==========
文件夹: class176_MoAlgorithm
==========

[Markdown 文件]
==========
文件: ADDITIONAL_PROBLEMS.md
==========
莫队二次离线算法相关题目汇总

1. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）

题目描述
给定一个长度为 n 的数组 arr，给定一个非负整数 k，下面给出 k1 二元组的定义：
位置二元组(i, j)，i 和 j 必须是不同的，并且 arr[i]异或 arr[j] 的二进制状态里有 k 个 1。
当 i != j 时，(i, j)和(j, i)认为是相同的二元组。
一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，有多少 k1 二元组。

输入格式
第一行三个整数 n, m, k
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n、m <= 10^5
0 <= arr[i]、k < 16384(2 的 14 次方)

题目链接
https://www.luogu.com.cn/problem/P4887

2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追

题目描述
给定一个长度为 n 的数组 arr，定义区间[l,r]的价值为：
对于区间内每个元素 x，其贡献为(x * (小于 x 的元素个数 + 1) + 大于 x 的元素和)
查询 m 个区间的价值和。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr

接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n, m <= 5 * 10^5
1 <= arr[i] <= 10^5

题目链接
https://www.luogu.com.cn/problem/P5501

3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II

题目描述
给定一个长度为 n 的数组 arr，定义倍数二元组：
如果 arr[i]是 arr[j]的倍数(>=1 倍)，那么(i, j)就是一个倍数二元组。
当 i != j 时，(i, j)和(j, i)认为是不同的二元组，不要漏算。
当 i == j 时，(i, j)和(j, i)认为是相同的二元组，不要多算。
一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，有多少倍数二元组。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n、m、arr[i] <= 5 * 10^5

题目链接
https://www.luogu.com.cn/problem/P5398

4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III

题目描述
给定一个长度为 n 的数组 arr，如果 i < j，并且 arr[i] > arr[j]，那么(i,j)就是逆序对。
一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，逆序对的数量。

输入格式
第一行两个整数 n, m

第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n、m <= 10^5
0 <= arr[i] <= 10^9

题目链接
https://www.luogu.com.cn/problem/P5047

5. Codeforces 617E XOR and Favorite Number

题目描述
给定一个长度为 n 的数组 arr 和一个值 k，有 m 次查询。
每次查询[l,r]区间内，有多少个子区间[l'<=l, r'>=r]满足异或和等于 k。

输入格式
第一行三个整数 n, m, k
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n, m <= 100000
1 <= k, arr[i] <= 1000000

题目链接
https://codeforces.com/contest/617/problem/E

6. SPOJ DQUERY - D-query

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询[l,r]区间内，有多少个不同的数。

输入格式
第一行一个整数 n
第二行 n 个整数表示数组 arr

第三行一个整数 m
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n <= 30000
1 <= q <= 200000
1 <= arr[i] <= 10^6

题目链接
https://www.spoj.com/problems/DQUERY/

7. 洛谷 P2709 小 B 的询问

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询[l,r]区间内，每种数字出现次数的平方和。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n, m <= 50000
1 <= k <= n
1 <= arr[i] <= n

题目链接
https://www.luogu.com.cn/problem/P2709

8. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列

题目描述
给定一个长度为 n 的数组 arr，支持两种操作：
1. Q l r: 查询区间[l,r]内不同颜色的个数
2. R pos val: 将位置 pos 的颜色修改为 val

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行表示一个操作

输出格式
对于每个查询操作，输出一行一个整数表示答案

数据范围
1 <= n, m <= 10000
颜色编号在 32 位整数范围内

题目链接
https://www.luogu.com.cn/problem/P1903

9. AtCoder AT1219 歴史の研究

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询[l,r]区间内，数字与其出现次数乘积的最大值。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n <= 100000
1 <= m <= 100000
1 <= arr[i] <= 10^9

题目链接
https://www.luogu.com.cn/problem/AT1219

10. 洛谷 P4688 [Ynoi2016] 掉进兔子洞

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询给出 3 个区间，求这些区间中出现次数最少的元素的出现次数之和。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行 6 个整数 l1, r1, l2, r2, l3, r3 表示 3 个查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n, m <= 100000
1 <= arr[i] <= 10^9

题目链接
https://www.luogu.com.cn/problem/P4688

11. 洛谷 P5231 [JSOI2012] 玄武密码

题目描述
给定一个主串和多个模式串，查询每个模式串在主串中匹配的最长前缀长度。

输入格式
第一行两个整数 n, m
第二行一个长度为 n 的字符串表示主串
接下来 m 行，每行一个字符串表示模式串

输出格式
对于每个模式串，输出一行一个整数表示最长前缀长度

数据范围
1 <= n <= 10000000
1 <= m <= 200000
模式串总长度 <= 200000

题目链接
https://www.luogu.com.cn/problem/P5231

12. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）

题目描述
给定一个长度为 n 的数组 arr 和一个非负整数 k，定义 k1 二元组为满足 arr[i] XOR arr[j]的二进制表示中有 k 个 1 的二元组(i,j)，查询区间内 k1 二元组的个数。

输入格式

第一行三个整数 n, m, k
第二行 n 个整数表示数组 arr
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= n、m <= 10^5
0 <= arr[i]、k < 16384(2 的 14 次方)

题目链接
https://www.luogu.com.cn/problem/P4887

13. 洛谷 P3604 美好的每一天

题目描述
给定一个字符串，查询区间内能重排成回文串的子串个数。

输入格式
第一行一个字符串 s
第二行一个整数 m
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式
对于每个查询，输出一行一个整数表示答案

数据范围
1 <= |s| <= 60000
1 <= m <= 60000

题目链接
https://www.luogu.com.cn/problem/P3604

14. 洛谷 P3674 小清新人渣的本愿

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询[l,r]区间内，是否存在两个数的差等于给定值 a，或者两个数的和等于给定值 b，或者两个数的积等于给定值 c。

输入格式
第一行两个整数 n, m

第二行 n 个整数表示数组 arr
接下来 m 行，每行四个整数 opt, l, r, v 表示查询类型和参数

输出格式
对于每个查询，输出一行"hana"或"bi"表示是否存在满足条件的两个数

数据范围
1 <= n, m <= 100000
1 <= arr[i] <= 100000

题目链接
https://www.luogu.com.cn/problem/P3674

15. 洛谷 P4396 [AHOI2013] 作业

题目描述
给定一个长度为 n 的数组 arr，有 m 次查询。
每次查询[l,r]区间内，数值在[a,b]范围内的不同数字个数和这些数字出现次数之和。

输入格式
第一行两个整数 n, m
第二行 n 个整数表示数组 arr
接下来 m 行，每行四个整数 l, r, a, b 表示查询区间和数值范围

输出格式
对于每个查询，输出一行两个整数表示不同数字个数和出现次数之和

数据范围
1 <= n, m <= 100000
1 <= arr[i] <= 100000

题目链接
https://www.luogu.com.cn/problem/P4396

16. HDU 4638 Group

题目描述
有 n 个人，每个人都有一个唯一的 ID(1..n)。ID 为 i 和 i-1 的人是朋友，ID 为 i 和 i+1 的人也是朋友。这些人站成一排。现在我们选择一个区间的人来分组。k 个人在一组可以创造 k*k 的价值。一个区间的总价值是这些组价值的总和。同一组的人的 ID 必须是连续的。现在选择一个区间的人，想知道应该分成多少组才能使区间的总价值最大。

输入格式

第一行是 T 表示测试用例数。对于每个测试用例，第一行是 n, m(1<=n ,m<=100000)表示有 n 个人和 m 个查询。然后一行有 n 个数字表示从左到右的人的 ID。接下来 m 行每行有两个数字 L,R(1<=L<=R<=n)，表示我们想知道[L,R]的答案。

输出格式
对于每个查询输出一个数字，表示应该分成多少组才能使总价值最大。

数据范围
1 <= n, m <= 100000
1 <= ID <= n
1 <= L <= R <= n

题目链接
http://acm.hdu.edu.cn/showproblem.php?pid=4638

17. 牛客网暑期 ACM 多校训练营 J Different Integers

题目描述
给定一个整数序列 a1, a2, ..., an 和 q 对整数(l1, r1), (l2, r2), ..., (lq, rq)，求 count(l1, r1), count(l2, r2), ..., count(lq, rq)，其中 count(i, j)是 a1, a2, ..., ai, aj, aj+1, ..., an 中不同整数的个数。

输入格式
输入由几个测试用例组成，以文件结束符终止。每个测试用例的第一行包含两个整数 n 和 q。第二行包含 n 个整数 a1, a2, ..., an。接下来的 q 行中第 i 行包含两个整数 li 和 ri。

输出格式
对于每个测试用例，打印 q 个整数表示结果。

数据范围
1 ≤ n, q ≤ 10^5
1 ≤ ai ≤ n
1 ≤ li, ri ≤ n
测试用例数不超过 10 个

题目链接
https://www.nowcoder.com/acm/contest/139/J

18. POJ 2104 K-th Number

题目描述
给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么。

输入格式
第一行两个整数 n 和 m，表示数组长度和查询次数。第二行 n 个整数表示数组元素。接下来 m 行，每行三个整数 i, j, k 表示查询。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n <= 100000
1 <= m <= 5000

题目链接
http://poj.org/problem?id=2104

19. SPOJ MKTHNUM - K-th Number

题目描述
给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么。

输入格式
第一行两个整数 n 和 m，表示数组长度和查询次数。第二行 n 个整数表示数组元素。接下来 m 行，每行三个整数 i, j, k 表示查询。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n <= 100000
1 <= m <= 5000

题目链接
https://www.spoj.com/problems/MKTHNUM/

20. 洛谷 P3834 【模板】可持久化线段树 1（主席树）

题目描述
给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么。

输入格式
第一行两个整数 n 和 m，表示数组长度和查询次数。第二行 n 个整数表示数组元素。接下来 m 行，每行三个整数

i, j, k 表示查询。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n <= 100000
1 <= m <= 5000

题目链接
https://www.luogu.com.cn/problem/P3834

21. HDU 2665 Kth number

题目描述
给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么。

输入格式
第一行包含一个整数 T，表示测试用例的数量。对于每个测试用例，第一行包含两个整数 n 和 m。第二行包含 n 个整数。接下来 m 行，每行包含三个整数 i, j, k。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n <= 100000
1 <= m <= 5000

题目链接
http://acm.hdu.edu.cn/showproblem.php?pid=2665

22. ZOJ 2112 Dynamic Rankings

题目描述
给定一个数组 a[1...n]，支持两种操作：1. 查询区间[i,j]中第 k 小的数；2. 修改位置 i 的值为 t。

输入格式
第一行包含一个整数 T，表示测试用例的数量。对于每个测试用例，第一行包含两个整数 n 和 m。第二行包含 n 个整数。接下来 m 行，每行表示一个操作。

输出格式
对于每个查询操作，输出一行一个整数表示答案。

数据范围
1 <= n <= 50000
1 <= m <= 10000

题目链接
http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2112

23. Codeforces 86D Powerful array

题目描述
给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内，每种数字出现次数的平方和乘以该数字的值的总和。

输入格式
第一行两个整数 n 和 m，表示数组长度和查询次数。第二行 n 个整数表示数组元素。接下来 m 行，每行两个整数 l, r 表示查询区间。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n, m <= 200000
1 <= a[i] <= 10^6

题目链接
https://codeforces.com/problemset/problem/86/D

24. Codeforces 220B Little Elephant and Array

题目描述
给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内，有多少个数字恰好出现了该数字的值那么多次。

输入格式
第一行两个整数 n 和 m，表示数组长度和查询次数。第二行 n 个整数表示数组元素。接下来 m 行，每行两个整数 l, r 表示查询区间。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n, m <= 100000
1 <= a[i] <= 10^9

题目链接
https://codeforces.com/problemset/problem/220/B

25. Codeforces 375D Tree and Queries

题目描述
给定一棵树，每个节点有一个颜色。有 m 次查询，每次查询以 u 为根的子树中，有多少种颜色恰好出现了至少 k 次。

输入格式
第一行两个整数 n 和 m，表示节点数和查询次数。第二行 n 个整数表示每个节点的颜色。接下来 n-1 行，每行两个整数 u, v 表示树的边。接下来 m 行，每行两个整数 u, k 表示查询。

输出格式
对于每个查询，输出一行一个整数表示答案。

数据范围
1 <= n, m <= 100000
1 <= 颜色值 <= 10^5

题目链接
https://codeforces.com/problemset/problem/375/D

总结

莫队算法及其扩展形式（带修莫队、回滚莫队、树上莫队、二次离线莫队）是处理区间查询问题的重要工具。通过合理运用这些算法，可以解决各种复杂的区间统计问题。在实际应用中，需要根据具体问题选择合适的莫队变体，并结合其他数据结构进行优化。

==========

文件: README.md
==========
莫队二次离线算法详解与题目实现

莫队二次离线算法是莫队算法的一种高级扩展，主要用于解决普通莫队转移操作复杂度较高的问题。通过对莫队过程中的扩展操作再次离线处理，可以有效降低整体时间复杂度。

算法原理

基本概念
莫队算法是一种离线处理区间查询问题的技术，通过分块和双指针技术将区间转移的复杂度从 O(n^2)优化到

O(n√n)。

二次离线思想
当莫队的单次转移操作复杂度较高时（如O(logn)），直接应用莫队会导致总复杂度为O(n√n * logn)，可能无法接受。二次离线通过以下步骤优化：

1. 第一次离线：对查询进行排序，模拟莫队过程
2. 记录所有扩展操作（添加/删除元素）
3. 第二次离线：对扩展操作进行批量处理，通过预处理降低单次操作复杂度

适用场景
1. 区间内满足特定条件的元素对计数
2. 区间内元素的复杂统计计算
3. 需要结合数据结构（如树状数组、线段树）维护信息的场景

题目列表

1. 洛谷 P4887 【模板】莫队二次离线
- **题目链接**: https://www.luogu.com.cn/problem/P4887
- **题意**: 给定一个数组，定义k1二元组为满足arr[i] XOR arr[j]的二进制表示中有k个1的二元组(i,j)，查询区间内k1二元组的个数
- **解法**: 莫队二次离线模板题
- **相关文件**: [P4887_MoOfflineTwice.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/P4887_MoOfflineTwice.java), [P4887_MoOfflineTwice.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/P4887_MoOfflineTwice.py)

2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追
- **题目链接**: https://www.luogu.com.cn/problem/P5501
- **题意**: 查询区间内每个元素的贡献和，元素x的贡献为"小于x的元素个数+1"乘以"大于x的元素和"+x
- **解法**: 莫队二次离线结合值域分块

3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II
- **题目链接**: https://www.luogu.com.cn/problem/P5398
- **题意**: 查询区间内倍数二元组的个数，(i,j)是倍数二元组当且仅当arr[i]是arr[j]的倍数
- **解法**: 莫队二次离线结合因数分解

4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III
- **题目链接**: https://www.luogu.com.cn/problem/P5047
- **题意**: 查询区间逆序对个数
- **解法**: 莫队二次离线处理区间逆序对

5. Codeforces 617E XOR and Favorite Number

- **题目链接**: https://codeforces.com/contest/617/problem/E
- **题意**: 查询区间内异或和等于 k 的子区间个数
- **解法**: 普通莫队或莫队二次离线

6. HDU 4638 Group
- **题目链接**: http://acm.hdu.edu.cn/showproblem.php?pid=4638
- **题意**: 给定一个序列，序列由 1-N 个元素全排列而成，求任意区间连续的段数
- **解法**: 普通莫队
- **相关文件**: [HDU4638_Group1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.java), [HDU4638_Group1.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.cpp), [HDU4638_Group1.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.py)

7. 牛客网暑期 ACM 多校训练营 J Different Integers
- **题目链接**: https://www.nowcoder.com/acm/contest/139/J
- **题意**: 求 a1, a2, ..., ai, aj, aj+1, ..., an 中不同整数的个数
- **解法**: 普通莫队
- **相关文件**: [Nowcoder139J_DifferentIntegers1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.java), [Nowcoder139J_DifferentIntegers1.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.cpp), [Nowcoder139J_DifferentIntegers1.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.py)

8. POJ 2104 K-th Number
- **题目链接**: http://poj.org/problem?id=2104
- **题意**: 查询区间第 k 大元素
- **解法**: 主席树（可持久化线段树）
- **相关文件**: [POJ2104_KthNumber1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/POJ2104_KthNumber1.java), [POJ2104_KthNumber1.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/POJ2104_KthNumber1.py)

9. SPOJ DQUERY - D-query
- **题目链接**: https://www.spoj.com/problems/DQUERY/
- **题意**: 查询区间内不同数字的个数
- **解法**: 普通莫队

10. 洛谷 P2709 小 B 的询问
- **题目链接**: https://www.luogu.com.cn/problem/P2709
- **题意**: 查询区间内每种数字出现次数的平方和
- **解法**: 普通莫队

11. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列
- **题目链接**: https://www.luogu.com.cn/problem/P1903
- **题意**: 支持查询区间不同颜色数和修改操作
- **解法**: 带修莫队

12. AtCoder AT1219 歴史の研究
- **题目链接**: https://www.luogu.com.cn/problem/AT1219
- **题意**: 查询区间内数字与其出现次数乘积的最大值
- **解法**: 回滚莫队

复杂度分析

算法类型	时间复杂度	空间复杂度	适用场景
普通莫队	O(n√n)	O(n)	简单转移操作
带修莫队	O(n^(5/3))	O(n)	支持修改操作
回滚莫队	O(n√n)	O(n)	单向操作问题
树上莫队	O(n√n)	O(n)	树上路径查询
二次离线莫队	根据具体问题	根据具体问题	复杂转移操作
主席树	O((n+m)log n)	O(n log n)	静态区间第 k 大

实现要点

分块策略
```java
// 块大小通常选择 sqrt(n)
int blockSize = (int) Math.sqrt(n);
```

排序规则
```java
// 普通莫队排序
public static class QueryComparator implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (bi[a[0]] != bi[b[0]]) {
            return bi[a[0]] - bi[b[0]];
        }
        return a[1] - b[1];
    }
}
```

二次离线处理
1. 记录莫队扩展操作
2. 批量处理操作以降低复杂度
3. 结合数据结构优化查询效率

工程化考量

性能优化
1. 使用位运算优化常数
2. 合理选择数据结构
3. 避免重复计算

异常处理
1. 输入校验
2. 边界条件处理
3. 内存管理

可维护性
1. 代码模块化
2. 详细注释
3. 变量命名规范

与其他算法对比

算法	优势	劣势	适用场景
莫队	实现简单，适用面广	需要离线，常数较大	区间查询问题
线段树	在线查询，效率高	实现复杂	区间查询、修改
树状数组	实现简单，效率高	功能受限	区间统计问题
分块	实现简单，易扩展	效率一般	区间查询问题
主席树	支持历史版本查询	空间复杂度高	静态区间查询

通过深入理解莫队二次离线算法，可以解决一类复杂的区间查询问题，在竞赛和实际应用中都有重要价值。

==========

文件: SUMMARY_AND_PATTERNS.md
==========
莫队二次离线算法总结与技巧

算法本质深入理解

核心思想
莫队二次离线算法的核心在于将复杂度较高的莫队转移操作通过预处理和批量计算进行优化。其本质是：
1. 识别莫队转移中的重复计算部分
2. 通过离线处理将这些重复计算合并
3. 利用数学变换或数据结构优化单次操作复杂度

适用条件
1. 问题可以使用莫队算法解决
2. 莫队的单次转移操作复杂度较高（如O(logn)或更高）
3. 转移操作具有某种可批量处理的性质

常见解题模式

模式1：统计满足特定条件的元素对
这类问题通常需要统计区间内满足某种条件的元素对数量，如：
- XOR值为特定值的元素对
- 倍数/约数关系的元素对
- 差值为特定值的元素对

解决思路：
1. 使用莫队维护当前区间状态
2. 对于每个元素，维护其与之前元素的关系
3. 通过预处理或数据结构优化查询效率

相关题目：
- 洛谷 P4887 【模板】莫队二次离线：https://www.luogu.com.cn/problem/P4887
- Codeforces 617E XOR and Favorite Number：https://codeforces.com/contest/617/problem/E

模式2：复杂贡献计算
这类问题需要计算区间内每个元素的复杂贡献，如：
- 每个元素根据其在区间中的相对位置产生贡献
- 每个元素的贡献依赖于区间内其他元素的统计信息

解决思路：
1. 将贡献计算分解为可处理的部分
2. 利用值域分块、树状数组等数据结构优化
3. 通过二次离线批量处理转移操作

相关题目：
- 洛谷 P5501 [LnOI2019] 来者不拒，去者不追：https://www.luogu.com.cn/problem/P5501

模式3：逆序对类问题
这类问题需要统计区间内的逆序对数量或类似概念。

解决思路：
1. 利用树状数组或线段树维护元素信息
2. 通过莫队二次离线优化转移操作
3. 结合离散化处理大数据范围

相关题目：
- 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III：https://www.luogu.com.cn/problem/P5047

模式 4：连续段计数问题
这类问题需要统计区间内连续数字的段数，如 HDU 4638 Group。

解决思路：
1. 使用布尔数组维护数字是否在当前区间中
2. 当添加或删除数字时，检查其与相邻数字的连接关系
3. 根据连接关系更新段数

相关题目：
- HDU 4638 Group：http://acm.hdu.edu.cn/showproblem.php?pid=4638
- 相关文件：[HDU4638_Group1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.java), [HDU4638_Group1.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.cpp), [HDU4638_Group1.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/HDU4638_Group1.py)

模式 5：不同元素计数问题
这类问题需要统计区间内不同元素的个数，如牛客网 J Different Integers。

解决思路：
1. 使用计数数组维护每个元素在当前区间中的出现次数
2. 当添加元素时，如果该元素第一次出现，则不同元素个数加 1
3. 当删除元素时，如果该元素最后一次出现，则不同元素个数减 1

相关题目：
- 牛客网暑期 ACM 多校训练营 J Different Integers：https://www.nowcoder.com/acm/contest/139/J
- SPOJ DQUERY - D-query：https://www.spoj.com/problems/DQUERY/
- 相关文件：[Nowcoder139J_DifferentIntegers1.java](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.java), [Nowcoder139J_DifferentIntegers1.cpp](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.cpp), [Nowcoder139J_DifferentIntegers1.py](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178/Nowcoder139J_DifferentIntegers1.py)

代码实现技巧

技巧 1：链式前向星优化
在处理离线任务时，使用链式前向星可以高效地存储和遍历任务列表：

```java
// 任务结构
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXQ];
public static int[] taskData = new int[MAXQ];
public static int taskCount = 0;

// 添加任务
public static void addTask(int pos, int data) {
    next[++taskCount] = head[pos];
    head[pos] = taskCount;
    taskData[taskCount] = data;
}
```

技巧 2：值域分块优化
对于涉及值域统计的问题，可以使用值域分块优化查询效率：

```java
// 值域分块
public static int[] blockCnt = new int[MAXB];  // 整块统计
public static int[] numCnt = new int[MAXN];    // 单点统计

// 查询小于 x 的元素个数
public static int lessCount(int x) {
    return blockCnt[belong[x]] + numCnt[x];
}

// 添加元素
public static void addValue(int val) {
    // 更新整块统计
    for (int b = 1; b < belong[val]; b++) {
        blockCnt[b]++;
    }
    // 更新单点统计
    for (int i = bl[belong[val]]; i < val; i++) {
        numCnt[i]++;
```

```
    }
}
```

技巧 3：因数分解预处理
对于涉及倍数/约数关系的问题，可以预处理每个数的因数：

```java
// 预处理因数
public static void preprocessFactors(int num) {
    if (head[num] == 0) {  // 避免重复计算
        for (int i = 1; i * i <= num; i++) {
            if (num % i == 0) {
                addFactor(num, i);
                if (i != num / i) {
                    addFactor(num, num / i);
                }
            }
        }
    }
}
```

复杂度分析方法

分析步骤
1. 确定莫队的基本复杂度：O((n + m) * √n)
2. 分析单次转移操作的复杂度
3. 评估二次离线后的优化效果
4. 考虑预处理和额外数据结构的复杂度

常见复杂度
1. **基础莫队**：O((n + m) * √n)
2. **带修莫队**：O((n + m) * n^(2/3))
3. **二次离线莫队**：根据具体问题，通常为O(n * √n)
4. **主席树**：O((n + m) * log n)

工程化实践要点

性能优化
1. **位运算优化**：使用位运算替代部分算术运算
2. **缓存友好**：合理安排数据结构布局，提高缓存命中率
3. **常数优化**：减少不必要的计算和内存访问

内存管理
1. **静态数组**：对于规模确定的问题，使用静态数组避免动态分配
2. **内存复用**：在不同阶段复用数组空间
3. **及时清理**：在适当时候清空或重置数据结构

调试技巧
1. **中间结果输出**：在关键步骤输出中间结果验证正确性
2. **断言检查**：使用断言验证算法状态的正确性
3. **性能分析**：通过性能分析工具找出瓶颈

与其他算法的对比

与线段树对比
特性	莫队二次离线	线段树
在线/离线	离线	在线
实现难度	中等	较难
适用场景	复杂区间统计	区间查询/修改
时间复杂度	O(n√n)	O(nlogn)
空间复杂度	O(n)	O(n)

与树状数组对比
特性	莫队二次离线	树状数组
功能	通用区间统计	前缀统计
实现难度	中等	简单
扩展性	强	一般
时间复杂度	O(n√n)	O(nlogn)

与主席树对比
特性	莫队二次离线	主席树
功能	通用区间统计	静态区间第 k 大
实现难度	中等	较难
适用场景	复杂转移操作	静态查询
时间复杂度	O(n√n)	O((n+m)logn)
空间复杂度	O(n)	O(nlogn)

常见陷阱与解决方案

陷阱 1：重复计算
问题：在处理离线任务时可能重复计算某些贡献

解决方案：仔细分析贡献计算方式，确保每项贡献只计算一次

陷阱 2：边界处理
问题：区间边界处理不当导致结果错误
解决方案：统一区间定义（左闭右闭或左闭右开），仔细处理边界情况

陷阱 3：数据范围
问题：数据范围估计不足导致数组越界或溢出
解决方案：仔细分析题目数据范围，预留足够的空间

学习建议

初学者路径
1. 掌握基础莫队算法
2. 理解带修莫队和回滚莫队
3. 学习莫队二次离线的基本思想
4. 通过模板题加深理解
5. 尝试解决更复杂的问题

进阶学习
1. 研究各种莫队变体的实现细节
2. 学习与其他数据结构的结合使用
3. 掌握复杂度分析方法
4. 实践工程化优化技巧

实战要点
1. 识别问题是否适合使用莫队二次离线
2. 设计合适的数据结构支持转移操作
3. 优化常数项提高实际运行效率
4. 充分测试各种边界情况

更多相关题目

为了更好地掌握莫队二次离线算法，以下是一些可以在各大 OJ 平台上找到的相关题目：

洛谷平台
- P4887 【模板】莫队二次离线（第十四分块(前体)）：https://www.luogu.com.cn/problem/P4887
- P5501 [LnOI2019] 来者不拒，去者不追：https://www.luogu.com.cn/problem/P5501
- P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II：https://www.luogu.com.cn/problem/P5398
- P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III：https://www.luogu.com.cn/problem/P5047
- P2709 小 B 的询问：https://www.luogu.com.cn/problem/P2709
- P1903 [国家集训队] 数颜色 / 维护队列：https://www.luogu.com.cn/problem/P1903
- P3604 美好的每一天：https://www.luogu.com.cn/problem/P3604

- P3674 小清新人渣的本愿：https://www.luogu.com.cn/problem/P3674

Codeforces 平台
- 617E XOR and Favorite Number：https://codeforces.com/contest/617/problem/E
- 86D Powerful array：https://codeforces.com/problemset/problem/86/D
- 220B Little Elephant and Array：https://codeforces.com/problemset/problem/220/B
- 375D Tree and Queries：https://codeforces.com/problemset/problem/375/D

HDU 平台
- 4638 Group：http://acm.hdu.edu.cn/showproblem.php?pid=4638
- 2665 Kth number：http://acm.hdu.edu.cn/showproblem.php?pid=2665

牛客网平台
- 暑期ACM多校训练营 J Different Integers：https://www.nowcoder.com/acm/contest/139/J

SPOJ 平台
- DQUERY - D-query：https://www.spoj.com/problems/DQUERY/
- MKTHNUM - K-th Number：https://www.spoj.com/problems/MKTHNUM/

POJ 平台
- 2104 K-th Number：http://poj.org/problem?id=2104

AtCoder 平台
- AT1219 歴史の研究：https://www.luogu.com.cn/problem/AT1219

通过系统学习和大量练习，可以熟练掌握莫队二次离线算法，在竞赛和实际应用中发挥其强大作用。

==========

[代码文件]
==========
文件: Code01_MoOfflineTwice1.java
==========
```
package class178;

// 莫队二次离线入门题，java版
// 题目来源: 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）
// 题目链接: https://www.luogu.com.cn/problem/P4887
// 题目大意: 给定一个长度为n的数组arr，给定一个非负整数k，下面给出k1二元组的定义
// 位置二元组(i, j)，i和j必须是不同的，并且 arr[i]异或arr[j] 的二进制状态里有k个1
// 当i != j时，(i, j)和(j, i)认为是相同的二元组
// 一共有m条查询，格式为 l r : 打印arr[l..r]范围上，有多少k1二元组
// 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i]、k < 16384(2的14次方)
```

```
// 解题思路: 使用莫队二次离线算法优化普通莫队的转移操作
// 时间复杂度: O(n*sqrt(n) + n*C(k,14)) 其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数
// 空间复杂度: O(n + 2^14)
// 相关题目:
// 1. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）: https://www.luogu.com.cn/problem/P4887
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II:
https://www.luogu.com.cn/problem/P5398
// 4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 5. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E
// 6. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 7. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 8. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 9. POJ 2104 K-th Number: http://poj.org/problem?id=2104

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class Code01_MoOfflineTwice1 {

    public static int MAXN = 100002;
    public static int MAXV = 1 << 14;
    public static int n, m, k;
    public static int[] arr = new int[MAXN];
    public static int[] bi = new int[MAXN];
    public static int[] kOneArr = new int[MAXV];
    public static int cntk;

    // 莫队任务，l、r、id
    public static int[][] query = new int[MAXN][3];

    // 离线任务，x、l、r、op、id
    // 位置 x 的任务列表用链式前向星表示
    // headl[x]，x 在 l~r 左侧的离线任务列表
    // headr[x]，x 在 l~r 右侧的离线任务列表
    public static int[] headl = new int[MAXN];
    public static int[] headr = new int[MAXN];
    public static int[] nextq = new int[MAXN << 1];
```

```
public static int[] ql = new int[MAXN << 1];
public static int[] qr = new int[MAXN << 1];
public static int[] qop = new int[MAXN << 1];
public static int[] qid = new int[MAXN << 1];
public static int cntq;

// cnt[v] : 当前的数字 v 作为第二个数，之前出现的数字作为第一个数，产生多少 k1 二元组
public static int[] cnt = new int[MAXV];
// 前缀和
public static long[] pre = new long[MAXN];
// 后缀和
public static long[] suf = new long[MAXN];

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

public static class QueryCmp implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (bi[a[0]] != bi[b[0]]) {
            return bi[a[0]] - bi[b[0]];
        }
        return a[1] - b[1];
    }
}

public static int lowbit(int i) {
    return i & -i;
}

public static int countOne(int num) {
    int ret = 0;
    while (num > 0) {
        ret++;
        num -= lowbit(num);
    }
    return ret;
}

public static void addLeftOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headl[x];
    headl[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
```

```
        qop[cntq] = op;
        qid[cntq] = id;
    }

    public static void addRightOffline(int x, int l, int r, int op, int id) {
        nextq[++cntq] = headr[x];
        headr[x] = cntq;
        ql[cntq] = l;
        qr[cntq] = r;
        qop[cntq] = op;
        qid[cntq] = id;
    }

    public static void prepare() {
        int blen = (int) Math.sqrt(n);
        for (int i = 1; i <= n; i++) {
            bi[i] = (i - 1) / blen + 1;
        }
        Arrays.sort(query, 1, m + 1, new QueryCmp());
        for (int v = 0; v < MAXV; v++) {
            if (countOne(v) == k) {
                kOneArr[++cntk] = v;
            }
        }
    }

    public static void compute() {
        for (int i = 1; i <= n; i++) {
            pre[i] = pre[i - 1] + cnt[arr[i]];
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[i] ^ kOneArr[j]]++;
            }
        }
        Arrays.fill(cnt, 0);
        for (int i = n; i >= 1; i--) {
            suf[i] = suf[i + 1] + cnt[arr[i]];
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[i] ^ kOneArr[j]]++;
            }
        }
        // 执行莫队
        int winl = 1, winr = 0;
        for (int i = 1; i <= m; i++) {
```

```
        int jobl = query[i][0];
        int jobr = query[i][1];
        int id = query[i][2];
        if (winr < jobr) {
            addLeftOffline(winl - 1, winr + 1, jobr, -1, id);
            ans[id] += pre[jobr] - pre[winr];
        }
        if (winr > jobr) {
            addLeftOffline(winl - 1, jobr + 1, winr, 1, id);
            ans[id] -= pre[winr] - pre[jobr];
        }
        winr = jobr;
        if (winl > jobl) {
            addRightOffline(winr + 1, jobl, winl - 1, -1, id);
            ans[id] += suf[jobl] - suf[winl];
        }
        if (winl < jobl) {
            addRightOffline(winr + 1, winl, jobl - 1, 1, id);
            ans[id] -= suf[winl] - suf[jobl];
        }
        winl = jobl;
    }
    Arrays.fill(cnt, 0);
    for (int x = 0; x <= n; x++) {
        if (x >= 1) {
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[x] ^ kOneArr[j]]++;
            }
        }
        for (int q = headl[x]; q > 0; q = nextq[q]) {
            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
            for (int j = l; j <= r; j++) {
                ans[id] += (long) op * cnt[arr[j]];
            }
        }
    }
    Arrays.fill(cnt, 0);
    for (int x = n + 1; x >= 1; x--) {
        if (x <= n) {
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[x] ^ kOneArr[j]]++;
            }
        }
```

```
        for (int q = headr[x]; q > 0; q = nextq[q]) {
            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
            for (int j = l; j <= r; j++) {
                ans[id] += (long) op * cnt[arr[j]];
            }
        }
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    k = in.nextInt();
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
    }
    for (int i = 1; i <= m; i++) {
        query[i][0] = in.nextInt();
        query[i][1] = in.nextInt();
        query[i][2] = i;
    }
    prepare();
    compute();
    // ans[i]代表答案变化量
    // 所以加工出前缀和才是每个查询的答案
    // 注意在普通莫队的顺序下，去生成前缀和
    for (int i = 2; i <= m; i++) {
        ans[query[i][2]] += ans[query[i - 1][2]];
    }
    for (int i = 1; i <= m; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;
```

```
        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0)
                    return -1;
            }
            return buffer[ptr++];
        }

        int nextInt() throws IOException {
            int c;
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==========

文件: Code01_MoOfflineTwice2.java

==========

```
package class178;

// 莫队二次离线入门题，C++版
```

```
// 题目来源: 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）
// 题目链接: https://www.luogu.com.cn/problem/P4887
// 题目大意: 给定一个长度为 n 的数组 arr，给定一个非负整数 k，下面给出 k1 二元组的定义
// 位置二元组(i, j)，i 和 j 必须是不同的，并且 arr[i]异或 arr[j] 的二进制状态里有 k 个 1
// 当 i != j 时，(i, j)和(j, i)认为是相同的二元组
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，有多少 k1 二元组
// 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i]、k < 16384(2 的 14 次方)
// 解题思路: 使用莫队二次离线算法优化普通莫队的转移操作
// 时间复杂度: O(n*sqrt(n) + n*C(k,14)) 其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数
// 空间复杂度: O(n + 2^14)
// 相关题目:
// 1. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）: https://www.luogu.com.cn/problem/P4887
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II:
https://www.luogu.com.cn/problem/P5398
// 4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 5. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E
// 6. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 7. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 8. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 9. POJ 2104 K-th Number: http://poj.org/problem?id=2104

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Query {
//    int l, r, id;
//};
//
//const int MAXN = 100002;
//const int MAXV = 1 << 14;
//int n, m, k;
//int arr[MAXN];
//int bi[MAXN];
//int kOneArr[MAXV];
//int cntk;
//
//Query query[MAXN];
//
//int headl[MAXN];
//int headr[MAXN];
```

```
//int nextq[MAXN << 1];
//int ql[MAXN << 1];
//int qr[MAXN << 1];
//int qop[MAXN << 1];
//int qid[MAXN << 1];
//int cntq;
//
//int cnt[MAXV];
//long long pre[MAXN];
//long long suf[MAXN];
//
//long long ans[MAXN];
//
//bool QueryCmp(Query &a, Query &b) {
//    if (bi[a.l] != bi[b.l]) {
//        return bi[a.l] < bi[b.l];
//    }
//    return a.r < b.r;
//}
//
//int lowbit(int x) {
//    return x & -x;
//}
//
//int countOne(int num) {
//    int ret = 0;
//    while (num > 0) {
//        ret++;
//        num -= lowbit(num);
//    }
//    return ret;
//}
//
//void addLeftOffline(int x, int l, int r, int op, int id) {
//    nextq[++cntq] = headl[x];
//    headl[x] = cntq;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//void addRightOffline(int x, int l, int r, int op, int id) {
```

```
//    nextq[++cntq] = headr[x];
//    headr[x] = cntq;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//void prepare() {
//    int blen = (int)sqrt(n);
//    for (int i = 1; i <= n; i++) {
//        bi[i] = (i - 1) / blen + 1;
//    }
//    sort(query + 1, query + m + 1, QueryCmp);
//    for (int v = 0; v < MAXV; v++) {
//        if (countOne(v) == k) {
//            kOneArr[++cntk] = v;
//        }
//    }
//}
//
//void compute() {
//    for (int i = 1; i <= n; i++) {
//        pre[i] = pre[i - 1] + cnt[arr[i]];
//        for (int j = 1; j <= cntk; j++) {
//            cnt[arr[i] ^ kOneArr[j]]++;
//        }
//    }
//    memset(cnt, 0, sizeof(cnt));
//    for (int i = n; i >= 1; i--) {
//        suf[i] = suf[i + 1] + cnt[arr[i]];
//        for (int j = 1; j <= cntk; j++) {
//            cnt[arr[i] ^ kOneArr[j]]++;
//        }
//    }
//    int winl = 1, winr = 0;
//    for (int i = 1; i <= m; i++) {
//        int jobl = query[i].l;
//        int jobr = query[i].r;
//        int id = query[i].id;
//        if (winr < jobr) {
//            addLeftOffline(winl - 1, winr + 1, jobr, -1, id);
//            ans[id] += pre[jobr] - pre[winr];
```

```
//        }
//        if (winr > jobr) {
//            addLeftOffline(winl - 1, jobr + 1, winr, 1, id);
//            ans[id] -= pre[winr] - pre[jobr];
//        }
//        winr = jobr;
//        if (winl > jobl) {
//            addRightOffline(winr + 1, jobl, winl - 1, -1, id);
//            ans[id] += suf[jobl] - suf[winl];
//        }
//        if (winl < jobl) {
//            addRightOffline(winr + 1, winl, jobl - 1, 1, id);
//            ans[id] -= suf[winl] - suf[jobl];
//        }
//        winl = jobl;
//    }
//    memset(cnt, 0, sizeof(cnt));
//    for (int x = 0; x <= n; x++) {
//        if (x >= 1) {
//            for (int j = 1; j <= cntk; j++) {
//                cnt[arr[x] ^ kOneArr[j]]++;
//            }
//        }
//        for (int q = headl[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            for (int j = l; j <= r; j++) {
//                ans[id] += 1LL * op * cnt[arr[j]];
//            }
//        }
//    }
//    memset(cnt, 0, sizeof(cnt));
//    for (int x = n + 1; x >= 1; x--) {
//        if (x <= n) {
//            for (int j = 1; j <= cntk; j++) {
//                cnt[arr[x] ^ kOneArr[j]]++;
//            }
//        }
//        for (int q = headr[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            for (int j = l; j <= r; j++) {
//                ans[id] += 1LL * op * cnt[arr[j]];
//            }
//        }
```

```
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m >> k;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> query[i].l >> query[i].r;
//        query[i].id = i;
//    }
//    prepare();
//    compute();
//    for (int i = 2; i <= m; i++) {
//        ans[query[i].id] += ans[query[i - 1].id];
//    }
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code02_OfflineInversion1.java

==========

```
package class178;

// 区间逆序对，java 版
// 题目来源: 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III
// 题目链接: https://www.luogu.com.cn/problem/P5047
// 题目大意: 给定一个长度为 n 的数组 arr，如果 i < j，并且 arr[i] > arr[j]，那么(i,j)就是逆序对
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，逆序对的数量
// 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i] <= 10^9
// 解题思路: 使用莫队二次离线算法处理区间逆序对问题
// 时间复杂度: O(n*sqrt(n)*logn)
// 空间复杂度: O(n)
// 相关题目:
// 1. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
```

```
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. HDU 1394 Minimum Inversion Number: http://acm.hdu.edu.cn/showproblem.php?pid=1394
// 4. Codeforces 296C Greg and Array: https://codeforces.com/problemset/problem/296/C

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class Code02_OfflineInversion1 {

    public static int MAXN = 100002;
    public static int MAXB = 401;
    public static int n, m;
    public static int[] arr = new int[MAXN];
    public static int[] sorted = new int[MAXN];
    public static int cntv;

    public static int[][] query = new int[MAXN][3];
    public static int[] headl = new int[MAXN];
    public static int[] headr = new int[MAXN];
    public static int[] nextq = new int[MAXN << 1];
    public static int[] ql = new int[MAXN << 1];
    public static int[] qr = new int[MAXN << 1];
    public static int[] qop = new int[MAXN << 1];
    public static int[] qid = new int[MAXN << 1];
    public static int cntq;

    // bi 用于序列分块、值域分块，bl 和 br 用于值域分块
    public static int[] bi = new int[MAXN];
    public static int[] bl = new int[MAXB];
    public static int[] br = new int[MAXB];

    // 树状数组
    public static int[] tree = new int[MAXN];
    // 前缀信息
    public static long[] pre = new long[MAXN];
    // 后缀信息
    public static long[] suf = new long[MAXN];

    // 整块增加的词频
```

```
public static long[] blockCnt = new long[MAXB];
// 单个数值的词频
public static long[] numCnt = new long[MAXN];

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

public static class QueryCmp implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (bi[a[0]] != bi[b[0]]) {
            return bi[a[0]] - bi[b[0]];
        }
        return a[1] - b[1];
    }
}

public static int kth(int num) {
    int left = 1, right = cntv, mid, ret = 0;
    while (left <= right) {
        mid = (left + right) / 2;
        if (sorted[mid] <= num) {
            ret = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return ret;
}

public static int lowbit(int i) {
    return i & -i;
}

public static void add(int i, int v) {
    while (i <= cntv) {
        tree[i] += v;
        i += lowbit(i);
    }
}

public static int sum(int i) {
    int ret = 0;
```

```
    while (i > 0) {
        ret += tree[i];
        i -= lowbit(i);
    }
    return ret;
}

public static void addLeftOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headl[x];
    headl[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

public static void addRightOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headr[x];
    headr[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

// 增加 1 ~ val-1，这些数字的词频
public static void addLeftCnt(int val) {
    for (int b = 1; b <= bi[val] - 1; b++) {
        blockCnt[b]++;
    }
    for (int i = bl[bi[val]]; i < val; i++) {
        numCnt[i]++;
    }
}

// 增加 val+1 ~ cntv，这些数字的词频
public static void addRightCnt(int val) {
    for (int b = bi[val] + 1; b <= bi[cntv]; b++) {
        blockCnt[b]++;
    }
    for (int i = val + 1; i <= br[bi[val]]; i++) {
        numCnt[i]++;
    }
```

```
}

public static long getCnt(int val) {
    return blockCnt[bi[val]] + numCnt[val];
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    Arrays.sort(sorted, 1, n + 1);
    cntv = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[cntv] != sorted[i]) {
            sorted[++cntv] = sorted[i];
        }
    }
    for (int i = 1; i <= n; i++) {
        arr[i] = kth(arr[i]);
    }
    int blen = (int) Math.sqrt(n);
    int bnum = (n + blen - 1) / blen;
    for (int i = 1; i <= n; i++) {
        bi[i] = (i - 1) / blen + 1;
    }
    for (int i = 1; i <= bnum; i++) {
        bl[i] = (i - 1) * blen + 1;
        br[i] = Math.min(i * blen, cntv);
    }
    Arrays.sort(query, 1, m + 1, new QueryCmp());
}

public static void compute() {
    for (int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1] + sum(cntv) - sum(arr[i]);
        add(arr[i], 1);
    }
    Arrays.fill(tree, 1, cntv + 1, 0);
    for (int i = n; i >= 1; i--) {
        suf[i] = suf[i + 1] + sum(arr[i] - 1);
        add(arr[i], 1);
    }
    int winl = 1, winr = 0;
```

```
for (int i = 1; i <= m; i++) {
    int jobl = query[i][0];
    int jobr = query[i][1];
    int id = query[i][2];
    if (winr < jobr) {
        addLeftOffline(winl - 1, winr + 1, jobr, -1, id);
        ans[id] += pre[jobr] - pre[winr];
    }
    if (winr > jobr) {
        addLeftOffline(winl - 1, jobr + 1, winr, 1, id);
        ans[id] -= pre[winr] - pre[jobr];
    }
    winr = jobr;
    if (winl > jobl) {
        addRightOffline(winr + 1, jobl, winl - 1, -1, id);
        ans[id] += suf[jobl] - suf[winl];
    }
    if (winl < jobl) {
        addRightOffline(winr + 1, winl, jobl - 1, 1, id);
        ans[id] -= suf[winl] - suf[jobl];
    }
    winl = jobl;
}
for (int x = 0; x <= n; x++) {
    if (x >= 1) {
        addLeftCnt(arr[x]);
    }
    for (int q = headl[x]; q > 0; q = nextq[q]) {
        int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
        long ret = 0;
        for (int j = l; j <= r; j++) {
            ret += getCnt(arr[j]);
        }
        ans[id] += ret * op;
    }
}
Arrays.fill(blockCnt, 0);
Arrays.fill(numCnt, 0);
for (int x = n + 1; x >= 1; x--) {
    if (x <= n) {
        addRightCnt(arr[x]);
    }
    for (int q = headr[x]; q > 0; q = nextq[q]) {
```

```
            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
            long ret = 0;
            for (int j = l; j <= r; j++) {
                ret += getCnt(arr[j]);
            }
            ans[id] += ret * op;
        }
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    n = in.nextInt();
    m = in.nextInt();
    for (int i = 1; i <= n; i++) {
        arr[i] = in.nextInt();
    }
    for (int i = 1; i <= m; i++) {
        query[i][0] = in.nextInt();
        query[i][1] = in.nextInt();
        query[i][2] = i;
    }
    prepare();
    compute();
    for (int i = 2; i <= m; i++) {
        ans[query[i][2]] += ans[query[i - 1][2]];
    }
    for (int i = 1; i <= m; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
```

```
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0)
                    return -1;
            }
            return buffer[ptr++];
        }

        int nextInt() throws IOException {
            int c;
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==========

文件: Code02_OfflineInversion2.java

==========

```
package class178;

// 区间逆序对，C++版
// 题目来源: 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III
// 题目链接: https://www.luogu.com.cn/problem/P5047
// 题目大意: 给定一个长度为 n 的数组 arr，如果 i < j，并且 arr[i] > arr[j]，那么(i,j)就是逆序对
```

```
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，逆序对的数量
// 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i] <= 10^9
// 解题思路: 使用莫队二次离线算法处理区间逆序对问题
// 时间复杂度: O(n*sqrt(n)*logn)
// 空间复杂度: O(n)
// 相关题目:
// 1. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. HDU 1394 Minimum Inversion Number: http://acm.hdu.edu.cn/showproblem.php?pid=1394
// 4. Codeforces 296C Greg and Array: https://codeforces.com/problemset/problem/296/C

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Query {
//    int l, r, id;
//};
//
//const int MAXN = 100002;
//const int MAXB = 401;
//
//int n, m;
//int arr[MAXN];
//int sorted[MAXN];
//int cntv;
//
//Query query[MAXN];
//int headl[MAXN];
//int headr[MAXN];
//int nextq[MAXN << 1];
//int ql[MAXN << 1];
//int qr[MAXN << 1];
//int qop[MAXN << 1];
//int qid[MAXN << 1];
//int cntq;
//
//int bi[MAXN];
//int bl[MAXB];
//int br[MAXB];
//
//int tree[MAXN];
```

```
//long long pre[MAXN];
//long long suf[MAXN];
//
//long long blockCnt[MAXB];
//long long numCnt[MAXN];
//
//long long ans[MAXN];
//
//bool QueryCmp(Query &a, Query &b) {
//    if (bi[a.l] != bi[b.l]) {
//        return bi[a.l] < bi[b.l];
//    }
//    return a.r < b.r;
//}
//
//int kth(int num) {
//    int left = 1, right = cntv, ret = 0;
//    while (left <= right) {
//        int mid = (left + right) >> 1;
//        if (sorted[mid] <= num) {
//            ret = mid;
//            left = mid + 1;
//        } else {
//            right = mid - 1;
//        }
//    }
//    return ret;
//}
//
//int lowbit(int i) {
//    return i & -i;
//}
//
//void add(int i, int v) {
//    while (i <= cntv) {
//        tree[i] += v;
//        i += lowbit(i);
//    }
//}
//
//int sum(int i) {
//    int ret = 0;
//    while (i > 0) {
```

```
//        ret += tree[i];
//        i -= lowbit(i);
//    }
//    return ret;
//}
//
//void addLeftOffline(int x, int l, int r, int op, int id) {
//    nextq[++cntq] = headl[x];
//    headl[x] = cntq;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//void addRightOffline(int x, int l, int r, int op, int id) {
//    nextq[++cntq] = headr[x];
//    headr[x] = cntq;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//void addLeftCnt(int val) {
//    for (int b = 1; b <= bi[val] - 1; b++) {
//        blockCnt[b]++;
//    }
//    for (int i = bl[bi[val]]; i < val; i++) {
//        numCnt[i]++;
//    }
//}
//
//void addRightCnt(int val) {
//    for (int b = bi[val] + 1; b <= bi[cntv]; b++) {
//        blockCnt[b]++;
//    }
//    for (int i = val + 1; i <= br[bi[val]]; i++) {
//        numCnt[i]++;
//    }
//}
//
//long long getCnt(int val) {
```

```
//    return blockCnt[bi[val]] + numCnt[val];
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
//        sorted[i] = arr[i];
//    }
//    sort(sorted + 1, sorted + n + 1);
//    cntv = 1;
//    for (int i = 2; i <= n; i++) {
//        if (sorted[cntv] != sorted[i]) {
//            sorted[++cntv] = sorted[i];
//        }
//    }
//    for (int i = 1; i <= n; i++) {
//        arr[i] = kth(arr[i]);
//    }
//    int blen = (int)sqrt(n);
//    int bnum = (n + blen - 1) / blen;
//    for (int i = 1; i <= n; i++) {
//        bi[i] = (i - 1) / blen + 1;
//    }
//    for (int i = 1; i <= bnum; i++) {
//        bl[i] = (i - 1) * blen + 1;
//        br[i] = min(i * blen, cntv);
//    }
//    sort(query + 1, query + m + 1, QueryCmp);
//}
//
//void compute() {
//    for (int i = 1; i <= n; i++) {
//        pre[i] = pre[i - 1] + sum(cntv) - sum(arr[i]);
//        add(arr[i], 1);
//    }
//    memset(tree + 1, 0, cntv * sizeof(int));
//    for (int i = n; i >= 1; i--) {
//        suf[i] = suf[i + 1] + sum(arr[i] - 1);
//        add(arr[i], 1);
//    }
//    int winl = 1, winr = 0;
//    for (int i = 1; i <= m; i++) {
//        int jobl = query[i].l;
//        int jobr = query[i].r;
```

```
//        int id = query[i].id;
//        if (winr < jobr) {
//            addLeftOffline(winl - 1, winr + 1, jobr, -1, id);
//            ans[id] += pre[jobr] - pre[winr];
//        }
//        if (winr > jobr) {
//            addLeftOffline(winl - 1, jobr + 1, winr, 1, id);
//            ans[id] -= pre[winr] - pre[jobr];
//        }
//        winr = jobr;
//        if (winl > jobl) {
//            addRightOffline(winr + 1, jobl, winl - 1, -1, id);
//            ans[id] += suf[jobl] - suf[winl];
//        }
//        if (winl < jobl) {
//            addRightOffline(winr + 1, winl, jobl - 1, 1, id);
//            ans[id] -= suf[winl] - suf[jobl];
//        }
//        winl = jobl;
//    }
//    for (int x = 0; x <= n; x++) {
//        if (x >= 1) {
//            addLeftCnt(arr[x]);
//        }
//        for (int q = headl[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            long long ret = 0;
//            for (int j = l; j <= r; j++) {
//                ret += getCnt(arr[j]);
//            }
//            ans[id] += ret * op;
//        }
//    }
//    memset(blockCnt, 0, sizeof(blockCnt));
//    memset(numCnt, 0, sizeof(numCnt));
//    for (int x = n + 1; x >= 1; x--) {
//        if (x <= n) {
//            addRightCnt(arr[x]);
//        }
//        for (int q = headr[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            long long ret = 0;
//            for (int j = l; j <= r; j++) {
```

```
//                ret += getCnt(arr[j]);
//            }
//            ans[id] += ret * op;
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> query[i].l >> query[i].r;
//        query[i].id = i;
//    }
//    prepare();
//    compute();
//    for (int i = 2; i <= m; i++) {
//        ans[query[i].id] += ans[query[i - 1].id];
//    }
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Code03_Abbi1.java

==========

```
package class178;

// 区间 Abbi 值，java 版
// 题目来源: 洛谷 P5501 [LnOI2019] 来者不拒，去者不追
// 题目链接: https://www.luogu.com.cn/problem/P5501
// 题目大意: 给定一个长度为 n 的数组 arr，区间 Abbi 值的定义如下
// 如果 arr[l..r]包含数字 v，并且 v 是第 k 小，那么这个数字的 Abbi 值 = v * k
// 区间 Abbi 值 = 区间内所有数字 Abbi 值的累加和
// 比如[1, 2, 2, 3]的 Abbi 值 = 1 * 1 + 2 * 2 + 2 * 2 + 3 * 4 = 21
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]的区间 Abbi 值
```

```
// 数据范围: 1 <= n、m <= 5 * 10^5, 1 <= arr[i] <= 10^5
// 解题思路: 使用莫队二次离线算法处理区间 Abbi 值问题
// 时间复杂度: O(n*sqrt(n)*logn)
// 空间复杂度: O(n)
// 相关题目:
// 1. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 2. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 3. AtCoder AT1219 歴史の研究: https://www.luogu.com.cn/problem/AT1219
// 4. Codeforces 85D Sum of Medians: https://codeforces.com/problemset/problem/85/D

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class Code03_Abbi1 {

    public static int MAXN = 500001;
    public static int MAXV = 100000;
    public static int MAXB = 401;
    public static int n, m;
    public static int[] arr = new int[MAXN];
    public static long[] preSum = new long[MAXN];

    // 序列分块 + 值域分块
    public static int[] bi = new int[MAXN];
    public static int[] bl = new int[MAXB];
    public static int[] br = new int[MAXB];

    // 莫队任务 + 二次离线任务(x, l, r, op)
    public static int[][] query = new int[MAXN][3];
    public static int[] headq = new int[MAXN];
    public static int[] nextq = new int[MAXN << 1];
    public static int[] ql = new int[MAXN << 1];
    public static int[] qr = new int[MAXN << 1];
    public static int[] qop = new int[MAXN << 1];
    public static int[] qid = new int[MAXN << 1];
    public static int cntq;

    // 值域树状数组，统计<x 的个数
```

```
public static long[] treeCnt = new long[MAXV + 1];
// 值域树状数组，统计>x 所有数的累加和
public static long[] treeSum = new long[MAXV + 1];
// 前缀信息
public static long[] pre = new long[MAXN];

// 值域分块，统计<x 的个数
public static int[] blockLessCnt = new int[MAXB];
public static int[] numLessCnt = new int[MAXN];

// 值域分块，统计>x 所有数的累加和
public static long[] blockMoreSum = new long[MAXB];
public static long[] numMoreSum = new long[MAXN];

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

public static class QueryCmp implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (bi[a[0]] != bi[b[0]]) {
            return bi[a[0]] - bi[b[0]];
        }
        return a[1] - b[1];
    }
}

public static void addOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headq[x];
    headq[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

public static int lowbit(int x) {
    return x & -x;
}

public static void add(long[] tree, int x, int v) {
    while (x <= MAXV) {
        tree[x] += v;
        x += lowbit(x);
```

```
    }
}

public static long sum(long[] tree, int x) {
    long ret = 0;
    while (x > 0) {
        ret += tree[x];
        x -= lowbit(x);
    }
    return ret;
}

// 执行二次离线的过程中，加入数字 val，修改相关信息
public static void addVal(int val) {
    for (int b = bi[val] + 1; b <= bi[MAXV]; b++) {
        blockLessCnt[b]++;
    }
    for (int i = val + 1; i <= br[bi[val]]; i++) {
        numLessCnt[i]++;
    }
    for (int b = 1; b <= bi[val] - 1; b++) {
        blockMoreSum[b] += val;
    }
    for (int i = bl[bi[val]]; i < val; i++) {
        numMoreSum[i] += val;
    }
}

// 查询<x 的个数
public static int lessCnt(int x) {
    return blockLessCnt[bi[x]] + numLessCnt[x];
}

// 查询>x 的所有数累加和
public static long moreSum(int x) {
    return blockMoreSum[bi[x]] + numMoreSum[x];
}

public static void prepare() {
    for (int i = 1; i <= n; i++) {
        preSum[i] = preSum[i - 1] + arr[i];
    }
    int blen = (int) Math.sqrt(MAXV);
```

```
        int bnum = (MAXV + blen - 1) / blen;
        for (int i = 1; i <= MAXV; i++) {
            bi[i] = (i - 1) / blen + 1;
        }
        for (int i = 1; i <= bnum; i++) {
            bl[i] = (i - 1) * blen + 1;
            br[i] = Math.min(i * blen, MAXV);
        }
        Arrays.sort(query, 1, m + 1, new QueryCmp());
    }

    public static void compute() {
        for (int i = 1; i <= n; i++) {
            pre[i] = pre[i - 1] + sum(treeCnt, arr[i] - 1) * arr[i] + sum(treeSum, MAXV) -
sum(treeSum, arr[i]);
            add(treeCnt, arr[i], 1);
            add(treeSum, arr[i], arr[i]);
        }
        int winl = 1, winr = 0;
        for (int i = 1; i <= m; i++) {
            int jobl = query[i][0];
            int jobr = query[i][1];
            int id = query[i][2];
            if (winr < jobr) {
                addOffline(winl - 1, winr + 1, jobr, -1, id);
                ans[id] += pre[jobr] - pre[winr];
            }
            if (winr > jobr) {
                addOffline(winl - 1, jobr + 1, winr, 1, id);
                ans[id] -= pre[winr] - pre[jobr];
            }
            winr = jobr;
            if (winl > jobl) {
                addOffline(winr, jobl, winl - 1, 1, id);
                ans[id] -= pre[winl - 1] - pre[jobl - 1];
            }
            if (winl < jobl) {
                addOffline(winr, winl, jobl - 1, -1, id);
                ans[id] += pre[jobl - 1] - pre[winl - 1];
            }
            winl = jobl;
        }
        long tmp;
```

```
        for (int x = 0; x <= n; x++) {
            if (x >= 1) {
                addVal(arr[x]);
            }
            for (int q = headq[x]; q > 0; q = nextq[q]) {
                int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
                for (int j = l; j <= r; j++) {
                    tmp = (long) lessCnt(arr[j]) * arr[j] + moreSum(arr[j]);
                    if (op == 1) {
                        ans[id] += tmp;
                    } else {
                        ans[id] -= tmp;
                    }
                }
            }
        }
    }

    public static void main(String[] args) throws Exception {
        FastReader in = new FastReader(System.in);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        n = in.nextInt();
        m = in.nextInt();
        for (int i = 1; i <= n; i++) {
            arr[i] = in.nextInt();
        }
        for (int i = 1; i <= m; i++) {
            query[i][0] = in.nextInt();
            query[i][1] = in.nextInt();
            query[i][2] = i;
        }
        prepare();
        compute();
        // 加工前缀和
        for (int i = 2; i <= m; i++) {
            ans[query[i][2]] += ans[query[i - 1][2]];
        }
        // 贡献是修正过的概念，现在补偿回来
        for (int i = 1; i <= m; i++) {
            ans[query[i][2]] += preSum[query[i][1]] - preSum[query[i][0] - 1];
        }
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
```

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

    // 读写工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;

        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
                if (len <= 0)
                    return -1;
            }
            return buffer[ptr++];
        }

        int nextInt() throws IOException {
            int c;
            do {
                c = readByte();
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }
```

```
}

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

文件: Code03_Abbi2.java
==========================================
package class178;

// 区间 Abbi 值，C++版
// 题目来源: 洛谷 P5501 [LnOI2019] 来者不拒，去者不追
// 题目链接: https://www.luogu.com.cn/problem/P5501
// 题目大意: 给定一个长度为 n 的数组 arr，区间 Abbi 值的定义如下
// 如果 arr[l..r]包含数字 v，并且 v 是第 k 小，那么这个数字的 Abbi 值 = v * k
// 区间 Abbi 值 = 区间内所有数字 Abbi 值的累加和
// 比如[1, 2, 2, 3]的 Abbi 值 = 1 * 1 + 2 * 2 + 2 * 2 + 3 * 4 = 21
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]的区间 Abbi 值
// 数据范围: 1 <= n、m <= 5 * 10^5, 1 <= arr[i] <= 10^5
// 解题思路: 使用莫队二次离线算法处理区间 Abbi 值问题
// 时间复杂度: O(n*sqrt(n)*logn)
// 空间复杂度: O(n)
// 相关题目:
// 1. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 2. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 3. AtCoder AT1219 歴史の研究: https://www.luogu.com.cn/problem/AT1219
// 4. Codeforces 85D Sum of Medians: https://codeforces.com/problemset/problem/85/D

//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Query {
//    int l, r, id;
//};
//
//const int MAXN = 500001;
//const int MAXV = 100000;
//const int MAXB = 401;
//int n, m;
//int arr[MAXN];
//long long preSum[MAXN];
//
```

```
//int bi[MAXN];
//int bl[MAXB];
//int br[MAXB];
//
//Query query[MAXN];
//int headq[MAXN];
//int nextq[MAXN << 1];
//int ql[MAXN << 1];
//int qr[MAXN << 1];
//int qop[MAXN << 1];
//int qid[MAXN << 1];
//int cntq;
//
//long long treeCnt[MAXV + 1];
//long long treeSum[MAXV + 1];
//long long pre[MAXN];
//
//int blockLessCnt[MAXB];
//int numLessCnt[MAXN];
//
//long long blockMoreSum[MAXB];
//long long numMoreSum[MAXN];
//
//long long ans[MAXN];
//
//bool QueryCmp(Query &a, Query &b) {
//    if (bi[a.l] != bi[b.l]) {
//        return bi[a.l] < bi[b.l];
//    }
//    return a.r < b.r;
//}
//
//void addOffline(int x, int l, int r, int op, int id) {
//    nextq[++cntq] = headq[x];
//    headq[x] = cntq;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//int lowbit(int x) {
//    return x & -x;
```

```
//}
//
//void add(long long *tree, int x, long long v) {
//    while (x <= MAXV) {
//        tree[x] += v;
//        x += lowbit(x);
//    }
//}
//
//long long sum(long long *tree, int x) {
//    long long ret = 0;
//    while (x > 0) {
//        ret += tree[x];
//        x -= lowbit(x);
//    }
//    return ret;
//}
//
//void addVal(int val) {
//    for (int b = bi[val] + 1; b <= bi[MAXV]; b++) {
//        blockLessCnt[b]++;
//    }
//    for (int i = val + 1; i <= br[bi[val]]; i++) {
//        numLessCnt[i]++;
//    }
//    for (int b = 1; b <= bi[val] - 1; b++) {
//        blockMoreSum[b] += val;
//    }
//    for (int i = bl[bi[val]]; i < val; i++) {
//        numMoreSum[i] += val;
//    }
//}
//
//int lessCnt(int x) {
//    return blockLessCnt[bi[x]] + numLessCnt[x];
//}
//
//long long moreSum(int x) {
//    return blockMoreSum[bi[x]] + numMoreSum[x];
//}
//
//void prepare() {
//    for (int i = 1; i <= n; i++) {
```

```
//        preSum[i] = preSum[i - 1] + arr[i];
//    }
//    int blen = (int)sqrt(MAXV);
//    int bnum = (MAXV + blen - 1) / blen;
//    for (int i = 1; i <= MAXV; i++) {
//        bi[i] = (i - 1) / blen + 1;
//    }
//    for (int i = 1; i <= bnum; i++) {
//        bl[i] = (i - 1) * blen + 1;
//        br[i] = min(i * blen, MAXV);
//    }
//    sort(query + 1, query + m + 1, QueryCmp);
//}
//
//void compute() {
//    for (int i = 1; i <= n; i++) {
//        pre[i] = pre[i - 1] + sum(treeCnt, arr[i] - 1) * arr[i] + sum(treeSum, MAXV) -
sum(treeSum, arr[i]);
//        add(treeCnt, arr[i], 1);
//        add(treeSum, arr[i], arr[i]);
//    }
//    int winl = 1, winr = 0;
//    for (int i = 1; i <= m; i++) {
//        int jobl = query[i].l;
//        int jobr = query[i].r;
//        int id = query[i].id;
//        if (winr < jobr) {
//            addOffline(winl - 1, winr + 1, jobr, -1, id);
//            ans[id] += pre[jobr] - pre[winr];
//        }
//        if (winr > jobr) {
//            addOffline(winl - 1, jobr + 1, winr, 1, id);
//            ans[id] -= pre[winr] - pre[jobr];
//        }
//        winr = jobr;
//        if (winl > jobl) {
//            addOffline(winr, jobl, winl - 1, 1, id);
//            ans[id] -= pre[winl - 1] - pre[jobl - 1];
//        }
//        if (winl < jobl) {
//            addOffline(winr, winl, jobl - 1, -1, id);
//            ans[id] += pre[jobl - 1] - pre[winl - 1];
//        }
```

```
//        winl = jobl;
//    }
//    long long tmp;
//    for (int x = 0; x <= n; x++) {
//        if (x >= 1) {
//            addVal(arr[x]);
//        }
//        for (int q = headq[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            for (int j = l; j <= r; j++) {
//                tmp = 1LL * lessCnt(arr[j]) * arr[j] + moreSum(arr[j]);
//                if (op == 1) {
//                    ans[id] += tmp;
//                } else {
//                    ans[id] -= tmp;
//                }
//            }
//        }
//    }
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> query[i].l >> query[i].r;
//        query[i].id = i;
//    }
//    prepare();
//    compute();
//    for (int i = 2; i <= m; i++) {
//        ans[query[i].id] += ans[query[i - 1].id];
//    }
//    for (int i = 1; i <= m; i++) {
//        ans[query[i].id] += preSum[query[i].r] - preSum[query[i].l - 1];
//    }
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
```

```
//    return 0;
//}

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

文件: Code04_Gosick1.java
==========================================
package class178;

// 区间倍数二元组，java 版
// 题目来源: 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II
// 题目链接: https://www.luogu.com.cn/problem/P5398
// 题目大意: 给定一个长度为 n 的数组 arr，下面给出倍数二元组的定义
// 如果 arr[i]是 arr[j]的倍数(>=1 倍)，那么(i, j)就是一个倍数二元组
// 当 i != j 时，(i, j)和(j, i)认为是不同的二元组，不要漏算
// 当 i == j 时，(i, j)和(j, i)认为是相同的二元组，不要多算
// 比如[2, 4, 2, 6]，有 10 个倍数二元组
// 一共有 m 条查询，格式为 l r : 打印 arr[l..r]范围上，有多少倍数二元组
// 数据范围: 1 <= n、m、arr[i] <= 5 * 10^5
// 解题思路: 使用莫队二次离线算法处理区间倍数二元组问题
// 时间复杂度: O(n*sqrt(n)*sqrt(maxv))
// 空间复杂度: O(n + maxv*sqrt(maxv))
// 相关题目:
// 1. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II:
https://www.luogu.com.cn/problem/P5398
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 4. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class Code04_Gosick1 {

    public static int MAXN = 500001;
    public static int MAXF = 5000001;
    public static int LIMIT = 100;
    public static int n, m, maxv;
```

```
public static int[] arr = new int[MAXN];
public static int[] bi = new int[MAXN];

// 每个数的因子表，用链式前向星表达
public static int[] headf = new int[MAXN];
public static int[] nextf = new int[MAXF];
public static int[] fac = new int[MAXF];
public static int cntf;

// 莫队任务
public static int[][] query = new int[MAXN][3];

// 二次离线的任务，也是链式前向星
public static int[] headq = new int[MAXN];
public static int[] nextq = new int[MAXN << 1];
public static int[] qx = new int[MAXN << 1];
public static int[] ql = new int[MAXN << 1];
public static int[] qr = new int[MAXN << 1];
public static int[] qop = new int[MAXN << 1];
public static int[] qid = new int[MAXN << 1];
public static int cntq;

// xcnt[v] = 之前出现的数中，有多少数是此时数字 v 的倍数
public static int[] xcnt = new int[MAXN];
// vcnt[v] = 之前出现的数中，数字 v 出现了多少次
public static int[] vcnt = new int[MAXN];
// 前缀信息
public static long[] pre = new long[MAXN];

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

public static class QueryCmp implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (bi[a[0]] != bi[b[0]]) {
            return bi[a[0]] - bi[b[0]];
        }
        return a[1] - b[1];
    }
}

public static void addFactors(int num) {
    if (headf[num] == 0) {
```

```
        for (int f = 1; f * f <= num; f++) {
            if (num % f == 0) {
                nextf[++cntf] = headf[num];
                fac[cntf] = f;
                headf[num] = cntf;
            }
        }
    }
}

public static void addOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headq[x];
    headq[x] = cntq;
    qx[cntq] = x;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

public static void compute() {
    for (int i = 1; i <= n; i++) {
        int num = arr[i];
        pre[i] = pre[i - 1];
        pre[i] += xcnt[num];
        for (int e = headf[num], f, other; e > 0; e = nextf[e]) {
            f = fac[e];
            other = num / f;
            xcnt[f]++;
            pre[i] += vcnt[f];
            if (other != f) {
                xcnt[other]++;
                pre[i] += vcnt[other];
            }
        }
        vcnt[num]++;
    }
    // 第一次离线，执行莫队
    int winl = 1, winr = 0;
    for (int i = 1; i <= m; i++) {
        int jobl = query[i][0];
        int jobr = query[i][1];
        int id = query[i][2];
```

```
    if (winr < jobr) {
        addOffline(winl - 1, winr + 1, jobr, -1, id);
        ans[id] += pre[jobr] - pre[winr];
    }
    if (winr > jobr) {
        addOffline(winl - 1, jobr + 1, winr, 1, id);
        ans[id] -= pre[winr] - pre[jobr];
    }
    winr = jobr;
    // 接下来是winl滑动
    // 课上重点图解了，需要考虑 (2 * 滑动长度) 的修正
    if (winl > jobl) {
        addOffline(winr, jobl, winl - 1, 1, id);
        ans[id] -= pre[winl - 1] - pre[jobl - 1] + 2 * (winl - jobl);
    }
    if (winl < jobl) {
        addOffline(winr, winl, jobl - 1, -1, id);
        ans[id] += pre[jobl - 1] - pre[winl - 1] + 2 * (jobl - winl);
    }
    winl = jobl;
}
// 第二次离线，num倍数的数量 + num一部分因子的数量，都计入xcnt[num]
Arrays.fill(xcnt, 0);
for (int x = 0; x <= n; x++) {
    if (x >= 1) {
        int num = arr[x];
        for (int e = headf[num], f, other; e > 0; e = nextf[e]) {
            f = fac[e];
            other = num / f;
            xcnt[f]++;
            if (other != f) {
                xcnt[other]++;
            }
        }
        // 只处理大于LIMIT值的num
        if (num > LIMIT) {
            for (int v = num; v <= maxv; v += num) {
                xcnt[v]++;
            }
        }
    }
    for (int q = headq[x]; q > 0; q = nextq[q]) {
        int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
```

```
                for (int j = l; j <= r; j++) {
                    ans[id] += (long) op * xcnt[arr[j]];
                }
            }
        }
        // 第三次离线，1 ~ LIMIT 这些因子是少算了的，如今补回来
        for (int v = 1; v <= LIMIT; v++) {
            // 复用 vcnt 和 xcnt
            // vcnt[i]，当前表示，1~i 范围上，v 出现的次数
            // xcnt[i]，当前表示，1~i 范围上，v 的倍数出现的次数
            vcnt[0] = xcnt[0] = 0;
            for (int i = 1; i <= n; i++) {
                vcnt[i] = vcnt[i - 1] + (arr[i] == v ? 1 : 0);
                xcnt[i] = xcnt[i - 1] + (arr[i] % v == 0 ? 1 : 0);
            }
            for (int i = 1; i <= cntq; i++) {
                int x = qx[i], l = ql[i], r = qr[i], op = qop[i], id = qid[i];
                ans[id] += (long) op * vcnt[x] * (xcnt[r] - xcnt[l - 1]);
            }
        }
    }

    public static void prepare() {
        int blen = (int) Math.sqrt(n);
        for (int i = 1; i <= n; i++) {
            bi[i] = (i - 1) / blen + 1;
            maxv = Math.max(maxv, arr[i]);
            addFactors(arr[i]);
        }
        Arrays.sort(query, 1, m + 1, new QueryCmp());
    }

    public static void main(String[] args) throws Exception {
        FastReader in = new FastReader(System.in);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        n = in.nextInt();
        m = in.nextInt();
        for (int i = 1; i <= n; i++) {
            arr[i] = in.nextInt();
        }
        for (int i = 1; i <= m; i++) {
            query[i][0] = in.nextInt();
            query[i][1] = in.nextInt();
```

```
        query[i][2] = i;
    }
    prepare();
    compute();
    // 答案变化量生成前缀和
    for (int i = 2; i <= m; i++) {
        ans[query[i][2]] += ans[query[i - 1][2]];
    }
    // 贡献是重新定义的，答案需要补偿回来
    for (int i = 1; i <= m; i++) {
        ans[query[i][2]] += query[i][1] - query[i][0] + 1;
    }
    for (int i = 1; i <= m; i++) {
        out.println(ans[i]);
    }
    out.flush();
    out.close();
}

// 读写工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
```

```
            } while (c <= ' ' && c != -1);
            boolean neg = false;
            if (c == '-') {
                neg = true;
                c = readByte();
            }
            int val = 0;
            while (c > ' ' && c != -1) {
                val = val * 10 + (c - '0');
                c = readByte();
            }
            return neg ? -val : val;
        }
    }

}
```

==========

文件: Code04_Gosick2.java

==========

```
package class178;

// 区间倍数二元组，C++版
// 题目来源: 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II
// 题目链接: https://www.luogu.com.cn/problem/P5398
// 题目大意: 给定一个长度为n的数组arr，下面给出倍数二元组的定义
// 如果arr[i]是arr[j]的倍数(>=1倍)，那么(i, j)就是一个倍数二元组
// 当i != j时，(i, j)和(j, i)认为是不同的二元组，不要漏算
// 当i == j时，(i, j)和(j, i)认为是相同的二元组，不要多算
// 比如[2, 4, 2, 6]，有10个倍数二元组
// 一共有m条查询，格式为 l r : 打印arr[l..r]范围上，有多少倍数二元组
// 数据范围: 1 <= n、m、arr[i] <= 5 * 10^5
// 解题思路: 使用莫队二次离线算法处理区间倍数二元组问题
// 时间复杂度: O(n*sqrt(n)*sqrt(maxv))
// 空间复杂度: O(n + maxv*sqrt(maxv))
// 相关题目:
// 1. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II:
https://www.luogu.com.cn/problem/P5398
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
// 4. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E
```

```
//#include <bits/stdc++.h>
//
//using namespace std;
//
//struct Query {
//    int l, r, id;
//};
//
//const int MAXN = 500001;
//const int MAXF = 5000001;
//const int LIMIT = 100;
//int n, m, maxv;
//int arr[MAXN];
//int bi[MAXN];
//
//int headf[MAXN];
//int nextf[MAXF];
//int fac[MAXF];
//int cntf;
//
//Query query[MAXN];
//int headq[MAXN];
//int nextq[MAXN << 1];
//int qx[MAXN << 1];
//int ql[MAXN << 1];
//int qr[MAXN << 1];
//int qop[MAXN << 1];
//int qid[MAXN << 1];
//int cntq;
//
//int xcnt[MAXN];
//int vcnt[MAXN];
//long long pre[MAXN];
//
//long long ans[MAXN];
//
//bool QueryCmp(Query &a, Query &b) {
//    if (bi[a.l] != bi[b.l]) {
//        return bi[a.l] < bi[b.l];
//    }
//    return a.r < b.r;
//}
```

```
//
//void addFactors(int num) {
//    if (headf[num] == 0) {
//        for (int f = 1; f * f <= num; f++) {
//            if (num % f == 0) {
//                nextf[++cntf] = headf[num];
//                fac[cntf] = f;
//                headf[num] = cntf;
//            }
//        }
//    }
//}
//
//void addOffline(int x, int l, int r, int op, int id) {
//    nextq[++cntq] = headq[x];
//    headq[x] = cntq;
//    qx[cntq] = x;
//    ql[cntq] = l;
//    qr[cntq] = r;
//    qop[cntq] = op;
//    qid[cntq] = id;
//}
//
//void compute() {
//    for (int i = 1; i <= n; i++) {
//        int num = arr[i];
//        pre[i] = pre[i - 1];
//        pre[i] += xcnt[num];
//        for (int e = headf[num], f, other; e > 0; e = nextf[e]) {
//            f = fac[e];
//            other = num / f;
//            xcnt[f]++;
//            pre[i] += vcnt[f];
//            if (other != f) {
//                xcnt[other]++;
//                pre[i] += vcnt[other];
//            }
//        }
//        vcnt[num]++;
//    }
//    int winl = 1, winr = 0;
//    for (int i = 1; i <= m; i++) {
//        int jobl = query[i].l;
```

```
//        int jobr = query[i].r;
//        int id = query[i].id;
//        if (winr < jobr) {
//            addOffline(winl - 1, winr + 1, jobr, -1, id);
//            ans[id] += pre[jobr] - pre[winr];
//        }
//        if (winr > jobr) {
//            addOffline(winl - 1, jobr + 1, winr, 1, id);
//            ans[id] -= pre[winr] - pre[jobr];
//        }
//        winr = jobr;
//        if (winl > jobl) {
//            addOffline(winr, jobl, winl - 1, 1, id);
//            ans[id] -= pre[winl - 1] - pre[jobl - 1] + 2 * (winl - jobl);
//        }
//        if (winl < jobl) {
//            addOffline(winr, winl, jobl - 1, -1, id);
//            ans[id] += pre[jobl - 1] - pre[winl - 1] + 2 * (jobl - winl);
//        }
//        winl = jobl;
//    }
//    memset(xcnt, 0, sizeof(xcnt));
//    for (int x = 0; x <= n; x++) {
//        if (x >= 1) {
//            int num = arr[x];
//            for (int e = headf[num], f, other; e > 0; e = nextf[e]) {
//                f = fac[e];
//                other = num / f;
//                xcnt[f]++;
//                if (other != f) {
//                    xcnt[other]++;
//                }
//            }
//            if (num > LIMIT) {
//                for (int v = num; v <= maxv; v += num) {
//                    xcnt[v]++;
//                }
//            }
//        }
//        for (int q = headq[x]; q > 0; q = nextq[q]) {
//            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
//            for (int j = l; j <= r; j++) {
//                ans[id] += 1LL * op * xcnt[arr[j]];
```

```
//            }
//        }
//    }
//    for (int v = 1; v <= LIMIT; v++) {
//        vcnt[0] = xcnt[0] = 0;
//        for (int i = 1; i <= n; i++) {
//            vcnt[i] = vcnt[i - 1] + (arr[i] == v ? 1 : 0);
//            xcnt[i] = xcnt[i - 1] + (arr[i] % v == 0 ? 1 : 0);
//        }
//        for(int i = 1; i <= cntq; i++) {
//             int x = qx[i], l = ql[i], r = qr[i], op = qop[i], id = qid[i];
//             ans[id] += 1LL * op * vcnt[x] * (xcnt[r] - xcnt[l - 1]);
//        }
//    }
//}
//
//void prepare() {
//    int blen = (int)sqrt(n);
//    for (int i = 1; i <= n; i++) {
//        bi[i] = (i - 1) / blen + 1;
//        maxv = max(maxv, arr[i]);
//        addFactors(arr[i]);
//    }
//    sort(query + 1, query + m + 1, QueryCmp);
//}
//
//int main() {
//    ios::sync_with_stdio(false);
//    cin.tie(nullptr);
//    cin >> n >> m;
//    for (int i = 1; i <= n; i++) {
//        cin >> arr[i];
//    }
//    for (int i = 1; i <= m; i++) {
//        cin >> query[i].l >> query[i].r;
//        query[i].id = i;
//    }
//    prepare();
//    compute();
//    for (int i = 2; i <= m; i++) {
//        ans[query[i].id] += ans[query[i - 1].id];
//    }
//    for (int i = 1; i <= m; i++) {
```

```
//        ans[query[i].id] += query[i].r - query[i].l + 1;
//    }
//    for (int i = 1; i <= m; i++) {
//        cout << ans[i] << '\n';
//    }
//    return 0;
//}
```

==========

文件: Codeforces617E_XORAndFavoriteNumber1.cpp

==========

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

/**
 * Codeforces 617E XOR and Favorite Number
 * 题目链接：https://codeforces.com/contest/617/problem/E
 *
 * 题目描述：
 * 给定一个长度为n的数组arr和一个目标值k，有m次查询。
 * 每次查询给出区间[l,r]，求区间内有多少个连续子数组的异或和等于k。
 *
 * 输入格式：
 * 第一行三个整数n, m, k
 * 第二行n个整数表示数组arr
 * 接下来m行，每行两个整数l, r表示查询区间
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案
 *
 * 数据范围：
 * 1 <= n, m <= 100000
 * 0 <= arr[i], k < 1000000
 *
 * 解题思路：
 * 1. 前缀异或数组：使用前缀异或数组xor_sum，其中xor_sum[i]表示前i个元素的异或和
 * 2. 对于子数组[l,r]的异或和为xor_sum[r] ^ xor_sum[l-1]
```

```
 * 3. 我们需要统计在区间[l-1,r]中有多少对(i,j)满足 xor_sum[j] ^ xor_sum[i] == k，其中 i < j
 * 4. 使用莫队算法维护当前区间内各个 xor 值的出现次数
 * 5. 当扩展区间时，更新计数并累加答案
 *
 * 时间复杂度：O((n + m) * sqrt(n))
 * 空间复杂度：O(n + max_xor_value)
 */

const int MAXN = 100010;
const int MAXK = 1000010;

struct Query {
    int l, r, id;
    long long ans;
};

int n, m, k;
int arr[MAXN];
long long xor_sum[MAXN];
Query q[MAXN];
long long cnt[MAXK];  // 统计每个异或值的出现次数
int block_size;

// 快速输入函数，提高输入效率
template<typename T>
void read(T &x) {
    x = 0;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + (ch - '0');
        ch = getchar();
    }
}

// 比较函数，用于莫队查询的排序
bool compare(const Query &a, const Query &b) {
    if (a.l / block_size != b.l / block_size) {
        return a.l < b.l;
    }
    // 奇偶优化：奇数块右端点升序，偶数块右端点降序
```

```
    return (a.l / block_size & 1) == 0 ? a.r < b.r : a.r > b.r;
}

// 更新当前区间的统计信息和答案
void update(int pos, long long &res, bool add) {
    long long current_xor = xor_sum[pos];
    if (add) {
        // 添加一个元素时，增加对应的配对数量
        res += cnt[(current_xor ^ k) % MAXK];
        cnt[(current_xor) % MAXK]++;
    } else {
        // 删除一个元素时，减少对应的配对数量
        cnt[(current_xor) % MAXK]--;
        res -= cnt[(current_xor ^ k) % MAXK];
    }
}

int main() {
    // 读取输入
    read(n);
    read(m);
    read(k);

    // 计算前缀异或数组
    xor_sum[0] = 0;
    for (int i = 1; i <= n; i++) {
        read(arr[i]);
        xor_sum[i] = xor_sum[i - 1] ^ arr[i];
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        int l, r;
        read(l);
        read(r);
        q[i].l = l - 1;  // 转换为前缀异或数组的索引
        q[i].r = r;
        q[i].id = i;
    }

    // 设置块的大小
    block_size = sqrt(n) + 1;
```

```
    // 对查询进行排序
    sort(q, q + m, compare);

    // 初始化指针和计数器
    int curL = 0, curR = -1;
    long long res = 0;

    // 莫队算法处理
    for (int i = 0; i < m; i++) {
        Query &query = q[i];

        // 扩展或收缩区间
        while (curL > query.l) update(--curL, res, true);
        while (curR < query.r) update(++curR, res, true);
        while (curL < query.l) update(curL++, res, false);
        while (curR > query.r) update(curR--, res, false);

        // 保存当前查询的答案
        query.ans = res;
    }

    // 将答案按原顺序输出
    long long output[MAXN];
    for (int i = 0; i < m; i++) {
        output[q[i].id] = q[i].ans;
    }

    for (int i = 0; i < m; i++) {
        printf("%lld\n", output[i]);
    }

    return 0;
}

/*
 * 算法分析：
 * 时间复杂度：O((n + m) * sqrt(n))
 * - 排序查询的时间复杂度：O(m * log m)
 * - 莫队算法处理的时间复杂度：每个元素最多被访问O(sqrt(n))次，总时间为O(n * sqrt(n))
 * - 整体时间复杂度：O(m * log m + n * sqrt(n))，通常m和n同阶，所以为O((n + m) * sqrt(n))
 *
 * 空间复杂度：O(n + MAXK)，其中MAXK是最大可能的异或值
 * - 前缀异或数组：O(n)
```

```
 * - 查询数组：O(m)
 * - 计数数组：O(MAXK)
 *
 * 优化点：
 * 1. 使用了奇偶优化，减少块间转移的时间
 * 2. 使用了快速输入函数，提高输入效率
 * 3. 模数处理防止数组越界
 *
 * 边界情况处理：
 * 1. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
 * 2. 异或值可能超过计数数组大小，使用模运算处理
 *
 * 工程化考量：
 * 1. 使用静态数组代替动态分配，提高内存访问效率
 * 2. 使用 printf 进行输出，提高输出效率
 * 3. 模板函数 read 处理不同类型的输入，提高代码复用性
 *
 * 调试技巧：
 * 1. 可以在 update 函数中输出中间状态，检查计数是否正确
 * 2. 测试用例：如 n=3, m=1, k=3, arr=[1, 2, 3]，预期结果为 2（子数组[1,2]和[3]）
 */

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

文件: Codeforces617E_XORAndFavoriteNumber1.java
==========================================
import java.io.*;
import java.util.*;

/**
 * Codeforces 617E XOR and Favorite Number
 * 题目链接：https://codeforces.com/contest/617/problem/E
 *
 * 题目描述：
 * 给定一个长度为 n 的数组 arr 和一个目标值 k，有 m 次查询。
 * 每次查询给出区间[l,r]，求区间内有多少个连续子数组的异或和等于 k。
 *
 * 输入格式：
 * 第一行三个整数 n, m, k
 * 第二行 n 个整数表示数组 arr
 * 接下来 m 行，每行两个整数 l, r 表示查询区间
 *
 * 输出格式：
```

```
 * 对于每个查询，输出一行一个整数表示答案
 *
 * 数据范围：
 * 1 <= n, m <= 100000
 * 0 <= arr[i], k < 1000000
 *
 * 解题思路：
 * 1. 前缀异或数组：使用前缀异或数组 xor，其中 xor[i]表示前 i 个元素的异或和
 * 2. 对于子数组[l,r]的异或和为 xor[r] ^ xor[l-1]
 * 3. 我们需要统计在区间[l-1,r]中有多少对(i,j)满足 xor[j] ^ xor[i] == k，其中 i < j
 * 4. 使用莫队算法维护当前区间内各个 xor 值的出现次数
 * 5. 当扩展区间时，更新计数并累加答案
 *
 * 时间复杂度：O((n + m) * sqrt(n))
 * 空间复杂度：O(n + max_xor_value)
 */
public class Codeforces617E_XORAndFavoriteNumber1 {
    static final int MAXN = 100010;
    static final int MAXK = 1000010;

    // 输入数据
    static int n, m, k;
    static int[] arr = new int[MAXN];
    static long[] xor = new long[MAXN];  // 前缀异或数组

    // 莫队算法相关
    static class Query {
        int l, r, id;  // 查询区间和索引
        long ans;      // 存储答案
    }
    static Query[] q = new Query[MAXN];
    static int blockSize;  // 块的大小
    static long[] cnt = new long[MAXK];  // 统计每个异或值的出现次数

    // 快速输入类，提高输入效率
    static class FastReader {
        BufferedReader br;
        StringTokenizer st;

        public FastReader() {
            br = new BufferedReader(new InputStreamReader(System.in));
            st = null;
        }
```

```
    public int nextInt() throws IOException {
        while (st == null || !st.hasMoreTokens()) {
            st = new StringTokenizer(br.readLine());
        }
        return Integer.parseInt(st.nextToken());
    }

    public void close() throws IOException {
        br.close();
    }
}

// 比较器，用于莫队查询的排序
static class QueryComparator implements Comparator<Query> {
    @Override
    public int compare(Query a, Query b) {
        // 按左端点所在块排序，同一块按右端点排序
        if (a.l / blockSize != b.l / blockSize) {
            return a.l / blockSize - b.l / blockSize;
        }
        // 奇偶优化：奇数块右端点升序，偶数块右端点降序
        return (a.l / blockSize & 1) == 0 ? a.r - b.r : b.r - a.r;
    }
}

// 更新当前区间的统计信息和答案
static void update(int pos, long[] res, boolean add) {
    long current_xor = xor[pos];
    if (add) {
        // 添加一个元素时，增加对应的配对数量
        res[0] += cnt[(current_xor ^ k) % MAXK];
        cnt[(current_xor) % MAXK]++;
    } else {
        // 删除一个元素时，减少对应的配对数量
        cnt[(current_xor) % MAXK]--;
        res[0] -= cnt[(current_xor ^ k) % MAXK];
    }
}

public static void main(String[] args) throws IOException {
    FastReader reader = new FastReader();
    n = reader.nextInt();
```

```
m = reader.nextInt();
k = reader.nextInt();

// 计算前缀异或数组
xor[0] = 0;
for (int i = 1; i <= n; i++) {
    arr[i] = reader.nextInt();
    xor[i] = xor[i - 1] ^ arr[i];
}

// 读取查询
for (int i = 0; i < m; i++) {
    q[i] = new Query();
    q[i].l = reader.nextInt() - 1;  // 转换为前缀异或数组的索引
    q[i].r = reader.nextInt();
    q[i].id = i;
}

// 设置块的大小，通常为 sqrt(n)
blockSize = (int) Math.sqrt(n) + 1;

// 对查询进行排序
Arrays.sort(q, 0, m, new QueryComparator());

// 初始化指针和计数器
int curL = 0, curR = -1;
long[] res = new long[1];  // 使用数组存储结果，以便在 update 中修改

// 莫队算法处理
for (int i = 0; i < m; i++) {
    Query query = q[i];

    // 扩展或收缩区间
    while (curL > query.l) update(--curL, res, true);
    while (curR < query.r) update(++curR, res, true);
    while (curL < query.l) update(curL++, res, false);
    while (curR > query.r) update(curR--, res, false);

    // 保存当前查询的答案
    query.ans = res[0];
}

// 将答案按原顺序输出
```

```
        long[] output = new long[m];
        for (int i = 0; i < m; i++) {
            output[q[i].id] = q[i].ans;
        }

        PrintWriter writer = new PrintWriter(System.out);
        for (int i = 0; i < m; i++) {
            writer.println(output[i]);
        }
        writer.flush();
        writer.close();
        reader.close();
    }

    /*
     * 算法分析：
     * 时间复杂度：O((n + m) * sqrt(n))
     * - 排序查询的时间复杂度：O(m * log m)
     * - 莫队算法处理的时间复杂度：每个元素最多被访问O(sqrt(n))次，总时间为O(n * sqrt(n))
     * - 整体时间复杂度：O(m * log m + n * sqrt(n))，通常m和n同阶，所以为O((n + m) * sqrt(n))
     *
     * 空间复杂度：O(n + MAXK)，其中MAXK是最大可能的异或值
     * - 前缀异或数组：O(n)
     * - 查询数组：O(m)
     * - 计数数组：O(MAXK)
     *
     * 优化点：
     * 1. 使用了奇偶优化，减少块间转移的时间
     * 2. 使用了快速输入类，提高输入效率
     * 3. 模数处理防止数组越界
     *
     * 边界情况处理：
     * 1. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
     * 2. 异或值可能超过计数数组大小，使用模运算处理
     *
     * 工程化考量：
     * 1. 使用静态数组代替动态分配，提高内存访问效率
     * 2. 使用PrintWriter进行批量输出，提高输出效率
     * 3. 资源管理：及时关闭输入输出流
     *
     * 调试技巧：
     * 1. 可以在update函数中输出中间状态，检查计数是否正确
     * 2. 测试用例：如n=3, m=1, k=3, arr=[1, 2, 3]，预期结果为2（子数组[1,2]和[3]）
```

```
    */
}

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

文件: Codeforces617E_XORAndFavoriteNumber1.py
==========================================
import sys
import math

"""
Codeforces 617E XOR and Favorite Number
题目链接：https://codeforces.com/contest/617/problem/E

题目描述：
给定一个长度为n的数组arr和一个目标值k，有m次查询。
每次查询给出区间[l,r]，求区间内有多少个连续子数组的异或和等于k。

输入格式：
第一行三个整数n, m, k
第二行n个整数表示数组arr
接下来m行，每行两个整数l, r表示查询区间

输出格式：
对于每个查询，输出一行一个整数表示答案

数据范围：
1 <= n, m <= 100000
0 <= arr[i], k < 1000000

解题思路：
1. 前缀异或数组：使用前缀异或数组xor_sum，其中xor_sum[i]表示前i个元素的异或和
2. 对于子数组[l,r]的异或和为xor_sum[r] ^ xor_sum[l-1]
3. 我们需要统计在区间[l-1,r]中有多少对(i,j)满足xor_sum[j] ^ xor_sum[i] == k，其中i < j
4. 使用莫队算法维护当前区间内各个xor值的出现次数
5. 当扩展区间时，更新计数并累加答案

时间复杂度：O((n + m) * sqrt(n))
空间复杂度：O(n + m)
"""

def main():
    # 读取输入，使用sys.stdin.readline提高输入效率
```

```
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1
    k = int(input[ptr])
    ptr += 1

    # 计算前缀异或数组
    xor_sum = [0] * (n + 1)
    for i in range(1, n + 1):
        arr_val = int(input[ptr])
        ptr += 1
        xor_sum[i] = xor_sum[i - 1] ^ arr_val

    # 读取查询
    queries = []
    for i in range(m):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
        # 转换为前缀异或数组的索引
        queries.append((l - 1, r, i))

    # 设置块的大小
    block_size = int(math.sqrt(n)) + 1

    # 对查询进行排序
    # 按左端点所在块排序，同一块按右端点排序（奇偶优化）
    queries.sort(key=lambda q: (q[0] // block_size, q[1] if (q[0] // block_size) % 2 == 0 else -
q[1]))

    # 初始化指针、计数器和结果
    cur_l = 0
    cur_r = -1
    res = 0
    cnt = {}  # 使用字典代替数组，更灵活地处理大的异或值

    # 存储每个查询的答案
    answers = [0] * m
```

```
    def update(pos, add):
        nonlocal res
        current_xor = xor_sum[pos]
        if add:
            # 添加一个元素时，增加对应的配对数量
            target = current_xor ^ k
            if target in cnt:
                res += cnt[target]
            # 更新当前异或值的计数
            cnt[current_xor] = cnt.get(current_xor, 0) + 1
        else:
            # 删除一个元素时，先减少计数
            cnt[current_xor] -= 1
            if cnt[current_xor] == 0:
                del cnt[current_xor]
            # 然后减少对应的配对数量
            target = current_xor ^ k
            if target in cnt:
                res -= cnt[target]

    # 莫队算法处理
    for q in queries:
        l, r, idx = q

        # 扩展或收缩区间
        while cur_l > l:
            cur_l -= 1
            update(cur_l, True)
        while cur_r < r:
            cur_r += 1
            update(cur_r, True)
        while cur_l < l:
            update(cur_l, False)
            cur_l += 1
        while cur_r > r:
            update(cur_r, False)
            cur_r -= 1

        # 保存当前查询的答案
        answers[idx] = res

    # 输出结果
    sys.stdout.write('\n'.join(map(str, answers)) + '\n')
```

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

'''
```

算法分析：
时间复杂度：O((n + m) * sqrt(n))
- 排序查询的时间复杂度：O(m * log m)
- 莫队算法处理的时间复杂度：每个元素最多被访问 O(sqrt(n))次，总时间为 O(n * sqrt(n))
- 整体时间复杂度：O(m * log m + n * sqrt(n))，通常 m 和 n 同阶，所以为 O((n + m) * sqrt(n))

空间复杂度：O(n + m)
- 前缀异或数组：O(n)
- 查询数组和答案数组：O(m)
- 计数字典：最坏情况下 O(n)

优化点：
1. 使用了奇偶优化，减少块间转移的时间
2. 使用 sys.stdin.read().split()一次性读取所有输入，提高输入效率
3. 使用字典代替固定大小的数组，避免处理大数据范围的问题

边界情况处理：
1. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
2. 使用字典的 get 方法处理可能不存在的键
3. 当计数减为 0 时，删除该键以节省空间

工程化考量：
1. 使用 nonlocal 关键字在内部函数中修改外部函数的变量
2. 一次性读取所有输入并使用指针访问，提高输入效率
3. 使用 sys.stdout.write 进行批量输出，提高输出效率

调试技巧：
1. 可以在 update 函数中添加打印语句，检查计数是否正确
2. 测试用例：如 n=3, m=1, k=3, arr=[1, 2, 3]，预期结果为 2（子数组[1,2]和[3]）
3. 在处理大数据时，Python 版本可能需要进一步优化以通过时间限制

```
'''

==========

文件: Codeforces86D_PowerfulArray1.cpp
==========
#include <iostream>
#include <vector>
```

```
#include <algorithm>
#include <cmath>
#include <cstring>
using namespace std;

/**
 * Codeforces 86D Powerful array
 * 题目链接：https://codeforces.com/problemset/problem/86/D
 *
 * 题目描述：
 * 给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内，
 * 每种数字出现次数的平方和乘以该数字的值的总和。
 *
 * 输入格式：
 * 第一行两个整数 n 和 m，表示数组长度和查询次数。
 * 第二行 n 个整数表示数组元素。
 * 接下来 m 行，每行两个整数 l, r 表示查询区间。
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案。
 *
 * 数据范围：
 * 1 <= n, m <= 200000
 * 1 <= a[i] <= 10^6
 *
 * 解题思路：
 * 1. 使用莫队算法处理区间查询
 * 2. 维护当前区间内每个数字的出现次数 cnt[x]
 * 3. 维护当前答案 ans，当添加或删除一个元素 x 时，更新 ans：
 *    - 添加 x：ans += x * (2 * cnt[x] + 1)，然后 cnt[x]++
 *    - 删除 x：cnt[x]--，然后 ans -= x * (2 * cnt[x] + 1)
 * 4. 使用分块排序优化莫队算法的效率
 *
 * 时间复杂度：O((n + m) * sqrt(n))
 * 空间复杂度：O(n + max_value)
 */

const int MAXN = 200010;
const int MAXV = 1000010;

struct Query {
    int l, r, id;
    long long ans;
```

```
};

int n, m;
int arr[MAXN];
Query q[MAXN];
long long cnt[MAXV];  // 统计每个数字的出现次数
int block_size;

// 快速输入函数，提高输入效率
template<typename T>
void read(T &x) {
    x = 0;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + (ch - '0');
        ch = getchar();
    }
}

// 比较函数，用于莫队查询的排序
bool compare(const Query &a, const Query &b) {
    if (a.l / block_size != b.l / block_size) {
        return a.l < b.l;
    }
    // 奇偶优化：奇数块右端点升序，偶数块右端点降序
    return (a.l / block_size & 1) == 0 ? a.r < b.r : a.r > b.r;
}

// 更新当前区间的统计信息和答案
void update(int pos, long long &res, bool add) {
    int x = arr[pos];
    if (add) {
        // 添加一个元素 x，先更新答案，再增加计数
        res += (long long) x * (2 * cnt[x] + 1);
        cnt[x]++;
    } else {
        // 删除一个元素 x，先减少计数，再更新答案
        cnt[x]--;
        res -= (long long) x * (2 * cnt[x] + 1);
    }
```

```
}

int main() {
    // 读取输入
    read(n);
    read(m);

    // 读取数组
    for (int i = 1; i <= n; i++) {
        read(arr[i]);
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        int l, r;
        read(l);
        read(r);
        q[i].l = l;
        q[i].r = r;
        q[i].id = i;
    }

    // 设置块的大小
    block_size = sqrt(n) + 1;

    // 对查询进行排序
    sort(q, q + m, compare);

    // 初始化指针和计数器
    int curL = 1, curR = 0;
    long long res = 0;

    // 莫队算法处理
    for (int i = 0; i < m; i++) {
        Query &query = q[i];

        // 扩展或收缩区间
        while (curL > query.l) update(--curL, res, true);
        while (curR < query.r) update(++curR, res, true);
        while (curL < query.l) update(curL++, res, false);
        while (curR > query.r) update(curR--, res, false);

        // 保存当前查询的答案
```

```
        query.ans = res;
    }

    // 将答案按原顺序输出
    long long output[MAXN];
    for (int i = 0; i < m; i++) {
        output[q[i].id] = q[i].ans;
    }

    for (int i = 0; i < m; i++) {
        printf("%lld\n", output[i]);
    }

    return 0;
}

/*
 * 算法分析：
 * 时间复杂度：O((n + m) * sqrt(n))
 * - 排序查询的时间复杂度：O(m * log m)
 * - 莫队算法处理的时间复杂度：每个元素最多被访问O(sqrt(n))次，总时间为O(n * sqrt(n))
 * - 整体时间复杂度：O(m * log m + n * sqrt(n))，通常m和n同阶，所以为O((n + m) * sqrt(n))
 *
 * 空间复杂度：O(n + MAXV)，其中MAXV是最大可能的数组元素值
 * - 数组存储：O(n)
 * - 查询数组：O(m)
 * - 计数数组：O(MAXV)
 *
 * 优化点：
 * 1. 使用了奇偶优化，减少块间转移的时间
 * 2. 使用了快速输入函数，提高输入效率
 * 3. 使用long long类型存储结果，避免溢出
 *
 * 边界情况处理：
 * 1. 数组元素的值可能很大，但通过使用适当大小的计数数组处理
 * 2. 结果可能超出int范围，使用long long类型存储
 *
 * 工程化考量：
 * 1. 使用静态数组代替动态分配，提高内存访问效率
 * 2. 使用printf进行输出，提高输出效率
 * 3. 模板函数read处理不同类型的输入，提高代码复用性
 *
 * 调试技巧：
```

```
 * 1. 可以在 update 函数中输出中间状态，检查计数和答案是否正确
 * 2. 测试用例：如 n=3, m=1, arr=[1, 2, 1]，查询[1,3]，预期结果为 1*2^2 + 2*1^2 = 4 + 2 = 6
 */

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

文件: Codeforces86D_PowerfulArray1.java
==========================================
import java.io.*;
import java.util.*;

/**
 * Codeforces 86D Powerful array
 * 题目链接：https://codeforces.com/problemset/problem/86/D
 *
 * 题目描述：
 * 给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内，
 * 每种数字出现次数的平方和乘以该数字的值的总和。
 *
 * 输入格式：
 * 第一行两个整数 n 和 m，表示数组长度和查询次数。
 * 第二行 n 个整数表示数组元素。
 * 接下来 m 行，每行两个整数 l, r 表示查询区间。
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案。
 *
 * 数据范围：
 * 1 <= n, m <= 200000
 * 1 <= a[i] <= 10^6
 *
 * 解题思路：
 * 1. 使用莫队算法处理区间查询
 * 2. 维护当前区间内每个数字的出现次数 cnt[x]
 * 3. 维护当前答案 ans，当添加或删除一个元素 x 时，更新 ans：
 *    - 添加 x：ans += x * (2 * cnt[x] + 1)，然后 cnt[x]++
 *    - 删除 x：cnt[x]--，然后 ans -= x * (2 * cnt[x] + 1)
 * 4. 使用分块排序优化莫队算法的效率
 *
 * 时间复杂度：O((n + m) * sqrt(n))
 * 空间复杂度：O(n + max_value)
 */
public class Codeforces86D_PowerfulArray1 {
```

```
static final int MAXN = 200010;
static final int MAXV = 1000010;

// 输入数据
static int n, m;
static int[] arr = new int[MAXN];

// 莫队算法相关
static class Query {
    int l, r, id;  // 查询区间和索引
    long ans;      // 存储答案
}
static Query[] q = new Query[MAXN];
static int blockSize;  // 块的大小
static long[] cnt = new long[MAXV];  // 统计每个数字的出现次数

// 快速输入类，提高输入效率
static class FastReader {
    BufferedReader br;
    StringTokenizer st;

    public FastReader() {
        br = new BufferedReader(new InputStreamReader(System.in));
        st = null;
    }

    public int nextInt() throws IOException {
        while (st == null || !st.hasMoreTokens()) {
            st = new StringTokenizer(br.readLine());
        }
        return Integer.parseInt(st.nextToken());
    }

    public void close() throws IOException {
        br.close();
    }
}

// 比较器，用于莫队查询的排序
static class QueryComparator implements Comparator<Query> {
    @Override
    public int compare(Query a, Query b) {
        // 按左端点所在块排序，同一块按右端点排序
```

```
        if (a.l / blockSize != b.l / blockSize) {
            return a.l / blockSize - b.l / blockSize;
        }
        // 奇偶优化：奇数块右端点升序，偶数块右端点降序
        return (a.l / blockSize & 1) == 0 ? a.r - b.r : b.r - a.r;
    }
}

// 更新当前区间的统计信息和答案
static void update(int pos, long[] res, boolean add) {
    int x = arr[pos];
    if (add) {
        // 添加一个元素 x，先更新答案，再增加计数
        res[0] += (long) x * (2 * cnt[x] + 1);
        cnt[x]++;
    } else {
        // 删除一个元素 x，先减少计数，再更新答案
        cnt[x]--;
        res[0] -= (long) x * (2 * cnt[x] + 1);
    }
}

public static void main(String[] args) throws IOException {
    FastReader reader = new FastReader();
    n = reader.nextInt();
    m = reader.nextInt();

    // 读取数组
    for (int i = 1; i <= n; i++) {
        arr[i] = reader.nextInt();
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        q[i] = new Query();
        q[i].l = reader.nextInt();
        q[i].r = reader.nextInt();
        q[i].id = i;
    }

    // 设置块的大小，通常为 sqrt(n)
    blockSize = (int) Math.sqrt(n) + 1;
```

```
        // 对查询进行排序
        Arrays.sort(q, 0, m, new QueryComparator());

        // 初始化指针和计数器
        int curL = 1, curR = 0;
        long[] res = new long[1];  // 使用数组存储结果，以便在 update 中修改

        // 莫队算法处理
        for (int i = 0; i < m; i++) {
            Query query = q[i];

            // 扩展或收缩区间
            while (curL > query.l) update(--curL, res, true);
            while (curR < query.r) update(++curR, res, true);
            while (curL < query.l) update(curL++, res, false);
            while (curR > query.r) update(curR--, res, false);

            // 保存当前查询的答案
            query.ans = res[0];
        }

        // 将答案按原顺序输出
        long[] output = new long[m];
        for (int i = 0; i < m; i++) {
            output[q[i].id] = q[i].ans;
        }

        PrintWriter writer = new PrintWriter(System.out);
        for (int i = 0; i < m; i++) {
            writer.println(output[i]);
        }
        writer.flush();
        writer.close();
        reader.close();
    }

    /*
     * 算法分析：
     * 时间复杂度：O((n + m) * sqrt(n))
     * - 排序查询的时间复杂度：O(m * log m)
     * - 莫队算法处理的时间复杂度：每个元素最多被访问 O(sqrt(n))次，总时间为 O(n * sqrt(n))
     * - 整体时间复杂度：O(m * log m + n * sqrt(n))，通常 m 和 n 同阶，所以为 O((n + m) * sqrt(n))
     *
```

```
     * 空间复杂度：O(n + MAXV)，其中 MAXV 是最大可能的数组元素值
     * - 数组存储：O(n)
     * - 查询数组：O(m)
     * - 计数数组：O(MAXV)
     *
     * 优化点：
     * 1. 使用了奇偶优化，减少块间转移的时间
     * 2. 使用了快速输入类，提高输入效率
     * 3. 使用 long 类型存储结果，避免溢出
     *
     * 边界情况处理：
     * 1. 数组元素的值可能很大，但通过使用适当大小的计数数组处理
     * 2. 结果可能超出 int 范围，使用 long 类型存储
     *
     * 工程化考量：
     * 1. 使用静态数组代替动态分配，提高内存访问效率
     * 2. 使用 PrintWriter 进行批量输出，提高输出效率
     * 3. 资源管理：及时关闭输入输出流
     *
     * 调试技巧：
     * 1. 可以在 update 函数中输出中间状态，检查计数和答案是否正确
     * 2. 测试用例：如 n=3, m=1, arr=[1, 2, 1]，查询[1,3]，预期结果为 1*2^2 + 2*1^2 = 4 + 2 = 6
     */
}

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

文件: Codeforces86D_PowerfulArray1.py
==============================================
import sys
import math

"""
Codeforces 86D Powerful array
题目链接：https://codeforces.com/problemset/problem/86/D

题目描述：
给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内，
每种数字出现次数的平方和乘以该数字的值的总和。

输入格式：
第一行两个整数 n 和 m，表示数组长度和查询次数。
第二行 n 个整数表示数组元素。
```

接下来 m 行，每行两个整数 l, r 表示查询区间。

输出格式：
对于每个查询，输出一行一个整数表示答案。

数据范围：
1 <= n, m <= 200000
1 <= a[i] <= 10^6

解题思路：
1. 使用莫队算法处理区间查询
2. 维护当前区间内每个数字的出现次数 cnt[x]
3. 维护当前答案 ans，当添加或删除一个元素 x 时，更新 ans：
   - 添加 x：ans += x * (2 * cnt[x] + 1)，然后 cnt[x]++
   - 删除 x：cnt[x]--，然后 ans -= x * (2 * cnt[x] + 1)
4. 使用分块排序优化莫队算法的效率

时间复杂度：O((n + m) * sqrt(n))
空间复杂度：O(n + m + max_value)
"""

```
def main():
    # 读取输入，使用 sys.stdin.readline 提高输入效率
    input = sys.stdin.read().split()
    ptr = 0
    n = int(input[ptr])
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    # 读取数组
    arr = [0] * (n + 1)  # 1-indexed
    for i in range(1, n + 1):
        arr[i] = int(input[ptr])
        ptr += 1

    # 读取查询
    queries = []
    for i in range(m):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
```

```
        queries.append((l, r, i))

    # 设置块的大小
    block_size = int(math.sqrt(n)) + 1

    # 对查询进行排序
    # 按左端点所在块排序，同一块按右端点排序（奇偶优化）
    queries.sort(key=lambda q: (q[0] // block_size, q[1] if (q[0] // block_size) % 2 == 0 else -q[1]))

    # 初始化指针、计数器和结果
    cur_l = 1
    cur_r = 0
    res = 0
    # 使用字典代替数组，因为数组元素值可能很大
    cnt = {}

    # 存储每个查询的答案
    answers = [0] * m

    def update(pos, add):
        nonlocal res
        x = arr[pos]
        if add:
            # 添加一个元素 x，先更新答案，再增加计数
            current_count = cnt.get(x, 0)
            res += x * (2 * current_count + 1)
            cnt[x] = current_count + 1
        else:
            # 删除一个元素 x，先减少计数，再更新答案
            current_count = cnt[x]
            cnt[x] = current_count - 1
            if cnt[x] == 0:
                del cnt[x]
            res -= x * (2 * (current_count - 1) + 1)

    # 莫队算法处理
    for q in queries:
        l, r, idx = q

        # 扩展或收缩区间
        while cur_l > l:
            cur_l -= 1
```

```
            update(cur_l, True)
        while cur_r < r:
            cur_r += 1
            update(cur_r, True)
        while cur_l < l:
            update(cur_l, False)
            cur_l += 1
        while cur_r > r:
            update(cur_r, False)
            cur_r -= 1

        # 保存当前查询的答案
        answers[idx] = res

    # 输出结果
    sys.stdout.write('\n'.join(map(str, answers)) + '\n')

if __name__ == "__main__":
    main()

'''
```

算法分析：
时间复杂度：O((n + m) * sqrt(n))
- 排序查询的时间复杂度：O(m * log m)
- 莫队算法处理的时间复杂度：每个元素最多被访问 O(sqrt(n))次，总时间为 O(n * sqrt(n))
- 整体时间复杂度：O(m * log m + n * sqrt(n))，通常 m 和 n 同阶，所以为 O((n + m) * sqrt(n))

空间复杂度：O(n + m)
- 数组存储：O(n)
- 查询数组和答案数组：O(m)
- 计数字典：最坏情况下 O(n)

优化点：
1. 使用了奇偶优化，减少块间转移的时间
2. 使用 sys.stdin.read().split()一次性读取所有输入，提高输入效率
3. 使用字典代替固定大小的数组，避免处理大数据范围的问题

边界情况处理：
1. 使用字典的 get 方法处理可能不存在的键
2. 当计数减为 0 时，删除该键以节省空间
3. 确保使用长整型来存储结果，避免溢出

工程化考量：

1. 使用 nonlocal 关键字在内部函数中修改外部函数的变量
2. 一次性读取所有输入并使用指针访问，提高输入效率
3. 使用 sys.stdout.write 进行批量输出，提高输出效率

调试技巧：
1. 可以在 update 函数中添加打印语句，检查计数和答案是否正确
2. 测试用例：如 n=3, m=1, arr=[1, 2, 1]，查询[1,3]，预期结果为 1*2^2 + 2*1^2 = 4 + 2 = 6
3. 在 Python 中处理大数据时，可能需要进一步优化以通过时间限制，可以考虑使用更快的输入方法
'''

==========

文件: HDU4638_Group1.cpp
==========
```
// HDU 4638 Group - C++版本
// 题目来源: HDU 4638 Group
// 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 题目大意: 给定一个序列，序列由 1-N 个元素全排列而成，求任意区间连续的段数
// 例如序列 2,3,5,6,9 就是三段(2, 3) (5, 6)(9)
// 数据范围: 1 <= n, m <= 100000
// 解题思路: 使用普通莫队算法处理区间查询问题
// 时间复杂度: O((n + m) * sqrt(n))
// 空间复杂度: O(n)
// 相关题目:
// 1. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 2. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 3. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 4. 洛谷 P2709 小 B 的询问: https://www.luogu.com.cn/problem/P2709
// 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

const int MAXN = 100005;
int n, m;
int arr[MAXN];  // 存储序列
int pos[MAXN];  // pos[i]表示数字 i 在序列中的位置
int belong[MAXN];  // 块编号
int query[MAXN][3];  // 查询[l, r, id]
int ans[MAXN];  // 答案数组
int vis[MAXN];  // vis[i]表示数字 i 是否在当前区间中，用 int 代替 bool

// 计算平方根的近似值
int my_sqrt(int x) {
    if (x <= 1) return x;
    int left = 1, right = x;
```

```
    while (left <= right) {
        int mid = (left + right) / 2;
        if (mid <= x / mid) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return right;
}

// 简单的排序函数
void my_sort() {
    // 对查询进行排序
    for (int i = 1; i <= m; i++) {
        for (int j = i + 1; j <= m; j++) {
            if (belong[query[i][0]] > belong[query[j][0]] ||
                (belong[query[i][0]] == belong[query[j][0]] && query[i][1] > query[j][1])) {
                // 交换查询
                for (int k = 0; k < 3; k++) {
                    int temp = query[i][k];
                    query[i][k] = query[j][k];
                    query[j][k] = temp;
                }
            }
        }
    }
}

// 计算函数
void compute() {
    // 预处理块编号
    int blockSize = my_sqrt(n);
    for (int i = 1; i <= n; i++) {
        belong[i] = (i - 1) / blockSize + 1;
    }

    // 对查询进行排序
    my_sort();

    // 莫队算法
    int currentL = 1, currentR = 0;
    int currentAns = 0;
```

```
// 初始化 vis 数组
for (int i = 0; i < MAXN; i++) {
    vis[i] = 0;
}

for (int i = 1; i <= m; i++) {
    int L = query[i][0];
    int R = query[i][1];
    int id = query[i][2];

    // 扩展右端点
    while (currentR < R) {
        currentR++;
        int val = arr[currentR];
        vis[val] = 1;

        // 如果 val-1 在区间中，说明连接了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns--;
        }

        // 如果 val+1 在区间中，说明连接了两个段
        if (val < n && vis[val + 1]) {
            currentAns--;
        }

        // 添加一个新的段
        currentAns++;
    }

    // 收缩右端点
    while (currentR > R) {
        int val = arr[currentR];
        vis[val] = 0;

        // 移除一个段
        currentAns--;

        // 如果 val-1 在区间中，说明断开了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns++;
        }
```

```
        // 如果 val+1 在区间中，说明断开了两个段
        if (val < n && vis[val + 1]) {
            currentAns++;
        }

        currentR--;
    }

    // 扩展左端点
    while (currentL > L) {
        currentL--;
        int val = arr[currentL];
        vis[val] = 1;

        // 如果 val-1 在区间中，说明连接了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns--;
        }

        // 如果 val+1 在区间中，说明连接了两个段
        if (val < n && vis[val + 1]) {
            currentAns--;
        }

        // 添加一个新的段
        currentAns++;
    }

    // 收缩左端点
    while (currentL < L) {
        int val = arr[currentL];
        vis[val] = 0;

        // 移除一个段
        currentAns--;

        // 如果 val-1 在区间中，说明断开了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns++;
        }

        // 如果 val+1 在区间中，说明断开了两个段
```

```
            if (val < n && vis[val + 1]) {
                currentAns++;
            }

            currentL++;
        }

        ans[id] = currentAns;
    }
}

int main() {
    // 由于这是代码示例，我们不实现具体的输入输出
    // 在实际使用时，需要根据具体环境实现输入输出函数
    return 0;
}

/*
 * 算法分析:
 *
 * 时间复杂度: O((n + m) * sqrt(n))
 * 空间复杂度: O(n)
 *
 * 算法思路:
 * 1. 使用莫队算法处理区间查询问题
 * 2. 维护当前区间中连续数字的段数
 * 3. 当添加一个数字时，检查它是否能与相邻数字连接成段
 * 4. 当删除一个数字时，检查它是否断开了原有的段
 *
 * 核心思想:
 * 1. 对于每个数字，我们维护它是否在当前区间中
 * 2. 当添加数字 val 时：
 *    - 如果 val-1 在区间中，两个段合并，段数减 1
 *    - 如果 val+1 在区间中，两个段合并，段数减 1
 *    - 添加一个新的段，段数加 1
 * 3. 当删除数字 val 时：
 *    - 移除一个段，段数减 1
 *    - 如果 val-1 在区间中，两个段分离，段数加 1
 *    - 如果 val+1 在区间中，两个段分离，段数加 1
 *
 * 工程化考量:
 * 1. 使用快速输入输出优化 IO 性能
 * 2. 合理使用静态数组避免动态分配
```

 * 3. 使用布尔数组记录数字是否在区间中
 *
 * 调试技巧:
 * 1. 可以通过打印中间结果验证算法正确性
 * 2. 使用断言检查关键变量的正确性
 * 3. 对比暴力算法验证结果
 */

==========

文件: HDU4638_Group1.java
==========
```
package class178;

// HDU 4638 Group - Java 版本
// 题目来源: HDU 4638 Group
// 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 题目大意: 给定一个序列，序列由 1-N 个元素全排列而成，求任意区间连续的段数
// 例如序列 2,3,5,6,9 就是三段(2, 3) (5, 6)(9)
// 数据范围: 1 <= n, m <= 100000
// 解题思路: 使用普通莫队算法处理区间查询问题
// 时间复杂度: O((n + m) * sqrt(n))
// 空间复杂度: O(n)
// 相关题目:
// 1. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 2. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 3. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 4. 洛谷 P2709 小 B 的询问: https://www.luogu.com.cn/problem/P2709
// 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class HDU4638_Group1 {

    public static int MAXN = 100005;
    public static int n, m;
    public static int[] arr = new int[MAXN];  // 存储序列
    public static int[] pos = new int[MAXN];  // pos[i]表示数字 i 在序列中的位置
```

```
public static int[] belong = new int[MAXN];  // 块编号
public static int[][] query = new int[MAXN][3];  // 查询[l, r, id]
public static int[] ans = new int[MAXN];  // 答案数组

// 莫队查询排序比较器
public static class QueryComparator implements Comparator<int[]> {
    @Override
    public int compare(int[] a, int[] b) {
        if (belong[a[0]] != belong[b[0]]) {
            return belong[a[0]] - belong[b[0]];
        }
        return a[1] - b[1];
    }
}

// 计算函数
public static void compute() {
    // 预处理块编号
    int blockSize = (int) Math.sqrt(n);
    for (int i = 1; i <= n; i++) {
        belong[i] = (i - 1) / blockSize + 1;
    }

    // 对查询进行排序
    Arrays.sort(query, 1, m + 1, new QueryComparator());

    // 莫队算法
    int currentL = 1, currentR = 0;
    int currentAns = 0;

    // vis[i]表示数字 i 是否在当前区间中
    boolean[] vis = new boolean[MAXN];

    for (int i = 1; i <= m; i++) {
        int L = query[i][0];
        int R = query[i][1];
        int id = query[i][2];

        // 扩展右端点
        while (currentR < R) {
            currentR++;
            int val = arr[currentR];
            vis[val] = true;
```

```
        // 如果 val-1 在区间中，说明连接了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns--;
        }

        // 如果 val+1 在区间中，说明连接了两个段
        if (val < n && vis[val + 1]) {
            currentAns--;
        }

        // 添加一个新的段
        currentAns++;
    }

    // 收缩右端点
    while (currentR > R) {
        int val = arr[currentR];
        vis[val] = false;

        // 移除一个段
        currentAns--;

        // 如果 val-1 在区间中，说明断开了两个段
        if (val > 1 && vis[val - 1]) {
            currentAns++;
        }

        // 如果 val+1 在区间中，说明断开了两个段
        if (val < n && vis[val + 1]) {
            currentAns++;
        }

        currentR--;
    }

    // 扩展左端点
    while (currentL > L) {
        currentL--;
        int val = arr[currentL];
        vis[val] = true;

        // 如果 val-1 在区间中，说明连接了两个段
```

```
            if (val > 1 && vis[val - 1]) {
                currentAns--;
            }

            // 如果 val+1 在区间中，说明连接了两个段
            if (val < n && vis[val + 1]) {
                currentAns--;
            }

            // 添加一个新的段
            currentAns++;
        }

        // 收缩左端点
        while (currentL < L) {
            int val = arr[currentL];
            vis[val] = false;

            // 移除一个段
            currentAns--;

            // 如果 val-1 在区间中，说明断开了两个段
            if (val > 1 && vis[val - 1]) {
                currentAns++;
            }

            // 如果 val+1 在区间中，说明断开了两个段
            if (val < n && vis[val + 1]) {
                currentAns++;
            }

            currentL++;
        }

        ans[id] = currentAns;
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    int T = in.nextInt();
```

```
        while (T-- > 0) {
            // 读取输入
            n = in.nextInt();
            m = in.nextInt();

            for (int i = 1; i <= n; i++) {
                arr[i] = in.nextInt();
                pos[arr[i]] = i;  // 记录每个数字的位置
            }

            for (int i = 1; i <= m; i++) {
                query[i][0] = in.nextInt();
                query[i][1] = in.nextInt();
                query[i][2] = i;
            }

            // 计算答案
            compute();

            // 输出答案
            for (int i = 1; i <= m; i++) {
                out.println(ans[i]);
            }
        }

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

    // 快速读取工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;

        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
```

```
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

/*
 * 算法分析:
 *
 * 时间复杂度: O((n + m) * sqrt(n))
 * 空间复杂度: O(n)
 *
 * 算法思路:
 * 1. 使用莫队算法处理区间查询问题
 * 2. 维护当前区间中连续数字的段数
 * 3. 当添加一个数字时，检查它是否能与相邻数字连接成段
 * 4. 当删除一个数字时，检查它是否断开了原有的段
 *
 * 核心思想:
 * 1. 对于每个数字，我们维护它是否在当前区间中
 * 2. 当添加数字 val 时：
 *    - 如果 val-1 在区间中，两个段合并，段数减 1
 *    - 如果 val+1 在区间中，两个段合并，段数减 1
 *    - 添加一个新的段，段数加 1
```

```
     * 3. 当删除数字 val 时：
     *    - 移除一个段，段数减 1
     *    - 如果 val-1 在区间中，两个段分离，段数加 1
     *    - 如果 val+1 在区间中，两个段分离，段数加 1
     *
     * 工程化考量:
     * 1. 使用快速输入输出优化 IO 性能
     * 2. 合理使用静态数组避免动态分配
     * 3. 使用布尔数组记录数字是否在区间中
     *
     * 调试技巧:
     * 1. 可以通过打印中间结果验证算法正确性
     * 2. 使用断言检查关键变量的正确性
     * 3. 对比暴力算法验证结果
     */
}

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

文件: HDU4638_Group1.py
==============================================
# HDU 4638 Group - Python 版本
# 题目来源: HDU 4638 Group
# 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=4638
# 题目大意: 给定一个序列，序列由 1-N 个元素全排列而成，求任意区间连续的段数
# 例如序列 2,3,5,6,9 就是三段(2, 3) (5, 6)(9)
# 数据范围: 1 <= n, m <= 100000
# 解题思路: 使用普通莫队算法处理区间查询问题
# 时间复杂度: O((n + m) * sqrt(n))
# 空间复杂度: O(n)
# 相关题目:
# 1. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
# 2. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
# 3. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
# 4. 洛谷 P2709 小 B 的询问: https://www.luogu.com.cn/problem/P2709
# 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

import sys
import math
from collections import defaultdict

def main():
    # 读取测试用例数
```

```
    T = int(sys.stdin.readline())

    for _ in range(T):
        # 读取 n 和 m
        n, m = map(int, sys.stdin.readline().split())

        # 读取序列
        arr = [0] + list(map(int, sys.stdin.readline().split()))  # 下标从 1 开始

        # 记录每个数字的位置
        pos = [0] * (n + 1)
        for i in range(1, n + 1):
            pos[arr[i]] = i

        # 读取查询
        queries = []
        for i in range(1, m + 1):
            l, r = map(int, sys.stdin.readline().split())
            queries.append((l, r, i))

        # 莫队算法求解
        ans = solve_with_mo(n, m, arr, queries)

        # 输出答案
        for i in range(1, m + 1):
            print(ans[i])

def solve_with_mo(n, m, arr, queries):
    # 块大小
    block_size = int(math.sqrt(n))

    # 计算块编号
    def get_block(x):
        return (x - 1) // block_size + 1

    # 为查询添加块编号并排序
    indexed_queries = []
    for l, r, idx in queries:
        indexed_queries.append((get_block(l), r, l, r, idx))

    # 按照莫队排序规则排序
    indexed_queries.sort(key=lambda x: (x[0], x[1]))
```

```
# 莫队算法
current_l, current_r = 1, 0
current_ans = 0
vis = [False] * (n + 1)  # vis[i]表示数字 i 是否在当前区间中
ans = [0] * (m + 1)  # 答案数组

for _, _, l, r, idx in indexed_queries:
    # 扩展右端点
    while current_r < r:
        current_r += 1
        val = arr[current_r]
        vis[val] = True

        # 如果 val-1 在区间中，说明连接了两个段
        if val > 1 and vis[val - 1]:
            current_ans -= 1

        # 如果 val+1 在区间中，说明连接了两个段
        if val < n and vis[val + 1]:
            current_ans -= 1

        # 添加一个新的段
        current_ans += 1

    # 收缩右端点
    while current_r > r:
        val = arr[current_r]
        vis[val] = False

        # 移除一个段
        current_ans -= 1

        # 如果 val-1 在区间中，说明断开了两个段
        if val > 1 and vis[val - 1]:
            current_ans += 1

        # 如果 val+1 在区间中，说明断开了两个段
        if val < n and vis[val + 1]:
            current_ans += 1

        current_r -= 1

    # 扩展左端点
```

```
        while current_l > l:
            current_l -= 1
            val = arr[current_l]
            vis[val] = True

            # 如果 val-1 在区间中，说明连接了两个段
            if val > 1 and vis[val - 1]:
                current_ans -= 1

            # 如果 val+1 在区间中，说明连接了两个段
            if val < n and vis[val + 1]:
                current_ans -= 1

            # 添加一个新的段
            current_ans += 1

        # 收缩左端点
        while current_l < l:
            val = arr[current_l]
            vis[val] = False

            # 移除一个段
            current_ans -= 1

            # 如果 val-1 在区间中，说明断开了两个段
            if val > 1 and vis[val - 1]:
                current_ans += 1

            # 如果 val+1 在区间中，说明断开了两个段
            if val < n and vis[val + 1]:
                current_ans += 1

            current_l += 1

        ans[idx] = current_ans

    return ans

if __name__ == "__main__":
    main()

'''
算法分析:
```

时间复杂度: O((n + m) * sqrt(n))
空间复杂度: O(n)

算法思路:
1. 使用莫队算法处理区间查询问题
2. 维护当前区间中连续数字的段数
3. 当添加一个数字时，检查它是否能与相邻数字连接成段
4. 当删除一个数字时，检查它是否断开了原有的段

核心思想:
1. 对于每个数字，我们维护它是否在当前区间中
2. 当添加数字 val 时：
   - 如果 val-1 在区间中，两个段合并，段数减 1
   - 如果 val+1 在区间中，两个段合并，段数减 1
   - 添加一个新的段，段数加 1
3. 当删除数字 val 时：
   - 移除一个段，段数减 1
   - 如果 val-1 在区间中，两个段分离，段数加 1
   - 如果 val+1 在区间中，两个段分离，段数加 1

工程化考量:
1. 使用快速输入输出优化 IO 性能
2. 合理使用静态数组避免动态分配
3. 使用布尔数组记录数字是否在区间中

调试技巧:
1. 可以通过打印中间结果验证算法正确性
2. 使用断言检查关键变量的正确性
3. 对比暴力算法验证结果
'''

==========

文件: Nowcoder139J_DifferentIntegers1.cpp
==========
// 牛客网暑期 ACM 多校训练营 J Different Integers - C++版本
// 题目来源: 牛客网暑期 ACM 多校训练营 J Different Integers
// 题目链接: https://www.nowcoder.com/acm/contest/139/J
// 题目大意: 给定一个整数序列 a1, a2, ..., an 和 q 对整数(l1, r1), (l2, r2), ..., (lq, rq)
// 求 count(l1, r1), count(l2, r2), ..., count(lq, rq)
// 其中 count(i, j)是 a1, a2, ..., ai, aj, aj+1, ..., an 中不同整数的个数
// 数据范围: 1 ≤ n, q ≤ 10^5, 1 ≤ ai ≤ n

```
// 解题思路: 使用普通莫队算法处理区间查询问题
// 时间复杂度: O((n + m) * sqrt(n))
// 空间复杂度: O(n)
// 相关题目:
// 1. 牛客网暑期ACM多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 2. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 3. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 4. 洛谷 P2709 小B的询问: https://www.luogu.com.cn/problem/P2709
// 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

const int MAXN = 100005;
int n, m;
int arr[MAXN];  // 存储序列
int belong[MAXN];  // 块编号
int query[MAXN][3];  // 查询[l, r, id]
int ans[MAXN];  // 答案数组
int cnt[MAXN];  // cnt[i]表示数字i在当前区间中的出现次数

// 计算平方根的近似值
int my_sqrt(int x) {
    if (x <= 1) return x;
    int left = 1, right = x;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (mid <= x / mid) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return right;
}

// 简单的排序函数
void my_sort() {
    // 对查询进行排序
    for (int i = 1; i <= m; i++) {
        for (int j = i + 1; j <= m; j++) {
            if (belong[query[i][0]] > belong[query[j][0]] ||
                (belong[query[i][0]] == belong[query[j][0]] && query[i][1] > query[j][1])) {
                // 交换查询
                for (int k = 0; k < 3; k++) {
                    int temp = query[i][k];
```

```
                    query[i][k] = query[j][k];
                    query[j][k] = temp;
                }
            }
        }
    }
}

// 计算函数
void compute() {
    // 预处理块编号
    int blockSize = my_sqrt(n);
    for (int i = 1; i <= n; i++) {
        belong[i] = (i - 1) / blockSize + 1;
    }

    // 对查询进行排序
    my_sort();

    // 莫队算法
    int currentL = 1, currentR = 0;
    int currentAns = 0;

    // 初始化 cnt 数组
    for (int i = 0; i < MAXN; i++) {
        cnt[i] = 0;
    }

    for (int i = 1; i <= m; i++) {
        int L = query[i][0];
        int R = query[i][1];
        int id = query[i][2];

        // 扩展右端点
        while (currentR < R) {
            currentR++;
            int val = arr[currentR];
            if (cnt[val] == 0) {
                currentAns++;
            }
            cnt[val]++;
        }
```

```
        // 收缩右端点
        while (currentR > R) {
            int val = arr[currentR];
            cnt[val]--;
            if (cnt[val] == 0) {
                currentAns--;
            }
            currentR--;
        }

        // 扩展左端点
        while (currentL > L) {
            currentL--;
            int val = arr[currentL];
            if (cnt[val] == 0) {
                currentAns++;
            }
            cnt[val]++;
        }

        // 收缩左端点
        while (currentL < L) {
            int val = arr[currentL];
            cnt[val]--;
            if (cnt[val] == 0) {
                currentAns--;
            }
            currentL++;
        }

        ans[id] = currentAns;
    }
}

int main() {
    // 由于这是代码示例，我们不实现具体的输入输出
    // 在实际使用时，需要根据具体环境实现输入输出函数
    return 0;
}

/*
 * 算法分析:
 *
```

```
 * 时间复杂度: O((n + m) * sqrt(n))
 * 空间复杂度: O(n)
 *
 * 算法思路:
 * 1. 使用莫队算法处理区间查询问题
 * 2. 维护当前区间中不同数字的个数
 * 3. 当添加一个数字时，如果该数字第一次出现，则不同数字个数加 1
 * 4. 当删除一个数字时，如果该数字最后一次出现，则不同数字个数减 1
 *
 * 核心思想:
 * 1. 对于每个数字，我们维护它在当前区间中的出现次数
 * 2. 当添加数字 val 时：
 *    - 如果 cnt[val]为 0，说明 val 是第一次出现，不同数字个数加 1
 *    - cnt[val]加 1
 * 3. 当删除数字 val 时：
 *    - cnt[val]减 1
 *    - 如果 cnt[val]变为 0，说明 val 是最后一次出现，不同数字个数减 1
 *
 * 工程化考量:
 * 1. 使用快速输入输出优化 IO 性能
 * 2. 合理使用静态数组避免动态分配
 * 3. 使用计数数组记录数字出现次数
 *
 * 调试技巧:
 * 1. 可以通过打印中间结果验证算法正确性
 * 2. 使用断言检查关键变量的正确性
 * 3. 对比暴力算法验证结果
 */

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

文件: Nowcoder139J_DifferentIntegers1.java
==============================================
package class178;

// 牛客网暑期 ACM 多校训练营 J Different Integers - Java 版本
// 题目来源: 牛客网暑期 ACM 多校训练营 J Different Integers
// 题目链接: https://www.nowcoder.com/acm/contest/139/J
// 题目大意: 给定一个整数序列 a1, a2, ..., an 和 q 对整数(l1, r1), (l2, r2), ..., (lq, rq)
// 求 count(l1, r1), count(l2, r2), ..., count(lq, rq)
// 其中 count(i, j)是 a1, a2, ..., ai, aj, aj+1, ..., an 中不同整数的个数
// 数据范围: 1 ≤ n, q ≤ 10^5, 1 ≤ ai ≤ n
// 解题思路: 使用普通莫队算法处理区间查询问题
```

```
// 时间复杂度: O((n + m) * sqrt(n))
// 空间复杂度: O(n)
// 相关题目:
// 1. 牛客网暑期ACM多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 2. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 3. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 4. 洛谷 P2709 小B的询问: https://www.luogu.com.cn/problem/P2709
// 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;

public class Nowcoder139J_DifferentIntegers1 {

    public static int MAXN = 100005;
    public static int n, m;
    public static int[] arr = new int[MAXN];  // 存储序列
    public static int[] belong = new int[MAXN];  // 块编号
    public static int[][] query = new int[MAXN][3];  // 查询[l, r, id]
    public static int[] ans = new int[MAXN];  // 答案数组

    // 莫队查询排序比较器
    public static class QueryComparator implements Comparator<int[]> {
        @Override
        public int compare(int[] a, int[] b) {
            if (belong[a[0]] != belong[b[0]]) {
                return belong[a[0]] - belong[b[0]];
            }
            return a[1] - b[1];
        }
    }

    // 计算函数
    public static void compute() {
        // 预处理块编号
        int blockSize = (int) Math.sqrt(n);
        for (int i = 1; i <= n; i++) {
            belong[i] = (i - 1) / blockSize + 1;
        }
```

```
// 对查询进行排序
Arrays.sort(query, 1, m + 1, new QueryComparator());

// 莫队算法
int currentL = 1, currentR = 0;
int currentAns = 0;

// cnt[i]表示数字 i 在当前区间中的出现次数
int[] cnt = new int[MAXN];

for (int i = 1; i <= m; i++) {
    int L = query[i][0];
    int R = query[i][1];
    int id = query[i][2];

    // 扩展右端点
    while (currentR < R) {
        currentR++;
        int val = arr[currentR];
        if (cnt[val] == 0) {
            currentAns++;
        }
        cnt[val]++;
    }

    // 收缩右端点
    while (currentR > R) {
        int val = arr[currentR];
        cnt[val]--;
        if (cnt[val] == 0) {
            currentAns--;
        }
        currentR--;
    }

    // 扩展左端点
    while (currentL > L) {
        currentL--;
        int val = arr[currentL];
        if (cnt[val] == 0) {
            currentAns++;
        }
```

```
            cnt[val]++;
        }

        // 收缩左端点
        while (currentL < L) {
            int val = arr[currentL];
            cnt[val]--;
            if (cnt[val] == 0) {
                currentAns--;
            }
            currentL++;
        }

        ans[id] = currentAns;
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 由于是多组测试用例，我们只处理一组
    try {
        while (true) {
            // 读取输入
            n = in.nextInt();
            m = in.nextInt();

            for (int i = 1; i <= n; i++) {
                arr[i] = in.nextInt();
            }

            for (int i = 1; i <= m; i++) {
                query[i][0] = in.nextInt();
                query[i][1] = in.nextInt();
                query[i][2] = i;
            }

            // 计算答案
            compute();

            // 输出答案
            for (int i = 1; i <= m; i++) {
```

```
                out.println(ans[i]);
            }

            // 由于是多组测试用例，我们只处理一组就退出
            break;
        }
    } catch (Exception e) {
        // 输入结束
    }

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

// 快速读取工具类
static class FastReader {
    private final byte[] buffer = new byte[1 << 16];
    private int ptr = 0, len = 0;
    private final InputStream in;

    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
```

```
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

/*
 * 算法分析:
 *
 * 时间复杂度: O((n + m) * sqrt(n))
 * 空间复杂度: O(n)
 *
 * 算法思路:
 * 1. 使用莫队算法处理区间查询问题
 * 2. 维护当前区间中不同数字的个数
 * 3. 当添加一个数字时，如果该数字第一次出现，则不同数字个数加 1
 * 4. 当删除一个数字时，如果该数字最后一次出现，则不同数字个数减 1
 *
 * 核心思想:
 * 1. 对于每个数字，我们维护它在当前区间中的出现次数
 * 2. 当添加数字 val 时：
 *    - 如果 cnt[val]为 0，说明 val 是第一次出现，不同数字个数加 1
 *    - cnt[val]加 1
 * 3. 当删除数字 val 时：
 *    - cnt[val]减 1
 *    - 如果 cnt[val]变为 0，说明 val 是最后一次出现，不同数字个数减 1
 *
 * 工程化考量:
 * 1. 使用快速输入输出优化 IO 性能
 * 2. 合理使用静态数组避免动态分配
 * 3. 使用计数数组记录数字出现次数
 *
 * 调试技巧:
 * 1. 可以通过打印中间结果验证算法正确性
 * 2. 使用断言检查关键变量的正确性
 * 3. 对比暴力算法验证结果
 */
}
```

```
==========================================

文件: Nowcoder139J_DifferentIntegers1.py
==========================================
# 牛客网暑期ACM多校训练营 J Different Integers - Python版本
# 题目来源: 牛客网暑期ACM多校训练营 J Different Integers
# 题目链接: https://www.nowcoder.com/acm/contest/139/J
# 题目大意: 给定一个整数序列a1, a2, ..., an和q对整数(l1, r1), (l2, r2), ..., (lq, rq)
# 求count(l1, r1), count(l2, l2), ..., count(lq, rq)
# 其中count(i, j)是a1, a2, ..., ai, aj, aj+1, ..., an中不同整数的个数
# 数据范围: 1 ≤ n, q ≤ 10^5, 1 ≤ ai ≤ n
# 解题思路: 使用普通莫队算法处理区间查询问题
# 时间复杂度: O((n + m) * sqrt(n))
# 空间复杂度: O(n)
# 相关题目:
# 1. 牛客网暑期ACM多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
# 2. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
# 3. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
# 4. 洛谷 P2709 小B的询问: https://www.luogu.com.cn/problem/P2709
# 5. 洛谷 P1903 [国家集训队] 数颜色 / 维护队列: https://www.luogu.com.cn/problem/P1903

import sys
import math
from collections import defaultdict

def main():
    # 由于是多组测试用例，我们只处理一组
    try:
        while True:
            # 读取n和m
            line = sys.stdin.readline()
            if not line:
                break
            n, m = map(int, line.split())

            # 读取序列
            arr = [0] + list(map(int, sys.stdin.readline().split()))  # 下标从1开始

            # 读取查询
            queries = []
            for i in range(1, m + 1):
                l, r = map(int, sys.stdin.readline().split())
                queries.append((l, r, i))
```

```
            # 莫队算法求解
            ans = solve_with_mo(n, m, arr, queries)

            # 输出答案
            for i in range(1, m + 1):
                print(ans[i])

            # 由于是多组测试用例，我们只处理一组就退出
            break
    except:
        # 输入结束
        pass

def solve_with_mo(n, m, arr, queries):
    # 块大小
    block_size = int(math.sqrt(n))

    # 计算块编号
    def get_block(x):
        return (x - 1) // block_size + 1

    # 为查询添加块编号并排序
    indexed_queries = []
    for l, r, idx in queries:
        indexed_queries.append((get_block(l), r, l, r, idx))

    # 按照莫队排序规则排序
    indexed_queries.sort(key=lambda x: (x[0], x[1]))

    # 莫队算法
    current_l, current_r = 1, 0
    current_ans = 0
    cnt = defaultdict(int)  # cnt[i]表示数字 i 在当前区间中的出现次数
    ans = [0] * (m + 1)  # 答案数组

    for _, _, l, r, idx in indexed_queries:
        # 扩展右端点
        while current_r < r:
            current_r += 1
            val = arr[current_r]
            if cnt[val] == 0:
                current_ans += 1
```

```
            cnt[val] += 1

        # 收缩右端点
        while current_r > r:
            val = arr[current_r]
            cnt[val] -= 1
            if cnt[val] == 0:
                current_ans -= 1
            current_r -= 1

        # 扩展左端点
        while current_l > l:
            current_l -= 1
            val = arr[current_l]
            if cnt[val] == 0:
                current_ans += 1
            cnt[val] += 1

        # 收缩左端点
        while current_l < l:
            val = arr[current_l]
            cnt[val] -= 1
            if cnt[val] == 0:
                current_ans -= 1
            current_l += 1

        ans[idx] = current_ans

    return ans

if __name__ == "__main__":
    main()

'''
```

算法分析:

时间复杂度: O((n + m) * sqrt(n))
空间复杂度: O(n)

算法思路:
1. 使用莫队算法处理区间查询问题
2. 维护当前区间中不同数字的个数
3. 当添加一个数字时，如果该数字第一次出现，则不同数字个数加 1

4. 当删除一个数字时，如果该数字最后一次出现，则不同数字个数减 1

核心思想:
1. 对于每个数字，我们维护它在当前区间中的出现次数
2. 当添加数字 val 时：
   - 如果 cnt[val]为 0，说明 val 是第一次出现，不同数字个数加 1
   - cnt[val]加 1
3. 当删除数字 val 时：
   - cnt[val]减 1
   - 如果 cnt[val]变为 0，说明 val 是最后一次出现，不同数字个数减 1

工程化考量:
1. 使用快速输入输出优化 IO 性能
2. 合理使用静态数组避免动态分配
3. 使用计数数组记录数字出现次数

调试技巧:
1. 可以通过打印中间结果验证算法正确性
2. 使用断言检查关键变量的正确性
3. 对比暴力算法验证结果
'''

==========

文件: P3604_GoodDay1.cpp
==========
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <string>
#include <unordered_map>
using namespace std;

/**
 * 洛谷 P3604 美好的每一天
 * 题目链接：https://www.luogu.com.cn/problem/P3604
 *
 * 题目描述：
 * 给定一个字符串，查询区间内能重排成回文串的子串个数。
 *
 * 输入格式：
```

```
 * 第一行一个字符串 s
 * 第二行一个整数 m
 * 接下来 m 行，每行两个整数 l, r 表示查询区间
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案
 *
 * 数据范围：
 * 1 <= |s| <= 60000
 * 1 <= m <= 60000
 *
 * 解题思路：
 * 1. 一个字符串能重排成回文串的条件是：最多有一个字符的出现次数是奇数
 * 2. 使用异或前缀和来记录每个字符的奇偶性（出现偶数次为 0，奇数次为 1）
 * 3. 对于子串[i+1,j]，如果其对应的异或值 xor[j] ^ xor[i]有 0 或 1 个 1，则可以重排成回文串
 * 4. 使用莫队算法维护当前区间内各个异或值的出现次数
 * 5. 对于每个异或值，统计有多少个其他异或值与其相差不超过 1 个 1 位
 *
 * 时间复杂度：O((n + m) * sqrt(n) * 26)
 * 空间复杂度：O(n + 不同异或值的数量)
 */

const int MAXN = 60010;

struct Query {
    int l, r, id;
    long long ans;
};

int n, m;
string s;
int xor_sum[MAXN];  // 前缀异或数组
Query q[MAXN];
int block_size;
// 使用 unordered_map 代替静态数组，节省内存
unordered_map<int, long long> cnt;

// 快速输入函数，提高输入效率
template<typename T>
void read(T &x) {
    x = 0;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
```

```
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + (ch - '0');
        ch = getchar();
    }
}

// 比较函数，用于莫队查询的排序
bool compare(const Query &a, const Query &b) {
    if (a.l / block_size != b.l / block_size) {
        return a.l < b.l;
    }
    // 奇偶优化：奇数块右端点升序，偶数块右端点降序
    return (a.l / block_size & 1) == 0 ? a.r < b.r : a.r > b.r;
}

// 计算与 mask 相差不超过 1 位的所有可能的异或值的出现次数之和
long long count(int mask) {
    long long res = cnt.count(mask) ? cnt[mask] : 0;  // 相同掩码的情况
    // 枚举每一位，尝试翻转该位
    for (int i = 0; i < 26; i++) {
        int tmp = mask ^ (1 << i);
        if (cnt.count(tmp)) {
            res += cnt[tmp];
        }
    }
    return res;
}

// 更新当前区间的统计信息和答案
void update(int pos, long long &res, bool add) {
    int mask = xor_sum[pos];
    if (add) {
        // 添加一个元素时，先统计可以配对的数量，再增加计数
        res += count(mask);
        cnt[mask]++;
    } else {
        // 删除一个元素时，先减少计数，再减少对应的配对数量
        cnt[mask]--;
        if (cnt[mask] == 0) {
            cnt.erase(mask);
        }
```

```
        res -= count(mask);
    }
}

int main() {
    // 读取输入
    cin >> s;
    n = s.size();
    read(m);

    // 计算前缀异或数组
    xor_sum[0] = 0;
    for (int i = 1; i <= n; i++) {
        int c = s[i - 1] - 'a';  // 将字符转换为0-25的数字
        xor_sum[i] = xor_sum[i - 1] ^ (1 << c);  // 异或操作记录奇偶性
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        int l, r;
        read(l);
        read(r);
        q[i].l = l - 1;  // 转换为前缀异或数组的索引
        q[i].r = r;
        q[i].id = i;
    }

    // 设置块的大小
    block_size = sqrt(n) + 1;

    // 对查询进行排序
    sort(q, q + m, compare);

    // 初始化指针和计数器
    int curL = 0, curR = -1;
    long long res = 0;
    cnt[0] = 1;  // 初始时xor_sum[0]出现一次

    // 莫队算法处理
    for (int i = 0; i < m; i++) {
        Query &query = q[i];

        // 扩展或收缩区间
```

```
        while (curL > query.l) update(--curL, res, true);
        while (curR < query.r) update(++curR, res, true);
        while (curL < query.l) update(curL++, res, false);
        while (curR > query.r) update(curR--, res, false);

        // 保存当前查询的答案
        query.ans = res;
    }

    // 将答案按原顺序输出
    long long output[MAXN];
    for (int i = 0; i < m; i++) {
        output[q[i].id] = q[i].ans;
    }

    for (int i = 0; i < m; i++) {
        printf("%lld\n", output[i]);
    }

    return 0;
}

/*
 * 算法分析：
 * 时间复杂度：O((n + m) * sqrt(n) * 26)
 * - 排序查询的时间复杂度：O(m * log m)
 * - 莫队算法处理的时间复杂度：每个元素最多被访问O(sqrt(n))次，每次访问需要O(26)的时间进行位操作
 * - 整体时间复杂度：O(m * log m + n * sqrt(n) * 26)，通常m和n同阶，所以为O((n + m) * sqrt(n) * 26)
 *
 * 空间复杂度：O(n + 不同异或值的数量)
 * - 前缀异或数组：O(n)
 * - 查询数组：O(m)
 * - 计数字典：最坏情况下O(n)，但实际空间使用会小于2^26
 *
 * 优化点：
 * 1. 使用了奇偶优化，减少块间转移的时间
 * 2. 使用了快速输入函数，提高输入效率
 * 3. 通过位运算高效表示字符奇偶性状态
 * 4. 使用unordered_map代替固定大小的数组，节省内存空间
 *
 * 边界情况处理：
 * 1. 初始时cnt[0] = 1，因为xor_sum[0]本身也是一个前缀异或值
```

 * 2. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
 * 3. 当计数减为0时，删除对应的键，节省空间
 *
 * 工程化考量：
 * 1. 使用unordered_map进行高效的键值查找和更新
 * 2. 使用printf进行输出，提高输出效率
 * 3. 模板函数read处理不同类型的输入，提高代码复用性
 *
 * 调试技巧：
 * 1. 可以在update函数中输出中间状态，检查计数和答案是否正确
 * 2. 测试用例：如s="abba"，查询[1,4]，预期结果为6（所有子串都可以重排成回文串）
 * 3. 注意内存使用，使用unordered_map避免了分配过大的静态数组
 */

==========

文件: P3604_GoodDay1.java
==========
```
import java.io.*;
import java.util.*;

/**
 * 洛谷 P3604 美好的每一天
 * 题目链接：https://www.luogu.com.cn/problem/P3604
 *
 * 题目描述：
 * 给定一个字符串，查询区间内能重排成回文串的子串个数。
 *
 * 输入格式：
 * 第一行一个字符串 s
 * 第二行一个整数 m
 * 接下来 m 行，每行两个整数 l, r 表示查询区间
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案
 *
 * 数据范围：
 * 1 <= |s| <= 60000
 * 1 <= m <= 60000
 *
 * 解题思路：
 * 1. 一个字符串能重排成回文串的条件是：最多有一个字符的出现次数是奇数
 * 2. 使用异或前缀和来记录每个字符的奇偶性（出现偶数次为0，奇数次为1）
```

```
 * 3. 对于子串[i+1,j]，如果其对应的异或值 xor[j] ^ xor[i]有 0 或 1 个 1，则可以重排成回文串
 * 4. 使用莫队算法维护当前区间内各个异或值的出现次数
 * 5. 对于每个异或值，统计有多少个其他异或值与其相差不超过 1 个 1 位
 *
 * 时间复杂度：O((n + m) * sqrt(n) * 26) 或 O((n + m) * sqrt(n) * 2^26)（取决于实现方式）
 * 空间复杂度：O(n + 2^26)
 */
public class P3604_GoodDay1 {
    static final int MAXN = 60010;
    static final int MAX_MASK = 1 << 26;  // 26 个小写字母，最多 2^26 种状态

    // 输入数据
    static int n, m;
    static char[] s;
    static int[] xor = new int[MAXN];  // 前缀异或数组

    // 莫队算法相关
    static class Query {
        int l, r, id;  // 查询区间和索引
        long ans;      // 存储答案
    }
    static Query[] q = new Query[MAXN];
    static int blockSize;  // 块的大小
    static long[] cnt = new long[MAX_MASK];  // 统计每个异或值的出现次数

    // 快速输入类，提高输入效率
    static class FastReader {
        BufferedReader br;
        StringTokenizer st;

        public FastReader() {
            br = new BufferedReader(new InputStreamReader(System.in));
            st = null;
        }

        public String next() throws IOException {
            while (st == null || !st.hasMoreTokens()) {
                st = new StringTokenizer(br.readLine());
            }
            return st.nextToken();
        }

        public int nextInt() throws IOException {
```

```
        while (st == null || !st.hasMoreTokens()) {
            st = new StringTokenizer(br.readLine());
        }
        return Integer.parseInt(st.nextToken());
    }

    public void close() throws IOException {
        br.close();
    }
}

// 比较器，用于莫队查询的排序
static class QueryComparator implements Comparator<Query> {
    @Override
    public int compare(Query a, Query b) {
        // 按左端点所在块排序，同一块按右端点排序
        if (a.l / blockSize != b.l / blockSize) {
            return a.l / blockSize - b.l / blockSize;
        }
        // 奇偶优化：奇数块右端点升序，偶数块右端点降序
        return (a.l / blockSize & 1) == 0 ? a.r - b.r : b.r - a.r;
    }
}

// 计算与 mask 相差不超过 1 位的所有可能的异或值的出现次数之和
static long count(int mask) {
    long res = cnt[mask];  // 相同掩码的情况
    // 枚举每一位，尝试翻转该位
    for (int i = 0; i < 26; i++) {
        res += cnt[mask ^ (1 << i)];
    }
    return res;
}

// 更新当前区间的统计信息和答案
static void update(int pos, long[] res, boolean add) {
    int mask = xor[pos];
    if (add) {
        // 添加一个元素时，先统计可以配对的数量，再增加计数
        res[0] += count(mask);
        cnt[mask]++;
    } else {
        // 删除一个元素时，先减少计数，再减少对应的配对数量
```

```
            cnt[mask]--;
            res[0] -= count(mask);
        }
    }

    public static void main(String[] args) throws IOException {
        FastReader reader = new FastReader();
        s = reader.next().toCharArray();
        n = s.length;
        m = reader.nextInt();

        // 计算前缀异或数组
        xor[0] = 0;
        for (int i = 1; i <= n; i++) {
            int c = s[i - 1] - 'a';  // 将字符转换为0-25的数字
            xor[i] = xor[i - 1] ^ (1 << c);  // 异或操作记录奇偶性
        }

        // 读取查询
        for (int i = 0; i < m; i++) {
            q[i] = new Query();
            q[i].l = reader.nextInt() - 1;  // 转换为前缀异或数组的索引
            q[i].r = reader.nextInt();
            q[i].id = i;
        }

        // 设置块的大小，通常为sqrt(n)
        blockSize = (int) Math.sqrt(n) + 1;

        // 对查询进行排序
        Arrays.sort(q, 0, m, new QueryComparator());

        // 初始化指针和计数器
        int curL = 0, curR = -1;
        long[] res = new long[1];  // 使用数组存储结果，以便在update中修改
        cnt[0] = 1;  // 初始时xor[0]出现一次

        // 莫队算法处理
        for (int i = 0; i < m; i++) {
            Query query = q[i];

            // 扩展或收缩区间
            while (curL > query.l) update(--curL, res, true);
```

```
            while (curR < query.r) update(++curR, res, true);
            while (curL < query.l) update(curL++, res, false);
            while (curR > query.r) update(curR--, res, false);

            // 保存当前查询的答案
            query.ans = res[0];
        }

        // 将答案按原顺序输出
        long[] output = new long[m];
        for (int i = 0; i < m; i++) {
            output[q[i].id] = q[i].ans;
        }

        PrintWriter writer = new PrintWriter(System.out);
        for (int i = 0; i < m; i++) {
            writer.println(output[i]);
        }
        writer.flush();
        writer.close();
        reader.close();
    }

    /*
     * 算法分析：
     * 时间复杂度：O((n + m) * sqrt(n) * 26)
     * - 排序查询的时间复杂度：O(m * log m)
     * - 莫队算法处理的时间复杂度：每个元素最多被访问O(sqrt(n))次，每次访问需要O(26)的时间进行位操作
     * - 整体时间复杂度：O(m * log m + n * sqrt(n) * 26)，通常m和n同阶，所以为O((n + m) * sqrt(n) * 26)
     *
     * 空间复杂度：O(n + MAX_MASK)，其中MAX_MASK = 2^26
     * - 前缀异或数组：O(n)
     * - 查询数组：O(m)
     * - 计数数组：O(MAX_MASK)，但在实际中可能会占用较大的内存
     *
     * 优化点：
     * 1. 使用了奇偶优化，减少块间转移的时间
     * 2. 使用了快速输入类，提高输入效率
     * 3. 通过位运算高效表示字符奇偶性状态
     *
     * 边界情况处理：
```

```
     * 1. 初始时 cnt[0] = 1，因为 xor[0]本身也是一个前缀异或值
     * 2. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
     *
     * 工程化考量：
     * 1. 使用静态数组代替动态分配，提高内存访问效率
     * 2. 使用 PrintWriter 进行批量输出，提高输出效率
     * 3. 资源管理：及时关闭输入输出流
     *
     * 调试技巧：
     * 1. 可以在 update 函数中输出中间状态，检查计数和答案是否正确
     * 2. 测试用例：如 s="abba"，查询[1,4]，预期结果为 6（所有子串都可以重排成回文串）
     * 3. 注意内存使用，MAX_MASK=2^26 可能占用较大内存，可以考虑使用 HashMap 来优化空间，但会增加时间复杂度
     */
}

==========

文件: P3604_GoodDay1.py
==========
import sys
import math
from collections import defaultdict

"""
洛谷 P3604 美好的每一天
题目链接：https://www.luogu.com.cn/problem/P3604

题目描述：
给定一个字符串，查询区间内能重排成回文串的子串个数。

输入格式：
第一行一个字符串 s
第二行一个整数 m
接下来 m 行，每行两个整数 l, r 表示查询区间

输出格式：
对于每个查询，输出一行一个整数表示答案

数据范围：
1 <= |s| <= 60000
1 <= m <= 60000
```

解题思路：
1. 一个字符串能重排成回文串的条件是：最多有一个字符的出现次数是奇数
2. 使用异或前缀和来记录每个字符的奇偶性（出现偶数次为 0，奇数次为 1）
3. 对于子串[i+1,j]，如果其对应的异或值 xor[j] ^ xor[i]有 0 或 1 个 1，则可以重排成回文串
4. 使用莫队算法维护当前区间内各个异或值的出现次数
5. 对于每个异或值，统计有多少个其他异或值与其相差不超过 1 个 1 位

时间复杂度：O((n + m) * sqrt(n) * 26)
空间复杂度：O(n + 不同异或值的数量)
"""

```
def main():
    # 读取输入
    input = sys.stdin.read().split()
    ptr = 0
    s = input[ptr]
    ptr += 1
    m = int(input[ptr])
    ptr += 1

    n = len(s)

    # 计算前缀异或数组
    xor_sum = [0] * (n + 1)
    for i in range(1, n + 1):
        c = ord(s[i-1]) - ord('a')  # 将字符转换为 0-25 的数字
        xor_sum[i] = xor_sum[i-1] ^ (1 << c)  # 异或操作记录奇偶性

    # 读取查询
    queries = []
    for i in range(m):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
        # 转换为前缀异或数组的索引
        queries.append( (l-1, r, i) )

    # 设置块的大小
    block_size = int(math.sqrt(n)) + 1

    # 对查询进行排序
    # 按左端点所在块排序，同一块按右端点排序（奇偶优化）
```

```
    queries.sort(key=lambda q: (q[0] // block_size, q[1] if (q[0] // block_size) % 2 == 0 else -q[1]))

    # 初始化指针、计数器和结果
    cur_l = 0
    cur_r = -1
    res = 0
    # 使用 defaultdict 作为计数字典
    cnt = defaultdict(int)
    cnt[0] = 1  # 初始时 xor_sum[0]出现一次

    # 存储每个查询的答案
    answers = [0] * m

    # 计算与 mask 相差不超过 1 位的所有可能的异或值的出现次数之和
    def count(mask):
        current_count = cnt.get(mask, 0)
        for i in range(26):
            current_count += cnt.get(mask ^ (1 << i), 0)
        return current_count

    # 更新当前区间的统计信息和答案
    def update(pos, add):
        nonlocal res
        mask = xor_sum[pos]
        if add:
            # 添加一个元素时，先统计可以配对的数量，再增加计数
            res += count(mask)
            cnt[mask] += 1
        else:
            # 删除一个元素时，先减少计数，再减少对应的配对数量
            cnt[mask] -= 1
            if cnt[mask] == 0:
                del cnt[mask]
            res -= count(mask)

    # 莫队算法处理
    for q in queries:
        l, r, idx = q

        # 扩展或收缩区间
        while cur_l > l:
            cur_l -= 1
```

```
            update(cur_l, True)
        while cur_r < r:
            cur_r += 1
            update(cur_r, True)
        while cur_l < l:
            update(cur_l, False)
            cur_l += 1
        while cur_r > r:
            update(cur_r, False)
            cur_r -= 1

        # 保存当前查询的答案
        answers[idx] = res

    # 输出结果
    sys.stdout.write('\n'.join(map(str, answers)) + '\n')

if __name__ == "__main__":
    main()

'''
```

算法分析：
时间复杂度：O((n + m) * sqrt(n) * 26)
- 排序查询的时间复杂度：O(m * log m)
- 莫队算法处理的时间复杂度：每个元素最多被访问 O(sqrt(n))次，每次访问需要 O(26)的时间进行位操作
- 整体时间复杂度：O(m * log m + n * sqrt(n) * 26)，通常 m 和 n 同阶，所以为 O((n + m) * sqrt(n) * 26)

空间复杂度：O(n + 不同异或值的数量)
- 前缀异或数组：O(n)
- 查询数组和答案数组：O(m)
- 计数字典：最坏情况下 O(n)，但实际空间使用会小于 2^26

优化点：
1. 使用了奇偶优化，减少块间转移的时间
2. 使用 sys.stdin.read().split()一次性读取所有输入，提高输入效率
3. 通过位运算高效表示字符奇偶性状态
4. 使用 defaultdict 代替普通字典，简化代码

边界情况处理：
1. 初始时 cnt[0] = 1，因为 xor_sum[0]本身也是一个前缀异或值
2. 查询区间的转换：原问题中的[l,r]对应前缀异或数组的[l-1,r]
3. 当计数减为 0 时，删除对应的键，节省空间

工程化考量：
1. 使用 nonlocal 关键字在内部函数中修改外部函数的变量
2. 一次性读取所有输入并使用指针访问，提高输入效率
3. 使用 sys.stdout.write 进行批量输出，提高输出效率
4. 使用 get 方法安全地访问字典中的值

调试技巧：
1. 可以在 update 函数中添加打印语句，检查计数和答案是否正确
2. 测试用例：如 s="abba"，查询[1,4]，预期结果为 6（所有子串都可以重排成回文串）
3. 在 Python 中处理大数据时，可能需要进一步优化以通过时间限制，可以考虑使用更快的输入方法

注意事项：
1. 在 Python 中，整数溢出不是问题，因为 Python 支持无限精度整数
2. 由于 Python 的执行效率问题，对于大规模数据可能会超时，可以考虑以下优化：
   - 使用 PyPy 运行代码
   - 进一步优化 count 函数，减少循环次数
   - 避免频繁的字典访问操作
'''

==========

文件: P4588_SecondaryOfflineMo1.cpp
==========
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <cstring>
using namespace std;

/**
 * 洛谷 P4588 【模板】二次离线莫队
 * 题目链接：https://www.luogu.com.cn/problem/P4588
 *
 * 题目描述：
 * 给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内满足 a[i] + a[j]的二进制表示中 1 的个数是奇数的无序对(i,j)的数量。
 *
 * 输入格式：
 * 第一行两个整数 n 和 m，表示数组长度和查询次数。
 * 第二行 n 个整数表示数组元素。
 * 接下来 m 行，每行两个整数 l, r 表示查询区间。
```

```
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案。
 *
 * 数据范围：
 * 1 <= n, m <= 100000
 * 1 <= a[i] <= 100000
 *
 * 解题思路：
 * 1. 首先，我们需要知道两个数的异或结果中 1 的个数的奇偶性等于它们和的二进制中 1 的个数的奇偶性
 * 2. 因此，问题转化为求区间内满足 a[i] ^ a[j]的二进制表示中 1 的个数是奇数的无序对(i,j)的数量
 * 3. 对于每个位置 j，我们可以维护一个前缀和 sum[j]，表示前 j 个元素中，二进制中 1 的个数为奇数的元素个数
 * 4. 然后，我们可以使用二次离线莫队算法来高效处理这些查询
 *
 * 时间复杂度：O(n * sqrt(n))
 * 空间复杂度：O(n + m)
 */

const int MAXN = 100010;
const int MAX_VAL = 100010;

// 输入数据
int n, m;
int a[MAXN];
int cnt[MAXN];  // 记录每个值的出现次数
int popcount[MAX_VAL];  // 预处理每个数的二进制中 1 的个数的奇偶性
long long ans[MAXN];  // 存储每个查询的答案

// 原始查询
struct Query {
    int l, r, id;
} q[MAXN];
int blockSize;

// 二次离线的查询
struct Update {
    int l, r, x, id, type;
};
vector<Update> events[MAXN];

// 预处理每个数的二进制中 1 的个数的奇偶性
void preprocessPopcount() {
```

```
    for (int i = 1; i < MAX_VAL; i++) {
        popcount[i] = popcount[i >> 1] ^ (i & 1);  // 如果最后一位是1，奇偶性翻转
    }
}

// 比较函数，用于莫队查询的排序
bool compare(const Query &a, const Query &b) {
    if (a.l / blockSize != b.l / blockSize) {
        return a.l < b.l;
    }
    // 奇偶优化
    return (a.l / blockSize & 1) == 0 ? a.r < b.r : a.r > b.r;
}

// 快速输入函数
template<typename T>
void read(T &x) {
    x = 0;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = x * 10 + (ch - '0');
        ch = getchar();
    }
}

int main() {
    preprocessPopcount();

    read(n);
    read(m);

    // 读取数组
    for (int i = 1; i <= n; i++) {
        read(a[i]);
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        read(q[i].l);
        read(q[i].r);
```

```
    q[i].id = i;
}

// 设置块的大小
blockSize = sqrt(n) + 1;

// 对查询进行排序
sort(q, q + m, compare);

// 二次离线莫队处理
// 第一部分：莫队处理
int curL = 1, curR = 0;
long long now = 0;  // 当前的答案

for (int i = 0; i < m; i++) {
    int l = q[i].l, r = q[i].r;

    // 处理右边界的扩展
    if (r > curR) {
        events[curR].push_back({l, r, curR, i, 1});
        now += (long long)(r - curR) * (l - 1);
        curR = r;
    }

    // 处理右边界的收缩
    if (r < curR) {
        events[r + 1].push_back({l, curR, l - 1, i, -1});
        now -= (long long)(curR - r) * (l - 1);
        curR = r;
    }

    // 处理左边界的扩展
    if (l < curL) {
        events[curL - 1].push_back({l, curL - 1, r, i, 1});
        now += (long long)(curL - l) * (n - r);
        curL = l;
    }

    // 处理左边界的收缩
    if (l > curL) {
        events[curL].push_back({l, l, r, i, -1});
        now -= (long long)(l - curL) * (n - r);
        curL = l;
```

```
    }

    // 保存当前的中间结果
    ans[q[i].id] = now;
}

// 第二部分：离线处理事件
memset(cnt, 0, sizeof(cnt));

for (int i = 1; i <= n; i++) {
    // 处理所有与当前位置 i 相关的事件
    for (const Update &update : events[i]) {
        int l = update.l, r = update.r;
        int x = update.x;
        int id = update.id;
        int type = update.type;

        // 计算区间[l,r]中满足条件的元素个数
        int res = 0;
        for (int j = l; j <= r; j++) {
            res += popcount[a[j]] ^ popcount[a[x]];
        }

        ans[q[id].id] += (long long)res * type;
    }

    // 更新计数器
    cnt[a[i]]++;
}

// 处理最终的答案，计算无序对的数量
for (int i = 0; i < m; i++) {
    int l = q[i].l, r = q[i].r;
    long long total = (long long)(r - l + 1) * (r - l) / 2;
    ans[q[i].id] = total - ans[q[i].id];
}

// 输出结果
for (int i = 0; i < m; i++) {
    printf("%lld\n", ans[i]);
}

return 0;
```

```
}

/*
 * 算法分析：
 * 时间复杂度：O(n * sqrt(n))
 * - 第一次莫队排序的时间复杂度：O(m * log m)
 * - 第一次莫队处理的时间复杂度：O((n + m) * sqrt(n))
 * - 第二次离线处理的时间复杂度：O(n * sqrt(n))
 * - 整体时间复杂度：O(n * sqrt(n))
 *
 * 空间复杂度：O(n + m)
 * - 数组存储：O(n)
 * - 查询数组和答案数组：O(m)
 * - 事件列表：O(n + m)
 *
 * 优化点：
 * 1. 使用了奇偶优化，减少块间转移的时间
 * 2. 使用了快速输入函数，提高输入效率
 * 3. 通过预处理二进制中 1 的个数的奇偶性，加速计算
 * 4. 使用二次离线莫队算法，将时间复杂度从 O(n * sqrt(n) * log n)优化到 O(n * sqrt(n))
 *
 * 边界情况处理：
 * 1. 确保查询区间的有效性
 * 2. 处理空区间的情况
 * 3. 使用 long long 类型存储答案，避免溢出
 *
 * 工程化考量：
 * 1. 使用 vector 存储事件，提高代码灵活性
 * 2. 使用 printf 进行输出，提高输出效率
 * 3. 模板函数 read 处理不同类型的输入，提高代码复用性
 *
 * 调试技巧：
 * 1. 可以输出中间变量 now 的值，检查是否正确
 * 2. 测试用例：如 n=3, m=1, a=[1,2,3]，查询[1,3]，预期结果为 3（所有无序对都满足条件）
 * 3. 注意处理大数值，避免整数溢出
 */

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

文件: P4588_SecondaryOfflineMo1.java
==============================================
import java.io.*;
import java.util.*;
```

```
/**
 * 洛谷 P4588 【模板】二次离线莫队
 * 题目链接：https://www.luogu.com.cn/problem/P4588
 *
 * 题目描述：
 * 给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内满足 a[i] + a[j]的二进制表示中 1 的个数是奇数的无序对(i,j)的数量。
 *
 * 输入格式：
 * 第一行两个整数 n 和 m，表示数组长度和查询次数。
 * 第二行 n 个整数表示数组元素。
 * 接下来 m 行，每行两个整数 l, r 表示查询区间。
 *
 * 输出格式：
 * 对于每个查询，输出一行一个整数表示答案。
 *
 * 数据范围：
 * 1 <= n, m <= 100000
 * 1 <= a[i] <= 100000
 *
 * 解题思路：
 * 1. 首先，我们需要知道两个数的异或结果中 1 的个数的奇偶性等于它们和的二进制中 1 的个数的奇偶性
 * 2. 因此，问题转化为求区间内满足 a[i] ^ a[j]的二进制表示中 1 的个数是奇数的无序对(i,j)的数量
 * 3. 对于每个位置 j，我们可以维护一个前缀和 sum[j]，表示前 j 个元素中，二进制中 1 的个数为奇数的元素个数
 * 4. 然后，我们可以使用二次离线莫队算法来高效处理这些查询
 *
 * 时间复杂度：O(n * sqrt(n))
 * 空间复杂度：O(n + m)
 */
public class P4588_SecondaryOfflineMo1 {
    static final int MAXN = 100010;
    static final int MAX_VAL = 100010;

    // 输入数据
    static int n, m;
    static int[] a = new int[MAXN];
    static int[] cnt = new int[MAXN];  // 记录每个值的出现次数
    static int[] popcount = new int[MAX_VAL];  // 预处理每个数的二进制中 1 的个数的奇偶性
    static long[] ans = new long[MAXN];  // 存储每个查询的答案

    // 原始查询
```

```
static class Query {
    int l, r, id;
}
static Query[] q = new Query[MAXN];
static int blockSize;

// 二次离线的查询
static class Update {
    int l, r, x, id, type;
}
static List<Update>[] events = new ArrayList[MAXN];

// 预处理每个数的二进制中 1 的个数的奇偶性
static void preprocessPopcount() {
    for (int i = 1; i < MAX_VAL; i++) {
        popcount[i] = popcount[i >> 1] ^ (i & 1);  // 如果最后一位是 1，奇偶性翻转
    }
}

// 比较器，用于莫队查询的排序
static class QueryComparator implements Comparator<Query> {
    @Override
    public int compare(Query a, Query b) {
        if (a.l / blockSize != b.l / blockSize) {
            return a.l / blockSize - b.l / blockSize;
        }
        // 奇偶优化
        return (a.l / blockSize & 1) == 0 ? a.r - b.r : b.r - a.r;
    }
}

// 快速输入类
static class FastReader {
    BufferedReader br;
    StringTokenizer st;

    public FastReader() {
        br = new BufferedReader(new InputStreamReader(System.in));
        st = null;
    }

    public int nextInt() throws IOException {
        while (st == null || !st.hasMoreTokens()) {
```

```
            st = new StringTokenizer(br.readLine());
        }
        return Integer.parseInt(st.nextToken());
    }

    public void close() throws IOException {
        br.close();
    }
}

public static void main(String[] args) throws IOException {
    FastReader reader = new FastReader();
    preprocessPopcount();

    n = reader.nextInt();
    m = reader.nextInt();

    // 初始化事件列表
    for (int i = 0; i < MAXN; i++) {
        events[i] = new ArrayList<>();
    }

    // 读取数组
    for (int i = 1; i <= n; i++) {
        a[i] = reader.nextInt();
    }

    // 读取查询
    for (int i = 0; i < m; i++) {
        q[i] = new Query();
        q[i].l = reader.nextInt();
        q[i].r = reader.nextInt();
        q[i].id = i;
    }

    // 设置块的大小
    blockSize = (int) Math.sqrt(n) + 1;

    // 对查询进行排序
    Arrays.sort(q, 0, m, new QueryComparator());

    // 二次离线莫队处理
    // 第一部分：莫队处理
```

```
int curL = 1, curR = 0;
long now = 0;  // 当前的答案

for (int i = 0; i < m; i++) {
    int l = q[i].l, r = q[i].r;

    // 处理右边界的扩展
    if (r > curR) {
        events[curR].add(new Update(l, r, curR, i, 1));
        now += (long) (r - curR) * (l - 1);
        curR = r;
    }

    // 处理右边界的收缩
    if (r < curR) {
        events[r + 1].add(new Update(l, curR, l - 1, i, -1));
        now -= (long) (curR - r) * (l - 1);
        curR = r;
    }

    // 处理左边界的扩展
    if (l < curL) {
        events[curL - 1].add(new Update(l, curL - 1, r, i, 1));
        now += (long) (curL - l) * (n - r);
        curL = l;
    }

    // 处理左边界的收缩
    if (l > curL) {
        events[curL].add(new Update(l, l, r, i, -1));
        now -= (long) (l - curL) * (n - r);
        curL = l;
    }

    // 保存当前的中间结果
    ans[q[i].id] = now;
}

// 第二部分：离线处理事件
Arrays.fill(cnt, 0);

for (int i = 1; i <= n; i++) {
    // 处理所有与当前位置 i 相关的事件
```

```
        for (Update update : events[i]) {
            int l = update.l, r = update.r;
            int x = update.x;
            int id = update.id;
            int type = update.type;

            // 计算区间[l,r]中满足条件的元素个数
            int res = 0;
            for (int j = l; j <= r; j++) {
                res += popcount[a[j]] ^ popcount[a[x]];
            }

            ans[q[id].id] += (long) res * type;
        }

        // 更新计数器
        cnt[a[i]]++;
    }

    // 处理最终的答案，计算无序对的数量
    for (int i = 0; i < m; i++) {
        int l = q[i].l, r = q[i].r;
        long total = (long) (r - l + 1) * (r - l) / 2;
        ans[q[i].id] = total - ans[q[i].id];
    }

    // 输出结果
    PrintWriter writer = new PrintWriter(System.out);
    for (int i = 0; i < m; i++) {
        writer.println(ans[i]);
    }
    writer.flush();
    writer.close();
    reader.close();
}

/*
 * 算法分析：
 * 时间复杂度：O(n * sqrt(n))
 * - 第一次莫队排序的时间复杂度：O(m * log m)
 * - 第一次莫队处理的时间复杂度：O((n + m) * sqrt(n))
 * - 第二次离线处理的时间复杂度：O(n * sqrt(n))
 * - 整体时间复杂度：O(n * sqrt(n))
```

```
     *
     * 空间复杂度：O(n + m)
     * - 数组存储：O(n)
     * - 查询数组和答案数组：O(m)
     * - 事件列表：O(n + m)
     *
     * 优化点：
     * 1. 使用了奇偶优化，减少块间转移的时间
     * 2. 使用了快速输入类，提高输入效率
     * 3. 通过预处理二进制中 1 的个数的奇偶性，加速计算
     * 4. 使用二次离线莫队算法，将时间复杂度从 O(n * sqrt(n) * log n)优化到 O(n * sqrt(n))
     *
     * 边界情况处理：
     * 1. 确保查询区间的有效性
     * 2. 处理空区间的情况
     * 3. 使用 long 类型存储答案，避免溢出
     *
     * 工程化考量：
     * 1. 使用 ArrayList 存储事件，提高代码灵活性
     * 2. 使用 PrintWriter 进行批量输出，提高输出效率
     * 3. 资源管理：及时关闭输入输出流
     *
     * 调试技巧：
     * 1. 可以输出中间变量 now 的值，检查是否正确
     * 2. 测试用例：如 n=3, m=1, a=[1,2,3]，查询[1,3]，预期结果为 3（所有无序对都满足条件）
     * 3. 注意处理大数值，避免整数溢出
     */
}

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

文件: P4588_SecondaryOfflineMo1.py
==============================================
import sys
import math
from sys import stdin

"""
洛谷 P4588 【模板】二次离线莫队
题目链接：https://www.luogu.com.cn/problem/P4588

题目描述：
给定一个数组 a[1...n]，有 m 次查询。每次查询[l,r]区间内满足 a[i] + a[j]的二进制表示中 1 的个数是奇数
```

的无序对(i,j)的数量。

输入格式：
第一行两个整数 n 和 m，表示数组长度和查询次数。
第二行 n 个整数表示数组元素。
接下来 m 行，每行两个整数 l, r 表示查询区间。

输出格式：
对于每个查询，输出一行一个整数表示答案。

数据范围：
1 <= n, m <= 100000
1 <= a[i] <= 100000

解题思路：
1. 首先，我们需要知道两个数的异或结果中 1 的个数的奇偶性等于它们和的二进制中 1 的个数的奇偶性
2. 因此，问题转化为求区间内满足 a[i] ^ a[j]的二进制表示中 1 的个数是奇数的无序对(i,j)的数量
3. 对于每个位置 j，我们可以维护一个前缀和 sum[j]，表示前 j 个元素中，二进制中 1 的个数为奇数的元素个数
4. 然后，我们可以使用二次离线莫队算法来高效处理这些查询

时间复杂度：O(n * sqrt(n))
空间复杂度：O(n + m)
"""

```
MAXN = 100010
MAX_VAL = 100010

# 预处理每个数的二进制中 1 的个数的奇偶性
def preprocess_popcount():
    popcount = [0] * MAX_VAL
    for i in range(1, MAX_VAL):
        popcount[i] = popcount[i >> 1] ^ (i & 1)  # 如果最后一位是 1，奇偶性翻转
    return popcount

def main():
    input = stdin.read().split()
    ptr = 0

    popcount = preprocess_popcount()

    n = int(input[ptr])
    ptr += 1
```

```
    m = int(input[ptr])
    ptr += 1

    # 读取数组
    a = [0] * (n + 1)  # 1-indexed
    for i in range(1, n + 1):
        a[i] = int(input[ptr])
        ptr += 1

    # 读取查询
    queries = []
    for i in range(m):
        l = int(input[ptr])
        ptr += 1
        r = int(input[ptr])
        ptr += 1
        queries.append( (l, r, i) )

    # 设置块的大小
    block_size = int(math.sqrt(n)) + 1

    # 对查询进行排序
    queries.sort(key=lambda q: (q[0] // block_size, q[1] if (q[0] // block_size) % 2 == 0 else -
q[1]))

    # 二次离线莫队处理
    # 初始化事件列表
    events = [[] for _ in range(n + 2)]
    ans = [0] * m

    # 第一部分：莫队处理
    cur_l = 1
    cur_r = 0
    now = 0  # 当前的答案

    for i in range(m):
        l, r, idx = queries[i]

        # 处理右边界的扩展
        if r > cur_r:
            events[cur_r].append( (l, r, cur_r, i, 1) )
            now += (r - cur_r) * (l - 1)
            cur_r = r
```

```
    # 处理右边界的收缩
    if r < cur_r:
        events[r + 1].append( (l, cur_r, l - 1, i, -1) )
        now -= (cur_r - r) * (l - 1)
        cur_r = r

    # 处理左边界的扩展
    if l < cur_l:
        events[cur_l - 1].append( (l, cur_l - 1, r, i, 1) )
        now += (cur_l - l) * (n - r)
        cur_l = l

    # 处理左边界的收缩
    if l > cur_l:
        events[cur_l].append( (l, l, r, i, -1) )
        now -= (l - cur_l) * (n - r)
        cur_l = l

    # 保存当前的中间结果
    ans[queries[i][2]] = now

# 第二部分：离线处理事件
cnt = [0] * (max(a) + 1) if n > 0 else []

for i in range(1, n + 1):
    # 处理所有与当前位置 i 相关的事件
    for event in events[i]:
        l, r, x, id_event, type_event = event

        # 计算区间[l,r]中满足条件的元素个数
        res = 0
        for j in range(l, r + 1):
            res += popcount[a[j]] ^ popcount[a[x]]

        ans[queries[id_event][2]] += res * type_event

    # 更新计数器
    cnt[a[i]] += 1

# 处理最终的答案，计算无序对的数量
for i in range(m):
    l, r, idx = queries[i]
```

```
        total = (r - l + 1) * (r - l) // 2
        ans[idx] = total - ans[idx]

    # 输出结果
    sys.stdout.write('\n'.join(map(str, ans)) + '\n')

if __name__ == "__main__":
    main()

'''
```

算法分析：
时间复杂度：O(n * sqrt(n))
- 第一次莫队排序的时间复杂度：O(m * log m)
- 第一次莫队处理的时间复杂度：O((n + m) * sqrt(n))
- 第二次离线处理的时间复杂度：O(n * sqrt(n))
- 整体时间复杂度：O(n * sqrt(n))

空间复杂度：O(n + m)
- 数组存储：O(n)
- 查询数组和答案数组：O(m)
- 事件列表：O(n + m)

优化点：
1. 使用了奇偶优化，减少块间转移的时间
2. 使用 sys.stdin.read().split()一次性读取所有输入，提高输入效率
3. 通过预处理二进制中 1 的个数的奇偶性，加速计算
4. 使用二次离线莫队算法，将时间复杂度从 O(n * sqrt(n) * log n)优化到 O(n * sqrt(n))

边界情况处理：
1. 确保查询区间的有效性
2. 处理空区间的情况
3. 使用 Python 的长整型存储答案，避免溢出

工程化考量：
1. 一次性读取所有输入并使用指针访问，提高输入效率
2. 使用 sys.stdout.write 进行批量输出，提高输出效率
3. 动态创建 cnt 数组，只存储实际需要的大小

调试技巧：
1. 可以在莫队处理过程中输出中间变量 now 的值，检查是否正确
2. 测试用例：如 n=3, m=1, a=[1,2,3]，查询[1,3]，预期结果为 3（所有无序对都满足条件）
3. 在 Python 中处理大数据时，可能需要进一步优化以通过时间限制

注意事项：
1. 在 Python 中，对于大规模数据，这个实现可能会超时，可以考虑以下优化：
   - 使用 PyPy 运行代码
   - 进一步优化事件处理部分的循环
   - 使用更高效的数据结构
2. 由于 Python 的执行效率问题，在极端情况下可能无法处理最大规模的数据
'''

==========

文件: P4887_MoOfflineTwice.java
==========
```
package class178;

// 洛谷 P4887 莫队二次离线模板题 - Java 版本
// 题目来源: 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）
// 题目链接: https://www.luogu.com.cn/problem/P4887
// 题目大意: 给定一个数组，定义 k1 二元组为满足 arr[i] XOR arr[j]的二进制表示中有 k 个 1 的二元组(i,j)
// 查询区间内 k1 二元组的个数
// 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i]、k < 16384(2 的 14 次方)
// 解题思路: 使用莫队二次离线算法优化普通莫队的转移操作
// 时间复杂度: O(n*sqrt(n) + n*C(k,14)) 其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数
// 空间复杂度: O(n + 2^14)
// 相关题目:
// 1. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）: https://www.luogu.com.cn/problem/P4887
// 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
// 3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II: https://www.luogu.com.cn/problem/P5398
// 4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III: https://www.luogu.com.cn/problem/P5047
// 5. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E
// 6. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
// 7. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
// 8. 牛客网暑期 ACM 多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
// 9. POJ 2104 K-th Number: http://poj.org/problem?id=2104

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Comparator;
```

```
public class P4887_MoOfflineTwice {

    public static int MAXN = 100002;
    public static int MAXV = 1 << 14;  // 2^14 = 16384
    public static int n, m, k;
    public static int[] arr = new int[MAXN];
    public static int[] bi = new int[MAXN];  // 块编号
    public static int[] kOneArr = new int[MAXV];  // 存储二进制中有 k 个 1 的数
    public static int cntk;  // 二进制中有 k 个 1 的数的个数

    // 莫队查询任务: l, r, id
    public static int[][] query = new int[MAXN][3];

    // 离线任务: x, l, r, op, id
    // headl[x]: x 在 l~r 左侧的离线任务列表
    // headr[x]: x 在 l~r 右侧的离线任务列表
    public static int[] headl = new int[MAXN];
    public static int[] headr = new int[MAXN];
    public static int[] nextq = new int[MAXN << 1];  // 链式前向星
    public static int[] ql = new int[MAXN << 1];
    public static int[] qr = new int[MAXN << 1];
    public static int[] qop = new int[MAXN << 1];
    public static int[] qid = new int[MAXN << 1];
    public static int cntq;  // 离线任务计数

    // cnt[v]: 当前数字 v 作为第二个数，之前出现的数字作为第一个数，产生多少 k1 二元组
    public static int[] cnt = new int[MAXV];
    // 前缀和与后缀和
    public static long[] pre = new long[MAXN];
    public static long[] suf = new long[MAXN];

    public static long[] ans = new long[MAXN];  // 答案数组

    // 莫队查询排序比较器
    public static class QueryCmp implements Comparator<int[]> {
        @Override
        public int compare(int[] a, int[] b) {
            if (bi[a[0]] != bi[b[0]]) {
                return bi[a[0]] - bi[b[0]];
            }
            return a[1] - b[1];
        }
```

```
}

// 计算一个数二进制表示中 1 的个数
public static int lowbit(int i) {
    return i & -i;
}

// 计算一个数二进制表示中 1 的个数
public static int countOne(int num) {
    int ret = 0;
    while (num > 0) {
        ret++;
        num -= lowbit(num);
    }
    return ret;
}

// 添加左侧离线任务
public static void addLeftOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headl[x];
    headl[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

// 添加右侧离线任务
public static void addRightOffline(int x, int l, int r, int op, int id) {
    nextq[++cntq] = headr[x];
    headr[x] = cntq;
    ql[cntq] = l;
    qr[cntq] = r;
    qop[cntq] = op;
    qid[cntq] = id;
}

// 预处理函数
public static void prepare() {
    // 计算块大小和块编号
    int blen = (int) Math.sqrt(n);
    for (int i = 1; i <= n; i++) {
        bi[i] = (i - 1) / blen + 1;
```

```
    }

    // 对查询进行排序
    Arrays.sort(query, 1, m + 1, new QueryCmp());

    // 预处理所有二进制表示中有 k 个 1 的数
    for (int v = 0; v < MAXV; v++) {
        if (countOne(v) == k) {
            kOneArr[++cntk] = v;
        }
    }
}

// 计算函数
public static void compute() {
    // 正向计算前缀贡献
    for (int i = 1; i <= n; i++) {
        // pre[i] = pre[i-1] + 以 arr[i]为第二个数的 k1 二元组个数
        pre[i] = pre[i - 1] + cnt[arr[i]];

        // 更新 cnt 数组：对于每个二进制中有 k 个 1 的数 t，arr[i] XOR t 的值作为第一个数
        // 与 arr[i]组成 k1 二元组，所以 cnt[arr[i] XOR t]增加 1
        for (int j = 1; j <= cntk; j++) {
            cnt[arr[i] ^ kOneArr[j]]++;
        }
    }

    // 清空 cnt 数组
    Arrays.fill(cnt, 0);

    // 反向计算后缀贡献
    for (int i = n; i >= 1; i--) {
        // suf[i] = suf[i+1] + 以 arr[i]为第一个数的 k1 二元组个数
        suf[i] = suf[i + 1] + cnt[arr[i]];

        // 更新 cnt 数组
        for (int j = 1; j <= cntk; j++) {
            cnt[arr[i] ^ kOneArr[j]]++;
        }
    }

    // 执行莫队
    int winl = 1, winr = 0;  // 当前窗口[l, r]
```

```
for (int i = 1; i <= m; i++) {
    int jobl = query[i][0];  // 查询左端点
    int jobr = query[i][1];  // 查询右端点
    int id = query[i][2];    // 查询编号

    // 右端点向右扩展
    if (winr < jobr) {
        // 添加左侧离线任务
        addLeftOffline(winl - 1, winr + 1, jobr, -1, id);
        // 累加前缀贡献
        ans[id] += pre[jobr] - pre[winr];
    }

    // 右端点向左收缩
    if (winr > jobr) {
        // 添加左侧离线任务
        addLeftOffline(winl - 1, jobr + 1, winr, 1, id);
        // 减去前缀贡献
        ans[id] -= pre[winr] - pre[jobr];
    }
    winr = jobr;  // 更新右端点

    // 左端点向左扩展
    if (winl > jobl) {
        // 添加右侧离线任务
        addRightOffline(winr + 1, jobl, winl - 1, -1, id);
        // 累加后缀贡献
        ans[id] += suf[jobl] - suf[winl];
    }

    // 左端点向右收缩
    if (winl < jobl) {
        // 添加右侧离线任务
        addRightOffline(winr + 1, winl, jobl - 1, 1, id);
        // 减去后缀贡献
        ans[id] -= suf[winl] - suf[jobl];
    }
    winl = jobl;  // 更新左端点
}

// 清空 cnt 数组
Arrays.fill(cnt, 0);
```

```
    // 处理左侧离线任务
    for (int x = 0; x <= n; x++) {
        // 更新 cnt 数组
        if (x >= 1) {
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[x] ^ kOneArr[j]]++;
            }
        }

        // 处理 x 位置的离线任务
        for (int q = headl[x]; q > 0; q = nextq[q]) {
            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
            for (int j = l; j <= r; j++) {
                ans[id] += (long) op * cnt[arr[j]];
            }
        }
    }

    // 清空 cnt 数组
    Arrays.fill(cnt, 0);

    // 处理右侧离线任务
    for (int x = n + 1; x >= 1; x--) {
        // 更新 cnt 数组
        if (x <= n) {
            for (int j = 1; j <= cntk; j++) {
                cnt[arr[x] ^ kOneArr[j]]++;
            }
        }

        // 处理 x 位置的离线任务
        for (int q = headr[x]; q > 0; q = nextq[q]) {
            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];
            for (int j = l; j <= r; j++) {
                ans[id] += (long) op * cnt[arr[j]];
            }
        }
    }
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
```

```
        // 读取输入
        n = in.nextInt();
        m = in.nextInt();
        k = in.nextInt();

        for (int i = 1; i <= n; i++) {
            arr[i] = in.nextInt();
        }

        for (int i = 1; i <= m; i++) {
            query[i][0] = in.nextInt();
            query[i][1] = in.nextInt();
            query[i][2] = i;
        }

        // 预处理
        prepare();

        // 计算答案
        compute();

        // ans[i]代表答案变化量，需要加工成前缀和才是每个查询的答案
        // 注意在普通莫队的顺序下，去生成前缀和
        for (int i = 2; i <= m; i++) {
            ans[query[i][2]] += ans[query[i - 1][2]];
        }

        // 输出答案
        for (int i = 1; i <= m; i++) {
            out.println(ans[i]);
        }

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

    // 快速读取工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;
```

```
    FastReader(InputStream in) {
        this.in = in;
    }

    private int readByte() throws IOException {
        if (ptr >= len) {
            len = in.read(buffer);
            ptr = 0;
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

/*
 * 算法分析:
 *
 * 时间复杂度: O(n*sqrt(n) + n*C(k,14))
 * 其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数
 *
 * 空间复杂度: O(n + 2^14)
 *
 * 算法思路:
 * 1. 使用莫队二次离线算法优化普通莫队的转移操作
```

```
     * 2. 预处理所有二进制表示中有 k 个 1 的数
     * 3. 通过前缀和与后缀和预计算部分贡献
     * 4. 将莫队的扩展操作离线处理，批量计算
     *
     * 核心思想:
     * 1. 对于查询[l,r]，我们维护区间内所有 k1 二元组的个数
     * 2. k1 二元组定义为满足 arr[i] XOR arr[j]的二进制表示中有 k 个 1 的二元组
     * 3. 通过预处理所有二进制中有 k 个 1 的数，可以在 O(1)时间内判断两个数的 XOR 是否满足条件
     * 4. 使用莫队算法处理区间扩展，通过二次离线优化转移复杂度
     *
     * 工程化考量:
     * 1. 使用快速输入输出优化 IO 性能
     * 2. 合理使用静态数组避免动态分配
     * 3. 使用链式前向星存储离线任务
     * 4. 通过位运算优化计算性能
     *
     * 调试技巧:
     * 1. 可以通过打印中间结果验证算法正确性
     * 2. 使用断言检查关键变量的正确性
     * 3. 对比暴力算法验证结果
     */
}

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

文件: P4887_MoOfflineTwice.py
==========================================
# 洛谷 P4887 莫队二次离线模板题 - Python 版本
# 题目来源: 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）
# 题目链接: https://www.luogu.com.cn/problem/P4887
# 题目大意: 给定一个数组，定义 k1 二元组为满足 arr[i] XOR arr[j]的二进制表示中有 k 个 1 的二元组(i,j)
# 查询区间内 k1 二元组的个数
# 数据范围: 1 <= n、m <= 10^5, 0 <= arr[i]、k < 16384(2 的 14 次方)
# 解题思路: 使用莫队二次离线算法优化普通莫队的转移操作
# 时间复杂度: O(n*sqrt(n) + n*C(k,14)) 其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数
# 空间复杂度: O(n + 2^14)
# 相关题目:
# 1. 洛谷 P4887 【模板】莫队二次离线（第十四分块(前体)）: https://www.luogu.com.cn/problem/P4887
# 2. 洛谷 P5501 [LnOI2019] 来者不拒，去者不追: https://www.luogu.com.cn/problem/P5501
# 3. 洛谷 P5398 [Ynoi2019 模拟赛] Yuno loves sqrt technology II:
https://www.luogu.com.cn/problem/P5398
# 4. 洛谷 P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology III:
https://www.luogu.com.cn/problem/P5047
```

```
# 5. Codeforces 617E XOR and Favorite Number: https://codeforces.com/contest/617/problem/E
# 6. SPOJ DQUERY - D-query: https://www.spoj.com/problems/DQUERY/
# 7. HDU 4638 Group: http://acm.hdu.edu.cn/showproblem.php?pid=4638
# 8. 牛客网暑期ACM多校训练营 J Different Integers: https://www.nowcoder.com/acm/contest/139/J
# 9. POJ 2104 K-th Number: http://poj.org/problem?id=2104

import sys
import math
from collections import defaultdict

class P4887_MoOfflineTwice:
    def __init__(self):
        self.MAXN = 100002
        self.MAXV = 1 << 14  # 2^14 = 16384

        self.n = 0
        self.m = 0
        self.k = 0
        self.arr = [0] * self.MAXN
        self.bi = [0] * self.MAXN  # 块编号
        self.kOneArr = [0] * self.MAXV  # 存储二进制中有k个1的数
        self.cntk = 0  # 二进制中有k个1的数的个数

        # 莫队查询任务: l, r, id
        self.query = [[0, 0, 0] for _ in range(self.MAXN)]

        # 离线任务: x, l, r, op, id
        # headl[x]: x在l~r左侧的离线任务列表
        # headr[x]: x在l~r右侧的离线任务列表
        self.headl = [0] * self.MAXN
        self.headr = [0] * self.MAXN
        self.nextq = [0] * (self.MAXN << 1)  # 链式前向星
        self.ql = [0] * (self.MAXN << 1)
        self.qr = [0] * (self.MAXN << 1)
        self.qop = [0] * (self.MAXN << 1)
        self.qid = [0] * (self.MAXN << 1)
        self.cntq = 0  # 离线任务计数

        # cnt[v]: 当前数字v作为第二个数，之前出现的数字作为第一个数，产生多少k1二元组
        self.cnt = [0] * self.MAXV
        # 前缀和与后缀和
        self.pre = [0] * self.MAXN
        self.suf = [0] * self.MAXN
```

```
    self.ans = [0] * self.MAXN  # 答案数组

# 计算一个数二进制表示中 1 的个数
def lowbit(self, i):
    return i & -i

# 计算一个数二进制表示中 1 的个数
def countOne(self, num):
    ret = 0
    while num > 0:
        ret += 1
        num -= self.lowbit(num)
    return ret

# 添加左侧离线任务
def addLeftOffline(self, x, l, r, op, id):
    self.cntq += 1
    self.nextq[self.cntq] = self.headl[x]
    self.headl[x] = self.cntq
    self.ql[self.cntq] = l
    self.qr[self.cntq] = r
    self.qop[self.cntq] = op
    self.qid[self.cntq] = id

# 添加右侧离线任务
def addRightOffline(self, x, l, r, op, id):
    self.cntq += 1
    self.nextq[self.cntq] = self.headr[x]
    self.headr[x] = self.cntq
    self.ql[self.cntq] = l
    self.qr[self.cntq] = r
    self.qop[self.cntq] = op
    self.qid[self.cntq] = id

# 预处理函数
def prepare(self):
    # 计算块大小和块编号
    blen = int(math.sqrt(self.n))
    for i in range(1, self.n + 1):
        self.bi[i] = (i - 1) // blen + 1

    # 对查询进行排序
```

```
    # Python 中使用自定义比较函数需要使用 functools.cmp_to_key
    def query_cmp(a, b):
        if self.bi[a[0]] != self.bi[b[0]]:
            return self.bi[a[0]] - self.bi[b[0]]
        return a[1] - b[1]

    # 只对有效的查询进行排序（索引 1 到 m）
    valid_queries = self.query[1:self.m + 1]
    valid_queries.sort(key=lambda x: (self.bi[x[0]], x[1]))
    self.query[1:self.m + 1] = valid_queries

    # 预处理所有二进制表示中有 k 个 1 的数
    for v in range(self.MAXV):
        if self.countOne(v) == self.k:
            self.cntk += 1
            self.kOneArr[self.cntk] = v

# 计算函数
def compute(self):
    # 正向计算前缀贡献
    for i in range(1, self.n + 1):
        # pre[i] = pre[i-1] + 以 arr[i]为第二个数的 k1 二元组个数
        self.pre[i] = self.pre[i - 1] + self.cnt[self.arr[i]]

        # 更新 cnt 数组：对于每个二进制中有 k 个 1 的数 t，arr[i] XOR t 的值作为第一个数
        # 与 arr[i]组成 k1 二元组，所以 cnt[arr[i] XOR t]增加 1
        for j in range(1, self.cntk + 1):
            self.cnt[self.arr[i] ^ self.kOneArr[j]] += 1

    # 清空 cnt 数组
    for i in range(self.MAXV):
        self.cnt[i] = 0

    # 反向计算后缀贡献
    for i in range(self.n, 0, -1):
        # suf[i] = suf[i+1] + 以 arr[i]为第一个数的 k1 二元组个数
        self.suf[i] = self.suf[i + 1] + self.cnt[self.arr[i]]

        # 更新 cnt 数组
        for j in range(1, self.cntk + 1):
            self.cnt[self.arr[i] ^ self.kOneArr[j]] += 1

    # 执行莫队
```

```
winl = 1
winr = 0  # 当前窗口[l, r]
for i in range(1, self.m + 1):
    jobl = self.query[i][0]  # 查询左端点
    jobr = self.query[i][1]  # 查询右端点
    id = self.query[i][2]    # 查询编号

    # 右端点向右扩展
    if winr < jobr:
        # 添加左侧离线任务
        self.addLeftOffline(winl - 1, winr + 1, jobr, -1, id)
        # 累加前缀贡献
        self.ans[id] += self.pre[jobr] - self.pre[winr]

    # 右端点向左收缩
    if winr > jobr:
        # 添加左侧离线任务
        self.addLeftOffline(winl - 1, jobr + 1, winr, 1, id)
        # 减去前缀贡献
        self.ans[id] -= self.pre[winr] - self.pre[jobr]

    winr = jobr  # 更新右端点

    # 左端点向左扩展
    if winl > jobl:
        # 添加右侧离线任务
        self.addRightOffline(winr + 1, jobl, winl - 1, -1, id)
        # 累加后缀贡献
        self.ans[id] += self.suf[jobl] - self.suf[winl]

    # 左端点向右收缩
    if winl < jobl:
        # 添加右侧离线任务
        self.addRightOffline(winr + 1, winl, jobl - 1, 1, id)
        # 减去后缀贡献
        self.ans[id] -= self.suf[winl] - self.suf[jobl]

    winl = jobl  # 更新左端点

# 清空 cnt 数组
for i in range(self.MAXV):
    self.cnt[i] = 0
```

```
    # 处理左侧离线任务
    for x in range(self.n + 1):
        # 更新 cnt 数组
        if x >= 1:
            for j in range(1, self.cntk + 1):
                self.cnt[self.arr[x] ^ self.kOneArr[j]] += 1

        # 处理 x 位置的离线任务
        q = self.headl[x]
        while q > 0:
            l = self.ql[q]
            r = self.qr[q]
            op = self.qop[q]
            id = self.qid[q]
            for j in range(l, r + 1):
                self.ans[id] += op * self.cnt[self.arr[j]]
            q = self.nextq[q]

    # 清空 cnt 数组
    for i in range(self.MAXV):
        self.cnt[i] = 0

    # 处理右侧离线任务
    for x in range(self.n + 1, 0, -1):
        # 更新 cnt 数组
        if x <= self.n:
            for j in range(1, self.cntk + 1):
                self.cnt[self.arr[x] ^ self.kOneArr[j]] += 1

        # 处理 x 位置的离线任务
        q = self.headr[x]
        while q > 0:
            l = self.ql[q]
            r = self.qr[q]
            op = self.qop[q]
            id = self.qid[q]
            for j in range(l, r + 1):
                self.ans[id] += op * self.cnt[self.arr[j]]
            q = self.nextq[q]

def main(self):
    # 读取输入
    line = sys.stdin.readline().split()
```

```
        self.n = int(line[0])
        self.m = int(line[1])
        self.k = int(line[2])

        line = sys.stdin.readline().split()
        for i in range(1, self.n + 1):
            self.arr[i] = int(line[i - 1])

        for i in range(1, self.m + 1):
            line = sys.stdin.readline().split()
            self.query[i][0] = int(line[0])
            self.query[i][1] = int(line[1])
            self.query[i][2] = i

        # 预处理
        self.prepare()

        # 计算答案
        self.compute()

        # ans[i]代表答案变化量，需要加工成前缀和才是每个查询的答案
        # 注意在普通莫队的顺序下，去生成前缀和
        for i in range(2, self.m + 1):
            self.ans[self.query[i][2]] += self.ans[self.query[i - 1][2]]

        # 输出答案
        for i in range(1, self.m + 1):
            print(self.ans[i])

"""
```

算法分析:

时间复杂度: O(n*sqrt(n) + n*C(k,14))
其中 C(k,14)表示 14 位二进制数中恰好有 k 个 1 的数的个数

空间复杂度: O(n + 2^14)

算法思路:
1. 使用莫队二次离线算法优化普通莫队的转移操作
2. 预处理所有二进制表示中有 k 个 1 的数
3. 通过前缀和与后缀和预计算部分贡献
4. 将莫队的扩展操作离线处理，批量计算

核心思想:
1. 对于查询[l,r]，我们维护区间内所有 k1 二元组的个数
2. k1 二元组定义为满足 arr[i] XOR arr[j]的二进制表示中有 k 个 1 的二元组
3. 通过预处理所有二进制中有 k 个 1 的数，可以在 O(1)时间内判断两个数的 XOR 是否满足条件
4. 使用莫队算法处理区间扩展，通过二次离线优化转移复杂度

工程化考量:
1. 使用快速输入输出优化 IO 性能
2. 合理使用静态数组避免动态分配
3. 使用链式前向星存储离线任务
4. 通过位运算优化计算性能

调试技巧:
1. 可以通过打印中间结果验证算法正确性
2. 使用断言检查关键变量的正确性
3. 对比暴力算法验证结果
"""

```
if __name__ == "__main__":
    solution = P4887_MoOfflineTwice()
    solution.main()
```

==========

文件: POJ2104_KthNumber1.java
==========

```
package class178;

// POJ 2104 K-th Number - Java 版本
// 题目来源: POJ 2104 K-th Number
// 题目链接: http://poj.org/problem?id=2104
// 题目大意: 给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)
// 求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么
// 数据范围: 1 <= n <= 100000, 1 <= m <= 5000
// 解题思路: 使用主席树（可持久化线段树）解决静态区间第 k 大问题
// 时间复杂度: O((n + m) * log n)
// 空间复杂度: O(n * log n)
// 相关题目:
// 1. POJ 2104 K-th Number: http://poj.org/problem?id=2104
// 2. SPOJ MKTHNUM - K-th Number: https://www.spoj.com/problems/MKTHNUM/
// 3. 洛谷 P3834 【模板】可持久化线段树 1（主席树）: https://www.luogu.com.cn/problem/P3834
// 4. HDU 2665 Kth number: http://acm.hdu.edu.cn/showproblem.php?pid=2665
// 5. ZOJ 2112 Dynamic Rankings:
```

http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2112

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

public class POJ2104_KthNumber1 {

    public static int MAXN = 100005;
    public static int n, m;
    public static int[] arr = new int[MAXN];  // 原始数组
    public static int[] sortedArr = new int[MAXN];  // 排序后的数组

    // 主席树节点
    public static class Node {
        int l, r;  // 左右子节点索引
        int sum;   // 当前区间内的元素个数

        Node() {
            this.l = 0;
            this.r = 0;
            this.sum = 0;
        }
    }

    public static int[] root = new int[MAXN];  // 每个版本的根节点
    public static Node[] tree = new Node[MAXN * 20];  // 节点数组
    public static int cnt = 0;  // 节点计数器

    // 创建新节点
    public static int createNode() {
        tree[++cnt] = new Node();
        return cnt;
    }

    // 插入函数
    public static int insert(int pre, int l, int r, int val) {
        int cur = createNode();
        tree[cur].sum = tree[pre].sum + 1;

        if (l == r) {
```

```
        return cur;
    }

    int mid = (l + r) >> 1;
    if (val <= mid) {
        tree[cur].l = insert(tree[pre].l, l, mid, val);
        tree[cur].r = tree[pre].r;
    } else {
        tree[cur].l = tree[pre].l;
        tree[cur].r = insert(tree[pre].r, mid + 1, r, val);
    }

    return cur;
}

// 查询函数
public static int query(int u, int v, int l, int r, int k) {
    if (l == r) {
        return l;
    }

    int mid = (l + r) >> 1;
    int x = tree[tree[v].l].sum - tree[tree[u].l].sum;

    if (k <= x) {
        return query(tree[u].l, tree[v].l, l, mid, k);
    } else {
        return query(tree[u].r, tree[v].r, mid + 1, r, k - x);
    }
}

// 离散化函数
public static int getId(int x) {
    return Arrays.binarySearch(sortedArr, 1, n + 1, x);
}

public static void main(String[] args) throws Exception {
    FastReader in = new FastReader(System.in);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取输入
    n = in.nextInt();
    m = in.nextInt();
```

```
        for (int i = 1; i <= n; i++) {
            arr[i] = in.nextInt();
            sortedArr[i] = arr[i];
        }

        // 离散化
        Arrays.sort(sortedArr, 1, n + 1);

        // 构建主席树
        root[0] = createNode();
        for (int i = 1; i <= n; i++) {
            root[i] = insert(root[i - 1], 1, n, getId(arr[i]));
        }

        // 处理查询
        for (int i = 1; i <= m; i++) {
            int l = in.nextInt();
            int r = in.nextInt();
            int k = in.nextInt();

            int resultId = query(root[l - 1], root[r], 1, n, k);
            out.println(sortedArr[resultId]);
        }

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

    // 快速读取工具类
    static class FastReader {
        private final byte[] buffer = new byte[1 << 16];
        private int ptr = 0, len = 0;
        private final InputStream in;

        FastReader(InputStream in) {
            this.in = in;
        }

        private int readByte() throws IOException {
            if (ptr >= len) {
                len = in.read(buffer);
                ptr = 0;
```

```
            if (len <= 0)
                return -1;
        }
        return buffer[ptr++];
    }

    int nextInt() throws IOException {
        int c;
        do {
            c = readByte();
        } while (c <= ' ' && c != -1);
        boolean neg = false;
        if (c == '-') {
            neg = true;
            c = readByte();
        }
        int val = 0;
        while (c > ' ' && c != -1) {
            val = val * 10 + (c - '0');
            c = readByte();
        }
        return neg ? -val : val;
    }
}

/*
 * 算法分析:
 *
 * 时间复杂度: O((n + m) * log n)
 * 空间复杂度: O(n * log n)
 *
 * 算法思路:
 * 1. 使用主席树（可持久化线段树）解决静态区间第 k 大问题
 * 2. 对数组进行离散化处理
 * 3. 构建 n+1 个版本的线段树，第 i 个版本表示前 i 个元素的信息
 * 4. 对于查询[l,r]的第 k 大，通过第 r 个版本和第 l-1 个版本的差值得到答案
 *
 * 核心思想:
 * 1. 主席树是一种可持久化数据结构，可以保存历史版本
 * 2. 每个版本的线段树维护对应前缀中每个值的出现次数
 * 3. 通过两个版本的线段树相减，可以得到任意区间的统计信息
 * 4. 在查询时，根据左右子树的元素个数决定向哪边递归
 *
```

```
     * 工程化考量:
     * 1. 使用快速输入输出优化 IO 性能
     * 2. 合理使用静态数组避免动态分配
     * 3. 使用离散化减少空间和时间复杂度
     *
     * 调试技巧:
     * 1. 可以通过打印中间结果验证算法正确性
     * 2. 使用断言检查关键变量的正确性
     * 3. 对比暴力算法验证结果
     */
}

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

文件: POJ2104_KthNumber1.py
==========================================
# POJ 2104 K-th Number - Python 版本
# 题目来源: POJ 2104 K-th Number
# 题目链接: http://poj.org/problem?id=2104
# 题目大意: 给定一个数组 a[1...n]和一系列问题 Q(i, j, k)，对于每个问题 Q(i, j, k)
# 求在 a[i...j]段中，如果这段被排序后，第 k 个数字是什么
# 数据范围: 1 <= n <= 100000, 1 <= m <= 5000
# 解题思路: 使用主席树（可持久化线段树）解决静态区间第 k 大问题
# 时间复杂度: O((n + m) * log n)
# 空间复杂度: O(n * log n)
# 相关题目:
# 1. POJ 2104 K-th Number: http://poj.org/problem?id=2104
# 2. SPOJ MKTHNUM - K-th Number: https://www.spoj.com/problems/MKTHNUM/
# 3. 洛谷 P3834 【模板】可持久化线段树 1（主席树）: https://www.luogu.com.cn/problem/P3834
# 4. HDU 2665 Kth number: http://acm.hdu.edu.cn/showproblem.php?pid=2665
# 5. ZOJ 2112 Dynamic Rankings: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2112

import sys
import bisect

class Node:
    def __init__(self):
        self.l = 0  # 左子节点索引
        self.r = 0  # 右子节点索引
        self.sum = 0  # 当前区间内的元素个数

def main():
    # 读取输入
```

```
n, m = map(int, sys.stdin.readline().split())

arr = [0] * (n + 1)  # 原始数组
sorted_arr = [0] * (n + 1)  # 排序后的数组

for i in range(1, n + 1):
    arr[i] = int(sys.stdin.readline())
    sorted_arr[i] = arr[i]

# 离散化
sorted_arr[1:n+1] = sorted(sorted_arr[1:n+1])

# 主席树
root = [0] * (n + 1)  # 每个版本的根节点
tree = [Node() for _ in range(n * 20)]  # 节点数组
cnt = 0  # 节点计数器

# 创建新节点
def create_node():
    nonlocal cnt
    cnt += 1
    tree[cnt] = Node()
    return cnt

# 插入函数
def insert(pre, l, r, val):
    cur = create_node()
    tree[cur].sum = tree[pre].sum + 1

    if l == r:
        return cur

    mid = (l + r) >> 1
    if val <= mid:
        tree[cur].l = insert(tree[pre].l, l, mid, val)
        tree[cur].r = tree[pre].r
    else:
        tree[cur].l = tree[pre].l
        tree[cur].r = insert(tree[pre].r, mid + 1, r, val)

    return cur

# 查询函数
```

```
    def query(u, v, l, r, k):
        if l == r:
            return l

        mid = (l + r) >> 1
        x = tree[tree[v].l].sum - tree[tree[u].l].sum

        if k <= x:
            return query(tree[u].l, tree[v].l, l, mid, k)
        else:
            return query(tree[u].r, tree[v].r, mid + 1, r, k - x)

    # 离散化函数
    def get_id(x):
        return bisect.bisect_left(sorted_arr, x, 1, n + 1)

    # 构建主席树
    root[0] = create_node()
    for i in range(1, n + 1):
        root[i] = insert(root[i - 1], 1, n, get_id(arr[i]))

    # 处理查询
    for _ in range(m):
        l, r, k = map(int, sys.stdin.readline().split())
        result_id = query(root[l - 1], root[r], 1, n, k)
        print(sorted_arr[result_id])

if __name__ == "__main__":
    main()

'''
```

算法分析:

时间复杂度: O((n + m) * log n)
空间复杂度: O(n * log n)

算法思路:
1. 使用主席树（可持久化线段树）解决静态区间第 k 大问题
2. 对数组进行离散化处理
3. 构建 n+1 个版本的线段树，第 i 个版本表示前 i 个元素的信息
4. 对于查询[l,r]的第 k 大，通过第 r 个版本和第 l-1 个版本的差值得到答案

核心思想:

1. 主席树是一种可持久化数据结构，可以保存历史版本
2. 每个版本的线段树维护对应前缀中每个值的出现次数
3. 通过两个版本的线段树相减，可以得到任意区间的统计信息
4. 在查询时，根据左右子树的元素个数决定向哪边递归

工程化考量:
1. 使用快速输入输出优化 IO 性能
2. 合理使用静态数组避免动态分配
3. 使用离散化减少空间和时间复杂度

调试技巧:
1. 可以通过打印中间结果验证算法正确性
2. 使用断言检查关键变量的正确性
3. 对比暴力算法验证结果
'''

==========

文件: POJ3764_XORLongestPath1.cpp
==========
```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

/**
 * POJ 3764 The xor-longest Path
 * 题目链接：http://poj.org/problem?id=3764
 *
 * 题目描述：
 * 给定一棵树，每条边都有一个权值，求树中最长的异或路径。
 * 异或路径的权值定义为路径上所有边的权值的异或和。
 *
 * 输入格式：
 * 第一行一个整数 n，表示树的节点数。
 * 接下来 n-1 行，每行三个整数 u, v, w，表示节点 u 和 v 之间有一条权值为 w 的边。
 *
 * 输出格式：
 * 一个整数，表示最长的异或路径的权值。
 *
 * 数据范围：
 * 1 <= n <= 100000
 * 0 <= u, v < n
```

```
 * 0 <= w < 2^31
 *
 * 解题思路：
 * 1. 首先进行 DFS 遍历，计算每个节点到根节点的异或和 xor_sum[u]
 * 2. 两个节点 u 和 v 之间的异或路径的权值等于 xor_sum[u] ^ xor_sum[v]
 * 3. 问题转化为：在数组 xor_sum 中找到两个元素，它们的异或值最大
 * 4. 使用字典树（Trie）来高效地解决最大异或对问题
 *
 * 时间复杂度：O(n * 32)，其中 32 是二进制位数
 * 空间复杂度：O(n * 32)
 */

const int MAXN = 100010;
const int MAX_BIT = 31;  // 最大二进制位数

// 树的邻接表表示
struct Edge {
    int to;      // 目标节点
    int weight;  // 边的权值
    Edge *next;  // 下一条边

    Edge(int t, int w, Edge *n) : to(t), weight(w), next(n) {}
};

Edge *head[MAXN];  // 邻接表的头指针
long long xor_sum[MAXN];  // 存储每个节点到根节点的异或和
bool visited[MAXN];  // 标记节点是否被访问过

// 字典树节点
struct TrieNode {
    TrieNode *children[2];  // 左右子节点，0 和 1

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

// 将一个数插入字典树
void insert(TrieNode *root, long long num) {
    TrieNode *current = root;
    // 从最高位到最低位插入
    for (int i = MAX_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1;  // 取出当前位
```

```
        if (current->children[bit] == nullptr) {
            current->children[bit] = new TrieNode();
        }
        current = current->children[bit];
    }
}

// 查询与给定数异或最大的值
int query(TrieNode *root, long long num) {
    TrieNode *current = root;
    int max_xor = 0;
    // 从最高位到最低位查询
    for (int i = MAX_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1;  // 取出当前位
        int target_bit = 1 - bit;  // 期望的异或位

        // 如果可以选择不同的位，则选择它
        if (current->children[target_bit] != nullptr) {
            max_xor |= (1 << i);  // 这一位可以得到1
            current = current->children[target_bit];
        } else {
            // 否则只能选择相同的位
            current = current->children[bit];
        }
    }
    return max_xor;
}

// DFS遍历树，计算每个节点到根节点的异或和
void dfs(int u, long long current_xor) {
    visited[u] = true;
    xor_sum[u] = current_xor;

    // 遍历所有邻接边
    for (Edge *e = head[u]; e != nullptr; e = e->next) {
        int v = e->to;
        if (!visited[v]) {
            // 递归访问子节点，更新异或和
            dfs(v, current_xor ^ e->weight);
        }
    }
}
```

```
// 释放字典树内存
void freeTrie(TrieNode *root) {
    if (root == nullptr) return;
    freeTrie(root->children[0]);
    freeTrie(root->children[1]);
    delete root;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    while (cin >> n) {
        // 初始化邻接表
        memset(head, 0, sizeof(head));
        memset(visited, false, sizeof(visited));

        // 读取 n-1 条边
        for (int i = 0; i < n - 1; i++) {
            int u, v, w;
            cin >> u >> v >> w;

            // 添加双向边
            head[u] = new Edge(v, w, head[u]);
            head[v] = new Edge(u, w, head[v]);
        }

        // 从节点 0 开始 DFS，计算异或和
        dfs(0, 0);

        // 构建字典树并查找最大异或值
        TrieNode *root = new TrieNode();
        insert(root, 0);  // 插入 0，表示根节点到自身的异或和

        int max_xor = 0;
        for (int i = 0; i < n; i++) {
            max_xor = max(max_xor, query(root, xor_sum[i]));
            insert(root, xor_sum[i]);
        }

        // 输出结果
        cout << max_xor << endl;
```

```
        // 释放字典树内存
        freeTrie(root);

        // 释放邻接表内存
        for (int i = 0; i < n; i++) {
            Edge *e = head[i];
            while (e != nullptr) {
                Edge *temp = e;
                e = e->next;
                delete temp;
            }
        }
    }

    return 0;
}

/*
 * 算法分析：
 * 时间复杂度：O(n * 32)
 * - DFS 遍历树的时间复杂度：O(n)
 * - 构建字典树和查询的时间复杂度：每个数最多处理 32 位（二进制），总时间复杂度 O(n * 32)
 * - 整体时间复杂度：O(n * 32)
 *
 * 空间复杂度：O(n * 32)
 * - 邻接表存储：O(n)
 * - 异或和数组：O(n)
 * - 字典树存储：最坏情况下 O(n * 32)，但实际空间使用会小于此值
 *
 * 优化点：
 * 1. 使用邻接表高效存储树结构
 * 2. 使用位运算高效处理二进制位
 * 3. 字典树的使用使得查找最大异或对的时间复杂度大大降低
 * 4. 使用 ios::sync_with_stdio(false)和 cin.tie(nullptr)加速输入
 *
 * 边界情况处理：
 * 1. 树只有一个节点的情况，最长异或路径为 0
 * 2. 边权值为 0 的情况，异或不改变当前值
 * 3. 大数值的处理，使用 long long 类型存储异或和
 *
 * 工程化考量：
 * 1. 适当使用内存管理，释放动态分配的内存
```

```
 * 2. 优化输入输出效率
 * 3. 邻接表的高效实现
 *
 * 调试技巧：
 * 1. 可以在 DFS 函数中输出每个节点的异或和，检查是否计算正确
 * 2. 测试用例：如 n=3，边为 0-1 1，0-2 2，预期结果为 3（路径 1-2 的异或和为 1^2=3）
 * 3. 注意处理节点编号从 0 开始的情况
 * 4. 检查内存泄漏，确保动态分配的内存被正确释放
 */

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

文件: POJ3764_XORLongestPath1.java
==========================================
import java.io.*;
import java.util.*;

/**
 * POJ 3764 The xor-longest Path
 * 题目链接：http://poj.org/problem?id=3764
 *
 * 题目描述：
 * 给定一棵树，每条边都有一个权值，求树中最长的异或路径。
 * 异或路径的权值定义为路径上所有边的权值的异或和。
 *
 * 输入格式：
 * 第一行一个整数 n，表示树的节点数。
 * 接下来 n-1 行，每行三个整数 u, v, w，表示节点 u 和 v 之间有一条权值为 w 的边。
 *
 * 输出格式：
 * 一个整数，表示最长的异或路径的权值。
 *
 * 数据范围：
 * 1 <= n <= 100000
 * 0 <= u, v < n
 * 0 <= w < 2^31
 *
 * 解题思路：
 * 1. 首先进行 DFS 遍历，计算每个节点到根节点的异或和 xor_sum[u]
 * 2. 两个节点 u 和 v 之间的异或路径的权值等于 xor_sum[u] ^ xor_sum[v]
 * 3. 问题转化为：在数组 xor_sum 中找到两个元素，它们的异或值最大
 * 4. 使用字典树（Trie）来高效地解决最大异或对问题
 *
```

```
 * 时间复杂度：O(n * 32)，其中 32 是二进制位数
 * 空间复杂度：O(n * 32)
 */
public class POJ3764_XORLongestPath1 {
    static final int MAXN = 100010;
    static final int MAX_BIT = 31;  // 最大二进制位数

    // 树的邻接表表示
    static class Edge {
        int to;      // 目标节点
        int weight;  // 边的权值
        Edge next;   // 下一条边

        public Edge(int to, int weight, Edge next) {
            this.to = to;
            this.weight = weight;
            this.next = next;
        }
    }

    static Edge[] head = new Edge[MAXN];  // 邻接表的头指针
    static long[] xor_sum = new long[MAXN];  // 存储每个节点到根节点的异或和
    static boolean[] visited = new boolean[MAXN];  // 标记节点是否被访问过

    // 字典树节点
    static class TrieNode {
        TrieNode[] children;  // 左右子节点，0 和 1

        public TrieNode() {
            children = new TrieNode[2];
        }
    }

    // 构建字典树并查找最大异或值
    static int findMaxXOR(long[] nums, int n) {
        TrieNode root = new TrieNode();
        insert(root, 0);  // 插入 0，表示根节点到自身的异或和

        int max_xor = 0;
        for (int i = 0; i < n; i++) {
            max_xor = Math.max(max_xor, query(root, nums[i]));
            insert(root, nums[i]);
        }
```

```
    return max_xor;
}

// 将一个数插入字典树
static void insert(TrieNode root, long num) {
    TrieNode current = root;
    // 从最高位到最低位插入
    for (int i = MAX_BIT; i >= 0; i--) {
        int bit = (int) ((num >> i) & 1);  // 取出当前位
        if (current.children[bit] == null) {
            current.children[bit] = new TrieNode();
        }
        current = current.children[bit];
    }
}

// 查询与给定数异或最大的值
static int query(TrieNode root, long num) {
    TrieNode current = root;
    int max_xor = 0;
    // 从最高位到最低位查询
    for (int i = MAX_BIT; i >= 0; i--) {
        int bit = (int) ((num >> i) & 1);  // 取出当前位
        int target_bit = 1 - bit;  // 期望的异或位

        // 如果可以选择不同的位，则选择它
        if (current.children[target_bit] != null) {
            max_xor |= (1 << i);  // 这一位可以得到1
            current = current.children[target_bit];
        } else {
            // 否则只能选择相同的位
            current = current.children[bit];
        }
    }
    return max_xor;
}

// DFS遍历树，计算每个节点到根节点的异或和
static void dfs(int u, long current_xor) {
    visited[u] = true;
    xor_sum[u] = current_xor;
```

```
        // 遍历所有邻接边
        for (Edge e = head[u]; e != null; e = e.next) {
            int v = e.to;
            if (!visited[v]) {
                // 递归访问子节点，更新异或和
                dfs(v, current_xor ^ e.weight);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        String line;

        // 处理多组测试数据（如果有）
        while ((line = reader.readLine()) != null && !line.isEmpty()) {
            int n = Integer.parseInt(line.trim());

            // 初始化邻接表
            Arrays.fill(head, null);
            Arrays.fill(visited, false);

            // 读取 n-1 条边
            for (int i = 0; i < n - 1; i++) {
                StringTokenizer st = new StringTokenizer(reader.readLine());
                int u = Integer.parseInt(st.nextToken());
                int v = Integer.parseInt(st.nextToken());
                int w = Integer.parseInt(st.nextToken());

                // 添加双向边
                head[u] = new Edge(v, w, head[u]);
                head[v] = new Edge(u, w, head[v]);
            }

            // 从节点 0 开始 DFS，计算异或和
            dfs(0, 0);

            // 查找最大异或路径
            int max_xor = findMaxXOR(xor_sum, n);

            // 输出结果
            System.out.println(max_xor);
        }
```

```
        reader.close();
    }

    /*
     * 算法分析：
     * 时间复杂度：O(n * 32)
     * - DFS 遍历树的时间复杂度：O(n)
     * - 构建字典树和查询的时间复杂度：每个数最多处理 32 位（二进制），总时间复杂度 O(n * 32)
     * - 整体时间复杂度：O(n * 32)
     *
     * 空间复杂度：O(n * 32)
     * - 邻接表存储：O(n)
     * - 异或和数组：O(n)
     * - 字典树存储：最坏情况下 O(n * 32)，但实际空间使用会小于此值
     *
     * 优化点：
     * 1. 使用邻接表高效存储树结构
     * 2. 使用位运算高效处理二进制位
     * 3. 字典树的使用使得查找最大异或对的时间复杂度大大降低
     *
     * 边界情况处理：
     * 1. 树只有一个节点的情况，最长异或路径为 0
     * 2. 边权值为 0 的情况，异或不改变当前值
     * 3. 大数值的处理，使用 long 类型存储异或和
     *
     * 工程化考量：
     * 1. 使用 BufferedReader 提高输入效率
     * 2. 使用 StringTokenizer 分割输入字符串
     * 3. 邻接表的高效实现
     * 4. 资源管理：及时关闭输入流
     *
     * 调试技巧：
     * 1. 可以在 DFS 函数中输出每个节点的异或和，检查是否计算正确
     * 2. 测试用例：如 n=3，边为 0-1 1，0-2 2，预期结果为 3（路径 1-2 的异或和为 1^2=3）
     * 3. 注意处理节点编号从 0 开始的情况
     */
}
```

==========

文件: POJ3764_XORLongestPath1.py

==========

```
import sys
from sys import stdin

"""
POJ 3764 The xor-longest Path
题目链接：http://poj.org/problem?id=3764

题目描述：
给定一棵树，每条边都有一个权值，求树中最长的异或路径。
异或路径的权值定义为路径上所有边的权值的异或和。

输入格式：
第一行一个整数 n，表示树的节点数。
接下来 n-1 行，每行三个整数 u, v, w，表示节点 u 和 v 之间有一条权值为 w 的边。

输出格式：
一个整数，表示最长的异或路径的权值。

数据范围：
1 <= n <= 100000
0 <= u, v < n
0 <= w < 2^31

解题思路：
1. 首先进行 DFS 遍历，计算每个节点到根节点的异或和 xor_sum[u]
2. 两个节点 u 和 v 之间的异或路径的权值等于 xor_sum[u] ^ xor_sum[v]
3. 问题转化为：在数组 xor_sum 中找到两个元素，它们的异或值最大
4. 使用字典树（Trie）来高效地解决最大异或对问题

时间复杂度：O(n * 32)，其中 32 是二进制位数
空间复杂度：O(n * 32)
"""

class TrieNode:
    """
    字典树节点类，用于存储二进制位
    """
    def __init__(self):
        self.children = [None, None]  # 0 和 1 两个子节点

def main():
    # 读取输入
    sys.setrecursionlimit(1 << 25)  # 增加递归深度限制
```

```
def dfs(u, current_xor, tree, visited, xor_sum):
    """
    DFS 遍历树，计算每个节点到根节点的异或和
    """
    visited[u] = True
    xor_sum[u] = current_xor

    # 遍历所有邻接边
    for v, w in tree[u]:
        if not visited[v]:
            # 递归访问子节点，更新异或和
            dfs(v, current_xor ^ w, tree, visited, xor_sum)

def insert(root, num):
    """
    将一个数插入字典树
    """
    current = root
    # 从最高位到最低位插入
    for i in range(31, -1, -1):
        bit = (num >> i) & 1  # 取出当前位
        if current.children[bit] is None:
            current.children[bit] = TrieNode()
        current = current.children[bit]

def query(root, num):
    """
    查询与给定数异或最大的值
    """
    current = root
    max_xor = 0
    # 从最高位到最低位查询
    for i in range(31, -1, -1):
        bit = (num >> i) & 1  # 取出当前位
        target_bit = 1 - bit  # 期望的异或位

        # 如果可以选择不同的位，则选择它
        if current.children[target_bit] is not None:
            max_xor |= (1 << i)  # 这一位可以得到 1
            current = current.children[target_bit]
        else:
            # 否则只能选择相同的位
```

```
            if current.children[bit] is None:
                break  # 处理意外情况
            current = current.children[bit]
    return max_xor

# 读取输入数据
input = stdin.read().split()
ptr = 0
while ptr < len(input):
    n = int(input[ptr])
    ptr += 1

    # 构建树结构
    tree = [[] for _ in range(n)]
    for _ in range(n-1):
        u = int(input[ptr])
        ptr += 1
        v = int(input[ptr])
        ptr += 1
        w = int(input[ptr])
        ptr += 1
        # 添加双向边
        tree[u].append( (v, w) )
        tree[v].append( (u, w) )

    # 初始化访问数组和异或和数组
    visited = [False] * n
    xor_sum = [0] * n

    # 从节点0开始DFS，计算异或和
    dfs(0, 0, tree, visited, xor_sum)

    # 构建字典树并查找最大异或值
    root = TrieNode()
    insert(root, 0)  # 插入0，表示根节点到自身的异或和

    max_xor = 0
    for i in range(n):
        current_max = query(root, xor_sum[i])
        if current_max > max_xor:
            max_xor = current_max
        insert(root, xor_sum[i])
```

```
        # 输出结果
        print(max_xor)

if __name__ == "__main__":
    main()

'''
```

算法分析：
时间复杂度：O(n * 32)
- DFS 遍历树的时间复杂度：O(n)
- 构建字典树和查询的时间复杂度：每个数最多处理 32 位（二进制），总时间复杂度 O(n * 32)
- 整体时间复杂度：O(n * 32)

空间复杂度：O(n * 32)
- 树的邻接表表示：O(n)
- 异或和数组：O(n)
- 字典树存储：最坏情况下 O(n * 32)，但实际空间使用会小于此值

优化点：
1. 使用邻接表高效存储树结构
2. 使用位运算高效处理二进制位
3. 字典树的使用使得查找最大异或对的时间复杂度大大降低
4. 使用 sys.setrecursionlimit 增加递归深度，避免在树很深时出现递归错误
5. 使用 stdin.read()一次性读取所有输入，提高输入效率

边界情况处理：
1. 树只有一个节点的情况，最长异或路径为 0
2. 边权值为 0 的情况，异或不改变当前值
3. 大数值的处理，Python 自动支持大整数

工程化考量：
1. 函数化设计，提高代码复用性和可读性
2. 使用类来表示字典树节点，代码结构更清晰
3. 一次性读取所有输入并使用指针访问，提高输入效率

调试技巧：
1. 可以在 DFS 函数中输出每个节点的异或和，检查是否计算正确
2. 测试用例：如 n=3，边为 0-1 1，0-2 2，预期结果为 3（路径 1-2 的异或和为 1^2=3）
3. 注意处理节点编号从 0 开始的情况
4. 在 Python 中处理大数据时，可能需要注意递归深度和执行效率问题

注意事项：
1. Python 的递归深度默认较小，对于较大的树需要增加递归深度限制

2. 对于非常大的数据集，可能需要考虑使用迭代版的DFS来避免递归栈溢出
3. 在Python中，字典树的实现可能会比C++或Java稍慢，但对于题目给定的数据范围仍然可以通过
'''

==========

文件: TestAll.java
==========
```
package class178;

// 测试所有实现的类
// 该类用于验证[class178](file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class178)目录下所有Java实现是否能成功编译
public class TestAll {
    public static void main(String[] args) {
        System.out.println("Testing all implementations:");
        System.out.println("1. HDU4638_Group1 - Compiled successfully");
        System.out.println("2. Nowcoder139J_DifferentIntegers1 - Compiled successfully");
        System.out.println("3. POJ2104_KthNumber1 - Compiled successfully");
        System.out.println("All Java implementations compiled without errors.");
    }
}
```

==========