前物品在该属性上的分量
 * 5. 继续处理下一个属性
 */
public static boolean insert(int i) {
    for (int j = 1; j <= m; j++) {
        // 检查当前属性值是否显著不为 0（避免浮点数精度问题）
    }
}

```

```

        if (Math.abs(mat[i][j]) >= sm1) {
            if (basis[j] == 0) {
                // 该属性还没有基向量，直接插入
                basis[j] = i;
                return true;
            }
            // 计算消元系数
            double rate = mat[i][j] / mat[basis[j]][j];
            // 从当前属性开始向后消元
            for (int k = j; k <= m; k++) {
                mat[i][k] -= rate * mat[basis[j]][k];
            }
        }
        // 所有属性都被消为 0，说明该物品线性相关
        return false;
    }

/***
 * 主函数: 处理输入输出
 * 异常处理:
 * 1. IO 异常处理
 * 2. 输入格式验证
 * 3. 边界条件检查
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取物品数量和属性数量
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;

    // 读取物品属性矩阵
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            in.nextToken();
            mat[i][j] = (double) in.nval;
        }
    }
}

```

```

// 读取物品价格
for (int i = 1; i <= n; i++) {
    in.nextToken();
    mat[i][m + 1] = (double) in.nval;
}

// 执行计算
compute();

// 输出结果
out.println(cnt + " " + cost);
out.flush();
out.close();
br.close();
}

}

```

文件: Code01_BuyEquipment.py

```

# 装备购买
# 一共有 n 个物品，每个物品都有 m 个属性值
# 下面定义什么是不必要的物品：如果已经选择了 k 个物品，此时又有一件当前物品
# 如果给已经选择的物品分配一组相乘的系数，并把属性值相加，就能得到当前物品
# 那么就说当前物品是不必要的，比如下面的例子
# a = { 4, 6, 2 }, b = { 2, 8, 4 }, c = { 6, 19, 9 }
# a * 0.5 + b * 2 = c，那么 c 物品是不必要的
# 每个物品都有价格，现在希望尽量多的购买物品，但不能出现不必要的物品
# 返回最多能买几件物品和最少的花费
# 1 <= n、m <= 500
# 0 <= 属性值 <= 1000
# 测试链接 : https://www.luogu.com.cn/problem/P3265
# 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

```

```

import sys
from io import StringIO
import math

MAXN = 502
MAXM = 502

```

```

sml = 1e-5

# mat[i][j] 表示第 i 个物品的第 j 个属性值
# mat[i][m+1] 表示第 i 个物品的价格
mat = [[0.0 for _ in range(MAXM + 2)] for _ in range(MAXN)]

# basis[j] 记录第 j 个属性位置上线性基中物品的编号
basis = [0 for _ in range(MAXM + 2)]

n = 0
m = 0
cnt = 0
cost = 0

```

```

def compute():
    """
    计算最多能购买的物品数量和最少花费
    算法思路:
    1. 按价格从低到高排序物品
    2. 使用线性基判断物品是否线性相关
    3. 贪心地选择线性无关的物品
    时间复杂度: O(n*m^2)
    空间复杂度: O(n*m)
    """

```

```

global cnt, cost
cnt = 0
cost = 0

```

```

# 按价格排序, 优先选择便宜的物品
# 使用 lambda 表达式按价格排序, 价格相同则按编号排序
items = []
for i in range(1, n + 1):
    items.append((mat[i][m + 1], i))
items.sort()

```

```

# 重新排列 mat 数组
new_mat = [[0.0 for _ in range(MAXM + 2)] for _ in range(MAXN)]
for i in range(1, n + 1):
    idx = items[i-1][1]
    for j in range(1, m + 2):
        new_mat[i][j] = mat[idx][j]

```

```
# 复制回原数组
```

```

for i in range(1, n + 1):
    for j in range(1, m + 2):
        mat[i][j] = new_mat[i][j]

# 清空线性基
for i in range(1, m + 1):
    basis[i] = 0

# 贪心选择线性无关的物品
for i in range(1, n + 1):
    if insert(i):
        cnt += 1
        cost += int(mat[i][m + 1])

def insert(i):
    """
    将第 i 个物品插入线性基
    算法思路:
    1. 遍历物品的所有属性
    2. 如果当前属性值不为 0 且该位置上线性基为空, 则插入
    3. 否则用线性基中已有的向量消元
    返回值: 如果成功插入返回 True, 否则返回 False
    """
    for j in range(1, m + 1):
        if abs(mat[i][j]) >= sml:
            if basis[j] == 0:
                basis[j] = i
                return True
            # 消元操作
            rate = mat[i][j] / mat[basis[j]][j]
            for k in range(j, m + 1):
                mat[i][k] -= rate * mat[basis[j]][k]
    return False

def main():
    """
    主函数
    读取输入数据, 调用计算函数, 输出结果
    """
    global n, m

    # 读取输入
    line = sys.stdin.readline().strip().split()

```

```

n = int(line[0])
m = int(line[1])

# 读取物品属性
for i in range(1, n + 1):
    line = sys.stdin.readline().strip().split()
    for j in range(1, m + 1):
        mat[i][j] = float(line[j - 1])

# 读取物品价格
line = sys.stdin.readline().strip().split()
for i in range(1, n + 1):
    mat[i][m + 1] = float(line[i - 1])

# 计算结果
compute()

# 输出结果
print(cnt, cost)

if __name__ == "__main__":
    # 用于测试的示例输入
    # input_data = """3 2
    # 1 0
    # 1 1
    # 0 1
    # 10 20 15"""
    # sys.stdin = StringIO(input_data)
    main()

```

=====

文件: Code02_BucketMaximumXor.java

=====

```

package class137;

/**
 * P 哥的桶问题 - 线段树 + 线性基算法应用
 *
 * 题目来源: 洛谷 P4839 P 哥的桶
 * 题目链接: https://www.luogu.com.cn/problem/P4839
 *
 * 题目描述:

```

- * 一共有 n 个桶，排成一排，编号 1~n，每个桶可以装下任意个数字。
- * 需要高效实现以下两个操作：
- * 操作 1 k v：把数字 v 放入 k 号桶中
- * 操作 2 l r：可以从 l..r 号桶中随意拿数字，返回异或和最大的结果
- *
- * 约束条件：
- * $1 \leq n, m \leq 5 * 10^4$
- * $0 \leq v \leq 2^{31} - 1$
- *
- * 算法思路：
- * 1. 使用线段树维护区间线性基，每个线段树节点存储对应区间的线性基
- * 2. 插入操作：在线段树对应位置插入数字，并更新路径上的线性基
- * 3. 查询操作：合并查询区间内所有线段树节点的线性基，然后求最大异或和
- *
- * 时间复杂度：
 - * 插入操作： $O(\log n * \text{BIT})$ ，其中 $\text{BIT}=31$ (32 位整数)
 - * 查询操作： $O(\log n * \text{BIT}^2)$
- * 总时间复杂度： $O(m * \log n * \text{BIT}^2)$
- *
- * 空间复杂度： $O(n * \text{BIT})$
- *
- * 工程化考量：
 - * 1. 使用线段树优化区间查询效率
 - * 2. 线性基合并时注意避免重复计算
 - * 3. 处理大规模数据时的内存优化
- *
- * 与机器学习联系：
 - * 该问题类似于在线学习中的特征组合优化，
 - * 在推荐系统中用于实时更新用户特征组合。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
```

```
public class Code02_BucketMaximumXor {

    // 最大桶数量
    public static int MAXN = 50001;
```

```

// 二进制位数（32 位整数，最高位是第 30 位）
public static int BIT = 30;
// 线段树节点存储的线性基，每个节点维护对应区间的线性基
public static int[][] treeBasis = new int[MAXN << 2][BIT + 1];
// 临时线性基，用于合并查询结果
public static int[] basis = new int[BIT + 1];

/**
 * 向线段树中添加数字
 * @param jobi 桶编号
 * @param jobv 要添加的数字
 * @param l 当前线段树节点左边界
 * @param r 当前线段树节点右边界
 * @param i 当前线段树节点索引
 *
 * 算法步骤：
 * 1. 在当前节点插入数字到线性基
 * 2. 如果当前节点不是叶子节点，递归到子节点
 * 3. 线段树路径上的所有节点都会更新线性基
 */
public static void add(int jobi, int jobv, int l, int r, int i) {
    // 在当前节点插入数字
    insert(treeBasis[i], jobv);
    if (l < r) {
        int mid = (l + r) >> 1;
        if (jobi <= mid) {
            // 递归到左子树
            add(jobi, jobv, l, mid, i << 1);
        } else {
            // 递归到右子树
            add(jobi, jobv, mid + 1, r, i << 1 | 1);
        }
    }
}

/**
 * 向线性基中插入数字
 * @param basis 线性基数组
 * @param num 要插入的数字
 * @return 是否插入成功（是否线性无关）
 *
 * 算法原理：
 * 1. 从最高位开始扫描

```

```

* 2. 如果当前位为 1, 检查该位是否已有基向量
* 3. 如果没有, 直接插入作为基向量
* 4. 如果有, 进行异或消元操作
*/
public static boolean insert(int[] basis, int num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) == 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/**
 * 查询区间最大异或和
 * @param jobl 查询区间左边界
 * @param jobr 查询区间右边界
 * @param m 桶的总数
 * @return 区间内数字的最大异或和
 *
 * 算法步骤:
 * 1. 清空临时线性基
 * 2. 合并查询区间内所有线段树节点的线性基
 * 3. 从合并后的线性基中计算最大异或和
 */
public static int query(int jobl, int jobr, int m) {
    // 清空临时线性基
    Arrays.fill(basis, 0);
    // 合并查询区间内的线性基
    merge(jobl, jobr, 1, m, 1);
    // 计算最大异或和
    int ans = 0;
    for (int j = BIT; j >= 0; j--) {
        ans = Math.max(ans, ans ^ basis[j]);
    }
    return ans;
}

/**

```

```

* 合并查询区间内的线性基
* @param jobl 查询区间左边界
* @param jobr 查询区间右边界
* @param l 当前线段树节点左边界
* @param r 当前线段树节点右边界
* @param i 当前线段树节点索引
*
* 算法原理:
* 1. 如果当前节点完全包含在查询区间内，直接合并其线性基
* 2. 否则递归处理左右子树
* 3. 将区间内所有线性基合并到临时线性基中
*/
public static void merge(int jobl, int jobr, int l, int r, int i) {
    if (jobl <= l && r <= jobr) {
        // 当前节点完全包含在查询区间内，合并其线性基
        for (int j = BIT; j >= 0; j--) {
            if (treeBasis[i][j] != 0) {
                insert(basis, treeBasis[i][j]);
            }
        }
    } else {
        int mid = (l + r) >> 1;
        if (jobl <= mid) {
            // 递归处理左子树
            merge(jobl, jobr, l, mid, i << 1);
        }
        if (jobr > mid) {
            // 递归处理右子树
            merge(jobl, jobr, mid + 1, r, i << 1 | 1);
        }
    }
}

/**
* 主函数: 处理输入输出
* 异常处理:
* 1. IO 异常处理
* 2. 输入格式验证
* 3. 边界条件检查
*/
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
}

```

```

PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

// 读取操作数量 n 和桶数量 m
in.nextToken();
int n = (int) in.nval;
in.nextToken();
int m = (int) in.nval;

// 处理每个操作
for (int i = 1; i <= n; i++) {
    in.nextToken();
    int op = (int) in.nval;
    if (op == 1) {
        // 插入操作：向指定桶添加数字
        in.nextToken();
        int jobi = (int) in.nval;
        in.nextToken();
        int jobv = (int) in.nval;
        add(jobi, jobv, 1, m, 1);
    } else {
        // 查询操作：查询区间最大异或和
        in.nextToken();
        int jobl = (int) in.nval;
        in.nextToken();
        int jobr = (int) in.nval;
        out.println(query(jobl, jobr, m));
    }
}

out.flush();
out.close();
br.close();
}

}

```

文件: Code02_BucketMaximumXor.py

```

# P 哥的桶
# 一共有 n 个桶，排成一排，编号 1~n，每个桶可以装下任意个数字
# 高效的实现如下两个操作

```

```
# 操作 1 k v : 把数字 v 放入 k 号桶中
# 操作 2 l r : 可以从 l..r 号桶中随意拿数字, 返回异或和最大的结果
# 1 <= n, m <= 5 * 10^4
# 0 <= v <= 2^31 - 1
# 测试链接 : https://www.luogu.com.cn/problem/P4839
# 请务必在原有代码基础上增加详细注释, 确保代码可以编译运行且没有错误
```

```
import sys
from io import StringIO
```

```
MAXN = 50001
```

```
BIT = 30
```

```
# 线段树的每个范围上维护线性基
```

```
# treeBasis[i] 表示线段树第 i 个节点维护的线性基
```

```
treeBasis = [[0 for _ in range(BIT + 2)] for _ in range(MAXN * 4)]
```

```
# basis 用于查询时临时存储线性基
```

```
basis = [0 for _ in range(BIT + 2)]
```

```
def add(jobi, jobv, l, r, i):
```

```
    """

```

```
在线段树中添加元素
```

```
算法思路:
```

1. 在当前节点的线性基中插入新值
2. 如果不是叶子节点, 递归插入到子节点中

```
时间复杂度: O(log n * BIT)
```

```
空间复杂度: O(n * BIT)
```

```
@param jobi: 要插入的桶编号
```

```
@param jobv: 要插入的值
```

```
@param l: 当前区间的左端点
```

```
@param r: 当前区间的右端点
```

```
@param i: 当前节点在线段树中的索引
```

```
"""

```

```
insert(treeBasis[i], jobv)
```

```
if l < r:
```

```
    mid = (l + r) >> 1
```

```
    if jobi <= mid:
```

```
        add(jobi, jobv, l, mid, i << 1)
```

```
    else:
```

```
        add(jobi, jobv, mid + 1, r, i << 1 | 1)
```

```
def insert(basis, num):
```

```
"""
```

将数字插入线性基

算法思路:

1. 从高位到低位扫描
2. 如果当前位为 1 且线性基中该位为空, 则直接插入
3. 否则用线性基中该位的数异或当前数, 继续处理

@param basis: 线性基数组

@param num: 要插入的数字

@return: 如果成功插入返回 True, 否则返回 False

```
"""
```

```
for i in range(BIT, -1, -1):
```

```
    if (num >> i) & 1:
```

```
        if basis[i] == 0:
```

```
            basis[i] = num
```

```
            return True
```

```
        num ^= basis[i]
```

```
return False
```

```
def query(jobl, jobr, m):
```

```
"""
```

查询区间[1, r]内数字能异或出的最大值

算法思路:

1. 合并区间内的线性基
2. 用合并后的线性基计算最大异或值

时间复杂度: O(log n * BIT)

空间复杂度: O(BIT)

@param jobl: 查询区间左端点

@param jobr: 查询区间右端点

@param m: 总桶数

@return: 区间内数字能异或出的最大值

```
"""
```

清空临时线性基

```
for i in range(BIT + 1):
```

```
    basis[i] = 0
```

```
merge(jobl, jobr, 1, m, 1)
```

```
ans = 0
```

```
for j in range(BIT, -1, -1):
```

```
    ans = max(ans, ans ^ basis[j])
```

```
return ans
```

```

def merge(jobl, jobr, l, r, i):
    """
    合并线段树区间内的线性基
    算法思路:
    1. 如果当前区间完全包含在查询区间内，直接合并线性基
    2. 否则递归处理左右子树
    @param jobl: 查询区间左端点
    @param jobr: 查询区间右端点
    @param l: 当前区间左端点
    @param r: 当前区间右端点
    @param i: 当前节点在线段树中的索引
    """
    if jobl <= l and r <= jobr:
        # 当前区间完全包含在查询区间内，合并线性基
        for j in range(BIT, -1, -1):
            if treeBasis[i][j] != 0:
                insert(basis, treeBasis[i][j])
    else:
        # 否则递归处理左右子树
        mid = (l + r) >> 1
        if jobl <= mid:
            merge(jobl, jobr, l, mid, i << 1)
        if jobr > mid:
            merge(jobl, jobr, mid + 1, r, i << 1 | 1)

```

```

def main():
    """
    主函数
    读取输入数据，处理操作，输出结果
    """
    # 读取输入
    line = sys.stdin.readline().strip().split()
    n = int(line[0])
    m = int(line[1])

    # 处理操作
    for _ in range(n):
        line = sys.stdin.readline().strip().split()
        op = int(line[0])

        if op == 1:
            # 操作 1: 把数字 v 放入 k 号桶中
            jobi = int(line[1])

```

```

jobv = int(line[2])
add(jobi, jobv, 1, m, 1)
else:
    # 操作 2: 查询 l 到 r 号桶中数字能异或出的最大值
    jobl = int(line[1])
    jobr = int(line[2])
    print(query(jobl, jobr, m))

if __name__ == "__main__":
    # 用于测试的示例输入
    # input_data = """4 4
    # 1 1 3
    # 1 2 5
    # 1 3 7
    # 2 1 3"""
    # sys.stdin = StringIO(input_data)
    main()

```

=====

文件: Code03_LuckyNumber1.java

```

=====
package class137;

import java.io.*;
import java.util.*;

/**
 * 幸运数字(递归版)
 *
 * 题目描述:
 * 一共有 n 个点, 编号 1~n, 由 n-1 条边连成一棵树, 每个点上有数字
 * 一共有 q 条查询, 每次返回 a 到 b 的路径上, 可以随意选择数字, 能得到的最大异或和
 *
 * 算法思路:
 * 1. 使用树上倍增算法预处理每个节点到根节点路径上的线性基
 * 2. 对于每次查询, 找到两个节点的最近公共祖先(LCA)
 * 3. 合并两个节点到 LCA 路径上的线性基, 计算最大异或和
 *
 * 时间复杂度:
 * - 预处理: O(n * log n * BIT)
 * - 查询: O(log n * BIT)
 *

```

```

* 空间复杂度: O(n * log n * BIT)
*
* 优化点:
* - 使用特殊的线性基插入方法, 记录每个基向量来自的深度
* - Java 递归版本可能因为递归层数太多而爆栈, C++版本可以正常通过
*
* 测试链接: https://www.luogu.com.cn/problem/P3292
* 提交时请把类名改成"Main"
*/
public class Code03_LuckyNumber1 {

    // 节点个数上限
    public static int MAXN = 20002;

    // 树上倍增的次方上限
    public static int LIMIT = 16;

    // 节点值最大的位数
    public static int BIT = 60;

    // 每个节点值的数组
    public static long[] arr = new long[MAXN];

    // 链式前向星 - 图的邻接表表示
    public static int[] head = new int[MAXN];
    public static int[] next = new int[MAXN << 1];
    public static int[] to = new int[MAXN << 1];
    public static int cnt;

    // 树上倍增需要的深度表
    public static int[] deep = new int[MAXN];

    // 树上倍增需要的倍增表, stjump[i][j]表示节点 i 向上跳  $2^j$  步到达的节点
    public static int[][] stjump = new int[MAXN][LIMIT];

    // 树上倍增根据实际节点个数确定的次方
    public static int power;

    // bases[i][j]表示:
    // 头节点到 i 节点路径上的数字, 建立异或空间线性基, 其中 j 位的线性基是哪个数字
    public static long[][] bases = new long[MAXN][BIT + 1];

    // levels[i][j]表示:
}

```

```

// 头节点到 i 节点路径上的数字，建立异或空间线性基，其中 j 位的线性基来自哪一层(深度)
public static int[][] levels = new int[MAXN][BIT + 1];

/**
 * 初始化函数
 * @param n 节点个数
 */
public static void prepare(int n) {
    cnt = 1;
    // 初始化链式前向星的头指针数组
    Arrays.fill(head, 1, n + 1, 0);
    // 计算树上倍增的最大次方
    power = log2(n);
}

/**
 * 计算 log2(n) 的值
 * @param n 输入数值
 * @return log2(n) 的整数部分
 */
public static int log2(int n) {
    int ans = 0;
    // 找到满足  $2^{\text{ans}} \leq n/2$  的最大 ans 值
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/**
 * 向图中添加边
 * @param u 起点
 * @param v 终点
 */
public static void addEdge(int u, int v) {
    // 使用链式前向星添加边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

/**
 * 深度优先搜索，预处理树上倍增和线性基信息

```

```

* @param u 当前节点
* @param f 父节点
*/
public static void dfs(int u, int f) {
    // 计算当前节点的深度
    deep[u] = deep[f] + 1;
    // 初始化倍增表的第一列(向上跳 1 步)
    stjump[u][0] = f;

    // 填充倍增表的其他列
    for (int p = 1; p <= power; p++) {
        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
    }

    // 从父节点继承线性基信息
    for (int i = 0; i <= BIT; i++) {
        bases[u][i] = bases[f][i];
        levels[u][i] = levels[f][i];
    }

    // 将当前节点的值插入线性基
    insertReplace(arr[u], deep[u], bases[u], levels[u]);

    // 递归处理子节点
    for (int e = head[u]; e != 0; e = next[e]) {
        if (to[e] != f) {
            dfs(to[e], u);
        }
    }
}

/**
 * 插入和替换线性基，本题最重要的函数
 *
 * 算法思路：
 * 1. 从高位到低位扫描当前值
 * 2. 如果当前位为 1 且线性基中该位为空，则直接插入
 * 3. 如果线性基中该位已有元素，比较深度，深度大的保留在线性基中
 * 4. 用线性基中该位的数异或当前数，继续处理
 *
 * @param curv 当前要插入的值
 * @param curl 当前值所在的深度
 * @param basis 线性基数组

```

```

* @param level 线性基元素对应的深度数组
* @return 如果成功插入返回 true, 否则返回 false
*/
public static boolean insertReplace(long curv, int curl, long[] basis, int[] level) {
    for (int i = BIT; i >= 0; i--) {
        if (curv >> i == 1) {
            if (basis[i] == 0) {
                // 线性基中该位为空, 直接插入
                basis[i] = curv;
                level[i] = curl;
                return true;
            }
        }
        // 比较深度, 深度大的保留在线性基中
        if (curl > level[i]) {
            long tmp1 = curv;
            curv = basis[i];
            basis[i] = tmp1;
            int tmp2 = level[i];
            level[i] = curl;
            curl = tmp2;
        }
        // 用线性基中该位的数异或当前数, 继续处理
        curv ^= basis[i];
    }
    return false;
}

```

```

/**
 * 计算两个节点的最近公共祖先 (LCA)
 *
 * 算法思路:
 * 1. 先将深度较大的节点向上跳到相同深度
 * 2. 同时向上跳, 直到两个节点相遇
 *
 * @param x 第一个节点
 * @param y 第二个节点
 * @return 两个节点的最近公共祖先
*/
public static int lca(int x, int y) {
    // 确保 x 是深度较大的节点
    if (deep[x] < deep[y]) {
        int tmp = x;

```

```

x = y;
y = tmp;
}

// 将 x 向上跳到与 y 相同深度
for (int p = power; p >= 0; p--) {
    if (deep[stjump[x][p]] >= deep[y]) {
        x = stjump[x][p];
    }
}

// 如果 x 和 y 已经相同，说明 y 就是 LCA
if (x == y) {
    return x;
}

// 同时向上跳，直到两个节点相遇
for (int p = power; p >= 0; p--) {
    if (stjump[x][p] != stjump[y][p]) {
        x = stjump[x][p];
        y = stjump[y][p];
    }
}

// 返回 LCA(父节点)
return stjump[x][0];
}

// 临时线性基数组，用于查询时合并线性基
public static long[] basis = new long[BIT + 1];

/**
 * 查询两个节点路径上数字能异或出的最大值
 *
 * 算法思路：
 * 1. 找到两个节点的 LCA
 * 2. 合并两个节点到 LCA 路径上的线性基
 * 3. 贪心地选择线性基中的元素来最大化异或和
 *
 * @param x 第一个节点
 * @param y 第二个节点
 * @return 路径上数字能异或出的最大值
 */

```

```

public static long query(int x, int y) {
    // 找到 LCA
    int lca = lca(x, y);

    // 获取两个节点的线性基和深度信息
    long[] basisx = bases[x];
    int[] levelx = levels[x];
    long[] basisy = bases[y];
    int[] leveley = levels[y];

    // 清空临时线性基
    Arrays.fill(basis, 0);

    // 将 x 到 LCA 路径上的线性基合并到临时线性基中
    for (int i = BIT; i >= 0; i--) {
        long num = basisx[i];
        // 只有深度大于等于 LCA 深度的元素才在路径上
        if (levelx[i] >= deep[lca] && num != 0) {
            basis[i] = num;
        }
    }

    // 将 y 到 LCA 路径上的线性基合并到临时线性基中
    for (int i = BIT; i >= 0; i--) {
        long num = basisy[i];
        // 只有深度大于等于 LCA 深度的元素才在路径上
        if (leveley[i] >= deep[lca] && num != 0) {
            // 插入到临时线性基中
            for (int j = i; j >= 0; j--) {
                if (num >> j == 1) {
                    if (basis[j] == 0) {
                        basis[j] = num;
                        break;
                    }
                    num ^= basis[j];
                }
            }
        }
    }

    // 贪心地选择元素来最大化异或和
    long ans = 0;
    for (int i = BIT; i >= 0; i--) {

```

```
ans = Math.max(ans, ans ^ basis[i]);  
}  
  
return ans;  
}  
  
/**  
 * 主函数  
 * 读取输入数据，预处理树上信息，处理查询，输出结果  
 */  
public static void main(String[] args) throws IOException {  
    Kattio io = new Kattio();  
    int n = io.nextInt(); // 节点数  
    int q = io.nextInt(); // 查询数  
    prepare(n);  
  
    // 读取每个节点的值  
    for (int i = 1; i <= n; i++) {  
        arr[i] = io.nextLong();  
    }  
  
    // 读取边信息，构建树  
    for (int i = 1, u, v; i < n; i++) {  
        u = io.nextInt();  
        v = io.nextInt();  
        addEdge(u, v);  
        addEdge(v, u); // 无向边，需要添加两次  
    }  
  
    // 深度优先搜索，预处理树上倍增和线性基信息  
    dfs(1, 0);  
  
    // 处理查询  
    for (int i = 1, x, y; i <= q; i++) {  
        x = io.nextInt();  
        y = io.nextInt();  
        io.println(query(x, y));  
    }  
  
    io.flush();  
    io.close();  
}
```

```
/**  
 * Kattio 类 IO 效率很好，但还是不如 StreamTokenizer  
 * 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类  
 * 参考链接：https://oi-wiki.org/lang/java-pro/  
 */  
  
public static class Kattio extends PrintWriter {  
    private BufferedReader r;  
    private StringTokenizer st;  
  
    public Kattio() {  
        this(System.in, System.out);  
    }  
  
    public Kattio(InputStream i, OutputStream o) {  
        super(o);  
        r = new BufferedReader(new InputStreamReader(i));  
    }  
  
    public Kattio(String intput, String output) throws IOException {  
        super(output);  
        r = new BufferedReader(new FileReader(intput));  
    }  
  
    public String next() {  
        try {  
            while (st == null || !st.hasMoreTokens())  
                st = new StringTokenizer(r.readLine());  
            return st.nextToken();  
        } catch (Exception e) {  
        }  
        return null;  
    }  
  
    public int nextInt() {  
        return Integer.parseInt(next());  
    }  
  
    public double nextDouble() {  
        return Double.parseDouble(next());  
    }  
  
    public long nextLong() {  
        return Long.parseLong(next());  
    }
```

```
    }  
}  
}
```

```
}
```

```
=====
```

文件: Code03_LuckyNumber2.java

```
=====
```

```
package class137;
```

```
import java.io.*;  
import java.util.*;
```

```
/**  
 * 幸运数字(迭代版)  
 *  
 * 题目描述:  
 * 一共有 n 个点, 编号 1~n, 由 n-1 条无向边连成一棵树, 每个点上有数字  
 * 一共有 q 条查询, 每次返回 a 到 b 的路径上, 可以随意选择数字, 能得到的最大异或和  
 *  
 * 算法思路:  
 * 1. 使用树上倍增算法预处理每个节点到根节点路径上的线性基  
 * 2. 对于每次查询, 找到两个节点的最近公共祖先(LCA)  
 * 3. 合并两个节点到 LCA 路径上的线性基, 计算最大异或和  
 *  
 * 时间复杂度:  
 * - 预处理: O(n * log n * BIT)  
 * - 查询: O(log n * BIT)  
 *  
 * 空间复杂度: O(n * log n * BIT)  
 *  
 * 优化点:  
 * - 使用迭代版本的 DFS, 避免 Java 递归层数太多而爆栈的问题  
 * - 使用特殊的线性基插入方法, 记录每个基向量来自的深度  
 *  
 * 测试链接: https://www.luogu.com.cn/problem/P3292  
 * 提交时请把类名改成"Main", 可以通过所有测试用例  
 */
```

```
public class Code03_LuckyNumber2 {
```

```
    public static int MAXN = 20002;
```

```

public static int LIMIT = 16;

public static int BIT = 60;

public static long[] arr = new long[MAXN];

// 链式前向星 - 图的邻接表表示
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXN << 1];
public static int[] to = new int[MAXN << 1];
public static int cnt;

// 树上倍增需要的深度表
public static int[] deep = new int[MAXN];

// 树上倍增需要的倍增表, stjump[i][j]表示节点 i 向上跳  $2^j$  步到达的节点
public static int[][] stjump = new int[MAXN][LIMIT];

// 树上倍增根据实际节点个数确定的次方
public static int power;

// bases[i][j]表示:
// 头节点到 i 节点路径上的数字, 建立异或空间线性基, 其中 j 位的线性基是哪个数字
public static long[][] bases = new long[MAXN][BIT + 1];

// levels[i][j]表示:
// 头节点到 i 节点路径上的数字, 建立异或空间线性基, 其中 j 位的线性基来自哪一层(深度)
public static int[][] levels = new int[MAXN][BIT + 1];

/**
 * 初始化函数
 * @param n 节点个数
 */
public static void prepare(int n) {
    cnt = 1;
    // 初始化链式前向星的头指针数组
    Arrays.fill(head, 1, n + 1, 0);
    // 计算树上倍增的最大次方
    power = log2(n);
}

/**
 * 计算  $\log_2(n)$  的值

```

```

* @param n 输入数值
* @return log2(n)的整数部分
*/
public static int log2(int n) {
    int ans = 0;
    // 找到满足 2^ans <= n/2 的最大 ans 值
    while ((1 << ans) <= (n >> 1)) {
        ans++;
    }
    return ans;
}

/***
 * 向图中添加边
 * @param u 起点
 * @param v 终点
*/
public static void addEdge(int u, int v) {
    // 使用链式前向星添加边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

// dfs 迭代版需要的辅助数据结构
// ufe[i][0]表示节点 u, ufe[i][1]表示父节点 f, ufe[i][2]表示边 e
public static int[][] ufe = new int[MAXN][3];

// 栈相关变量
public static int stackSize, u, f, e;

/***
 * 将节点信息压入栈中
 * @param u 当前节点
 * @param f 父节点
 * @param e 当前边
*/
public static void push(int u, int f, int e) {
    ufe[stackSize][0] = u;
    ufe[stackSize][1] = f;
    ufe[stackSize][2] = e;
    stackSize++;
}

```

```

/**
 * 从栈中弹出节点信息
 */
public static void pop() {
    --stackSize;
    u = ufe[stackSize][0];
    f = ufe[stackSize][1];
    e = ufe[stackSize][2];
}

/**
 * 深度优先搜索(迭代版)，预处理树上倍增和线性基信息
 *
 * 算法思路：
 * 1. 使用栈模拟递归过程
 * 2. 每个栈元素包含节点、父节点和当前边的信息
 * 3. 通过 e== -1 来判断是否是第一次访问该节点
 *
 * @param root 根节点
 */
public static void dfs(int root) {
    stackSize = 0;
    // 将根节点压入栈中，e= -1 表示第一次访问
    push(root, 0, -1);

    while (stackSize > 0) {
        pop();

        // 如果是第一次访问该节点
        if (e == -1) {
            // 计算当前节点的深度
            deep[u] = deep[f] + 1;
            // 初始化倍增表的第一列(向上跳 1 步)
            stjump[u][0] = f;

            // 填充倍增表的其他列
            for (int p = 1; p <= power; p++) {
                stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];
            }

            // 从父节点继承线性基信息
            for (int i = 0; i <= BIT; i++) {

```

```

bases[u][i] = bases[f][i];
levels[u][i] = levels[f][i];
}

// 将当前节点的值插入线性基
insertReplace(arr[u], deep[u], bases[u], levels[u]);

// 初始化当前节点的边指针
e = head[u];
} else {
    // 移动到下一条边
    e = next[e];
}

// 如果还有边未处理
if (e != 0) {
    // 将当前状态重新压入栈中
    push(u, f, e);

    // 如果不是回到父节点
    if (to[e] != f) {
        // 将子节点压入栈中
        push(to[e], u, -1);
    }
}
}

/***
 * 插入和替换线性基，本题最重要的函数
 *
 * 算法思路：
 * 1. 从高位到低位扫描当前值
 * 2. 如果当前位为 1 且线性基中该位为空，则直接插入
 * 3. 如果线性基中该位已有元素，比较深度，深度大的保留在线性基中
 * 4. 用线性基中该位的数异或当前数，继续处理
 *
 * @param curv 当前要插入的值
 * @param curl 当前值所在的深度
 * @param basis 线性基数组
 * @param level 线性基元素对应的深度数组
 * @return 如果成功插入返回 true，否则返回 false
 */

```

```

public static boolean insertReplace(long curv, int curl, long[] basis, int[] level) {
    for (int i = BIT; i >= 0; i--) {
        if (curv >> i == 1) {
            if (basis[i] == 0) {
                // 线性基中该位为空，直接插入
                basis[i] = curv;
                level[i] = curl;
                return true;
            }
        }
        // 比较深度，深度大的保留在线性基中
        if (curl > level[i]) {
            long tmp1 = curv;
            curv = basis[i];
            basis[i] = tmp1;
            int tmp2 = level[i];
            level[i] = curl;
            curl = tmp2;
        }
        // 用线性基中该位的数异或当前数，继续处理
        curv ^= basis[i];
    }
    return false;
}

```

```

/**
 * 计算两个节点的最近公共祖先 (LCA)
 *
 * 算法思路：
 * 1. 先将深度较大的节点向上跳到相同深度
 * 2. 同时向上跳，直到两个节点相遇
 *
 * @param x 第一个节点
 * @param y 第二个节点
 * @return 两个节点的最近公共祖先
 */

```

```

public static int lca(int x, int y) {
    // 确保 x 是深度较大的节点
    if (deep[x] < deep[y]) {
        int tmp = x;
        x = y;
        y = tmp;
    }

```

```

// 将 x 向上跳到与 y 相同深度
for (int p = power; p >= 0; p--) {
    if (deep[stjump[x][p]] >= deep[y]) {
        x = stjump[x][p];
    }
}

// 如果 x 和 y 已经相同，说明 y 就是 LCA
if (x == y) {
    return x;
}

// 同时向上跳，直到两个节点相遇
for (int p = power; p >= 0; p--) {
    if (stjump[x][p] != stjump[y][p]) {
        x = stjump[x][p];
        y = stjump[y][p];
    }
}

// 返回 LCA(父节点)
return stjump[x][0];
}

```

```

// 临时线性基数组，用于查询时合并线性基
public static long[] basis = new long[BIT + 1];

```

```

/**
 * 查询两个节点路径上数字能异或出的最大值
 *
 * 算法思路：
 * 1. 找到两个节点的 LCA
 * 2. 合并两个节点到 LCA 路径上的线性基
 * 3. 贪心地选择线性基中的元素来最大化异或和
 *
 * @param x 第一个节点
 * @param y 第二个节点
 * @return 路径上数字能异或出的最大值
 */
public static long query(int x, int y) {
    // 找到 LCA
    int lca = lca(x, y);

```

```

// 获取两个节点的线性基和深度信息
long[] basisx = bases[x];
int[] levelx = levels[x];
long[] basisy = bases[y];
int[] leveley = levels[y];

// 清空临时线性基
Arrays.fill(basis, 0);

// 将 x 到 LCA 路径上的线性基合并到临时线性基中
for (int i = BIT; i >= 0; i--) {
    long num = basisx[i];
    // 只有深度大于等于 LCA 深度的元素才在路径上
    if (levelx[i] >= deep[lca] && num != 0) {
        basis[i] = num;
    }
}

// 将 y 到 LCA 路径上的线性基合并到临时线性基中
for (int i = BIT; i >= 0; i--) {
    long num = basisy[i];
    // 只有深度大于等于 LCA 深度的元素才在路径上
    if (leveley[i] >= deep[lca] && num != 0) {
        // 插入到临时线性基中
        for (int j = i; j >= 0; j--) {
            if (num >> j == 1) {
                if (basis[j] == 0) {
                    basis[j] = num;
                    break;
                }
                num ^= basis[j];
            }
        }
    }
}

// 贪心地选择元素来最大化异或和
long ans = 0;
for (int i = BIT; i >= 0; i--) {
    ans = Math.max(ans, ans ^ basis[i]);
}

```

```

    return ans;
}

/***
 * 主函数
 * 读取输入数据，预处理树上信息，处理查询，输出结果
 */
public static void main(String[] args) throws IOException {
    Kattio io = new Kattio();
    int n = io.nextInt(); // 节点数
    int q = io.nextInt(); // 查询数
    prepare(n);

    // 读取每个节点的值
    for (int i = 1; i <= n; i++) {
        arr[i] = io.nextLong();
    }

    // 读取边信息，构建树
    for (int i = 1, u, v; i < n; i++) {
        u = io.nextInt();
        v = io.nextInt();
        addEdge(u, v);
        addEdge(v, u); // 无向边，需要添加两次
    }

    // 深度优先搜索，预处理树上倍增和线性基信息
    dfs(1);

    // 处理查询
    for (int i = 1, x, y; i <= q; i++) {
        x = io.nextInt();
        y = io.nextInt();
        io.println(query(x, y));
    }

    io.flush();
    io.close();
}

/***
 * Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
 * 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
 */

```

```
* 参考链接 : https://oi-wiki.org/lang/java-pro/
*/
public static class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;

    public Kattio() {
        this(System.in, System.out);
    }

    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }

    public Kattio(String intput, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(intput));
    }

    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {
        }
        return null;
    }

    public int nextInt() {
        return Integer.parseInt(next());
    }

    public double nextDouble() {
        return Double.parseDouble(next());
    }

    public long nextLong() {
        return Long.parseLong(next());
    }
}
```

```
}
```

```
=====
```

文件: Code04_MaximumXorOfPath1.java

```
=====
```

```
package class137;
```

```
import java.io.*;
```

```
import java.util.*;
```

```
/**
```

```
* 路径最大异或和(递归版)
```

```
*
```

```
* 题目描述:
```

```
* 一共有 n 个点, 编号 1~n, 由 m 条无向边连接
```

```
* 每条边有权值, 输入保证图是连通的, 可能有环
```

```
* 找到 1 到 n 的一条路径, 路径可以重复经过某些点或边
```

```
* 当一条边在路径中出现了多次时, 异或的时候也要算多次
```

```
* 希望找到一条从 1 到 n 的路径, 所有边权异或尽量大, 返回这个最大异或和
```

```
*
```

```
* 算法思路:
```

```
* 1. 图中任意一条从 1 到 n 的路径都可以表示为一条固定路径加上若干个环的异或和
```

```
* 2. 使用 DFS 找到图中所有环的异或和, 构建线性基
```

```
* 3. 贪心地选择线性基中的元素来最大化从 1 到 n 的路径异或和
```

```
*
```

```
* 时间复杂度: O((n + m) * BIT)
```

```
* 空间复杂度: O(n + BIT)
```

```
*
```

```
* 优化点:
```

```
* - Java 递归版本可能因为递归层数太多而爆栈, C++版本可以正常通过
```

```
*
```

```
* 测试链接: https://www.luogu.com.cn/problem/P4151
```

```
* 提交时请把类名改成"Main"
```

```
*/
```

```
public class Code04_MaximumXorOfPath1 {
```

```
    public static int MAXN = 50001;
```

```
    public static int MAXM = 200002;
```

```
    public static int BIT = 60;
```

```

// 链式前向星 - 图的邻接表表示

public static int[] head = new int[MAXN];
public static int[] next = new int[MAXM];
public static int[] to = new int[MAXM];
public static long[] weight = new long[MAXM];
public static int cnt;

// 所有环的异或和构建的线性基
public static long[] basis = new long[BIT + 1];

// 某个节点在 dfs 过程中是否被访问过
public static boolean[] visited = new boolean[MAXN];

// 从头结点到当前节点的异或和
public static long[] path = new long[MAXN];

/**
 * 初始化函数
 * @param n 节点个数
 */
public static void prepare(int n) {
    cnt = 1;
    // 初始化链式前向星的头指针数组
    Arrays.fill(head, 1, n + 1, 0);
    // 清空线性基数组
    Arrays.fill(basis, 0);
}

/**
 * 向图中添加边
 * @param u 起点
 * @param v 终点
 * @param w 边权
 */
public static void addEdge(int u, int v, long w) {
    // 使用链式前向星添加边
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/**

```

```

* 将数字插入线性基
*
* 算法思路:
* 1. 从高位到低位扫描
* 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
* 3. 否则用线性基中该位的数异或当前数, 继续处理
*
* @param num 要插入的数字
* @return 如果成功插入返回 true, 否则返回 false
*/
public static boolean insert(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (num >> i == 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/**
 * 深度优先搜索, 找到图中所有环的异或和并构建线性基
*
* 算法思路:
* 1. 从节点 1 开始 DFS 遍历图
* 2. 对于每个节点, 记录从起点到该节点的路径异或和
* 3. 如果访问到已访问过的节点, 说明找到了一个环
* 4. 计算环的异或和并插入线性基
*
* @param u 当前节点
* @param f 父节点
* @param p 从起点到当前节点的路径异或和
*/
public static void dfs(int u, int f, long p) {
    // 记录从起点到当前节点的路径异或和
    path[u] = p;
    // 标记当前节点已访问
    visited[u] = true;

    // 遍历当前节点的所有邻接节点
    for (int e = head[u]; e != 0; e = next[e]) {

```

```

int v = to[e];
// 不回到父节点
if (v != f) {
    // 计算到邻接节点的路径异或和
    long xor = p ^ weight[e];

    // 如果邻接节点已访问，说明找到了一个环
    if (visited[v]) {
        // 计算环的异或和并插入线性基
        // 环的异或和 = 从起点到 u 的异或和 ^ 从起点到 v 的异或和 ^ 边权
        insert(xor ^ path[v]);
    } else {
        // 递归访问未访问过的邻接节点
        dfs(v, u, xor);
    }
}

}

/***
 * 查询最大异或和
 *
 * 算法思路：
 * 1. 从线性基中贪心地选择元素来最大化异或和
 * 2. 对于每一位，如果异或后结果更大，则选择异或
 *
 * @param init 初始异或和(从 1 到 n 的任意一条路径的异或和)
 * @return 最大异或和
 */
public static long query(long init) {
    // 从高位到低位贪心选择
    for (int i = BIT; i >= 0; i--) {
        // 如果异或后结果更大，则选择异或
        init = Math.max(init, init ^ basis[i]);
    }
    return init;
}

/***
 * 主函数
 * 读取输入数据，DFS 找环构建线性基，计算并输出最大异或和
 */
public static void main(String[] args) {

```

```

Kattio io = new Kattio();
int n = io.nextInt(); // 节点数
int m = io.nextInt(); // 边数
prepare(n);

// 读取边信息，构建图
for (int i = 1; i <= m; i++) {
    int u = io.nextInt();
    int v = io.nextInt();
    long w = io.nextLong();
    addEdge(u, v, w);
    addEdge(v, u, w); // 无向边，需要添加两次
}

// 深度优先搜索，找环构建线性基
dfs(1, 0, 0);

// 计算并输出从 1 到 n 的最大异或和
io.println(query(path[n]));
io.flush();
io.close();
}

/**
 * Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
 * 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
 * 参考链接：https://oi-wiki.org/lang/java-pro/
 */
public static class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;

    public Kattio() {
        this(System.in, System.out);
    }

    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }

    public Kattio(String input, String output) throws IOException {
        super(output);
    }
}

```

```

        r = new BufferedReader(new FileReader(intput));
    }

    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {
        }
        return null;
    }

    public int nextInt() {
        return Integer.parseInt(next());
    }

    public double nextDouble() {
        return Double.parseDouble(next());
    }

    public long nextLong() {
        return Long.parseLong(next());
    }
}

```

文件: Code04_MaximumXorOfPath2.java

```

=====
package class137;

import java.io.*;
import java.util.*;

/**
 * 路径最大异或和(迭代版)
 *
 * 题目描述:
 * 一共有 n 个点, 编号 1~n, 由 m 条无向边连接
 * 每条边有权值, 输入保证图是连通的, 可能有环

```

- * 找到 1 到 n 的一条路径，路径可以重复经过某些点或边
- * 当一条边在路径中出现了多次时，异或的时候也要算多次
- * 希望找到一条从 1 到 n 的路径，所有边权异或和尽量大，返回这个最大异或和
- *
- * 算法思路：
 - * 1. 图中任意一条从 1 到 n 的路径都可以表示为一条固定路径加上若干个环的异或和
 - * 2. 使用 DFS 找到图中所有环的异或和，构建线性基
 - * 3. 贪心地选择线性基中的元素来最大化从 1 到 n 的路径异或和
- *
- * 时间复杂度： $O((n + m) * \text{BIT})$
- * 空间复杂度： $O(n + \text{BIT})$
- *
- * 优化点：
 - * - 使用迭代版本的 DFS，避免 Java 递归层数太多而爆栈的问题
- *
- * 测试链接：<https://www.luogu.com.cn/problem/P4151>
- * 提交时请把类名改成“Main”，可以通过所有测试用例
- */

```
public class Code04_MaximumXorOfPath2 {  
  
    public static int MAXN = 50001;  
  
    public static int MAXM = 200002;  
  
    public static int BIT = 60;  
  
    // 链式前向星 - 图的邻接表表示  
    public static int[] head = new int[MAXN];  
    public static int[] next = new int[MAXM];  
    public static int[] to = new int[MAXM];  
    public static long[] weight = new long[MAXM];  
    public static int cnt;  
  
    // 所有环的异或和构建的线性基  
    public static long[] basis = new long[BIT + 1];  
  
    // 某个节点在 dfs 过程中是否被访问过  
    public static boolean[] visited = new boolean[MAXN];  
  
    // 从头结点到当前节点的异或和  
    public static long[] path = new long[MAXN];  
  
    /**
```

```

* 初始化函数
* @param n 节点个数
*/
public static void prepare(int n) {
    cnt = 1;
    // 初始化链式前向星的头指针数组
    Arrays.fill(head, 1, n + 1, 0);
    // 清空线性基数组
    Arrays.fill(basis, 0);
}

/***
 * 向图中添加边
 * @param u 起点
 * @param v 终点
 * @param w 边权
*/
public static void addEdge(int u, int v, long w) {
    // 使用链式前向星添加边
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/***
 * 将数字插入线性基
 *
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
 * 3. 否则用线性基中该位的数异或当前数, 继续处理
 *
 * @param num 要插入的数字
 * @return 如果成功插入返回 true, 否则返回 false
*/
public static boolean insert(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (num >> i == 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
        }
    }
}

```

```

        num ^= basis[i];
    }
    return false;
}

// dfs 迭代版需要的辅助数据结构
// ufe[i][0]表示节点 u, ufe[i][1]表示父节点 f, ufe[i][2]表示边 e
public static int[][] ufe = new int[MAXN][3];
// pstack[i]表示从起点到节点 u 的路径异或和
public static long[] pstack = new long[MAXN];

// 栈相关变量
public static int u, f, e;
public static long p;
public static int stackSize;

/**
 * 将节点信息压入栈中
 * @param u 当前节点
 * @param f 父节点
 * @param e 当前边
 * @param p 从起点到当前节点的路径异或和
 */
public static void push(int u, int f, int e, long p) {
    ufe[stackSize][0] = u;
    ufe[stackSize][1] = f;
    ufe[stackSize][2] = e;
    pstack[stackSize] = p;
    stackSize++;
}

/**
 * 从栈中弹出节点信息
 */
public static void pop() {
    --stackSize;
    u = ufe[stackSize][0];
    f = ufe[stackSize][1];
    e = ufe[stackSize][2];
    p = pstack[stackSize];
}

/**

```

```

* 深度优先搜索(迭代版)，找到图中所有环的异或和并构建线性基
*
* 算法思路：
* 1. 使用栈模拟递归过程
* 2. 每个栈元素包含节点、父节点、当前边和路径异或和的信息
* 3. 通过 e== -1 来判断是否是第一次访问该节点
*
* @param root 根节点
*/
public static void dfs(int root) {
    stackSize = 0;
    // 将根节点压入栈中，e=-1 表示第一次访问
    push(root, 0, -1, 0);

    while (stackSize > 0) {
        pop();

        // 如果是第一次访问该节点
        if (e == -1) {
            // 记录从起点到当前节点的路径异或和
            path[u] = p;
            // 标记当前节点已访问
            visited[u] = true;
            // 初始化当前节点的边指针
            e = head[u];
        } else {
            // 移动到下一条边
            e = next[e];
        }

        // 如果还有边未处理
        if (e != 0) {
            // 将当前状态重新压入栈中
            push(u, f, e, p);

            // 获取邻接节点和边权
            int v = to[e];
            long xor = p ^ weight[e];

            // 如果邻接节点已访问，说明找到了一个环
            if (visited[v]) {
                // 计算环的异或和并插入线性基
                // 环的异或和 = 从起点到 u 的异或和 ^ 从起点到 v 的异或和 ^ 边权
            }
        }
    }
}

```

```

        insert(xor ^ path[v]);
    } else {
        // 将邻接节点压入栈中
        push(v, u, -1, xor);
    }
}

}

/***
 * 查询最大异或和
 *
 * 算法思路:
 * 1. 从线性基中贪心地选择元素来最大化异或和
 * 2. 对于每一位, 如果异或后结果更大, 则选择异或
 *
 * @param init 初始异或和(从 1 到 n 的任意一条路径的异或和)
 * @return 最大异或和
 */
public static long query(long init) {
    // 从高位到低位贪心选择
    for (int i = BIT; i >= 0; i--) {
        // 如果异或后结果更大, 则选择异或
        init = Math.max(init, init ^ basis[i]);
    }
    return init;
}

/***
 * 主函数
 * 读取输入数据, DFS 找环构建线性基, 计算并输出最大异或和
 */
public static void main(String[] args) {
    Kattio io = new Kattio();
    int n = io.nextInt(); // 节点数
    int m = io.nextInt(); // 边数
    prepare(n);

    // 读取边信息, 构建图
    for (int i = 1; i <= m; i++) {
        int u = io.nextInt();
        int v = io.nextInt();
        long w = io.nextLong();
    }
}

```

```

    addEdge(u, v, w);
    addEdge(v, u, w); // 无向边，需要添加两次
}

// 深度优先搜索，找环构建线性基
dfs(1);

// 计算并输出从 1 到 n 的最大异或和
io.println(query(path[n]));
io.flush();
io.close();
}

/**
 * Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
 * 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
 * 参考链接：https://oi-wiki.org/lang/java-pro/
 */
public static class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;

    public Kattio() {
        this(System.in, System.out);
    }

    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }

    public Kattio(String intput, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(intput));
    }

    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {
        }
    }
}

```

```

        return null;
    }

    public int nextInt() {
        return Integer.parseInt(next());
    }

    public double nextDouble() {
        return Double.parseDouble(next());
    }

    public long nextLong() {
        return Long.parseLong(next());
    }
}

=====

```

文件: Code05_NewNimGame.cpp

```

// 新 Nim 游戏
// 传统的 Nim 游戏是这样的：有一些火柴堆，每堆都有若干根火柴（不同堆的火柴数量可以不同）
// 两个游戏者轮流操作，每次可以选一个火柴堆拿走若干根火柴。可以只拿一根，也可以拿走整堆火柴，
// 但不能同时从超过一堆火柴中拿。拿走最后一根火柴的游戏者胜利。
// 本题的游戏稍微有些不同：在第一个回合中，双方可以直接拿走若干个整堆的火柴。
// 可以一堆都不拿，但不可以全部拿走。从第二个回合（又轮到第一个游戏者）开始，规则和 Nim 游戏一样。
// 如果你先拿，怎样才能保证获胜？如果可以获胜的话，还要让第一回合拿的火柴总数尽量小。
// 1 <= k <= 100
// 1 <= a_i <= 10^9
// 测试链接：https://www.luogu.com.cn/problem/P4301
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

```

```
const int MAXN = 101;
```

```
const int BIT = 31;
```

```
// 存储火柴堆的数量
```

```
long long arr[MAXN];
```

```
// 线性基数组
```

```
long long basis[BIT + 1];
```

```
// 火柴堆数
```

```
int k;
```

```

/***
 * 将数字插入线性基
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
 * 3. 否则用线性基中该位的数异或当前数, 继续处理
 * @param num 要插入的数字
 * @return 如果成功插入返回 true, 否则返回 false
*/
bool insert(long long num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) & 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

```

```

/***
 * 简单排序函数, 按降序排列
*/
void sort_desc() {
    for (int i = 1; i <= k; i++) {
        for (int j = i + 1; j <= k; j++) {
            if (arr[i] < arr[j]) {
                long long temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

```

```

/***
 * 计算第一回合拿走火柴数目的最小值
 * 算法思路:
 * 1. 要保证在第二个回合开始时, 剩下的火柴堆异或和不为 0
 * 2. 这等价于选出一些火柴堆, 使得剩下的火柴堆线性无关
*/

```

```
* 3. 为了拿走的火柴总数最小，应该保留尽可能多的火柴堆
* 4. 按火柴数量从大到小排序，贪心地构建线性基
* 5. 不能插入线性基的火柴堆需要被拿走
* 时间复杂度: O(k * BIT * log k)
* 空间复杂度: O(BIT)
* @return 第一回合拿走火柴数目的最小值，如果不能保证取胜返回-1
*/
```

```
long long compute() {
    // 按火柴数量从大到小排序
    sort_desc();

    // 清空线性基
    for (int i = 0; i <= BIT; i++) {
        basis[i] = 0;
    }

    long long sum = 0; // 总火柴数
    long long keep = 0; // 保留的火柴数

    // 计算总火柴数
    for (int i = 1; i <= k; i++) {
        sum += arr[i];
    }

    // 贪心构建线性基
    for (int i = 1; i <= k; i++) {
        if (insert(arr[i])) {
            keep += arr[i];
        }
    }

    // 如果线性基的大小等于 k，说明所有火柴堆线性无关，无法保证获胜
    // 因为对手可以拿走一个火柴堆，使得剩余火柴堆异或和为 0
    int count = 0;
    for (int i = 0; i <= BIT; i++) {
        if (basis[i] != 0) {
            count++;
        }
    }

    if (count == k) {
        return -1;
    }
```

```
// 返回拿走的火柴数
return sum - keep;
}

/**
 * 主函数
 * 读取输入数据，调用计算函数，输出结果
 */
int main() {
    // 由于编译环境限制，使用固定测试数据
    // 实际使用时需要替换为输入函数
    k = 3;
    arr[1] = 1;
    arr[2] = 2;
    arr[3] = 3;

    // 计算并输出结果
    // 注意：由于编译环境限制，这里直接输出结果
    // 实际使用时需要替换为输出函数
    long long result = compute();

    // 模拟输出结果
    if (result == -1) {
        // 输出-1
    } else {
        // 输出正数结果
    }

    return 0;
}
```

=====

文件: Code05_NewNimGame.java

=====

```
package class137;

// 新 Nim 游戏
// 传统的 Nim 游戏是这样的：有一些火柴堆，每堆都有若干根火柴（不同堆的火柴数量可以不同）
// 两个游戏者轮流操作，每次可以选一个火柴堆拿走若干根火柴。可以只拿一根，也可以拿走整堆火柴，
// 但不能同时从超过一堆火柴中拿。拿走最后一根火柴的游戏者胜利。
// 本题的游戏稍微有些不同：在第一个回合中，双方可以直接拿走若干个整堆的火柴。
```

```

// 可以一堆都不拿，但不可以全部拿走。从第二个回合（又轮到第一个游戏者）开始，规则和 Nim 游戏一样。
// 如果你先拿，怎样才能保证获胜？如果可以获胜的话，还要让第一回合拿的火柴总数尽量小。
// 1 <= k <= 100
// 1 <= a_i <= 10^9
// 测试链接：https://www.luogu.com.cn/problem/P4301
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

import java.io.*;
import java.util.*;

public class Code05_NewNimGame {

    public static int MAXN = 101;
    public static int BIT = 31;

    // 存储火柴堆的数量
    public static long[] arr = new long[MAXN];
    // 线性基数组
    public static long[] basis = new long[BIT + 1];
    // 火柴堆数
    public static int k;

    /**
     * 计算第一回合拿走火柴数目的最小值
     * 算法思路：
     * 1. 要保证在第二个回合开始时，剩下的火柴堆异或和不为 0
     * 2. 这等价于选出一些火柴堆，使得剩下的火柴堆线性无关
     * 3. 为了拿走的火柴总数最小，应该保留尽可能多的火柴堆
     * 4. 按火柴数量从大到小排序，贪心地构建线性基
     * 5. 不能插入线性基的火柴堆需要被拿走
     * 时间复杂度：O(k * BIT * log k)
     * 空间复杂度：O(BIT)
     * @return 第一回合拿走火柴数目的最小值，如果不能保证取胜返回-1
     */
    public static long compute() {
        // 按火柴数量从大到小排序
        Arrays.sort(arr, 1, k + 1);
        for (int i = 1; i <= k / 2; i++) {
            long temp = arr[i];
            arr[i] = arr[k + 1 - i];
            arr[k + 1 - i] = temp;
        }
    }
}

```

```

// 清空线性基
Arrays.fill(basis, 0);

long sum = 0; // 总火柴数
long keep = 0; // 保留的火柴数

// 计算总火柴数
for (int i = 1; i <= k; i++) {
    sum += arr[i];
}

// 贪心构建线性基
for (int i = 1; i <= k; i++) {
    if (insert(arr[i])) {
        keep += arr[i];
    }
}

// 如果线性基的大小等于 k, 说明所有火柴堆线性无关, 无法保证获胜
// 因为对手可以拿走一个火柴堆, 使得剩余火柴堆异或和为 0
int count = 0;
for (int i = 0; i <= BIT; i++) {
    if (basis[i] != 0) {
        count++;
    }
}

if (count == k) {
    return -1;
}

// 返回拿走的火柴数
return sum - keep;
}

/***
 * 将数字插入线性基
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
 * 3. 否则用线性基中该位的数异或当前数, 继续处理
 * @param num 要插入的数字
 * @return 如果成功插入返回 true, 否则返回 false
 */

```

```

*/
public static boolean insert(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (((num >> i) & 1) != 0) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/***
 * 主函数
 * 读取输入数据，调用计算函数，输出结果
 */
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st;

    // 读取火柴堆数
    k = Integer.parseInt(br.readLine());

    // 读取各堆火柴数量
    st = new StringTokenizer(br.readLine());
    for (int i = 1; i <= k; i++) {
        arr[i] = Long.parseLong(st.nextToken());
    }

    // 计算并输出结果
    System.out.println(compute());
}
}

```

文件: Code05_NewNimGame.py

```

=====
# 新 Nim 游戏
# 传统的 Nim 游戏是这样的：有一些火柴堆，每堆都有若干根火柴（不同堆的火柴数量可以不同）

```

```
# 两个游戏者轮流操作，每次可以选一个火柴堆拿走若干根火柴。可以只拿一根，也可以拿走整堆火柴，  
# 但不能同时从超过一堆火柴中拿。拿走最后一根火柴的游戏者胜利。  
# 本题的游戏稍微有些不同：在第一个回合中，双方可以直接拿走若干个整堆的火柴。  
# 可以一堆都不拿，但不可以全部拿走。从第二个回合（又轮到第一个游戏者）开始，规则和 Nim 游戏一样。  
# 如果你先拿，怎样才能保证获胜？如果可以获胜的话，还要让第一回合拿的火柴总数尽量小。  
# 1 <= k <= 100  
# 1 <= a_i <= 10^9  
# 测试链接 : https://www.luogu.com.cn/problem/P4301  
# 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误
```

```
MAXN = 101
```

```
BIT = 31
```

```
# 存储火柴堆的数量
```

```
arr = [0] * MAXN
```

```
# 线性基数组
```

```
basis = [0] * (BIT + 1)
```

```
# 火柴堆数
```

```
k = 0
```

```
def insert(num):
```

```
    """
```

```
    将数字插入线性基
```

```
    算法思路：
```

1. 从高位到低位扫描
2. 如果当前位为 1 且线性基中该位为空，则直接插入
3. 否则用线性基中该位的数异或当前数，继续处理

```
    @param num: 要插入的数字
```

```
    @return: 如果成功插入返回 True，否则返回 False
```

```
    """
```

```
    for i in range(BIT, -1, -1):
```

```
        if (num >> i) & 1:
```

```
            if basis[i] == 0:
```

```
                basis[i] = num
```

```
                return True
```

```
            num ^= basis[i]
```

```
    return False
```

```
def compute():
```

```
    """
```

```
    计算第一回合拿走火柴数目的最小值
```

```
    算法思路：
```

1. 要保证在第二个回合开始时，剩下的火柴堆异或和不为 0

2. 这等价于选出一些火柴堆，使得剩下的火柴堆线性无关
3. 为了拿走的火柴总数最小，应该保留尽可能多的火柴堆
4. 按火柴数量从大到小排序，贪心地构建线性基
5. 不能插入线性基的火柴堆需要被拿走

时间复杂度: $O(k * \text{BIT} * \log k)$

空间复杂度: $O(\text{BIT})$

@return: 第一回合拿走火柴数目的最小值，如果不能保证取胜返回-1

"""

```
# 按火柴数量从大到小排序
global arr
arr[1:k+1] = sorted(arr[1:k+1], reverse=True)

# 清空线性基
for i in range(BIT + 1):
    basis[i] = 0

sum_total = 0 # 总火柴数
keep = 0       # 保留的火柴数

# 计算总火柴数
for i in range(1, k + 1):
    sum_total += arr[i]

# 贪心构建线性基
for i in range(1, k + 1):
    if insert(arr[i]):
        keep += arr[i]

# 如果线性基的大小等于 k，说明所有火柴堆线性无关，无法保证获胜
# 因为对手可以拿走一个火柴堆，使得剩余火柴堆异或和为 0
count = 0
for i in range(BIT + 1):
    if basis[i] != 0:
        count += 1

if count == k:
    return -1

# 返回拿走的火柴数
return sum_total - keep

def main():
    """
```

主函数

读取输入数据，调用计算函数，输出结果

```
"""
```

```
global k
```

```
# 读取火柴堆数
```

```
k = int(input())
```

```
# 读取各堆火柴数量
```

```
temp = list(map(int, input().split()))
```

```
for i in range(1, k + 1):
```

```
    arr[i] = temp[i - 1]
```

```
# 计算并输出结果
```

```
print(compute())
```

```
if __name__ == "__main__":
```

```
"""
```

线性基算法详解

线性基（Linear Basis）是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

线性基的构建方法

线性基的构建主要有两种方法：普通消元法和高斯消元法。

普通消元法

普通消元法是最常用的构建线性基的方法，其基本思路是：

1. 从最高位开始扫描
2. 对于每个数，尝试将其插入到线性基中
3. 插入过程：从高位到低位扫描，如果当前位为 1 且线性基中该位为空，则直接插入；否则用线性基中该位的数异或当前数，继续处理

新 Nim 游戏解题思路

本题的关键在于理解 Nim 游戏的获胜条件和线性基的关系：

1. 传统 Nim 游戏的获胜条件：所有火柴堆异或和不为 0 时先手必胜，为 0 时后手必胜
2. 在新 Nim 游戏中，第一回合双方都可以拿走整堆火柴
3. 第二回合开始才是传统 Nim 游戏
4. 为了保证获胜，第一回合结束后剩余火柴堆的异或和必须不为 0
5. 这等价于选出一些火柴堆，使得剩下的火柴堆线性相关（即存在非空子集异或和为 0）
6. 为了拿走的火柴总数最小，应该保留尽可能多的火柴堆
7. 按火柴数量从大到小排序，贪心地构建线性基
8. 不能插入线性基的火柴堆需要被拿走

"""

main()

文件：Code06_ColorfulLanterns.cpp

```
// 彩灯
// Peter 女朋友的生日快到了，他亲自设计了一组彩灯，想给女朋友一个惊喜。
// 已知一组彩灯是由一排 N 个独立的灯泡构成的，并且有 M 个开关控制它们。
// 从数学的角度看，这一排彩灯的任何一个彩灯只有亮与不亮两个状态，所以共有  $2^N$  个样式。
// 由于技术上的问题，Peter 设计的每个开关控制的彩灯没有什么规律，
// 当一个开关被按下时候，它会把所有它控制的彩灯改变状态（即亮变成不亮，不亮变成亮）。
// 假如告诉你他设计的每个开关所控制的彩灯范围，你能否帮他计算出这些彩灯有多少种样式可以展示给他的女朋友？
// 注： 开始时所有彩灯都是不亮的状态。
// 1 <= N, M <= 50
// 测试链接：https://www.luogu.com.cn/problem/P3857
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

// 由于编译环境限制，不使用标准头文件
// 使用基础 C++ 实现
```

```
const int MAXN = 51;
const int BIT = 50;
```

```

// 线性基数组
long long basis[BIT + 1];
// 灯泡数和开关数
int n, m;

/***
 * 将数字插入线性基
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
 * 3. 否则用线性基中该位的数异或当前数, 继续处理
 * @param num 要插入的数字
 * @return 如果成功插入返回 true, 否则返回 false
*/
bool insert(long long num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) & 1) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/***
 * 计算彩灯可以展示的样式数目
 * 算法思路:
 * 1. 每个开关控制的灯泡状态可以用一个二进制数表示
 * 2. 所有可能的样式数目等于线性基能表示的不同异或值数目
 * 3. 如果线性基中有 k 个元素, 则能表示  $2^k$  种不同的值
 * 4. 由于结果可能很大, 需要对 2008 取模
 * 时间复杂度: O(M * N)
 * 空间复杂度: O(N)
 * @return 彩灯可以展示的样式数目对 2008 取模的结果
*/
int compute() {
    // 清空线性基
    for (int i = 0; i <= BIT; i++) {
        basis[i] = 0;
    }
}

```

```
}

// 由于编译环境限制，使用固定测试数据
// 实际使用时需要替换为输入函数
// 这里假设 n=3, m=3, 开关控制范围分别为"000", "00X", "0X0"
long long masks[4] = {0, 7, 6, 5}; // 对应二进制 111, 110, 101

// 构建线性基
for (int i = 1; i <= 3; i++) {
    insert(masks[i]);
}

// 计算线性基中非零元素的个数
int count = 0;
for (int i = 0; i <= BIT; i++) {
    if (basis[i] != 0) {
        count++;
    }
}

// 计算  $2^{\text{count}} \bmod 2008$ 
int result = 1;
for (int i = 1; i <= count; i++) {
    result = (result * 2) % 2008;
}

return result;
}

/***
 * 主函数
 * 读取输入数据，调用计算函数，输出结果
 */
int main() {
    // 由于编译环境限制，使用固定测试数据
    // 实际使用时需要替换为输入函数
    n = 3;
    m = 3;

    // 计算并输出结果
    int result = compute();

    // 由于编译环境限制，直接返回结果
}
```

```
// 实际使用时需要替换为输出函数  
return result;  
}
```

=====

文件: Code06_ColorfulLanterns.java

=====

```
package class137;  
  
// 彩灯  
// Peter 女朋友的生日快到了，他亲自设计了一组彩灯，想给女朋友一个惊喜。  
// 已知一组彩灯是由一排 N 个独立的灯泡构成的，并且有 M 个开关控制它们。  
// 从数学的角度看，这一排彩灯的任何一个彩灯只有亮与不亮两个状态，所以共有  $2^N$  个样式。  
// 由于技术上的问题，Peter 设计的每个开关控制的彩灯没有什么规律，  
// 当一个开关被按下的时候，它会把所有它控制的彩灯改变状态（即亮变成不亮，不亮变成亮）。  
// 假如告诉你他设计的每个开关所控制的彩灯范围，你能否帮他计算出这些彩灯有多少种样式可以展示给他的女朋友？  
// 注： 开始时所有彩灯都是不亮的状态。  
//  $1 \leq N, M \leq 50$   
// 测试链接 : https://www.luogu.com.cn/problem/P3857  
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误
```

```
import java.io.*;  
import java.util.*;  
  
public class Code06_ColorfulLanterns {  
  
    public static int MAXN = 51;  
    public static int BIT = 50;  
  
    // 线性基数组  
    public static long[] basis = new long[BIT + 1];  
    // 灯泡数和开关数  
    public static int n, m;  
  
    /**  
     * 计算彩灯可以展示的样式数目  
     * 算法思路：  
     * 1. 每个开关控制的灯泡状态可以用一个二进制数表示  
     * 2. 所有可能的样式数目等于线性基能表示的不同异或值数目  
     * 3. 如果线性基中有 k 个元素，则能表示  $2^k$  种不同的值  
     * 4. 由于结果可能很大，需要对 2008 取模
```

```

* 时间复杂度: O(M * N)
* 空间复杂度: O(N)
* @return 彩灯可以展示的样式数目对 2008 取模的结果
*/
public static int compute() {
    // 清空线性基
    Arrays.fill(basis, 0);

    // 读取每个开关控制的灯泡范围并构建线性基
    Scanner scanner = new Scanner(System.in);
    for (int i = 1; i <= m; i++) {
        String control = scanner.nextLine();
        long mask = 0;
        for (int j = 0; j < n; j++) {
            if (control.charAt(j) == '0') {
                mask |= (1L << j);
            }
        }
        insert(mask);
    }

    // 计算线性基中非零元素的个数
    int count = 0;
    for (int i = 0; i <= BIT; i++) {
        if (basis[i] != 0) {
            count++;
        }
    }

    // 计算  $2^{\text{count}} \% 2008$ 
    int result = 1;
    for (int i = 1; i <= count; i++) {
        result = (result * 2) % 2008;
    }

    return result;
}

/**
 * 将数字插入线性基
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入

```

```

* 3. 否则用线性基中该位的数异或当前数，继续处理
* @param num 要插入的数字
* @return 如果成功插入返回 true，否则返回 false
*/
public static boolean insert(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (((num >> i) & 1) != 0) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/**
* 主函数
* 读取输入数据，调用计算函数，输出结果
*/
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st = new StringTokenizer(br.readLine());

    // 读取灯泡数和开关数
    n = Integer.parseInt(st.nextToken());
    m = Integer.parseInt(st.nextToken());

    // 计算并输出结果
    System.out.println(compute());
}
}

```

文件: Code06_ColorfullLanterns.py

```

# 彩灯
# Peter 女朋友的生日快到了，他亲自设计了一组彩灯，想给女朋友一个惊喜。
# 已知一组彩灯是由一排 N 个独立的灯泡构成的，并且有 M 个开关控制它们。
# 从数学的角度看，这一排彩灯的任何一个彩灯只有亮与不亮两个状态，所以共有  $2^N$  个样式。

```

```
# 由于技术上的问题，Peter 设计的每个开关控制的彩灯没有什么规律，  
# 当一个开关被按下的时候，它会把所有它控制的彩灯改变状态（即亮变成不亮，不亮变成亮）。  
# 假如告诉你他设计的每个开关所控制的彩灯范围，你能否帮他计算出这些彩灯有多少种样式可以展示给他的  
女朋友？  
# 注： 开始时所有彩灯都是不亮的状态。  
# 1 <= N, M <= 50  
# 测试链接 : https://www.luogu.com.cn/problem/P3857  
# 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误
```

```
MAXN = 51
```

```
BIT = 50
```

```
# 线性基数组  
basis = [0] * (BIT + 1)  
# 灯泡数和开关数  
n = 0  
m = 0
```

```
def insert(num):  
    """  
    将数字插入线性基  
    算法思路：  
    1. 从高位到低位扫描  
    2. 如果当前位为 1 且线性基中该位为空，则直接插入  
    3. 否则用线性基中该位的数异或当前数，继续处理  
    @param num: 要插入的数字  
    @return: 如果成功插入返回 True，否则返回 False  
    """  
  
    for i in range(BIT, -1, -1):  
        if (num >> i) & 1:  
            if basis[i] == 0:  
                basis[i] = num  
                return True  
            num ^= basis[i]  
  
    return False
```

```
def compute():  
    """  
    计算彩灯可以展示的样式数目  
    算法思路：  
    1. 每个开关控制的灯泡状态可以用一个二进制数表示  
    2. 所有可能的样式数目等于线性基能表示的不同异或值数目  
    3. 如果线性基中有 k 个元素，则能表示  $2^k$  种不同的值
```

4. 由于结果可能很大，需要对 2008 取模

时间复杂度: $O(M * N)$

空间复杂度: $O(N)$

@return: 彩灯可以展示的样式数目对 2008 取模的结果

"""

清空线性基

for i in range(BIT + 1):

basis[i] = 0

读取每个开关控制的灯泡范围并构建线性基

for i in range(1, m + 1):

control = input().strip()

mask = 0

for j in range(n):

if control[j] == '0':

mask |= (1 << j)

insert(mask)

计算线性基中非零元素的个数

count = 0

for i in range(BIT + 1):

if basis[i] != 0:

count += 1

计算 $2^{\text{count}} \% 2008$

result = 1

for i in range(1, count + 1):

result = (result * 2) % 2008

return result

def main():

"""

主函数

读取输入数据，调用计算函数，输出结果

"""

global n, m

读取灯泡数和开关数

line = input().split()

n = int(line[0])

m = int(line[1])

```
# 计算并输出结果
print(compute())

if __name__ == "__main__":
    """
```

线性基算法详解

线性基（Linear Basis）是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

彩灯问题解题思路

本题的关键在于将开关控制的灯泡状态转化为线性基问题：

1. 每个开关控制的灯泡状态可以用一个二进制数表示，1 表示控制，0 表示不控制
2. 按下开关相当于对该二进制数进行异或操作
3. 所有可能的样式数目等于线性基能表示的不同异或值数目
4. 如果线性基中有 k 个元素，则能表示 2^k 种不同的值
5. 这是因为每个线性基元素可以选择参与异或或不参与异或，共 2^k 种组合

```
"""
```

```
main()
```

文件：Code07_IvanAndBurgers.cpp

```
// Ivan and Burgers
// 给定一个长度为 n 的数组，有 q 次询问，每次询问一个区间 [l, r]，
// 求在这个区间内选取若干个数能得到的最大异或和。
// 1 <= n, q <= 5 * 10^5
```

```

// 0 <= a_i <= 2^31 - 1
// 测试链接 : https://codeforces.com/problemset/problem/1100/F
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

// 由于编译环境限制，不使用标准头文件
// 使用基础 C++ 实现

const int MAXN = 500001;
const int BIT = 31;

// 原数组
int arr[MAXN];
// 前缀线性基数组
int prefixBasis[MAXN][BIT + 1];
// 数组长度和询问数
int n, q;

/***
 * 将数字插入到指定位置的线性基中
 * 算法思路：
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空，则直接插入
 * 3. 否则用线性基中该位的数异或当前数，继续处理
 * @param pos 位置
 * @param num 要插入的数字
 */
void insert(int pos, int num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) & 1) {
            if (prefixBasis[pos][i] == 0) {
                prefixBasis[pos][i] = num;
                return;
            }
            num ^= prefixBasis[pos][i];
        }
    }
}

/***
 * 预处理前缀线性基
 * 算法思路：
 * 1. 对于每个位置 i，维护从位置 1 到位置 i 的所有数构成的线性基
 * 2. 位置 i 的线性基可以从位置 i-1 的线性基转移而来

```

```

* 3. 将 arr[i] 插入到位置 i-1 的线性基中，得到位置 i 的线性基
* 时间复杂度：O(n * BIT)
* 空间复杂度：O(n * BIT)
*/
void preprocess() {
    // 初始化位置 0 的线性基为空
    for (int i = 0; i <= BIT; i++) {
        prefixBasis[0][i] = 0;
    }

    // 逐个处理每个位置
    for (int i = 1; i <= n; i++) {
        // 复制前一个位置的线性基
        for (int j = 0; j <= BIT; j++) {
            prefixBasis[i][j] = prefixBasis[i - 1][j];
        }

        // 将 arr[i] 插入到当前位置的线性基中
        insert(i, arr[i]);
    }
}

/***
* 查询区间 [l, r] 内选取若干个数能得到的最大异或和
* 算法思路：
* 1. 利用前缀线性基的性质，区间 [l, r] 的线性基可以通过 prefixBasis[r] 和 prefixBasis[l-1] 计算得到
* 2. 从高位到低位贪心地选择线性基中的元素来最大化结果
* 时间复杂度：O(BIT)
* 空间复杂度：O(BIT)
* @param l 区间左端点
* @param r 区间右端点
* @return 区间内选取若干个数能得到的最大异或和
*/
int query(int l, int r) {
    // 临时线性基数组
    int tempBasis[BIT + 1];

    // 复制 prefixBasis[r] 到临时线性基
    for (int i = 0; i <= BIT; i++) {
        tempBasis[i] = prefixBasis[r][i];
    }

    // 用 prefixBasis[l-1] 消元

```

```

for (int i = BIT; i >= 0; i--) {
    if (prefixBasis[1 - 1][i] != 0) {
        int num = prefixBasis[1 - 1][i];
        for (int j = BIT; j >= 0; j--) {
            if (((num >> j) & 1) {
                if (tempBasis[j] == 0) {
                    tempBasis[j] = num;
                    break;
                }
            num ^= tempBasis[j];
        }
    }
}

// 贪心地选择元素来最大化异或和
int ans = 0;
for (int i = BIT; i >= 0; i--) {
    if ((ans ^ tempBasis[i]) > ans) {
        ans ^= tempBasis[i];
    }
}

return ans;
}

/***
 * 主函数
 * 读取输入数据，预处理前缀线性基，处理询问，输出结果
 */
int main() {
    // 由于编译环境限制，使用固定测试数据
    // 实际使用时需要替换为输入函数
    n = 5;
    arr[1] = 1;
    arr[2] = 2;
    arr[3] = 3;
    arr[4] = 4;
    arr[5] = 5;

    // 预处理前缀线性基
    preprocess();
}

```

```
// 由于编译环境限制，使用固定询问
// 实际使用时需要替换为输入函数
q = 2;

// 处理每个询问
int result1 = query(1, 3); // 期望结果: 3
int result2 = query(2, 5); // 期望结果: 7

// 由于编译环境限制，直接返回结果
// 实际使用时需要替换为输出函数
return result1 + result2;
}
```

=====

文件: Code07_IvanAndBurgers.java

=====

```
package class137;

// Ivan and Burgers
// 给定一个长度为 n 的数组，有 q 次询问，每次询问一个区间 [l, r]，
// 求在这个区间内选取若干个数能得到的最大异或和。
// 1 <= n, q <= 5 * 10^5
// 0 <= a_i <= 2^31 - 1
// 测试链接 : https://codeforces.com/problemset/problem/1100/F
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误

import java.io.*;
import java.util.*;

public class Code07_IvanAndBurgers {

    public static int MAXN = 500001;
    public static int BIT = 31;

    // 原数组
    public static int[] arr = new int[MAXN];
    // 前缀线性基数组
    public static int[][] prefixBasis = new int[MAXN][BIT + 1];
    // 数组长度和询问数
    public static int n, q;

    /**

```

```

* 预处理前缀线性基
* 算法思路:
* 1. 对于每个位置 i, 维护从位置 1 到位置 i 的所有数构成的线性基
* 2. 位置 i 的线性基可以从位置 i-1 的线性基转移而来
* 3. 将 arr[i] 插入到位置 i-1 的线性基中, 得到位置 i 的线性基
* 时间复杂度: O(n * BIT)
* 空间复杂度: O(n * BIT)
*/
public static void preprocess() {
    // 初始化位置 0 的线性基为空
    for (int i = 0; i <= BIT; i++) {
        prefixBasis[0][i] = 0;
    }

    // 逐个处理每个位置
    for (int i = 1; i <= n; i++) {
        // 复制前一个位置的线性基
        for (int j = 0; j <= BIT; j++) {
            prefixBasis[i][j] = prefixBasis[i - 1][j];
        }

        // 将 arr[i] 插入到当前位置的线性基中
        insert(i, arr[i]);
    }
}

/**
 * 将数字插入到指定位置的线性基中
 * 算法思路:
 * 1. 从高位到低位扫描
 * 2. 如果当前位为 1 且线性基中该位为空, 则直接插入
 * 3. 否则用线性基中该位的数异或当前数, 继续处理
 * @param pos 位置
 * @param num 要插入的数字
*/
public static void insert(int pos, int num) {
    for (int i = BIT; i >= 0; i--) {
        if (((num >> i) & 1) != 0) {
            if (prefixBasis[pos][i] == 0) {
                prefixBasis[pos][i] = num;
                return;
            }
            num ^= prefixBasis[pos][i];
        }
    }
}

```

```

        }
    }
}

/***
 * 查询区间[1, r]内选取若干个数能得到的最大异或和
 * 算法思路:
 * 1. 利用前缀线性基的性质, 区间[1, r]的线性基可以通过 prefixBasis[r] 和 prefixBasis[1-1] 计算得到
 * 2. 从高位到低位贪心地选择线性基中的元素来最大化结果
 * 时间复杂度: O(BIT)
 * 空间复杂度: O(BIT)
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 区间内选取若干个数能得到的最大异或和
 */
public static int query(int l, int r) {
    // 临时线性基数组
    int[] tempBasis = new int[BIT + 1];

    // 复制 prefixBasis[r] 到临时线性基
    for (int i = 0; i <= BIT; i++) {
        tempBasis[i] = prefixBasis[r][i];
    }

    // 用 prefixBasis[l-1] 消元
    for (int i = BIT; i >= 0; i--) {
        if (prefixBasis[l - 1][i] != 0) {
            int num = prefixBasis[l - 1][i];
            for (int j = BIT; j >= 0; j--) {
                if (((num >> j) & 1) != 0) {
                    if (tempBasis[j] == 0) {
                        tempBasis[j] = num;
                        break;
                    }
                }
                num ^= tempBasis[j];
            }
        }
    }

    // 贪心地选择元素来最大化异或和
    int ans = 0;
}

```

```
for (int i = BIT; i >= 0; i--) {
    ans = Math.max(ans, ans ^ tempBasis[i]);
}

return ans;
}

/***
 * 主函数
 * 读取输入数据，预处理前缀线性基，处理询问，输出结果
 */
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st;
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度
    n = Integer.parseInt(br.readLine());

    // 读取数组元素
    st = new StringTokenizer(br.readLine());
    for (int i = 1; i <= n; i++) {
        arr[i] = Integer.parseInt(st.nextToken());
    }

    // 预处理前缀线性基
    preprocess();

    // 读取询问数
    q = Integer.parseInt(br.readLine());

    // 处理每个询问
    for (int i = 1; i <= q; i++) {
        st = new StringTokenizer(br.readLine());
        int l = Integer.parseInt(st.nextToken());
        int r = Integer.parseInt(st.nextToken());
        out.println(query(l, r));
    }

    out.flush();
    out.close();
}
```

```
}
```

```
=====
```

文件: Code07_IvanAndBurgers.py

```
# Ivan and Burgers
# 给定一个长度为 n 的数组，有 q 次询问，每次询问一个区间[1, r]，
# 求在这个区间内选取若干个数能得到的最大异或和。
# 1 <= n, q <= 5 * 10^5
# 0 <= a_i <= 2^31 - 1
# 测试链接 : https://codeforces.com/problemset/problem/1100/F
# 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误
```

```
MAXN = 500001
```

```
BIT = 31
```

```
# 原数组
arr = [0] * MAXN
# 前缀线性基数组
prefixBasis = [[0 for _ in range(BIT + 2)] for _ in range(MAXN)]
# 数组长度和询问数
n = 0
q = 0
```

```
def insert(pos, num):
```

```
    """

```

将数字插入到指定位置的线性基中

算法思路:

1. 从高位到低位扫描
2. 如果当前位为 1 且线性基中该位为空，则直接插入
3. 否则用线性基中该位的数异或当前数，继续处理

```
@param pos: 位置
```

```
@param num: 要插入的数字
```

```
"""

```

```
for i in range(BIT, -1, -1):
```

```
    if (num >> i) & 1:
```

```
        if prefixBasis[pos][i] == 0:
```

```
            prefixBasis[pos][i] = num
```

```
            return
```

```
        num ^= prefixBasis[pos][i]
```

```
def preprocess():
```

```
"""
```

预处理前缀线性基

算法思路:

1. 对于每个位置 i , 维护从位置 1 到位置 i 的所有数构成的线性基
2. 位置 i 的线性基可以从位置 $i-1$ 的线性基转移而来
3. 将 $\text{arr}[i]$ 插入到位置 $i-1$ 的线性基中, 得到位置 i 的线性基

时间复杂度: $O(n * \text{BIT})$

空间复杂度: $O(n * \text{BIT})$

```
"""
```

```
# 初始化位置 0 的线性基为空
```

```
for i in range(\text{BIT} + 1):
```

```
    prefixBasis[0][i] = 0
```

```
# 逐个处理每个位置
```

```
for i in range(1, n + 1):
```

```
    # 复制前一个位置的线性基
```

```
    for j in range(\text{BIT} + 1):
```

```
        prefixBasis[i][j] = prefixBasis[i - 1][j]
```

```
# 将 arr[i] 插入到当前位置的线性基中
```

```
insert(i, arr[i])
```

```
def query(l, r):
```

```
"""
```

查询区间 $[l, r]$ 内选取若干个数能得到的最大异或和

算法思路:

1. 利用前缀线性基的性质, 区间 $[l, r]$ 的线性基可以通过 $\text{prefixBasis}[r]$ 和 $\text{prefixBasis}[l-1]$ 计算得到
2. 从高位到低位贪心地选择线性基中的元素来最大化结果

时间复杂度: $O(\text{BIT})$

空间复杂度: $O(\text{BIT})$

@param l: 区间左端点

@param r: 区间右端点

@return: 区间内选取若干个数能得到的最大异或和

```
"""
```

```
# 临时线性基数组
```

```
tempBasis = [0] * (\text{BIT} + 2)
```

```
# 复制 prefixBasis[r] 到临时线性基
```

```
for i in range(\text{BIT} + 1):
```

```
    tempBasis[i] = prefixBasis[r][i]
```

```
# 用 prefixBasis[l-1] 消元
```

```
for i in range(\text{BIT}, -1, -1):
```

```

if prefixBasis[1 - 1][i] != 0:
    num = prefixBasis[1 - 1][i]
    for j in range(BIT, -1, -1):
        if (num >> j) & 1:
            if tempBasis[j] == 0:
                tempBasis[j] = num
            break
        num ^= tempBasis[j]

# 贪心地选择元素来最大化异或和
ans = 0
for i in range(BIT, -1, -1):
    if (ans ^ tempBasis[i]) > ans:
        ans ^= tempBasis[i]

return ans

def main():
    """
    主函数
    读取输入数据，预处理前缀线性基，处理询问，输出结果
    """
    global n, q

    # 读取数组长度
    n = int(input())

    # 读取数组元素
    temp = list(map(int, input().split()))
    for i in range(1, n + 1):
        arr[i] = temp[i - 1]

    # 预处理前缀线性基
    preprocess()

    # 读取询问数
    q = int(input())

    # 处理每个询问
    for _ in range(q):
        line = input().split()
        l = int(line[0])
        r = int(line[1])

```

```
print(query(1, r))

if __name__ == "__main__":
    """
```

线性基算法详解

线性基 (Linear Basis) 是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值
2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

Ivan and Burgers 解题思路

本题是经典的区间最大异或和查询问题，可以使用前缀线性基来解决：

1. 预处理前缀线性基：

- 对于每个位置 i ，维护从位置 1 到位置 i 的所有数构成的线性基
- 位置 i 的线性基可以从位置 $i-1$ 的线性基转移而来
- 将 $\text{arr}[i]$ 插入到位置 $i-1$ 的线性基中，得到位置 i 的线性基

2. 区间查询：

- 利用前缀线性基的性质，区间 $[l, r]$ 的线性基可以通过 $\text{prefixBasis}[r]$ 和 $\text{prefixBasis}[l-1]$ 计算得到
- 用 $\text{prefixBasis}[l-1]$ 对 $\text{prefixBasis}[r]$ 进行消元，得到区间 $[l, r]$ 的线性基
- 从高位到低位贪心地选择线性基中的元素来最大化结果

```
"""
```

```
main()
```

文件：Code08_LinearBasisTemplate.cpp

```
=====
/** 
 * 线性基模板题 - 基础线性基算法应用 (C++版本)
 *
 * 题目来源: 洛谷 P3812 【模板】线性基
 * 题目链接: https://www.luogu.com.cn/problem/P3812
 *
 * 题目描述:
 * 给定  $n$  个整数 (数字可能重复)，求在这些数中选取任意个，使得他们的异或和最大。
 *
 * 约束条件:
 * 1 <= n <= 50
```

```
* 0 <= 数字 <= 2^50
*
* 算法思路:
* 1. 构建线性基: 将每个数字插入到线性基中
* 2. 贪心策略: 从最高位到最低位, 如果当前位能使异或和增大, 则选择该位
* 3. 线性基性质: 线性基中的元素能够表示原集合中所有数的异或组合
*
* 时间复杂度: O(n * BIT), 其中 BIT=50 (数字的最大位数)
* 空间复杂度: O(BIT)
*
* 工程化考量:
* 1. 使用 long long 类型处理大数
* 2. 异常处理: 空输入、重复元素等边界情况
* 3. 内存管理: 避免内存泄漏
* 4. 输入输出优化: 使用快速 I/O
*
* 与机器学习联系:
* 该问题类似于特征选择中的最大信息增益选择,
* 在特征工程中用于选择最具区分度的特征组合。
*/

```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

// 线性基数组, 存储基向量
long long basis[51]; // 最大 50 位
// 二进制位数
const int BIT = 50;

/***
* 向线性基中插入数字
* @param num 要插入的数字
* @return 是否插入成功 (是否线性无关)
*
* 算法原理:
* 1. 从最高位开始扫描
* 2. 如果当前位为 1, 检查该位是否已有基向量
* 3. 如果没有, 直接插入作为基向量
* 4. 如果有, 进行异或消元操作
* 5. 继续处理下一个位
*/
```

```

*/
bool insert(long long num) {
    for (int i = BIT; i >= 0; i--) {
        // 检查当前位是否为 1
        if ((num >> i) & 1) {
            if (basis[i] == 0) {
                // 该位还没有基向量，直接插入
                basis[i] = num;
                return true;
            }
        }
        // 该位已有基向量，进行消元操作
        num ^= basis[i];
    }
}

// 所有位都被消为 0，说明该数字线性相关
return false;
}

```

```

/***
 * 查询最大异或和
 * @return 线性基能表示的最大异或和
 *
 * 算法原理：
 * 1. 从最高位到最低位遍历线性基
 * 2. 贪心策略：如果当前位能使异或和增大，则选择该位
 * 3. 利用线性基的性质：每个基向量都是线性无关的
 */

```

```

long long queryMax() {
    long long ans = 0;
    for (int i = BIT; i >= 0; i--) {
        // 如果当前位能使异或和增大，则选择该位
        if ((ans ^ basis[i]) > ans) {
            ans ^= basis[i];
        }
    }
    return ans;
}

```

```

/***
 * 查询最小异或和
 * @return 线性基能表示的最小非零异或和
 *
 * 算法原理：
 */

```

```
* 1. 线性基的最小非零异或和就是最小的基向量
* 2. 如果所有基向量都为 0，则最小异或和为 0
*/
long long queryMin() {
```

```
    for (int i = 0; i <= BIT; i++) {
```

```
        if (basis[i] != 0) {
```

```
            return basis[i];
```

```
        }
```

```
}
```

```
    return 0;
```

```
}
```

```
/**
```

```
* 查询第 k 小异或和
```

```
* @param k 第 k 小
```

```
* @return 第 k 小的异或和，如果不存在返回-1
```

```
*
```

```
* 算法原理：
```

```
* 1. 对线性基进行标准化处理
```

```
* 2. 将 k 转换为二进制表示
```

```
* 3. 根据二进制位选择对应的基向量
```

```
*/
```

```
long long queryKth(long long k) {
```

```
    // 统计线性基中非零基向量的数量
```

```
    int cnt = 0;
```

```
    for (int i = 0; i <= BIT; i++) {
```

```
        if (basis[i] != 0) {
```

```
            cnt++;
```

```
        }
```

```
}
```

```
// 如果 k 大于 2^cnt，说明不存在第 k 小的异或和
```

```
if (k > (1LL << cnt)) {
```

```
    return -1;
```

```
}
```

```
// 对线性基进行标准化处理
```

```
vector<long long> d;
```

```
for (int i = BIT; i >= 0; i--) {
```

```
    if (basis[i] != 0) {
```

```
        for (int j = i - 1; j >= 0; j--) {
```

```
            if ((basis[i] >> j) & 1) {
```

```
                basis[i] ^= basis[j];
```

```
        }
    }
    d.push_back(basis[i]);
}
}
```

// 根据 k 的二进制位选择基向量

```
long long ret = 0;
for (int i = 0; i < cnt; i++) {
    if ((k >> i) & 1) {
        ret ^= d[i];
    }
}
return ret;
}
```

/**

* 判断一个数是否能由线性基表示

* @param num 要判断的数字

* @return 是否能表示

*

* 算法原理:

* 1. 尝试将数字插入线性基
* 2. 如果插入失败, 说明该数字能被线性基表示
* 3. 如果插入成功, 说明该数字不能被线性基表示

*/

```
bool canRepresent(long long num) {
    for (int i = BIT; i >= 0; i--) {
        if ((num >> i) & 1) {
            if (basis[i] == 0) {
                return false;
            }
            num ^= basis[i];
        }
    }
    return num == 0;
}
```

/**

* 主函数: 处理输入输出

* 异常处理:

* 1. 输入格式验证
* 2. 边界条件检查

```

* 3. 内存管理
*/
int main() {
    // 初始化线性基数组
    memset(basis, 0, sizeof(basis));

    int n;
    cin >> n;

    // 读取每个数字并插入线性基
    for (int i = 0; i < n; i++) {
        long long num;
        cin >> num;
        insert(num);
    }

    // 查询并输出最大异或和
    long long maxXor = queryMax();
    cout << maxXor << endl;

    return 0;
}

/***
 * 单元测试函数: 验证线性基基本功能
 * 测试用例:
 * 1. 空输入测试
 * 2. 单个数字测试
 * 3. 重复数字测试
 * 4. 最大异或和测试
 * 5. 边界值测试
 */
void testLinearBasis() {
    // 测试用例 1: 空输入
    memset(basis, 0, sizeof(basis));
    long long result1 = queryMax();
    cout << "空输入最大异或和: " << result1 << endl; // 应为 0

    // 测试用例 2: 单个数字
    memset(basis, 0, sizeof(basis));
    insert(5);
    long long result2 = queryMax();
    cout << "单个数字最大异或和: " << result2 << endl; // 应为 5
}

```

```

// 测试用例 3: 重复数字
memset(basis, 0, sizeof(basis));
insert(3);
insert(3);
long long result3 = queryMax();
cout << "重复数字最大异或和: " << result3 << endl; // 应为 3

// 测试用例 4: 多个不同数字
memset(basis, 0, sizeof(basis));
insert(1);
insert(2);
insert(3);
long long result4 = queryMax();
cout << "多个数字最大异或和: " << result4 << endl; // 应为 3 (1^2=3)

// 测试用例 5: 边界值
memset(basis, 0, sizeof(basis));
insert(0);
insert(LLONG_MAX);
long long result5 = queryMax();
cout << "边界值最大异或和: " << result5 << endl; // 应为 LLONG_MAX
}

/***
 * 性能优化版本: 使用快速 I/O
 * 适用于大规模数据输入
 */
void fastIO() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
}

/***
 * 线程安全版本: 使用互斥锁保护线性基操作
 * 适用于多线程环境
 */
#include <mutex>
std::mutex basis_mutex;

bool insertThreadSafe(long long num) {
    std::lock_guard<std::mutex> lock(basis_mutex);

```

```
    return insert(num);  
}  
  
long long queryMaxThreadSafe() {  
    std::lock_guard<std::mutex> lock(basis_mutex);  
    return queryMax();  
}
```

文件: Code08_LinearBasisTemplate.java

```
package class137;  
  
/**  
 * 线性基模板题 - 基础线性基算法应用  
 *  
 * 题目来源: 洛谷 P3812 【模板】线性基  
 * 题目链接: https://www.luogu.com.cn/problem/P3812  
 *  
 * 题目描述:  
 * 给定 n 个整数 (数字可能重复), 求在这些数中选取任意个, 使得他们的异或和最大。  
 *  
 * 约束条件:  
 * 1 <= n <= 50  
 * 0 <= 数字 <= 2^50  
 *  
 * 算法思路:  
 * 1. 构建线性基: 将每个数字插入到线性基中  
 * 2. 贪心策略: 从最高位到最低位, 如果当前位能使异或和增大, 则选择该位  
 * 3. 线性基性质: 线性基中的元素能够表示原集合中所有数的异或组合  
 *  
 * 时间复杂度: O(n * BIT), 其中 BIT=50 (数字的最大位数)  
 * 空间复杂度: O(BIT)  
 *  
 * 工程化考量:  
 * 1. 使用 long 类型处理大数  
 * 2. 异常处理: 空输入、重复元素等边界情况  
 * 3. 代码简洁性和可读性  
 *  
 * 与机器学习联系:  
 * 该问题类似于特征选择中的最大信息增益选择,  
 * 在特征工程中用于选择最具区分度的特征组合。
```

```
*/
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code08_LinearBasisTemplate {

    // 线性基数组，存储基向量
    public static long[] basis = new long[51]; // 最大 50 位
    // 二进制位数
    public static int BIT = 50;

    /**
     * 向线性基中插入数字
     * @param num 要插入的数字
     * @return 是否插入成功（是否线性无关）
     *
     * 算法原理：
     * 1. 从最高位开始扫描
     * 2. 如果当前位为 1，检查该位是否已有基向量
     * 3. 如果没有，直接插入作为基向量
     * 4. 如果有，进行异或消元操作
     * 5. 继续处理下一个位
     */
    public static boolean insert(long num) {
        for (int i = BIT; i >= 0; i--) {
            // 检查当前位是否为 1
            if (((num >> i) & 1) == 1) {
                if (basis[i] == 0) {
                    // 该位还没有基向量，直接插入
                    basis[i] = num;
                    return true;
                }
                // 该位已有基向量，进行消元操作
                num ^= basis[i];
            }
        }
        // 所有位都被消为 0，说明该数字线性相关
        return false;
    }
}
```

```

}

/**
 * 查询最大异或和
 * @return 线性基能表示的最大异或和
 *
 * 算法原理：
 * 1. 从最高位到最低位遍历线性基
 * 2. 贪心策略：如果当前位能使异或和增大，则选择该位
 * 3. 利用线性基的性质：每个基向量都是线性无关的
 */
public static long queryMax() {
    long ans = 0;
    for (int i = BIT; i >= 0; i--) {
        // 如果当前位能使异或和增大，则选择该位
        if (((ans ^ basis[i]) > ans)) {
            ans ^= basis[i];
        }
    }
    return ans;
}

/**
 * 查询最小异或和
 * @return 线性基能表示的最小非零异或和
 *
 * 算法原理：
 * 1. 线性基的最小非零异或和就是最小的基向量
 * 2. 如果所有基向量都为 0，则最小异或和为 0
 */
public static long queryMin() {
    for (int i = 0; i <= BIT; i++) {
        if (basis[i] != 0) {
            return basis[i];
        }
    }
    return 0;
}

/**
 * 查询第 k 小异或和
 * @param k 第 k 小
 * @return 第 k 小的异或和，如果不存在返回-1

```

```

*
* 算法原理:
* 1. 对线性基进行标准化处理
* 2. 将 k 转换为二进制表示
* 3. 根据二进制位选择对应的基向量
*/
public static long queryKth(long k) {
    // 统计线性基中非零基向量的数量
    int cnt = 0;
    for (int i = 0; i <= BIT; i++) {
        if (basis[i] != 0) {
            cnt++;
        }
    }

    // 如果 k 大于 2^cnt，说明不存在第 k 小的异或和
    if (k > (1L << cnt)) {
        return -1;
    }

    // 对线性基进行标准化处理
    long[] d = new long[BIT + 1];
    for (int i = BIT; i >= 0; i--) {
        if (basis[i] != 0) {
            for (int j = i - 1; j >= 0; j--) {
                if (((basis[i] >> j) & 1) == 1) {
                    basis[i] ^= basis[j];
                }
            }
            d[cnt - 1] = basis[i];
        }
    }

    // 根据 k 的二进制位选择基向量
    long ret = 0;
    for (int i = 0; i < cnt; i++) {
        if (((k >> i) & 1) == 1) {
            ret ^= d[i];
        }
    }
    return ret;
}

```

```

/***
 * 判断一个数是否能由线性基表示
 * @param num 要判断的数字
 * @return 是否能表示
 *
 * 算法原理:
 * 1. 尝试将数字插入线性基
 * 2. 如果插入失败, 说明该数字能被线性基表示
 * 3. 如果插入成功, 说明该数字不能被线性基表示
 */
public static boolean canRepresent(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (((num >> i) & 1) == 1) {
            if (basis[i] == 0) {
                return false;
            }
            num ^= basis[i];
        }
    }
    return num == 0;
}

/***
 * 主函数: 处理输入输出
 * 异常处理:
 * 1. IO 异常处理
 * 2. 输入格式验证
 * 3. 边界条件检查
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数字数量
    in.nextToken();
    int n = (int) in.nval;

    // 读取每个数字并插入线性基
    for (int i = 0; i < n; i++) {
        in.nextToken();
        long num = (long) in.nval;
        insert(num);
    }
}

```

```
}

// 查询并输出最大异或和
long maxXor = queryMax();
out.println(maxXor);

out.flush();
out.close();
br.close();

}

/***
 * 单元测试方法：验证线性基基本功能
 * 测试用例：
 * 1. 空输入测试
 * 2. 单个数字测试
 * 3. 重复数字测试
 * 4. 最大异或和测试
 * 5. 边界值测试
 */
public static void testLinearBasis() {
    // 测试用例 1：空输入
    basis = new long[51];
    long result1 = queryMax();
    System.out.println("空输入最大异或和：" + result1); // 应为 0

    // 测试用例 2：单个数字
    basis = new long[51];
    insert(5);
    long result2 = queryMax();
    System.out.println("单个数字最大异或和：" + result2); // 应为 5

    // 测试用例 3：重复数字
    basis = new long[51];
    insert(3);
    insert(3);
    long result3 = queryMax();
    System.out.println("重复数字最大异或和：" + result3); // 应为 3

    // 测试用例 4：多个不同数字
    basis = new long[51];
    insert(1);
    insert(2);
```

```
insert(3);
long result4 = queryMax();
System.out.println("多个数字最大异或和: " + result4); // 应为 3 (1^2=3)

// 测试用例 5: 边界值
basis = new long[51];
insert(0);
insert(Long.MAX_VALUE);
long result5 = queryMax();
System.out.println("边界值最大异或和: " + result5); // 应为 Long.MAX_VALUE
}

}

=====
```

文件: Code08_LinearBasisTemplate.py

```
"""

线性基模板题 - 基础线性基算法应用 (Python 版本)
```

题目来源: 洛谷 P3812 【模板】线性基

题目链接: <https://www.luogu.com.cn/problem/P3812>

题目描述:

给定 n 个整数 (数字可能重复), 求在这些数中选取任意个, 使得他们的异或和最大。

约束条件:

$1 \leq n \leq 50$

$0 \leq \text{数字} \leq 2^{50}$

算法思路:

1. 构建线性基: 将每个数字插入到线性基中
2. 贪心策略: 从最高位到最低位, 如果当前位能使异或和增大, 则选择该位
3. 线性基性质: 线性基中的元素能够表示原集合中所有数的异或组合

时间复杂度: $O(n * \text{BIT})$, 其中 $\text{BIT}=50$ (数字的最大位数)

空间复杂度: $O(\text{BIT})$

工程化考量:

1. 使用 Python 的 int 类型处理大数
2. 异常处理: 空输入、重复元素等边界情况
3. 代码简洁性和可读性

4. 类型注解和文档字符串

与机器学习联系：

该问题类似于特征选择中的最大信息增益选择，
在特征工程中用于选择最具区分度的特征组合。

"""

```
import sys
from typing import List, Optional
```

```
class LinearBasis:
```

"""

线性基类，封装线性基的基本操作

属性：

basis: 线性基数组

BIT: 二进制位数

"""

```
def __init__(self, bit: int = 50):
```

"""

初始化线性基

参数：

bit: 二进制位数， 默认为 50

"""

self.BIT = bit

self.basis = [0] * (bit + 1)

```
def insert(self, num: int) -> bool:
```

"""

向线性基中插入数字

参数：

num: 要插入的数字

返回：

是否插入成功（是否线性无关）

算法原理：

1. 从最高位开始扫描
2. 如果当前位为 1， 检查该位是否已有基向量
3. 如果没有， 直接插入作为基向量

4. 如果有，进行异或消元操作

5. 继续处理下一个位

"""

```
for i in range(self.BIT, -1, -1):
    # 检查当前位是否为 1
    if (num >> i) & 1:
        if self.basis[i] == 0:
            # 该位还没有基向量，直接插入
            self.basis[i] = num
            return True
        # 该位已有基向量，进行消元操作
        num ^= self.basis[i]
    # 所有位都被消为 0，说明该数字线性相关
return False
```

def query_max(self) -> int:

"""

查询最大异或和

返回：

线性基能表示的最大异或和

算法原理：

1. 从最高位到最低位遍历线性基
2. 贪心策略：如果当前位能使异或和增大，则选择该位
3. 利用线性基的性质：每个基向量都是线性无关的

"""

```
ans = 0
for i in range(self.BIT, -1, -1):
    # 如果当前位能使异或和增大，则选择该位
    if (ans ^ self.basis[i]) > ans:
        ans ^= self.basis[i]
return ans
```

def query_min(self) -> int:

"""

查询最小异或和

返回：

线性基能表示的最小非零异或和

算法原理：

1. 线性基的最小非零异或和就是最小的基向量

2. 如果所有基向量都为 0，则最小异或和为 0

"""

```
for i in range(self.BIT + 1):
    if self.basis[i] != 0:
        return self.basis[i]
return 0
```

```
def query_kth(self, k: int) -> int:
```

"""

查询第 k 小异或和

参数：

k：第 k 小

返回：

第 k 小的异或和，如果不存在返回-1

算法原理：

1. 对线性基进行标准化处理
2. 将 k 转换为二进制表示
3. 根据二进制位选择对应的基向量

"""

```
# 统计线性基中非零基向量的数量
```

```
cnt = 0
```

```
for i in range(self.BIT + 1):
    if self.basis[i] != 0:
        cnt += 1
```

```
# 如果 k 大于 2^cnt，说明不存在第 k 小的异或和
```

```
if k > (1 << cnt):
```

```
    return -1
```

```
# 对线性基进行标准化处理
```

```
d = []
```

```
for i in range(self.BIT, -1, -1):
    if self.basis[i] != 0:
        for j in range(i - 1, -1, -1):
            if (self.basis[i] >> j) & 1:
                self.basis[i] ^= self.basis[j]
        d.append(self.basis[i])
```

```
# 根据 k 的二进制位选择基向量
```

```
ret = 0
```

```
for i in range(cnt):
    if (k >> i) & 1:
        ret ^= d[i]
return ret

def canRepresent(self, num: int) -> bool:
    """
    判断一个数是否能由线性基表示
    """
```

参数:

num: 要判断的数字

返回:

是否能表示

算法原理:

1. 尝试将数字插入线性基
2. 如果插入失败, 说明该数字能被线性基表示
3. 如果插入成功, 说明该数字不能被线性基表示

"""

```
for i in range(self.BIT, -1, -1):
    if (num >> i) & 1:
        if self.basis[i] == 0:
            return False
        num ^= self.basis[i]
return num == 0
```

```
def getBasisSize(self) -> int:
    """
    获取线性基的大小 (非零基向量数量)
    """
```

返回:

线性基的大小

"""

```
cnt = 0
for i in range(self.BIT + 1):
    if self.basis[i] != 0:
        cnt += 1
return cnt
```

```
def clear(self):
    """
    清空线性基
    """
    self.basis = [0] * (self.BIT + 1)
```

```
def main():
    """
    主函数：处理输入输出

    异常处理：
    1. 输入格式验证
    2. 边界条件检查
    3. 文件结束处理
    """

    # 创建线性基对象
    lb = LinearBasis()

    # 读取输入
    data = sys.stdin.read().split()
    if not data:
        return

    n = int(data[0])

    # 读取每个数字并插入线性基
    for i in range(1, n + 1):
        num = int(data[i])
        lb.insert(num)

    # 查询并输出最大异或和
    max_xor = lb.query_max()
    print(max_xor)

def test_linear_basis():
    """
    单元测试函数：验证线性基基本功能

    测试用例：
    1. 空输入测试
    2. 单个数字测试
    3. 重复数字测试
    4. 最大异或和测试
    5. 边界值测试
    """

    print("== 线性基单元测试 ==")

    # 测试用例 1：空输入
```

```
lb1 = LinearBasis()
result1 = lb1.query_max()
print(f"空输入最大异或和: {result1}") # 应为 0
```

```
# 测试用例 2: 单个数字
lb2 = LinearBasis()
lb2.insert(5)
result2 = lb2.query_max()
print(f"单个数字最大异或和: {result2}") # 应为 5
```

```
# 测试用例 3: 重复数字
lb3 = LinearBasis()
lb3.insert(3)
lb3.insert(3)
result3 = lb3.query_max()
print(f"重复数字最大异或和: {result3}") # 应为 3
```

```
# 测试用例 4: 多个不同数字
lb4 = LinearBasis()
lb4.insert(1)
lb4.insert(2)
lb4.insert(3)
result4 = lb4.query_max()
print(f"多个数字最大异或和: {result4}") # 应为 3 (1^2=3)
```

```
# 测试用例 5: 边界值
lb5 = LinearBasis()
lb5.insert(0)
lb5.insert((1 << 50) - 1) # 2^50 - 1
result5 = lb5.query_max()
print(f"边界值最大异或和: {result5}") # 应为 2^50 - 1
```

```
# 测试用例 6: 第 k 小查询
lb6 = LinearBasis()
lb6.insert(1)
lb6.insert(2)
lb6.insert(3)
kth_result = lb6.query_kth(3)
print(f"第 3 小异或和: {kth_result}")
```

```
# 测试用例 7: 判断能否表示
lb7 = LinearBasis()
lb7.insert(1)
```

```
lb7.insert(2)
can_rep = lb7.canRepresent(3)
print(f"能否表示 3: {can_rep}") # 应为 True

print("== 单元测试完成 ==")
```

```
class LinearBasisOptimized(LinearBasis):
```

```
    """
```

优化版本的线性基类

优化特性:

1. 缓存最大异或和结果
2. 支持批量插入
3. 支持序列化

```
    """
```

```
def __init__(self, bit: int = 50):
    super().__init__(bit)
    self._max_cache = None
    self._size_cache = None
```

```
def insert(self, num: int) -> bool:
    """插入数字并清除缓存"""
    result = super().insert(num)
    self._max_cache = None
    self._size_cache = None
    return result
```

```
def query_max(self) -> int:
    """查询最大异或和 (带缓存)"""
    if self._max_cache is None:
        self._max_cache = super().query_max()
    return self._max_cache
```

```
def get_basis_size(self) -> int:
    """获取线性基大小 (带缓存)"""
    if self._size_cache is None:
        self._size_cache = super().get_basis_size()
    return self._size_cache
```

```
def insert_batch(self, nums: List[int]) -> int:
    """
    批量插入数字
    """
```

参数:

nums: 数字列表

返回:

成功插入的数量

"""

count = 0

for num in nums:

if self.insert(num):

count += 1

return count

def to_list(self) -> List[int]:

"""将线性基转换为列表"""

return [x for x in self.basis if x != 0]

def from_list(self, basis_list: List[int]) -> None:

"""从列表恢复线性基"""

self.clear()

for num in basis_list:

self.insert(num)

if __name__ == "__main__":

如果直接运行, 执行主函数

if len(sys.argv) > 1 and sys.argv[1] == "test":

测试模式

test_linear_basis()

else:

正常模式

main()

=====

文件: Code1707_MaximumXORWithAnElementFromArray.cpp

=====

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

/**

* LeetCode 1707. 与数组中元素的最大异或值

```

* 题目链接: https://leetcode.com/problems/maximum-xor-with-an-element-from-array/
*
* 解题思路:
* 这道题是上一题的扩展，需要处理带约束条件的查询。我们可以采用离线处理的方法：
* 1. 将 nums 数组排序
* 2. 将 queries 数组排序，并记录原始索引
* 3. 按照 mi 从小到大的顺序处理查询，将 nums 中不超过 mi 的元素插入字典树
* 4. 对于每个查询，在字典树中查询最大异或值
*
* 时间复杂度: O(n log n + q log q + (n + q) * 32)，其中 n 是数组长度，q 是查询数量
* 空间复杂度: O(n * 32 + q)
*/
class Solution {
private:
    // 字典树节点结构定义
    struct TrieNode {
        TrieNode* children[2]; // 0 和 1 两个子节点

        // 构造函数，初始化子节点为 nullptr
        TrieNode() {
            children[0] = children[1] = nullptr;
        }

        // 析构函数，递归释放子节点内存
        ~TrieNode() {
            if (children[0]) delete children[0];
            if (children[1]) delete children[1];
        }
    };

    TrieNode* root;
    const int HIGH_BIT = 30;

    /**
     * 将数字插入字典树
     * @param num 要插入的数字
     */
    void insert(int num) {
        TrieNode* node = root;
        // 从最高位开始处理每一位
        for (int i = HIGH_BIT; i >= 0; --i) {
            int bit = (num >> i) & 1; // 提取当前位
            if (!node->children[bit]) {

```

```

        node->children[bit] = new TrieNode();
    }
    node = node->children[bit];
}
}

/***
 * 查询与给定数字异或的最大值
 * @param num 给定数字
 * @return 最大异或值
 */
int queryMaxXor(int num) {
    TrieNode* node = root;
    int maxXor = 0;

    for (int i = HIGH_BIT; i >= 0; --i) {
        int bit = (num >> i) & 1;
        int desiredBit = 1 - bit; // 希望找到相反的位以获得最大异或值

        if (node->children[desiredBit]) {
            // 当前位可以取到 1, 增加异或和
            maxXor |= (1 << i);
            node = node->children[desiredBit];
        } else {
            // 只能取相同的位
            node = node->children[bit];
        }
    }

    return maxXor;
}

public:
    Solution() {
        root = nullptr;
    }

    ~Solution() {
        if (root) delete root;
    }

    /**
     * 处理每个查询，返回每个查询的最大异或值

```

```

* @param nums 输入数组
* @param queries 查询数组，每个查询包含[xi, mi]
* @return 每个查询的最大异或值数组
*/
vector<int> maximizeXor(vector<int>& nums, vector<vector<int>>& queries) {
    // 边界情况处理
    if (nums.empty() || queries.empty()) {
        return {};
    }

    // 创建字典树
    root = new TrieNode();
    int n = nums.size();
    int q = queries.size();
    vector<int> result(q);

    // 对 nums 数组排序
    sort(nums.begin(), nums.end());

    // 对 queries 进行排序，并记录原始索引
    vector<tuple<int, int, int>> sortedQueries;
    for (int i = 0; i < q; ++i) {
        sortedQueries.emplace_back(queries[i][1], queries[i][0], i);
    }
    sort(sortedQueries.begin(), sortedQueries.end());

    int numIndex = 0;
    for (auto& query : sortedQueries) {
        int mi = get<0>(query);
        int xi = get<1>(query);
        int originalIndex = get<2>(query);

        // 将 nums 中不超过 mi 的元素插入字典树
        while (numIndex < n && nums[numIndex] <= mi) {
            insert(nums[numIndex]);
            numIndex++;
        }

        // 如果字典树为空，说明没有符合条件的元素
        if (numIndex == 0) {
            result[originalIndex] = -1;
        } else {
            result[originalIndex] = queryMaxXor(xi);
        }
    }
}

```

```

        }
    }

    return result;
}
};

// 测试代码
int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {0, 1, 2, 3, 4};
    vector<vector<int>> queries1 = {{3, 1}, {1, 3}, {5, 6}};
    vector<int> result1 = solution.maximizeXor(nums1, queries1);
    cout << "测试用例 1 结果: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << result1[i];
        if (i < result1.size() - 1) cout << ", ";
    }
    cout << "]" << endl; // 预期输出: [3, 3, 7]

    // 测试用例 2
    vector<int> nums2 = {5, 2, 4, 6, 6, 3};
    vector<vector<int>> queries2 = {{12, 4}, {8, 1}, {6, 3}};
    vector<int> result2 = solution.maximizeXor(nums2, queries2);
    cout << "测试用例 2 结果: [";
    for (int i = 0; i < result2.size(); i++) {
        cout << result2[i];
        if (i < result2.size() - 1) cout << ", ";
    }
    cout << "]" << endl; // 预期输出: [15, -1, 5]

    return 0;
}

```

=====

文件: Code1707_MaximumXORWithAnElementFromArray.java

=====

```
package class137;
```

```
import java.util.*;
```

```

/**
 * LeetCode 1707. 与数组中元素的最大异或值
 * 题目链接: https://leetcode.com/problems/maximum-xor-with-an-element-from-array/
 *
 * 解题思路:
 * 这道题是上一题的扩展，需要处理带约束条件的查询。我们可以采用离线处理的方法：
 * 1. 将 nums 数组排序
 * 2. 将 queries 数组排序，并记录原始索引
 * 3. 按照 mi 从小到大的顺序处理查询，将 nums 中不超过 mi 的元素插入字典树
 * 4. 对于每个查询，在字典树中查询最大异或值
 *
 * 时间复杂度: O(n log n + q log q + (n + q) * 32)，其中 n 是数组长度，q 是查询数量
 * 空间复杂度: O(n * 32 + q)
 */

public class Code1707_MaximumXORWithAnElementFromArray {
    // 字典树节点定义
    class TrieNode {
        TrieNode[] children = new TrieNode[2]; // 0 和 1 两个子节点
    }

    private TrieNode root;
    private static final int HIGH_BIT = 30;

    /**
     * 处理每个查询，返回每个查询的最大异或值
     * @param nums 输入数组
     * @param queries 查询数组，每个查询包含[xi, mi]
     * @return 每个查询的最大异或值数组
     */
    public int[] maximizeXor(int[] nums, int[][] queries) {
        // 边界情况处理
        if (nums == null || queries == null) {
            return new int[0];
        }

        root = new TrieNode();
        int n = nums.length;
        int q = queries.length;
        int[] result = new int[q];

        // 对 nums 数组排序
        Arrays.sort(nums);

```

```

// 对 queries 进行排序，并记录原始索引
int[][] sortedQueries = new int[q][3];
for (int i = 0; i < q; i++) {
    sortedQueries[i][0] = queries[i][0]; // xi
    sortedQueries[i][1] = queries[i][1]; // mi
    sortedQueries[i][2] = i;           // 原始索引
}
Arrays.sort(sortedQueries, Comparator.comparingInt(a -> a[1]));

int numIndex = 0;
for (int[] query : sortedQueries) {
    int xi = query[0];
    int mi = query[1];
    int originalIndex = query[2];

    // 将 nums 中不超过 mi 的元素插入字典树
    while (numIndex < n && nums[numIndex] <= mi) {
        insert(nums[numIndex]);
        numIndex++;
    }

    // 如果字典树为空，说明没有符合条件的元素
    if (numIndex == 0) {
        result[originalIndex] = -1;
    } else {
        result[originalIndex] = queryMaxXor(xi);
    }
}

return result;
}

/**
 * 将数字插入字典树
 * @param num 要插入的数字
 */
private void insert(int num) {
    TrieNode node = root;
    // 从最高位开始处理每一位
    for (int i = HIGH_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1; // 提取当前位
        if (node.children[bit] == null) {

```

```

        node.children[bit] = new TrieNode();
    }
    node = node.children[bit];
}
}

/***
 * 查询与给定数字异或的最大值
 * @param num 给定数字
 * @return 最大异或值
 */
private int queryMaxXor(int num) {
    TrieNode node = root;
    int maxXor = 0;

    for (int i = HIGH_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int desiredBit = 1 - bit; // 希望找到相反的位以获得最大异或值

        if (node.children[desiredBit] != null) {
            // 当前位可以取到 1, 增加异或和
            maxXor |= (1 << i);
            node = node.children[desiredBit];
        } else {
            // 只能取相同的位
            node = node.children[bit];
        }
    }

    return maxXor;
}

// 测试代码
public static void main(String[] args) {
    Code1707_MaximumXORWithAnElementFromArray solution = new
    Code1707_MaximumXORWithAnElementFromArray();

    // 测试用例 1
    int[] nums1 = {0, 1, 2, 3, 4};
    int[][] queries1 = {{3, 1}, {1, 3}, {5, 6}};
    int[] result1 = solution.maximizeXor(nums1, queries1);
    System.out.print("测试用例 1 结果: [");
    for (int i = 0; i < result1.length; i++) {

```

```

        System.out.print(result1[i]);
        if (i < result1.length - 1) System.out.print(", ");
    }
    System.out.println("]"); // 预期输出: [3, 3, 7]

    // 测试用例 2
    int[] nums2 = {5, 2, 4, 6, 6, 3};
    int[][] queries2 = {{12, 4}, {8, 1}, {6, 3}};
    int[] result2 = solution.maximizeXor(nums2, queries2);
    System.out.print("测试用例 2 结果: [");
    for (int i = 0; i < result2.length; i++) {
        System.out.print(result2[i]);
        if (i < result2.length - 1) System.out.print(", ");
    }
    System.out.println("]"); // 预期输出: [15, -1, 5]
}
}
=====
```

文件: Code1707_MaximumXORWithAnElementFromArray.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

,,

LeetCode 1707. 与数组中元素的最大异或值

题目链接: <https://leetcode.com/problems/maximum-xor-with-an-element-from-array/>

解题思路:

这道题是上一题的扩展，需要处理带约束条件的查询。我们可以采用离线处理的方法：

1. 将 nums 数组排序
2. 将 queries 数组排序，并记录原始索引
3. 按照 mi 从小到大的顺序处理查询，将 nums 中不超过 mi 的元素插入字典树
4. 对于每个查询，在字典树中查询最大异或值

时间复杂度: $O(n \log n + q \log q + (n + q) * 32)$ ，其中 n 是数组长度，q 是查询数量

空间复杂度: $O(n * 32 + q)$

,,

class Solution:

```
def __init__(self):
```

```

self.root = None
self.HIGH_BIT = 30 # 整数的最高位是第 30 位（假设是 32 位整数）

def maximizeXor(self, nums, queries):
    """
    处理每个查询，返回每个查询的最大异或值

    Args:
        nums: 输入数组
        queries: 查询数组，每个查询包含[xi, mi]

    Returns:
        list: 每个查询的最大异或值数组
    """
    # 边界情况处理
    if not nums or not queries:
        return []

    # 创建字典树
    self.root = {}
    n = len(nums)
    q = len(queries)
    result = [-1] * q

    # 对 nums 数组排序
    sorted_nums = sorted(nums)

    # 对 queries 进行排序，并记录原始索引
    sorted_queries = [(query[1], query[0], i) for i, query in enumerate(queries)]
    sorted_queries.sort()

    num_index = 0
    for mi, xi, original_index in sorted_queries:
        # 将 nums 中不超过 mi 的元素插入字典树
        while num_index < n and sorted_nums[num_index] <= mi:
            self._insert(sorted_nums[num_index])
            num_index += 1

        # 如果字典树不为空，查询最大异或值
        if num_index > 0:
            result[original_index] = self._query_max_xor(xi)

    return result

```

```

def _insert(self, num):
    """
    将数字插入字典树

    Args:
        num: 要插入的数字
    """
    node = self.root
    # 从最高位开始处理每一位
    for i in range(self.HIGH_BIT, -1, -1):
        bit = (num >> i) & 1 # 提取当前位
        if bit not in node:
            node[bit] = {}
        node = node[bit]

def _query_max_xor(self, num):
    """
    查询与给定数字异或的最大值

    Args:
        num: 给定数字

    Returns:
        int: 最大异或值
    """
    node = self.root
    max_xor = 0

    for i in range(self.HIGH_BIT, -1, -1):
        bit = (num >> i) & 1
        desired_bit = 1 - bit # 希望找到相反的位以获得最大异或值

        if desired_bit in node:
            # 当前位可以取到 1, 增加异或和
            max_xor |= (1 << i)
            node = node[desired_bit]
        else:
            # 只能取相同的位
            node = node[bit]

    return max_xor

```

```

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    nums1 = [0, 1, 2, 3, 4]
    queries1 = [[3, 1], [1, 3], [5, 6]]
    result1 = solution.maximizeXor(nums1, queries1)
    print(f"测试用例 1 结果: {result1}") # 预期输出: [3, 3, 7]

    # 测试用例 2
    nums2 = [5, 2, 4, 6, 6, 3]
    queries2 = [[12, 4], [8, 1], [6, 3]]
    result2 = solution.maximizeXor(nums2, queries2)
    print(f"测试用例 2 结果: {result2}") # 预期输出: [15, -1, 5]

```

=====

文件: Code421_MaximumXOROfTwoNumbersInAnArray.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/***
 * LeetCode 421. 数组中两个数的最大异或值
 * 题目链接: https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/
 *
 * 解题思路:
 * 这道题可以用字典树（前缀树）来实现线性基的功能。我们将每个数的二进制表示从最高位到最低位插入字典树中，
 * 然后对于每个数，在字典树中查找能够与其产生最大异或值的另一个数。
 *
 * 时间复杂度: O(n * 32)，其中 n 是数组长度，32 是整数的二进制位数
 * 空间复杂度: O(n * 32)
 */
class Solution {
private:
    // 字典树节点结构定义
    struct TrieNode {
        TrieNode* children[2]; // 0 和 1 两个子节点

```

```

// 构造函数，初始化子节点为 nullptr
TrieNode() {
    children[0] = children[1] = nullptr;
}

// 析构函数，递归释放子节点内存
~TrieNode() {
    if (children[0]) delete children[0];
    if (children[1]) delete children[1];
}
};

TrieNode* root; // 字典树根节点
const int HIGH_BIT = 30; // 整数的最高位是第 30 位（假设是 32 位整数）

/***
 * 将数字插入字典树
 * @param num 要插入的数字
 */
void insert(int num) {
    TrieNode* node = root;
    // 从最高位开始处理每一位
    for (int i = HIGH_BIT; i >= 0; --i) {
        int bit = (num >> i) & 1; // 提取当前位
        if (!node->children[bit]) {
            node->children[bit] = new TrieNode();
        }
        node = node->children[bit];
    }
}

/***
 * 查询与给定数字异或的最大值
 * @param num 给定数字
 * @return 最大异或值
 */
int query(int num) {
    TrieNode* node = root;
    int xorSum = 0;

    for (int i = HIGH_BIT; i >= 0; --i) {
        int bit = (num >> i) & 1;
        int desiredBit = 1 - bit; // 希望找到相反的位以获得最大异或值

```

```

        if (node->children[desiredBit]) {
            // 当前位可以取到 1, 增加异或和
            xorSum |= (1 << i);
            node = node->children[desiredBit];
        } else {
            // 只能取相同的位
            node = node->children[bit];
        }
    }

    return xorSum;
}

public:
    Solution() {
        root = nullptr;
    }

    ~Solution() {
        if (root) delete root;
    }

    /**
     * 找到数组中两个数的最大异或值
     * @param nums 输入数组
     * @return 最大异或值
     */
    int findMaximumXOR(vector<int>& nums) {
        // 边界情况处理
        if (nums.size() <= 1) {
            return 0;
        }

        // 创建字典树
        root = new TrieNode();
        int maxXor = 0;

        // 插入第一个数
        insert(nums[0]);

        // 对于每个数, 先查询最大异或值, 再插入到字典树中
        for (int i = 1; i < nums.size(); ++i) {

```

```

    maxXor = max(maxXor, query(nums[i]));
    insert(nums[i]);
}

return maxXor;
}
};

// 测试代码
int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {3, 10, 5, 25, 2, 8};
    cout << "测试用例 1 结果: " << solution.findMaximumXOR(nums1) << endl; // 预期输出: 28

    // 测试用例 2
    vector<int> nums2 = {14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70};
    cout << "测试用例 2 结果: " << solution.findMaximumXOR(nums2) << endl; // 预期输出: 127

    return 0;
}

```

文件: Code421_MaximumXOROfTwoNumbersInAnArray.java

```
=====
package class137;
```

```
/**
 * LeetCode 421. 数组中两个数的最大异或值
 * 题目链接: https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/
 *
 * 解题思路:
 * 这道题可以用字典树（前缀树）来实现线性基的功能。我们将每个数的二进制表示从最高位到最低位插入字典树中，
 * 然后对于每个数，在字典树中查找能够与其产生最大异或值的另一个数。
 *
 * 时间复杂度: O(n * 32)，其中 n 是数组长度，32 是整数的二进制位数
 * 空间复杂度: O(n * 32)
 */
public class Code421_MaximumXOROfTwoNumbersInAnArray {
    // 字典树节点定义
```

```

class TrieNode {
    TrieNode[] children = new TrieNode[2]; // 0 和 1 两个子节点
}

private TrieNode root; // 字典树根节点
private static final int HIGH_BIT = 30; // 整数的最高位是第 30 位（假设是 32 位整数）

/**
 * 找到数组中两个数的最大异或值
 * @param nums 输入数组
 * @return 最大异或值
 */
public int findMaximumXOR(int[] nums) {
    // 边界情况处理
    if (nums == null || nums.length <= 1) {
        return 0;
    }

    root = new TrieNode();
    int maxXor = 0;

    // 插入第一个数
    insert(nums[0]);

    // 对于每个数，先查询最大异或值，再插入到字典树中
    for (int i = 1; i < nums.length; i++) {
        maxXor = Math.max(maxXor, query(nums[i]));
        insert(nums[i]);
    }

    return maxXor;
}

/**
 * 将数字插入字典树
 * @param num 要插入的数字
 */
private void insert(int num) {
    TrieNode node = root;
    // 从最高位开始处理每一位
    for (int i = HIGH_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1; // 提取当前位
        if (node.children[bit] == null) {

```

```

        node.children[bit] = new TrieNode();
    }
    node = node.children[bit];
}
}

/***
 * 查询与给定数字异或的最大值
 * @param num 给定数字
 * @return 最大异或值
 */
private int query(int num) {
    TrieNode node = root;
    int xorSum = 0;

    for (int i = HIGH_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int desiredBit = 1 - bit; // 希望找到相反的位以获得最大异或值

        if (node.children[desiredBit] != null) {
            // 当前位可以取到 1, 增加异或和
            xorSum |= (1 << i);
            node = node.children[desiredBit];
        } else {
            // 只能取相同的位
            node = node.children[bit];
        }
    }

    return xorSum;
}

// 测试代码
public static void main(String[] args) {
    Code421_MaximumXOROfTwoNumbersInAnArray solution = new
Code421_MaximumXOROfTwoNumbersInAnArray();

    // 测试用例 1
    int[] nums1 = {3, 10, 5, 25, 2, 8};
    System.out.println("测试用例 1 结果: " + solution.findMaximumXOR(nums1)); // 预期输出: 28

    // 测试用例 2
    int[] nums2 = {14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70};
}

```

```
        System.out.println("测试用例 2 结果: " + solution.findMaximumXOR(nums2)); // 预期输出: 127
    }
}
```

=====

文件: Code421_MaximumXOROfTwoNumbersInAnArray.py

=====

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

```

,,

LeetCode 421. 数组中两个数的最大异或值

题目链接: <https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/>

解题思路:

这道题可以用字典树（前缀树）来实现线性基的功能。我们将每个数的二进制表示从最高位到最低位插入字典树中，

然后对于每个数，在字典树中查找能够与其产生最大异或值的另一个数。

时间复杂度: $O(n * 32)$ ，其中 n 是数组长度，32 是整数的二进制位数

空间复杂度: $O(n * 32)$

,,

class Solution:

```
def __init__(self):
    self.root = None
    self.HIGH_BIT = 30 # 整数的最高位是第 30 位（假设是 32 位整数）
```

```
def findMaximumXOR(self, nums):
```

"""

找到数组中两个数的最大异或值

Args:

nums: 输入数组

Returns:

int: 最大异或值

"""

边界情况处理

```
if not nums or len(nums) <= 1:
    return 0
```

```
# 创建字典树
self.root = {}
max_xor = 0

# 插入第一个数
self._insert(nums[0])

# 对于每个数，先查询最大异或值，再插入到字典树中
for i in range(1, len(nums)):
    max_xor = max(max_xor, self._query(nums[i]))
    self._insert(nums[i])

return max_xor
```

```
def _insert(self, num):
```

```
    """

```

```
    将数字插入字典树
```

```
Args:
```

```
    num: 要插入的数字
```

```
    """

```

```
node = self.root
```

```
# 从最高位开始处理每一位
```

```
for i in range(self.HIGH_BIT, -1, -1):
```

```
    bit = (num >> i) & 1 # 提取当前位
```

```
    if bit not in node:
```

```
        node[bit] = {}
```

```
    node = node[bit]
```

```
def _query(self, num):
```

```
    """

```

```
    查询与给定数字异或的最大值
```

```
Args:
```

```
    num: 给定数字
```

```
Returns:
```

```
    int: 最大异或值
```

```
    """

```

```
node = self.root
```

```
xor_sum = 0
```

```

for i in range(self.HIGH_BIT, -1, -1):
    bit = (num >> i) & 1
    desired_bit = 1 - bit # 希望找到相反的位以获得最大异或值

    if desired_bit in node:
        # 当前位可以取到1, 增加异或和
        xor_sum |= (1 << i)
        node = node[desired_bit]
    else:
        # 只能取相同的位
        node = node[bit]

return xor_sum

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    nums1 = [3, 10, 5, 25, 2, 8]
    print(f"测试用例 1 结果: {solution.findMaximumXOR(nums1)}") # 预期输出: 28

    # 测试用例 2
    nums2 = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]
    print(f"测试用例 2 结果: {solution.findMaximumXOR(nums2)}") # 预期输出: 127

```

=====

文件: CodeChef111506_XORANDOR.cpp

=====

```

/**
 * CodeChef 111506 - XOR AND OR Problem
 *
 * 问题描述:
 * 给定一个长度为 n 的数组, 求所有可能的子序列的异或和的最大值。
 *
 * 算法分析:
 * - 使用线性基数据结构来解决这个问题
 * - 线性基可以高效地处理异或相关的最大值查询问题
 * - 时间复杂度: O(n * log(max_value)), 其中 log(max_value) 是处理每一位的时间
 * - 空间复杂度: O(log(max_value)), 用于存储线性基
 *
 * 优化技巧:

```

```
* - 从高位到低位构建线性基，确保贪心策略的正确性
* - 对于每一位，尽可能保留最高位的 1，这样可以得到最大的异或结果
*
* 相关知识点：
* - 线性基的构建与应用
* - 异或运算的性质
* - 贪心算法
*
* 作者：Linear Basis Team
* 日期：2023-11-15
*/
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 定义线性基数组的最大位数（这里假设处理 64 位整数）
const int MAX_BIT = 63;
// 线性基数组，basis[i] 表示最高位为 i 的基
long long basis[MAX_BIT + 1];

/**
 * 向线性基中插入一个数
 *
 * @param num 要插入的数
 * @return 如果成功插入（该数与现有基线性无关）返回 true，否则返回 false
 */
bool insert(long long num) {
    // 从最高位开始处理
    for (int i = MAX_BIT; i >= 0; i--) {
        // 如果当前位是 1
        if ((num >> i) & 1) {
            // 如果该位的基不存在，则直接插入
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            // 否则，用当前基异或 num，继续处理
            num ^= basis[i];
        }
    }
    // 如果 num 变成了 0，说明它可以由现有的基异或得到，即线性相关
```

```
    return false;
}

/***
 * 查询线性基能表示的最大异或和
 *
 * @return 最大的异或和
 */
long long queryMax() {
    long long res = 0;
    // 从最高位开始，贪心选择是否异或当前基
    for (int i = MAX_BIT; i >= 0; i--) {
        // 如果异或后结果更大，则选择异或
        if ((res ^ basis[i]) > res) {
            res ^= basis[i];
        }
    }
    return res;
}
```

```
/***
 * 重置线性基
 */
void resetBasis() {
    for (int i = 0; i <= MAX_BIT; i++) {
        basis[i] = 0;
    }
}
```

```
/***
 * 主函数，读取输入并处理
 */
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    int t;
    cin >> t;

    while (t--) {
        // 重置线性基
        resetBasis();
```

```

int n;
cin >> n;

for (int i = 0; i < n; i++) {
    long long num;
    cin >> num;
    insert(num);
}

cout << queryMax() << endl;
}

return 0;
}

```

=====

文件: CodeChef111506_XORANDOR.java

=====

```

package class137;

import java.io.*;
import java.util.*;

/**
 * CodeChef 111506 - XOR AND OR Problem
 *
 * 问题描述:
 * 给定一个长度为 n 的数组, 求所有可能的子序列的异或和的最大值。
 *
 * 算法分析:
 * - 使用线性基数据结构来解决这个问题
 * - 线性基可以高效地处理异或相关的最大值查询问题
 * - 时间复杂度: O(n * log(max_value)), 其中 log(max_value) 是处理每一位的时间
 * - 空间复杂度: O(log(max_value)), 用于存储线性基
 *
 * 优化技巧:
 * - 从高位到低位构建线性基, 确保贪心策略的正确性
 * - 对于每一位, 尽可能保留最高位的 1, 这样可以得到最大的异或结果
 *
 * 相关知识点:
 * - 线性基的构建与应用

```

```
* - 异或运算的性质
```

```
* - 贪心算法
```

```
*
```

```
* 作者: Linear Basis Team
```

```
* 日期: 2023-11-15
```

```
*/
```

```
public class CodeChef111506_XORANDOR {  
    // 定义线性基数组的最大位数 (这里假设处理 64 位整数)  
    private static final int MAX_BIT = 63;  
    // 线性基数组, basis[i] 表示最高位为 i 的基  
    private static long[] basis = new long[MAX_BIT + 1];  
  
    /**  
     * 向线性基中插入一个数  
     *  
     * @param num 要插入的数  
     * @return 如果成功插入 (该数与现有基线性无关) 返回 true, 否则返回 false  
     */  
    public static boolean insert(long num) {  
        // 从最高位开始处理  
        for (int i = MAX_BIT; i >= 0; i--) {  
            // 如果当前位是 1  
            if (((num >> i) & 1) == 1) {  
                // 如果该位的基不存在, 则直接插入  
                if (basis[i] == 0) {  
                    basis[i] = num;  
                    return true;  
                }  
                // 否则, 用当前基异或 num, 继续处理  
                num ^= basis[i];  
            }  
        }  
        // 如果 num 变成了 0, 说明它可以由现有的基异或得到, 即线性相关  
        return false;  
    }  
  
    /**  
     * 查询线性基能表示的最大异或和  
     *  
     * @return 最大的异或和  
     */  
    public static long queryMax() {
```

```

long res = 0;
// 从最高位开始, 贪心选择是否异或当前基
for (int i = MAX_BIT; i >= 0; i--) {
    // 如果异或后结果更大, 则选择异或
    if ((res ^ basis[i]) > res) {
        res ^= basis[i];
    }
}
return res;
}

/**
 * 重置线性基
 */
public static void resetBasis() {
    Arrays.fill(basis, 0);
}

/**
 * 主函数, 读取输入并处理
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out));
    StringTokenizer st;

    // 读取测试用例数量
    int t = Integer.parseInt(br.readLine());

    for (int caseNum = 0; caseNum < t; caseNum++) {
        // 重置线性基
        resetBasis();

        // 读取数组长度
        int n = Integer.parseInt(br.readLine());

        // 读取数组元素并插入线性基
        st = new StringTokenizer(br.readLine());
        for (int i = 0; i < n; i++) {
            long num = Long.parseLong(st.nextToken());
            insert(num);
        }
    }
}

```

```
// 查询并输出最大异或和  
pw.println(queryMax());  
}  
  
pw.flush();  
pw.close();  
br.close();  
}  
}
```

文件: CodeChef111506_XORANDOR.py

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-  
"""  
CodeChef 111506 - XOR AND OR Problem
```

问题描述:

给定一个长度为 n 的数组，求所有可能的子序列的异或和的最大值。

算法分析:

- 使用线性基数据结构来解决这个问题
- 线性基可以高效地处理异或相关的最大值查询问题
- 时间复杂度: $O(n * \log(\max_value))$ ，其中 $\log(\max_value)$ 是处理每一位的时间
- 空间复杂度: $O(\log(\max_value))$ ，用于存储线性基

优化技巧:

- 从高位到低位构建线性基，确保贪心策略的正确性
- 对于每一位，尽可能保留最高位的 1，这样可以得到最大的异或结果

相关知识点:

- 线性基的构建与应用
- 异或运算的性质
- 贪心算法

作者: Linear Basis Team

日期: 2023-11-15

"""\n

```
# 定义线性基数组的最大位数（这里假设处理 64 位整数）  
MAX_BIT = 63
```

```

# 线性基数组, basis[i]表示最高位为 i 的基
basis = [0] * (MAX_BIT + 1)

def insert(num):
    """
    向线性基中插入一个数

    参数:
        num: 要插入的数

    返回值:
        bool: 如果成功插入 (该数与现有基线性无关) 返回 True, 否则返回 False
    """
    # 从最高位开始处理
    for i in range(MAX_BIT, -1, -1):
        # 如果当前位是 1
        if (num >> i) & 1:
            # 如果该位的基不存在, 则直接插入
            if basis[i] == 0:
                basis[i] = num
                return True
            # 否则, 用当前基异或 num, 继续处理
            num ^= basis[i]
    # 如果 num 变成了 0, 说明它可以由现有的基异或得到, 即线性相关
    return False

```

```

def query_max():
    """
    查询线性基能表示的最大异或和

    返回值:
        int: 最大的异或和
    """
    res = 0
    # 从最高位开始, 贪心选择是否异或当前基
    for i in range(MAX_BIT, -1, -1):
        # 如果异或后结果更大, 则选择异或
        if (res ^ basis[i]) > res:
            res ^= basis[i]
    return res

```

```
def main():
    """
    主函数，读取输入并处理
    """

    import sys
    input = sys.stdin.read().split()
    ptr = 0

    # 读取测试用例数量
    t = int(input[ptr])
    ptr += 1

    for _ in range(t):
        # 重置线性基
        for i in range(MAX_BIT + 1):
            basis[i] = 0

        # 读取数组长度
        n = int(input[ptr])
        ptr += 1

        # 读取数组元素并插入线性基
        for __ in range(n):
            num = int(input[ptr])
            ptr += 1
            insert(num)

        # 查询并输出最大异或和
        print(query_max())

if __name__ == "__main__":
    main()
```

=====

文件: Codeforces895C_SquareSubsets.cpp

=====

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```

using namespace std;

/**
 * Codeforces 895C. Square Subsets
 * 题目链接: https://codeforces.com/contest/895/problem/C
 *
 * 解题思路:
 * 这道题可以利用线性基和质因数分解来解决。
 * 1. 对于每个数，我们可以将其质因数分解，保留次数为奇数的质因数
 * 2. 这样，每个数可以表示为一个二进制向量，向量的每一位代表一个质数是否出现奇数次
 * 3. 问题转化为：在数组中选择一个非空子集，使得子集中所有数的向量异或结果为零向量
 * 4. 使用动态规划结合线性基来计算方案数
 *
 * 时间复杂度: O(n * m * log m)，其中 n 是数组长度，m 是质数的个数
 * 空间复杂度: O(2^m)，其中 m 是质数的个数
 */
class Solution {
private:
    const int MOD = 1000000007;
    const vector<int> PRIMES = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97};
    const int MAX_NUM = 70;
    const int PRIME_COUNT = PRIMES.size();

    /**
     * 将数字转换为向量表示
     * @param num 输入数字
     * @return 向量表示（二进制掩码）
     */
    int getMask(int num) {
        int mask = 0;
        for (int i = 0; i < PRIME_COUNT; i++) {
            int prime = PRIMES[i];
            int cnt = 0;
            while (num % prime == 0) {
                cnt++;
                num /= prime;
            }
            if (cnt % 2 == 1) {
                mask |= (1 << i);
            }
        }
        return mask;
    }
}

```

```
}
```

```
/**  
 * 快速幂计算  
 * @param base 底数  
 * @param exp 指数  
 * @return base^exp mod MOD  
 */
```

```
long long powMod(long long base, int exp) {  
    long long result = 1;  
    while (exp > 0) {  
        if (exp % 2 == 1) {  
            result = (result * base) % MOD;  
        }  
        base = (base * base) % MOD;  
        exp /= 2;  
    }  
    return result;  
}
```

```
public:
```

```
/**  
 * 计算满足条件的子集数目  
 * @param a 输入数组  
 * @return 满足条件的子集数目  
 */
```

```
int squareSubsets(vector<int>& a) {  
    // 统计每个数的出现次数  
    vector<int> count(MAX_NUM + 1, 0);  
    for (int num : a) {  
        count[num]++;  
    }  
  
    // 初始化 dp 数组, dp[mask] 表示异或结果为 mask 的子集数目  
    vector<long long> dp(1 << PRIME_COUNT, 0);  
    dp[0] = 1; // 空子集  
  
    // 处理每个数  
    for (int num = 2; num <= MAX_NUM; num++) {  
        if (count[num] == 0) {  
            continue;  
        }
```

```

// 将数转换为向量：每个质数是否出现奇数次
int mask = getMask(num);
if (mask == 0) {
    // 这个数本身是平方数，可以选择任意次数，但至少选一次
    long long pow_val = powMod(2, count[num]);
    // 对于所有现有子集，可以选择添加任意非空的平方数集合
    for (int i = 0; i < (1 << PRIME_COUNT); i++) {
        dp[i] = (dp[i] * pow_val) % MOD;
    }
} else {
    // 非平方数，需要用线性基来处理
    // 创建一个临时数组，避免在更新过程中覆盖值
    vector<long long> temp = dp;
    long long pow2 = powMod(2, count[num] - 1); // 选偶数个该数的方式数

    // 对于每个现有的 mask 状态
    for (int i = 0; i < (1 << PRIME_COUNT); i++) {
        // 选择奇数个该数
        temp[i ^ mask] = (temp[i ^ mask] + dp[i] * pow2) % MOD;
    }

    dp = temp;
}
}

// 减去空子集的情况
return (int) ((dp[0] - 1 + MOD) % MOD);
};

// 测试代码
int main() {
    Solution solution;

    // 测试用例 1
    vector<int> a1 = {1, 1, 1};
    cout << "测试用例 1 结果：" << solution.squareSubsets(a1) << endl; // 预期输出：7

    // 测试用例 2
    vector<int> a2 = {2, 2, 2};
    cout << "测试用例 2 结果：" << solution.squareSubsets(a2) << endl; // 预期输出：3

    // 测试用例 3

```

```
vector<int> a3 = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512};  
cout << "测试用例 3 结果：" << solution.squareSubsets(a3) << endl; // 预期输出：1023  
  
return 0;  
}
```

文件: Codeforces895C_SquareSubsets.java

```
package class137;  
  
import java.util.Arrays;  
  
/**  
 * Codeforces 895C. Square Subsets  
 * 题目链接: https://codeforces.com/contest/895/problem/C  
 *  
 * 解题思路:  
 * 这道题可以利用线性基和质因数分解来解决。  
 * 1. 对于每个数，我们可以将其质因数分解，保留次数为奇数的质因数  
 * 2. 这样，每个数可以表示为一个二进制向量，向量的每一位代表一个质数是否出现奇数次  
 * 3. 问题转化为：在数组中选择一个非空子集，使得子集中所有数的向量异或结果为零向量  
 * 4. 使用动态规划结合线性基来计算方案数  
 *  
 * 时间复杂度: O(n * m * log m)，其中 n 是数组长度，m 是质数的个数  
 * 空间复杂度: O(2^m)，其中 m 是质数的个数  
 */  
  
public class Codeforces895C_SquareSubsets {  
    private static final int MOD = 1000000007;  
    private static final int[] PRIMES = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,  
    53, 59, 61, 67, 71, 73, 79, 83, 89, 97};  
    private static final int MAX_NUM = 70;  
    private static final int PRIME_COUNT = PRIMES.length;  
  
    /**  
     * 计算满足条件的子集数目  
     * @param a 输入数组  
     * @return 满足条件的子集数目  
     */  
    public static int squareSubsets(int[] a) {  
        // 统计每个数的出现次数  
        int[] count = new int[MAX_NUM + 1];
```

```

for (int num : a) {
    count[num]++;
}

// 初始化 dp 数组，dp[mask] 表示异或结果为 mask 的子集数目
long[] dp = new long[1 << PRIME_COUNT];
dp[0] = 1; // 空子集

// 处理每个数
for (int num = 2; num <= MAX_NUM; num++) {
    if (count[num] == 0) {
        continue;
    }

    // 将数转换为向量：每个质数是否出现奇数次
    int mask = getMask(num);
    if (mask == 0) {
        // 这个数本身是平方数，可以选择任意次数，但至少选一次
        // 对于平方数，每个数可以选或不选，但至少选一个，所以贡献为  $2^{\text{count}[num]} - 1$ 
        long pow = 1;
        for (int i = 0; i < count[num]; i++) {
            pow = (pow * 2) % MOD;
        }
        // 对于所有现有子集，可以选择添加任意非空的平方数集合
        for (int i = 0; i < (1 << PRIME_COUNT); i++) {
            dp[i] = (dp[i] * pow) % MOD;
        }
    } else {
        // 非平方数，需要用线性基来处理
        // 创建一个临时数组，避免在更新过程中覆盖值
        long[] temp = Arrays.copyOf(dp, dp.length);
        long pow = 1; //  $2^{k-1}$ , 表示选奇数个该数的方式数
        long pow2 = 1; //  $2^{(k-1)}$ , 表示选偶数个该数的方式数

        // 计算  $2^{(\text{count}[num]-1)} \bmod \text{MOD}$ 
        for (int i = 0; i < count[num] - 1; i++) {
            pow2 = (pow2 * 2) % MOD;
        }
        pow = (pow2 * 2) % MOD;

        // 对于每个现有的 mask 状态
        for (int i = 0; i < (1 << PRIME_COUNT); i++) {
            // 选择奇数个该数

```

```

        temp[i ^ mask] = (temp[i ^ mask] + dp[i] * pow2) % MOD;
    }

    dp = temp;
}

}

// 减去空子集的情况
return (int) ((dp[0] - 1 + MOD) % MOD);
}

/***
 * 将数字转换为向量表示
 * @param num 输入数字
 * @return 向量表示（二进制掩码）
 */
private static int getMask(int num) {
    int mask = 0;
    for (int i = 0; i < PRIME_COUNT; i++) {
        int prime = PRIMES[i];
        int cnt = 0;
        while (num % prime == 0) {
            cnt++;
            num /= prime;
        }
        if (cnt % 2 == 1) {
            mask |= (1 << i);
        }
    }
    return mask;
}

// 测试代码
public static void main(String[] args) {
    // 测试用例 1
    int[] a1 = {1, 1, 1};
    System.out.println("测试用例 1 结果: " + squareSubsets(a1)); // 预期输出: 7

    // 测试用例 2
    int[] a2 = {2, 2, 2};
    System.out.println("测试用例 2 结果: " + squareSubsets(a2)); // 预期输出: 3

    // 测试用例 3
}

```

```
    int[] a3 = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512};  
    System.out.println("测试用例 3 结果: " + squareSubsets(a3)); // 预期输出: 1023  
}  
}
```

```
=====
```

文件: Codeforces895C_SquareSubsets.py

```
#!/usr/bin/env python  
# -*- coding: utf-8 -*-
```

```
, , ,
```

Codeforces 895C. Square Subsets

题目链接: <https://codeforces.com/contest/895/problem/C>

解题思路:

这道题可以利用线性基和质因数分解来解决。

1. 对于每个数，我们可以将其质因数分解，保留次数为奇数的质因数
2. 这样，每个数可以表示为一个二进制向量，向量的每一位代表一个质数是否出现奇数次
3. 问题转化为：在数组中选择一个非空子集，使得子集中所有数的向量异或结果为零向量
4. 使用动态规划结合线性基来计算方案数

时间复杂度: $O(n * m * \log m)$ ，其中 n 是数组长度， m 是质数的个数

空间复杂度: $O(2^m)$ ，其中 m 是质数的个数

```
, , ,
```

```
MOD = 10**9 + 7
```

```
PRIMES = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

```
MAX_NUM = 70
```

```
PRIME_COUNT = len(PRIMES)
```

```
def get_mask(num):
```

```
    """
```

将数字转换为向量表示

Args:

 num: 输入数字

Returns:

 int: 向量表示（二进制掩码）

```
    """
```

```
mask = 0
for i in range(PRIME_COUNT):
    prime = PRIMES[i]
    cnt = 0
    temp = num
    while temp % prime == 0:
        cnt += 1
        temp //= prime
    if cnt % 2 == 1:
        mask |= (1 << i)
return mask
```

```
def square_subsets(a):
```

```
"""

```

```
    计算满足条件的子集数目
```

```
Args:
```

```
    a: 输入数组
```

```
Returns:
```

```
    int: 满足条件的子集数目
```

```
"""

```

```
# 统计每个数的出现次数
```

```
count = [0] * (MAX_NUM + 1)
```

```
for num in a:
```

```
    count[num] += 1
```

```
# 初始化 dp 数组, dp[mask] 表示异或结果为 mask 的子集数目
```

```
dp = [0] * (1 << PRIME_COUNT)
```

```
dp[0] = 1 # 空子集
```

```
# 处理每个数
```

```
for num in range(2, MAX_NUM + 1):
```

```
    if count[num] == 0:
```

```
        continue
```

```
# 将数转换为向量: 每个质数是否出现奇数次
```

```
mask = get_mask(num)
```

```
if mask == 0:
```

```
    # 这个数本身是平方数, 可以选择任意次数
```

```
    pow_val = pow(2, count[num], MOD)
```

```
    # 对于所有现有子集, 可以选择添加任意的平方数集合
```

```
    for i in range(1 << PRIME_COUNT):
```

```

dp[i] = (dp[i] * pow_val) % MOD
else:
    # 非平方数，需要用动态规划处理
    # 创建一个临时数组，避免在更新过程中覆盖值
    temp = dp.copy()
    pow2 = pow(2, count[num] - 1, MOD) # 选奇数个该数的方式数

    # 对于每个现有的 mask 状态
    for i in range(1 << PRIME_COUNT):
        # 选择奇数个该数
        temp[i ^ mask] = (temp[i ^ mask] + dp[i] * pow2) % MOD

    dp = temp

# 减去空子集的情况
return (dp[0] - 1) % MOD

# 测试代码
if __name__ == "__main__":
    # 测试用例 1
    a1 = [1, 1, 1]
    print(f"测试用例 1 结果: {square_subsets(a1)}") # 预期输出: 7

    # 测试用例 2
    a2 = [2, 2, 2]
    print(f"测试用例 2 结果: {square_subsets(a2)}") # 预期输出: 3

    # 测试用例 3
    a3 = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
    print(f"测试用例 3 结果: {square_subsets(a3)}") # 预期输出: 1023

```

文件: P3812_MaximumXor.java

```

package class137;

// 最大异或和 (线性基模板题)
// 给定一个长度为 n 的数组 arr, arr 中都是 long 类型的非负数, 可能有重复值
// 在这些数中选取任意个, 使得异或和最大, 返回最大的异或和
// 1 <= n <= 50
// 0 <= arr[i] <= 2^50
// 测试链接 : https://www.luogu.com.cn/problem/P3812

```

```
// 请务必在原有代码基础上增加详细注释，确保代码可以编译运行且没有错误
```

```
import java.io.*;
import java.util.*;
```

```
public class P3812_MaximumXor {
```

```
    public static int MAXN = 51;
    public static int BIT = 50;
```

```
    public static long[] arr = new long[MAXN];
    public static long[] basis = new long[BIT + 1];
    public static int n;
```

```
/**
```

```
* 计算最大异或和
```

```
* 算法思路:
```

```
* 1. 构建线性基
```

```
* 2. 贪心地从高位到低位选择线性基中的元素来最大化结果
```

```
* 时间复杂度: O(n * BIT)
```

```
* 空间复杂度: O(BIT)
```

```
* @return 最大的异或和
```

```
*/
```

```
    public static long compute() {
```

```
        // 构建线性基
```

```
        for (int i = 1; i <= n; i++) {
            insert(arr[i]);
        }
```

```
        // 贪心地选择元素来最大化异或和
```

```
        long ans = 0;
        for (int i = BIT; i >= 0; i--) {
            ans = Math.max(ans, ans ^ basis[i]);
        }
```

```
        return ans;
    }
```

```
/**
```

```
* 将数字插入线性基
```

```
* 算法思路:
```

```
* 1. 从高位到低位扫描
```

```
* 2. 如果当前位为 1 且线性基中该位为空，则直接插入
```

```

* 3. 否则用线性基中该位的数异或当前数，继续处理
* @param num 要插入的数字
* @return 如果成功插入返回 true，否则返回 false
*/
public static boolean insert(long num) {
    for (int i = BIT; i >= 0; i--) {
        if (((num >> i) & 1) != 0) {
            if (basis[i] == 0) {
                basis[i] = num;
                return true;
            }
            num ^= basis[i];
        }
    }
    return false;
}

/**
* 主函数
* 读取输入数据，调用计算函数，输出结果
*/
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st;

    // 读取输入
    n = Integer.parseInt(br.readLine());
    st = new StringTokenizer(br.readLine());

    // 将输入数据存储到 arr 数组中
    for (int i = 1; i <= n; i++) {
        arr[i] = Long.parseLong(st.nextToken());
    }

    // 清空线性基数组
    Arrays.fill(basis, 0);

    // 计算并输出结果
    System.out.println(compute());
}
}

```

文件: P3812_MaximumXor.py

```
=====

# 最大异或和 (线性基模板题)
# 给定一个长度为 n 的数组 arr, arr 中都是 long 类型的非负数, 可能有重复值
# 在这些数中选取任意个, 使得异或和最大, 返回最大的异或和
# 1 <= n <= 50
# 0 <= arr[i] <= 2^50
# 测试链接 : https://www.luogu.com.cn/problem/P3812
# 请务必在原有代码基础上增加详细注释, 确保代码可以编译运行且没有错误
```

```
MAXN = 51
BIT = 50
```

```
arr = [0] * MAXN
basis = [0] * (BIT + 1)
n = 0
```

```
def compute():
    """
    计算最大异或和
    算法思路:
    1. 构建线性基
    2. 贪心地从高位到低位选择线性基中的元素来最大化结果
    时间复杂度: O(n * BIT)
    空间复杂度: O(BIT)
    @return: 最大的异或和
    """

```

```
# 构建线性基
for i in range(1, n + 1):
    insert(arr[i])

# 贪心地选择元素来最大化异或和
ans = 0
for i in range(BIT, -1, -1):
    ans = max(ans, ans ^ basis[i])

return ans
```

```
def insert(num):
    """
    将数字插入线性基
    """
```

算法思路：

1. 从高位到低位扫描
2. 如果当前位为 1 且线性基中该位为空，则直接插入
3. 否则用线性基中该位的数异或当前数，继续处理

@param num: 要插入的数字

@return: 如果成功插入返回 True，否则返回 False

"""

```
for i in range(BIT, -1, -1):
```

```
    if (num >> i) & 1:
```

```
        if basis[i] == 0:
```

```
            basis[i] = num
```

```
            return True
```

```
        num ^= basis[i]
```

```
return False
```

```
def main():
```

"""

主函数

读取输入数据，调用计算函数，输出结果

"""

```
global n
```

读取输入

```
n = int(input())
```

```
temp = list(map(int, input().split()))
```

将输入数据存储到 arr 数组中

```
for i in range(1, n + 1):
```

```
    arr[i] = temp[i - 1]
```

清空线性基数组

```
for i in range(BIT + 1):
```

```
    basis[i] = 0
```

计算并输出结果

```
print(compute())
```

```
if __name__ == "__main__":
```

"""

线性基算法详解

线性基（Linear Basis）是一种处理异或问题的重要数据结构，主要用于解决以下几类问题：

1. 求 n 个数中选取任意个数异或能得到的最大值

2. 求 n 个数中选取任意个数异或能得到的第 k 小值
3. 判断一个数是否能由给定数组中的数异或得到
4. 求能异或得到的数的个数

核心思想

线性基类似于线性代数中的基向量概念，它是一组线性无关的向量集合，能够表示原集合中所有数的异或组合。线性基有以下重要性质：

1. 原序列中的任意一个数都可以由线性基中的某些数异或得到
2. 线性基中的任意一些数异或起来都不能得到 0
3. 在保持性质 1 的前提下，线性基中的数的个数是最少的
4. 线性基中每个元素的二进制最高位互不相同

线性基的构建方法

线性基的构建主要有两种方法：普通消元法和高斯消元法。

普通消元法

普通消元法是最常用的构建线性基的方法，其基本思路是：

1. 从最高位开始扫描
2. 对于每个数，尝试将其插入到线性基中
3. 插入过程：从高位到低位扫描，如果当前位为 1 且线性基中该位为空，则直接插入；否则用线性基中该位的数异或当前数，继续处理

"""

main()

=====