

=====

文件夹: class058_BitsetAndBitManipulation

=====

[Markdown 文件]

=====

文件: README.md

=====

位集算法与位操作专题 (Bitset Algorithms & Bit Manipulation)

目录

- [算法概述] (#算法概述)
- [核心概念] (#核心概念)
- [题目分类] (#题目分类)
- [代码实现] (#代码实现)
- [复杂度分析] (#复杂度分析)
- [工程化考量] (#工程化考量)
- [面试技巧] (#面试技巧)
- [扩展应用] (#扩展应用)

算法概述

位集算法是利用二进制位来表示和操作数据的高效技术。通过位运算，可以实现集合操作、状态压缩、优化计算等多种功能。

核心优势

- **空间效率**: 使用位表示状态，极大节省内存
- **时间效率**: 位运算具有极高的执行速度
- **并行处理**: 多位同时操作，实现并行计算

核心概念

基本位操作

```
```cpp
// 设置位
x |= (1 << n); // 设置第 n 位为 1
x &= ~(1 << n); // 设置第 n 位为 0
```

```
// 检查位
if (x & (1 << n)) // 检查第 n 位是否为 1
```

```
// 切换位
x ^= (1 << n); // 切换第 n 位
```

```
// 获取最低位的 1
lowbit = x & -x;

// 清除最低位的 1
x &= (x - 1);
...
...
```

#### #### 常用位运算技巧

1. **Brian Kernighan 算法**: 计算 1 的个数
2. **位掩码技术**: 状态压缩和权限管理
3. **格雷码**: 相邻状态只有一位不同
4. **快速幂算法**: 利用二进制分解指数

#### ## 题目分类

##### #### 基础位操作

1. **LeetCode 191 - 位 1 的个数**
  - 题目: 计算无符号整数的二进制表示中 1 的个数
  - 最优解: Brian Kernighan 算法  $O(k)$   $k$  为 1 的个数
  - 关键技巧: `n & (n-1)` 清除最低位的 1
  - 链接: <https://leetcode.com/problems/number-of-1-bits/>
2. **LeetCode 231 - 2 的幂**
  - 题目: 判断整数是否是 2 的幂
  - 最优解: 位运算  $O(1)$
  - 关键技巧: `n & (n-1) == 0`
  - 链接: <https://leetcode.com/problems/power-of-two/>
3. **LeetCode 342 - 4 的幂**
  - 题目: 判断整数是否是 4 的幂
  - 最优解: 位运算 + 数学  $O(1)$
  - 关键技巧: 检查是否是 2 的幂且 1 出现在奇数位
  - 链接: <https://leetcode.com/problems/power-of-four/>
4. **LeetCode 136 - 只出现一次的数字**
  - 题目: 找出数组中只出现一次的数字
  - 最优解: 异或运算  $O(n)$
  - 关键技巧: `a ^ a = 0`, `a ^ 0 = a`
  - 链接: <https://leetcode.com/problems/single-number/>
5. **LeetCode 137 - 只出现一次的数字 II**
  - 题目: 找出数组中只出现一次的数字 (其他出现三次)

- 最优解：位状态机  $O(n)$
- 关键技巧：使用两个变量模拟三进制计数
- 链接：<https://leetcode.com/problems/single-number-ii/>

## 6. \*\*LeetCode 260 - 只出现一次的数字 III\*\*

- 题目：找出数组中两个只出现一次的数字
- 最优解：分组异或  $O(n)$
- 关键技巧：根据差异位分组
- 链接：<https://leetcode.com/problems/single-number-iii/>

## #### 位集应用

### 7. \*\*LeetCode 78 - 子集\*\*

- 题目：求数组的所有子集
- 最优解：位枚举  $O(n \cdot 2^n)$
- 关键技巧：使用二进制位表示元素选择
- 链接：<https://leetcode.com/problems/subsets/>

### 8. \*\*LeetCode 90 - 子集 II\*\*

- 题目：求包含重复元素数组的所有子集
- 最优解：位枚举 + 排序去重  $O(n \cdot 2^n)$
- 关键技巧：使用集合去重
- 链接：<https://leetcode.com/problems/subsets-ii/>

### 9. \*\*POJ 2443 - Set Operation\*\*

- 题目：查询两个元素是否同时属于至少一个集合
- 最优解：bitset 优化  $O(N \cdot C + Q)$
- 关键技巧：使用 bitset 记录元素在哪些集合中出现
- 链接：<http://poj.org/problem?id=2443>

### 10. \*\*LeetCode 2166 - Design Bitset\*\*

- 题目：设计一个支持多种操作的位集数据结构
- 最优解：懒标记优化  $O(1)$
- 关键技巧：使用懒标记优化 flip 操作
- 链接：<https://leetcode.com/problems/design-bitset/>

## #### 位运算优化

### 11. \*\*AtCoder AGC020 C - Median Sum\*\*

- 题目：计算所有子集和的中位数
- 最优解：bitset 优化 DP  $O(N * \text{sum} / 32)$
- 关键技巧：使用 bitset 表示所有可能的子集和
- 链接：[https://atcoder.jp/contests/agc020/tasks/agc020\\_c](https://atcoder.jp/contests/agc020/tasks/agc020_c)

### 12. \*\*LeetCode 338 - Counting Bits\*\*

- 题目：计算 0 到 n 每个数字的二进制表示中 1 的个数
- 最优解：动态规划  $O(n)$
- 关键技巧：``dp[i] = dp[i >> 1] + (i & 1)``
- 链接：<https://leetcode.com/problems/counting-bits/>

#### 13. \*\*LeetCode 268 - Missing Number\*\*

- 题目：找到数组中缺失的数字
- 最优解：异或运算  $O(n)$
- 关键技巧：利用异或的性质
- 链接：<https://leetcode.com/problems/missing-number/>

#### 14. \*\*LeetCode 190 - Reverse Bits\*\*

- 题目：颠倒二进制位
- 最优解：位操作  $O(1)$
- 关键技巧：逐位颠倒
- 链接：<https://leetcode.com/problems/reverse-bits/>

### ### 高级位算法

#### 15. \*\*LeetCode 52 - N 皇后 II\*\*

- 题目：计算 N 皇后问题的解决方案数量
- 最优解：位运算回溯  $O(n!)$
- 关键技巧：使用位运算记录列和对角线占用
- 链接：<https://leetcode.com/problems/n-queens-ii/>

#### 16. \*\*LeetCode 37 - 解数独\*\*

- 题目：解数独问题
- 最优解：位运算 + 回溯  $O(9^m)$
- 关键技巧：使用位掩码记录数字使用情况
- 链接：<https://leetcode.com/problems/sudoku-solver/>

#### 17. \*\*LeetCode 464 - 我能赢吗\*\*

- 题目：博弈论问题，判断先手是否能赢
- 最优解：状态压缩 DP + 记忆化搜索
- 关键技巧：使用位掩码表示已选择的数字
- 链接：<https://leetcode.com/problems/can-i-win/>

#### 18. \*\*LeetCode 473 - 火柴拼正方形\*\*

- 题目：判断是否能用所有火柴拼成正方形
- 最优解：状态压缩 DP + 回溯
- 关键技巧：使用位掩码表示已使用的火柴
- 链接：<https://leetcode.com/problems/matchsticks-to-square/>

#### 19. \*\*LeetCode 698 - 划分为 k 个相等的子集\*\*

- 题目：将数组划分为 k 个和相等的子集
- 最优解：状态压缩 DP + 回溯
- 关键技巧：使用位掩码表示已使用的元素
- 链接：<https://leetcode.com/problems/partition-to-k-equal-sum-subsets/>

## 20. \*\*Codeforces 1556D – Take a Guess\*\*

- 题目：交互式问题，通过 AND 和 OR 操作猜出数组
- 最优解：位运算 + 交互式算法
- 关键技巧：利用 AND 和 OR 操作恢复原数组
- 链接：<https://codeforces.com/contest/1556/problem/D>

### ### HackerRank 题目

#### 21. \*\*HackerRank – Lonely Integer\*\*

- 题目：找出数组中只出现一次的数字
- 最优解：异或运算  $O(n)$
- 关键技巧： $\text{`a} \wedge \text{a} = 0$ ,  $\text{`a} \wedge 0 = \text{a}$
- 链接：<https://www.hackerrank.com/challenges/ctci-lonely-integer/problem>

#### 22. \*\*HackerRank – Maximizing XOR\*\*

- 题目：在给定范围内找到两个数的最大异或值
- 最优解：Trie 树  $O(32*n)$
- 关键技巧：使用 Trie 树存储二进制表示
- 链接：<https://www.hackerrank.com/challenges/maximizing-xor/problem>

#### 23. \*\*HackerRank – Sum vs XOR\*\*

- 题目：找到满足  $n+x = n \wedge x$  的 x 的个数
- 最优解：位运算  $O(\log n)$
- 关键技巧：分析加法和异或的关系
- 链接：<https://www.hackerrank.com/challenges/sum-vs-xor/problem>

#### 24. \*\*HackerRank – Flipping Bits\*\*

- 题目：翻转 32 位无符号整数的所有位
- 最优解：位运算  $O(1)$
- 关键技巧：使用异或操作翻转所有位
- 链接：<https://www.hackerrank.com/challenges/flipping-bits/problem>

### ### 洛谷题目

#### 25. \*\*洛谷 P7076 – 动物园\*\*

- 题目：动物园问题，利用位运算优化解法
- 最优解：位运算 + 组合计数
- 关键技巧：将属性转化为二进制位操作
- 链接：<https://www.luogu.com.cn/problem/P7076>

## 26. \*\*洛谷 P4424 - 寻宝游戏\*\*

- 题目：从 0 开始对 n 个数进行 AND 或 OR 操作得到目标数的方案数
- 最优解：位运算 + 思维题
- 关键技巧：位运算思维
- 链接：<https://www.luogu.com.cn/problem/P4424>

## ## 补充题目（从各大算法平台收集）

### #### LeetCode 额外题目

#### 27. \*\*LeetCode 29 - 两数相除\*\*

- 题目：不使用乘法、除法和 mod 运算符实现两数相除
- 最优解：位运算 + 二分查找  $O(\log n)$
- 关键技巧：使用左移实现快速乘法
- 链接：<https://leetcode.com/problems/divide-two-integers/>

#### 28. \*\*LeetCode 50 - Pow(x, n)\*\*

- 题目：实现幂函数
- 最优解：快速幂算法  $O(\log n)$
- 关键技巧：将指数分解为二进制
- 链接：<https://leetcode.com/problems/powx-n/>

#### 29. \*\*LeetCode 60 - 排列序列\*\*

- 题目：返回第 k 个排列
- 最优解：数学 + 位标记  $O(n^2)$
- 关键技巧：使用阶乘计算确定每位数字
- 链接：<https://leetcode.com/problems/permutation-sequence/>

#### 30. \*\*LeetCode 89 - 格雷编码\*\*

- 题目：生成格雷编码序列
- 最优解：镜像反射法  $O(2^n)$
- 关键技巧：`G(i) = i ^ (i >> 1)`
- 链接：<https://leetcode.com/problems/gray-code/>

#### 31. \*\*LeetCode 134 - 加油站\*\*

- 题目：找到能环绕一圈的加油站
- 最优解：贪心算法  $O(n)$
- 关键技巧：累计油量差
- 链接：<https://leetcode.com/problems/gas-station/>

### #### Codeforces 题目

#### 32. \*\*Codeforces 1338A - Powered Addition\*\*

- 题目：通过加 2 的幂使数组非递减
- 最优解：贪心 + 位运算  $O(n)$

- 关键技巧：计算需要的最大 2 的幂
- 链接：<https://codeforces.com/contest/1338/problem/A>

### 33. \*\*Codeforces 1556D – Take a Guess\*\*

- 题目：交互式问题，通过 AND 和 OR 操作猜出数组
- 最优解：位运算 + 交互式算法
- 关键技巧：利用 AND 和 OR 操作恢复原数组
- 链接：<https://codeforces.com/contest/1556/problem/D>

### 34. \*\*Codeforces 1322A – Unusual Competitions\*\*

- 题目：通过交换使括号序列合法
- 最优解：贪心 + 位运算
- 关键技巧：分析括号序列的性质
- 链接：<https://codeforces.com/contest/1322/problem/A>

## #### AtCoder 题目

### 35. \*\*AtCoder ABC194E – Mex Min\*\*

- 题目：计算所有长度为 M 的子数组的 mex 的最小值
- 最优解：滑动窗口 + 位运算
- 关键技巧：使用位运算维护 mex
- 链接：[https://atcoder.jp/contests/abc194/tasks/abc194\\_e](https://atcoder.jp/contests/abc194/tasks/abc194_e)

### 36. \*\*AtCoder ARC118C – Coprime Set\*\*

- 题目：构造一个互质集合
- 最优解：数学 + 位运算构造
- 关键技巧：利用互质的性质
- 链接：[https://atcoder.jp/contests/arc118/tasks/arc118\\_c](https://atcoder.jp/contests/arc118/tasks/arc118_c)

## #### 其他平台题目

### 37. \*\*USACO – Subset Sum\*\*

- 题目：子集和问题
- 最优解：位集 DP  $O(n*sum)$
- 关键技巧：使用 bitset 优化
- 链接：<http://www.usaco.org/index.php?page=viewproblem2&cpid=100>

### 38. \*\*SPOJ – BALIFE\*\*

- 题目：负载平衡问题
- 最优解：贪心 + 位运算优化
- 关键技巧：分析负载转移
- 链接：<https://www.spoj.com/problems/BALIFE/>

## ## 代码实现

### ### Java 实现

所有题目都有完整的 Java 实现，包含：

- 详细注释说明算法思路
- 时间/空间复杂度分析
- 边界情况处理
- 单元测试示例

### ### C++ 实现

所有题目都有完整的 C++ 实现，包含：

- 标准库 `bitset` 使用
- 位运算优化技巧
- 模板编程应用
- 性能测试代码

### ### Python 实现

所有题目都有完整的 Python 实现，包含：

- Pythonic 的位操作写法
- 生成器表达式应用
- 装饰器性能测试
- 类型注解支持

## ## 复杂度分析

### ### 时间复杂度分类

1. **\*\* $O(1)$  操作\*\***: 基本位检查、设置、清除
2. **\*\* $O(\log n)$  操作\*\***: 快速幂、二分位操作
3. **\*\* $O(n)$  操作\*\***: 遍历位、统计 1 的个数
4. **\*\* $O(n \cdot 2^n)$  操作\*\***: 子集枚举、状态压缩
5. **\*\* $O(n!)$  操作\*\***: 全排列、N 皇后问题

### ### 空间复杂度优化

1. **\*\*原地操作\*\***: 多数位运算可以原地进行
2. **\*\*位压缩\*\***: 使用位表示状态，极大节省空间
3. **\*\*流式处理\*\***: 对于大数据可以流式处理

## ## 工程化考量

### ### 1. 异常处理与边界情况

```
```java
// 检查整数溢出
public static int divide(int dividend, int divisor) {
    if (dividend == Integer.MIN_VALUE && divisor == -1) {
        return Integer.MAX_VALUE; // 处理溢出
    }
}
```

```
    }
    // ... 其他逻辑
}

// 处理负数情况
public static int hammingWeight(int n) {
    // 处理负数的二进制表示
    return Integer.bitCount(n);
}
```

```

### ### 2. 性能优化策略

```
```cpp
// 使用内联函数
inline int lowbit(int x) {
    return x & -x;
}
```

```

### // 循环展开

```
for (int i = 0; i < 32; i += 4) {
 count += (n >> i) & 1;
 count += (n >> (i+1)) & 1;
 count += (n >> (i+2)) & 1;
 count += (n >> (i+3)) & 1;
}
```

```

3. 可测试性设计

```
```python
import unittest

class TestBitAlgorithms(unittest.TestCase):
 def test_hamming_weight(self):
 self.assertEqual(BitAlgorithm.hammingWeight(0b1011), 3)
 self.assertEqual(BitAlgorithm.hammingWeight(0), 0)
 self.assertEqual(BitAlgorithm.hammingWeight(0xFFFFFFFF), 32)
```

```

4. 文档化与使用说明

每个算法都包含：

- 函数签名和参数说明
- 返回值说明
- 使用示例

- 常见问题解答

面试技巧

笔试核心策略

1. **模板准备**: 提前准备常用位操作模板
2. **边界测试**: 测试 0、负数、边界值
3. **性能分析**: 分析时间/空间复杂度
4. **代码简洁**: 使用位运算简化代码

面试表达技巧

1. **思路清晰**: 先讲暴力解法，再讲优化思路
2. **复杂度分析**: 明确时间/空间复杂度
3. **边界处理**: 讨论各种边界情况
4. **优化空间**: 提出进一步优化可能

常见问题回答模板

**Q: 如何判断一个数是否是 2 的幂? **

A: 使用 `n & (n-1) == 0` 且 `n > 0`。因为 2 的幂的二进制表示中只有 1 个 1，清除最低位的 1 后应该为 0。

**Q: 如何计算一个数的二进制中 1 的个数? **

A: 有三种方法: 1) Brian Kernighan 算法 $O(k)$; 2) 查表法 $O(1)$ 但需要预处理; 3) 分治法 $O(\log n)$ 。根据场景选择合适方法。

扩展应用

机器学习中的位运算

1. **特征哈希**: 使用位运算实现特征哈希
2. **布隆过滤器**: 使用位数组实现概率数据结构
3. **神经网络量化**: 使用低位宽表示权重

图像处理应用

1. **位图操作**: 使用位运算处理二值图像
2. **颜色空间**: 使用位运算转换颜色空间
3. **图像压缩**: 使用位运算进行无损压缩

自然语言处理

1. **词向量**: 使用位表示词汇特征
2. **布隆过滤器**: 用于拼写检查器
3. **压缩算法**: 使用位运算优化文本压缩

系统编程

1. **权限管理**: 使用位掩码表示权限
2. **内存管理**: 使用位图管理内存分配
3. **网络协议**: 使用位操作解析协议头

实战建议

学习路径

1. **基础掌握**: 熟练掌握基本位操作
2. **算法理解**: 理解位运算在算法中的应用
3. **实战练习**: 大量练习各种位运算题目
4. **工程应用**: 在实际项目中应用位运算优化

调试技巧

1. **二进制打印**: 打印变量的二进制表示
2. **位操作验证**: 逐步验证位操作结果
3. **边界测试**: 测试各种边界情况
4. **性能分析**: 使用性能分析工具优化

进阶方向

1. **SIMD 编程**: 学习使用 SIMD 指令进行并行位操作
2. **GPU 编程**: 在 GPU 上实现位运算算法
3. **分布式计算**: 研究位运算在分布式系统中的应用
4. **硬件优化**: 了解硬件层面的位运算优化

通过系统学习和实践，位集算法将成为你解决复杂问题的重要工具，在算法竞赛和工程实践中发挥巨大作用。

完整代码文件列表

基础实现

- `Code01_Bitset.java` - 位集基础实现
- `Code02_DesignBitsetTest.java` - 位集设计测试
- `Code03_SetOperation.cpp` - 集合位操作
- `Code03_SetOperation.py` - Python 集合位操作
- `Code04_BitManipulation.java` - 位操作技巧
- `Code05_SubsetGeneration.java` - 子集生成
- `Code06_GrayCode.java` - 格雷编码
- `Code07_SingleNumber.java` - 只出现一次的数字
- `Code08_BitwiseOperations.java` - 位运算应用
- `Code09_BitSetApplications.java` - 位集应用
- `Code10_BitAlgorithmOptimization.java` - 位算法优化
- `Code11_MaxLength.java` - 最大长度问题

补充实现 (Code12-22)

- `Code12_CountBits.java/.cpp/.py` - 比特位计数
- `Code13_PowerOfTwo.java/.cpp/.py` - 2 的幂判断
- `Code14_SingleNumberVariants.java/.cpp/.py` - 只出现一次的数字变种
- `Code15_BitSetApplications.java/.cpp/.py` - 位集应用扩展
- `Code16_BitMaskDP.java/.cpp/.py` - 位掩码动态规划
- `Code17_BitwiseOperations.java/.cpp/.py` - 位运算高级技巧
- `Code18_BitManipulationAdvanced.java/.cpp/.py` - 高级位操作
- `Code19_BitAlgorithmOptimizations.java/.cpp/.py` - 位算法优化
- `Code20_BitSetApplicationsAdvanced.java/.cpp/.py` - 高级位集应用
- `Code21_BitAlgorithmComprehensive.java/.cpp/.py` - 综合位算法
- `Code22_BitAlgorithmApplications.java/.cpp/.py` - 位算法应用

每个文件都包含完整的实现、测试、性能分析和工程化考量，确保代码质量和可维护性。

最后更新：2025年10月28日

作者：算法之旅学习系统

=====

[代码文件]

=====

文件：Code01_Bitset.java

=====

```
package class032;
```

```
import java.util.HashSet;
```

```
// 位图的实现
```

```
// Bitset(int size) - 创建一个能容纳 size 个元素的位集
```

```
// void add(int num) - 将数字 num 添加到位集中
```

```
// void remove(int num) - 从位集中移除数字 num
```

```
// void reverse(int num) - 反转位集中数字 num 的状态（存在变不存在，不存在变存在）
```

```
// boolean contains(int num) - 检查数字 num 是否在位集中
```

```
public class Code01_Bitset {
```

```
// 位图的实现
```

```
// 使用时 num 不要超过初始化的大小
```

```
public static class Bitset {
```

```
    // 使用整数数组来存储位信息，每个整数可以表示 32 位
```

```
    public int[] set;
```

```
    // n 个数字 : 0~n-1
```

```

// 构造函数，创建一个能容纳 n 个元素的位集
// 参数 n 表示位集能处理的最大数字范围是 0 到 n-1
public Bitset(int n) {
    // a/b 如果结果想向上取整，可以写成 : (a+b-1)/b
    // 前提是 a 和 b 都是非负数
    // 计算需要多少个整数来表示 n 个位
    // 例如: n=100，则需要(100+31)/32 = 4 个整数来表示 100 个位
    set = new int[(n + 31) / 32];
}

// 将数字 num 添加到位集中
// 通过将对应位设置为 1 来表示该数字存在
// 时间复杂度: O(1)
public void add(int num) {
    // num / 32 确定 num 在数组中的哪个整数
    // num % 32 确定 num 在该整数中的哪一位
    // 1 << (num % 32) 创建一个只有第(num % 32)位为 1 的数
    // 使用按位或操作将该位置为 1
    set[num / 32] |= 1 << (num % 32);
}

// 从位集中移除数字 num
// 通过将对应位设置为 0 来表示该数字不存在
// 时间复杂度: O(1)
public void remove(int num) {
    // ~(1 << (num % 32)) 创建一个只有第(num % 32)位为 0，其他位都为 1 的数
    // 使用按位与操作将该位置为 0
    set[num / 32] &= ~(1 << (num % 32));
}

// 反转位集中数字 num 的状态
// 如果 num 存在则移除，如果不存在则添加
// 时间复杂度: O(1)
public void reverse(int num) {
    // 1 << (num % 32) 创建一个只有第(num % 32)位为 1 的数
    // 使用按位异或操作切换该位的状态
    set[num / 32] ^= 1 << (num % 32);
}

// 检查数字 num 是否在位集中
// 时间复杂度: O(1)
// 返回值: 如果 num 在位集中返回 true，否则返回 false
public boolean contains(int num) {

```

```
// (set[num / 32] >> (num % 32)) 将第(num % 32)位移到最低位
// & 1 提取最低位
// == 1 判断该位是否为 1
return ((set[num / 32] >> (num % 32)) & 1) == 1;
}

}

// 对数据测试
// 使用自定义的Bitset 和 Java 内置的 HashSet 进行对比测试，验证实现的正确性
public static void main(String[] args) {
    int n = 1000;          // 测试数据范围: 0~999
    int testTimes = 10000; // 测试操作次数
    System.out.println("测试开始");
    // 实现的位图结构
    Bitset bitSet = new Bitset(n);
    // 直接用 HashSet 做对比测试
    HashSet<Integer> hashSet = new HashSet<>();
    System.out.println("调用阶段开始");
    for (int i = 0; i < testTimes; i++) {
        double decide = Math.random(); // 随机决定执行哪种操作
        // number -> 0 ~ n-1, 等概率得到
        int number = (int) (Math.random() * n); // 随机生成测试数字
        if (decide < 0.333) {
            // 33.3%概率执行添加操作
            bitSet.add(number);
            hashSet.add(number);
        } else if (decide < 0.666) {
            // 33.3%概率执行删除操作
            bitSet.remove(number);
            hashSet.remove(number);
        } else {
            // 33.4%概率执行反转操作
            bitSet.reverse(number);
            if (hashSet.contains(number)) {
                hashSet.remove(number);
            } else {
                hashSet.add(number);
            }
        }
    }
    System.out.println("调用阶段结束");
    System.out.println("验证阶段开始");
}
```

```
// 验证两个数据结构的结果是否一致
for (int i = 0; i < n; i++) {
    if (bitSet.contains(i) != hashSet.contains(i)) {
        System.out.println("出错了!");
    }
}
System.out.println("验证阶段结束");
System.out.println("测试结束");
}

=====

文件: Code02_DesignBitsetTest.java
```

```
=====
package class032;

// 位图的实现
// BitSet 是一种能以紧凑形式存储位的数据结构
// BitSet(int n) : 初始化 n 个位, 所有位都是 0
// void fix(int i) : 将下标 i 的位上的值更新为 1
// void unfix(int i) : 将下标 i 的位上的值更新为 0
// void flip() : 翻转所有位的值
// boolean all() : 是否所有位都是 1
// boolean one() : 是否至少有一位是 1
// int count() : 返回所有位中 1 的数量
// String toString() : 返回所有位的状态
public class Code02_DesignBitsetTest {

    // 测试链接 : https://leetcode-cn.com/problems/design-bitset/
    // 实现一个高效的位集数据结构, 支持多种位操作
    class BitSet {
        // 使用 int 数组存储位信息, 每个 int 可以存储 32 位
        private int[] set;
        // 位的总数
        private final int size;
        // 当前 0 的个数, 用于优化操作
        private int zeros;
        // 当前 1 的个数, 用于优化操作
        private int ones;
        // 是否翻转的标记, 用于优化 flip 操作
        // true 表示逻辑状态与实际存储状态相反
```

```
private boolean reverse;

// 构造函数，初始化 n 个位，所有位都是 0
// 参数 n 表示位集的大小
public Bitset(int n) {
    // 计算需要多少个 int 来存储 n 位
    // (n + 31) / 32 是向上取整的写法
    // 例如：n=100，则需要(100+31)/32 = 4 个 int 来存储 100 位
    set = new int[(n + 31) / 32];
    // 位的总数
    size = n;
    // 初始状态下所有位都是 0，所以 0 的个数为 n
    zeros = n;
    // 初始状态下所有位都是 0，所以 1 的个数为 0
    ones = 0;
    // 初始状态下没有翻转
    reverse = false;
}

// 把 i 这个数字加入到位图
// 将下标 i 的位上的值更新为 1
// 参数 i 表示要设置为 1 的位的下标
public void fix(int i) {
    // 计算 i 在数组中的位置和位偏移
    // index 确定在 set 数组中的哪个 int
    int index = i / 32;
    // bit 确定在该 int 中的哪一位
    int bit = i % 32;

    if (!reverse) {
        // 位图所有位的状态，维持原始含义
        // 0 : 不存在
        // 1 : 存在
        // 检查该位是否为 0
        if ((set[index] & (1 << bit)) == 0) {
            // 该位实际是 0，需要设置为 1
            // 0 的个数减少 1
            zeros--;
            // 1 的个数增加 1
            ones++;
            // 使用按位或操作将该位设置为 1
            set[index] |= (1 << bit);
        }
    }
}
```

```

    } else {
        // 位图所有位的状态，翻转了
        // 0 : 存在
        // 1 : 不存在
        // 检查该位是否为 1 (在逻辑上是 0)
        if ((set[index] & (1 << bit)) != 0) {
            // 该位实际是 1，但在逻辑上是 0，需要设置为 1 (即实际设置为 0)
            // 0 的个数减少 1
            zeros--;
            // 1 的个数增加 1
            ones++;
            // 使用异或操作将该位设置为 0
            set[index] ^= (1 << bit);
        }
    }
}

// 把 i 这个数字从位图中移除
// 将下标 i 的位上的值更新为 0
// 参数 i 表示要设置为 0 的位的下标
public void unfix(int i) {
    // 计算 i 在数组中的位置和位偏移
    // index 确定在 set 数组中的哪个 int
    int index = i / 32;
    // bit 确定在该 int 中的哪一位
    int bit = i % 32;

    if (!reverse) {
        // 位图所有位的状态，维持原始含义
        // 检查该位是否为 1
        if ((set[index] & (1 << bit)) != 0) {
            // 该位实际是 1，需要设置为 0
            // 1 的个数减少 1
            ones--;
            // 0 的个数增加 1
            zeros++;
            // 使用异或操作将该位设置为 0
            set[index] ^= (1 << bit);
        }
    } else {
        // 位图所有位的状态，翻转了
        // 检查该位是否为 0 (在逻辑上是 1)
        if ((set[index] & (1 << bit)) == 0) {

```

```
// 该位实际是 0，但在逻辑上是 1，需要设置为 0（即实际设置为 1）
// 1 的个数减少 1
ones--;
// 0 的个数增加 1
zeros++;
// 使用按位或操作将该位设置为 1
set[index] |= (1 << bit);
}

}

}

// 翻转所有位的值
// 使用懒标记优化，避免每次都实际翻转所有位
public void flip() {
    // 切换翻转标记
    reverse = !reverse;
    // 翻转后，0 和 1 的个数互换
    // 这是基于数学原理：0 变 1，1 变 0
    int tmp = zeros;
    zeros = ones;
    ones = tmp;
}

}

// 检查所有位是否都是 1
// 返回值：如果所有位都是 1 返回 true，否则返回 false
public boolean all() {
    // 所有位都是 1 当且仅当 1 的个数等于总位数
    return ones == size;
}

}

// 检查是否至少有一位是 1
// 返回值：如果至少有一位是 1 返回 true，否则返回 false
public boolean one() {
    // 至少有一位是 1 当且仅当 1 的个数大于 0
    return ones > 0;
}

}

// 返回所有位中 1 的数量
// 返回值：1 的数量
public int count() {
    // 直接返回维护的 1 的个数，避免重新计算
    return ones;
}
```

```

// 返回所有位的状态
// 返回值：表示所有位状态的字符串
public String toString() {
    // 使用 StringBuilder 提高字符串拼接效率
    StringBuilder builder = new StringBuilder();
    // 遍历每一位
    for (int i = 0, k = 0, number, status; i < size; k++) {
        // 获取第 k 个 int
        number = set[k];
        // 处理该 int 中的每一位
        for (int j = 0; j < 32 && i < size; j++, i++) {
            // 提取第 j 位的值
            status = (number >> j) & 1;
            // 根据是否翻转来确定实际的位值
            status ^= reverse ? 1 : 0;
            // 将位值添加到结果字符串中
            builder.append(status);
        }
    }
    return builder.toString();
}

}

```

}

}

}

文件: Code03_SetOperation.cpp

```

=====

#include <iostream>
#include <bitset>
#include <vector>
#include <cstring>
using namespace std;

// POJ 2443 Set Operation
// 题目链接: http://poj.org/problem?id=2443
// 题目大意:
// 给定 N 个集合, 第 i 个集合 S(i) 有 C(i) 个元素 (集合可以包含两个相同的元素)。
// 集合中的每个元素都用 1~10000 的正数表示。
// 查询两个给定元素 i 和 j 是否同时属于至少一个集合。

```

```
// 换句话说，确定是否存在一个数字 k (1 ≤ k ≤ N)，使得元素 i 和元素 j 都属于 S(k)。  
//  
// 输入：  
// 第一行包含一个整数 N (1 ≤ N ≤ 1000)，表示集合的数量。  
// 接下来 N 行，每行以数字 C(i) (1 ≤ C(i) ≤ 10000) 开始，然后是 C(i) 个数字，  
// 这些数字用空格分隔，给出集合中的元素（这些 C(i) 个数字不需要彼此不同）。  
// 第 N+2 行包含一个数字 Q (1 ≤ Q ≤ 200000)，表示查询的数量。  
// 然后是 Q 行。每行包含一对数字 i 和 j (1 ≤ i, j ≤ 10000, i 可以等于 j)，  
// 描述需要回答的元素。  
//  
// 输出：  
// 对于每个查询，在一行中，如果存在这样的数字 k，打印"Yes"；否则打印"No"。  
//  
// 解题思路：  
// 使用 bitset 优化的方法：  
// 1. 对于每个元素 x，我们用一个 bitset 记录它在哪些集合中出现过  
// 2. 对于查询(x, y)，我们检查 vis[x] & vis[y] 是否为 0  
// 如果不为 0，说明存在至少一个集合同时包含 x 和 y  
// 时间复杂度：O(N*C + Q) 其中 C 是集合的平均大小  
// 空间复杂度：O(10000 * N / 32) = O(312500) bit
```

```
const int MAXN = 1001;      // 集合数量最大值  
const int MAXV = 10001;     // 元素值最大值
```

```
// 使用 bitset 优化的方法  
// vis[x] 表示元素 x 在哪些集合中出现过  
// 例如：如果元素 5 在第 1、3、4 个集合中出现过，那么 vis[5] 的第 1、3、4 位为 1  
bitset<MAXN> vis[MAXV]; // vis[x] 表示元素 x 在哪些集合中出现过
```

```
// 主函数，处理输入并输出结果  
int main() {  
    // 优化输入输出速度，关闭 stdio 同步，解除 cin 与 cout 的绑定  
    ios::sync_with_stdio(false);  
    cin.tie(0);  
  
    int n; // 集合数量  
    // 读取集合数量  
    cin >> n;  
  
    // 读取每个集合  
    // 对于第 i 个集合，读取其中的所有元素，并在对应元素的 bitset 中将第 i 位置为 1  
    for (int i = 1; i ≤ n; i++) {  
        int c; // 第 i 个集合中元素的数量
```

```
// 读取第 i 个集合中元素的数量
cin >> c;
// 读取第 i 个集合中的所有元素
for (int j = 1; j <= c; j++) {
    int x; // 集合中的元素
    // 读取元素值
    cin >> x;
    // 标记元素 x 在第 i 个集合中出现过
    // set(i)方法将 bitset 中第 i 位设置为 1
    vis[x].set(i);
}
}

// 处理查询
int q; // 查询数量
// 读取查询数量
cin >> q;
// 处理每个查询
for (int i = 0; i < q; i++) {
    int x, y; // 查询的两个元素
    // 读取查询的两个元素
    cin >> x >> y;

    // 检查是否存在一个集合同时包含 x 和 y
    // 通过按位与操作检查是否有共同的集合
    // (vis[x] & vis[y]) 计算两个 bitset 的按位与结果
    // .any() 检查结果中是否有任何位为 1
    if ((vis[x] & vis[y]).any()) {
        // 存在至少一个集合同时包含 x 和 y
        cout << "Yes\n";
    } else {
        // 不存在同时包含 x 和 y 的集合
        cout << "No\n";
    }
}

return 0;
}
```

=====

文件: Code03_SetOperation.java

=====

```
package class032;

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

// POJ 2443 Set Operation
// 题目链接: http://poj.org/problem?id=2443
// 题目大意:
// 给定 N 个集合, 第 i 个集合 S(i) 有 C(i) 个元素 (集合可以包含两个相同的元素)。
// 集合中的每个元素都用 1~10000 的正数表示。
// 查询两个给定元素 i 和 j 是否同时属于至少一个集合。
// 换句话说, 确定是否存在一个数字 k ( $1 \leq k \leq N$ ), 使得元素 i 和元素 j 都属于 S(k)。
//
// 输入:
// 第一行包含一个整数 N ( $1 \leq N \leq 1000$ ), 表示集合的数量。
// 接下来 N 行, 每行以数字 C(i) ( $1 \leq C(i) \leq 10000$ ) 开始, 然后是 C(i) 个数字,
// 这些数字用空格分隔, 给出集合中的元素 (这些 C(i) 个数字不需要彼此不同)。
// 第 N+2 行包含一个数字 Q ( $1 \leq Q \leq 200000$ ), 表示查询的数量。
// 然后是 Q 行。每行包含一对数字 i 和 j ( $1 \leq i, j \leq 10000$ , i 可以等于 j),
// 描述需要回答的元素。
//
// 输出:
// 对于每个查询, 在一行中, 如果存在这样的数字 k, 打印"Yes"; 否则打印"No"。
//
// 解题思路:
// 使用 bitset 优化的方法:
// 1. 对于每个元素 x, 我们用一个 bitset 记录它在哪些集合中出现过
// 2. 对于查询(x, y), 我们检查 vis[x] & vis[y] 是否为 0
// 如果不为 0, 说明存在至少一个集合同时包含 x 和 y
// 时间复杂度: O(N*C + Q) 其中 C 是集合的平均大小
// 空间复杂度: O(10000 * N / 32) = O(312500) bit
public class Code03_SetOperation {

    // 使用 Java 自带的 BitSet 实现
    // 利用 Java 标准库中的 BitSet 类来解决问题
    static class Solution1 {
        // 主函数, 处理输入并输出结果
        public static void main(String[] args) throws IOException {
            // 使用 BufferedReader 提高输入效率
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            // 读取集合数量
            int n = Integer.parseInt(reader.readLine());
```

```
// vis[x]表示元素 x 在哪些集合中出现过
// 使用位运算，第 i 位为 1 表示元素 x 在第 i 个集合中出现过
BitSet[] vis = new BitSet[10001];
// 初始化每个元素的 BitSet
for (int i = 0; i <= 10000; i++) {
    vis[i] = new BitSet(n + 1);
}

// 读取每个集合
for (int i = 1; i <= n; i++) {
    // 分割输入行，获取元素列表
    String[] parts = reader.readLine().split(" ");
    // 获取集合中元素的数量
    int c = Integer.parseInt(parts[0]);
    // 处理集合中的每个元素
    for (int j = 1; j <= c; j++) {
        // 获取元素值
        int x = Integer.parseInt(parts[j]);
        // 标记元素 x 在第 i 个集合中出现过
        // set(i)方法将 BitSet 中第 i 位设置为 true
        vis[x].set(i);
    }
}

// 处理查询
// 读取查询数量
int q = Integer.parseInt(reader.readLine());
// 使用 StringBuilder 提高输出效率
StringBuilder result = new StringBuilder();
// 处理每个查询
for (int i = 0; i < q; i++) {
    // 分割查询行，获取两个元素
    String[] parts = reader.readLine().split(" ");
    // 获取查询的第一个元素
    int x = Integer.parseInt(parts[0]);
    // 获取查询的第二个元素
    int y = Integer.parseInt(parts[1]);

    // 检查是否存在一个集合同时包含 x 和 y
    // 克隆 vis[x]以避免修改原始数据
    BitSet intersection = (BitSet) vis[x].clone();
    // 计算 vis[x]和 vis[y]的交集
    intersection.and(vis[y]);
}
```

```
// 检查交集是否为空
if (!intersection.isEmpty()) {
    // 存在至少一个集合同时包含 x 和 y
    result.append("Yes\n");
} else {
    // 不存在同时包含 x 和 y 的集合
    result.append("No\n");
}

// 输出所有查询结果
System.out.print(result.toString());
}

// 自定义 BitSet 实现
// 通过自定义 BitSet 类来理解位运算的底层实现原理
static class Solution2 {
    // 自定义 BitSet 类，用于存储元素在哪些集合中出现过
    // 使用整数数组来模拟 BitSet 的功能
    static class CustomBitSet {
        // 使用 int 数组存储位信息，每个 int 可以存储 32 位
        private int[] bits;

        // 构造函数，初始化足够大的数组
        // 参数 n 表示需要存储的位数
        public CustomBitSet(int n) {
            // 计算需要多少个 int 来存储 n 位
            // (n + 31) / 32 是向上取整的写法
            // 例如：n=100，则需要(100+31)/32 = 4 个 int 来存储 100 位
            bits = new int[(n + 31) / 32];
        }

        // 设置第 i 位为 1
        // 参数 i 表示要设置的位索引
        public void set(int i) {
            // i / 32 确定在数组中的哪个 int
            // i % 32 确定在该 int 中的哪一位
            // 1 << (i % 32) 创建一个只有第(i % 32)位为 1 的数
            // 使用按位或操作将该位置为 1
            bits[i / 32] |= (1 << (i % 32));
        }
    }
}
```

```

// 检查第 i 位是否为 1
// 参数 i 表示要检查的位索引
// 返回值：如果第 i 位为 1 返回 true，否则返回 false
public boolean get(int i) {
    // (bits[i / 32] >> (i % 32)) 将第(i % 32)位移到最低位
    // & 1 提取最低位
    return ((bits[i / 32] >> (i % 32)) & 1) == 1;
}

// 按位与操作
// 参数 other 表示要与当前 BitSet 进行按位与操作的另一个 BitSet
// 返回值：返回一个新的 CustomBitSet，其每一位都是两个操作数对应位的按位与结果
public CustomBitSet and(CustomBitSet other) {
    // 创建结果 BitSet，大小与当前 BitSet 相同
    CustomBitSet result = new CustomBitSet(bits.length * 32);
    // 对每一位进行按位与操作
    for (int i = 0; i < bits.length; i++) {
        result.bits[i] = this.bits[i] & other.bits[i];
    }
    return result;
}

// 检查是否有任何位为 1
// 返回值：如果有任何位为 1 返回 false，如果所有位都为 0 返回 true
public boolean isEmpty() {
    // 检查每一位是否都为 0
    for (int i = 0; i < bits.length; i++) {
        if (bits[i] != 0) {
            return false;
        }
    }
    return true;
}

// 主函数，处理输入并输出结果
public static void main(String[] args) throws IOException {
    // 使用 BufferedReader 提高输入效率
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    // 读取集合数量
    int n = Integer.parseInt(reader.readLine());
}

```

```
// vis[x]表示元素 x 在哪些集合中出现过
CustomBitSet[] vis = new CustomBitSet[10001];
// 初始化每个元素的 CustomBitSet
for (int i = 0; i <= 10000; i++) {
    vis[i] = new CustomBitSet(n + 1);
}

// 读取每个集合
for (int i = 1; i <= n; i++) {
    // 分割输入行，获取元素列表
    String[] parts = reader.readLine().split(" ");
    // 获取集合中元素的数量
    int c = Integer.parseInt(parts[0]);
    // 处理集合中的每个元素
    for (int j = 1; j <= c; j++) {
        // 获取元素值
        int x = Integer.parseInt(parts[j]);
        // 标记元素 x 在第 i 个集合中出现过
        vis[x].set(i);
    }
}

// 处理查询
// 读取查询数量
int q = Integer.parseInt(reader.readLine());
// 使用 StringBuilder 提高输出效率
StringBuilder result = new StringBuilder();
// 处理每个查询
for (int i = 0; i < q; i++) {
    // 分割查询行，获取两个元素
    String[] parts = reader.readLine().split(" ");
    // 获取查询的第一个元素
    int x = Integer.parseInt(parts[0]);
    // 获取查询的第二个元素
    int y = Integer.parseInt(parts[1]);

    // 检查是否存在一个集合同时包含 x 和 y
    // 计算 vis[x] 和 vis[y] 的按位与结果
    CustomBitSet intersection = vis[x].and(vis[y]);

    // 检查按位与结果是否为空
    if (!intersection.isEmpty()) {
        // 存在至少一个集合同时包含 x 和 y
        result.append("1\n");
    } else {
        result.append("0\n");
    }
}

// 打印结果
System.out.println(result);
```

```

        result.append("Yes\n");
    } else {
        // 不存在同时包含 x 和 y 的集合
        result.append("No\n");
    }
}

// 输出所有查询结果
System.out.print(result.toString());
}

}

// 测试用例
// 提供关于该问题的说明信息
public static void main(String[] args) {
    System.out.println("POJ 2443 Set Operation 解题方案");
    System.out.println("该问题使用 bitset 优化来高效处理集合操作查询");
    System.out.println("时间复杂度: O(N*C + Q) 其中 C 是集合的平均大小");
    System.out.println("空间复杂度: O(10000 * N / 32) = O(312500) bit");
}
}

```

=====

文件: Code03_SetOperation.py

=====

```

# POJ 2443 Set Operation
# 题目链接: http://poj.org/problem?id=2443
# 题目大意:
# 给定 N 个集合, 第 i 个集合 S(i) 有 C(i) 个元素 (集合可以包含两个相同的元素)。
# 集合中的每个元素都用 1~10000 的正数表示。
# 查询两个给定元素 i 和 j 是否同时属于至少一个集合。
# 换句话说, 确定是否存在一个数字 k (1 ≤ k ≤ N), 使得元素 i 和元素 j 都属于 S(k)。
#
# 输入:
# 第一行包含一个整数 N (1 ≤ N ≤ 1000), 表示集合的数量。
# 接下来 N 行, 每行以数字 C(i) (1 ≤ C(i) ≤ 10000) 开始, 然后是 C(i) 个数字,
# 这些数字用空格分隔, 给出集合中的元素 (这些 C(i) 个数字不需要彼此不同)。
# 第 N+2 行包含一个数字 Q (1 ≤ Q ≤ 200000), 表示查询的数量。
# 然后是 Q 行。每行包含一对数字 i 和 j (1 ≤ i, j ≤ 10000, i 可以等于 j),
# 描述需要回答的元素。
#
# 输出:

```

```

# 对于每个查询，在一行中，如果存在这样的数字 k，打印"Yes"；否则打印"No"。
#
# 解题思路：
# 使用 bitset 优化的方法：
# 1. 对于每个元素 x，我们用一个 bitset 记录它在哪些集合中出现过
# 2. 对于查询(x, y)，我们检查 vis[x] & vis[y] 是否为 0
# 如果不为 0，说明存在至少一个集合同时包含 x 和 y
# 时间复杂度：O(N*C + Q) 其中 C 是集合的平均大小
# 空间复杂度：O(10000 * N / 32) = O(312500) bit

# Python 中没有内置的 bitset，但我们可以使用 set 来模拟
# 或者使用第三方库 bitarray

# 方法 1：使用 set 模拟 bitset
# 利用 Python 的 set 数据结构来模拟 bitset 的功能
def solve_with_set():
    # 读取集合数量
    n = int(input())

    # vis[x] 表示元素 x 在哪些集合中出现过
    # 使用 set 列表来存储每个元素在哪些集合中出现过
    vis = [set() for _ in range(10001)]

    # 读取每个集合
    for i in range(1, n + 1):
        # 读取一行输入并转换为整数列表
        line = list(map(int, input().split()))
        # 获取集合中元素的数量
        c = line[0]
        # 处理集合中的每个元素
        for j in range(1, c + 1):
            # 获取元素值
            x = line[j]
            # 将集合编号 i 添加到元素 x 的集合中
            # 表示元素 x 在第 i 个集合中出现过
            vis[x].add(i)

    # 处理查询
    # 读取查询数量
    q = int(input())
    # 存储查询结果
    results = []
    # 处理每个查询

```

```
for _ in range(q):
    # 读取查询的两个元素
    x, y = map(int, input().split())

    # 检查是否存在一个集合同时包含 x 和 y
    # 通过求交集检查是否有共同的集合
    # 如果 vis[x] 和 vis[y] 的交集不为空，说明存在至少一个集合同时包含 x 和 y
    if vis[x] & vis[y]: # 如果交集不为空
        results.append("Yes")
    else:
        results.append("No")

# 输出所有查询结果
for result in results:
    print(result)

# 方法 2：使用位运算模拟 bitset
# 利用 Python 整数的位运算功能来模拟 bitset
def solve_with_bitwise():
    # 读取集合数量
    n = int(input())

    # vis[x] 表示元素 x 在哪些集合中出现过，使用整数的位来表示
    # 每个整数的第 i 位为 1 表示元素 x 在第 i 个集合中出现过
    vis = [0 for _ in range(10001)]

    # 读取每个集合
    for i in range(1, n + 1):
        # 读取一行输入并转换为整数列表
        line = list(map(int, input().split()))
        # 获取集合中元素的数量
        c = line[0]
        # 处理集合中的每个元素
        for j in range(1, c + 1):
            # 获取元素值
            x = line[j]
            # 使用位运算标记元素 x 在第 i 个集合中出现过
            # 1 << i 创建一个只有第 i 位为 1 的数
            # |= 按位或操作，将第 i 位设置为 1
            vis[x] |= (1 << i)

    # 处理查询
    # 读取查询数量
```

```
q = int(input())
# 存储查询结果
results = []
# 处理每个查询
for _ in range(q):
    # 读取查询的两个元素
    x, y = map(int, input().split())

    # 检查是否存在一个集合同时包含 x 和 y
    # 通过按位与操作检查是否有共同的集合
    # vis[x] & vis[y] 计算两个整数的按位与结果
    # 如果结果不为 0, 说明存在至少一个集合同时包含 x 和 y
    if vis[x] & vis[y]: # 如果按位与结果不为 0
        results.append("Yes")
    else:
        results.append("No")

# 输出所有查询结果
for result in results:
    print(result)

# 方法 3: 使用 bitarray 库 (如果安装了的话)
# 需要先安装: pip install bitarray
```
from bitarray import bitarray

def solve_with_bitarray():
 # 读取集合数量
 n = int(input())

 # vis[x] 表示元素 x 在哪些集合中出现过
 # 使用 bitarray 列表来存储每个元素在哪些集合中出现过
 vis = [bitarray(n + 1) for _ in range(10001)]
 # 初始化所有 bitarray 为全 0
 for b in vis:
 b.setall(0)

 # 读取每个集合
 for i in range(1, n + 1):
 # 读取一行输入并转换为整数列表
 line = list(map(int, input().split()))
 # 获取集合中元素的数量
 c = line[0]
```

```
处理集合中的每个元素
for j in range(1, c + 1):
 # 获取元素值
 x = line[j]
 # 将第 i 位设置为 1, 表示元素 x 在第 i 个集合中出现过
 vis[x][i] = 1

处理查询
读取查询数量
q = int(input())
存储查询结果
results = []
处理每个查询
for _ in range(q):
 # 读取查询的两个元素
 x, y = map(int, input().split())

 # 检查是否存在一个集合同时包含 x 和 y
 # 通过按位与操作检查是否有共同的集合
 # 计算 vis[x] 和 vis[y] 的按位与结果
 intersection = vis[x] & vis[y]
 # 检查是否有任何位为 1
 if intersection.any(): # 如果有任何位为 1
 results.append("Yes")
 else:
 results.append("No")

输出所有查询结果
for result in results:
 print(result)
,,,

程序入口点
if __name__ == "__main__":
 # 选择其中一种方法来解决问题
 # solve_with_set() # 使用 set 模拟
 solve_with_bitwise() # 使用位运算模拟 bitset
 # solve_with_bitarray() # 使用 bitarray 库
```

=====

文件: Code04\_MedianSum.cpp

=====

```
#include <iostream>
#include <bitset>
#include <vector>
using namespace std;

// AtCoder AGC020 C - Median Sum
// 题目链接: https://atcoder.jp/contests/agc020/tasks/agc020_c
// 题目大意:
// 给定 N 个整数 A1, A2, ..., AN。
// 考虑 A 的所有非空子序列的和。有 $2^N - 1$ 个这样的和，这是一个奇数。
// 将这些和按非递减顺序排列为 S1, S2, ..., S_{2^N - 1}。
// 找到这个列表的中位数 S_{2^{N-1}}。
//
// 约束条件:
// 1 ≤ N ≤ 2000
// 1 ≤ Ai ≤ 2000
// 所有输入值都是整数。
//
// 输入:
// 输入以以下格式从标准输入给出:
// N
// A1 A2 ... AN
//
// 输出:
// 打印 A 的所有非空子序列的和按排序后的中位数。
//
// 解题思路:
// 这是一个经典的 bitset 优化 DP 问题。
// 1. 使用 bitset 来表示所有可能的子集和
// 2. bitset 的第 i 位为 1 表示存在一个子集的和为 i
// 3. 对于每个元素 x, 我们执行: dp |= dp << x
// 这表示将之前所有可达的和都加上 x, 同时保留原来的和
// 4. 中位数的计算有一个技巧:
// 所有子集和的总和为 sum, 那么中位数就是从 $(sum+1)/2$ 开始第一个可达的和
//
// 时间复杂度: O(N * sum / 32) 其中 sum 是所有元素的和
// 空间复杂度: O(sum) bit

// 定义最大可能的和: 2000 * 2000 = 4,000,000
const int MAX_SUM = 4000001;

// 主函数, 处理输入并输出结果
int main() {
```

```

// 优化输入输出速度，关闭 stdio 同步，解除 cin 与 cout 的绑定
ios::sync_with_stdio(false);
cin.tie(0);

int n; // 元素数量
// 读取元素数量
cin >> n;

// 存储输入的元素
vector<int> a(n);
int sum = 0; // 所有元素的总和
// 读取所有元素并计算总和
for (int i = 0; i < n; i++) {
 cin >> a[i];
 sum += a[i];
}

// 使用 bitset 优化的 DP
// dp 的第 i 位为 1 表示存在一个子集的和为 i
bitset<MAX_SUM> dp;
// 初始状态，空集的和为 0
// 将 dp 的第 0 位设置为 1
dp[0] = 1;

// 对于每个元素，更新所有可能的子集和
for (int i = 0; i < n; i++) {
 // dp |= dp << a[i]
 // 这表示既保留原来的和，又加上 a[i] 后的新和
 // dp << a[i] 将 dp 中所有为 1 的位向左移动 a[i] 位
 // dp | (dp << a[i]) 按位或操作，将原来的和与新和合并
 dp |= (dp << a[i]);
}

// 找到中位数
// 所有子集和的总数是 $2^N - 1$ ，中位数是第 $2^{(N-1)}$ 个
// 有一个数学技巧：从 $(sum+1)/2$ 开始第一个可达的和就是中位数
// 循环从 $(sum+1)/2$ 开始，找到第一个 dp[i] 为 1 的位置
for (int i = (sum + 1) / 2; ; i++) {
 // 检查 dp 的第 i 位是否为 1
 if (dp[i]) {
 // 找到中位数，输出并结束程序
 cout << i << "\n";
 break;
 }
}

```

```
 }
}

return 0;
}
```

---

文件: Code04\_MedianSum.java

---

```
package class032;

import java.util.*;
import java.io.*;
import java.math.BigInteger;

// AtCoder AGC020 C - Median Sum
// 题目链接: https://atcoder.jp/contests/agc020/tasks/agc020_c
// 题目大意:
// 给定 N 个整数 A1, A2, ..., AN。
// 考虑 A 的所有非空子序列的和。有 $2^N - 1$ 个这样的和，这是一个奇数。
// 将这些和按非递减顺序排列为 S1, S2, ..., S_{2^N - 1}。
// 找到这个列表的中位数 S_{2^{N-1}}。
//
// 约束条件:
// 1 ≤ N ≤ 2000
// 1 ≤ Ai ≤ 2000
// 所有输入值都是整数。
//
// 输入:
// 输入以以下格式从标准输入给出:
// N
// A1 A2 ... AN
//
// 输出:
// 打印 A 的所有非空子序列的和按排序后的中位数。
//
// 解题思路:
// 这是一个经典的 bitset 优化 DP 问题。
// 1. 使用 bitset 来表示所有可能的子集和
// 2. bitset 的第 i 位为 1 表示存在一个子集的和为 i
// 3. 对于每个元素 x, 我们执行: dp |= dp << x
// 这表示将之前所有可达的和都加上 x, 同时保留原来的和
```

```

// 4. 中位数的计算有一个技巧:
// 所有子集和的总和为 sum, 那么中位数就是从(sum+1)/2 开始第一个可达的和
//
// 时间复杂度: O(N * sum / 32) 其中 sum 是所有元素的和
// 空间复杂度: O(sum) bit

public class Code04_MedianSum {

 // 使用 Java 自带的 BitSet 实现
 // 利用 Java 标准库中的 BitSet 类来解决问题
 static class Solution1 {

 // 主函数, 处理输入并输出结果
 public static void main(String[] args) throws IOException {
 // 使用 BufferedReader 提高输入效率
 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
 // 读取元素数量
 int n = Integer.parseInt(reader.readLine());

 // 读取所有元素
 String[] parts = reader.readLine().split(" ");
 int[] a = new int[n];
 int sum = 0; // 所有元素的总和
 // 处理每个元素, 同时计算总和
 for (int i = 0; i < n; i++) {
 a[i] = Integer.parseInt(parts[i]);
 sum += a[i];
 }

 // 使用 bitset 优化的 DP
 // dp 的第 i 位为 1 表示存在一个子集的和为 i
 BitSet dp = new BitSet(sum + 1);
 // 初始状态, 空集的和为 0
 // set(0) 将 dp 的第 0 位设置为 true
 dp.set(0);

 // 对于每个元素, 更新所有可能的子集和
 for (int i = 0; i < n; i++) {
 // dp << a[i] 表示将所有现有的和都加上 a[i]
 // dp | (dp << a[i]) 表示既保留原来的和, 又加上 a[i] 后的新和
 // get(0, dp.length()) 获取 dp 的一个副本
 BitSet shifted = (BitSet) dp.clone();
 // 手动实现左移操作
 // 先清空高位, 再执行左移
 }
 }
 }
}

```

```

shifted.clear();
for (int j = dp.nextSetBit(0); j >= 0; j = dp.nextSetBit(j+1)) {
 shifted.set(j + a[i]);
}
// or(shifted) 将 dp 与 shifted 按位或，合并原来的和与新和
dp.or(shifted);
}

// 找到中位数
// 所有子集和的总数是 $2^N - 1$, 中位数是第 $2^{(N-1)}$ 个
// 有一个数学技巧：从 $(sum+1)/2$ 开始第一个可达的和就是中位数
// 循环从 $(sum+1)/2$ 开始，找到第一个 dp.get(i) 为 true 的位置
for (int i = (sum + 1) / 2; ; i++) {
 // 检查 dp 的第 i 位是否为 true
 if (dp.get(i)) {
 // 找到中位数，输出并结束程序
 System.out.println(i);
 break;
 }
}
}

// 自定义 BitSet 实现
// 通过自定义 BitSet 类来理解位运算的底层实现原理
static class Solution2 {
 // 自定义 BitSet 类，用于优化 DP 过程
 // 使用 long 数组来存储位信息，每个 long 可以存储 64 位，比 int 的 32 位更高效
 static class CustomBitSet {
 // 使用 long 数组存储位信息，每个 long 可以存储 64 位
 private long[] bits;

 // 构造函数，初始化足够大的数组
 // 参数 n 表示需要存储的位数
 public CustomBitSet(int n) {
 // 计算需要多少个 long 来存储 n 位
 // $(n + 63) / 64$ 是向上取整的写法
 // 例如：n=100，则需要 $(100+63)/64 = 2$ 个 long 来存储 100 位
 bits = new long[(n + 63) / 64];
 }

 // 设置第 i 位为 1
 // 参数 i 表示要设置的位索引
 }
}

```

```

public void set(int i) {
 // i / 64 确定在数组中的哪个 long
 // i % 64 确定在该 long 中的哪一位
 // 1L << (i % 64) 创建一个只有第(i % 64)位为 1 的数
 // 使用按位或操作将该位置为 1
 bits[i / 64] |= (1L << (i % 64));
}

// 检查第 i 位是否为 1
// 参数 i 表示要检查的位索引
// 返回值：如果第 i 位为 1 返回 true，否则返回 false
public boolean get(int i) {
 // (bits[i / 64] >> (i % 64)) 将第(i % 64)位移到最低位
 // & 1L 提取最低位
 // == 1L 判断该位是否为 1
 return ((bits[i / 64] >> (i % 64)) & 1L) == 1L;
}

// 按位或操作，相当于 dp |= dp << x
// 参数 other 表示要与当前 BitSet 进行操作的另一个 BitSet
// 参数 shift 表示要左移的位数
public void orShiftLeft(CustomBitSet other, int shift) {
 if (shift == 0) {
 // 如果 shift 为 0，直接按位或
 for (int i = 0; i < bits.length && i < other.bits.length; i++) {
 bits[i] |= other.bits[i];
 }
 } else {
 // 实现 dp |= other << shift
 // 计算需要移动的 long 数量
 int longShift = shift / 64;
 // 计算需要移动的位数
 int bitShift = shift % 64;

 // 从高位到低位处理，避免覆盖还未处理的数据
 for (int i = bits.length - 1; i >= 0; i--) {
 long value = 0;
 // 计算源位置
 int srcPos = i - longShift;

 // 如果源位置有效，则处理当前 long 的低位部分
 if (srcPos >= 0 && srcPos < other.bits.length) {
 value |= other.bits[srcPos] << bitShift;
 }
 bits[i] |= value;
 }
 }
}

```

```

 }

 // 如果 bitShift 大于 0 且下一个源位置有效，则处理当前 long 的高位部分
 if (bitShift > 0 && srcPos + 1 < other.bits.length && srcPos + 1 >= 0) {
 // >>> 表示无符号右移
 value |= other.bits[srcPos + 1] >>> (64 - bitShift);
 }

 // 按位或操作，将计算结果合并到当前 bits 中
 bits[i] |= value;
}

}

}

// 主函数，处理输入并输出结果
public static void main(String[] args) throws IOException {
 // 使用 BufferedReader 提高输入效率
 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
 // 读取元素数量
 int n = Integer.parseInt(reader.readLine());

 // 读取所有元素
 String[] parts = reader.readLine().split(" ");
 int[] a = new int[n];
 int sum = 0; // 所有元素的总和
 // 处理每个元素，同时计算总和
 for (int i = 0; i < n; i++) {
 a[i] = Integer.parseInt(parts[i]);
 sum += a[i];
 }

 // 使用自定义 bitset 优化的 DP
 // dp 的第 i 位为 1 表示存在一个子集的和为 i
 CustomBitSet dp = new CustomBitSet(sum + 1);
 // 初始状态，空集的和为 0
 dp.set(0);

 // 对于每个元素，更新所有可能的子集和
 for (int i = 0; i < n; i++) {
 // dp |= dp << a[i]
 // 执行按位或左移操作
 CustomBitSet temp = new CustomBitSet(sum + 1);

```

```

 for (int j = 0; j < dp.bits.length * 64; j++) {
 if (dp.get(j)) {
 temp.set(j + a[i]);
 }
 }

 for (int j = 0; j < dp.bits.length && j < temp.bits.length; j++) {
 dp.bits[j] |= temp.bits[j];
 }
 }

 // 找到中位数
 // 从(sum+1)/2 开始第一个可达的和就是中位数
 // 循环从(sum+1)/2 开始，找到第一个 dp.get(i) 为 true 的位置
 for (int i = (sum + 1) / 2; ; i++) {
 // 检查 dp 的第 i 位是否为 true
 if (dp.get(i)) {
 // 找到中位数，输出并结束程序
 System.out.println(i);
 break;
 }
 }
}

// 测试用例
// 提供关于该问题的说明信息
public static void main(String[] args) {
 System.out.println("AtCoder AGC020 C - Median Sum 解题方案");
 System.out.println("该问题使用 bitset 优化 DP 来高效计算所有子集和");
 System.out.println("时间复杂度: O(N * sum / 64) 其中 sum 是所有元素的和");
 System.out.println("空间复杂度: O(sum) bit");
 System.out.println("关键技巧: 中位数是从(sum+1)/2 开始第一个可达的和");
}
}

```

文件: Code04\_MedianSum.py

```

AtCoder AGC020 C - Median Sum
题目链接: https://atcoder.jp/contests/agc020/tasks/agc020_c
题目大意:
给定 N 个整数 A1, A2, ..., AN。

```

```
考虑 A 的所有非空子序列的和。有 $2^N - 1$ 个这样的和，这是一个奇数。
将这些和按非递减顺序排列为 S1, S2, ..., S_{2^{N-1}}。
找到这个列表的中位数 S_{2^{N-1}}。
#
约束条件：
$1 \leq N \leq 2000$
$1 \leq A_i \leq 2000$
所有输入值都是整数。
#
输入：
输入以以下格式从标准输入给出：
N
A1 A2 ... AN
#
输出：
打印 A 的所有非空子序列的和按排序后的中位数。
#
解题思路：
这是一个经典的 bitset 优化 DP 问题。
1. 使用 bitset 来表示所有可能的子集和
2. bitset 的第 i 位为 1 表示存在一个子集的和为 i
3. 对于每个元素 x，我们执行：dp |= dp << x
这表示将之前所有可达的和都加上 x，同时保留原来的和
4. 中位数的计算有一个技巧：
所有子集和的总和为 sum，那么中位数就是从 $(sum+1)/2$ 开始第一个可达的和
#
时间复杂度：O(N * sum / 32) 其中 sum 是所有元素的和
空间复杂度：O(sum) bit

Python 中没有内置的 bitset，但我们可以使用 set 来模拟
或者使用第三方库 bitarray

方法 1：使用位运算模拟 bitset
利用 Python 整数的位运算功能来模拟 bitset
def solve_with_bitwise():
 # 读取元素数量
 n = int(input())
 # 读取所有元素
 a = list(map(int, input().split()))

 # 计算所有元素的总和
 sum_a = sum(a)
```

```

使用整数的位来模拟 bitset
dp 的第 i 位为 1 表示存在一个子集的和为 i
dp = 1 # 初始状态, 空集的和为 0, 即第 0 位为 1

对于每个元素, 更新所有可能的子集和
for x in a:
 # dp |= dp << x
 # 这表示既保留原来的和, 又加上 x 后的新和
 # dp << x 将 dp 中所有为 1 的位向左移动 x 位
 # dp | (dp << x) 按位或操作, 将原来的和与新和合并
 dp |= (dp << x)

找到中位数
有一个数学技巧: 从(sum+1)/2 开始第一个可达的和就是中位数
计算目标位置
target = (sum_a + 1) // 2
循环找到第一个可达的和
while True:
 # 检查 dp 的第 target 位是否为 1
 # dp & (1 << target) 提取第 target 位
 if dp & (1 << target):
 # 找到中位数, 输出并结束程序
 print(target)
 break
 target += 1

方法 2: 使用 set 来模拟可达的和
利用 Python 的 set 数据结构来记录所有可达的和
def solve_with_set():
 # 读取元素数量
 n = int(input())
 # 读取所有元素
 a = list(map(int, input().split()))

 # 计算所有元素的总和
 sum_a = sum(a)

 # 使用 set 来记录所有可达的和
 dp = {0} # 初始状态, 空集的和为 0

 # 对于每个元素, 更新所有可能的子集和
 for x in a:
 # 将之前所有可达的和都加上 x, 同时保留原来的和

```

```

new_dp = set()
遍历当前所有可达的和
for val in dp:
 new_dp.add(val) # 保留原来的和
 new_dp.add(val + x) # 加上 x 后的新和
更新 dp 为新的可达和集合
dp = new_dp

转换为排序后的列表
sums = sorted(list(dp))

找到中位数
所有子集和的总数是 $2^N - 1$, 中位数是第 $2^{(N-1)}$ 个(从 1 开始计数)
在 0 索引中, 它是第 $2^{(N-1)} - 1$ 个元素
计算中位数的索引位置
median_index = (1 << (n - 1)) - 1 # $2^{(N-1)} - 1$
输出中位数
print(sums[median_index])

方法 3: 使用 bitarray 库 (如果安装了的话)
需要先安装: pip install bitarray
```
from bitarray import bitarray

def solve_with_bitarray():
    # 读取元素数量
    n = int(input())
    # 读取所有元素
    a = list(map(int, input().split()))

    # 计算所有元素的总和
    sum_a = sum(a)

    # 使用 bitarray 来模拟 bitset
    # dp 的第 i 位为 1 表示存在一个子集的和为 i
    dp = bitarray(sum_a + 1)
    # 初始化为全 0
    dp.setall(0)
    # 初始状态, 空集的和为 0
    dp[0] = 1

    # 对于每个元素, 更新所有可能的子集和
    for x in a:
```
```

```

dp |= dp << x
复制 dp
shifted = dp.copy()
左移 x 位
shifted <<= x
按位或操作
dp |= shifted

找到中位数
从(sum+1)/2 开始第一个可达的和就是中位数
循环找到第一个可达的和
for i in range((sum_a + 1) // 2, sum_a + 1):
 if dp[i]:
 print(i)
 break
,
,
,
程序入口点
if __name__ == "__main__":
 # 选择其中一种方法来解决问题
 solve_with_bitwise() # 使用位运算模拟 bitset (推荐, 效率最高)
 # solve_with_set() # 使用 set 模拟 (容易理解但效率较低)
 # solve_with_bitarray() # 使用 bitarray 库 (需要额外安装)

```

=====

文件: Code05\_DesignBitset.cpp

=====

```

#include <iostream>
#include <vector>
#include <string>
#include <bitset>

// LeetCode 2166. Design Bitset
// 题目链接: https://leetcode.com/problems/design-bitset/
// 题目大意:
// 实现一个 Bitset 类, 支持以下操作:
// 1. Bitset(int size): 用 size 个位初始化 Bitset, 所有位都是 0
// 2. void fix(int idx): 将下标为 idx 的位更新为 1
// 3. void unfix(int idx): 将下标为 idx 的位更新为 0
// 4. void flip(): 翻转所有位的值
// 5. boolean all(): 检查所有位是否都是 1
// 6. boolean one(): 检查是否至少有一位是 1

```

```

// 7. int count(): 返回所有位中 1 的数量
// 8. String toString(): 返回所有位的状态

// 解题思路:
// 1. 使用 vector<unsigned int> 来存储位信息, 每个 unsigned int 可以存储 32 位
// 2. 使用懒标记优化 flip 操作, 避免每次都实际翻转所有位
// 3. 维护实际的 1 的个数, 避免每次 count 都重新计算
// 时间复杂度分析:
// - fix, unfix: O(1)
// - flip: O(1)
// - all, one, count: O(1)
// - toString: O(size)
// 空间复杂度: O(size/32)

using namespace std;

// Bitset 类定义
// 实现一个高效的位集数据结构, 支持多种位操作
class Bitset {
private:
 // 存储位信息的数组, 每个 unsigned int 可以存储 32 位
 vector<unsigned int> bits;
 // 位的总数
 int size;
 // 当前 1 的个数, 用于优化 count 操作
 int ones;
 // 是否翻转的标记, 用于优化 flip 操作
 // true 表示逻辑状态与实际存储状态相反
 bool flipped;

public:
 // 构造函数, 初始化 size 个位, 所有位都是 0
 // 参数 size 表示位集的大小
 Bitset(int size) {
 // 计算需要多少个 unsigned int 来存储 size 位
 // (size + 31) / 32 是向上取整的写法
 // 例如: size=100, 则需要(100+31)/32 = 4 个 unsigned int 来存储 100 位
 this->bits = vector<unsigned int>((size + 31) / 32, 0);
 this->size = size;
 // 初始状态下所有位都是 0, 所以 1 的个数为 0
 this->ones = 0;
 // 初始状态下没有翻转
 this->flipped = false;
 }
}

```

```

}

// 将下标为 idx 的位更新为 1
// 参数 idx 表示要设置为 1 的位的下标
void fix(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 unsigned int
 int arrayIdx = idx / 32;
 // bitIdx 确定在该 unsigned int 中的哪一位
 int bitIdx = idx % 32;

 // 如果当前状态(考虑翻转)下该位是 0, 则设置为 1
 if (flipped) {
 // 如果翻转了, 实际的 1 在 bits 中是 0
 // 检查该位是否为 0
 if ((bits[arrayIdx] & (1U << bitIdx)) != 0) {
 // 该位实际是 1, 但在逻辑上是 0, 需要设置为 1 (即实际设置为 0)
 // 使用异或操作将该位设置为 0
 bits[arrayIdx] ^= (1U << bitIdx);
 // 1 的个数增加 1
 ones++;
 }
 } else {
 // 如果没有翻转, 实际的 1 在 bits 中是 1
 // 检查该位是否为 0
 if ((bits[arrayIdx] & (1U << bitIdx)) == 0) {
 // 该位实际是 0, 需要设置为 1
 // 使用按位或操作将该位设置为 1
 bits[arrayIdx] |= (1U << bitIdx);
 // 1 的个数增加 1
 ones++;
 }
 }
}

// 将下标为 idx 的位更新为 0
// 参数 idx 表示要设置为 0 的位的下标
void unfix(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 unsigned int
 int arrayIdx = idx / 32;
 // bitIdx 确定在该 unsigned int 中的哪一位
 int bitIdx = idx % 32;
}

```

```

// 如果当前状态(考虑翻转)下该位是 1, 则设置为 0
if (flipped) {
 // 如果翻转了, 实际的 0 在 bits 中是 1
 // 检查该位是否为 1
 if ((bits[arrayIdx] & (1U << bitIdx)) == 0) {
 // 该位实际是 0, 但在逻辑上是 1, 需要设置为 0 (即实际设置为 1)
 // 使用按位或操作将该位设置为 1
 bits[arrayIdx] |= (1U << bitIdx);
 // 1 的个数减少 1
 ones--;
 }
} else {
 // 如果没有翻转, 实际的 0 在 bits 中是 0
 // 检查该位是否为 1
 if ((bits[arrayIdx] & (1U << bitIdx)) != 0) {
 // 该位实际是 1, 需要设置为 0
 // 使用异或操作将该位设置为 0
 bits[arrayIdx] ^= (1U << bitIdx);
 // 1 的个数减少 1
 ones--;
 }
}
}

// 翻转所有位的值
// 使用懒标记优化, 避免每次都实际翻转所有位
void flip() {
 // 切换翻转标记
 flipped = !flipped;
 // 翻转后, 1 的个数变为总位数减去原来的 1 的个数
 // 这是基于数学原理: 0 变 1, 1 变 0, 所以 1 的个数变为 size-ones
 ones = size - ones;
}

// 检查所有位是否都是 1
// 返回值: 如果所有位都是 1 返回 true, 否则返回 false
bool all() {
 // 所有位都是 1 当且仅当 1 的个数等于总位数
 return ones == size;
}

// 检查是否至少有一位是 1

```

```
// 返回值：如果至少有一位是 1 返回 true，否则返回 false
bool one() {
 // 至少有一位是 1 当且仅当 1 的个数大于 0
 return ones > 0;
}

// 返回所有位中 1 的数量
// 返回值：1 的数量
int count() {
 // 直接返回维护的 1 的个数，避免重新计算
 return ones;
}

// 返回所有位的状态
// 返回值：表示所有位状态的字符串
string toString() {
 // 存储结果字符串
 string result;
 // 遍历每一位
 for (int i = 0; i < size; i++) {
 // 计算第 i 位在数组中的位置和位偏移
 int arrayIdx = i / 32;
 int bitIdx = i % 32;

 // 根据是否翻转来确定实际的位值
 int bitValue;
 if (flipped) {
 // 如果翻转了，实际的 1 在 bits 中是 0
 // 检查 bits 中该位是否为 1
 bitValue = ((bits[arrayIdx] & (1U << bitIdx)) != 0) ? 0 : 1;
 } else {
 // 如果没有翻转，实际的 1 在 bits 中是 1
 // 检查 bits 中该位是否为 1
 bitValue = ((bits[arrayIdx] & (1U << bitIdx)) != 0) ? 1 : 0;
 }
 // 将位值添加到结果字符串中
 result += (bitValue ? '1' : '0');
 }
 return result;
};

// 测试用例
```

```
// 验证 Bitset 类的正确性
int main() {
 cout << "LeetCode 2166. Design Bitset 解题测试" << endl;

 // 创建一个 5 位的 Bitset
 Bitset bs(5);

 // 初始状态: "00000"
 cout << "Initial: " << bs.toString() << endl; // 应该输出 "00000"

 // fix(3) -> "00010"
 bs.fix(3);
 cout << "After fix(3): " << bs.toString() << endl; // 应该输出 "00010"

 // fix(1) -> "01010"
 bs.fix(1);
 cout << "After fix(1): " << bs.toString() << endl; // 应该输出 "01010"

 // flip() -> "10101"
 bs.flip();
 cout << "After flip(): " << bs.toString() << endl; // 应该输出 "10101"

 // all() -> false
 cout << "all(): " << (bs.all() ? "true" : "false") << endl; // 应该输出 false

 // unfix(0) -> "00101"
 bs.unfix(0);
 cout << "After unfix(0): " << bs.toString() << endl; // 应该输出 "00101"

 // flip() -> "11010"
 bs.flip();
 cout << "After flip(): " << bs.toString() << endl; // 应该输出 "11010"

 // one() -> true
 cout << "one(): " << (bs.one() ? "true" : "false") << endl; // 应该输出 true

 // fix(3) -> "11010" (已经是 1 了, 无变化)
 bs.fix(3);
 cout << "After fix(3): " << bs.toString() << endl; // 应该输出 "11010"

 // count() -> 4
 cout << "count(): " << bs.count() << endl; // 应该输出 4
```

```
// toString() -> "11010"
cout << "toString(): " << bs.toString() << endl; // 应该输出 "11010"

return 0;
}
```

---

文件: Code05\_DesignBitset.java

---

```
package class032;

import java.util.*;

// LeetCode 2166. Design Bitset
// 题目链接: https://leetcode.com/problems/design-bitset/
// 题目大意:
// 实现一个 Bitset 类, 支持以下操作:
// 1. Bitset(int size): 用 size 个位初始化 Bitset, 所有位都是 0
// 2. void fix(int idx): 将下标为 idx 的位更新为 1
// 3. void unfix(int idx): 将下标为 idx 的位更新为 0
// 4. void flip(): 翻转所有位的值
// 5. boolean all(): 检查所有位是否都是 1
// 6. boolean one(): 检查是否至少有一位是 1
// 7. int count(): 返回所有位中 1 的数量
// 8. String toString(): 返回所有位的状态
```

```
// 解题思路:
// 1. 使用 int 数组来存储位信息, 每个 int 可以存储 32 位
// 2. 使用懒标记优化 flip 操作, 避免每次都实际翻转所有位
// 3. 维护实际的 1 的个数, 避免每次 count 都重新计算
// 时间复杂度分析:
// - fix, unfix: O(1)
// - flip: O(1)
// - all, one, count: O(1)
// - toString: O(size)
// 空间复杂度: O(size/32)
```

```
public class Code05_DesignBitset {

 // Design Bitset 实现类
 // 实现一个高效的位集数据结构, 支持多种位操作
 static class Bitset {
```

```
// 存储位信息的数组，每个 int 可以存储 32 位
private int[] bits;
// 位的总数
private int size;
// 当前 1 的个数，用于优化 count 操作
private int ones;
// 是否翻转的标记，用于优化 flip 操作
// true 表示逻辑状态与实际存储状态相反
private boolean flipped;

// 构造函数，初始化 size 个位，所有位都是 0
// 参数 size 表示位集的大小
public Bitset(int size) {
 // 计算需要多少个 int 来存储 size 位
 // (size + 31) / 32 是向上取整的写法
 // 例如：size=100，则需要(100+31)/32 = 4 个 int 来存储 100 位
 this.bits = new int[(size + 31) / 32];
 this.size = size;
 // 初始状态下所有位都是 0，所以 1 的个数为 0
 this.ones = 0;
 // 初始状态下没有翻转
 this.flipped = false;
}

// 将下标为 idx 的位更新为 1
// 参数 idx 表示要设置为 1 的位的下标
public void fix(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 int
 int arrayIdx = idx / 32;
 // bitIdx 确定在该 int 中的哪一位
 int bitIdx = idx % 32;

 // 如果当前状态(考虑翻转)下该位是 0，则设置为 1
 if (flipped) {
 // 如果翻转了，实际的 1 在 bits 中是 0
 // 检查该位是否为 1(在逻辑上是 0)
 if ((bits[arrayIdx] & (1 << bitIdx)) != 0) {
 // 该位实际是 1，但在逻辑上是 0，需要设置为 1(即实际设置为 0)
 // 使用异或操作将该位设置为 0
 bits[arrayIdx] ^= (1 << bitIdx);
 // 1 的个数增加 1
 ones++;
 }
 }
}
```

```

 }

 } else {
 // 如果没有翻转，实际的 1 在 bits 中是 1
 // 检查该位是否为 0
 if ((bits[arrayIdx] & (1 << bitIdx)) == 0) {
 // 该位实际是 0，需要设置为 1
 // 使用按位或操作将该位设置为 1
 bits[arrayIdx] |= (1 << bitIdx);
 // 1 的个数增加 1
 ones++;
 }
 }
}

```

```

// 将下标为 idx 的位更新为 0
// 参数 idx 表示要设置为 0 的位的下标
public void unfix(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 int
 int arrayIdx = idx / 32;
 // bitIdx 确定在该 int 中的哪一位
 int bitIdx = idx % 32;

 // 如果当前状态(考虑翻转)下该位是 1，则设置为 0
 if (flipped) {
 // 如果翻转了，实际的 0 在 bits 中是 1
 // 检查该位是否为 0(在逻辑上是 1)
 if ((bits[arrayIdx] & (1 << bitIdx)) == 0) {
 // 该位实际是 0，但在逻辑上是 1，需要设置为 0(即实际设置为 1)
 // 使用按位或操作将该位设置为 1
 bits[arrayIdx] |= (1 << bitIdx);
 // 1 的个数减少 1
 ones--;
 }
 } else {
 // 如果没有翻转，实际的 0 在 bits 中是 0
 // 检查该位是否为 1
 if ((bits[arrayIdx] & (1 << bitIdx)) != 0) {
 // 该位实际是 1，需要设置为 0
 // 使用异或操作将该位设置为 0
 bits[arrayIdx] ^= (1 << bitIdx);
 // 1 的个数减少 1
 ones--;
 }
 }
}

```

```
 }
 }
}

// 翻转所有位的值
// 使用懒标记优化，避免每次都实际翻转所有位
public void flip() {
 // 切换翻转标记
 flipped = !flipped;
 // 翻转后，1 的个数变为总位数减去原来的 1 的个数
 // 这是基于数学原理：0 变 1，1 变 0，所以 1 的个数变为 size-ones
 ones = size - ones;
}

// 检查所有位是否都是 1
// 返回值：如果所有位都是 1 返回 true，否则返回 false
public boolean all() {
 // 所有位都是 1 当且仅当 1 的个数等于总位数
 return ones == size;
}

// 检查是否至少有一位是 1
// 返回值：如果至少有一位是 1 返回 true，否则返回 false
public boolean one() {
 // 至少有一位是 1 当且仅当 1 的个数大于 0
 return ones > 0;
}

// 返回所有位中 1 的数量
// 返回值：1 的数量
public int count() {
 // 直接返回维护的 1 的个数，避免重新计算
 return ones;
}

// 返回所有位的状态
// 返回值：表示所有位状态的字符串
public String toString() {
 // 使用 StringBuilder 提高字符串拼接效率
 StringBuilder sb = new StringBuilder();
 // 遍历每一位
 for (int i = 0; i < size; i++) {
 // 计算第 i 位在数组中的位置和位偏移
 }
}
```

```

 int arrayIdx = i / 32;
 int bitIdx = i % 32;

 // 根据是否翻转来确定实际的位值
 int bitValue;
 if (flipped) {
 // 如果翻转了，实际的 1 在 bits 中是 0
 // 检查 bits 中该位是否为 1
 bitValue = ((bits[arrayIdx] & (1 << bitIdx)) != 0) ? 0 : 1;
 } else {
 // 如果没有翻转，实际的 1 在 bits 中是 1
 // 检查 bits 中该位是否为 1
 bitValue = ((bits[arrayIdx] & (1 << bitIdx)) != 0) ? 1 : 0;
 }

 // 将位值添加到结果字符串中
 sb.append(bitValue);
 }

 return sb.toString();
}

}

// 测试用例
// 验证 Bitset 类的正确性
public static void main(String[] args) {
 System.out.println("LeetCode 2166. Design Bitset 解题测试");

 // 创建一个 5 位的 Bitset
 Bitset bs = new Bitset(5);

 // 初始状态: "00000"
 System.out.println("Initial: " + bs.toString()); // 应该输出 "00000"

 // fix(3) -> "00010"
 bs.fix(3);
 System.out.println("After fix(3): " + bs.toString()); // 应该输出 "00010"

 // fix(1) -> "01010"
 bs.fix(1);
 System.out.println("After fix(1): " + bs.toString()); // 应该输出 "01010"

 // flip() -> "10101"
 bs.flip();
 System.out.println("After flip(): " + bs.toString()); // 应该输出 "10101"
}

```

```

// all() -> false
System.out.println("all(): " + bs.all()); // 应该输出 false

// unfix(0) -> "00101"
bs.unfix(0);
System.out.println("After unfix(0): " + bs.toString()); // 应该输出 "00101"

// flip() -> "11010"
bs.flip();
System.out.println("After flip(): " + bs.toString()); // 应该输出 "11010"

// one() -> true
System.out.println("one(): " + bs.one()); // 应该输出 true

// fix(3) -> "11010" (已经是 1 了, 无变化)
bs.fix(3);
System.out.println("After fix(3): " + bs.toString()); // 应该输出 "11010"

// count() -> 4
System.out.println("count(): " + bs.count()); // 应该输出 4

// toString() -> "11010"
System.out.println("toString(): " + bs.toString()); // 应该输出 "11010"
}

}

```

=====

文件: Code05\_DesignBitset.py

=====

```

LeetCode 2166. Design Bitset
题目链接: https://leetcode.com/problems/design-bitset/
题目大意:
实现一个 Bitset 类, 支持以下操作:
1. Bitset(int size): 用 size 个位初始化 Bitset, 所有位都是 0
2. void fix(int idx): 将下标为 idx 的位更新为 1
3. void unfix(int idx): 将下标为 idx 的位更新为 0
4. void flip(): 翻转所有位的值
5. boolean all(): 检查所有位是否都是 1
6. boolean one(): 检查是否至少有一位是 1
7. int count(): 返回所有位中 1 的数量
8. String toString(): 返回所有位的状态

```

```
解题思路:
1. 使用整数的位来模拟 bitset
2. 使用懒标记优化 flip 操作，避免每次都实际翻转所有位
3. 维护实际的 1 的个数，避免每次 count 都重新计算
时间复杂度分析：
- fix, unfix: O(1)
- flip: O(1)
- all, one, count: O(1)
- toString: O(size)
空间复杂度: O(size/32)
```

```
class Bitset:
 def __init__(self, size):
 """
 构造函数，初始化 size 个位，所有位都是 0
 :param size: 位的总数
 """

 # 计算需要多少个整数来存储 size 位
 # (size + 31) // 32 是向上取整的写法
 # 例如：size=100，则需要(100+31)//32 = 4 个整数来存储 100 位
 self.bits = [0] * ((size + 31) // 32)
 # 位的总数
 self.size = size
 # 当前 1 的个数，用于优化 count 操作
 self.ones = 0
 # 是否翻转的标记，用于优化 flip 操作
 # True 表示逻辑状态与实际存储状态相反
 self.flipped = False

 def fix(self, idx):
 """
 将下标为 idx 的位更新为 1
 :param idx: 位的下标
 """

 # 计算 idx 在数组中的位置和位偏移
 # array_idx 确定在 bits 数组中的哪个整数
 array_idx = idx // 32
 # bit_idx 确定在该整数中的哪一位
 bit_idx = idx % 32

 # 如果当前状态(考虑翻转)下该位是 0，则设置为 1
 if self.flipped:
```

```

如果翻转了，实际的 1 在 bits 中是 0
检查该位是否为 1（在逻辑上是 0）
if (self.bits[array_idx] & (1 << bit_idx)) != 0:
 # 该位实际是 1，但在逻辑上是 0，需要设置为 1（即实际设置为 0）
 # 使用异或操作将该位设置为 0
 self.bits[array_idx] ^= (1 << bit_idx)
 # 1 的个数增加 1
 self.ones += 1

else:
 # 如果没有翻转，实际的 1 在 bits 中是 1
 # 检查该位是否为 0
 if (self.bits[array_idx] & (1 << bit_idx)) == 0:
 # 该位实际是 0，需要设置为 1
 # 使用按位或操作将该位设置为 1
 self.bits[array_idx] |= (1 << bit_idx)
 # 1 的个数增加 1
 self.ones += 1

def unfix(self, idx):
 """
 将下标为 idx 的位更新为 0
 :param idx: 位的下标
 """

 # 计算 idx 在数组中的位置和位偏移
 # array_idx 确定在 bits 数组中的哪个整数
 array_idx = idx // 32
 # bit_idx 确定在该整数中的哪一位
 bit_idx = idx % 32

 # 如果当前状态(考虑翻转)下该位是 1，则设置为 0
 if self.flipped:
 # 如果翻转了，实际的 0 在 bits 中是 1
 # 检查该位是否为 0（在逻辑上是 1）
 if (self.bits[array_idx] & (1 << bit_idx)) == 0:
 # 该位实际是 0，但在逻辑上是 1，需要设置为 0（即实际设置为 1）
 # 使用按位或操作将该位设置为 1
 self.bits[array_idx] |= (1 << bit_idx)
 # 1 的个数减少 1
 self.ones -= 1

 else:
 # 如果没有翻转，实际的 0 在 bits 中是 0
 # 检查该位是否为 1
 if (self.bits[array_idx] & (1 << bit_idx)) != 0:

```

```
该位实际是 1，需要设置为 0
使用异或操作将该位设置为 0
self.bits[array_idx] ^= (1 << bit_idx)
1 的个数减少 1
self.ones -= 1

def flip(self):
 """翻转所有位的值"""
 # 切换翻转标记
 self.flipped = not self.flipped
 # 翻转后，1 的个数变为总位数减去原来的 1 的个数
 # 这是基于数学原理：0 变 1，1 变 0，所以 1 的个数变为 size-ones
 self.ones = self.size - self.ones

def all(self):
 """
 检查所有位是否都是 1
 :return: 如果所有位都是 1 返回 True，否则返回 False
 """
 # 所有位都是 1 当且仅当 1 的个数等于总位数
 return self.ones == self.size

def one(self):
 """
 检查是否至少有一位是 1
 :return: 如果至少有一位是 1 返回 True，否则返回 False
 """
 # 至少有一位是 1 当且仅当 1 的个数大于 0
 return self.ones > 0

def count(self):
 """
 返回所有位中 1 的数量
 :return: 1 的数量
 """
 # 直接返回维护的 1 的个数，避免重新计算
 return self.ones

def toString(self):
 """
 返回所有位的状态
 :return: 表示所有位状态的字符串
 """

```

```
存储结果列表
result = []
遍历每一位
for i in range(self.size):
 # 计算第 i 位在数组中的位置和位偏移
 array_idx = i // 32
 bit_idx = i % 32

 # 根据是否翻转来确定实际的位值
 if self.flipped:
 # 如果翻转了，实际的 1 在 bits 中是 0
 # 检查 bits 中该位是否为 1
 bit_value = 0 if (self.bits[array_idx] & (1 << bit_idx)) != 0 else 1
 else:
 # 如果没有翻转，实际的 1 在 bits 中是 1
 # 检查 bits 中该位是否为 1
 bit_value = 1 if (self.bits[array_idx] & (1 << bit_idx)) != 0 else 0

 # 将位值添加到结果列表中
 result.append(str(bit_value))

将结果列表连接成字符串并返回
return ''.join(result)

测试用例
验证 Bitset 类的正确性
if __name__ == "__main__":
 print("LeetCode 2166. Design Bitset 解题测试")

 # 创建一个 5 位的 Bitset
 bs = Bitset(5)

 # 初始状态: "00000"
 print("Initial:", bs.toString()) # 应该输出 "00000"

 # fix(3) -> "00010"
 bs.fix(3)
 print("After fix(3):", bs.toString()) # 应该输出 "00010"

 # fix(1) -> "01010"
 bs.fix(1)
 print("After fix(1):", bs.toString()) # 应该输出 "01010"
```

```

flip() -> "10101"
bs.flip()
print("After flip():", bs.toString()) # 应该输出 "10101"

all() -> false
print("all()", bs.all()) # 应该输出 False

unfix(0) -> "00101"
bs.unfix(0)
print("After unfix(0):", bs.toString()) # 应该输出 "00101"

flip() -> "11010"
bs.flip()
print("After flip():", bs.toString()) # 应该输出 "11010"

one() -> true
print("one()", bs.one()) # 应该输出 True

fix(3) -> "11010" (已经是 1 了, 无变化)
bs.fix(3)
print("After fix(3):", bs.toString()) # 应该输出 "11010"

count() -> 4
print("count()", bs.count()) # 应该输出 4

toString() -> "11010"
print("toString()", bs.toString()) # 应该输出 "11010"
=====
```

文件: Code06\_HackerRankJavaBitSet.cpp

```
=====
```

```

#include <iostream>
#include <vector>
#include <string>
#include <bitset>

// HackerRank Java BitSet
// 题目链接: https://www.hackerrank.com/challenges/java-bitset/problem
// 题目大意:
// 给定两个 BitSet, 大小为 n, 初始时所有位都为 0
// 执行一系列操作, 每次操作后打印两个 BitSet 中 1 的个数
```

```
// 操作包括:
// AND 1 2: 将 BitSet1 与 BitSet2 进行按位与操作, 结果存储在 BitSet1 中
// OR 1 2: 将 BitSet1 与 BitSet2 进行按位或操作, 结果存储在 BitSet1 中
// XOR 1 2: 将 BitSet1 与 BitSet2 进行按位异或操作, 结果存储在 BitSet1 中
// FLIP 1 2: 将 BitSet1 中下标为 2 的位翻转
// SET 1 2: 将 BitSet1 中下标为 2 的位设置为 1

// 解题思路:
// 1. 使用 std::bitset 来模拟 Java 的 BitSet
// 2. 根据操作类型执行相应的位运算操作
// 3. 每次操作后手动计算并打印两个 BitSet 中 1 的个数
// 时间复杂度分析:
// - AND, OR, XOR: O(n/32)
// - FLIP, SET: O(1)
// - count(): O(n/32)
// 空间复杂度: O(n)
```

```
using namespace std;

// 自定义 BitSet 类, 模拟 Java 的 BitSet 功能
// 实现一个高效的位集数据结构, 支持多种位操作
class BitSet {
private:
 // 使用 vector 存储位信息, 每个 unsigned int 可以存储 32 位
 vector<unsigned int> bits;
 // BitSet 的大小
 int size;

public:
 // 构造函数
 // 参数 n 表示 BitSet 的大小
 BitSet(int n) {
 size = n;
 // 计算需要多少个 unsigned int 来存储 n 位
 // (n + 31) / 32 是向上取整的写法
 // 例如: n=100, 则需要(100+31)/32 = 4 个 unsigned int 来存储 100 位
 bits = vector<unsigned int>((n + 31) / 32, 0);
 }

 // 按位与操作
 // 参数 other 表示要与当前 BitSet 进行按位与操作的另一个 BitSet
 void andOp(const BitSet& other) {
 // 对每一位所在的 unsigned int 进行按位与操作
```

```

 for (int i = 0; i < bits.size(); i++) {
 bits[i] &= other.bits[i];
 }
}

// 按位或操作
// 参数 other 表示要与当前 BitSet 进行按位或操作的另一个 BitSet
void orOp(const BitSet& other) {
 // 对每一位所在的 unsigned int 进行按位或操作
 for (int i = 0; i < bits.size(); i++) {
 bits[i] |= other.bits[i];
 }
}

// 按位异或操作
// 参数 other 表示要与当前 BitSet 进行按位异或操作的另一个 BitSet
void xorOp(const BitSet& other) {
 // 对每一位所在的 unsigned int 进行按位异或操作
 for (int i = 0; i < bits.size(); i++) {
 bits[i] ^= other.bits[i];
 }
}

// 翻转指定位置的位
// 参数 idx 表示要翻转的位的下标
void flip(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 unsigned int
 int arrayIdx = idx / 32;
 // bitIdx 确定在该 unsigned int 中的哪一位
 int bitIdx = idx % 32;
 // 使用异或操作翻转指定位
 // 1U << bitIdx 创建一个只有第 bitIdx 位为 1 的数
 // ^= 异或操作，实现翻转效果
 bits[arrayIdx] ^= (1U << bitIdx);
}

// 设置指定位置的位为 1
// 参数 idx 表示要设置为 1 的位的下标
void set(int idx) {
 // 计算 idx 在数组中的位置和位偏移
 // arrayIdx 确定在 bits 数组中的哪个 unsigned int
 int arrayIdx = idx / 32;

```

```
// bitIdx 确定在该 unsigned int 中的哪一位
int bitIdx = idx % 32;
// 使用按位或操作将指定位设置为 1
// 1U << bitIdx 创建一个只有第 bitIdx 位为 1 的数
// |= 按位或操作，将指定位设置为 1
bits[arrayIdx] |= (1U << bitIdx);

}

// 计算 1 的个数
// 返回值：1 的个数
int count() const {
 int result = 0;
 // 遍历每一位所在的 unsigned int
 for (int i = 0; i < bits.size(); i++) {
 // 使用内置函数计算一个 unsigned int 中 1 的个数
 // __builtin_popcount 是 GCC 内置函数，用于计算 32 位整数中 1 的个数
 result += __builtin_popcount(bits[i]);
 }
 return result;
}

};

// 主函数，处理输入并输出结果
int main() {
 // 优化输入输出速度，关闭 stdio 同步，解除 cin 与 cout 的绑定
 ios::sync_with_stdio(false);
 cin.tie(0);

 int n, m;
 // 读取 n 和 m
 // n 表示 BitSet 的大小，m 表示操作的数量
 cin >> n >> m;

 // 初始化两个 BitSet
 // 创建一个 BitSet 向量，索引 0 不使用，1 和 2 分别对应题目中的 BitSet1 和 BitSet2
 vector<BitSet> bitSets(3);
 // 初始化 BitSet1，大小为 n，初始时所有位都为 0
 bitSets[1] = BitSet(n);
 // 初始化 BitSet2，大小为 n，初始时所有位都为 0
 bitSets[2] = BitSet(n);

 // 执行 m 次操作
 // 循环处理每个操作
```

```

for (int i = 0; i < m; i++) {
 // 读取操作指令
 string operation;
 int set1, set2;
 cin >> operation >> set1 >> set2;

 // 根据操作类型执行相应的操作
 if (operation == "AND") {
 // 将 BitSet[set1]与 BitSet[set2]进行按位与操作，结果存储在 BitSet[set1]中
 // 按位与操作：两个位都为 1 时结果才为 1，否则为 0
 bitSets[set1].andOp(bitSets[set2]);
 } else if (operation == "OR") {
 // 将 BitSet[set1]与 BitSet[set2]进行按位或操作，结果存储在 BitSet[set1]中
 // 按位或操作：两个位中至少有一个为 1 时结果为 1，否则为 0
 bitSets[set1].orOp(bitSets[set2]);
 } else if (operation == "XOR") {
 // 将 BitSet[set1]与 BitSet[set2]进行按位异或操作，结果存储在 BitSet[set1]中
 // 按位异或操作：两个位不同时结果为 1，相同时为 0
 bitSets[set1].xorOp(bitSets[set2]);
 } else if (operation == "FLIP") {
 // 将 BitSet[set1]中下标为 set2 的位翻转
 // 翻转操作：0 变 1，1 变 0
 bitSets[set1].flip(set2);
 } else if (operation == "SET") {
 // 将 BitSet[set1]中下标为 set2 的位设置为 1
 // 设置操作：将指定位置为 1
 bitSets[set1].set(set2);
 }
}

// 打印两个 BitSet 中 1 的个数
// 每次操作后都要输出两个 BitSet 中 1 的个数
cout << bitSets[1].count() << " " << bitSets[2].count() << "\n";
}
}

return 0;
}

```

文件: Code06\_HackerRankJavaBitSet.java

```

=====
package class032;
=====
```

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

// HackerRank Java BitSet
// 题目链接: https://www.hackerrank.com/challenges/java-bitset/problem
// 题目大意:
// 给定两个 BitSet，大小为 n，初始时所有位都为 0
// 执行一系列操作，每次操作后打印两个 BitSet 中 1 的个数

// 操作包括:
// AND 1 2: 将 BitSet1 与 BitSet2 进行按位与操作，结果存储在 BitSet1 中
// OR 1 2: 将 BitSet1 与 BitSet2 进行按位或操作，结果存储在 BitSet1 中
// XOR 1 2: 将 BitSet1 与 BitSet2 进行按位异或操作，结果存储在 BitSet1 中
// FLIP 1 2: 将 BitSet1 中下标为 2 的位翻转
// SET 1 2: 将 BitSet1 中下标为 2 的位设置为 1

// 解题思路:
// 1. 使用 Java 内置的 BitSet 类
// 2. 根据操作类型执行相应的 BitSet 方法
// 3. 每次操作后打印两个 BitSet 中 1 的个数
// 时间复杂度分析:
// - AND, OR, XOR: O(n/64)
// - FLIP, SET: O(1)
// - count(): O(n/64)
// 空间复杂度: O(n)

public class Code06_HackerRankJavaBitSet {

 // 主函数，处理输入并输出结果
 public static void main(String[] args) throws IOException {
 // 为了提高输入输出效率，使用 BufferedReader 和 BufferedWriter
 // BufferedReader 用于高效读取输入
 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
 // BufferedWriter 用于高效输出结果
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.out));

 // 读取 n 和 m
 // n 表示 BitSet 的大小，m 表示操作的数量
 String[] line = reader.readLine().split(" ");
 int n = Integer.parseInt(line[0]); // BitSet 的大小
 int m = Integer.parseInt(line[1]); // 操作的数量

 // 初始化两个 BitSet
```

```
// 创建一个 BitSet 数组，索引 0 不使用，1 和 2 分别对应题目中的 BitSet1 和 BitSet2
BitSet[] bitSets = new BitSet[3];
// 初始化 BitSet1，大小为 n，初始时所有位都为 0
bitSets[1] = new BitSet(n);
// 初始化 BitSet2，大小为 n，初始时所有位都为 0
bitSets[2] = new BitSet(n);

// 执行 m 次操作
// 循环处理每个操作
for (int i = 0; i < m; i++) {
 // 读取操作指令
 line = reader.readLine().split(" ");
 // 获取操作类型
 String operation = line[0];
 // 获取第一个操作数 (BitSet 编号)
 int set1 = Integer.parseInt(line[1]);
 // 获取第二个操作数 (BitSet 编号或位索引)
 int set2 = Integer.parseInt(line[2]);

 // 根据操作类型执行相应操作
 switch (operation) {
 case "AND":
 // 将 BitSet[set1] 与 BitSet[set2] 进行按位与操作，结果存储在 BitSet[set1] 中
 // 按位与操作：两个位都为 1 时结果才为 1，否则为 0
 bitSets[set1].and(bitSets[set2]);
 break;
 case "OR":
 // 将 BitSet[set1] 与 BitSet[set2] 进行按位或操作，结果存储在 BitSet[set1] 中
 // 按位或操作：两个位中至少有一个为 1 时结果为 1，否则为 0
 bitSets[set1].or(bitSets[set2]);
 break;
 case "XOR":
 // 将 BitSet[set1] 与 BitSet[set2] 进行按位异或操作，结果存储在 BitSet[set1] 中
 // 按位异或操作：两个位不同时结果为 1，相同时为 0
 bitSets[set1].xor(bitSets[set2]);
 break;
 case "FLIP":
 // 将 BitSet[set1] 中下标为 set2 的位翻转
 // 翻转操作：0 变 1，1 变 0
 bitSets[set1].flip(set2);
 break;
 case "SET":
 // 将 BitSet[set1] 中下标为 set2 的位设置为 1
 }
}
```

```
// 设置操作：将指定位置为 1
bitSets[set1].set(set2);
break;
}

// 打印两个 BitSet 中 1 的个数
// cardinality()方法返回 BitSet 中 1 的个数
// 每次操作后都要输出两个 BitSet 中 1 的个数
writer.write(bitSets[1].cardinality() + " " + bitSets[2].cardinality() + "\n");
}

// 刷新输出缓冲区，确保所有输出都被写入
writer.flush();
}

// 测试用例
// 验证程序的正确性
public static void test() {
 System.out.println("HackerRank Java BitSet 解题测试");

 // 创建两个大小为 5 的 BitSet
 BitSet bs1 = new BitSet(5);
 BitSet bs2 = new BitSet(5);

 // 初始状态: bs1 = "00000", bs2 = "00000"
 // cardinality() 返回 BitSet 中 1 的个数
 System.out.println("Initial: " + bs1.cardinality() + " " + bs2.cardinality()); // 应该输出 "0 0"

 // SET 1 4 -> bs1 = "00001"
 // 将 bs1 中下标为 4 的位设置为 1
 bs1.set(4);
 System.out.println("After SET 1 4: " + bs1.cardinality() + " " + bs2.cardinality()); // 应该输出 "1 0"

 // FLIP 2 2 -> bs2 = "00100"
 // 将 bs2 中下标为 2 的位翻转 (0 变 1)
 bs2.flip(2);
 System.out.println("After FLIP 2 2: " + bs1.cardinality() + " " + bs2.cardinality()); // 应该输出 "1 1"

 // OR 2 1 -> bs2 = "00101"
 // 将 bs2 与 bs1 进行按位或操作
```

```
 bs2. or (bs1) ;
 System.out.println("After OR 2 1: " + bs1.cardinality() + " " + bs2.cardinality()); //
应该输出 "1 2"
 }
}
```

---

文件: Code06\_HackerRankJavaBitSet.py

---

```
HackerRank Java BitSet
题目链接: https://www.hackerrank.com/challenges/java-bitset/problem
题目大意:
给定两个 BitSet，大小为 n，初始时所有位都为 0
执行一系列操作，每次操作后打印两个 BitSet 中 1 的个数
```

```
操作包括:
AND 1 2: 将 BitSet1 与 BitSet2 进行按位与操作，结果存储在 BitSet1 中
OR 1 2: 将 BitSet1 与 BitSet2 进行按位或操作，结果存储在 BitSet1 中
XOR 1 2: 将 BitSet1 与 BitSet2 进行按位异或操作，结果存储在 BitSet1 中
FLIP 1 2: 将 BitSet1 中下标为 2 的位翻转
SET 1 2: 将 BitSet1 中下标为 2 的位设置为 1
```

```
解题思路:
1. 使用整数的位来模拟 BitSet
2. 根据操作类型执行相应的位运算操作
3. 每次操作后计算并打印两个 BitSet 中 1 的个数
时间复杂度分析:
```

```
- AND, OR, XOR: O(n/32)
- FLIP, SET: O(1)
- count(): O(n/32)
空间复杂度: O(n)
```

```
class BitSet:
 def __init__(self, n):
 """
 构造函数，初始化大小为 n 的 BitSet，所有位都为 0
 :param n: BitSet 的大小
 """

 # BitSet 的大小
 self.size = n
 # 计算需要多少个整数来存储 n 位
 # (n + 31) // 32 是向上取整的写法
```

```

例如: n=100, 则需要(100+31)//32 = 4 个整数来存储 100 位
self.bits = [0] * ((n + 31) // 32)

def and_op(self, other):
 """
 按位与操作
 :param other: 另一个 BitSet 对象
 """
 # 对每一位进行按位与操作
 for i in range(len(self.bits)):
 self.bits[i] &= other.bits[i]

def or_op(self, other):
 """
 按位或操作
 :param other: 另一个 BitSet 对象
 """
 # 对每一位进行按位或操作
 for i in range(len(self.bits)):
 self.bits[i] |= other.bits[i]

def xor_op(self, other):
 """
 按位异或操作
 :param other: 另一个 BitSet 对象
 """
 # 对每一位进行按位异或操作
 for i in range(len(self.bits)):
 self.bits[i] ^= other.bits[i]

def flip(self, idx):
 """
 翻转指定位置的位
 :param idx: 位的下标
 """
 # 计算 idx 在数组中的位置和位偏移
 # array_idx 确定在 bits 数组中的哪个整数
 array_idx = idx // 32
 # bit_idx 确定在该整数中的哪一位
 bit_idx = idx % 32
 # 使用异或操作翻转指定位
 # 1 << bit_idx 创建一个只有第 bit_idx 位为 1 的数
 # ^= 异或操作, 实现翻转效果

```

```
self.bits[array_idx] ^= (1 << bit_idx)

def set_bit(self, idx):
 """
 设置指定位置的位为 1
 :param idx: 位的下标
 """

 # 计算 idx 在数组中的位置和位偏移
 # array_idx 确定在 bits 数组中的哪个整数
 array_idx = idx // 32
 # bit_idx 确定在该整数中的哪一位
 bit_idx = idx % 32
 # 使用按位或操作将指定位设置为 1
 # 1 << bit_idx 创建一个只有第 bit_idx 位为 1 的数
 # |= 按位或操作，将指定位设置为 1
 self.bits[array_idx] |= (1 << bit_idx)

def count(self):
 """
 计算 1 的个数
 :return: 1 的个数
 """

 result = 0
 # 遍历每一位所在的整数
 for bit in self.bits:
 # 计算一个整数中 1 的个数
 # bin(bit) 将整数转换为二进制字符串
 # .count('1') 统计字符串中'1'的个数
 result += bin(bit).count('1')
 return result

def main():
 """主函数，处理输入并输出结果"""
 # 读取 n 和 m
 # n 表示 BitSet 的大小，m 表示操作的数量
 n, m = map(int, input().split())

 # 初始化两个 BitSet
 # 创建一个 BitSet 列表，索引 0 不使用，1 和 2 分别对应题目中的 BitSet1 和 BitSet2
 bit_sets = [None, BitSet(n), BitSet(n)]

 # 执行 m 次操作
 # 循环处理每个操作
```

```
for _ in range(m):
 # 读取操作指令
 line = input().split()
 # 获取操作类型
 operation = line[0]
 # 获取第一个操作数 (BitSet 编号)
 set1 = int(line[1])
 # 获取第二个操作数 (BitSet 编号或位索引)
 set2 = int(line[2])

 # 根据操作类型执行相应的操作
 if operation == "AND":
 # 将 BitSet[set1]与 BitSet[set2]进行按位与操作，结果存储在 BitSet[set1]中
 # 按位与操作：两个位都为 1 时结果才为 1，否则为 0
 bit_sets[set1].and_op(bit_sets[set2])
 elif operation == "OR":
 # 将 BitSet[set1]与 BitSet[set2]进行按位或操作，结果存储在 BitSet[set1]中
 # 按位或操作：两个位中至少有一个为 1 时结果为 1，否则为 0
 bit_sets[set1].or_op(bit_sets[set2])
 elif operation == "XOR":
 # 将 BitSet[set1]与 BitSet[set2]进行按位异或操作，结果存储在 BitSet[set1]中
 # 按位异或操作：两个位不同时结果为 1，相同时为 0
 bit_sets[set1].xor_op(bit_sets[set2])
 elif operation == "FLIP":
 # 将 BitSet[set1]中下标为 set2 的位翻转
 # 翻转操作：0 变 1，1 变 0
 bit_sets[set1].flip(set2)
 elif operation == "SET":
 # 将 BitSet[set1]中下标为 set2 的位设置为 1
 # 设置操作：将指定位置为 1
 bit_sets[set1].set_bit(set2)

 # 打印两个 BitSet 中 1 的个数
 # 每次操作后都要输出两个 BitSet 中 1 的个数
 print(bit_sets[1].count(), bit_sets[2].count())

测试用例
def test():
 """测试用例，验证程序的正确性"""
 print("HackerRank Java BitSet 解题测试")

 # 创建两个大小为 5 的 BitSet
 bs1 = BitSet(5)
```

```

bs2 = BitSet(5)

初始状态: bs1 = "00000", bs2 = "00000"
count() 返回 BitSet 中 1 的个数
print("Initial:", bs1.count(), bs2.count()) # 应该输出 "0 0"

SET 1 4 -> bs1 = "00001"
将 bs1 中下标为 4 的位设置为 1
bs1.set_bit(4)
print("After SET 1 4:", bs1.count(), bs2.count()) # 应该输出 "1 0"

FLIP 2 2 -> bs2 = "00100"
将 bs2 中下标为 2 的位翻转 (0 变 1)
bs2.flip(2)
print("After FLIP 2 2:", bs1.count(), bs2.count()) # 应该输出 "1 1"

OR 2 1 -> bs2 = "00101"
将 bs2 与 bs1 进行按位或操作
bs2.or_op(bs1)
print("After OR 2 1:", bs1.count(), bs2.count()) # 应该输出 "1 2"

程序入口点
if __name__ == "__main__":
 # 运行测试用例
 test()

如果需要运行主程序, 取消下面的注释
main()

```

=====

文件: Code07\_RankingTheCows.cpp

```

=====

#include <iostream>
#include <vector>
#include <bitset>

// POJ 3275 Ranking the Cows
// 题目链接: http://poj.org/problem?id=3275
// 题目大意:
// FJ 想按照奶牛产奶的能力给她们排序。现在已知有 N 头奶牛 (1 ≤ N ≤ 1,000)。
// FJ 通过比较, 已经知道了 M (1 ≤ M ≤ 10,000) 对相对关系。
// 问你最少还要确定多少对牛的关系才能将所有的牛按照一定顺序排序起来。

```

```
// 解题思路：
// 1. 这是一个传递闭包问题
// 2. 使用 Floyd 算法求传递闭包
// 3. 用 bitset 优化 Floyd 算法，将时间复杂度从 O(N^3) 优化到 O(N^3/32)
// 4. 统计已知关系数，用完全图的关系数减去已知关系数就是答案
// 时间复杂度：O(N^3/32)
// 空间复杂度：O(N^2/32)

using namespace std;

// 最大奶牛数
const int MAXN = 1005;

// 使用 bitset 优化的邻接矩阵
// graph[i] 表示第 i 头牛能直接或间接到达的所有牛
// 例如：graph[i][j] 为 1 表示第 i 头牛能到达第 j 头牛
bitset<MAXN> graph[MAXN];

// 主函数，处理输入并输出结果
int main() {
 // 优化输入输出速度，关闭 stdio 同步，解除 cin 与 cout 的绑定
 ios::sync_with_stdio(false);
 cin.tie(0);

 int n, m;
 // 读取 N 和 M
 // n 表示奶牛的数量，m 表示已知的关系对数
 cin >> n >> m;

 // 初始化邻接矩阵
 // reset() 将所有位初始化为 0
 for (int i = 1; i <= n; i++) {
 graph[i].reset();
 }

 // 读取已知的 M 对关系
 // 每一对关系表示 a > b，即 a 到 b 有一条有向边
 for (int i = 0; i < m; i++) {
 int a, b;
 // 读取一对关系 a > b
 cin >> a >> b;
 // a > b，即 a 到 b 有一条有向边
```

```

 // set(b) 将第 b 位设置为 1, 表示 a 能到达 b
 graph[a].set(b);
}

// Floyd 求传递闭包, 使用 bitset 优化
// 通过 Floyd 算法计算所有奶牛之间的可达关系
// 枚举中间节点 k
for (int k = 1; k <= n; k++) {
 // 枚举起点 i
 for (int i = 1; i <= n; i++) {
 // 如果 i 到 k 有路径, 则 i 到 k 能到达的所有点, i 也能到达
 // graph[i][k] 检查第 k 位是否为 1, 即 i 是否能到达 k
 if (graph[i][k]) {
 // graph[i] |= graph[k] 将 graph[i] 与 graph[k] 按位或
 // 这表示 i 能到达 k 能到达的所有点
 graph[i] |= graph[k];
 }
 }
}

// 统计已知关系数
// 计算所有已知的奶牛之间的关系对数
int known = 0;
// 遍历每头牛
for (int i = 1; i <= n; i++) {
 // count() 返回 bitset 中 1 的个数
 // 即第 i 头牛能到达的牛的数量
 known += graph[i].count();
}

// 完全图的关系数是 n*(n-1)/2
// 答案是还需要确定的关系数
// 完全图有 n*(n-1)/2 对关系, 减去已知的关系数就是还需要确定的关系数
int result = n * (n - 1) / 2 - known;
// 输出结果
cout << result << "\n";

return 0;
}
=====

文件: Code07_RankingTheCows.java

```

```
=====
package class032;

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

// POJ 3275 Ranking the Cows
// 题目链接: http://poj.org/problem?id=3275
// 题目大意:
// FJ 想按照奶牛产奶的能力给她们排序。现在已知有 N 头奶牛 ($1 \leq N \leq 1,000$)。
// FJ 通过比较, 已经知道了 M ($1 \leq M \leq 10,000$) 对相对关系。
// 问你最少还要确定多少对牛的关系才能将所有的牛按照一定顺序排序起来。
```

```
// 解题思路:
// 1. 这是一个传递闭包问题
// 2. 使用 Floyd 算法求传递闭包
// 3. 使用 BitSet 优化 Floyd 算法, 将时间复杂度从 $O(N^3)$ 优化到 $O(N^3/64)$
// 4. 统计已知关系数, 用完全图的关系数减去已知关系数就是答案
// 时间复杂度: $O(N^3/64)$
// 空间复杂度: $O(N^2/64)$
```

```
public class Code07_RankingTheCows {

 // 主函数, 处理输入并输出结果
 public static void main(String[] args) throws IOException {
 // 使用 BufferedReader 提高输入效率
 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

 // 读取 N 和 M
 // N 表示奶牛的数量, M 表示已知的关系对数
 String[] line = reader.readLine().split(" ");
 int n = Integer.parseInt(line[0]); // 奶牛的数量
 int m = Integer.parseInt(line[1]); // 已知的关系对数

 // 使用 BitSet 优化的邻接矩阵
 // graph[i] 表示第 i 头牛能直接或间接到达的所有牛
 BitSet[] graph = new BitSet[n + 1];
 // 初始化每头牛的 BitSet
 for (int i = 1; i <= n; i++) {
 graph[i] = new BitSet(n + 1);
 }

 // 读取已知的 M 对关系
```

```

// 每一对关系表示 a > b, 即 a 到 b 有一条有向边
for (int i = 0; i < m; i++) {
 line = reader.readLine().split(" ");
 int a = Integer.parseInt(line[0]); // 较大的奶牛编号
 int b = Integer.parseInt(line[1]); // 较小的奶牛编号
 // a > b, 即 a 到 b 有一条有向边
 // 在 graph[a] 中将 b 的位置为 1, 表示 a 能到达 b
 graph[a].set(b);
}

// Floyd 求传递闭包, 使用 BitSet 优化
// 通过 Floyd 算法计算所有奶牛之间的可达关系
for (int k = 1; k <= n; k++) {
 // 枚举中间节点 k
 for (int i = 1; i <= n; i++) {
 // 如果 i 到 k 有路径, 则 i 到 k 能到达的所有点, i 也能到达
 // get(k) 检查第 k 位是否为 1, 即 i 是否能到达 k
 if (graph[i].get(k)) {
 // or(graph[k]) 将 graph[i] 与 graph[k] 按位或
 // 这表示 i 能到达 k 能到达的所有点
 graph[i].or(graph[k]);
 }
 }
}

// 统计已知关系数
// 计算所有已知的奶牛之间的关系对数
int known = 0;
for (int i = 1; i <= n; i++) {
 // cardinality() 返回 BitSet 中 1 的个数
 // 即第 i 头牛能到达的牛的数量
 known += graph[i].cardinality();
}

// 完全图的关系数是 n*(n-1)/2
// 答案是还需要确定的关系数
// 完全图有 n*(n-1)/2 对关系, 减去已知的关系数就是还需要确定的关系数
int result = n * (n - 1) / 2 - known;
// 输出结果
System.out.println(result);
}

// 测试用例

```

```
public static void test() {
 System.out.println("POJ 3275 Ranking the Cows 解题测试");
 // 由于这是在线评测题目，测试用例需要按照题目要求构造
}
}
=====
```

文件: Code07\_RankingTheCows.py

```
POJ 3275 Ranking the Cows
题目链接: http://poj.org/problem?id=3275
题目大意:
FJ 想按照奶牛产奶的能力给她们排序。现在已知有 N 头奶牛 ($1 \leq N \leq 1,000$)。
FJ 通过比较, 已经知道了 M ($1 \leq M \leq 10,000$) 对相对关系。
问你最少还要确定多少对牛的关系才能将所有的牛按照一定顺序排序起来。
```

```
解题思路:
1. 这是一个传递闭包问题
2. 使用 Floyd 算法求传递闭包
3. 使用整数位运算优化 Floyd 算法
4. 统计已知关系数, 用完全图的关系数减去已知关系数就是答案
时间复杂度: O(N^3/32)
空间复杂度: O(N^2/32)
```

```
def main():
 """主函数, 处理输入并输出结果"""
 # 读取输入
 # n 表示奶牛的数量, m 表示已知的关系对数
 n, m = map(int, input().split())

 # 使用整数数组模拟 bitset 优化的邻接矩阵
 # graph[i] 表示第 i 头牛能到达的牛的集合, 用整数的位来表示
 # 例如: graph[i] 的第 j 位为 1 表示第 i 头牛能到达第 j 头牛
 graph = [0] * (n + 1)
```

```
 # 读取已知的 M 对关系
 # 每一对关系表示 a > b, 即 a 到 b 有一条有向边
 for _ in range(m):
 # 读取一对关系 a > b
 a, b = map(int, input().split())
 # a > b, 即 a 到 b 有一条有向边
 # 在 graph[a] 中将 b 的位置为 1, 表示 a 能到达 b
```

```

1 << b 创建一个只有第 b 位为 1 的数
|= 按位或操作，将第 b 位设置为 1
graph[a] |= (1 << b)

Floyd 求传递闭包，使用位运算优化
通过 Floyd 算法计算所有奶牛之间的可达关系
枚举中间节点 k
for k in range(1, n + 1):
 # 枚举起点 i
 for i in range(1, n + 1):
 # 如果 i 到 k 有路径，则 i 到 k 能到达的所有点，i 也能到达
 # graph[i] & (1 << k) 检查第 k 位是否为 1，即 i 是否能到达 k
 if graph[i] & (1 << k):
 # graph[i] |= graph[k] 将 graph[i] 与 graph[k] 按位或
 # 这表示 i 能到达 k 能到达的所有点
 graph[i] |= graph[k]

统计已知关系数
计算所有已知的奶牛之间的关系对数
known = 0
遍历每头牛
for i in range(1, n + 1):
 # 统计第 i 头牛能到达的牛的数量
 # bin(graph[i]) 将整数转换为二进制字符串
 # .count('1') 统计字符串中 '1' 的个数
 known += bin(graph[i]).count('1')

完全图的关系数是 n*(n-1)/2
答案是还需要确定的关系数
完全图有 n*(n-1)/2 对关系，减去已知的关系数就是还需要确定的关系数
result = n * (n - 1) // 2 - known
输出结果
print(result)

测试用例
def test():
 """测试用例"""
 print("POJ 3275 Ranking the Cows 解题测试")
 # 由于这是在线评测题目，测试用例需要按照题目要求构造

程序入口点
if __name__ == "__main__":
 # 运行测试用例

```

```
test()
```

```
如果需要运行主程序，取消下面的注释
```

```
main()
```

```
=====
```

文件: Code08\_CompatibleNumbers.cpp

```
=====
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <bitset>
```

```
// Codeforces 165E Compatible Numbers
```

```
// 题目链接: https://codeforces.com/problemset/problem/165/E
```

```
// 题目大意:
```

```
// 给定一个长度为 n 的数组，对于数组中的每个数，找到数组中另一个数，
```

```
// 使得这两个数的按位与结果为 0。如果不存在这样的数，输出-1。
```

```
// 解题思路:
```

```
// 1. 两个数按位与结果为 0，意味着它们在二进制表示中没有同为 1 的位
```

```
// 2. 对于每个数 x，我们需要找到一个数 y，使得 x & y = 0
```

```
// 3. 这等价于找到一个数 y，使得 y 是 x 的按位取反的子集
```

```
// 4. 我们可以使用 SOS DP (Sum over Subsets DP) 来预处理每个数的子集
```

```
// 5. 对于每个数 x，我们查找 x 按位取反后是否有子集在数组中存在
```

```
// 时间复杂度: O(n + 3^k)，其中 k 是位数(22 位)
```

```
// 空间复杂度: O(2^k)
```

```
using namespace std;
```

```
// 2^22，因为题目中数的最大值是 4*10^6 < 2^22
```

```
const int MAXV = 1 << 22;
```

```
// 主函数，处理输入并输出结果
```

```
int main() {
```

```
 // 优化输入输出速度，关闭 stdio 同步，解除 cin 与 cout 的绑定
```

```
 ios::sync_with_stdio(false);
```

```
 cin.tie(0);
```

```
 int n;
```

```
 // 读取数组长度
```

```
 cin >> n;
```

```

// 存储输入数组
vector<int> a(n);
// 使用 bitset 标记数组中存在哪些数
// exists[i] 为 1 表示数 i 在数组中存在
bitset<MAXV> exists;
// 记录每个数在数组中的位置
// pos[i] 表示数 i 在数组中的位置, -1 表示不存在
vector<int> pos(MAXV, -1);

// 读取数组
for (int i = 0; i < n; i++) {
 // 读取第 i 个元素
 cin >> a[i];
 // 标记这个数存在
 exists.set(a[i]);
 // 记录这个数在数组中的位置
 pos[a[i]] = i;
}

// 存储答案, 初始化为 -1 表示未找到兼容数
vector<int> answer(n, -1);

// 对于每个数, 找到与它兼容的数
for (int i = 0; i < n; i++) {
 // 当前处理的数
 int x = a[i];
 // x 的按位取反(22 位)
 // (1 << 22) - 1 创建一个 22 位全为 1 的数
 // x ^ ((1 << 22) - 1) 对 x 进行按位异或, 实现按位取反
 int complement = x ^ ((1 << 22) - 1);

 // 查找 complement 的子集是否有在数组中存在的
 // 使用 SOS DP 的思想
 // mask 表示当前检查的 complement 的子集
 int mask = complement;
 // 循环枚举 complement 的所有子集
 while (mask > 0) {
 // 检查 mask 对应的数是否在数组中存在
 // exists[mask] 检查第 mask 位是否为 1
 if (exists[mask]) {
 // 找到兼容数, 记录其在原数组中的位置
 answer[i] = pos[mask];
 // 找到后跳出循环
 }
 }
}

```

```

 break;
 }
 // 枚举下一个子集
 // (mask - 1) & complement 计算 mask 的下一个子集
 mask = (mask - 1) & complement;
}

// 特殊情况：检查 0 是否在数组中
// 0 与任何数按位与都为 0，所以如果数组中有 0，它与任何数都兼容
if (answer[i] == -1 && exists[0]) {
 answer[i] = pos[0];
}
}

// 输出答案
for (int i = 0; i < n; i++) {
 if (answer[i] == -1) {
 // 未找到兼容数，输出-1
 cout << "-1 ";
 } else {
 // 找到兼容数，输出该数的值
 cout << a[answer[i]] << " ";
 }
}
// 输出换行符
cout << "\n";

return 0;
}

```

文件: Code08\_CompatibleNumbers.java

```

=====
package class032;

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

// Codeforces 165E Compatible Numbers
// 题目链接: https://codeforces.com/problemset/problem/165/E
// 题目大意:
// 给定一个长度为 n 的数组，对于数组中的每个数，找到数组中另一个数，

```

```
// 使得这两个数的按位与结果为 0。如果不存在这样的数，输出-1。

// 解题思路：
// 1. 两个数按位与结果为 0，意味着它们在二进制表示中没有同为 1 的位
// 2. 对于每个数 x，我们需要找到一个数 y，使得 $x \& y = 0$
// 3. 这等价于找到一个数 y，使得 y 是 x 的按位取反的子集
// 4. 我们可以使用 SOS DP (Sum over Subsets DP) 来预处理每个数的子集
// 5. 对于每个数 x，我们查找 x 按位取反后是否有子集在数组中存在
// 时间复杂度： $O(n + 3^k)$ ，其中 k 是位数(22 位)
// 空间复杂度： $O(2^k)$
```

```
public class Code08_CompatibleNumbers {

 // 主函数，处理输入并输出结果
 public static void main(String[] args) throws IOException {
 // 使用 BufferedReader 提高输入效率
 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

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

 // 读取数组元素
 String[] parts = reader.readLine().split(" ");
 int[] a = new int[n];
 // 使用 BitSet 标记数组中存在哪些数
 BitSet exists = new BitSet(1 << 22);
 // 记录每个数在数组中的位置
 int[] pos = new int[1 << 22];
 // 初始化 pos 数组为-1，表示数不存在
 Arrays.fill(pos, -1);

 // 读取数组
 for (int i = 0; i < n; i++) {
 // 读取第 i 个元素
 a[i] = Integer.parseInt(parts[i]);
 // 标记这个数存在
 exists.set(a[i]);
 // 记录这个数在数组中的位置
 pos[a[i]] = i;
 }

 // 存储答案数组，初始化为-1 表示未找到兼容数
 int[] answer = new int[n];
```

```

Arrays.fill(answer, -1);

// 对于每个数，找到与它兼容的数
for (int i = 0; i < n; i++) {
 // 当前处理的数
 int x = a[i];
 // x 的按位取反(22 位)
 // (1 << 22) - 1 创建一个 22 位全为 1 的数
 // x ^ ((1 << 22) - 1) 对 x 进行按位异或，实现按位取反
 int complement = x ^ ((1 << 22) - 1);

 // 查找 complement 的子集是否有在数组中存在的
 // 使用 SOS DP 的思想
 // mask 表示当前检查的 complement 的子集
 int mask = complement;
 // 循环枚举 complement 的所有子集
 while (mask > 0) {
 // 检查 mask 对应的数是否在数组中存在
 if (exists.get(mask)) {
 // 找到兼容数，记录其在原数组中的位置
 answer[i] = pos[mask];
 // 找到后跳出循环
 break;
 }
 // 枚举下一个子集
 // (mask - 1) & complement 计算 mask 的下一个子集
 mask = (mask - 1) & complement;
 }

 // 特殊情况：检查 0 是否在数组中
 // 0 与任何数按位与都为 0，所以如果数组中有 0，它与任何数都兼容
 if (answer[i] == -1 && exists.get(0)) {
 answer[i] = pos[0];
 }
}

// 输出答案
// 使用 StringBuilder 提高字符串拼接效率
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++) {
 if (answer[i] == -1) {
 // 未找到兼容数，输出-1
 sb.append("-1 ");
 }
}

```

```

 } else {
 // 找到兼容数，输出该数的值
 sb.append(a[answer[i]]).append(" ");
 }
 }
 // 输出结果，使用 trim() 去除末尾空格
 System.out.println(sb.toString().trim());
}

// 测试用例
public static void test() {
 System.out.println("Codeforces 165E Compatible Numbers 解题测试");
 // 由于这是在线评测题目，测试用例需要按照题目要求构造
}
}
=====

文件: Code08_CompatibleNumbers.py
=====

Codeforces 165E Compatible Numbers
题目链接: https://codeforces.com/problemset/problem/165/E
题目大意:
给定一个长度为 n 的数组，对于数组中的每个数，找到数组中另一个数，
使得这两个数的按位与结果为 0。如果不存在这样的数，输出-1。

解题思路:
1. 两个数按位与结果为 0，意味着它们在二进制表示中没有同为 1 的位
2. 对于每个数 x，我们需要找到一个数 y，使得 $x \& y = 0$
3. 这等价于找到一个数 y，使得 y 是 x 的按位取反的子集
4. 我们可以使用 SOS DP (Sum over Subsets DP) 来预处理每个数的子集
5. 对于每个数 x，我们查找 x 按位取反后是否有子集在数组中存在
时间复杂度: $O(n + 3^k)$ ，其中 k 是位数(22 位)
空间复杂度: $O(2^k)$

def main():
 """主函数，处理输入并输出结果"""
 # 读取数组长度
 n = int(input())
 # 读取数组元素
 a = list(map(int, input().split()))

 # 标记数组中存在哪些数

```

```

exists[i] 为 True 表示数 i 在数组中存在
exists = [False] * (1 << 22)
记录每个数在数组中的位置
pos[i] 表示数 i 在数组中的位置, -1 表示不存在
pos = [-1] * (1 << 22)

读取数组
for i in range(n):
 # 标记这个数存在
 exists[a[i]] = True
 # 记录这个数在数组中的位置
 pos[a[i]] = i

存储答案, 初始化为-1 表示未找到兼容数
answer = [-1] * n

对于每个数, 找到与它兼容的数
for i in range(n):
 # 当前处理的数
 x = a[i]
 # x 的按位取反(22 位)
 # (1 << 22) - 1 创建一个 22 位全为 1 的数
 # x ^ ((1 << 22) - 1) 对 x 进行按位异或, 实现按位取反
 complement = x ^ ((1 << 22) - 1)

 # 查找 complement 的子集是否有在数组中存在的
 # 使用 SOS DP 的思想
 # mask 表示当前检查的 complement 的子集
 mask = complement
 # 循环枚举 complement 的所有子集
 while mask > 0:
 # 检查 mask 对应的数是否在数组中存在
 if exists[mask]:
 # 找到兼容数, 记录其在原数组中的位置
 answer[i] = pos[mask]
 # 找到后跳出循环
 break
 # 枚举下一个子集
 # (mask - 1) & complement 计算 mask 的下一个子集
 mask = (mask - 1) & complement

 # 特殊情况: 检查 0 是否在数组中
 # 0 与任何数按位与都为 0, 所以如果数组中有 0, 它与任何数都兼容

```

```

 if answer[i] == -1 and exists[0]:
 answer[i] = pos[0]

输出答案
result = []
for i in range(n):
 if answer[i] == -1:
 # 未找到兼容数，输出-1
 result.append("-1")
 else:
 # 找到兼容数，输出该数的值
 result.append(str(a[answer[i]]))

使用 join 将结果连接成字符串并输出
print(" ".join(result))

测试用例
def test():
 """测试用例"""
 print("Codeforces 165E Compatible Numbers 解题测试")
 # 由于这是在线评测题目，测试用例需要按照题目要求构造

程序入口点
if __name__ == "__main__":
 # 运行测试用例
 test()

如果需要运行主程序，取消下面的注释
main()

```

=====

文件: Code09\_IsUnique.cpp

=====

```

// LeetCode 面试题 01.01. 判定字符是否唯一
// 题目链接: https://leetcode-cn.com/problems/is-unique-lcci/
// 题目大意:
// 实现一个算法，确定一个字符串 s 的所有字符是否全都不同。
//
// 示例 1:
// 输入: s = "leetcode"
// 输出: false
//

```

```
// 示例 2:
// 输入: s = "abc"
// 输出: true

//
// 限制:
// 0 <= len(s) <= 100
// 如果你不使用额外的数据结构，会很加分。

//
// 解题思路:
// 使用位运算优化的方法:
// 1. 由于字符集可能是 ASCII 或 Unicode，但题目中通常假设是小写字母或有限范围
// 2. 我们可以使用一个整数或位集合来表示每个字符是否出现过
// 3. 对于每个字符，检查对应的位是否已经被设置，如果是则返回 false，否则设置该位

// 时间复杂度: O(n)，其中 n 是字符串的长度
// 空间复杂度: O(1)，使用了固定大小的位集合或整数
```

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <algorithm>
#include <bitset>
#include <chrono>

// 方法 1: 使用哈希集合
// 优点: 简单直观，适用于任意字符集
// 缺点: 使用了额外的数据结构
// 参数 astr: 待检查的字符串
// 返回值: 如果字符串中所有字符都唯一返回 true，否则返回 false
bool isUnique1(const std::string& astr) {
 // 边界条件检查
 // 如果字符串为空，认为所有字符都唯一
 if (astr.empty()) {
 return true;
 }

 // 使用哈希集合存储已出现的字符
 // unordered_set 的查找和插入操作平均时间复杂度为 O(1)
 std::unordered_set<char> seen;

 // 遍历字符串中的每个字符
 // 使用范围 for 循环遍历字符串
 for (char c : astr) {
```

```

// 检查字符是否已经在集合中
// find(c) 查找字符 c，如果找到返回指向该元素的迭代器，否则返回 end()
// != seen.end() 表示找到了该字符
if (seen.find(c) != seen.end()) {
 // 如果字符已经出现过，说明有重复
 return false;
}

// 将字符添加到集合中
// insert(c) 将字符 c 插入到集合中
seen.insert(c);

}

// 所有字符都不重复
return true;
}

// 方法 2：使用位运算模拟 Bitset（仅适用于小写字母 a-z）
// 优点：空间效率更高，不使用额外的数据结构
// 缺点：仅适用于小写字母范围
// 参数 astr：待检查的字符串（假设只包含小写字母）
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
bool isUnique2(const std::string& astr) {
 // 边界条件检查
 // 如果字符串为空，认为所有字符都唯一
 if (astr.empty()) {
 return true;
 }

 // 鸽巢原理：如果字符串长度超过字母表大小，必然有重复
 // 小写字母只有 26 个，如果字符串长度超过 26，必然有重复字符
 if (astr.length() > 26) {
 return false;
 }

 // 使用整数的二进制位来存储字符出现情况
 // 使用一个整数的低 26 位来表示字符 a-z 是否出现
 int checker = 0;

 // 遍历字符串中的每个字符
 for (char c : astr) {
 // 检查字符是否为小写字母
 // 如果不是小写字母，回退到方法 1 处理任意字符集
 if (c < 'a' || c > 'z') {

```

```

 // 回退到哈希集合方法，处理任意字符集
 return isUnique1(astr);
}

// 计算字符对应的位位置
// c - 'a' 将字符映射到 0-25 的范围
int bit = c - 'a';

// 检查该位是否已经被设置
// (1 << bit) 是将 1 左移 bit 位，创建一个只有第 bit 位为 1 的数
// checker & (1 << bit) 按位与操作，检查 checker 的第 bit 位是否为 1
// 如果结果大于 0，说明该位已经被设置，即字符重复
if ((checker & (1 << bit)) > 0) {
 // 字符重复，返回 false
 return false;
}

// 设置该位为 1
// checker |= (1 << bit) 按位或操作，将 checker 的第 bit 位设置为 1
checker |= (1 << bit);

// 所有字符都不重复
return true;
}

// 方法 3：使用 C++ 的 bitset
// 优点：适用于较大的字符集，效率高
// 缺点：需要预先知道字符集大小
// 参数 astr：待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
bool isUnique3(const std::string& astr) {
 // 边界条件检查
 // 如果字符串为空，认为所有字符都唯一
 if (astr.empty()) {
 return true;
 }

 // 使用 bitset 存储字符是否出现过（假设是 ASCII 字符集）
 // bitset<256> 表示一个 256 位的位集合，用于存储 ASCII 字符
 std::bitset<256> seen;

 // 遍历字符串中的每个字符

```

```
for (char c : astr) {
 // 获取字符的 ASCII 码值
 // static_cast<unsigned char>(c) 将 char 转换为 unsigned char
 // 这样可以正确处理负数字符值
 unsigned char uc = static_cast<unsigned char>(c);

 // 检查字符是否已经出现过
 // test(uc) 检查第 uc 位是否为 1
 if (seen.test(uc)) {
 // 如果字符已经出现过，说明有重复
 return false;
 }

 // 标记字符已出现
 // set(uc) 将第 uc 位设置为 1
 seen.set(uc);
}

// 所有字符都不重复
return true;
}

// 方法 4：不使用额外数据结构（排序后比较相邻元素）
// 优点：不使用额外的数据结构
// 缺点：时间复杂度较高，且会修改原始数据（这里使用副本）
// 参数 astr：待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
bool isUnique4(const std::string& astr) {
 // 边界条件检查
 // 如果字符串为空，认为所有字符都唯一
 if (astr.empty()) {
 return true;
 }

 // 鸽巢原理
 // 如果字符串长度超过字符集大小，必然有重复
 // 假设是 ASCII 字符集，最多有 256 个不同的字符
 if (astr.length() > 256) {
 return false;
 }

 // 创建字符串副本并排序
 // 这样不会修改原始字符串
```

```
std::string sorted_str = astr;
// 对字符串进行排序
std::sort(sorted_str.begin(), sorted_str.end());

// 检查相邻字符是否相同
// 遍历排序后的字符串，比较相邻元素
for (size_t i = 0; i < sorted_str.length() - 1; ++i) {
 // 如果相邻字符相同，说明有重复
 if (sorted_str[i] == sorted_str[i + 1]) {
 return false;
 }
}

// 所有字符都不重复
return true;
}

// 方法 5：工程化版本，增加异常处理和参数验证
// 参数 astr：待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
bool isUniqueWithValidation(const std::string& astr) {
 try {
 // 参数验证（在 C++ 中，传入空指针会导致未定义行为，所以这里假设 astr 有效）

 // 鸽巢原理快速判断
 // 如果字符串长度超过字符集大小，必然有重复
 // 假设使用扩展 ASCII 字符集，最多 128 个字符
 if (astr.length() > 128) {
 return false;
 }

 // 使用 bitset 实现
 // 创建一个大小为 128 的 bitset，用于存储 ASCII 字符
 std::bitset<128> seen;

 // 遍历字符串中的每个字符
 for (char c : astr) {
 // 获取字符的 ASCII 码值
 // static_cast<unsigned char>(c) 将 char 转换为 unsigned char
 unsigned char uc = static_cast<unsigned char>(c);

 // 检查是否超出处理范围
 // 如果字符的 ASCII 码值超过 127，说明是扩展 ASCII 或 Unicode 字符
 }
 } catch (...) {
 // 处理异常
 }
}
```

```
 if (uc >= 128) {
 // 对于扩展 ASCII 或 Unicode 字符，使用哈希集合处理
 return isUnique1(astr);
 }

 // 检查字符是否已经出现过
 // test(uc) 检查第 uc 位是否为 1
 if (seen.test(uc)) {
 // 字符重复，返回 false
 return false;
 }

 // 标记字符已出现
 // set(uc) 将第 uc 位设置为 1
 seen.set(uc);
}

// 所有字符都不重复
return true;
}

catch (const std::exception& e) {
 // 记录异常（在实际应用中可以使用日志）
 // 在生产环境中，应该使用日志框架记录异常
 std::cerr << "Error in isUniqueWithValidation: " << e.what() << std::endl;
 // 异常情况下保守返回 false
 return false;
}

}

// 单元测试
void runTests() {
 std::cout << "Running unit tests..." << std::endl;

 // 定义测试用例结构体
 struct TestCase {
 std::string input; // 输入字符串
 bool expected; // 期望的输出结果
 };

 // 定义测试用例
 std::vector<TestCase> testCases = {
 {"leetcode", false}, // 有重复字符
 {"abc", true}, // 无重复字符
 };
}
```

```
 {"", true}, // 空字符串
 {"AbCdEfG", true}, // 包含大小写字母
 {"a", true}, // 单个字符
 {"abcdefghijklmnopqrstuvwxyz", true}, // 包含所有小写字母
 {"abcdefghijklmnopqrstuvwxyzabc", false} // 有重复字符
 };

// 定义要测试的函数和名称结构体
struct Method {
 bool (*func)(const std::string&); // 函数指针
 std::string name; // 方法名称
};

// 定义所有测试方法
std::vector<Method> methods = {
 {isUnique1, "Method 1 (HashSet)" },
 {isUnique2, "Method 2 (Bitwise)" },
 {isUnique3, "Method 3 (Bitset)" },
 {isUnique4, "Method 4 (Sorting)" },
 {isUniqueWithValidation, "Method 5 (With Validation)" }
};

// 测试所有方法
for (const auto& method : methods) {
 std::cout << "\n" << method.name << ":" << std::endl;
 // 遍历所有测试用例
 for (const auto& test : testCases) {
 // 调用被测试的方法
 bool result = method.func(test.input);
 // 判断测试结果是否正确
 const std::string& status = (result == test.expected) ? "PASS" : "FAIL";
 // 输出测试结果
 std::cout << " Input: \\" << test.input << "\\" -> Result: "
 << (result ? "true" : "false")
 << " (Expected: " << (test.expected ? "true" : "false")
 << ") - " << status << std::endl;
 }
}

// 性能测试
void performanceTest() {
 std::cout << "\nRunning performance tests..." << std::endl;
```

```
// 生成测试数据
// 创建一个包含所有小写字母的字符串
std::string uniqueStr;
for (char c = 'a'; c <= 'z'; ++c) {
 uniqueStr += c;
}

// 创建一个有重复字符的字符串
std::string duplicateStr = uniqueStr + 'a';

// 定义要测试的函数和名称结构体
struct Method {
 bool (*func)(const std::string&); // 函数指针
 std::string name; // 方法名称
};

// 定义所有测试方法
std::vector<Method> methods = {
 {isUnique1, "Method 1 (HashSet)" },
 {isUnique2, "Method 2 (Bitwise)" },
 {isUnique3, "Method 3 (Bitset)" },
 {isUnique4, "Method 4 (Sorting)" },
 {isUniqueWithValidation, "Method 5 (With Validation)" }
};

// 定义测试迭代次数
const int iterations = 100000;

// 测试每种方法的性能
for (const auto& method : methods) {
 std::cout << "\n" << method.name << " performance:" << std::endl;

 // 测试唯一字符串
 // 记录开始时间
 auto start = std::chrono::high_resolution_clock::now();
 // 执行多次测试以获得更准确的结果
 for (int i = 0; i < iterations; ++i) {
 method.func(uniqueStr);
 }
 // 记录结束时间
 auto end = std::chrono::high_resolution_clock::now();
 // 计算执行时间（微秒）
}
```

```
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
 // 计算平均执行时间
 double avgTime = static_cast<double>(duration) / iterations;
 std::cout << " Unique string average time: " << avgTime << " μs" << std::endl;

 // 测试重复字符串
 // 记录开始时间
 start = std::chrono::high_resolution_clock::now();
 // 执行多次测试以获得更准确的结果
 for (int i = 0; i < iterations; ++i) {
 method.func(duplicateStr);
 }
 // 记录结束时间
 end = std::chrono::high_resolution_clock::now();
 // 计算执行时间（微秒）
 duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
 // 计算平均执行时间
 avgTime = static_cast<double>(duration) / iterations;
 std::cout << " Duplicate string average time: " << avgTime << " μs" << std::endl;
}

}

// 主函数，程序入口点
int main() {
 std::cout << "LeetCode 面试题 01.01. 判定字符是否唯一" << std::endl;
 std::cout << "Using bitwise operations for optimization" << std::endl;

 // 运行单元测试
 runTests();

 // 运行性能测试
 performanceTest();

 // 复杂度分析
 std::cout << "\n复杂度分析:" << std::endl;
 std::cout << "位运算方法 (isUnique2):" << std::endl;
 std::cout << " 时间复杂度: O(n)，其中 n 是字符串的长度" << std::endl;
 std::cout << " 空间复杂度: O(1)，仅使用一个整数存储位信息" << std::endl;
 std::cout << " 优势: 不需要额外的数据结构，空间效率高" << std::endl;
 std::cout << " 限制: 仅适用于有限范围的字符 (如小写字母 a-z)" << std::endl;

 std::cout << "\n适用场景总结:" << std::endl;
```

```
 std::cout << "1. 当输入字符集较小时（如只有小写字母），位运算方法效率最高" << std::endl;
 std::cout << "2. 当输入字符集较大时，bitset 方法更通用且高效" << std::endl;
 std::cout << "3. 当不允许使用额外数据结构时，排序方法是一种选择，但效率较低" << std::endl;
 std::cout << "4. 在工程实践中，应根据具体场景选择合适的方法，并考虑异常处理和边界情况" << std::endl;

 return 0;
}
```

=====

文件: Code09\_IsUnique.java

=====

```
package class032;

import java.util.BitSet;

// LeetCode 面试题 01.01. 判定字符是否唯一
// 题目链接: https://leetcode-cn.com/problems/is-unique-lcci/
// 题目大意:
// 实现一个算法，确定一个字符串 s 的所有字符是否全都不同。
//
// 示例 1:
// 输入: s = "leetcode"
// 输出: false
//
// 示例 2:
// 输入: s = "abc"
// 输出: true
//
// 限制:
// 0 <= len(s) <= 100
// 如果你不使用额外的数据结构，会很加分。
//
// 解题思路:
// 使用位运算(Bitset)优化的方法:
// 1. 由于字符集可能是 ASCII 或 Unicode，但题目中通常假设是小写字母或有限范围
// 2. 我们可以使用一个整数或位集合来表示每个字符是否出现过
// 3. 对于每个字符，检查对应的位是否已经被设置，如果是则返回 false，否则设置该位
//
// 时间复杂度: O(n)，其中 n 是字符串的长度
// 空间复杂度: O(1)，使用了固定大小的位集合
public class Code09_IsUnique {
```

```
// 方法 1：使用 Java 内置的 BitSet 实现
// 优点：适用于任意字符集，不受字符范围限制
// 参数 astr：待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
public static boolean isUnique1(String astr) {
 // 边界条件检查
 // 如果字符串为 null 或为空，认为所有字符都唯一
 if (astr == null || astr.isEmpty()) {
 return true;
 }

 // 使用 BitSet 存储字符是否出现过
 // BitSet 的大小为 256，假设是 ASCII 字符集
 BitSet bitSet = new BitSet(256);

 // 遍历字符串中的每个字符
 for (int i = 0; i < astr.length(); i++) {
 // 获取第 i 个字符
 char c = astr.charAt(i);

 // 检查当前字符是否已经出现过
 // get(c) 检查第 c 位是否为 true
 if (bitSet.get(c)) {
 // 如果该字符已经出现过，说明有重复字符
 return false;
 }

 // 标记当前字符已经出现
 // set(c) 将第 c 位设置为 true
 bitSet.set(c);
 }

 // 所有字符都不重复
 return true;
}

// 方法 2：使用位运算模拟 Bitset（仅适用于小写字母 a-z）
// 优点：空间效率更高，仅使用一个整数
// 参数 astr：待检查的字符串（仅包含小写字母 a-z）
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
public static boolean isUnique2(String astr) {
 // 边界条件检查
```

```
// 如果字符串为 null 或为空，认为所有字符都唯一
if (astr == null || astr.isEmpty()) {
 return true;
}

// 使用一个整数的二进制位来存储字符出现情况
// 假设输入只包含小写字母 a-z，使用一个整数的低 26 位来表示
int checker = 0;

// 遍历字符串中的每个字符
for (int i = 0; i < astr.length(); i++) {
 // 获取第 i 个字符
 char c = astr.charAt(i);

 // 如果不是小写字母，使用其他方法
 // 这是为了处理输入不符合假设的情况
 if (c < 'a' || c > 'z') {
 // 回退到 BitSet 方法，处理任意字符集
 return isUnique1(astr);
 }

 // 计算字符对应的位位置
 // c - 'a' 将字符映射到 0-25 的范围
 int bit = c - 'a';

 // 检查该位是否已经被设置
 // (checker & (1 << bit)) > 0 检查第 bit 位是否为 1
 if ((checker & (1 << bit)) > 0) {
 // 如果该位已经为 1，说明字符重复
 return false;
 }

 // 设置该位为 1
 // checker |= (1 << bit) 将第 bit 位设置为 1
 checker |= (1 << bit);
}

// 所有字符都不重复
return true;
}

// 方法 3：不使用额外数据结构的方法（仅适用于有限字符集）
// 思路：对字符串进行排序，然后检查相邻字符是否相同
```

```
// 缺点：时间复杂度较高，且会修改原字符串（如果不是副本的话）
// 参数 astr: 待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
public static boolean isUnique3(String astr) {
 // 边界条件检查
 // 如果字符串为 null 或为空，认为所有字符都唯一
 if (astr == null || astr.isEmpty()) {
 return true;
 }

 // 如果字符串长度超过字符集大小，必然有重复
 // 假设是 ASCII 字符集，最多有 256 个不同的字符
 if (astr.length() > 256) {
 // 根据鸽巢原理，长度超过字符集大小的字符串必然有重复字符
 return false;
 }

 // 转换为字符数组并排序
 // toCharArray() 将字符串转换为字符数组
 char[] chars = astr.toCharArray();
 // 对字符数组进行排序
 java.util.Arrays.sort(chars);

 // 检查相邻字符是否相同
 for (int i = 0; i < chars.length - 1; i++) {
 // 如果相邻字符相同，说明有重复
 if (chars[i] == chars[i + 1]) {
 return false;
 }
 }

 // 所有字符都不重复
 return true;
}

// 工程化改进版本：增加参数验证和异常处理
// 参数 astr: 待检查的字符串
// 返回值：如果字符串中所有字符都唯一返回 true，否则返回 false
public static boolean isUniqueWithValidation(String astr) {
 try {
 // 参数验证
 // 检查输入参数是否为 null
 if (astr == null) {
```

```
// 抛出异常，说明输入参数不能为 null
throw new IllegalArgumentException("Input string cannot be null");
}

// 如果字符串过长，提前返回 false (鸽巢原理)
// 假设使用的是 Unicode 字符集的一个子集，最多 128 个字符
if (astr.length() > 128) {
 // 根据鸽巢原理，长度超过字符集大小的字符串必然有重复字符
 return false;
}

// 使用 BitSet 实现
// 创建一个大小为 128 的 BitSet，用于存储 ASCII 字符
BitSet seen = new BitSet(128);

// 遍历字符串中的每个字符
for (int i = 0; i < astr.length(); i++) {
 // 获取第 i 个字符
 char c = astr.charAt(i);

 // 仅处理 ASCII 可见字符（可根据实际需求调整）
 // ASCII 可见字符范围是 32-126
 if (c >= 32 && c <= 126) {
 // 检查当前字符是否已经出现过
 if (seen.get(c)) {
 // 如果该字符已经出现过，说明有重复字符
 return false;
 }
 // 标记当前字符已经出现
 seen.set(c);
 }
}

// 所有字符都不重复
return true;
} catch (Exception e) {
 // 记录异常日志（在实际应用中）
 // 在生产环境中，应该使用日志框架记录异常
 System.err.println("Error in isUnique: " + e.getMessage());
 // 异常情况下保守返回 false
 return false;
}
}
```

```
// 单元测试方法
// 验证各种实现方法的正确性
public static void runTests() {
 // 测试用例 1: 普通字符串, 有重复字符
 String test1 = "leetcode";
 System.out.println("Test 1: " + test1 + " -> " + isUnique1(test1) + " (Expected: false)");
}

// 测试用例 2: 普通字符串, 无重复字符
String test2 = "abc";
System.out.println("Test 2: " + test2 + " -> " + isUnique1(test2) + " (Expected: true)");

// 测试用例 3: 空字符串
String test3 = "";
System.out.println("Test 3: Empty string -> " + isUnique1(test3) + " (Expected: true)");

// 测试用例 4: 包含大小写字母
String test4 = "AbCdEfG";
System.out.println("Test 4: " + test4 + " -> " + isUnique1(test4) + " (Expected: true)");

// 测试用例 5: 边界情况, 只有一个字符
String test5 = "a";
System.out.println("Test 5: " + test5 + " -> " + isUnique1(test5) + " (Expected: true)");

// 测试不同方法的结果一致性
System.out.println("\nMethod comparison:");
System.out.println("isUnique1(\"abc\"): " + isUnique1("abc"));
System.out.println("isUnique2(\"abc\"): " + isUnique2("abc"));
System.out.println("isUnique3(\"abc\"): " + isUnique3("abc"));
System.out.println("isUniqueWithValidation(\"abc\"): " + isUniqueWithValidation("abc"));
}

// 主函数, 程序入口点
public static void main(String[] args) {
 System.out.println("LeetCode 面试题 01.01. 判定字符是否唯一");
 System.out.println("使用位运算优化实现");

 // 运行单元测试
 runTests();

 // 性能测试
 System.out.println("\nPerformance Testing:");
}
```

```

// 生成一个较长的唯一字符串
// 创建一个包含所有小写字母的字符串
StringBuilder sb = new StringBuilder();
for (char c = 'a'; c <= 'z'; c++) {
 sb.append(c);
}
String longUniqueStr = sb.toString();

// 测试性能
long startTime, endTime;

// 测试方法 1 性能
startTime = System.nanoTime();
for (int i = 0; i < 10000; i++) {
 isUnique1(longUniqueStr);
}
endTime = System.nanoTime();
System.out.println("Method 1 average time: " + ((endTime - startTime) / 10000) + " ns");

// 测试方法 2 性能（仅小写字母）
startTime = System.nanoTime();
for (int i = 0; i < 10000; i++) {
 isUnique2(longUniqueStr);
}
endTime = System.nanoTime();
System.out.println("Method 2 average time: " + ((endTime - startTime) / 10000) + " ns");

// 复杂度分析
System.out.println("\n 复杂度分析:");
System.out.println("时间复杂度: O(n), 其中 n 是字符串的长度");
System.out.println("空间复杂度: O(1), 使用了固定大小的位集合或整数");
}

}
=====

文件: Code09_IsUnique.py
=====

LeetCode 面试题 01.01. 判定字符是否唯一
题目链接: https://leetcode-cn.com/problems/is-unique-lcci/
题目大意:
实现一个算法，确定一个字符串 s 的所有字符是否全都不同。

```

```

LeetCode 面试题 01.01. 判定字符是否唯一
题目链接: https://leetcode-cn.com/problems/is-unique-lcci/
题目大意:
实现一个算法，确定一个字符串 s 的所有字符是否全都不同。

```

```

示例 1:
输入: s = "leetcode"
输出: false

示例 2:
输入: s = "abc"
输出: true

限制:
0 <= len(s) <= 100
如果你不使用额外的数据结构，会很加分。

解题思路:
使用位运算优化的方法:
1. 由于字符集可能是 ASCII 或 Unicode，但题目中通常假设是小写字母或有限范围
2. 我们可以使用一个整数或位集合来表示每个字符是否出现过
3. 对于每个字符，检查对应的位是否已经被设置，如果是则返回 false，否则设置该位

时间复杂度: O(n)，其中 n 是字符串的长度
空间复杂度: O(1)，使用了固定大小的位集合或整数
```

```
方法 1: 使用集合实现
优点: 简单直观，适用于任意字符集
缺点: 使用了额外的数据结构
```

```
def is_unique_1(s: str) -> bool:
 """
 使用集合判断字符是否唯一
```

参数:

s: 输入字符串

返回:

bool: 如果所有字符都唯一返回 True，否则返回 False

时间复杂度: O(n)

空间复杂度: O(k)，其中 k 是字符集大小，最坏情况下为 O(n)

"""

```
边界条件检查
```

```
如果字符串为 None 或为空，认为所有字符都唯一
```

```
if s is None or len(s) == 0:
```

```
 return True
```

```
使用集合存储已出现的字符
集合的查找和插入操作平均时间复杂度为 O(1)
seen = set()
```

```
遍历字符串中的每个字符
for char in s:
 # 检查字符是否已经在集合中
 # char in seen 的时间复杂度为 O(1)
 if char in seen:
 # 如果字符已经出现过，说明有重复
 return False
 # 将字符添加到集合中
 # seen.add(char) 的时间复杂度为 O(1)
 seen.add(char)

所有字符都不重复
return True
```

```
方法 2：使用位运算模拟 Bitset（仅适用于小写字母 a-z）
优点：空间效率更高，不使用额外的数据结构
缺点：仅适用于小写字母范围
```

```
def is_unique_2(s: str) -> bool:
"""
使用位运算判断字符是否唯一（仅适用于小写字母）
```

参数：

s: 输入字符串（假设只包含小写字母）

返回：

bool: 如果所有字符都唯一返回 True，否则返回 False

时间复杂度：O(n)

空间复杂度：O(1)，仅使用一个整数

```
"""
```

# 边界条件检查

# 如果字符串为 None 或为空，认为所有字符都唯一

```
if s is None or len(s) == 0:
```

```
 return True
```

# 鸽巢原理：如果字符串长度超过字母表大小，必然有重复

# 小写字母只有 26 个，如果字符串长度超过 26，必然有重复字符

```
if len(s) > 26:
 return False

使用整数的二进制位来存储字符出现情况
使用一个整数的低 26 位来表示字符 a-z 是否出现
checker = 0

遍历字符串中的每个字符
for char in s:
 # 检查字符是否为小写字母
 # 如果不是小写字母, 回退到方法 1 处理任意字符集
 if not 'a' <= char <= 'z':
 # 回退到集合方法, 处理任意字符集
 return is_unique_1(s)

 # 计算字符对应的位位置
 # ord(char) 获取字符的 ASCII 码值
 # ord('a') 获取字符'a'的 ASCII 码值
 # bit 表示字符 char 在 checker 中的位位置 (0-25)
 bit = ord(char) - ord('a')

 # 检查该位是否已经被设置
 # (1 << bit) 是将 1 左移 bit 位, 创建一个只有第 bit 位为 1 的数
 # checker & (1 << bit) 按位与操作, 检查 checker 的第 bit 位是否为 1
 # 如果结果大于 0, 说明该位已经被设置, 即字符重复
 if (checker & (1 << bit)) > 0:
 # 字符重复, 返回 False
 return False

 # 设置该位为 1
 # checker |= (1 << bit) 按位或操作, 将 checker 的第 bit 位设置为 1
 checker |= (1 << bit)

所有字符都不重复
return True

方法 3: 不使用额外数据结构 (排序后比较相邻元素)
优点: 不使用额外的数据结构
缺点: 时间复杂度较高

def is_unique_3(s: str) -> bool:
 """
 通过排序后比较相邻元素判断字符是否唯一
 """
```

参数:

s: 输入字符串

返回:

bool: 如果所有字符都唯一返回 True, 否则返回 False

时间复杂度:  $O(n \log n)$ , 排序的时间复杂度

空间复杂度:  $O(n)$ , 用于存储排序后的字符数组

"""

# 边界条件检查

# 如果字符串为 None 或为空, 认为所有字符都唯一

if s is None or len(s) == 0:

return True

# 鸽巢原理

# 如果字符串长度超过字符集大小, 必然有重复

# 假设是 ASCII 字符集, 最多有 256 个不同的字符

if len(s) > 256:

return False

# 排序字符串

# sorted(s) 返回一个新的排序后的字符列表

# 时间复杂度为  $O(n \log n)$

sorted\_chars = sorted(s)

# 检查相邻字符是否相同

# 遍历排序后的字符数组, 比较相邻元素

for i in range(len(sorted\_chars) - 1):

# 如果相邻字符相同, 说明有重复

if sorted\_chars[i] == sorted\_chars[i + 1]:

return False

# 所有字符都不重复

return True

# 方法 4: 工程化版本, 增加异常处理和参数验证

def is\_unique\_with\_validation(s: str) -> bool:

"""

工程化版本, 增加异常处理和参数验证

参数:

s: 输入字符串

返回：

bool：如果所有字符都唯一返回 True，否则返回 False

时间复杂度：O(n)

空间复杂度：O(1)，使用固定大小的数组

"""

try:

# 参数验证

# 检查输入参数是否为 None

if s is None:

# 抛出异常，说明输入参数不能为 None

raise ValueError("Input string cannot be None")

# 鸽巢原理快速判断

# 如果字符串长度超过字符集大小，必然有重复

# 假设使用扩展 ASCII 字符集，最多 128 个字符

if len(s) > 128:

return False

# 使用固定大小的布尔数组（模拟 Bitset）

# 适用于 ASCII 字符（0-127）

# 创建一个大小为 128 的布尔数组，初始化为 False

char\_set = [False] \* 128

# 遍历字符串中的每个字符

for char in s:

# 获取字符的 ASCII 码值

# ord(char) 返回字符的 ASCII 码值

val = ord(char)

# 检查是否超出处理范围

# 如果字符的 ASCII 码值超过 127，说明是扩展 ASCII 或 Unicode 字符

if val >= 128:

# 对于扩展 ASCII 或 Unicode 字符，回退到集合方法处理

return is\_unique\_1(s)

# 检查字符是否已经出现过

# char\_set[val] 为 True 表示字符已经出现过

if char\_set[val]:

# 字符重复，返回 False

return False

```
标记字符已出现
将 char_set[val] 设置为 True，表示字符已经出现
char_set[val] = True

所有字符都不重复
return True

except Exception as e:
 # 记录异常（在实际应用中可以使用日志）
 # 在生产环境中，应该使用日志框架记录异常
 print(f"Error in is_unique_with_validation: {e}")
 # 异常情况下保守返回 False
 return False

单元测试
def run_tests():
 """
 运行单元测试
 """

 # 定义测试用例
 # 每个测试用例是一个元组，包含输入字符串和期望的输出结果
 test_cases = [
 ("leetcode", False), # 有重复字符
 ("abc", True), # 无重复字符
 ("", True), # 空字符串
 ("AbCdEfG", True), # 包含大小写字母
 ("a", True), # 单个字符
 ("abcdefghijklmnopqrstuvwxyz", True), # 包含所有小写字母
 ("abcdefghijklmnopqrstuvwxyzabc", False) # 有重复字符
]

 print("Running unit tests... \n")

 # 定义所有测试方法
 methods = [
 (is_unique_1, "Method 1 (Set)"),
 (is_unique_2, "Method 2 (Bitwise)"),
 (is_unique_3, "Method 3 (Sorting)"),
 (is_unique_with_validation, "Method 4 (With Validation)")
]

 # 测试所有方法
 for method, method_name in methods:
```

```
print(f"Testing {method_name}:")
遍历所有测试用例
for s, expected in test_cases:
 # 调用被测试的方法
 result = method(s)
 # 判断测试结果是否正确
 status = "PASS" if result == expected else "FAIL"
 # 输出测试结果
 print(f' {s} -> {result} (Expected: {expected}) - {status}')
print()

性能测试
def performance_test():
 """
 性能测试
 """
 import time

 # 生成测试数据
 # 1. 长字符串，所有字符唯一
 # 创建一个包含所有小写字母的字符串
 unique_str = ''.join(chr(ord('a') + i) for i in range(26))

 # 2. 长字符串，有重复字符
 # 在唯一字符串的基础上添加一个重复字符
 duplicate_str = unique_str + 'a'

 # 定义所有测试方法
 methods = [
 (is_unique_1, "Method 1 (Set)"),
 (is_unique_2, "Method 2 (Bitwise)"),
 (is_unique_3, "Method 3 (Sorting)"),
 (is_unique_with_validation, "Method 4 (With Validation)")
]

 print("Running performance tests...\n")

 # 测试每种方法的性能
 for method, method_name in methods:
 print(f"Performance of {method_name}:")

 # 测试唯一字符串
 # 记录开始时间
```

```
start_time = time.time()
执行多次测试以获得更准确的结果
iterations = 10000
循环执行方法
for _ in range(iterations):
 method(unique_str)
记录结束时间
end_time = time.time()
计算平均执行时间（转换为微秒）
avg_time = (end_time - start_time) * 1000000 / iterations
print(f" Unique string average time: {avg_time:.2f} μs")

测试重复字符串
记录开始时间
start_time = time.time()
循环执行方法
for _ in range(iterations):
 method(duplicate_str)
记录结束时间
end_time = time.time()
计算平均执行时间（转换为微秒）
avg_time = (end_time - start_time) * 1000000 / iterations
print(f" Duplicate string average time: {avg_time:.2f} μs")
print()

主函数
if __name__ == "__main__":
 print("LeetCode 面试题 01.01. 判定字符是否唯一")
 print("使用位运算优化实现\n")

 # 运行单元测试
 run_tests()

 # 运行性能测试
 performance_test()

 # 复杂度分析
 print("复杂度分析:")
 print("位运算方法 (is_unique_2):")
 print(" 时间复杂度: O(n), 其中 n 是字符串的长度")
 print(" 空间复杂度: O(1), 仅使用一个整数存储位信息")
 print(" 优势: 不需要额外的数据结构, 空间效率高")
 print(" 限制: 仅适用于有限范围的字符 (如小写字母 a-z) ")
```

```
print("\n适用场景总结:")
print("1. 当输入字符集较小时（如只有小写字母），位运算方法效率最高")
print("2. 当输入字符集较大时，集合方法更通用")
print("3. 当不允许使用额外数据结构时，排序方法是一种选择，但效率较低")
print("4. 在工程实践中，应根据具体场景选择合适的方法，并考虑异常处理和边界情况")
```

---

文件: Code10\_MaximumXOR.cpp

---

```
// LeetCode 421. 数组中两个数的最大异或值
// 题目链接: https://leetcode-cn.com/problems/maximum-xor-of-two-numbers-in-an-array/
// 题目大意:
// 给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n。
//
// 进阶：你可以在 O(n) 的时间解决这个问题吗？
//
// 示例 1:
// 输入: nums = [3,10,5,25,2,8]
// 输出: 28
// 解释: 最大的结果是 5 XOR 25 = 28.
//
// 示例 2:
// 输入: nums = [0]
// 输出: 0
//
// 示例 3:
// 输入: nums = [2,4]
// 输出: 6
//
// 示例 4:
// 输入: nums = [8,10,2]
// 输出: 10
//
// 示例 5:
// 输入: nums = [14,70,53,83,49,91,36,80,92,51,66,70]
// 输出: 127
//
// 提示:
// 1 ≤ nums.length ≤ 2 * 10^4
// 0 ≤ nums[i] ≤ 2^31 - 1
//
```

```

// 解题思路:
// 方法一: 暴力解法 (O(n2)时间复杂度, 不推荐)
// 方法二: 前缀树(字典树)优化的位运算方法 (O(n)时间复杂度)
// 方法三: 基于异或性质的位运算方法 (O(n)时间复杂度)
//
// 这里我们主要实现方法二和方法三, 它们都是基于位运算的高效解法

#include <iostream>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <chrono>

// 方法一: 暴力解法
int findMaximumXOR1(const std::vector<int>& nums) {
 if (nums.size() <= 1) {
 return 0;
 }

 int maxResult = 0;

 // 遍历所有可能的数对
 for (size_t i = 0; i < nums.size(); ++i) {
 for (size_t j = i + 1; j < nums.size(); ++j) {
 int currentXOR = nums[i] ^ nums[j];
 maxResult = std::max(maxResult, currentXOR);
 }
 }

 return maxResult;
}

// 方法二: 基于位运算和集合的方法
int findMaximumXOR2(const std::vector<int>& nums) {
 if (nums.size() <= 1) {
 return 0;
 }

 int maxResult = 0;
 int mask = 0;

 // 从最高位到最低位依次确定结果的每一位
 for (int i = 30; i >= 0; --i) { // 31位整数, 忽略符号位 (因为 nums[i] >= 0)
 mask |= 1 << i;
 std::unordered_set<int> seen;
 for (int num : nums) {
 if ((num & mask) == 0) {
 seen.insert(num);
 } else {
 seen.insert(num ^ mask);
 }
 }
 maxResult |= seen.size() == 1 ? 0 : 1;
 }

 return maxResult;
}

```

```

// 构建当前位的掩码
mask |= (1 << i);

// 存储所有数在当前掩码下的前缀
std::unordered_set<int> prefixes;
for (int num : nums) {
 prefixes.insert(num & mask);
}

// 假设当前位为 1，构造可能的最大值
int tempMax = maxResult | (1 << i);

// 检查是否存在两个数，它们的前缀异或等于 tempMax
bool found = false;
for (int prefix : prefixes) {
 // 如果 prefix ^ target = tempMax，那么 target = prefix ^ tempMax
 if (prefixes.count(prefix ^ tempMax)) {
 // 找到可行的解，设置当前位为 1
 maxResult = tempMax;
 found = true;
 break;
 }
}

// 如果没有找到，当前位保持为 0 (maxResult 不变)
if (found) {
 // 可以继续处理下一位
}
}

return maxResult;
}

// 方法三：前缀树（字典树）方法
class TrieNode {
public:
 TrieNode* children[2]; // 0 和 1 两个子节点

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

```

```

~TrieNode() {
 // 递归释放内存
 if (children[0]) delete children[0];
 if (children[1]) delete children[1];
}
};

int findMaximumXOR3(const std::vector<int>& nums) {
 if (nums.size() <= 1) {
 return 0;
 }

 // 构建前缀树
 TrieNode* root = new TrieNode();

 // 向前缀树中插入一个数
 auto insert = [] (TrieNode* root, int num) {
 TrieNode* node = root;

 // 从最高位到最低位插入
 for (int i = 30; i >= 0; --i) { // 31位整数，忽略符号位
 int bit = (num >> i) & 1;

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

 node = node->children[bit];
 }
 };

 // 在已有的前缀树中查找与给定数异或结果最大的数
 auto search = [] (TrieNode* root, int num) -> int {
 TrieNode* node = root;
 int xorResult = 0;

 // 从最高位到最低位查找
 for (int i = 30; i >= 0; --i) { // 31位整数，忽略符号位
 int bit = (num >> i) & 1;
 int desiredBit = 1 - bit; // 寻找相反的位以最大化异或结果

 if (node->children[desiredBit]) {
 // 可以找到相反的位，异或结果的当前位为1

```

```

 xorResult |= (1 << i);
 node = node->children[desiredBit];
 } else if (node->children[bit]) {
 // 找不到相反的位，只能使用相同的位
 node = node->children[bit];
 } else {
 // 两个子节点都不存在，提前结束
 break;
 }
}

return xorResult;
};

// 先插入第一个数
insert(root, nums[0]);

int maxResult = 0;

// 对于每个数，插入并查找能产生最大异或值的数
for (size_t i = 1; i < nums.size(); ++i) {
 maxResult = std::max(maxResult, search(root, nums[i]));
 insert(root, nums[i]);
}

// 释放内存
delete root;

return maxResult;
}

// 方法四：工程化版本，增加异常处理和参数验证
int findMaximumXORWithValidation(const std::vector<int>& nums) {
 try {
 // 参数验证（在 C++ 中，传入空指针会导致未定义行为，所以这里假设 nums 有效）

 // 边界情况处理
 if (nums.size() <= 1) {
 return 0;
 }

 // 验证所有元素是否为非负数
 for (int num : nums) {

```

```

 if (num < 0) {
 throw std::invalid_argument("All elements must be non-negative integers");
 }
}

// 使用方法二实现
return findMaximumXOR2(nums);
}

catch (const std::exception& e) {
 // 记录异常（在实际应用中可以使用日志）
 std::cerr << "Error in findMaximumXORWithValidation: " << e.what() << std::endl;
 // 异常情况下返回 0
 return 0;
}

}

// 单元测试
void runTests() {
 std::cout << "Running unit tests..." << std::endl;

 struct TestCase {
 std::vector<int> input;
 int expected;
 };

 std::vector<TestCase> testCases = {
 {{3, 10, 5, 25, 2, 8}, 28},
 {{0}, 0},
 {{2, 4}, 6},
 {{8, 10, 2}, 10},
 {{14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70}, 127}
 };
}

// 定义要测试的函数和名称
struct Method {
 int (*func)(const std::vector<int>&);
 std::string name;
};

std::vector<Method> methods = {
 {findMaximumXOR1, "Method 1 (Brute Force)" },
 {findMaximumXOR2, "Method 2 (Bitwise with Set)" },
 {findMaximumXOR3, "Method 3 (Trie)" },
};

```

```

{findMaximumXORWithValidation, "Method 4 (With Validation)"}
};

// 测试所有方法
for (const auto& method : methods) {
 std::cout << "\n" << method.name << ":" << std::endl;
 for (const auto& test : testCases) {
 int result = method.func(test.input);
 const std::string& status = (result == test.expected) ? "PASS" : "FAIL";
 std::cout << " Input: [";
 for (size_t i = 0; i < test.input.size(); ++i) {
 std::cout << test.input[i];
 if (i < test.input.size() - 1) std::cout << ", ";
 }
 std::cout << "] -> Result: " << result
 << " (Expected: " << test.expected
 << ") - " << status << std::endl;
 }
}

// 性能测试
void performanceTest() {
 std::cout << "\nRunning performance tests..." << std::endl;

 // 生成大规模测试数据
 std::vector<int> largeNums;
 for (int i = 0; i < 10000; ++i) {
 largeNums.push_back(i);
 }

 // 仅在小规模数据上测试方法 1
 std::vector<int> smallNums = {3, 10, 5, 25, 2, 8};
 auto start = std::chrono::high_resolution_clock::now();
 int result1 = findMaximumXOR1(smallNums);
 auto end = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
 std::cout << "Performance of Method 1 (Brute Force): " << duration << " μs" << std::endl;
 std::cout << "Result: " << result1 << std::endl;

 // 测试方法 2
 start = std::chrono::high_resolution_clock::now();
 int result2 = findMaximumXOR2(largeNums);
}
```

```
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "\nPerformance of Method 2 (Bitwise with Set): " << duration << " μs" <<
std::endl;
std::cout << "Result: " << result2 << std::endl;

// 测试方法 3
start = std::chrono::high_resolution_clock::now();
int result3 = findMaximumXOR3(largeNums);
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "\nPerformance of Method 3 (Trie): " << duration << " μs" << std::endl;
std::cout << "Result: " << result3 << std::endl;

// 测试方法 4
start = std::chrono::high_resolution_clock::now();
int result4 = findMaximumXORWithValidation(largeNums);
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "\nPerformance of Method 4 (With Validation): " << duration << " μs" <<
std::endl;
std::cout << "Result: " << result4 << std::endl;
}

// 主函数
int main() {
 std::cout << "LeetCode 421. 数组中两个数的最大异或值" << std::endl;
 std::cout << "Using bitwise operations for optimization" << std::endl;

 // 运行单元测试
 runTests();

 // 运行性能测试
 performanceTest();

 // 复杂度分析
 std::cout << "\n复杂度分析:" << std::endl;
 std::cout << "方法一（暴力解法）：" << std::endl;
 std::cout << " 时间复杂度: O(n2)，其中 n 是数组长度" << std::endl;
 std::cout << " 空间复杂度: O(1)" << std::endl;
 std::cout << " 优点: 实现简单" << std::endl;
 std::cout << " 缺点: 对于大数组效率低" << std::endl;
```

```

 std::cout << "\n方法二（基于位运算和集合）：" << std::endl;
 std::cout << " 时间复杂度: O(n)，每个位处理需要 O(n)时间，总共处理 32 个位" << std::endl;
 std::cout << " 空间复杂度: O(n)，用于存储前缀集合" << std::endl;
 std::cout << " 优点: 时间效率高，实现相对简单" << std::endl;

 std::cout << "\n方法三（前缀树）：" << std::endl;
 std::cout << " 时间复杂度: O(n)，构建树和查询都是 O(n)时间" << std::endl;
 std::cout << " 空间复杂度: O(n)，用于存储前缀树" << std::endl;
 std::cout << " 优点: 位操作思想清晰，扩展性好" << std::endl;
 std::cout << " 注意: 实际实现中需要注意内存管理，避免内存泄漏" << std::endl;

 std::cout << "\n适用场景总结:" << std::endl;
 std::cout << "1. 对于小数组，可以使用暴力解法" << std::endl;
 std::cout << "2. 对于大数组，应使用方法二或方法三，它们的时间复杂度都是 O(n)" << std::endl;
 std::cout << "3. 在工程实践中，方法二实现更简洁，而方法三更能体现位运算的思想" << std::endl;
 std::cout << "4. 当需要处理大量相似查询时，前缀树方法更具优势" << std::endl;

 return 0;
}

```

---

文件: Code10\_MaximumXOR.java

---

```

package class032;

import java.util.HashSet;
import java.util.Set;

// LeetCode 421. 数组中两个数的最大异或值
// 题目链接: https://leetcode-cn.com/problems/maximum-xor-of-two-numbers-in-an-array/
// 题目大意:
// 给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n。
//
// 进阶：你可以在 O(n) 的时间解决这个问题吗？
//
// 示例 1:
// 输入: nums = [3, 10, 5, 25, 2, 8]
// 输出: 28
// 解释: 最大的结果是 5 XOR 25 = 28.
//
// 示例 2:
// 输入: nums = [0]

```

```
// 输出: 0
//
// 示例 3:
// 输入: nums = [2, 4]
// 输出: 6
//
// 示例 4:
// 输入: nums = [8, 10, 2]
// 输出: 10
//
// 示例 5:
// 输入: nums = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]
// 输出: 127
//
// 提示:
// 1 <= nums.length <= 2 * 10^4
// 0 <= nums[i] <= 2^31 - 1
//
// 解题思路:
// 方法一: 暴力解法 (O(n^2)时间复杂度, 不推荐)
// 方法二: 前缀树(字典树)优化的位运算方法 (O(n)时间复杂度)
// 方法三: 基于异或性质的位运算方法 (O(n)时间复杂度)
//
// 这里我们主要实现方法二和方法三, 它们都是基于位运算的高效解法
```

```
public class Code10_MaximumXOR {

 // 方法一: 暴力解法
 // 时间复杂度: O(n^2)
 // 空间复杂度: O(1)
 // 参数 nums: 输入的整数数组
 // 返回值: 数组中任意两个数异或的最大值
 public static int findMaximumXOR1(int[] nums) {
 // 初始化最大异或结果为 0
 int maxResult = 0;

 // 遍历所有可能的数对
 // 外层循环遍历第一个数
 for (int i = 0; i < nums.length; i++) {
 // 内层循环遍历第二个数 (j > i 避免重复计算)
 for (int j = i + 1; j < nums.length; j++) {
 // 计算当前两个数的异或值
 // ^ 表示按位异或操作
 maxResult = Math.max(maxResult, nums[i] ^ nums[j]);
 }
 }
 return maxResult;
 }
}
```

```

 int currentXOR = nums[i] ^ nums[j];
 // 更新最大异或结果
 // Math.max 返回两个数中的较大值
 maxResult = Math.max(maxResult, currentXOR);
 }
}

// 返回最大异或结果
return maxResult;
}

// 方法二：基于位运算和集合的方法
// 时间复杂度: O(n)
// 空间复杂度: O(n)
// 参数 nums: 输入的整数数组
// 返回值: 数组中任意两个数异或的最大值
public static int findMaximumXOR2(int[] nums) {
 // 初始化最大异或结果为 0
 int maxResult = 0;
 // 初始化掩码为 0，用于提取数字的前缀
 int mask = 0;

 // 从最高位到最低位依次确定结果的每一位
 // 31 位整数，但符号位为 0（因为题目中 nums[i] >= 0），所以从第 30 位开始
 for (int i = 30; i >= 0; i--) {
 // 构建当前位的掩码
 // mask |= (1 << i) 将 mask 的第 i 位设置为 1
 // 这样 mask 就包含了从最高位到当前位的所有位
 mask |= (1 << i);

 // 存储所有数在当前掩码下的前缀
 // 使用 HashSet 存储前缀，便于快速查找
 Set<Integer> prefixes = new HashSet<>();
 // 遍历数组中的每个数
 for (int num : nums) {
 // num & mask 提取 num 在当前掩码下的前缀
 // 例如：如果 mask 是 11100000，那么 num & mask 就提取 num 的高 3 位
 prefixes.add(num & mask);
 }

 // 假设当前位为 1，构造可能的最大值
 // maxResult | (1 << i) 将 maxResult 的第 i 位设置为 1
 // 这是我们希望得到的最大异或结果
 }
}

```

```

int tempMax = maxResult | (1 << i);

// 检查是否存在两个数，它们的前缀异或等于 tempMax
// 遍历所有前缀
for (int prefix : prefixes) {
 // 如果 prefix ^ target = tempMax, 那么 target = prefix ^ tempMax
 // 这是基于异或运算的性质：如果 a ^ b = c, 那么 a ^ c = b
 // 我们要找是否存在另一个前缀 target, 使得 prefix ^ target = tempMax
 if (prefixes.contains(prefix ^ tempMax)) {
 // 找到可行的解，说明最大异或结果的当前位可以为 1
 // 设置当前位为 1
 maxResult = tempMax;
 // 找到解后跳出循环，继续处理下一位
 break;
 }
}

// 如果没有找到，当前位保持为 0 (maxResult 不变)
// 这是因为我们初始化 tempMax 时已经将 maxResult 的当前位设为 1
// 如果找不到匹配的前缀对，说明当前位必须为 0
}

// 返回最大异或结果
return maxResult;
}

// 方法三：前缀树（字典树）方法
// 时间复杂度: O(n)
// 空间复杂度: O(n)
// 参数 nums: 输入的整数数组
// 返回值: 数组中任意两个数异或的最大值
public static int findMaximumXOR3(int[] nums) {
 // 参数验证
 // 检查输入数组是否为 null 或长度为 0
 if (nums == null || nums.length == 0) {
 // 如果数组为空，返回 0
 return 0;
 }

 // 构建前缀树
 // 创建前缀树的根节点
 TrieNode root = new TrieNode();
 // 遍历数组中的每个数，将其插入到前缀树中
}

```

```
for (int num : nums) {
 // 调用 insert 方法将 num 插入到前缀树中
 insert(root, num);
}

// 初始化最大异或结果为 0
int maxResult = 0;

// 对于每个数，在前缀树中寻找能产生最大异或值的另一个数
// 遍历数组中的每个数
for (int num : nums) {
 // 调用 search 方法在前缀树中查找与 num 异或结果最大的数
 // Math.max 更新最大异或结果
 maxResult = Math.max(maxResult, search(root, num));
}

// 返回最大异或结果
return maxResult;
}

// 前缀树节点定义
// 前缀树节点用于存储二进制数的每一位
private static class TrieNode {
 // children 数组存储 0 和 1 两个子节点
 // children[0] 表示当前位为 0 的子节点
 // children[1] 表示当前位为 1 的子节点
 TrieNode[] children;

 // 构造函数
 public TrieNode() {
 // 初始化 children 数组，大小为 2
 children = new TrieNode[2];
 }
}

// 向前缀树中插入一个数
// 参数 root: 前缀树的根节点
// 参数 num: 要插入的数
private static void insert(TrieNode root, int num) {
 // 从根节点开始
 TrieNode node = root;

 // 从最高位到最低位插入
 for (int i = 31; i >= 0; i--) {
 int bit = (num >> i) & 1;
 if (node.children[bit] == null) {
 node.children[bit] = new TrieNode();
 }
 node = node.children[bit];
 }
}
```

```

// 31 位整数，忽略符号位，所以从第 30 位开始到第 0 位
for (int i = 30; i >= 0; i--) {
 // 提取 num 的第 i 位
 // (num >> i) & 1 将 num 右移 i 位，然后与 1 进行按位与操作
 // 这样可以提取 num 的第 i 位（0 或 1）
 int bit = (num >> i) & 1;

 // 如果当前位对应的子节点不存在，则创建新节点
 if (node.children[bit] == null) {
 // 创建新的子节点
 node.children[bit] = new TrieNode();
 }

 // 移动到子节点
 node = node.children[bit];
}

// 在已有的前缀树中查找与给定数异或结果最大的数
// 参数 root：前缀树的根节点
// 参数 num：给定的数
// 返回值：num 与前缀树中某个数异或的最大值
private static int search(TrieNode root, int num) {
 // 从根节点开始
 TrieNode node = root;
 // 初始化异或结果为 0
 int xor = 0;

 // 从最高位到最低位查找
 // 31 位整数，忽略符号位，所以从第 30 位开始到第 0 位
 for (int i = 30; i >= 0; i--) {
 // 提取 num 的第 i 位
 int bit = (num >> i) & 1;
 // 寻找相反的位以最大化异或结果
 // 1 - bit 可以得到 bit 的相反值（0 变 1，1 变 0）
 int desiredBit = 1 - bit;

 // 如果存在相反的位
 if (node.children[desiredBit] != null) {
 // 可以找到相反的位，异或结果的当前位为 1
 // xor |= (1 << i) 将 xor 的第 i 位设置为 1
 xor |= (1 << i);
 // 移动到相反位对应的子节点
 node = node.children[desiredBit];
 }
 }
}

```

```
 node = node.children[desiredBit];
 } else {
 // 找不到相反的位，只能使用相同的位
 // 移动到相同位对应的子节点
 node = node.children[bit];
 }
}

// 返回异或结果
return xor;
}

// 工程化改进版本：增加参数验证和异常处理
// 参数 nums：输入的整数数组
// 返回值：数组中任意两个数异或的最大值
public static int findMaximumXORWithValidation(int[] nums) {
 try {
 // 参数验证
 // 检查输入数组是否为 null
 if (nums == null) {
 // 抛出异常，说明输入参数不能为 null
 throw new IllegalArgumentException("Input array cannot be null");
 }

 // 边界情况处理
 // 检查数组长度
 if (nums.length <= 1) {
 // 如果数组只有一个元素或为空，最大异或值为 0
 return nums.length == 1 ? 0 : 0;
 }

 // 使用方法二实现
 // 调用 findMaximumXOR2 方法计算最大异或值
 return findMaximumXOR2(nums);
 } catch (Exception e) {
 // 记录异常日志（在实际应用中）
 // 在生产环境中，应该使用日志框架记录异常
 System.err.println("Error in findMaximumXOR: " + e.getMessage());
 // 异常情况下返回 0
 return 0;
 }
}
```

```
// 单元测试方法
// 验证各种实现方法的正确性
public static void runTests() {
 // 测试用例 1
 int[] nums1 = {3, 10, 5, 25, 2, 8};
 System.out.println("Test 1: [3,10,5,25,2,8] -> " + findMaximumXOR2(nums1) + " (Expected: 28)");
}

// 测试用例 2
int[] nums2 = {0};
System.out.println("Test 2: [0] -> " + findMaximumXOR2(nums2) + " (Expected: 0)");

// 测试用例 3
int[] nums3 = {2, 4};
System.out.println("Test 3: [2,4] -> " + findMaximumXOR2(nums3) + " (Expected: 6)");

// 测试用例 4
int[] nums4 = {8, 10, 2};
System.out.println("Test 4: [8,10,2] -> " + findMaximumXOR2(nums4) + " (Expected: 10)");

// 测试用例 5
int[] nums5 = {14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70};
System.out.println("Test 5: [14,70,53,83,49,91,36,80,92,51,66,70] -> " +
 findMaximumXOR2(nums5) + " (Expected: 127)");

// 测试不同方法的结果一致性
System.out.println("\nMethod comparison:");
System.out.println("Test 1 results:");
System.out.println(" findMaximumXOR1: " + findMaximumXOR1(nums1));
System.out.println(" findMaximumXOR2: " + findMaximumXOR2(nums1));
System.out.println(" findMaximumXOR3: " + findMaximumXOR3(nums1));
System.out.println(" findMaximumXORWithValidation: " +
findMaximumXORWithValidation(nums1));
}

// 性能测试方法
// 测试不同方法的性能
public static void performanceTest() {
 // 生成大规模测试数据
 // 创建一个包含 10000 个随机整数的数组
 int[] largeNums = new int[10000];
 for (int i = 0; i < largeNums.length; i++) {
 // 生成 0 到 Integer.MAX_VALUE 之间的随机整数
 }
}
```

```

 largeNums[i] = (int)(Math.random() * Integer.MAX_VALUE);
 }

// 测试方法一时间（仅在小规模数据上测试，大规模数据会超时）
int[] smallNums = {3, 10, 5, 25, 2, 8};
// 记录开始时间
long startTime = System.nanoTime();
// 调用方法一计算结果
int result1 = findMaximumXOR1(smallNums);
// 记录结束时间
long endTime = System.nanoTime();
System.out.println("\nPerformance of findMaximumXOR1: " +
 (endTime - startTime) + " ns, Result: " + result1);

// 测试方法二时间
// 记录开始时间
startTime = System.nanoTime();
// 调用方法二计算结果
int result2 = findMaximumXOR2(largeNums);
// 记录结束时间
endTime = System.nanoTime();
System.out.println("Performance of findMaximumXOR2: " +
 (endTime - startTime) / 1000 + " μs, Result: " + result2);

// 测试方法三时间
// 记录开始时间
startTime = System.nanoTime();
// 调用方法三计算结果
int result3 = findMaximumXOR3(largeNums);
// 记录结束时间
endTime = System.nanoTime();
System.out.println("Performance of findMaximumXOR3: " +
 (endTime - startTime) / 1000 + " μs, Result: " + result3);
}

// 主函数，程序入口点
public static void main(String[] args) {
 System.out.println("LeetCode 421. 数组中两个数的最大异或值");
 System.out.println("使用位运算优化实现");

 // 运行单元测试
 runTests();
}

```

```

// 运行性能测试
performanceTest();

// 复杂度分析
System.out.println("\n 复杂度分析:");
System.out.println("方法一（暴力解法）:");
System.out.println(" 时间复杂度: O(n2)，其中 n 是数组长度");
System.out.println(" 空间复杂度: O(1)");
System.out.println(" 优点: 实现简单");
System.out.println(" 缺点: 对于大数组效率低");

System.out.println("\n 方法二（基于位运算和集合）:");
System.out.println(" 时间复杂度: O(n)，每个位处理需要 O(n) 时间，总共处理 32 个位");
System.out.println(" 空间复杂度: O(n)，用于存储前缀集合");
System.out.println(" 优点: 时间效率高，实现相对简单");

System.out.println("\n 方法三（前缀树）:");
System.out.println(" 时间复杂度: O(n)，构建树和查询都是 O(n) 时间");
System.out.println(" 空间复杂度: O(n)，用于存储前缀树");
System.out.println(" 优点: 位操作思想清晰，扩展性好");

System.out.println("\n 适用场景总结:");
System.out.println("1. 对于小数组，可以使用暴力解法");
System.out.println("2. 对于大数组，应使用方法二或方法三，它们的时间复杂度都是 O(n)");
System.out.println("3. 在工程实践中，方法二实现更简洁，而方法三更能体现位运算的思想");
}

}

```

文件: Code10\_MaximumXOR.py

```

LeetCode 421. 数组中两个数的最大异或值
题目链接: https://leetcode-cn.com/problems/maximum-xor-of-two-numbers-in-an-array/
题目大意:
给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n。
#
进阶：你可以在 O(n) 的时间解决这个问题吗？
#
示例 1:
输入: nums = [3,10,5,25,2,8]
输出: 28
解释: 最大的结果是 5 XOR 25 = 28.

```

```

示例 2:
输入: nums = [0]
输出: 0

示例 3:
输入: nums = [2, 4]
输出: 6

示例 4:
输入: nums = [8, 10, 2]
输出: 10

示例 5:
输入: nums = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]
输出: 127

提示:
1 <= nums.length <= 2 * 10^4
0 <= nums[i] <= 2^31 - 1

解题思路:
方法一: 暴力解法 ($O(n^2)$ 时间复杂度, 不推荐)
方法二: 前缀树(字典树)优化的位运算方法 ($O(n)$ 时间复杂度)
方法三: 基于异或性质的位运算方法 ($O(n)$ 时间复杂度)

这里我们主要实现方法二和方法三, 它们都是基于位运算的高效解法
```

# 方法一: 暴力解法

```
def find_maximum_xor_1(nums: list) -> int:
```

```
 """
```

使用暴力解法计算数组中两个数的最大异或值

参数:

nums: 整数数组

返回:

int: 最大异或结果

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

```
"""
```

# 参数验证和边界条件检查

```

如果数组为空或只有一个元素，最大异或值为 0
if not nums or len(nums) <= 1:
 return 0

初始化最大异或结果为 0
max_result = 0

遍历所有可能的数对
外层循环遍历第一个数
for i in range(len(nums)):
 # 内层循环遍历第二个数 (j > i 避免重复计算和自己与自己异或)
 for j in range(i + 1, len(nums)):
 # 计算当前两个数的异或值
 # ^ 表示按位异或操作
 current_xor = nums[i] ^ nums[j]
 # 更新最大异或结果
 # max 返回两个数中的较大值
 max_result = max(max_result, current_xor)

返回最大异或结果
return max_result

```

# 方法二：基于位运算和集合的方法

```

def find_maximum_xor_2(nums: list) -> int:
 """

```

使用位运算和集合优化计算数组中两个数的最大异或值

参数：

nums：整数数组

返回：

int：最大异或结果

时间复杂度：O(n)

空间复杂度：O(n)

"""

# 参数验证和边界条件检查

# 如果数组为空或只有一个元素，最大异或值为 0

```

if not nums or len(nums) <= 1:
 return 0

```

# 初始化最大异或结果为 0

```

max_result = 0

```

```

初始化掩码为 0，用于提取数字的前缀
mask = 0

从最高位到最低位依次确定结果的每一位
31 位整数，但符号位为 0（因为题目中 nums[i] >= 0），所以从第 30 位开始到第 0 位
for i in range(30, -1, -1):
 # 构建当前位的掩码
 # mask |= (1 << i) 将 mask 的第 i 位设置为 1
 # 这样 mask 就包含了从最高位到当前位的所有位
 mask |= (1 << i)

 # 存储所有数在当前掩码下的前缀
 # 使用 set 存储前缀，便于快速查找
 prefixes = set()
 # 遍历数组中的每个数
 for num in nums:
 # num & mask 提取 num 在当前掩码下的前缀
 # 例如：如果 mask 是 11100000，那么 num & mask 就提取 num 的高 3 位
 prefixes.add(num & mask)

 # 假设当前位为 1，构造可能的最大值
 # max_result | (1 << i) 将 max_result 的第 i 位设置为 1
 # 这是我们希望得到的最大异或结果
 temp_max = max_result | (1 << i)

 # 检查是否存在两个数，它们的前缀异或等于 temp_max
 # 遍历所有前缀
 for prefix in prefixes:
 # 如果 prefix ^ target = temp_max，那么 target = prefix ^ temp_max
 # 这是基于异或运算的性质：如果 a ^ b = c，那么 a ^ c = b
 # 我们要找是否存在另一个前缀 target，使得 prefix ^ target = temp_max
 if (prefix ^ temp_max) in prefixes:
 # 找到可行的解，说明最大异或结果的当前位可以为 1
 # 设置当前位为 1
 max_result = temp_max
 # 找到解后跳出循环，继续处理下一位
 break

 # 如果没有找到，当前位保持为 0 (max_result 不变)
 # 这是因为我们初始化 temp_max 时已经将 max_result 的当前位设为 1
 # 如果找不到匹配的前缀对，说明当前位必须为 0

返回最大异或结果

```

```
return max_result

方法三：前缀树（字典树）方法
class TrieNode:
 """前缀树节点类"""
 def __init__(self):
 # 0 和 1 两个子节点
 # children[0] 表示当前位为 0 的子节点
 # children[1] 表示当前位为 1 的子节点
 self.children = [None, None] # type: list[TrieNode | None]
```

```
def find_maximum_xor_3(nums: list) -> int:
```

```
 """
 使用前缀树计算数组中两个数的最大异或值
```

参数：

nums：整数数组

返回：

int：最大异或结果

时间复杂度：O(n)

空间复杂度：O(n)

```
"""
```

```
参数验证和边界条件检查
```

```
如果数组为空或只有一个元素，最大异或值为 0
```

```
if not nums or len(nums) <= 1:
```

```
 return 0
```

```
构建前缀树
```

```
创建前缀树的根节点
```

```
root = TrieNode()
```

```
向前缀树中插入一个数
```

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

```
 # 从根节点开始
```

```
 node = root
```

```
 # 从最高位到最低位插入
```

```
 # 31 位整数，忽略符号位，所以从第 30 位开始到第 0 位
```

```
 for i in range(30, -1, -1):
```

```
 # 提取 num 的第 i 位
```

```
 # (num >> i) & 1 将 num 右移 i 位，然后与 1 进行按位与操作
```

```
 # 这样可以提取 num 的第 i 位 (0 或 1)
```

```

bit = (num >> i) & 1
如果当前位对应的子节点不存在，则创建新节点
if not node.children[bit]:
 node.children[bit] = TrieNode()
移动到子节点
node = node.children[bit]

在已有的前缀树中查找与给定数异或结果最大的数
def search(num):
 # 从根节点开始
 node = root
 # 初始化异或结果为 0
 xor = 0
 # 从最高位到最低位查找
 # 31 位整数，忽略符号位，所以从第 30 位开始到第 0 位
 for i in range(30, -1, -1):
 # 提取 num 的第 i 位
 bit = (num >> i) & 1
 # 寻找相反的位以最大化异或结果
 # 1 - bit 可以得到 bit 的相反值（0 变 1，1 变 0）
 desired_bit = 1 - bit

 # 如果存在相反的位
 if node.children[desired_bit]:
 # 可以找到相反的位，异或结果的当前位为 1
 # xor |= (1 << i) 将 xor 的第 i 位设置为 1
 xor |= (1 << i)
 # 移动到相反位对应的子节点
 node = node.children[desired_bit]
 else:
 # 找不到相反的位，只能使用相同的位
 # 检查相同位对应的子节点是否存在
 if node.children[bit]: # 确保子节点存在
 # 移动到相同位对应的子节点
 node = node.children[bit]
 else:
 # 如果子节点不存在，提前结束
 break
 # 返回异或结果
 return xor

先插入第一个数到前缀树中
insert(nums[0])

```

```
初始化最大异或结果为 0
max_result = 0

对于每个数，插入并查找能产生最大异或值的数
从第二个数开始处理
for num in nums[1:]:
 # 在前缀树中查找与 num 异或结果最大的数
 # max 更新最大异或结果
 max_result = max(max_result, search(num))
 # 将当前数插入到前缀树中
 insert(num)

返回最大异或结果
return max_result

方法四：工程化版本，增加异常处理和参数验证
def find_maximum_xor_with_validation(nums: list) -> int:
 """
 工程化版本，增加异常处理和参数验证
 """

 参数：
 nums: 整数数组

 返回：
 int: 最大异或结果

 时间复杂度: O(n)
 空间复杂度: O(n)
 """

```

```
try:
 # 参数验证
 # 检查输入是否为列表类型
 if not isinstance(nums, list):
 # 抛出类型错误异常
 raise TypeError("Input must be a list")

 # 边界情况处理
 # 检查数组长度
 if len(nums) <= 1:
 # 如果数组只有一个元素或为空，最大异或值为 0
 return 0
```

```

验证所有元素是否为非负整数
for num in nums:
 # 检查元素是否为整数且非负
 if not isinstance(num, int) or num < 0:
 # 抛出值错误异常
 raise ValueError("All elements must be non-negative integers")

使用方法二实现
调用 find_maximum_xor_2 方法计算最大异或值
return find_maximum_xor_2(nums)

except Exception as e:
 # 记录异常（在实际应用中可以使用日志）
 # 在生产环境中，应该使用日志框架记录异常
 print(f"Error in find_maximum_xor_with_validation: {e}")
 # 异常情况下返回 0
 return 0

单元测试
def run_tests():
 """
 运行单元测试
 """

 # 定义测试用例
 # 每个测试用例是一个元组，包含输入数组和期望的输出结果
 test_cases = [
 ([3, 10, 5, 25, 2, 8], 28), # 示例 1
 ([0], 0), # 示例 2
 ([2, 4], 6), # 示例 3
 ([8, 10, 2], 10), # 示例 4
 ([14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70], 127) # 示例 5
]

 print("Running unit tests...\n")

 # 定义所有测试方法
 methods = [
 (find_maximum_xor_1, "Method 1 (Brute Force)"),
 (find_maximum_xor_2, "Method 2 (Bitwise with Set)"),
 (find_maximum_xor_3, "Method 3 (Trie)"),
 (find_maximum_xor_with_validation, "Method 4 (With Validation)")
]

 # 测试所有方法

```

```
for method, method_name in methods:
 print(f"Testing {method_name}:")
 # 遍历所有测试用例
 for nums, expected in test_cases:
 # 调用被测试的方法
 result = method(nums)
 # 判断测试结果是否正确
 status = "PASS" if result == expected else "FAIL"
 # 输出测试结果
 print(f" {nums} -> {result} (Expected: {expected}) - {status}")
 print()

性能测试
def performance_test():
 """
 性能测试
 """
 import time

 # 生成大规模测试数据
 # 创建一个包含 0 到 9999 的整数数组
 large_nums = [i for i in range(10000)]

 # 仅在小规模数据上测试方法 1 (避免超时)
 small_nums = [3, 10, 5, 25, 2, 8]
 # 记录开始时间
 start_time = time.time()
 # 调用方法 1 计算结果
 result1 = find_maximum_xor_1(small_nums)
 # 记录结束时间
 end_time = time.time()
 print(f"Performance of Method 1 (Brute Force): {((end_time - start_time) * 1000):.2f} ms")
 print(f"Result: {result1}\n")

 # 测试方法 2
 # 记录开始时间
 start_time = time.time()
 # 调用方法 2 计算结果
 result2 = find_maximum_xor_2(large_nums)
 # 记录结束时间
 end_time = time.time()
 print(f"Performance of Method 2 (Bitwise with Set): {((end_time - start_time) * 1000):.2f} ms")
```

```
print(f"Result: {result2}\n")

测试方法 3
记录开始时间
start_time = time.time()
调用方法 3 计算结果
result3 = find_maximum_xor_3(large_nums)
记录结束时间
end_time = time.time()
print(f"Performance of Method 3 (Trie): {((end_time - start_time) * 1000):.2f} ms")
print(f"Result: {result3}\n")

测试方法 4
记录开始时间
start_time = time.time()
调用方法 4 计算结果
result4 = find_maximum_xor_with_validation(large_nums)
记录结束时间
end_time = time.time()
print(f"Performance of Method 4 (With Validation): {((end_time - start_time) * 1000):.2f} ms")
print(f"Result: {result4}\n")

主函数
if __name__ == "__main__":
 print("LeetCode 421. 数组中两个数的最大异或值")
 print("使用位运算优化实现\n")

运行单元测试
run_tests()

运行性能测试
performance_test()

复杂度分析
print("复杂度分析:")
print("方法一（暴力解法）:")
print(" 时间复杂度: O(n2)，其中 n 是数组长度")
print(" 空间复杂度: O(1)")
print(" 优点: 实现简单")
print(" 缺点: 对于大数组效率低")

print("\n方法二（基于位运算和集合）:")
```

```
print(" 时间复杂度: O(n)，每个位处理需要 O(n)时间，总共处理 32 个位")
print(" 空间复杂度: O(n)，用于存储前缀集合")
print(" 优点: 时间效率高，实现相对简单")

print("\n 方法三（前缀树）:")
print(" 时间复杂度: O(n)，构建树和查询都是 O(n)时间")
print(" 空间复杂度: O(n)，用于存储前缀树")
print(" 优点: 位操作思想清晰，扩展性好")

print("\n 适用场景总结:")
print("1. 对于小数组，可以使用暴力解法")
print("2. 对于大数组，应使用方法二或方法三，它们的时间复杂度都是 O(n) ")
print("3. 在工程实践中，方法二实现更简洁，而方法三更能体现位运算的思想")
print("4. 当需要处理大量相似查询时，前缀树方法更具优势")
```

=====

文件: Code11\_MaxLength.cpp

```
// LeetCode 1239. 串联字符串的最大长度
// 题目链接: https://leetcode-cn.com/problems/maximum-length-of-a-concatenated-string-with-
unique-characters/
// 题目大意:
// 给定一个字符串数组 arr，字符串 s 是将 arr 某一子序列字符串连接所得的字符串，如果 s 中的每个字
符都只出现过一次，
// 请返回所有可能的 s 中最长长度。
//
// 示例 1:
// 输入: arr = ["un", "iq", "ue"]
// 输出: 4
// 解释: 所有可能的串联组合是 "", "un", "iq", "ue", "uniq" 和 "ique"，最大长度为 4。
//
// 示例 2:
// 输入: arr = ["cha", "r", "act", "ers"]
// 输出: 6
// 解释: 可能的解答是 "chaers"，长度为 6。
//
// 示例 3:
// 输入: arr = ["abcdefghijklmnopqrstuvwxyz"]
// 输出: 26
//
// 提示:
// 1 <= arr.length <= 16
```

```
// 1 <= arr[i].length <= 26
// arr[i] 中只含有小写英文字母
//
// 解题思路：
// 由于题目中要求字符串中的每个字符只出现一次，我们可以使用位掩码来表示每个字符串中包含的字符。
// 对于长度为 16 的数组，我们可以使用回溯算法或动态规划来求解最长长度。
// 这里我们使用位运算优化，通过掩码来判断字符是否重复。
```

#

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
#include <unordered_set>
#include <regex>
```

```
// 方法一：回溯算法 + 位运算
```

```
int maxLength1(const std::vector<std::string>& arr) {
 // 边界条件检查
 if (arr.empty()) {
 return 0;
 }
```

```
// 过滤掉自身包含重复字符的字符串，并将其转换为位掩码
```

```
std::vector<int> masks;
for (const std::string& s : arr) {
 int mask = 0;
 bool is_valid = true;

 for (char c : s) {
 int bit = 1 << (c - 'a');
 // 检查当前字符是否已经在 mask 中设置
 if (mask & bit) {
 is_valid = false;
 break;
 }
 mask |= bit;
 }
}
```

```
if (is_valid) {
 masks.push_back(mask);
}
```

```

// 回溯函数
std::function<int(int, int)> backtrack = [&](int index, int current_mask) {
 // 基本情况：已经处理完所有字符串
 if (index == masks.size()) {
 // 计算 current_mask 中设置的位的数量，即字符数量
 return __builtin_popcount(current_mask);
 }

 // 不选当前字符串
 int max_len = backtrack(index + 1, current_mask);

 // 选当前字符串（如果不会导致重复字符）
 int current_mask_val = masks[index];
 if ((current_mask & current_mask_val) == 0) { // 没有共同的字符
 max_len = std::max(max_len, backtrack(index + 1, current_mask | current_mask_val));
 }

 return max_len;
};

// 调用回溯函数，从索引 0 和空掩码开始
return backtrack(0, 0);
}

```

```

// 方法二：动态规划 + 位运算
int maxLength2(const std::vector<std::string>& arr) {
 // 边界条件检查
 if (arr.empty()) {
 return 0;
 }

 // 过滤掉自身包含重复字符的字符串，并将其转换为位掩码
 std::vector<int> valid_masks;
 for (const std::string& s : arr) {
 int mask = 0;
 bool is_valid = true;

 for (char c : s) {
 int bit = 1 << (c - 'a');
 if (mask & bit) {
 is_valid = false;
 break;
 }
 mask |= bit;
 }
 if (is_valid)
 valid_masks.push_back(mask);
 }
}

// 根据位掩码计算字符串长度
int calculateLength(int mask) {
 int count = 0;
 while (mask != 0) {
 if (mask & 1)
 count++;
 mask >>= 1;
 }
 return count;
}

// 动态规划
int dp[1000][1000];
int dpLength2(const std::vector<std::string>& arr) {
 int n = arr.size();
 if (n == 0)
 return 0;
 if (n == 1)
 return calculateLength(arr[0]);
 if (dp[0][0] != -1)
 return dp[0][0];

 dp[0][0] = calculateLength(arr[0]);
 for (int i = 1; i < n; i++) {
 dp[i][0] = calculateLength(arr[i]);
 for (int j = 1; j < i; j++) {
 int current_mask = arr[i] | arr[j];
 dp[i][j] = dp[i-1][j] + (current_mask == arr[i] ? 1 : 0);
 if (current_mask == arr[i])
 dp[i][j] += calculateLength(arr[i]);
 }
 }
 return dp[n-1][n-1];
}

```

```

 }
 mask |= bit;
 }

 if (is_valid && mask != 0) { // 确保 mask 不为 0 (空字符串被过滤)
 valid_masks.push_back(mask);
 }
}

// 动态规划: dp 存储所有可能的有效掩码组合
std::unordered_set<int> dp;
dp.insert(0); // 初始状态: 空字符串
int max_len = 0;

// 对于每个有效的字符串掩码
for (int mask : valid_masks) {
 // 创建一个临时集合, 存储新的组合
 std::unordered_set<int> temp;

 // 对于当前的所有组合
 for (int existing_mask : dp) {
 // 如果当前 mask 和已有 mask 没有重叠的位
 if ((existing_mask & mask) == 0) {
 int combined_mask = existing_mask | mask;
 temp.insert(combined_mask);
 max_len = std::max(max_len, __builtin_popcount(combined_mask));
 }
 }
}

// 将新的组合添加到 dp 中
dp.insert(temp.begin(), temp.end());
}

return max_len;
}

// 方法三: 优化的回溯算法
int maxLength3(const std::vector<std::string>& arr) {
 // 过滤无效字符串并转换为掩码
 std::vector<int> masks;
 std::vector<int> lengths;
 for (const std::string& s : arr) {
 int mask = 0;

```

```

bool valid = true;

for (char c : s) {
 int bit = 1 << (c - 'a');
 if (mask & bit) {
 valid = false;
 break;
 }
 mask |= bit;
}

if (valid) {
 masks.push_back(mask);
 lengths.push_back(__builtin_popcount(mask));
}
}

// 存储最大长度
int max_len = 0;

// 优化的回溯函数
std::function<void(int, int, int)> optimized_backtrack =
 [&](int index, int current_mask, int current_length) {
 // 更新最大长度
 if (current_length > max_len) {
 max_len = current_length;
 }

 // 剪枝：如果剩余的字符串即使全部选上也无法超过当前最大长度，提前返回
 int remaining_max_length = current_length;
 for (int i = index; i < masks.size(); i++) {
 if ((current_mask & masks[i]) == 0) {
 remaining_max_length += lengths[i];
 }
 }

 if (remaining_max_length <= max_len) {
 return; // 剪枝
 }

 // 回溯
 for (int i = index; i < masks.size(); i++) {
 if ((current_mask & masks[i]) == 0) { // 没有共同的字符

```

```

 optimized_backtrack(i + 1,
 current_mask | masks[i],
 current_length + lengths[i]);
 }
 }
 } ;

// 调用优化的回溯函数
optimized_backtrack(0, 0, 0);
return max_len;
}

// 方法四：工程化改进版本，增加参数验证和异常处理
int maxLengthWithValidation(const std::vector<std::string>& arr) {
 try {
 // 参数验证
 // 在 C++ 中，不需要检查 arr 是否为 nullptr，因为 vector 是值类型

 // 检查数组长度是否在题目限制范围内
 if (arr.size() > 16) {
 throw std::invalid_argument("Input list size exceeds maximum allowed length of 16");
 }

 // 验证每个字符串是否符合要求
 std::regex lowercase_regex("^*[a-z]*$");
 for (const std::string& s : arr) {
 if (s.length() > 26) {
 throw std::invalid_argument("String element exceeds maximum allowed length of 26");
 }
 // 检查是否只包含小写英文字母
 if (!std::regex_match(s, lowercase_regex)) {
 throw std::invalid_argument("String elements must contain only lowercase English letters");
 }
 }

 // 使用方法三实现
 return maxLength3(arr);
 } catch (const std::exception& e) {
 // 记录异常日志（在实际应用中）
 std::cerr << "Error in maxLengthWithValidation: " << e.what() << std::endl;
 // 异常情况下返回 0
 }
}

```

```

 return 0;
}

}

// 单元测试函数
void runTests() {
 std::cout << "Running unit tests..." << std::endl;

 struct TestCase {
 std::vector<std::string> input;
 int expected;
 };

 std::vector<TestCase> testCases = {
 {"un", "iq", "ue"}, 4},
 {"cha", "r", "act", "ers"}, 6},
 {"abcdefghijklmnopqrstuvwxyz"}, 26},
 {"abc", "def", "a", "ghi", "abb"}, 9}
 };
}

// 定义要测试的函数和名称
struct Method {
 int (*func)(const std::vector<std::string>&);
 std::string name;
};

std::vector<Method> methods = {
 {maxLength1, "Method 1 (Backtracking)" },
 {maxLength2, "Method 2 (Dynamic Programming)" },
 {maxLength3, "Method 3 (Optimized Backtracking)" },
 {maxLengthWithValidation, "Method 4 (With Validation)" }
};

// 测试所有方法
for (const auto& method : methods) {
 std::cout << "\n" << method.name << ":" << std::endl;
 for (const auto& test : testCases) {
 int result = method.func(test.input);
 const std::string& status = (result == test.expected) ? "PASS" : "FAIL";
 std::cout << " Input: [";
 for (size_t i = 0; i < test.input.size(); ++i) {
 std::cout << "\\" << test.input[i] << "\\";
 if (i < test.input.size() - 1) std::cout << ", ";
 }
 std::cout << "]";
 }
}

```

```

 }

 std::cout << "] -> Result: " << result
 << " (Expected: " << test.expected
 << ") - " << status << std::endl;
 }

}

}

// 性能测试函数
void performanceTest() {
 std::cout << "\nRunning performance tests..." << std::endl;

 // 生成测试数据：所有字母组合
 std::vector<std::string> largeArr;
 for (int i = 0; i < 10; i++) {
 // 生成不重复字符的字符串
 std::string s;
 for (int j = 0; j < 5 && (i * 5 + j) < 26; j++) {
 s += (char)('a' + i * 5 + j);
 }
 largeArr.push_back(s);
 }

 // 测试方法一时间
 auto start = std::chrono::high_resolution_clock::now();
 int result1 = maxLength1(largeArr);
 auto end = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
 std::cout << "\nPerformance of Method 1 (Backtracking): " << duration << " μs, Result: " << result1 << std::endl;

 // 测试方法二时间
 start = std::chrono::high_resolution_clock::now();
 int result2 = maxLength2(largeArr);
 end = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
 std::cout << "Performance of Method 2 (Dynamic Programming): " << duration << " μs, Result: "
 << result2 << std::endl;

 // 测试方法三时间
 start = std::chrono::high_resolution_clock::now();
 int result3 = maxLength3(largeArr);
 end = std::chrono::high_resolution_clock::now();

```

```

duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "Performance of Method 3 (Optimized Backtracking): " << duration << " μs,
Result: " << result3 << std::endl;

// 测试方法四时间
start = std::chrono::high_resolution_clock::now();
int result4 = maxLengthWithValidation(largeArr);
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "Performance of Method 4 (With Validation): " << duration << " μs, Result: " <<
result4 << std::endl;
}

// 主函数
int main() {
 std::cout << "LeetCode 1239. 串联字符串的最大长度" << std::endl;
 std::cout << "Using bitwise operations for optimization" << std::endl;

 // 运行单元测试
 runTests();

 // 运行性能测试
 performanceTest();

 // 复杂度分析
 std::cout << "\n复杂度分析:" << std::endl;
 std::cout << "方法一（回溯算法） :" << std::endl;
 std::cout << " 时间复杂度: $O(2^n)$, 其中 n 是有效字符串的数量（过滤后）" << std::endl;
 std::cout << " 空间复杂度: $O(n)$, 递归调用栈的深度" << std::endl;

 std::cout << "\n方法二（动态规划） :" << std::endl;
 std::cout << " 时间复杂度: $O(n * 2^m)$, 其中 n 是字符串数量, m 是字符集大小（最多 26）" <<
std::endl;
 std::cout << " 空间复杂度: $O(2^m)$, 存储所有可能的掩码组合" << std::endl;

 std::cout << "\n方法三（优化的回溯算法） :" << std::endl;
 std::cout << " 时间复杂度: $O(2^n)$, 但通过剪枝优化实际运行时间" << std::endl;
 std::cout << " 空间复杂度: $O(n)$ " << std::endl;
 std::cout << " 优点: 利用剪枝减少不必要的计算, 对于大多数情况效率更高" << std::endl;

 std::cout << "\n适用场景总结:" << std::endl;
 std::cout << "1. 当数组长度较小时, 三种方法都可以使用" << std::endl;
 std::cout << "2. 当字符串包含大量重复字符时, 方法三的剪枝效果更好" << std::endl;
}

```

```
 std::cout << "3. 当需要更稳定的性能时，方法二的动态规划更可靠" << std::endl;
 std::cout << "4. 在工程实践中，应根据具体数据特征选择合适的方法" << std::endl;

 return 0;
}
```

---

文件: Code11\_MaxLength.java

```
package class032;

import java.util.ArrayList;
import java.util.List;

// LeetCode 1239. 串联字符串的最大长度
// 题目链接: https://leetcode-cn.com/problems/maximum-length-of-a-concatenated-string-with-
unique-characters/
// 题目大意:
// 给定一个字符串数组 arr, 字符串 s 是将 arr 某一子序列字符串连接所得的字符串, 如果 s 中的每个字
// 符都只出现过一次,
// 请返回所有可能的 s 中最长长度。
//
// 示例 1:
// 输入: arr = ["un", "iq", "ue"]
// 输出: 4
// 解释: 所有可能的串联组合是 "", "un", "iq", "ue", "uniq" 和 "ique", 最大长度为 4。
//
// 示例 2:
// 输入: arr = ["cha", "r", "act", "ers"]
// 输出: 6
// 解释: 可能的解答是 "chaers", 长度为 6。
//
// 示例 3:
// 输入: arr = ["abcdefghijklmnopqrstuvwxyz"]
// 输出: 26
//
// 提示:
// 1 <= arr.length <= 16
// 1 <= arr[i].length <= 26
// arr[i] 中只含有小写英文字母
//
// 解题思路:
```

```
// 由于题目中要求字符串中的每个字符只出现一次，我们可以使用位掩码来表示每个字符串中包含的字符。
// 对于长度为 16 的数组，我们可以使用回溯算法或动态规划来求解最长长度。
// 这里我们使用位运算优化，通过掩码来判断字符是否重复。
```

```
public class Code11_MaxLength {

 // 方法一：回溯算法 + 位运算
 public static int maxLength1(List<String> arr) {
 // 边界条件检查
 if (arr == null || arr.isEmpty()) {
 return 0;
 }

 // 过滤掉自身包含重复字符的字符串
 List<Integer> masks = new ArrayList<>();
 for (String s : arr) {
 int mask = 0;
 boolean isValid = true;

 for (char c : s.toCharArray()) {
 int bit = 1 << (c - 'a');
 // 检查当前字符是否已经在 mask 中设置
 if ((mask & bit) != 0) {
 isValid = false;
 break;
 }
 mask |= bit;
 }

 if (isValid) {
 masks.add(mask);
 }
 }

 // 调用回溯函数
 return backtrack(masks, 0, 0);
 }

 // 回溯函数
 // masks: 所有有效字符串的位掩码列表
 // index: 当前处理到的字符串索引
 // currentMask: 当前已选字符串的位掩码
 private static int backtrack(List<Integer> masks, int index, int currentMask) {
```

```

// 基本情况：已经处理完所有字符串
if (index == masks.size()) {
 // 计算 currentMask 中设置的位的数量，即字符数量
 return Integer.bitCount(currentMask);
}

// 不选当前字符串
int maxLength = backtrack(masks, index + 1, currentMask);

// 选当前字符串（如果不会导致重复字符）
int currentMaskVal = masks.get(index);
if ((currentMask & currentMaskVal) == 0) { // 没有共同的字符
 maxLength = Math.max(maxLength, backtrack(masks, index + 1, currentMask |
currentMaskVal));
}

return maxLength;
}

// 方法二：动态规划 + 位运算
public static int maxLength2(List<String> arr) {
 // 边界条件检查
 if (arr == null || arr.isEmpty()) {
 return 0;
 }

 // 过滤掉自身包含重复字符的字符串，并转换为位掩码
 List<Integer> validMasks = new ArrayList<>();
 for (String s : arr) {
 int mask = 0;
 boolean isValid = true;

 for (char c : s.toCharArray()) {
 int bit = 1 << (c - 'a');
 if ((mask & bit) != 0) {
 isValid = false;
 break;
 }
 mask |= bit;
 }

 if (isValid && mask != 0) { // 确保 mask 不为 0 (空字符串被过滤)
 validMasks.add(mask);
 }
 }
}

```

```

 }

}

// 动态规划: dp 存储所有可能的有效掩码组合
List<Integer> dp = new ArrayList<>();
dp.add(0); // 初始状态: 空字符串

int maxLen = 0;

// 对于每个有效的字符串掩码
for (int mask : validMasks) {
 // 创建一个临时列表, 存储新的组合
 List<Integer> temp = new ArrayList<>();

 // 对于当前的所有组合
 for (int existingMask : dp) {
 // 如果当前 mask 和已有 mask 没有重叠的位
 if ((existingMask & mask) == 0) {
 int combinedMask = existingMask | mask;
 temp.add(combinedMask);
 maxLen = Math.max(maxLen, Integer.bitCount(combinedMask));
 }
 }
}

// 将新的组合添加到 dp 中
dp.addAll(temp);

}

return maxLen;
}

// 方法三: 优化的回溯算法
public static int maxLength3(List<String> arr) {
 // 预处理: 过滤无效字符串并转换为掩码
 List<Integer> masks = new ArrayList<>();
 for (String s : arr) {
 int mask = 0;
 boolean valid = true;

 for (char c : s.toCharArray()) {
 int bit = 1 << (c - 'a');
 if ((mask & bit) != 0) {
 valid = false;
 }
 }
 if (valid) {
 masks.add(mask);
 }
 }
 return maxLength2(masks);
}

```

```

 break;
 }
 mask |= bit;
}

if (valid) {
 masks.add(mask);
}
}

// 创建长度数组，避免重复计算 bitCount
int[] lengths = new int[masks.size()];
for (int i = 0; i < masks.size(); i++) {
 lengths[i] = Integer.bitCount(masks.get(i));
}

// 优化的回溯函数
int[] result = {0}; // 使用数组作为引用传递
optimizedBacktrack(masks, lengths, 0, 0, 0, result);
return result[0];
}

private static void optimizedBacktrack(List<Integer> masks, int[] lengths, int index,
 int currentMask, int currentLength, int[] result) {
 // 更新最大长度
 result[0] = Math.max(result[0], currentLength);

 // 剪枝：如果剩余的字符串即使全部选上也无法超过当前最大长度，提前返回
 int remainingMaxLength = currentLength;
 for (int i = index; i < masks.size(); i++) {
 if ((currentMask & masks.get(i)) == 0) {
 remainingMaxLength += lengths[i];
 }
 }

 if (remainingMaxLength <= result[0]) {
 return; // 剪枝
 }

 // 回溯
 for (int i = index; i < masks.size(); i++) {
 if ((currentMask & masks.get(i)) == 0) { // 没有共同的字符
 optimizedBacktrack(masks, lengths, i + 1,

```

```
 currentMask | masks.get(i),
 currentLength + lengths[i], result);
 }
}
}

// 工程化改进版本：增加参数验证和异常处理
public static int maxLengthWithValidation(List<String> arr) {
 try {
 // 参数验证
 if (arr == null) {
 throw new IllegalArgumentException("Input list cannot be null");
 }

 // 检查数组长度是否在题目限制范围内
 if (arr.size() > 16) {
 throw new IllegalArgumentException("Input list size exceeds maximum allowed length of 16");
 }

 // 验证每个字符串是否符合要求
 for (String s : arr) {
 if (s == null) {
 throw new IllegalArgumentException("String elements cannot be null");
 }
 if (s.length() > 26) {
 throw new IllegalArgumentException("String element exceeds maximum allowed length of 26");
 }
 // 检查是否只包含小写英文字母
 if (!s.matches("[a-z]+")) {
 throw new IllegalArgumentException("String elements must contain only lowercase English letters");
 }
 }
 }

 // 使用方法三实现
 return maxLength3(arr);
} catch (Exception e) {
 // 记录异常日志（在实际应用中）
 System.err.println("Error in maxLength: " + e.getMessage());
 // 异常情况下返回 0
 return 0;
}
```

```

 }

}

// 单元测试方法
public static void runTests() {
 // 测试用例 1
 List<String> arr1 = new ArrayList<>();
 arr1.add("un");
 arr1.add("iq");
 arr1.add("ue");
 System.out.println("Test 1: [\"un\", \"iq\", \"ue\"] -> " + maxLength2(arr1) + " (Expected: 4)");
}

// 测试用例 2
List<String> arr2 = new ArrayList<>();
arr2.add("cha");
arr2.add("r");
arr2.add("act");
arr2.add("ers");
System.out.println("Test 2: [\"cha\", \"r\", \"act\", \"ers\"] -> " + maxLength2(arr2) + " (Expected: 6)");

// 测试用例 3
List<String> arr3 = new ArrayList<>();
arr3.add("abcdefghijklmnopqrstuvwxyz");
System.out.println("Test 3: [\"abcdefghijklmnopqrstuvwxyz\"] -> " + maxLength2(arr3) + " (Expected: 26)");

// 测试用例 4: 包含无效字符串（自身有重复字符）
List<String> arr4 = new ArrayList<>();
arr4.add("abc");
arr4.add("def");
arr4.add("a");
arr4.add("ghi");
arr4.add("abb"); // 自身有重复字符，应被过滤
System.out.println("Test 4: [\"abc\", \"def\", \"a\", \"ghi\", \"abb\"] -> " + maxLength2(arr4) + " (Expected: 9)");

// 测试不同方法的结果一致性
System.out.println("\nMethod comparison:");
System.out.println("Test 1 results:");
System.out.println(" maxLength1: " + maxLength1(arr1));
System.out.println(" maxLength2: " + maxLength2(arr1));

```

```

 System.out.println("maxLength3: " + maxLength3(arr1));
 System.out.println("maxLengthWithValidation: " + maxLengthWithValidation(arr1));
 }

// 性能测试方法
public static void performanceTest() {
 // 生成测试数据：所有字母组合
 List<String> largeArr = new ArrayList<>();
 for (int i = 0; i < 10; i++) {
 // 生成不重复字符的字符串
 StringBuilder sb = new StringBuilder();
 for (int j = 0; j < 5 && (i * 5 + j) < 26; j++) {
 sb.append((char) ('a' + i * 5 + j));
 }
 largeArr.add(sb.toString());
 }
}

// 测试方法一时间
long startTime = System.nanoTime();
int result1 = maxLength1(largeArr);
long endTime = System.nanoTime();
System.out.println("\nPerformance of maxLength1: " +
 (endTime - startTime) / 1000 + " μs, Result: " + result1);

// 测试方法二时间
startTime = System.nanoTime();
int result2 = maxLength2(largeArr);
endTime = System.nanoTime();
System.out.println("Performance of maxLength2: " +
 (endTime - startTime) / 1000 + " μs, Result: " + result2);

// 测试方法三时间
startTime = System.nanoTime();
int result3 = maxLength3(largeArr);
endTime = System.nanoTime();
System.out.println("Performance of maxLength3: " +
 (endTime - startTime) / 1000 + " μs, Result: " + result3);
}

public static void main(String[] args) {
 System.out.println("LeetCode 1239. 串联字符串的最大长度");
 System.out.println("使用位运算优化实现");
}

```

```

// 运行单元测试
runTests();

// 运行性能测试
performanceTest();

// 复杂度分析
System.out.println("\n 复杂度分析:");
System.out.println("方法一（回溯算法）:");
System.out.println(" 时间复杂度: O(2^n)，其中 n 是有效字符串的数量（过滤后）");
System.out.println(" 空间复杂度: O(n)，递归调用栈的深度");

System.out.println("\n 方法二（动态规划）:");
System.out.println(" 时间复杂度: O(n * 2^m)，其中 n 是字符串数量，m 是字符集大小（最多 26）");
System.out.println(" 空间复杂度: O(2^m)，存储所有可能的掩码组合");

System.out.println("\n 方法三（优化的回溯算法）:");
System.out.println(" 时间复杂度: O(2^n)，但通过剪枝优化实际运行时间");
System.out.println(" 空间复杂度: O(n)");
System.out.println(" 优点: 利用剪枝减少不必要的计算，对于大多数情况效率更高");

System.out.println("\n 适用场景总结:");
System.out.println("1. 当数组长度较小时，三种方法都可以使用");
System.out.println("2. 当字符串包含大量重复字符时，方法三的剪枝效果更好");
System.out.println("3. 当需要更稳定的性能时，方法二的动态规划更可靠");
System.out.println("4. 在工程实践中，应根据具体数据特征选择合适的方法");
}

}

```

文件: Code11\_MaxLength.py

```

LeetCode 1239. 串联字符串的最大长度
题目链接: https://leetcode-cn.com/problems/maximum-length-of-a-concatenated-string-with-unique-
characters/
题目大意:
给定一个字符串数组 arr，字符串 s 是将 arr 某一子序列字符串连接所得的字符串，如果 s 中的每个字符
都只出现过一次，
请返回所有可能的 s 中最长长度。
#
示例 1:

```

```
输入: arr = ["un", "iq", "ue"]
输出: 4
解释: 所有可能的串联组合是 "", "un", "iq", "ue", "uniq" 和 "ique", 最大长度为 4。
#
示例 2:
输入: arr = ["cha", "r", "act", "ers"]
输出: 6
解释: 可能的解答是 "chaers", 长度为 6。
#
示例 3:
输入: arr = ["abcdefghijklmnopqrstuvwxyz"]
输出: 26
#
提示:
1 <= arr.length <= 16
1 <= arr[i].length <= 26
arr[i] 中只含有小写英文字母
#
解题思路:
由于题目中要求字符串中的每个字符只出现一次, 我们可以使用位掩码来表示每个字符串中包含的字符。
对于长度为 16 的数组, 我们可以使用回溯算法或动态规划来求解最长长度。
这里我们使用位运算优化, 通过掩码来判断字符是否重复。
```

```
import time
import re

方法一: 回溯算法 + 位运算
def max_length1(arr):
 """
 使用回溯算法和位运算求解串联字符串的最大长度
 """

 Args:
```

arr: 字符串列表

Returns:

满足条件的最长字符串长度

时间复杂度:  $O(2^n)$ , 其中  $n$  是有效字符串的数量

空间复杂度:  $O(n)$ , 递归调用栈的深度

"""
# 边界条件检查
if not arr:
 return 0

```
过滤掉自身包含重复字符的字符串，并将其转换为位掩码
masks = []
for s in arr:
 mask = 0
 is_valid = True

 for c in s:
 bit = 1 << (ord(c) - ord('a'))
 # 检查当前字符是否已经在 mask 中设置
 if mask & bit:
 is_valid = False
 break
 mask |= bit

 if is_valid:
 masks.append(mask)
```

```
定义回溯函数
def backtrack(index, current_mask):
 """
 回溯函数，探索所有可能的组合
```

Args:

index: 当前处理到的字符串索引  
current\_mask: 当前已选字符串的位掩码

Returns:

从当前状态开始能得到的最长字符串长度

# 基本情况：已经处理完所有字符串

```
if index == len(masks):
 # 计算 current_mask 中设置的位的数量，即字符数量
 return bin(current_mask).count('1')
```

# 不选当前字符串

```
max_len = backtrack(index + 1, current_mask)
```

# 选当前字符串（如果不会导致重复字符）

```
current_mask_val = masks[index]
if (current_mask & current_mask_val) == 0: # 没有共同的字符
 max_len = max(max_len, backtrack(index + 1, current_mask | current_mask_val))
```

```

 return max_len

调用回溯函数，从索引 0 和空掩码开始
return backtrack(0, 0)

方法二：动态规划 + 位运算
def max_length2(arr):
 """
 使用动态规划和位运算求解串联字符串的最大长度

 Args:
 arr: 字符串列表

 Returns:
 满足条件的最长字符串长度

 时间复杂度: O(n * 2^m)，其中 n 是字符串数量，m 是字符集大小（最多 26）
 空间复杂度: O(2^m)，存储所有可能的掩码组合
 """
 # 边界条件检查
 if not arr:
 return 0

 # 过滤掉自身包含重复字符的字符串，并将其转换为位掩码
 valid_masks = []
 for s in arr:
 mask = 0
 is_valid = True

 for c in s:
 bit = 1 << (ord(c) - ord('a'))
 if mask & bit:
 is_valid = False
 break
 mask |= bit

 if is_valid and mask != 0: # 确保 mask 不为 0 (空字符串被过滤)
 valid_masks.append(mask)

 # 动态规划: dp 存储所有可能的有效掩码组合
 dp = [0] # 初始状态: 空字符串
 max_len = 0

```

```
对于每个有效的字符串掩码
for mask in valid_masks:
 # 对于当前的所有组合
 for i in range(len(dp)):
 existing_mask = dp[i]
 # 如果当前 mask 和已有 mask 没有重叠的位
 if (existing_mask & mask) == 0:
 combined_mask = existing_mask | mask
 # 检查是否已经在 dp 中，避免重复
 if combined_mask not in dp:
 dp.append(combined_mask)
 max_len = max(max_len, bin(combined_mask).count('1'))

return max_len
```

### # 方法三：优化的回溯算法

```
def max_length3(arr):
 """
 使用优化的回溯算法和位运算求解串联字符串的最大长度
 通过预计算长度和剪枝优化性能
 """

 使用优化的回溯算法和位运算求解串联字符串的最大长度
 通过预计算长度和剪枝优化性能
```

Args:

arr: 字符串列表

Returns:

满足条件的最长字符串长度

时间复杂度:  $O(2^n)$ , 但通过剪枝优化实际运行时间

空间复杂度:  $O(n)$

"""

### # 过滤无效字符串并转换为掩码

```
masks = []
lengths = []
for s in arr:
 mask = 0
 valid = True

 for c in s:
 bit = 1 << (ord(c) - ord('a'))
 if mask & bit:
 valid = False
 break
 mask |= bit
```

```

if valid:
 masks.append(mask)
 lengths.append(bin(mask).count('1'))

全局变量，用于存储最大长度
max_len = [0]

def optimized_backtrack(index, current_mask, current_length):
 """
 优化的回溯函数，包含剪枝

 Args:
 index: 当前处理到的字符串索引
 current_mask: 当前已选字符串的位掩码
 current_length: 当前已选字符串的总长度
 """
 # 更新最大长度
 if current_length > max_len[0]:
 max_len[0] = current_length

 # 剪枝：如果剩余的字符串即使全部选上也无法超过当前最大长度，提前返回
 remaining_max_length = current_length
 for i in range(index, len(masks)):
 if (current_mask & masks[i]) == 0:
 remaining_max_length += lengths[i]

 if remaining_max_length <= max_len[0]:
 return # 剪枝

 # 回溯
 for i in range(index, len(masks)):
 if (current_mask & masks[i]) == 0: # 没有共同的字符
 optimized_backtrack(i + 1,
 current_mask | masks[i],
 current_length + lengths[i])

 # 调用优化的回溯函数
 optimized_backtrack(0, 0, 0)
 return max_len[0]

工程化改进版本：增加参数验证和异常处理
def max_length_with_validation(arr):

```

```
"""
```

工程化版本，增加了参数验证和异常处理

Args:

arr: 字符串列表

Returns:

满足条件的最长字符串长度，如果输入无效则返回 0

Raises:

ValueError: 如果输入参数无效

```
"""
```

try:

# 参数验证

if arr is None:

```
 raise ValueError("Input list cannot be None")
```

# 检查数组长度是否在题目限制范围内

if len(arr) > 16:

```
 raise ValueError("Input list size exceeds maximum allowed length of 16")
```

# 验证每个字符串是否符合要求

for s in arr:

if s is None:

```
 raise ValueError("String elements cannot be None")
```

if len(s) > 26:

```
 raise ValueError("String element exceeds maximum allowed length of 26")
```

# 检查是否只包含小写英文字母

if not re.match(r'^[a-z]+\$', s):

```
 raise ValueError("String elements must contain only lowercase English letters")
```

# 使用方法三实现

```
return max_length3(arr)
```

except Exception as e:

# 记录异常日志（在实际应用中）

```
print(f"Error in max_length_with_validation: {e}")
```

# 异常情况下返回 0

```
return 0
```

# 单元测试函数

```
def run_tests():
```

```
"""
```

运行单元测试，验证不同方法的正确性

```
"""
```

```
print("Running unit tests...")
```

```
测试用例 1
```

```
arr1 = ["un", "iq", "ue"]
```

```
expected1 = 4
```

```
result1_1 = max_length1(arr1)
```

```
result1_2 = max_length2(arr1)
```

```
result1_3 = max_length3(arr1)
```

```
result1_4 = max_length_with_validation(arr1)
```

```
print(f"\nTest 1: ['un', 'iq', 'ue']")
```

```
print(f" max_length1: {result1_1} (Expected: {expected1}) - {'PASS' if result1_1 == expected1 else 'FAIL'}")
```

```
print(f" max_length2: {result1_2} (Expected: {expected1}) - {'PASS' if result1_2 == expected1 else 'FAIL'}")
```

```
print(f" max_length3: {result1_3} (Expected: {expected1}) - {'PASS' if result1_3 == expected1 else 'FAIL'}")
```

```
print(f" max_length_with_validation: {result1_4} (Expected: {expected1}) - {'PASS' if result1_4 == expected1 else 'FAIL'}")
```

```
测试用例 2
```

```
arr2 = ["cha", "r", "act", "ers"]
```

```
expected2 = 6
```

```
result2_1 = max_length1(arr2)
```

```
result2_2 = max_length2(arr2)
```

```
result2_3 = max_length3(arr2)
```

```
result2_4 = max_length_with_validation(arr2)
```

```
print(f"\nTest 2: ['cha', 'r', 'act', 'ers']")
```

```
print(f" max_length1: {result2_1} (Expected: {expected2}) - {'PASS' if result2_1 == expected2 else 'FAIL'}")
```

```
print(f" max_length2: {result2_2} (Expected: {expected2}) - {'PASS' if result2_2 == expected2 else 'FAIL'}")
```

```
print(f" max_length3: {result2_3} (Expected: {expected2}) - {'PASS' if result2_3 == expected2 else 'FAIL'}")
```

```
print(f" max_length_with_validation: {result2_4} (Expected: {expected2}) - {'PASS' if result2_4 == expected2 else 'FAIL'}")
```

```
测试用例 3
```

```
arr3 = ["abcdefghijklmnopqrstuvwxyz"]
```

```
expected3 = 26
```

```
result3_1 = max_length1(arr3)
```

```

result3_2 = max_length2(arr3)
result3_3 = max_length3(arr3)
result3_4 = max_length_with_validation(arr3)

print(f"\nTest 3: ['abcdefghijklmnopqrstuvwxyz']")
print(f" max_length1: {result3_1} (Expected: {expected3}) - {'PASS' if result3_1 == expected3 else 'FAIL'}")
print(f" max_length2: {result3_2} (Expected: {expected3}) - {'PASS' if result3_2 == expected3 else 'FAIL'}")
print(f" max_length3: {result3_3} (Expected: {expected3}) - {'PASS' if result3_3 == expected3 else 'FAIL'}")
print(f" max_length_with_validation: {result3_4} (Expected: {expected3}) - {'PASS' if result3_4 == expected3 else 'FAIL'}")

```

# 测试用例 4: 包含无效字符串（自身有重复字符）

```

arr4 = ["abc", "def", "a", "ghi", "abb"] # "abb"自身有重复字符，应被过滤
expected4 = 9 # "abc" + "def" + "ghi" 但包含重复的 'a'，所以实际是 "abc" + "def" + "ghi" 中的一部分

```

```

result4_1 = max_length1(arr4)
result4_2 = max_length2(arr4)
result4_3 = max_length3(arr4)
result4_4 = max_length_with_validation(arr4)

```

```

print(f"\nTest 4: ['abc', 'def', 'a', 'ghi', 'abb']")
print(f" max_length1: {result4_1} (Expected: {expected4}) - {'PASS' if result4_1 == expected4 else 'FAIL'}")
print(f" max_length2: {result4_2} (Expected: {expected4}) - {'PASS' if result4_2 == expected4 else 'FAIL'}")
print(f" max_length3: {result4_3} (Expected: {expected4}) - {'PASS' if result4_3 == expected4 else 'FAIL'}")
print(f" max_length_with_validation: {result4_4} (Expected: {expected4}) - {'PASS' if result4_4 == expected4 else 'FAIL'}")

```

# 性能测试函数

```

def performance_test():
 """
 运行性能测试，比较不同方法的效率
 """
 print("\nRunning performance tests...")

```

# 生成测试数据：所有字母组合

```

large_arr = []
for i in range(10):

```

```
生成不重复字符的字符串
s = ''.join([chr(ord('a') + i * 5 + j) for j in range(5) if i * 5 + j < 26])
large_arr.append(s)

测试方法一时间
start_time = time.time()
result1 = max_length1(large_arr)
end_time = time.time()
print(f"\nPerformance of max_length1: {((end_time - start_time) * 1_000_000):.2f} μs,
Result: {result1}")

测试方法二时间
start_time = time.time()
result2 = max_length2(large_arr)
end_time = time.time()
print(f"Performance of max_length2: {((end_time - start_time) * 1_000_000):.2f} μs, Result:
{result2}")

测试方法三时间
start_time = time.time()
result3 = max_length3(large_arr)
end_time = time.time()
print(f"Performance of max_length3: {((end_time - start_time) * 1_000_000):.2f} μs, Result:
{result3}")

主函数
def main():
 """
 主函数，运行测试并输出复杂度分析
 """
 print("LeetCode 1239. 串联字符串的最大长度")
 print("Using bitwise operations for optimization")

 # 运行单元测试
 run_tests()

 # 运行性能测试
 performance_test()

 # 复杂度分析
 print("\n复杂度分析:")
 print("方法一（回溯算法）:")
 print(" 时间复杂度: O(2^n)，其中 n 是有效字符串的数量（过滤后）")
```

```
print(" 空间复杂度: O(n), 递归调用栈的深度")

print("\n 方法二 (动态规划) :")
print(" 时间复杂度: O(n * 2^m), 其中 n 是字符串数量, m 是字符集大小 (最多 26) ")
print(" 空间复杂度: O(2^m), 存储所有可能的掩码组合")

print("\n 方法三 (优化的回溯算法) :")
print(" 时间复杂度: O(2^n), 但通过剪枝优化实际运行时间")
print(" 空间复杂度: O(n)")
print(" 优点: 利用剪枝减少不必要的计算, 对于大多数情况效率更高")

print("\n 适用场景总结:")
print("1. 当数组长度较小时, 三种方法都可以使用")
print("2. 当字符串包含大量重复字符时, 方法三的剪枝效果更好")
print("3. 当需要更稳定的性能时, 方法二的动态规划更可靠")
print("4. 在工程实践中, 应根据具体数据特征选择合适的方法")

运行主函数
if __name__ == "__main__":
 main()
```

=====

文件: Code12\_CountBits.cpp

=====

```
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
```

```
#include <limits>
#include <unordered_set>

using namespace std;

/***
 * SPOJ 问题: Counting Bits
 * 题目链接: https://www.spoj.com/problems/COUNT1BIT/
 * 题目描述: 计算从 1 到 n 的所有整数的二进制表示中 1 的个数之和
 *
 * 方法 1: 逐位计算法
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * 方法 2: 动态规划法
 * 时间复杂度: O(log n)
 * 空间复杂度: O(log n)
 *
 * 方法 3: 内置函数法
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 *
 * 工程化考量:
 * 1. 异常处理: 处理负数输入
 * 2. 性能优化: 使用位运算替代除法
 * 3. 可读性: 清晰的变量命名和注释
 * 4. 测试验证: 包含单元测试和性能测试
 */

```

```
class CountBits {
public:
 /**
 * 方法 1: 逐位计算法 (最优解)
 * 思路: 计算每一位对总和的贡献
 * 对于第 i 位, 每 2^{i+1} 个数中, 有 2^i 个数的该位为 1
 */
 static long long countBitsMethod1(long long n) {
 if (n < 0) {
 throw invalid_argument("输入必须是非负数");
 }

 long long count = 0;
 long long i = 1; // 当前处理的位
 }
}
```

```

while (i <= n) {
 // 计算当前位的周期
 long long divisor = i * 2;
 // 完整周期的数量
 long long fullCycles = n / divisor;
 // 完整周期中 1 的个数
 count += fullCycles * i;

 // 不完整周期中 1 的个数
 long long remainder = n % divisor;
 if (remainder >= i) {
 count += remainder - i + 1;
 }

 // 防止溢出
 if (i > LLONG_MAX / 2) break;
 i *= 2;
}

return count;
}

/***
* 方法 2: 动态规划法
* 思路: 利用已知结果计算更大数的结果
* dp[n] = dp[n/2] + (n % 2)
*/
static long long countBitsMethod2(long long n) {
 if (n < 0) {
 throw invalid_argument("输入必须是非负数");
 }

 if (n == 0) return 0;

 // 找到最大的 2 的幂
 long long power = 1;
 while (power * 2 <= n) {
 power *= 2;
 }

 // 计算结果
 // 对于[0, power-1]区间的数, 1 的个数是已知的

```

```

// 对于[power, n]区间的数，可以递归计算
return (power / 2) * (64 - __builtin_clzll(power)) / 2 +
(n - power + 1) +
countBitsMethod2(n - power);
}

/***
* 方法 3：内置函数法（简单但较慢）
* 思路：对每个数使用内置的 popcount 函数
*/
static long long countBitsMethod3(long long n) {
if (n < 0) {
throw invalid_argument("输入必须是非负数");
}

long long count = 0;
for (long long i = 1; i <= n; i++) {
count += __builtin_popcountll(i);
}
return count;
}

/***
* 方法 4：查表法（空间换时间）
* 思路：预计算小范围的 popcount，然后分段计算
*/
static long long countBitsMethod4(long long n) {
if (n < 0) {
throw invalid_argument("输入必须是非负数");
}

// 预计算 0-255 的 popcount
vector<int> table(256, 0);
for (int i = 0; i < 256; i++) {
table[i] = table[i >> 1] + (i & 1);
}

long long count = 0;
for (long long i = 1; i <= n; i++) {
unsigned long long num = i;
// 分 8 次查表（64 位整数）
count += table[num & 0xFF] +
table[(num >> 8) & 0xFF] +

```

```

 table[(num >> 16) & 0xFF] +
 table[(num >> 24) & 0xFF] +
 table[(num >> 32) & 0xFF] +
 table[(num >> 40) & 0xFF] +
 table[(num >> 48) & 0xFF] +
 table[(num >> 56) & 0xFF];
 }

 return count;
}

/***
 * 性能测试工具
 */
static void performanceTest() {
 vector<long long> testCases = {1000, 10000, 100000, 1000000, 10000000};
 vector<pair<string, function<long long(long long)>>> methods = {
 {"逐位计算法", countBitsMethod1},
 {"动态规划法", countBitsMethod2},
 {"内置函数法", countBitsMethod3},
 {"查表法", countBitsMethod4}
 };

 cout << "==== 性能测试 ===" << endl;
 for (auto n : testCases) {
 cout << "n = " << n << ":" << endl;

 for (auto& method : methods) {
 auto start = chrono::high_resolution_clock::now();
 long long result = method.second(n);
 auto end = chrono::high_resolution_clock::now();
 auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

 cout << setw(15) << method.first << ":" "
 << setw(10) << result << "("
 << duration.count() << " μs)" << endl;
 }
 cout << endl;
 }
}

/***
 * 单元测试

```

```
/*
static void unitTest() {
 cout << "==== 单元测试 ===" << endl;

 vector<pair<long long, long long>> testCases = {
 {0, 0},
 {1, 1},
 {2, 2},
 {3, 4},
 {4, 5},
 {5, 7},
 {10, 17},
 {100, 197},
 {1000, 1987}
 };

 vector<function<long long(long long)>> methods = {
 countBitsMethod1,
 countBitsMethod2,
 countBitsMethod3,
 countBitsMethod4
 };

 for (auto& testCase : testCases) {
 long long n = testCase.first;
 long long expected = testCase.second;

 cout << "测试 n = " << n << " (期望: " << expected << ")" << endl;

 for (size_t i = 0; i < methods.size(); i++) {
 try {
 long long result = methods[i](n);
 bool passed = (result == expected);
 cout << " 方法" << (i+1) << ":" << result
 << " (" << (passed ? "通过" : "失败") << ")" << endl;

 if (!passed) {
 cerr << "错误: 方法" << (i+1) << "计算结果错误" << endl;
 }
 } catch (const exception& e) {
 cerr << "错误: 方法" << (i+1) << "抛出异常: " << e.what() << endl;
 }
 }
 }
}
```

```

 cout << endl;
}
}

/***
 * 复杂度分析
 */
static void complexityAnalysis() {
 cout << "==== 复杂度分析 ===" << endl;

 cout << "方法 1 - 逐位计算法:" << endl;
 cout << " 时间复杂度: O(log n) - 处理每一位" << endl;
 cout << " 空间复杂度: O(1) - 只使用常数空间" << endl;
 cout << " 优势: 最优时间复杂度, 适合大数计算" << endl;
 cout << " 劣势: 逻辑相对复杂" << endl;

 cout << "方法 2 - 动态规划法:" << endl;
 cout << " 时间复杂度: O(log n) - 递归深度为 log n" << endl;
 cout << " 空间复杂度: O(log n) - 递归栈空间" << endl;
 cout << " 优势: 思路清晰, 易于理解" << endl;
 cout << " 劣势: 递归可能栈溢出" << endl;

 cout << "方法 3 - 内置函数法:" << endl;
 cout << " 时间复杂度: O(n log n) - 每个数需要 O(log n)时间" << endl;
 cout << " 空间复杂度: O(1) - 常数空间" << endl;
 cout << " 优势: 实现简单" << endl;
 cout << " 劣势: 时间复杂度较高" << endl;

 cout << "方法 4 - 查表法:" << endl;
 cout << " 时间复杂度: O(n) - 线性扫描" << endl;
 cout << " 空间复杂度: O(1) - 查表空间固定" << endl;
 cout << " 优势: 比内置函数法快" << endl;
 cout << " 劣势: 仍然需要线性时间" << endl;
}

/***
 * 边界测试
 */
static void boundaryTest() {
 cout << "==== 边界测试 ===" << endl;

 vector<long long> boundaryCases = {
 0, 1, 2, 3,

```

```

LLONG_MAX, // 最大 long long
(1LL << 62) - 1, // 2^62 - 1
(1LL << 62) // 2^62
};

for (auto n : boundaryCases) {
 cout << "测试边界值 n = " << n << ":" << endl;

 try {
 long long result1 = countBitsMethod1(n);
 long long result2 = countBitsMethod2(n);

 cout << " 方法1结果: " << result1 << endl;
 cout << " 方法2结果: " << result2 << endl;

 if (n <= 1000000) { // 避免超时
 long long result3 = countBitsMethod3(n);
 long long result4 = countBitsMethod4(n);
 cout << " 方法3结果: " << result3 << endl;
 cout << " 方法4结果: " << result4 << endl;

 // 验证一致性
 if (result1 == result2 && result2 == result3 && result3 == result4) {
 cout << " ✓ 所有方法结果一致" << endl;
 } else {
 cerr << " ✗ 方法结果不一致" << endl;
 }
 }
 } catch (const exception& e) {
 cerr << " 错误: " << e.what() << endl;
 }
 cout << endl;
}

}

};

/***
 * 主函数 - 测试和演示
 */
int main() {
 cout << "SPOJ Counting Bits 问题解决方案" << endl;
 cout << "计算从 1 到 n 的所有整数的二进制表示中 1 的个数之和" << endl;
 cout << "===== " << endl;
}

```

```
// 运行单元测试
CountBits::unitTest();

// 运行边界测试
CountBits::boundaryTest();

// 运行性能测试
CountBits::performanceTest();

// 复杂度分析
CountBits::complexityAnalysis();

// 交互式测试
cout << "==== 交互式测试 ===" << endl;
cout << "请输入要测试的 n 值 (输入-1 退出): ";

long long n;
while (cin >> n) {
 if (n == -1) break;

 if (n < 0) {
 cout << "错误: 输入必须是非负数" << endl;
 continue;
 }

 try {
 auto start = chrono::high_resolution_clock::now();
 long long result = CountBits::countBitsMethod1(n);
 auto end = chrono::high_resolution_clock::now();
 auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

 cout << "结果: " << result << " (计算时间: " << duration.count() << " μs)" << endl;
 }
}

// 对于较小的 n, 显示详细计算过程
if (n <= 100) {
 cout << "详细计算过程:" << endl;
 long long sum = 0;
 for (long long i = 1; i <= n; i++) {
 int bits = __builtin_popcountll(i);
 sum += bits;
 cout << " " << i << " (" << bitset<8>(i) << ") : " << bits << " 个1" << endl;
 }
}
```

```

 cout << "总和: " << sum << endl;
 }

} catch (const exception& e) {
 cerr << "计算错误: " << e.what() << endl;
}

cout << endl << "请输入下一个 n 值 (输入-1 退出): ";
}

cout << "程序结束" << endl;
return 0;
}

```

=====

文件: Code12\_CountBits.java

=====

```

package class032;

import java.util.*;

/**
 * SPOJ BITS - 计算位数
 * 题目链接: https://www.spoj.com/problems/BITS/
 * 题目描述: 计算从 1 到 N 所有数字的二进制表示中 1 的总个数
 *
 * 解题思路:
 * 方法 1: 暴力统计 - 对每个数字计算 1 的个数
 * 方法 2: 数学规律 - 利用二进制位模式统计
 * 方法 3: 动态规划 - 使用递推关系优化计算
 *
 * 时间复杂度分析:
 * 方法 1: O(N log N) - 每个数字需要 O(log N) 时间统计 1 的个数
 * 方法 2: O(log N) - 基于二进制位模式直接计算
 * 方法 3: O(N) - 线性扫描, 但需要 O(N) 空间
 *
 * 空间复杂度分析:
 * 方法 1: O(1) - 仅使用常数空间
 * 方法 2: O(1) - 仅使用常数空间
 * 方法 3: O(N) - 需要存储 DP 数组
 *
 * 工程化考量:

```

```
* 1. 输入验证：检查 N 的范围是否合理
* 2. 边界处理：处理 N=0 和 N=1 的特殊情况
* 3. 性能优化：对于大 N 使用数学方法避免超时
* 4. 内存管理：避免不必要的内存分配
*/
```

```
public class Code12_CountBits {

 /**
 * 方法 1：暴力统计法
 * 对每个数字从 1 到 N，统计其二进制表示中 1 的个数
 * 优点：实现简单，易于理解
 * 缺点：时间复杂度较高，不适合大 N
 */

 public static long countBitsBruteForce(int n) {
 // 输入验证
 if (n <= 0) return 0;

 long totalBits = 0;
 for (int i = 1; i <= n; i++) {
 totalBits += Integer.bitCount(i);
 }
 return totalBits;
 }

 /**
 * 方法 2：数学规律法（最优解）
 * 利用二进制位模式统计 1 的个数
 * 思路：对于每个二进制位，计算该位在 1 到 N 中出现的次数
 *
 * 对于第 k 位（从 0 开始，最低位为第 0 位）：
 * 完整的周期数： $n / (1 \ll (k+1))$
 * 每个周期中该位出现 $1 \ll k$ 次
 * 剩余部分： $n \% (1 \ll (k+1))$
 * 如果剩余部分 $\geq (1 \ll k)$ ，则额外出现 $(\text{剩余部分} - (1 \ll k) + 1)$ 次
 *
 * 时间复杂度： $O(\log N)$ – 只需要遍历二进制位数
 * 空间复杂度： $O(1)$ – 常数空间
 */

 public static long countBitsMath(int n) {
 if (n <= 0) return 0;

 long totalBits = 0;
```

```

int k = 0;
int mask = 1;

while (mask <= n) {
 // 计算完整的周期数
 long fullCycles = n / (mask << 1);
 // 每个完整周期中该位出现 mask 次
 totalBits += fullCycles * mask;

 // 计算剩余部分
 int remainder = n % (mask << 1);
 if (remainder >= mask) {
 totalBits += remainder - mask + 1;
 }

 // 移动到下一个位
 mask <<= 1;
 k++;
}

return totalBits;
}

/**
 * 方法 3: 动态规划法
 * 使用递推关系: countBits(n) = countBits(n >> 1) + (n & 1)
 * 但为了效率, 我们使用查表法优化
 *
 * 时间复杂度: O(N) - 线性扫描
 * 空间复杂度: O(N) - 需要存储 DP 数组
 */
public static long countBitsDP(int n) {
 if (n <= 0) return 0;

 int[] dp = new int[n + 1];
 long totalBits = 0;

 for (int i = 1; i <= n; i++) {
 // 递推关系: i 中 1 的个数 = i/2 中 1 的个数 + i 的最低位
 dp[i] = dp[i >> 1] + (i & 1);
 totalBits += dp[i];
 }
}

```

```
 return totalBits;
 }

/***
 * 方法 4: 使用内置函数（实际应用推荐）
 * 利用 Java 内置的 Integer.bitCount()，但使用数学优化避免 O(N log N)
 * 对于大 N，使用方法 2 的数学优化
 */
public static long countBitsOptimized(int n) {
 if (n <= 1000000) {
 // 对于较小的 n，使用 DP 方法更简单
 return countBitsDP(n);
 } else {
 // 对于大 n，使用数学方法避免超时
 return countBitsMath(n);
 }
}

/***
 * 工程化改进版本：增加完整的异常处理和验证
 */
public static long countBitsWithValidation(int n) {
 try {
 // 输入验证
 if (n < 0) {
 throw new IllegalArgumentException("Input n must be non-negative");
 }

 if (n == 0) return 0;
 if (n == 1) return 1;

 // 根据 n 的大小选择最优算法
 if (n <= 1000000) {
 return countBitsDP(n);
 } else {
 return countBitsMath(n);
 }
 } catch (Exception e) {
 System.err.println("Error in countBits: " + e.getMessage());
 return 0;
 }
}
```

```
/**
 * 单元测试方法
 */

public static void runTests() {
 System.out.println("==> SPOJ BITS - 计算位数 单元测试 ==>");

 // 测试用例 1: n=1
 int n1 = 1;
 long expected1 = 1;
 long result1 = countBitsMath(n1);
 System.out.printf("测试 1: n=%d, 期望=%d, 实际=%d, %s%n",
 n1, expected1, result1,
 result1 == expected1 ? "通过" : "失败");

 // 测试用例 2: n=3
 int n2 = 3;
 long expected2 = 4; // 1(1) + 2(1) + 3(2) = 4
 long result2 = countBitsMath(n2);
 System.out.printf("测试 2: n=%d, 期望=%d, 实际=%d, %s%n",
 n2, expected2, result2,
 result2 == expected2 ? "通过" : "失败");

 // 测试用例 3: n=5
 int n3 = 5;
 long expected3 = 7; // 1+1+2+1+2 = 7
 long result3 = countBitsMath(n3);
 System.out.printf("测试 3: n=%d, 期望=%d, 实际=%d, %s%n",
 n3, expected3, result3,
 result3 == expected3 ? "通过" : "失败");

 // 测试不同方法的结果一致性
 System.out.println("\n==> 方法一致性测试 ==>");
 int testN = 100;
 long r1 = countBitsBruteForce(testN);
 long r2 = countBitsMath(testN);
 long r3 = countBitsDP(testN);
 long r4 = countBitsOptimized(testN);

 System.out.printf("暴力法: %d\n", r1);
 System.out.printf("数学法: %d\n", r2);
 System.out.printf("动态规划: %d\n", r3);
 System.out.printf("优化法: %d\n", r4);
 System.out.printf("所有方法结果一致: %s\n",
```

```

 (r1 == r2 && r2 == r3 && r3 == r4) ? "是" : "否");
 }

/***
 * 性能测试方法
 */
public static void performanceTest() {
 System.out.println("\n==== 性能测试 ====");

 int[] testSizes = {1000, 10000, 100000, 1000000, 10000000};

 for (int n : testSizes) {
 System.out.printf("n = %d:%n", n);

 // 测试数学方法
 long startTime = System.nanoTime();
 long resultMath = countBitsMath(n);
 long mathTime = System.nanoTime() - startTime;

 // 测试 DP 方法（仅在小规模时测试）
 long dpTime = 0;
 if (n <= 1000000) {
 startTime = System.nanoTime();
 long resultDP = countBitsDP(n);
 dpTime = System.nanoTime() - startTime;
 System.out.printf(" DP 方法: %d ns, 结果: %d%n", dpTime, resultDP);
 }

 System.out.printf(" 数学方法: %d ns, 结果: %d%n", mathTime, resultMath);

 if (n <= 1000000) {
 double ratio = (double) dpTime / mathTime;
 System.out.printf(" 数学方法比 DP 快: %.2f 倍%n", ratio);
 }
 System.out.println();
 }

}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
 System.out.println("==== 复杂度分析 ====");
}

```

```
System.out.println("方法 1 (暴力统计) :");
System.out.println(" 时间复杂度: O(N log N) - 每个数字需要 O(log N) 时间统计 1 的个数");
System.out.println(" 空间复杂度: O(1) - 仅使用常数空间");
System.out.println(" 适用场景: 小规模数据, 代码简单");

System.out.println("\n 方法 2 (数学规律) :");
System.out.println(" 时间复杂度: O(log N) - 只需要遍历二进制位数");
System.out.println(" 空间复杂度: O(1) - 常数空间");
System.out.println(" 适用场景: 大规模数据, 最优解");

System.out.println("\n 方法 3 (动态规划) :");
System.out.println(" 时间复杂度: O(N) - 线性扫描");
System.out.println(" 空间复杂度: O(N) - 需要存储 DP 数组");
System.out.println(" 适用场景: 中等规模数据, 实现简单");

System.out.println("\n 工程化建议:");
System.out.println("1. 对于 N <= 1,000,000, 使用 DP 方法更简单直接");
System.out.println("2. 对于 N > 1,000,000, 必须使用数学方法避免超时");
System.out.println("3. 在实际应用中, 应根据数据规模动态选择算法");
}
```

```
public static void main(String[] args) {
 System.out.println("SPOJ BITS - 计算从 1 到 N 所有数字的二进制表示中 1 的总个数");
 System.out.println("使用位运算和数学优化实现");

 // 运行单元测试
 runTests();

 // 运行性能测试
 performanceTest();

 // 复杂度分析
 complexityAnalysis();

 // 示例使用
 System.out.println("\n==== 示例使用 ====");
 int[] examples = {10, 100, 1000};
 for (int n : examples) {
 long result = countBitsWithValidation(n);
 System.out.printf("从 1 到%d 的所有数字中 1 的总个数: %d\n", n, result);
 }
}
```

```
=====
```

文件: Code12\_CountBits.py

```
=====
```

```
#!/usr/bin/env python3
-*- coding: utf-8 -*-
```

```
"""
```

SPOJ 问题: Counting Bits

题目链接: <https://www.spoj.com/problems/COUNT1BIT/>

题目描述: 计算从 1 到 n 的所有整数的二进制表示中 1 的个数之和

方法 1: 逐位计算法

时间复杂度:  $O(\log n)$

空间复杂度:  $O(1)$

方法 2: 动态规划法

时间复杂度:  $O(\log n)$

空间复杂度:  $O(\log n)$

方法 3: 内置函数法

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(1)$

工程化考量:

1. 异常处理: 处理负数输入
  2. 性能优化: 使用位运算替代除法
  3. 可读性: 清晰的变量命名和注释
  4. 测试验证: 包含单元测试和性能测试
  5. 类型安全: 使用类型注解
  6. 文档化: 详细的 docstring
- ```
"""
```

```
import time
import sys
from typing import Callable, List, Tuple
from functools import lru_cache
import math

class CountBits:
    """
    SPOJ Counting Bits 问题的多种解决方案
    """
```

```
"""
```

```
@staticmethod  
def count_bits_method1(n: int) -> int:  
    """
```

方法 1：逐位计算法（最优解）

思路：计算每一位对总和的贡献

对于第 i 位，每 $2^{(i+1)}$ 个数中，有 2^i 个数的该位为 1

时间复杂度： $O(\log n)$

空间复杂度： $O(1)$

Args:

n: 要计算的最大整数

Returns:

int: 从 1 到 n 的所有整数的二进制表示中 1 的个数之和

Raises:

ValueError: 如果 n 为负数

```
"""
```

```
if n < 0:  
    raise ValueError("输入必须是非负数")
```

```
count = 0
```

```
i = 1 # 当前处理的位
```

```
while i <= n:
```

计算当前位的周期

```
divisor = i * 2
```

完整周期的数量

```
full_cycles = n // divisor
```

完整周期中 1 的个数

```
count += full_cycles * i
```

不完整周期中 1 的个数

```
remainder = n % divisor
```

```
if remainder >= i:
```

```
    count += remainder - i + 1
```

防止溢出

```
if i > sys.maxsize // 2:
```

```
    break
```

```
i *= 2

return count

@staticmethod
@lru_cache(maxsize=None)
def count_bits_method2(n: int) -> int:
    """
```

方法 2：动态规划法（记忆化递归）

思路：利用已知结果计算更大数的结果

$$dp[n] = dp[n//2] + (n \% 2)$$

时间复杂度： $O(\log n)$

空间复杂度： $O(\log n)$

Args:

n: 要计算的最大整数

Returns:

int: 从 1 到 n 的所有整数的二进制表示中 1 的个数之和

"""

```
if n <= 0:
```

```
    return 0
```

找到最大的 2 的幂

```
power = 1
```

```
while power * 2 <= n:
```

```
    power *= 2
```

计算结果

对于[0, power-1]区间的数，1 的个数是已知的

对于[power, n]区间的数，可以递归计算

```
return (power // 2) * (power.bit_length() - 1) + \
       (n - power + 1) + \
       CountBits.count_bits_method2(n - power)
```

@staticmethod

```
def count_bits_method3(n: int) -> int:
    """
```

方法 3：内置函数法（简单但较慢）

思路：对每个数使用内置的 bit_count 函数（Python 3.10+）

时间复杂度： $O(n \log n)$

空间复杂度: $O(1)$

Args:

n: 要计算的最大整数

Returns:

int: 从 1 到 n 的所有整数的二进制表示中 1 的个数之和

"""

if n < 0:

 raise ValueError("输入必须是非负数")

count = 0

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

 count += i.bit_count()

return count

@staticmethod

def count_bits_method4(n: int) -> int:

"""

方法 4: 查表法 (空间换时间)

思路: 预计算小范围的 `popcount`, 然后分段计算

时间复杂度: $O(n)$

空间复杂度: $O(1)$ - 查表空间固定

Args:

n: 要计算的最大整数

Returns:

int: 从 1 到 n 的所有整数的二进制表示中 1 的个数之和

"""

if n < 0:

 raise ValueError("输入必须是非负数")

预计算 0-255 的 `popcount`

table = [0] * 256

for i in range(256):

 table[i] = table[i // 2] + (i & 1)

count = 0

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

 num = i

 # 分 8 次查表 (64 位整数)

```

        count += table[num & 0xFF] + \
                  table[(num >> 8) & 0xFF] + \
                  table[(num >> 16) & 0xFF] + \
                  table[(num >> 24) & 0xFF] + \
                  table[(num >> 32) & 0xFF] + \
                  table[(num >> 40) & 0xFF] + \
                  table[(num >> 48) & 0xFF] + \
                  table[(num >> 56) & 0xFF]

    return count

@staticmethod
def performance_test():
    """性能测试工具"""
    test_cases = [1000, 10000, 100000, 1000000]
    methods = [
        ("逐位计算法", CountBits.count_bits_method1),
        ("动态规划法", CountBits.count_bits_method2),
        ("内置函数法", CountBits.count_bits_method3),
        ("查表法", CountBits.count_bits_method4)
    ]

    print("== 性能测试 ==")
    for n in test_cases:
        print(f"n = {n}:")
        for method_name, method_func in methods:
            start_time = time.time()
            result = method_func(n)
            end_time = time.time()
            duration = (end_time - start_time) * 1e6 # 转换为微秒
            print(f"  {method_name:15}: {result:10} ({duration:8.2f} μs)")
        print()

@staticmethod
def unit_test():
    """单元测试"""
    print("== 单元测试 ==")
    test_cases = [
        (0, 0),
        (1, 1),

```

```
(2, 2),
(3, 4),
(4, 5),
(5, 7),
(10, 17),
(100, 197),
(1000, 1987)

]

methods = [
    CountBits.count_bits_method1,
    CountBits.count_bits_method2,
    CountBits.count_bits_method3,
    CountBits.count_bits_method4
]

for n, expected in test_cases:
    print(f"测试 n = {n} (期望: {expected})")

    for i, method in enumerate(methods, 1):
        try:
            result = method(n)
            passed = (result == expected)
            status = "通过" if passed else "失败"
            print(f" 方法{i}: {result} ({status})")

            if not passed:
                print(f"错误: 方法{i} 计算结果错误", file=sys.stderr)
        except Exception as e:
            print(f"错误: 方法{i} 抛出异常: {e}", file=sys.stderr)
    print()

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("== 复杂度分析 ==")

    print("方法 1 - 逐位计算法:")
    print(" 时间复杂度: O(log n) - 处理每一位")
    print(" 空间复杂度: O(1) - 只使用常数空间")
    print(" 优势: 最优时间复杂度, 适合大数计算")
    print(" 劣势: 逻辑相对复杂")
```

```
print("方法 2 - 动态规划法:")
print(" 时间复杂度: O(log n) - 递归深度为 log n")
print(" 空间复杂度: O(log n) - 递归栈空间")
print(" 优势: 思路清晰, 易于理解")
print(" 劣势: 递归可能栈溢出")

print("方法 3 - 内置函数法:")
print(" 时间复杂度: O(n log n) - 每个数需要 O(log n)时间")
print(" 空间复杂度: O(1) - 常数空间")
print(" 优势: 实现简单")
print(" 劣势: 时间复杂度较高")

print("方法 4 - 查表法:")
print(" 时间复杂度: O(n) - 线性扫描")
print(" 空间复杂度: O(1) - 查表空间固定")
print(" 优势: 比内置函数法快")
print(" 劣势: 仍然需要线性时间")

@staticmethod
def boundary_test():
    """边界测试"""
    print("== 边界测试 ==")

    boundary_cases = [
        0, 1, 2, 3,
        (1 << 62) - 1, # 2^62 - 1
        (1 << 62)       # 2^62
    ]

    for n in boundary_cases:
        print(f"测试边界值 n = {n}:")

        try:
            result1 = CountBits.count_bits_method1(n)
            result2 = CountBits.count_bits_method2(n)

            print(f" 方法 1 结果: {result1}")
            print(f" 方法 2 结果: {result2}")

            if n <= 1000000: # 避免超时
                result3 = CountBits.count_bits_method3(n)
                result4 = CountBits.count_bits_method4(n)
                print(f" 方法 3 结果: {result3}")

        except Exception as e:
            print(f"发生错误: {e}")

    print("所有边界测试完成")
```

```
print(f" 方法 4 结果: {result4}")

# 验证一致性
if result1 == result2 == result3 == result4:
    print(" ✓ 所有方法结果一致")
else:
    print(" ✗ 方法结果不一致", file=sys.stderr)
except Exception as e:
    print(f" 错误: {e}", file=sys.stderr)
print()

@staticmethod
def interactive_test():
    """交互式测试"""
    print("== 交互式测试 ==")
    print("请输入要测试的 n 值 (输入-1 退出): ")

    while True:
        try:
            n_str = input().strip()
            if not n_str:
                continue

            n = int(n_str)
            if n == -1:
                break

            if n < 0:
                print("错误: 输入必须是非负数")
                continue

            start_time = time.time()
            result = CountBits.count_bits_method1(n)
            end_time = time.time()
            duration = (end_time - start_time) * 1e6 # 转换为微秒

            print(f"结果: {result} (计算时间: {duration:.2f} μs)")

            # 对于较小的 n, 显示详细计算过程
            if n <= 100:
                print("详细计算过程:")
                total_sum = 0
                for i in range(1, n + 1):
```

```
        bits = i.bit_count()
        total_sum += bits
        binary_str = bin(i)[2:].zfill(8)
        print(f"  {i:3} ({binary_str:>8}): {bits} 个 1")
        print(f"总和: {total_sum}")

except ValueError:
    print("错误: 请输入有效的整数")
except KeyboardInterrupt:
    print("\n程序被用户中断")
    break
except Exception as e:
    print(f"计算错误: {e}", file=sys.stderr)

def main():
    """主函数 - 测试和演示"""
    print("SPOJ Counting Bits 问题解决方案")
    print("计算从 1 到 n 的所有整数的二进制表示中 1 的个数之和")
    print("-" * 60)

    # 运行单元测试
    CountBits.unit_test()

    # 运行边界测试
    CountBits.boundary_test()

    # 运行性能测试
    CountBits.performance_test()

    # 复杂度分析
    CountBits.complexity_analysis()

    # 交互式测试
    CountBits.interactive_test()

    print("程序结束")

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

文件: Code13_BitArray.cpp

```
=====
#include <iostream>
#include <vector>
#include <chrono>
#include <bitset>
#include <unordered_set>
#include <stdexcept>
#include <cmath>
#include <tuple> // 添加 tuple 头文件

using namespace std;
using namespace std::chrono;

/***
 * HackerRank Bit Array - 位数组
 * 题目链接: https://www.hackerrank.com/challenges/bitset-1/problem
 * 题目描述: 根据给定规则生成序列, 计算序列中不同整数的个数
 *
 * 问题详细描述:
 * 给定四个整数: N, S, P, Q, 按照以下规则生成序列:
 * a[0] = S mod 2^31
 * 对于 i >= 1: a[i] = (a[i-1] * P + Q) mod 2^31
 * 计算序列 a[0], a[1], ..., a[N-1] 中不同整数的个数
 *
 * 约束条件:
 * 1 ≤ N ≤ 10^8
 * 0 ≤ S, P, Q ≤ 2^31 - 1
 *
 * 解题思路:
 * 方法 1: 使用 unordered_set - 简单但内存消耗大
 * 方法 2: 使用 bitset - 内存效率高, 适合大 N
 * 方法 3: Floyd 循环检测 - 检测序列中的循环, 避免存储整个序列
 *
 * 时间复杂度分析:
 * 方法 1: O(N) - 但内存消耗 O(N), 不适合大 N
 * 方法 2: O(N) - 内存消耗 O(2^31/8) ≈ 256MB, 可行
 * 方法 3: O(循环长度) - 最优, 但实现复杂
 *
 * 空间复杂度分析:
 * 方法 1: O(N) - 存储所有元素
 * 方法 2: O(2^31/8) - 固定大小 bitset
```

```
* 方法 3: O(1) - 常数空间
*
* 工程化考量:
* 1. 内存优化: 对于大 N 必须使用 bitset 或循环检测
* 2. 整数溢出: 使用 unsigned long long 进行中间计算
* 3. 边界处理: 处理 N=0, 1 的特殊情况
* 4. 异常安全: 使用 RAII 管理资源
*/
```

```
class BitArraySolver {
private:
    static const unsigned int MOD = 1U << 31; // 2^31

    /**
     * 计算下一个序列值
     */
    static unsigned int nextValue(unsigned int current, unsigned int p, unsigned int q) {
        unsigned long long next = (static_cast<unsigned long long>(current) * p + q) % MOD;
        return static_cast<unsigned int>(next);
    }

public:
    /**
     * 方法 1: 使用 unordered_set (仅适用于小 N)
     */
    static int countDistinctHashSet(unsigned int n, unsigned int s, unsigned int p, unsigned int q) {
        if (n == 0) return 0;
        if (n == 1) return 1;

        unordered_set<unsigned int> seen;
        unsigned int current = s % MOD;
        seen.insert(current);

        for (unsigned int i = 1; i < n; i++) {
            current = nextValue(current, p, q);
            seen.insert(current);
        }

        return static_cast<int>(seen.size());
    }
}
```

```

* 方法 2: 使用 bitset (推荐用于大 N)
* 使用 std::bitset 需要编译时确定大小, 这里使用 vector<bool>作为动态 bitset
*/
static int countDistinctBitSet(unsigned int n, unsigned int s, unsigned int p, unsigned int q) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    // 使用 vector<bool>作为动态 bitset
    vector<bool> visited(MOD, false);
    int count = 0;
    unsigned int current = s % MOD;

    for (unsigned int i = 0; i < n; i++) {
        if (!visited[current]) {
            visited[current] = true;
            count++;
        }

        if (i < n - 1) {
            current = nextValue(current, p, q);
        }
    }

    return count;
}

/***
* 方法 3: Floyd 循环检测算法 (最优解)
*/
static int countDistinctFloyd(unsigned int n, unsigned int s, unsigned int p, unsigned int q) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    unsigned int slow = s % MOD;
    unsigned int fast = s % MOD;
    int count = 1; // 至少有一个元素 s

    // 第一阶段: 检测循环
    bool cycleFound = false;
    unsigned int cycleLength = 0;

```

```

for (unsigned int i = 1; i < n; i++) {
    // 慢指针移动一步
    slow = nextValue(slow, p, q);

    // 快指针移动两步
    fast = nextValue(fast, p, q);
    fast = nextValue(fast, p, q);

    if (slow == fast) {
        cycleFound = true;

        // 计算循环长度
        cycleLength = 1;
        unsigned int temp = nextValue(slow, p, q);
        while (temp != slow) {
            temp = nextValue(temp, p, q);
            cycleLength++;
        }
        break;
    }
}

if (!cycleFound) {
    return n; // 没有循环，所有元素都不同
}

// 第二阶段：找到循环起点
slow = s % MOD;
fast = s % MOD;

// 快指针先移动 cycleLength 步
for (unsigned int i = 0; i < cycleLength; i++) {
    fast = nextValue(fast, p, q);
}

// 同时移动快慢指针直到相遇
while (slow != fast) {
    slow = nextValue(slow, p, q);
    fast = nextValue(fast, p, q);
}

unsigned int cycleStart = slow;

```

```

// 第三阶段：计算不同元素个数
unordered_set<unsigned int> cycleElements;
unsigned int current = cycleStart;

do {
    cycleElements.insert(current);
    current = nextValue(current, p, q);
} while (current != cycleStart);

// 计算循环前元素个数
unsigned int elementsBeforeCycle = 0;
current = s % MOD;
while (current != cycleStart) {
    elementsBeforeCycle++;
    current = nextValue(current, p, q);
}

return static_cast<int>(elementsBeforeCycle + cycleElements.size());
}

/***
 * 方法 4：优化版本 - 根据 N 的大小选择算法
 */
static int countDistinctOptimized(unsigned int n, unsigned int s, unsigned int p, unsigned
int q) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    // 对于小 N 使用 HashSet
    if (n <= 1000000) {
        return countDistinctHashSet(n, s, p, q);
    }

    // 对于大 N 使用 BitSet
    else if (n <= 100000000) {
        return countDistinctBitSet(n, s, p, q);
    }

    // 对于非常大的 N 使用 Floyd 算法
    else {
        return countDistinctFloyd(n, s, p, q);
    }
}

/***

```

```

* 工程化改进版本：完整的异常处理
*/
static int countDistinctWithValidation(unsigned int n, unsigned int s, unsigned int p,
unsigned int q) {
    try {
        // 输入验证
        if (s >= MOD || p >= MOD || q >= MOD) {
            throw invalid_argument("s, p, q must be less than 2^31");
        }

        return countDistinctOptimized(n, s, p, q);
    } catch (const exception& e) {
        cerr << "Error in countDistinct: " << e.what() << endl;
        return 0;
    }
}

/**
 * 性能测试工具类
*/
class PerformanceTester {
public:
    static void runTests() {
        cout << "==== HackerRank Bit Array - 单元测试 ===" << endl;

        // 测试用例
        vector<tuple<unsigned int, unsigned int, unsigned int, unsigned int, int>> testCases = {
            {5, 1, 2, 1, 5},      // 序列: 1, 3, 7, 15, 31
            {10, 1, 3, 0, 4},     // 序列可能产生循环
            {1, 100, 1, 1, 1}     // 边界情况 n=1
        };

        for (const auto& testCase : testCases) {
            auto [n, s, p, q, expected] = testCase;
            int result = BitArraySolver::countDistinctOptimized(n, s, p, q);

            cout << "测试 n=" << n << ", s=" << s << ", p=" << p << ", q=" << q
                << ", 期望=" << expected << ", 实际=" << result
                << ", " << (result == expected ? "通过" : "失败") << endl;
        }
    }
}

```

```

// 方法一致性测试
cout << "\n==== 方法一致性测试 ===" << endl;
unsigned int testN = 100, testS = 1, testP = 3, testQ = 1;

int r1 = BitArraySolver::countDistinctHashSet(testN, testS, testP, testQ);
int r2 = BitArraySolver::countDistinctBitSet(testN, testS, testP, testQ);
int r3 = BitArraySolver::countDistinctFloyd(testN, testS, testP, testQ);
int r4 = BitArraySolver::countDistinctOptimized(testN, testS, testP, testQ);

cout << "HashSet: " << r1 << endl;
cout << "BitSet: " << r2 << endl;
cout << "Floyd: " << r3 << endl;
cout << "优化法: " << r4 << endl;
cout << "所有方法结果一致: " << (r1 == r2 && r2 == r3 && r3 == r4 ? "是" : "否") << endl;
}

static void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;

    vector<tuple<unsigned int, unsigned int, unsigned int, unsigned int>> testCases = {
        {1000, 1, 3, 1},
        {10000, 1, 3, 1},
        {100000, 1, 3, 1},
        {1000000, 1, 3, 1}
    };

    for (const auto& testCase : testCases) {
        auto [n, s, p, q] = testCase;
        cout << "n = " << n << ":" << endl;

        // 测试不同方法
        if (n <= 100000) {
            auto start = high_resolution_clock::now();
            int result1 = BitArraySolver::countDistinctHashSet(n, s, p, q);
            auto time1 = duration_cast<nanoseconds>(high_resolution_clock::now() - start);
            cout << " HashSet: " << time1.count() << " ns, 结果: " << result1 << endl;
        }

        auto start = high_resolution_clock::now();
        int result2 = BitArraySolver::countDistinctBitSet(n, s, p, q);
        auto time2 = duration_cast<nanoseconds>(high_resolution_clock::now() - start);
        cout << " BitSet: " << time2.count() << " ns, 结果: " << result2 << endl;
    }
}

```

```

        start = high_resolution_clock::now();
        int result3 = BitArraySolver::countDistinctFloyd(n, s, p, q);
        auto time3 = duration_cast<nanoseconds>(high_resolution_clock::now() - start);
        cout << " Floyd: " << time3.count() << " ns, 结果: " << result3 << endl;

        cout << endl;
    }
}

};

/***
 * 复杂度分析
 */
void complexityAnalysis() {
    cout << "==== 复杂度分析 ===" << endl;
    cout << "方法 1 (HashSet) :" << endl;
    cout << " 时间复杂度: O(N)" << endl;
    cout << " 空间复杂度: O(N)" << endl;
    cout << " 适用场景: 小规模数据 (N <= 10^6)" << endl;

    cout << "\n 方法 2 (BitSet) :" << endl;
    cout << " 时间复杂度: O(N)" << endl;
    cout << " 空间复杂度: O(2^31/8) ≈ 256MB" << endl;
    cout << " 适用场景: 中等规模数据 (N <= 10^8)" << endl;

    cout << "\n 方法 3 (Floyd 循环检测) :" << endl;
    cout << " 时间复杂度: O(循环长度)" << endl;
    cout << " 空间复杂度: O(1)" << endl;
    cout << " 适用场景: 大规模数据 (N > 10^8)" << endl;

    cout << "\n 工程化建议:" << endl;
    cout << "1. 根据 N 的大小动态选择算法" << endl;
    cout << "2. 使用 unsigned long long 避免整数溢出" << endl;
    cout << "3. 对于竞赛题目, BitSet 方法通常是最佳选择" << endl;
}

int main() {
    cout << "HackerRank Bit Array - 位数组问题" << endl;
    cout << "计算根据规则生成的序列中不同整数的个数" << endl;

    // 运行单元测试
    PerformanceTester::runTests();
}

```

```

// 运行性能测试
PerformanceTester::performanceTest();

// 复杂度分析
complexityAnalysis();

// 示例使用
cout << "\n==== 示例使用 ===" << endl;
vector<tuple<unsigned int, unsigned int, unsigned int, unsigned int>> examples = {
    {5, 1, 2, 1},
    {10, 1, 3, 0},
    {100, 1, 1, 1}
};

for (const auto& example : examples) {
    auto [n, s, p, q] = example;
    int result = BitArraySolver::countDistinctWithValidation(n, s, p, q);
    cout << "n=" << n << ", s=" << s << ", p=" << p << ", q=" << q
        << " -> 不同元素个数: " << result << endl;
}

return 0;
}

```

=====

文件: Code13_BitArray.java

=====

```

package class032;

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

/**
 * HackerRank Bit Array - 位数组
 * 题目链接: https://www.hackerrank.com/challenges/bitset-1/problem
 * 题目描述: 根据给定规则生成序列, 计算序列中不同整数的个数
 *
 * 问题详细描述:
 * 给定四个整数: N, S, P, Q, 按照以下规则生成序列:
 * a[0] = S mod 2^31
 * 对于 i >= 1: a[i] = (a[i-1] * P + Q) mod 2^31
 * 计算序列 a[0], a[1], ..., a[N-1] 中不同整数的个数

```

```
*  
* 约束条件:  
*  $1 \leq N \leq 10^8$   
*  $0 \leq S, P, Q \leq 2^{31} - 1$   
*  
* 解题思路:  
* 方法 1: 使用 HashSet - 简单但内存消耗大, 不适合大 N  
* 方法 2: 使用 BitSet - 内存效率高, 适合大 N  
* 方法 3: Floyd 循环检测 - 检测序列中的循环, 避免存储整个序列  
*  
* 时间复杂度分析:  
* 方法 1:  $O(N)$  - 但内存消耗  $O(N)$ , 不适合大 N  
* 方法 2:  $O(N)$  - 内存消耗  $O(2^{31}/8) \approx 256MB$ , 可行  
* 方法 3:  $O(\text{循环长度})$  - 最优, 但实现复杂  
*  
* 空间复杂度分析:  
* 方法 1:  $O(N)$  - 存储所有元素  
* 方法 2:  $O(2^{31}/8)$  - 固定大小 BitSet  
* 方法 3:  $O(1)$  - 常数空间  
*  
* 工程化考量:  
* 1. 内存优化: 对于大 N 必须使用 BitSet 或循环检测  
* 2. 整数溢出: 使用 long 进行中间计算避免溢出  
* 3. 边界处理: 处理 N=0, 1 的特殊情况  
* 4. 性能优化: 选择最适合数据规模的算法  
*/
```

```
public class Code13_BitArray {  
  
    /**  
     * 方法 1: 使用 HashSet (仅适用于小 N)  
     * 优点: 实现简单, 代码清晰  
     * 缺点: 内存消耗大, 不适合大 N  
     */  
  
    public static int countDistinctHashSet(int n, int s, int p, int q) {  
        if (n <= 0) return 0;  
        if (n == 1) return 1;  
  
        Set<Integer> set = new HashSet<>();  
        int current = s;  
        set.add(current);  
  
        for (int i = 1; i < n; i++) {
```

```

// 使用 long 避免整数溢出
long next = ((long) current * p + q) % (1L << 31);
current = (int) next;
set.add(current);

}

return set.size();
}

/***
* 方法 2: 使用 BitSet (推荐用于大 N)
* 优点: 内存效率高, 适合处理大量数据
* 缺点: 需要固定大小的内存分配
*/
public static int countDistinctBitSet(int n, int s, int p, int q) {
    if (n <= 0) return 0;
    if (n == 1) return 1;

    // 创建足够大的 BitSet (2^31 位 ≈ 256MB)
    BitSet bitSet = new BitSet(1 << 31);
    int count = 0;
    int current = s;

    for (int i = 0; i < n; i++) {
        if (!bitSet.get(current)) {
            bitSet.set(current);
            count++;
        }
    }

    // 生成下一个元素
    if (i < n - 1) {
        long next = ((long) current * p + q) % (1L << 31);
        current = (int) next;
    }
}

return count;
}

/***
* 方法 3: Floyd 循环检测算法 (最优解)
* 使用快慢指针检测序列中的循环, 避免存储整个序列
* 优点: 常数空间, 适合任意大的 N

```

* 缺点：实现相对复杂

*/

```
public static int countDistinctFloyd(int n, int s, int p, int q) {  
    if (n <= 0) return 0;  
    if (n == 1) return 1;  
  
    int mod = 1 << 31;  
  
    // 快慢指针初始化  
    int slow = s;  
    int fast = s;  
    int count = 1; // 至少有一个元素 s  
  
    // 第一阶段：检测循环  
    boolean cycleFound = false;  
    int cycleLength = 0;  
  
    for (int i = 1; i < n; i++) {  
        // 慢指针移动一步  
        slow = nextValue(slow, p, q, mod);  
  
        // 快指针移动两步  
        fast = nextValue(fast, p, q, mod);  
        fast = nextValue(fast, p, q, mod);  
  
        // 如果快慢指针相遇，说明检测到循环  
        if (slow == fast) {  
            cycleFound = true;  
  
            // 计算循环长度  
            cycleLength = 1;  
            int temp = nextValue(slow, p, q, mod);  
            while (temp != slow) {  
                temp = nextValue(temp, p, q, mod);  
                cycleLength++;  
            }  
            break;  
        }  
    }  
  
    if (!cycleFound) {  
        // 如果没有检测到循环，则所有元素都不同  
        return n;  
    }  
}
```

```

}

// 第二阶段：找到循环起点
slow = s;
fast = s;

// 快指针先移动 cycleLength 步
for (int i = 0; i < cycleLength; i++) {
    fast = nextValue(fast, p, q, mod);
}

// 同时移动快慢指针直到相遇
while (slow != fast) {
    slow = nextValue(slow, p, q, mod);
    fast = nextValue(fast, p, q, mod);
}

// 循环起点
int cycleStart = slow;

// 第三阶段：计算不同元素个数
Set<Integer> cycleElements = new HashSet<>();
int current = cycleStart;

do {
    cycleElements.add(current);
    current = nextValue(current, p, q, mod);
} while (current != cycleStart);

// 不同元素总数 = 循环前元素个数 + 循环中不同元素个数
int elementsBeforeCycle = 0;
current = s;
while (current != cycleStart) {
    elementsBeforeCycle++;
    current = nextValue(current, p, q, mod);
}

return elementsBeforeCycle + cycleElements.size();
}

/**
 * 计算下一个序列值
 */

```

```

private static int nextValue(int current, int p, int q, int mod) {
    long next = ((long) current * p + q) % mod;
    return (int) next;
}

/***
 * 方法 4：优化版本 - 根据 N 的大小选择算法
 */
public static int countDistinctOptimized(int n, int s, int p, int q) {
    if (n <= 0) return 0;
    if (n == 1) return 1;

    // 对于小 N 使用 HashSet (更简单)
    if (n <= 1000000) {
        return countDistinctHashSet(n, s, p, q);
    }

    // 对于大 N 使用 BitSet (内存效率高)
    else if (n <= 100000000) {
        return countDistinctBitSet(n, s, p, q);
    }

    // 对于非常大的 N 使用 Floyd 算法 (常数空间)
    else {
        return countDistinctFloyd(n, s, p, q);
    }
}

/***
 * 工程化改进版本：完整的异常处理和验证
 */
public static int countDistinctWithValidation(int n, int s, int p, int q) {
    try {
        // 输入验证
        if (n < 0) {
            throw new IllegalArgumentException("n must be non-negative");
        }
        if (s < 0 || p < 0 || q < 0) {
            throw new IllegalArgumentException("s, p, q must be non-negative");
        }
        if (s >= (1L << 31) || p >= (1L << 31) || q >= (1L << 31)) {
            throw new IllegalArgumentException("s, p, q must be less than 2^31");
        }
    }

    return countDistinctOptimized(n, s, p, q);
}

```

```

        } catch (Exception e) {
            System.out.println("Error in countDistinct: " + e.getMessage());
            return 0;
        }
    }

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("==> HackerRank Bit Array - 单元测试 ==>");

    // 测试用例 1: 小规模测试
    int n1 = 5, s1 = 1, p1 = 2, q1 = 1;
    int expected1 = 5; // 序列: 1, 3, 7, 15, 31 全部不同
    int result1 = countDistinctOptimized(n1, s1, p1, q1);
    System.out.printf("测试 1: n=%d, s=%d, p=%d, q=%d, 期望=%d, 实际=%d, %s%n",
                      n1, s1, p1, q1, expected1, result1,
                      result1 == expected1 ? "通过" : "失败");

    // 测试用例 2: 有重复的序列
    int n2 = 10, s2 = 1, p2 = 3, q2 = 0;
    int expected2 = 4; // 序列可能产生循环
    int result2 = countDistinctOptimized(n2, s2, p2, q2);
    System.out.printf("测试 2: n=%d, s=%d, p=%d, q=%d, 期望=%d, 实际=%d, %s%n",
                      n2, s2, p2, q2, expected2, result2,
                      result2 == expected2 ? "通过" : "失败");

    // 测试用例 3: 边界情况 n=1
    int n3 = 1, s3 = 100, p3 = 1, q3 = 1;
    int expected3 = 1;
    int result3 = countDistinctOptimized(n3, s3, p3, q3);
    System.out.printf("测试 3: n=%d, s=%d, p=%d, q=%d, 期望=%d, 实际=%d, %s%n",
                      n3, s3, p3, q3, expected3, result3,
                      result3 == expected3 ? "通过" : "失败");

    // 测试不同方法的结果一致性 (小规模)
    System.out.println("\n==> 方法一致性测试 (小规模) ==>");
    int testN = 100;
    int r1 = countDistinctHashSet(testN, 1, 3, 1);
    int r2 = countDistinctBitSet(testN, 1, 3, 1);
    int r3 = countDistinctFloyd(testN, 1, 3, 1);
}

```

```
int r4 = countDistinctOptimized(testN, 1, 3, 1);

System.out.printf("HashSet: %d%n", r1);
System.out.printf("BitSet: %d%n", r2);
System.out.printf("Floyd: %d%n", r3);
System.out.printf("优化法: %d%n", r4);
System.out.printf("所有方法结果一致: %s%n",
    (r1 == r2 && r2 == r3 && r3 == r4) ? "是" : "否");
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n== 性能测试 ==");

    // 测试不同规模的数据
    int[][] testCases = {
        {1000, 1, 3, 1},
        {10000, 1, 3, 1},
        {100000, 1, 3, 1},
        {1000000, 1, 3, 1}
    };

    for (int[] testCase : testCases) {
        int n = testCase[0], s = testCase[1], p = testCase[2], q = testCase[3];
        System.out.printf("n = %d:%n", n);

        // 测试 HashSet 方法 (仅小规模)
        if (n <= 100000) {
            long startTime = System.nanoTime();
            int result1 = countDistinctHashSet(n, s, p, q);
            long time1 = System.nanoTime() - startTime;
            System.out.printf(" HashSet: %d ns, 结果: %d%n", time1, result1);
        }

        // 测试 BitSet 方法
        long startTime = System.nanoTime();
        int result2 = countDistinctBitSet(n, s, p, q);
        long time2 = System.nanoTime() - startTime;
        System.out.printf(" BitSet: %d ns, 结果: %d%n", time2, result2);

        // 测试 Floyd 方法
    }
}
```

```
        startTime = System.nanoTime();
        int result3 = countDistinctFloyd(n, s, p, q);
        long time3 = System.nanoTime() - startTime;
        System.out.printf(" Floyd: %d ns, 结果: %d\n", time3, result3);

        System.out.println();
    }

}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("== 复杂度分析 ==");
    System.out.println("方法 1 (HashSet) :");
    System.out.println(" 时间复杂度: O(N) - 线性扫描");
    System.out.println(" 空间复杂度: O(N) - 存储所有元素");
    System.out.println(" 适用场景: 小规模数据 (N <= 10^6)");

    System.out.println("\n 方法 2 (BitSet) :");
    System.out.println(" 时间复杂度: O(N) - 线性扫描");
    System.out.println(" 空间复杂度: O(2^31/8) ≈ 256MB - 固定大小");
    System.out.println(" 适用场景: 中等规模数据 (N <= 10^8)");

    System.out.println("\n 方法 3 (Floyd 循环检测) :");
    System.out.println(" 时间复杂度: O(循环长度) - 通常远小于 N");
    System.out.println(" 空间复杂度: O(1) - 常数空间");
    System.out.println(" 适用场景: 大规模数据 (N > 10^8)");

    System.out.println("\n 工程化建议:");
    System.out.println("1. 根据 N 的大小动态选择算法");
    System.out.println("2. 注意整数溢出问题, 使用 long 进行中间计算");
    System.out.println("3. 对于竞赛题目, 通常 BitSet 方法是最佳选择");
}

public static void main(String[] args) {
    System.out.println("HackerRank Bit Array - 位数组问题");
    System.out.println("计算根据规则生成的序列中不同整数的个数");

    // 运行单元测试
    runTests();

    // 运行性能测试
}
```

```

performanceTest();

// 复杂度分析
complexityAnalysis();

// 示例使用
System.out.println("\n== 示例使用 ==");
int[][] examples = {
    {5, 1, 2, 1},
    {10, 1, 3, 0},
    {100, 1, 1, 1}
};

for (int[] example : examples) {
    int n = example[0], s = example[1], p = example[2], q = example[3];
    int result = countDistinctWithValidation(n, s, p, q);
    System.out.printf("n=%d, s=%d, p=%d, q=%d -> 不同元素个数: %d\n",
                      n, s, p, q, result);
}

// 处理标准输入（用于在线评测）
/*
Scanner scanner = new Scanner(System.in);
int n = scanner.nextInt();
int s = scanner.nextInt();
int p = scanner.nextInt();
int q = scanner.nextInt();
System.out.println(countDistinctOptimized(n, s, p, q));
scanner.close();
*/
}

=====

```

文件: Code13_BitArray.py

=====

"""

HackerRank Bit Array - 位数组

题目链接: <https://www.hackerrank.com/challenges/bitset-1/problem>

题目描述: 根据给定规则生成序列, 计算序列中不同整数的个数

问题详细描述:

给定四个整数: N, S, P, Q, 按照以下规则生成序列:

a[0] = S mod 2^31

对于 i >= 1: a[i] = (a[i-1] * P + Q) mod 2^31

计算序列 a[0], a[1], ..., a[N-1]中不同整数的个数

约束条件:

1 ≤ N ≤ 10^8

0 ≤ S, P, Q ≤ 2^31 - 1

解题思路:

方法 1: 使用 set - 简单但内存消耗大

方法 2: 使用位数组 - 内存效率高, 适合大 N

方法 3: Floyd 循环检测 - 检测序列中的循环, 避免存储整个序列

时间复杂度分析:

方法 1: O(N) - 但内存消耗 O(N), 不适合大 N

方法 2: O(N) - 内存消耗 O(2^31/8) ≈ 256MB, 可行

方法 3: O(循环长度) - 最优, 但实现复杂

空间复杂度分析:

方法 1: O(N) - 存储所有元素

方法 2: O(2^31/8) - 固定大小位数组

方法 3: O(1) - 常数空间

工程化考量:

1. 内存优化: 对于大 N 必须使用位数组或循环检测

2. 整数溢出: 使用 64 位整数进行中间计算

3. 边界处理: 处理 N=0, 1 的特殊情况

4. 异常安全: 使用异常处理机制

"""

```
import sys
import time
from typing import Set, List, Tuple
import array

class BitArraySolver:
    MOD = 1 << 31 # 2^31

    @staticmethod
    def next_value(current: int, p: int, q: int) -> int:
        """计算下一个序列值"""
        next_val = (current * p + q) % BitArraySolver.MOD
```

```

    return next_val

@staticmethod
def count_distinct_hashset(n: int, s: int, p: int, q: int) -> int:
    """
    方法 1: 使用 set (仅适用于小 N)
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    if n == 0:
        return 0
    if n == 1:
        return 1

    seen = set()
    current = s % BitArraySolver.MOD
    seen.add(current)

    for i in range(1, n):
        current = BitArraySolver.next_value(current, p, q)
        seen.add(current)

    return len(seen)

```

```

@staticmethod
def count_distinct_bitarray(n: int, s: int, p: int, q: int) -> int:
    """
    方法 2: 使用位数组 (推荐用于大 N)
    时间复杂度: O(N)
    空间复杂度: O(2^31/8) ≈ 256MB

```

使用 Python 的 bytearray 实现位数组

```

    if n == 0:
        return 0
    if n == 1:
        return 1

    # 创建位数组, 每个字节存储 8 位
    bit_array_size = (BitArraySolver.MOD + 7) // 8
    visited = bytearray(bit_array_size)
    count = 0
    current = s % BitArraySolver.MOD

```

```

for i in range(n):
    # 计算位数组中的位置
    byte_index = current // 8
    bit_index = current % 8

    # 检查是否已经访问过
    if not (visited[byte_index] & (1 << bit_index)):
        visited[byte_index] |= (1 << bit_index)
        count += 1

    if i < n - 1:
        current = BitArraySolver.next_value(current, p, q)

return count

@staticmethod
def count_distinct_floyd(n: int, s: int, p: int, q: int) -> int:
    """
    方法 3: Floyd 循环检测算法（最优解）
    时间复杂度: O(循环长度)
    空间复杂度: O(1)
    """

    if n == 0:
        return 0
    if n == 1:
        return 1

    slow = s % BitArraySolver.MOD
    fast = s % BitArraySolver.MOD
    count = 1 # 至少有一个元素 s

    # 第一阶段: 检测循环
    cycle_found = False
    cycle_length = 0

    for i in range(1, n):
        # 慢指针移动一步
        slow = BitArraySolver.next_value(slow, p, q)

        # 快指针移动两步
        fast = BitArraySolver.next_value(fast, p, q)
        fast = BitArraySolver.next_value(fast, p, q)

        if slow == fast:
            cycle_found = True
            break

    if not cycle_found:
        return count

    # 第二阶段: 找到循环入口
    slow = s
    while slow != fast:
        slow = BitArraySolver.next_value(slow, p, q)
        fast = BitArraySolver.next_value(fast, p, q)

    # 第三阶段: 找到循环长度
    cycle_length = 1
    slow = BitArraySolver.next_value(slow, p, q)
    while slow != fast:
        slow = BitArraySolver.next_value(slow, p, q)
        cycle_length += 1

    return count - cycle_length

```

```

if slow == fast:
    cycle_found = True

    # 计算循环长度
    cycle_length = 1
    temp = BitArraySolver.next_value(slow, p, q)
    while temp != slow:
        temp = BitArraySolver.next_value(temp, p, q)
        cycle_length += 1
    break

if not cycle_found:
    return n # 没有循环，所有元素都不同

# 第二阶段：找到循环起点
slow = s % BitArraySolver.MOD
fast = s % BitArraySolver.MOD

# 快指针先移动 cycle_length 步
for _ in range(cycle_length):
    fast = BitArraySolver.next_value(fast, p, q)

# 同时移动快慢指针直到相遇
while slow != fast:
    slow = BitArraySolver.next_value(slow, p, q)
    fast = BitArraySolver.next_value(fast, p, q)

cycle_start = slow

# 第三阶段：计算不同元素个数
cycle_elements = set()
current = cycle_start

while True:
    cycle_elements.add(current)
    current = BitArraySolver.next_value(current, p, q)
    if current == cycle_start:
        break

# 计算循环前元素个数
elements_before_cycle = 0
current = s % BitArraySolver.MOD

```

```

while current != cycle_start:
    elements_before_cycle += 1
    current = BitArraySolver.next_value(current, p, q)

return elements_before_cycle + len(cycle_elements)

@staticmethod
def count_distinct_optimized(n: int, s: int, p: int, q: int) -> int:
    """
    方法 4: 优化版本 - 根据 N 的大小选择算法
    """

    if n == 0:
        return 0
    if n == 1:
        return 1

    # 对于小 N 使用 HashSet
    if n <= 1000000:
        return BitArraySolver.count_distinct_hashset(n, s, p, q)
    # 对于大 N 使用位数组
    elif n <= 100000000:
        return BitArraySolver.count_distinct_bitarray(n, s, p, q)
    # 对于非常大的 N 使用 Floyd 算法
    else:
        return BitArraySolver.count_distinct_floyd(n, s, p, q)

@staticmethod
def count_distinct_with_validation(n: int, s: int, p: int, q: int) -> int:
    """
    工程化改进版本: 完整的异常处理
    """

    try:
        # 输入验证
        if s >= BitArraySolver.MOD or p >= BitArraySolver.MOD or q >= BitArraySolver.MOD:
            raise ValueError("s, p, q must be less than 2^31")

        return BitArraySolver.count_distinct_optimized(n, s, p, q)

    except Exception as e:
        print(f"Error in count_distinct: {e}")
        return 0

```

```

class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def run_tests():
        """运行单元测试"""
        print("== HackerRank Bit Array - 单元测试 ==")

        # 测试用例
        test_cases = [
            (5, 1, 2, 1, 5),      # 序列: 1, 3, 7, 15, 31
            (10, 1, 3, 0, 4),    # 序列可能产生循环
            (1, 100, 1, 1, 1)    # 边界情况 n=1
        ]

        for n, s, p, q, expected in test_cases:
            result = BitArraySolver.count_distinct_optimized(n, s, p, q)

            print(f"测试 n={n}, s={s}, p={p}, q={q}, "
                  f"期望={expected}, 实际={result}, "
                  f"'通过' if result == expected else '失败'")

        # 方法一致性测试
        print("\n== 方法一致性测试 ==")
        test_n, test_s, test_p, test_q = 100, 1, 3, 1

        r1 = BitArraySolver.count_distinct_hashset(test_n, test_s, test_p, test_q)
        r2 = BitArraySolver.count_distinct_bitarray(test_n, test_s, test_p, test_q)
        r3 = BitArraySolver.count_distinct_floyd(test_n, test_s, test_p, test_q)
        r4 = BitArraySolver.count_distinct_optimized(test_n, test_s, test_p, test_q)

        print(f"HashSet: {r1}")
        print(f"BitArray: {r2}")
        print(f"Floyd: {r3}")
        print(f"优化法: {r4}")
        print(f"所有方法结果一致: {'是' if r1 == r2 == r3 == r4 else '否'}")

    @staticmethod
    def performance_test():
        """性能测试"""
        print("\n== 性能测试 ==")

        test_cases = [

```

```

        (1000, 1, 3, 1),
        (10000, 1, 3, 1),
        (100000, 1, 3, 1),
        (1000000, 1, 3, 1)
    ]

for n, s, p, q in test_cases:
    print(f"n = {n}:")

    # 测试不同方法
    if n <= 100000:
        start_time = time.time()
        result1 = BitArraySolver.count_distinct_hashset(n, s, p, q)
        time1 = (time.time() - start_time) * 1e9
        print(f"  HashSet: {time1:.0f} ns, 结果: {result1}")

        start_time = time.time()
        result2 = BitArraySolver.count_distinct_bitarray(n, s, p, q)
        time2 = (time.time() - start_time) * 1e9
        print(f"  BitArray: {time2:.0f} ns, 结果: {result2}")

        start_time = time.time()
        result3 = BitArraySolver.count_distinct_floyd(n, s, p, q)
        time3 = (time.time() - start_time) * 1e9
        print(f"  Floyd: {time3:.0f} ns, 结果: {result3}")

    print()

def complexity_analysis():
    """复杂度分析"""
    print("== 复杂度分析 ===")
    print("方法 1 (HashSet) :")
    print("  时间复杂度: O(N)")
    print("  空间复杂度: O(N)")
    print("  适用场景: 小规模数据 (N <= 10^6)")

    print("\n方法 2 (BitArray) :")
    print("  时间复杂度: O(N)")
    print("  空间复杂度: O(2^31/8) ≈ 256MB")
    print("  适用场景: 中等规模数据 (N <= 10^8)")

    print("\n方法 3 (Floyd 循环检测) :")

```

```
print(" 时间复杂度: O(循环长度) ")
print(" 空间复杂度: O(1) ")
print(" 适用场景: 大规模数据 (N > 10^8) ")

print("\n 工程化建议:")
print("1. 根据 N 的大小动态选择算法")
print("2. 使用 64 位整数避免整数溢出")
print("3. 对于竞赛题目, BitArray 方法通常是最佳选择")

def main():
    """主函数"""
    print("HackerRank Bit Array - 位数组问题")
    print("计算根据规则生成的序列中不同整数的个数")

    # 运行单元测试
    PerformanceTester.run_tests()

    # 运行性能测试
    PerformanceTester.performance_test()

    # 复杂度分析
    complexity_analysis()

    # 示例使用
    print("\n==== 示例使用 ====")
    examples = [
        (5, 1, 2, 1),
        (10, 1, 3, 0),
        (100, 1, 1, 1)
    ]

    for n, s, p, q in examples:
        result = BitArraySolver.count_distinct_with_validation(n, s, p, q)
        print(f"n={n}, s={s}, p={p}, q={q} -> 不同元素个数: {result}")

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

```
=====
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
```

```
using namespace std;
```

```
/**  
 * LeetCode 191 - 位 1 的个数  
 * 题目链接: https://leetcode.com/problems/number-of-1-bits/  
 * 题目描述: 编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数  
 *  
 * 解题思路:  
 * 方法 1: 逐位检查 - 检查每一位是否为 1  
 * 方法 2: Brian Kernighan 算法 - 利用 n & (n-1) 清除最低位的 1  
 * 方法 3: 查表法 - 预计算 8 位数的 1 的个数  
 * 方法 4: 并行计算 - 使用位运算并行计算  
 *  
 * 时间复杂度分析:  
 * 方法 1: O(32) - 固定 32 次循环  
 * 方法 2: O(k) - k 为 1 的个数  
 * 方法 3: O(1) - 查表操作  
 * 方法 4: O(1) - 常数时间位运算  
 *  
 * 空间复杂度分析:
```

```
* 方法 1: O(1) - 常数空间
* 方法 2: O(1) - 常数空间
* 方法 3: O(256) - 256 字节的查找表
* 方法 4: O(1) - 常数空间
*
* 工程化考量:
* 1. 性能优化: 选择最优算法
* 2. 可移植性: 处理不同整数大小
* 3. 边界处理: 处理 0 和最大值的特殊情况
*/
```

```
class BitManipulation {
public:
    /**
     * 方法 1: 逐位检查
     * 时间复杂度: O(32) - 固定 32 次循环
     * 空间复杂度: O(1)
     */
    static int hammingWeight1(uint32_t n) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            if (n & (1 << i)) {
                count++;
            }
        }
        return count;
    }

    /**
     * 方法 2: Brian Kernighan 算法
     * 利用 n & (n-1) 清除最低位的 1
     * 时间复杂度: O(k) - k 为 1 的个数
     * 空间复杂度: O(1)
     */
    static int hammingWeight2(uint32_t n) {
        int count = 0;
        while (n != 0) {
            n &= (n - 1);
            count++;
        }
        return count;
    }
}
```

```

/***
 * 方法 3: 查表法
 * 预计算 8 位数的 1 的个数
 * 时间复杂度: O(1) - 查表操作
 * 空间复杂度: O(256) - 256 字节的查找表
 */
static int hammingWeight3(uint32_t n) {
    static const int table[256] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
    } ;

    return table[n & 0xFF] +
        table[(n >> 8) & 0xFF] +
        table[(n >> 16) & 0xFF] +
        table[(n >> 24) & 0xFF];
}

/***
 * 方法 4: 并行计算
 * 使用位运算并行计算 1 的个数
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
static int hammingWeight4(uint32_t n) {
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555);
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333);
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
    n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF);
    n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF);
    return n;
}

```

```

/***
 * LeetCode 231 - 2 的幂
 * 题目链接: https://leetcode.com/problems/power-of-two/
 * 判断一个数是否是 2 的幂

```

```

*/
static bool isPowerOfTwo(int n) {
    if (n <= 0) return false;
    return (n & (n - 1)) == 0;
}

/***
 * LeetCode 342 - 4 的幂
 * 题目链接: https://leetcode.com/problems/power-of-four/
 * 判断一个数是否是 4 的幂
*/
static bool isPowerOfFour(int n) {
    if (n <= 0) return false;
    // 必须是 2 的幂，且 1 出现在奇数位
    return (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}

/***
 * LeetCode 136 - 只出现一次的数字
 * 题目链接: https://leetcode.com/problems/single-number/
 * 使用异或操作找出只出现一次的数字
*/
static int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

/***
 * LeetCode 137 - 只出现一次的数字 II
 * 题目链接: https://leetcode.com/problems/single-number-ii/
 * 每个数字出现三次，只有一个出现一次
*/
static int singleNumberII(vector<int>& nums) {
    int ones = 0, twos = 0;
    for (int num : nums) {
        ones = (ones ^ num) & ~twos;
        twos = (twos ^ num) & ~ones;
    }
    return ones;
}

```

```

/***
 * LeetCode 260 - 只出现一次的数字 III
 * 题目链接: https://leetcode.com/problems/single-number-iii/
 * 有两个数字只出现一次，其余出现两次
 */
static vector<int> singleNumberIII(vector<int>& nums) {
    int xor_all = 0;
    for (int num : nums) {
        xor_all ^= num;
    }

    // 找到最低位的 1
    int lowest_bit = xor_all & (-xor_all);

    int group1 = 0, group2 = 0;
    for (int num : nums) {
        if (num & lowest_bit) {
            group1 ^= num;
        } else {
            group2 ^= num;
        }
    }

    return {group1, group2};
}

```

```

/***
 * LeetCode 190 - 颠倒二进制位
 * 题目链接: https://leetcode.com/problems/reverse-bits/
 */
static uint32_t reverseBits(uint32_t n) {
    n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1);
    n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2);
    n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4);
    n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8);
    n = (n >> 16) | (n << 16);
    return n;
}
```

```

/***
 * LeetCode 338 - 比特位计数
 * 题目链接: https://leetcode.com/problems/counting-bits/

```

```

* 计算 0 到 n 每个数的二进制表示中 1 的个数
*/
static vector<int> countBits(int n) {
    vector<int> result(n + 1, 0);
    for (int i = 1; i <= n; i++) {
        result[i] = result[i & (i - 1)] + 1;
    }
    return result;
}

/***
* LeetCode 201 - 数字范围按位与
* 题目链接: https://leetcode.com/problems/bitwise-and-of-numbers-range/
* 计算区间[m, n]内所有数字的按位与
*/
static int rangeBitwiseAnd(int m, int n) {
    int shift = 0;
    while (m < n) {
        m >>= 1;
        n >>= 1;
        shift++;
    }
    return m << shift;
}

/***
* LeetCode 371 - 两整数之和
* 题目链接: https://leetcode.com/problems/sum-of-two-integers/
* 不使用+和-运算符计算两数之和
*/
static int getSum(int a, int b) {
    while (b != 0) {
        int carry = (a & b) << 1;
        a = a ^ b;
        b = carry;
    }
    return a;
};

class PerformanceTester {
public:
    static void testHammingWeight() {

```

```

cout << "==== 位 1 的个数性能测试 ===" << endl;

vector<uint32_t> test_cases = {
    0, 1, 15, 255, 1023, 0xFFFFFFFF, 0x55555555, 0xAAAAAAA
};

for (uint32_t n : test_cases) {
    cout << "测试数字: " << n << " (" << bitset<32>(n) << ")" << endl;

    auto start = chrono::high_resolution_clock::now();
    int r1 = BitManipulation::hammingWeight1(n);
    auto time1 = chrono::duration_cast<chrono::nanoseconds>(
        chrono::high_resolution_clock::now() - start).count();

    start = chrono::high_resolution_clock::now();
    int r2 = BitManipulation::hammingWeight2(n);
    auto time2 = chrono::duration_cast<chrono::nanoseconds>(
        chrono::high_resolution_clock::now() - start).count();

    start = chrono::high_resolution_clock::now();
    int r3 = BitManipulation::hammingWeight3(n);
    auto time3 = chrono::duration_cast<chrono::nanoseconds>(
        chrono::high_resolution_clock::now() - start).count();

    start = chrono::high_resolution_clock::now();
    int r4 = BitManipulation::hammingWeight4(n);
    auto time4 = chrono::duration_cast<chrono::nanoseconds>(
        chrono::high_resolution_clock::now() - start).count();

    cout << " 方法 1(逐位): " << r1 << ", 时间: " << time1 << " ns" << endl;
    cout << " 方法 2(Kernighan): " << r2 << ", 时间: " << time2 << " ns" << endl;
    cout << " 方法 3(查表): " << r3 << ", 时间: " << time3 << " ns" << endl;
    cout << " 方法 4(并行): " << r4 << ", 时间: " << time4 << " ns" << endl;
    cout << " 结果一致: " << (r1 == r2 && r2 == r3 && r3 == r4 ? "是" : "否") << endl;
    cout << endl;
}

static void runUnitTests() {
    cout << "==== 位操作单元测试 ===" << endl;

    // 测试 hammingWeight
    assert(BitManipulation::hammingWeight1(11) == 3);
}

```

```
assert(BitManipulation::hammingWeight2(11) == 3);
assert(BitManipulation::hammingWeight3(11) == 3);
assert(BitManipulation::hammingWeight4(11) == 3);
cout << "hammingWeight 测试通过" << endl;

// 测试 2 的幂
assert(BitManipulation::isPowerOfTwo(1) == true);
assert(BitManipulation::isPowerOfTwo(16) == true);
assert(BitManipulation::isPowerOfTwo(15) == false);
cout << "isPowerOfTwo 测试通过" << endl;

// 测试 4 的幂
assert(BitManipulation::isPowerOfFour(16) == true);
assert(BitManipulation::isPowerOfFour(8) == false);
cout << "isPowerOfFour 测试通过" << endl;

// 测试只出现一次的数字
vector<int> nums1 = {2, 2, 1};
assert(BitManipulation::singleNumber(nums1) == 1);
cout << "singleNumber 测试通过" << endl;

// 测试比特位计数
vector<int> result = BitManipulation::countBits(5);
vector<int> expected = {0, 1, 1, 2, 1, 2};
assert(result == expected);
cout << "countBits 测试通过" << endl;

// 测试数字范围按位与
assert(BitManipulation::rangeBitwiseAnd(5, 7) == 4);
cout << "rangeBitwiseAnd 测试通过" << endl;

// 测试两整数之和
assert(BitManipulation::getSum(3, 5) == 8);
cout << "getSum 测试通过" << endl;

cout << "所有单元测试通过!" << endl;
}

};

int main() {
    cout << "位操作算法实现" << endl;
    cout << "包含 LeetCode 多个位操作相关题目的解决方案" << endl;
    cout << "=====
```

```

// 运行单元测试
PerformanceTester::runUnitTests();

// 运行性能测试
PerformanceTester::testHammingWeight();

// 示例使用
cout << "==== 示例使用 ===" << endl;

// 测试 hammingWeight
uint32_t test_num = 11;
cout << "数字 " << test_num << " 的二进制表示中 1 的个数:" << endl;
cout << " 方法 1: " << BitManipulation::hammingWeight1(test_num) << endl;
cout << " 方法 2: " << BitManipulation::hammingWeight2(test_num) << endl;
cout << " 方法 3: " << BitManipulation::hammingWeight3(test_num) << endl;
cout << " 方法 4: " << BitManipulation::hammingWeight4(test_num) << endl;

// 测试 2 的幂
cout << "\n2 的幂判断:" << endl;
cout << "16 是 2 的幂: " << BitManipulation::isPowerOfTwo(16) << endl;
cout << "15 是 2 的幂: " << BitManipulation::isPowerOfTwo(15) << endl;

// 测试比特位计数
vector<int> bits_count = BitManipulation::countBits(5);
cout << "\n0 到 5 的比特位计数: ";
for (int count : bits_count) {
    cout << count << " ";
}
cout << endl;

return 0;
}
=====

文件: Code14_BitManipulation.py
=====
"""

位操作算法实现
包含 LeetCode 多个位操作相关题目的解决方案

题目列表:

```

题目列表:

1. LeetCode 191 - 位 1 的个数
2. LeetCode 231 - 2 的幂
3. LeetCode 342 - 4 的幂
4. LeetCode 136 - 只出现一次的数字
5. LeetCode 137 - 只出现一次的数字 II
6. LeetCode 260 - 只出现一次的数字 III
7. LeetCode 190 - 颠倒二进制位
8. LeetCode 338 - 比特位计数
9. LeetCode 201 - 数字范围按位与
10. LeetCode 371 - 两整数之和

时间复杂度分析：

- 位 1 的个数: $O(1)$ - 固定 32 位操作
- 2 的幂/4 的幂: $O(1)$ - 常数时间判断
- 只出现一次的数字: $O(n)$ - 线性扫描
- 颠倒二进制位: $O(1)$ - 固定操作
- 比特位计数: $O(n)$ - 线性计算
- 数字范围按位与: $O(1)$ - 常数时间
- 两整数之和: $O(1)$ - 常数时间循环

空间复杂度分析：

- 大部分算法: $O(1)$ - 常数空间
- 比特位计数: $O(n)$ - 存储结果数组
- 只出现一次的数字 III: $O(1)$ - 常数空间

工程化考量：

1. 性能优化：选择最优位操作算法
2. 边界处理：处理 0 和负数的特殊情况
3. 异常安全：使用异常处理机制
4. 可读性：添加详细注释说明位操作原理

"""

```
class BitManipulation:
```

```
    """位操作算法类"""
```

```
    @staticmethod
    def hamming_weight(n: int) -> int:
        """
```

```
        LeetCode 191 - 位 1 的个数
        计算无符号整数二进制表示中 1 的个数
```

方法 1: Brian Kernighan 算法

时间复杂度: $O(k)$ - k 为 1 的个数

空间复杂度: O(1)

"""

```
count = 0
while n:
    n &= n - 1 # 清除最低位的 1
    count += 1
return count
```

@staticmethod

```
def hamming_weight_parallel(n: int) -> int:
```

"""

方法 2: 并行计算

使用位运算并行计算 1 的个数

时间复杂度: O(1)

空间复杂度: O(1)

"""

```
n = (n & 0x55555555) + ((n >> 1) & 0x55555555)
n = (n & 0x33333333) + ((n >> 2) & 0x33333333)
n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F)
n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF)
n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF)
return n
```

@staticmethod

```
def is_power_of_two(n: int) -> bool:
```

"""

LeetCode 231 - 2 的幂

判断一个数是否是 2 的幂

原理: 2 的幂的二进制表示只有 1 个 1

时间复杂度: O(1)

空间复杂度: O(1)

"""

```
if n <= 0:
    return False
return (n & (n - 1)) == 0
```

@staticmethod

```
def is_power_of_four(n: int) -> bool:
```

"""

LeetCode 342 - 4 的幂

判断一个数是否是 4 的幂

原理：必须是 2 的幂，且 1 出现在奇数位

时间复杂度：O(1)

空间复杂度：O(1)

"""

```
if n <= 0:  
    return False
```

检查是否是 2 的幂且 1 出现在奇数位

```
return (n & (n - 1)) == 0 and (n & 0x55555555) != 0
```

@staticmethod

```
def single_number(nums: list) -> int:
```

"""

LeetCode 136 - 只出现一次的数字

数组中只有一个数字出现一次，其余出现两次

原理：使用异或操作，相同数字异或为 0

时间复杂度：O(n)

空间复杂度：O(1)

"""

```
result = 0
```

```
for num in nums:
```

```
    result ^= num
```

```
return result
```

@staticmethod

```
def single_number_ii(nums: list) -> int:
```

"""

LeetCode 137 - 只出现一次的数字 II

每个数字出现三次，只有一个出现一次

原理：使用位计数，统计每位出现 1 的次数

时间复杂度：O(n)

空间复杂度：O(1)

"""

```
ones, twos = 0, 0
```

```
for num in nums:
```

```
    # 更新 ones 和 twos
```

```
    ones = (ones ^ num) & ~twos
```

```
    twos = (twos ^ num) & ~ones
```

```
return ones
```

@staticmethod

```
def single_number_iii(nums: list) -> list:
```

```
"""
```

LeetCode 260 - 只出现一次的数字 III

有两个数字只出现一次，其余出现两次

原理：先异或得到两个数的异或结果，然后根据最低位 1 分组

时间复杂度：O(n)

空间复杂度：O(1)

```
"""
```

```
# 得到两个只出现一次的数字的异或
```

```
xor_all = 0
```

```
for num in nums:
```

```
    xor_all ^= num
```

```
# 找到最低位的 1（两个数字在该位不同）
```

```
lowest_bit = xor_all & -xor_all
```

```
group1, group2 = 0, 0
```

```
for num in nums:
```

```
    if num & lowest_bit:
```

```
        group1 ^= num
```

```
    else:
```

```
        group2 ^= num
```

```
return [group1, group2]
```

```
@staticmethod
```

```
def reverse_bits(n: int) -> int:
```

```
"""
```

LeetCode 190 - 颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位

原理：使用分治思想，先交换相邻位，再交换相邻 2 位，依此类推

时间复杂度：O(1)

空间复杂度：O(1)

```
"""
```

```
# 确保是 32 位无符号整数
```

```
n = n & 0xFFFFFFFF
```

```
# 分治交换
```

```
n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1)
```

```
n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2)
```

```
n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4)
```

```
n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8)
```

```
n = (n >> 16) | (n << 16)
```

```
return n & 0xFFFFFFFF
```

```
@staticmethod
```

```
def count_bits(n: int) -> list:
```

```
"""
```

```
LeetCode 338 - 比特位计数
```

```
计算 0 到 n 每个数的二进制表示中 1 的个数
```

原理：动态规划，利用 $i \& (i-1)$ 清除最低位的 1

时间复杂度： $O(n)$

空间复杂度： $O(n)$

```
"""
```

```
if n < 0:
```

```
    return []
```

```
result = [0] * (n + 1)
```

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

```
    result[i] = result[i & (i - 1)] + 1
```

```
return result
```

```
@staticmethod
```

```
def range_bitwise_and(m: int, n: int) -> int:
```

```
"""
```

```
LeetCode 201 - 数字范围按位与
```

```
计算区间 [m, n] 内所有数字的按位与
```

原理：找到 m 和 n 的公共前缀

时间复杂度： $O(1)$

空间复杂度： $O(1)$

```
"""
```

```
shift = 0
```

```
while m < n:
```

```
    m >>= 1
```

```
    n >>= 1
```

```
    shift += 1
```

```
return m << shift
```

```
@staticmethod
```

```
def get_sum(a: int, b: int) -> int:
```

```
"""
```

LeetCode 371 - 两整数之和
不使用+和-运算符计算两数之和

原理：使用位运算模拟加法

时间复杂度：O(1) - 最多 32 次循环

空间复杂度：O(1)

"""

```
# 32 位整数处理
```

```
MASK = 0xFFFFFFFF
```

```
MAX_INT = 0x7FFFFFFF
```

```
a &= MASK
```

```
b &= MASK
```

```
while b != 0:
```

```
    carry = ((a & b) << 1) & MASK
```

```
    a = (a ^ b) & MASK
```

```
    b = carry
```

```
# 处理负数
```

```
return a if a <= MAX_INT else ~(a ^ MASK)
```

```
class PerformanceTester:
```

```
    """性能测试工具类"""
```

```
@staticmethod
```

```
def test_hamming_weight():
```

```
    """测试位 1 的个数算法性能"""
```

```
    print("== 位 1 的个数性能测试 ==")
```

```
test_cases = [0, 1, 15, 255, 1023, 0xFFFFFFFF, 0x55555555, 0xAAAAAAA]
```

```
for n in test_cases:
```

```
    print(f"测试数字: {n} (二进制: {bin(n)})")
```

```
import time
```

```
start = time.time()
```

```
r1 = BitManipulation.hamming_weight(n)
```

```
time1 = (time.time() - start) * 1e9
```

```
start = time.time()
```

```
r2 = BitManipulation.hamming_weight_parallel(n)
time2 = (time.time() - start) * 1e9

print(f" Kernighan 算法: {r1}, 时间: {time1:.0f} ns")
print(f" 并行算法: {r2}, 时间: {time2:.0f} ns")
print(f" 结果一致: {'是' if r1 == r2 else '否'}")
print()

@staticmethod
def run_unit_tests():
    """运行单元测试"""
    print("== 位操作单元测试 ===")

    # 测试位 1 的个数
    assert BitManipulation.hamming_weight(11) == 3
    assert BitManipulation.hamming_weight_parallel(11) == 3
    print("hamming_weight 测试通过")

    # 测试 2 的幂
    assert BitManipulation.is_power_of_two(16) == True
    assert BitManipulation.is_power_of_two(15) == False
    print("is_power_of_two 测试通过")

    # 测试 4 的幂
    assert BitManipulation.is_power_of_four(16) == True
    assert BitManipulation.is_power_of_four(8) == False
    print("is_power_of_four 测试通过")

    # 测试只出现一次的数字
    nums1 = [2, 2, 1]
    assert BitManipulation.single_number(nums1) == 1
    print("single_number 测试通过")

    # 测试只出现一次的数字 II
    nums2 = [2, 2, 3, 2]
    assert BitManipulation.single_number_ii(nums2) == 3
    print("single_number_ii 测试通过")

    # 测试只出现一次的数字 III
    nums3 = [1, 2, 1, 3, 2, 5]
    result = BitManipulation.single_number_iii(nums3)
    assert sorted(result) == [3, 5]
    print("single_number_iii 测试通过")
```

```
# 测试颠倒二进制位
assert BitManipulation.reverse_bits(43261596) == 964176192
print("reverse_bits 测试通过")

# 测试比特位计数
result = BitManipulation.count_bits(5)
expected = [0, 1, 1, 2, 1, 2]
assert result == expected
print("count_bits 测试通过")

# 测试数字范围按位与
assert BitManipulation.range_bitwise_and(5, 7) == 4
print("range_bitwise_and 测试通过")

# 测试两整数之和
assert BitManipulation.get_sum(3, 5) == 8
print("get_sum 测试通过")

print("所有单元测试通过!")
```

```
@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n==== 复杂度分析 ====")

    algorithms = {
        "hamming_weight": {
            "time": "O(k) - k 为 1 的个数",
            "space": "O(1)",
            "desc": "Brian Kernighan 算法"
        },
        "is_power_of_two": {
            "time": "O(1)",
            "space": "O(1)",
            "desc": "检查二进制表示"
        },
        "single_number": {
            "time": "O(n)",
            "space": "O(1)",
            "desc": "异或操作"
        },
        "reverse_bits": {

```

```
        "time": "O(1)",
        "space": "O(1)",
        "desc": "分治交换"
    },
    "count_bits": {
        "time": "O(n)",
        "space": "O(n)",
        "desc": "动态规划"
    }
}

for name, info in algorithms.items():
    print(f'{name}:')
    print(f'    时间复杂度: {info["time"]}')
    print(f'    空间复杂度: {info["space"]}')
    print(f'    描述: {info["desc"]}')
    print()

def main():
    """主函数"""
    print("位操作算法实现")
    print("包含 LeetCode 多个位操作相关题目的解决方案")
    print("=" * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

    # 运行性能测试
    PerformanceTester.test_hamming_weight()

    # 复杂度分析
    PerformanceTester.complexity_analysis()

    # 示例使用
    print("==示例使用==")

    # 测试位 1 的个数
    test_num = 11
    print(f"数字 {test_num} 的二进制表示中 1 的个数:")
    print(f"    Kernighan 算法: {BitManipulation.hamming_weight(test_num)}")
    print(f"    并行算法: {BitManipulation.hamming_weight_parallel(test_num)}")
```

```

# 测试 2 的幂
print(f"\n2 的幂判断:")
print(f"16 是 2 的幂: {BitManipulation.is_power_of_two(16)}")
print(f"15 是 2 的幂: {BitManipulation.is_power_of_two(15)}")

# 测试只出现一次的数字
nums = [4, 1, 2, 1, 2]
print(f"\n只出现一次的数字: {BitManipulation.single_number(nums)}")

# 测试比特位计数
bits_count = BitManipulation.count_bits(5)
print(f"\n0 到 5 的比特位计数: {bits_count}")

# 测试两整数之和
print(f"\n3 + 5 = {BitManipulation.get_sum(3, 5)} (不使用+运算符)")

```

```

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

文件: Code14_SubsetSum.cpp

```

=====
```

```

#include <iostream>
#include <vector>
#include <numeric>
#include <set>
#include <algorithm>
#include <stdexcept>
#include <chrono>
#include <cstdlib> // 添加 rand() 函数支持
#include <ctime> // 添加时间种子支持

using namespace std;
using namespace std::chrono;

/**
 * LeetCode 416 Partition Equal Subset Sum - 子集和问题
 * 题目链接: https://leetcode.com/problems/partition-equal-subset-sum/
 * 题目描述: 给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，使得两个子集的
 * 元素和相等
 *
```

- * 解题思路:
- * 方法 1: 回溯法 - 暴力搜索所有子集, 时间复杂度高
- * 方法 2: 动态规划 (0-1 背包问题) - 使用二维 DP 数组
- * 方法 3: 动态规划优化 - 使用一维 DP 数组, 空间优化
- * 方法 4: Bitset 优化 - 使用位运算加速 DP
- *
- * 时间复杂度分析:
- * 方法 1: $O(2^N)$ - 指数级, 不可行
- * 方法 2: $O(N * \text{sum})$ - 伪多项式时间
- * 方法 3: $O(N * \text{sum})$ - 空间优化版本
- * 方法 4: $O(N * \text{sum}/64)$ - 使用 bitset 优化常数因子
- *
- * 空间复杂度分析:
- * 方法 1: $O(N)$ - 递归栈空间
- * 方法 2: $O(N * \text{sum})$ - 二维 DP 数组
- * 方法 3: $O(\text{sum})$ - 一维 DP 数组
- * 方法 4: $O(\text{sum}/64)$ - bitset 空间
- *
- * 工程化考量:
- * 1. 输入验证: 检查数组是否为空, 元素是否为正整数
- * 2. 边界处理: 处理总和为奇数的情况 (直接返回 false)
- * 3. 性能优化: 根据数据规模选择最优算法
- * 4. 内存管理: 使用 bitset 优化大内存消耗

*/

```

class SubsetSumSolver {
public:
    /**
     * 方法 1: 回溯法 (仅用于教学, 实际不可行)
     */
    static bool canPartitionBacktrack(const vector<int>& nums) {
        if (nums.empty()) return false;

        int totalSum = accumulate(nums.begin(), nums.end(), 0);
        if (totalSum % 2 != 0) return false;

        int target = totalSum / 2;
        return backtrack(nums, 0, 0, target);
    }

private:
    static bool backtrack(const vector<int>& nums, int index, int currentSum, int target) {
        if (currentSum == target) return true;

```

```

    if (currentSum > target || index >= nums.size()) return false;

    // 选择当前元素
    if (backtrack(nums, index + 1, currentSum + nums[index], target)) {
        return true;
    }

    // 不选择当前元素
    return backtrack(nums, index + 1, currentSum, target);
}

public:
/***
 * 方法 2: 动态规划 (二维 DP)
 */
static bool canPartitionDP2D(const vector<int>& nums) {
    if (nums.empty()) return false;

    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    int n = nums.size();

    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

    // 初始化: 和为 0 的子集总是存在
    for (int i = 0; i <= n; i++) {
        dp[i][0] = true;
    }

    // 动态规划填表
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            // 不选当前元素
            dp[i][j] = dp[i - 1][j];

            // 选当前元素
            if (j >= nums[i - 1]) {
                dp[i][j] = dp[i][j] || dp[i - 1][j - nums[i - 1]];
            }
        }
    }
}

```

```

        return dp[n][target];
    }

/***
 * 方法 3: 动态规划优化 (一维 DP)
 */
static bool canPartitionDP1D(const vector<int>& nums) {
    if (nums.empty()) return false;

    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    int n = nums.size();

    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int i = 0; i < n; i++) {
        // 从后往前遍历避免重复计算
        for (int j = target; j >= nums[i]; j--) {
            dp[j] = dp[j] || dp[j - nums[i]];
        }
    }

    // 提前终止
    if (dp[target]) return true;
}

return dp[target];
}

/***
 * 方法 4: Bitset 优化 (最优解)
 * 使用 C++ 的 bitset 进行优化
 */
static bool canPartitionBitset(const vector<int>& nums) {
    if (nums.empty()) return false;

    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;

```

```

// 使用 bitset，每个 bit 代表一个和是否可达
bitset<10001> bitSet; // 假设 target 最大为 10000
bitSet[0] = 1; // 和为 0 可达

for (int num : nums) {
    // 将 bitset 左移 num 位，然后与原来的 bitset 取或
    bitSet |= (bitSet << num);

    // 提前终止
    if (bitSet[target]) return true;
}

return bitSet[target];
}

/**
 * 方法 5：动态 bitset（适用于大 target）
 * 使用 vector<bool> 模拟动态 bitset
 */
static bool canPartitionDynamicBitset(const vector<int>& nums) {
    if (nums.empty()) return false;

    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;

    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        // 从后往前更新
        for (int j = target; j >= num; j--) {
            if (dp[j - num]) {
                dp[j] = true;
            }
        }
    }

    if (dp[target]) return true;
}

return dp[target];

```

```
}

/***
 * 优化版本：根据数据规模选择算法
 */
static bool canPartitionOptimized(const vector<int>& nums) {
    if (nums.empty()) return false;

    int totalSum = accumulate(nums.begin(), nums.end(), 0);
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    int n = nums.size();

    // 根据数据规模选择算法
    if (n <= 20) {
        return canPartitionBacktrack(nums);
    } else if (target <= 10000) {
        return canPartitionDP1D(nums);
    } else {
        return canPartitionDynamicBitset(nums);
    }
}

/***
 * 工程化改进版本：完整的异常处理
 */
static bool canPartitionWithValidation(const vector<int>& nums) {
    try {
        // 输入验证
        if (nums.empty()) {
            return false;
        }

        // 检查元素是否为正整数
        for (int num : nums) {
            if (num <= 0) {
                throw invalid_argument("All elements must be positive integers");
            }
        }

        return canPartitionOptimized(nums);
    }
}
```

```

    } catch (const exception& e) {
        cerr << "Error in canPartition: " << e.what() << endl;
        return false;
    }
}

};

/***
 * 性能测试工具类
 */
class PerformanceTester {
public:
    static void runTests() {
        cout << "==== LeetCode 416 Partition Equal Subset Sum - 单元测试 ===" << endl;

        // 测试用例
        vector<pair<vector<int>, bool>> testCases = {
            {{1, 5, 11, 5}, true},      // 可以平分
            {{1, 2, 3, 5}, false},     // 不能平分 (总和为奇数)
            {{}, false}                // 空数组
        };

        for (const auto& testCase : testCases) {
            const auto& nums = testCase.first;
            bool expected = testCase.second;
            bool result = SubsetSumSolver::canPartitionOptimized(nums);

            cout << "测试: ";
            for (int num : nums) cout << num << " ";
            cout << ", 期望=" << (expected ? "true" : "false")
                << ", 实际=" << (result ? "true" : "false")
                << ", " << (result == expected ? "通过" : "失败") << endl;
        }

        // 方法一致性测试
        cout << "\n==== 方法一致性测试 ===" << endl;
        vector<int> testNums = {1, 5, 10, 6};

        bool r1 = SubsetSumSolver::canPartitionDP2D(testNums);
        bool r2 = SubsetSumSolver::canPartitionDP1D(testNums);
        bool r3 = SubsetSumSolver::canPartitionBitset(testNums);
        bool r4 = SubsetSumSolver::canPartitionOptimized(testNums);
    }
}

```

```

cout << "二维 DP: " << (r1 ? "true" : "false") << endl;
cout << "一维 DP: " << (r2 ? "true" : "false") << endl;
cout << "Bitset: " << (r3 ? "true" : "false") << endl;
cout << "优化法: " << (r4 ? "true" : "false") << endl;
cout << "所有方法结果一致: " << (r1 == r2 && r2 == r3 && r3 == r4 ? "是" : "否") << endl;
}

static void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;

    // 生成测试数据
    vector<vector<int>> testCases = {
        generateRandomArray(20, 100),      // 小规模
        generateRandomArray(100, 100),     // 中等规模
        generateRandomArray(200, 100)      // 较大规模
    };

    for (int i = 0; i < testCases.size(); i++) {
        const auto& nums = testCases[i];
        cout << "测试用例 " << i + 1 << ": 数组长度=" << nums.size() << endl;

        // 测试一维 DP
        auto start = high_resolution_clock::now();
        bool result1 = SubsetSumSolver::canPartitionDP1D(nums);
        auto time1 = duration_cast<nanoseconds>(high_resolution_clock::now() - start);

        // 测试 Bitset 优化
        start = high_resolution_clock::now();
        bool result2 = SubsetSumSolver::canPartitionBitset(nums);
        auto time2 = duration_cast<nanoseconds>(high_resolution_clock::now() - start);

        cout << " 一维 DP: " << time1.count() << " ns, 结果: " << (result1 ? "true" :
"false") << endl;
        cout << "  Bitset: " << time2.count() << " ns, 结果: " << (result2 ? "true" :
"false") << endl;

        if (time1 > 0) {
            double ratio = static_cast<double>(time1) / time2;
            cout << "  Bitset 比一维 DP 快: " << ratio << "倍" << endl;
        }
        cout << endl;
    }
}

```

```
private:
    static vector<int> generateRandomArray(int size, int maxValue) {
        vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = rand() % maxValue + 1;
        }
        return arr;
    }
};

/***
 * 复杂度分析
 */
void complexityAnalysis() {
    cout << "==== 复杂度分析 ===" << endl;
    cout << "方法 1 (回溯法) :" << endl;
    cout << " 时间复杂度: O(2^N)" << endl;
    cout << " 空间复杂度: O(N)" << endl;
    cout << " 适用场景: N <= 20" << endl;

    cout << "\n方法 2 (二维 DP) :" << endl;
    cout << " 时间复杂度: O(N * sum)" << endl;
    cout << " 空间复杂度: O(N * sum)" << endl;
    cout << " 适用场景: 小规模数据" << endl;

    cout << "\n方法 3 (一维 DP) :" << endl;
    cout << " 时间复杂度: O(N * sum)" << endl;
    cout << " 空间复杂度: O(sum)" << endl;
    cout << " 适用场景: 中等规模数据" << endl;

    cout << "\n方法 4 (Bitset 优化) :" << endl;
    cout << " 时间复杂度: O(N * sum/64)" << endl;
    cout << " 空间复杂度: O(sum/64)" << endl;
    cout << " 适用场景: 大规模数据" << endl;

    cout << "\n工程化建议:" << endl;
    cout << "1. 使用 bitset 可以获得常数级别的性能优化" << endl;
    cout << "2. 注意 C++ bitset 的大小需要在编译时确定" << endl;
    cout << "3. 对于动态大小的 bitset, 使用 vector<bool>" << endl;
}

int main() {
```

```

cout << "LeetCode 416 Partition Equal Subset Sum - 子集和问题" << endl;
cout << "判断是否可以将数组分割成两个和相等的子集" << endl;

// 运行单元测试
PerformanceTester::runTests();

// 运行性能测试
PerformanceTester::performanceTest();

// 复杂度分析
complexityAnalysis();

// 示例使用
cout << "\n==> 示例使用 ==>" << endl;
vector<vector<int>> examples = {
    {1, 5, 11, 5},
    {1, 2, 3, 5},
    {1, 2, 3, 4, 5, 6, 7}
};

for (const auto& nums : examples) {
    bool result = SubsetSumSolver::canPartitionWithValidation(nums);
    cout << "数组: ";
    for (int num : nums) cout << num << " ";
    cout << " -> 是否可以平分: " << (result ? "true" : "false") << endl;
}

return 0;
}

```

=====

文件: Code14_SubsetSum.java

=====

```

package class032;

import java.util.*;
import java.util.stream.*;

/**
 * LeetCode 416 Partition Equal Subset Sum - 子集和问题
 * 题目链接: https://leetcode.com/problems/partition-equal-subset-sum/
 * 题目描述: 给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，使得两个子集的

```

元素和相等

*

* 解题思路:

- * 方法 1: 回溯法 - 暴力搜索所有子集, 时间复杂度高
- * 方法 2: 动态规划 (0-1 背包问题) - 使用二维 DP 数组
- * 方法 3: 动态规划优化 - 使用一维 DP 数组, 空间优化
- * 方法 4: Bitset 优化 - 使用位运算加速 DP

*

* 时间复杂度分析:

- * 方法 1: $O(2^N)$ - 指数级, 不可行
- * 方法 2: $O(N * \text{sum})$ - 伪多项式时间
- * 方法 3: $O(N * \text{sum})$ - 空间优化版本
- * 方法 4: $O(N * \text{sum}/32)$ - 使用 bitset 优化常数因子

*

* 空间复杂度分析:

- * 方法 1: $O(N)$ - 递归栈空间
- * 方法 2: $O(N * \text{sum})$ - 二维 DP 数组
- * 方法 3: $O(\text{sum})$ - 一维 DP 数组
- * 方法 4: $O(\text{sum}/32)$ - bitset 空间

*

* 工程化考量:

- * 1. 输入验证: 检查数组是否为空, 元素是否为正整数
- * 2. 边界处理: 处理总和为奇数的情况 (直接返回 false)
- * 3. 性能优化: 根据数据规模选择最优算法
- * 4. 内存管理: 使用 bitset 优化大内存消耗

*/

```
public class Code14_SubsetSum {  
  
    /**  
     * 方法 1: 回溯法 (仅用于教学, 实际不可行)  
     * 优点: 实现简单, 易于理解  
     * 缺点: 时间复杂度高, 仅适用于小规模数据  
     */  
  
    public static boolean canPartitionBacktrack(int[] nums) {  
        if (nums == null || nums.length == 0) return false;  
  
        int totalSum = Arrays.stream(nums).sum();  
        // 如果总和是奇数, 不可能平分  
        if (totalSum % 2 != 0) return false;  
  
        int target = totalSum / 2;  
        return backtrack(nums, 0, 0, target);  
    }  
}
```

```

}

private static boolean backtrack(int[] nums, int index, int currentSum, int target) {
    if (currentSum == target) return true;
    if (currentSum > target || index >= nums.length) return false;

    // 选择当前元素
    if (backtrack(nums, index + 1, currentSum + nums[index], target)) {
        return true;
    }

    // 不选择当前元素
    return backtrack(nums, index + 1, currentSum, target);
}

/***
 * 方法 2: 动态规划 (二维 DP)
 * 使用标准的 0-1 背包问题解法
 * dp[i][j] 表示前 i 个元素能否组成和为 j 的子集
 */
public static boolean canPartitionDP2D(int[] nums) {
    if (nums == null || nums.length == 0) return false;

    int totalSum = Arrays.stream(nums).sum();
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    int n = nums.length;

    boolean[][] dp = new boolean[n + 1][target + 1];

    // 初始化: 和为 0 的子集总是存在 (不选任何元素)
    for (int i = 0; i <= n; i++) {
        dp[i][0] = true;
    }

    // 动态规划填表
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            // 不选当前元素
            dp[i][j] = dp[i - 1][j];

            // 选当前元素 (如果 j >= nums[i-1])
            if (j >= nums[i - 1]) {
                dp[i][j] |= dp[i - 1][j - nums[i - 1]];
            }
        }
    }
}

```

```
        if (j >= nums[i - 1]) {
            dp[i][j] = dp[i][j] || dp[i - 1][j - nums[i - 1]];
        }
    }
}
```

```
return dp[n][target];
}
```

```
/**
```

```
* 方法 3: 动态规划优化 (一维 DP)
* 空间优化版本, 使用滚动数组
* 注意: 需要从后往前遍历避免重复计算
*/
```

```
public static boolean canPartitionDP1D(int[] nums) {
    if (nums == null || nums.length == 0) return false;
```

```
    int totalSum = Arrays.stream(nums).sum();
    if (totalSum % 2 != 0) return false;
```

```
    int target = totalSum / 2;
    int n = nums.length;
```

```
    boolean[] dp = new boolean[target + 1];
    dp[0] = true; // 和为 0 的子集总是存在
```

```
    for (int i = 0; i < n; i++) {
        // 从后往前遍历, 避免重复计算
        for (int j = target; j >= nums[i]; j--) {
            dp[j] = dp[j] || dp[j - nums[i]];
        }
    }
```

```
    // 提前终止: 如果已经找到目标值
    if (dp[target]) return true;
}
```

```
return dp[target];
}
```

```
/**
```

```
* 方法 4: Bitset 优化 (最优解)
* 使用 BitSet 来加速动态规划, 将时间复杂度优化常数因子
* 原理: 使用位运算来同时更新多个状态
```

```

*/
public static boolean canPartitionBitset(int[] nums) {
    if (nums == null || nums.length == 0) return false;

    int totalSum = Arrays.stream(nums).sum();
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;

    // 使用 BitSet 来记录可达的和
    BitSet bitSet = new BitSet(target + 1);
    bitSet.set(0); // 和为 0 总是可达

    for (int num : nums) {
        // 创建当前 bitSet 的副本，用于左移操作
        BitSet newBitSet = (BitSet) bitSet.clone();

        // 将 bitSet 左移 num 位，相当于给所有可达和加上 num
        for (int i = bitSet.nextSetBit(0); i >= 0; i = bitSet.nextSetBit(i + 1)) {
            int newSum = i + num;
            if (newSum <= target) {
                newBitSet.set(newSum);
            }
        }
    }

    bitSet.or(newBitSet); // 合并新的可达和

    // 提前终止
    if (bitSet.get(target)) {
        return true;
    }
}

return bitSet.get(target);
}

/***
 * 方法 5：更高效的 BitSet 优化
 * 使用单个 BitSet 和位运算技巧
 */
public static boolean canPartitionBitsetOptimized(int[] nums) {
    if (nums == null || nums.length == 0) return false;
}

```

```

int totalSum = Arrays.stream(nums).sum();
if (totalSum % 2 != 0) return false;

int target = totalSum / 2;

BitSet bitSet = new BitSet(target + 1);
bitSet.set(0);

for (int num : nums) {
    // 使用位运算技巧: bitSet 左移 num 位然后与原来的 bitSet 取或
    BitSet temp = bitSet.get(0, target - num + 1); // 获取 0 到 target-num 的子集
    for (int i = temp.nextSetBit(0); i >= 0; i = temp.nextSetBit(i + 1)) {
        int newSum = i + num;
        if (newSum <= target) {
            bitSet.set(newSum);
        }
    }
}

if (bitSet.get(target)) {
    return true;
}
}

return bitSet.get(target);
}

/***
 * 方法 6: 终极 Bitset 优化 (竞赛常用)
 * 使用位运算直接操作 long 数组, 性能最优
 */
public static boolean canPartitionBitsetUltimate(int[] nums) {
    if (nums == null || nums.length == 0) return false;

    int totalSum = Arrays.stream(nums).sum();
    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;

    // 使用 long 数组模拟 bitset, 每个 long 可以表示 64 位
    int size = (target >> 6) + 1; // target/64 + 1
    long[] bitset = new long[size];
    bitset[0] = 1L; // 第 0 位设为 1 (表示和为 0 可达)

```

```
for (int num : nums) {
    // 复制当前 bitset
    long[] newBitset = bitset.clone();

    // 对每个 long 进行位操作
    for (int i = 0; i < size; i++) {
        if (bitset[i] != 0) {
            // 计算左移 num 位后的位置
            int shift = num;
            int newIndex = i + (shift >> 6);
            int bitPos = shift & 63;

            if (newIndex < size) {
                newBitset[newIndex] |= bitset[i] << bitPos;

                // 处理跨 long 的情况
                if (bitPos > 0 && newIndex + 1 < size) {
                    newBitset[newIndex + 1] |= bitset[i] >>> (64 - bitPos);
                }
            }
        }
    }

    // 合并新的 bitset
    for (int i = 0; i < size; i++) {
        bitset[i] |= newBitset[i];
    }

    // 检查目标是否可达
    int targetIndex = target >> 6;
    int targetBit = target & 63;
    if ((bitset[targetIndex] & (1L << targetBit)) != 0) {
        return true;
    }
}

return false;
}

/**
 * 优化版本：根据数据规模选择算法
 */
public static boolean canPartitionOptimized(int[] nums) {
```

```
if (nums == null || nums.length == 0) return false;

int totalSum = Arrays.stream(nums).sum();
if (totalSum % 2 != 0) return false;

int target = totalSum / 2;
int n = nums.length;

// 根据数据规模选择算法
if (n <= 20) {
    // 小规模数据使用回溯法（更简单）
    return canPartitionBacktrack(nums);
} else if (target <= 10000) {
    // 中等规模使用一维 DP
    return canPartitionDP1D(nums);
} else {
    // 大规模数据使用 bitset 优化
    return canPartitionBitsetOptimized(nums);
}

}

/***
 * 工程化改进版本：完整的异常处理和验证
 */
public static boolean canPartitionWithValidation(int[] nums) {
    try {
        // 输入验证
        if (nums == null) {
            throw new IllegalArgumentException("Input array cannot be null");
        }
        if (nums.length == 0) {
            return false;
        }

        // 检查元素是否为正整数
        for (int num : nums) {
            if (num <= 0) {
                throw new IllegalArgumentException("All elements must be positive integers");
            }
        }

        return canPartitionOptimized(nums);
    }
}
```

```
        } catch (Exception e) {
            System.out.println("Error in canPartition: " + e.getMessage());
            return false;
        }
    }

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("== LeetCode 416 Partition Equal Subset Sum - 单元测试 ==");

    // 测试用例 1: 可以平分
    int[] nums1 = {1, 5, 11, 5};
    boolean expected1 = true;
    boolean result1 = canPartitionOptimized(nums1);
    System.out.printf("测试 1: %s, 期望=%b, 实际=%b, %s%n",
                      Arrays.toString(nums1), expected1, result1,
                      result1 == expected1 ? "通过" : "失败");

    // 测试用例 2: 不能平分 (总和为奇数)
    int[] nums2 = {1, 2, 3, 5};
    boolean expected2 = false;
    boolean result2 = canPartitionOptimized(nums2);
    System.out.printf("测试 2: %s, 期望=%b, 实际=%b, %s%n",
                      Arrays.toString(nums2), expected2, result2,
                      result2 == expected2 ? "通过" : "失败");

    // 测试用例 3: 边界情况 (空数组)
    int[] nums3 = {};
    boolean expected3 = false;
    boolean result3 = canPartitionOptimized(nums3);
    System.out.printf("测试 3: %s, 期望=%b, 实际=%b, %s%n",
                      Arrays.toString(nums3), expected3, result3,
                      result3 == expected3 ? "通过" : "失败");

    // 测试不同方法的结果一致性
    System.out.println("\n== 方法一致性测试 ==");
    int[] testNums = {1, 5, 10, 6};
    boolean r1 = canPartitionDP2D(testNums);
    boolean r2 = canPartitionDP1D(testNums);
    boolean r3 = canPartitionBitset(testNums);
    boolean r4 = canPartitionOptimized(testNums);
```

```
System.out.printf("二维 DP: %b%n", r1);
System.out.printf("一维 DP: %b%n", r2);
System.out.printf("Bitset: %b%n", r3);
System.out.printf("优化法: %b%n", r4);
System.out.printf("所有方法结果一致: %s%n",
                  (r1 == r2 && r2 == r3 && r3 == r4) ? "是" : "否");
}

/**
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n== 性能测试 ===");

    // 生成测试数据
    int[][] testCases = {
        generateRandomArray(20, 100),      // 小规模
        generateRandomArray(100, 100),     // 中等规模
        generateRandomArray(200, 100)      // 较大规模
    };

    for (int i = 0; i < testCases.length; i++) {
        int[] nums = testCases[i];
        System.out.printf("测试用例 %d: 数组长度=%d%n", i + 1, nums.length);

        // 测试一维 DP
        long startTime = System.nanoTime();
        boolean result1 = canPartitionDP1D(nums);
        long time1 = System.nanoTime() - startTime;
        System.out.printf(" 一维 DP: %d ns, 结果: %b%n", time1, result1);

        // 测试 Bitset 优化
        startTime = System.nanoTime();
        boolean result2 = canPartitionBitsetOptimized(nums);
        long time2 = System.nanoTime() - startTime;
        System.out.printf("  Bitset 优化: %d ns, 结果: %b%n", time2, result2);

        if (time1 > 0) {
            double ratio = (double) time1 / time2;
            System.out.printf("  Bitset 比一维 DP 快: %.2f 倍%n", ratio);
        }
    }
    System.out.println();
}
```

```

    }

}

/***
 * 生成随机数组用于测试
 */
private static int[] generateRandomArray(int size, int maxValue) {
    Random random = new Random();
    int[] arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(maxValue) + 1; // 生成 1 到 maxValue 的随机数
    }
    return arr;
}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("== 复杂度分析 ==");
    System.out.println("方法 1 (回溯法) :");
    System.out.println("  时间复杂度: O(2^N) - 指数级, 仅适用于 N <= 20");
    System.out.println("  空间复杂度: O(N) - 递归栈空间");

    System.out.println("\n 方法 2 (二维 DP) :");
    System.out.println("  时间复杂度: O(N * sum) - 伪多项式时间");
    System.out.println("  空间复杂度: O(N * sum) - 二维数组");

    System.out.println("\n 方法 3 (一维 DP) :");
    System.out.println("  时间复杂度: O(N * sum)");
    System.out.println("  空间复杂度: O(sum) - 一维数组");

    System.out.println("\n 方法 4 (Bitset 优化) :");
    System.out.println("  时间复杂度: O(N * sum/32) - 常数优化");
    System.out.println("  空间复杂度: O(sum/32) - bitset 空间");

    System.out.println("\n 工程化建议:");
    System.out.println("1. 对于小规模数据 (N <= 20), 回溯法更简单");
    System.out.println("2. 对于中等规模, 一维 DP 是平衡的选择");
    System.out.println("3. 对于大规模数据, 必须使用 bitset 优化");
    System.out.println("4. 注意提前终止优化: 一旦找到目标就返回");
}

```

```

public static void main(String[] args) {
    System.out.println("LeetCode 416 Partition Equal Subset Sum - 子集和问题");
    System.out.println("判断是否可以将数组分割成两个和相等的子集");

    // 运行单元测试
    runTests();

    // 运行性能测试
    performanceTest();

    // 复杂度分析
    complexityAnalysis();

    // 示例使用
    System.out.println("\n==== 示例使用 ====");
    int[][] examples = {
        {1, 5, 11, 5},
        {1, 2, 3, 5},
        {1, 2, 3, 4, 5, 6, 7}
    };

    for (int[] nums : examples) {
        boolean result = canPartitionWithValidation(nums);
        System.out.printf("数组 %s 是否可以平分: %b\n", Arrays.toString(nums), result);
    }
}

```

文件: Code14_SubsetSum.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

LeetCode 416 Partition Equal Subset Sum - 子集和问题

题目链接: <https://leetcode.com/problems/partition-equal-subset-sum/>

题目描述: 给定一个只包含正整数的非空数组, 判断是否可以将这个数组分割成两个子集, 使得两个子集的元素和相等

解题思路:

方法 1: 回溯法 - 暴力搜索所有子集, 时间复杂度高

方法 2: 动态规划 (0-1 背包问题) - 使用二维 DP 数组

方法 3: 动态规划优化 - 使用一维 DP 数组, 空间优化

方法 4: 使用 set 优化 - Python 特有的优化方式

时间复杂度分析:

方法 1: $O(2^N)$ - 指数级, 不可行

方法 2: $O(N * \text{sum})$ - 伪多项式时间

方法 3: $O(N * \text{sum})$ - 空间优化版本

方法 4: $O(N * 2^{(N/2)})$ - 折半搜索优化

空间复杂度分析:

方法 1: $O(N)$ - 递归栈空间

方法 2: $O(N * \text{sum})$ - 二维 DP 数组

方法 3: $O(\text{sum})$ - 一维 DP 数组

方法 4: $O(2^{(N/2)})$ - 存储折半结果

工程化考量:

1. 输入验证: 检查数组是否为空, 元素是否为正整数

2. 边界处理: 处理总和为奇数的情况 (直接返回 false)

3. 性能优化: 根据数据规模选择最优算法

4. Python 特性: 利用 Python 的 set 进行优化

"""

```
from typing import List
import time
from functools import lru_cache

class SubsetSumSolver:
    """子集和求解器类"""

    @staticmethod
    def can_partition_backtrack(nums: List[int]) -> bool:
        """
        方法 1: 回溯法 (仅用于教学, 实际不可行)
        """

Args:
```

nums: 整数数组

Returns:

bool: 是否可以平分

"""
if not nums:
 return False

```

total_sum = sum(nums)
if total_sum % 2 != 0:
    return False

target = total_sum // 2

@lru_cache(maxsize=None)
def backtrack(index: int, current_sum: int) -> bool:
    if current_sum == target:
        return True
    if current_sum > target or index >= len(nums):
        return False

    # 选择当前元素
    if backtrack(index + 1, current_sum + nums[index]):
        return True

    # 不选择当前元素
    return backtrack(index + 1, current_sum)

return backtrack(0, 0)

```

```

@staticmethod
def can_partition_dp_2d(nums: List[int]) -> bool:
    """

```

方法 2：动态规划（二维 DP）

Args:

nums: 整数数组

Returns:

bool: 是否可以平分

"""

if not nums:

return False

total_sum = sum(nums)

if total_sum % 2 != 0:

return False

target = total_sum // 2

n = len(nums)

```

# 创建二维 DP 数组
dp = [[False] * (target + 1) for _ in range(n + 1)]

# 初始化: 和为 0 总是可达
for i in range(n + 1):
    dp[i][0] = True

# 动态规划填表
for i in range(1, n + 1):
    for j in range(1, target + 1):
        # 不选当前元素
        dp[i][j] = dp[i - 1][j]

        # 选当前元素
        if j >= nums[i - 1]:
            dp[i][j] = dp[i][j] or dp[i - 1][j - nums[i - 1]]

return dp[n][target]

```

```

@staticmethod
def can_partition_dp_1d(nums: List[int]) -> bool:
    """

```

方法 3: 动态规划优化 (一维 DP)

Args:

nums: 整数数组

Returns:

bool: 是否可以平分

```

"""
if not nums:
    return False

```

```

total_sum = sum(nums)
if total_sum % 2 != 0:
    return False

```

```

target = total_sum // 2
n = len(nums)

```

```

dp = [False] * (target + 1)
dp[0] = True  # 和为 0 总是可达

```

```
for num in nums:
    # 从后往前遍历避免重复计算
    for j in range(target, num - 1, -1):
        if dp[j - num]:
            dp[j] = True

    # 提前终止
    if dp[target]:
        return True

return dp[target]
```

```
@staticmethod
def can_partition_set_optimized(nums: List[int]) -> bool:
    """
```

方法 4：使用 set 优化（Python 特有）
利用 Python 的 set 特性进行优化

Args:

nums: 整数数组

Returns:

bool: 是否可以平分

"""

```
if not nums:
    return False
```

```
total_sum = sum(nums)
```

```
if total_sum % 2 != 0:
```

```
    return False
```

```
target = total_sum // 2
```

```
# 使用 set 记录所有可达的和
reachable = {0}
```

```
for num in nums:
```

创建新的可达集合（避免在迭代中修改）

```
new_reachable = set()
```

```
for val in reachable:
```

```
    new_sum = val + num
```

```
    if new_sum == target:
```

```

        return True
    if new_sum < target:
        new_reachable.add(new_sum)

    # 合并新的可达集合
    reachable |= new_reachable

return target in reachable

@staticmethod
def can_partition_meet_in_middle(nums: List[int]) -> bool:
    """
    方法 5: 折半搜索 (适用于中等规模数据)
    将数组分成两半, 分别计算所有可能的子集和
    """

Args:
    nums: 整数数组

Returns:
    bool: 是否可以平分
    """
if not nums:
    return False

total_sum = sum(nums)
if total_sum % 2 != 0:
    return False

target = total_sum // 2

# 将数组分成两半
n = len(nums)
left = nums[:n//2]
right = nums[n//2:]

# 计算左半部分的所有子集和
left_sums = {0}
for num in left:
    new_sums = set()
    for s in left_sums:
        new_sum = s + num
        if new_sum == target:
            return True
        new_sums.add(new_sum)
    left_sums = new_sums

```

```

        if new_sum <= target:
            new_sums.add(new_sum)
            left_sums |= new_sums

# 计算右半部分的所有子集和
right_sums = {0}
for num in right:
    new_sums = set()
    for s in right_sums:
        new_sum = s + num
        if new_sum == target:
            return True
        if new_sum <= target:
            new_sums.add(new_sum)
    right_sums |= new_sums

# 检查是否存在 left_sum + right_sum = target
for left_sum in left_sums:
    needed = target - left_sum
    if needed in right_sums:
        return True

return False

```

```

@staticmethod
def can_partition_optimized(nums: List[int]) -> bool:
    """

```

优化版本：根据数据规模选择算法

Args:

nums: 整数数组

Returns:

bool: 是否可以平分

"""

if not nums:

return False

total_sum = sum(nums)

if total_sum % 2 != 0:

return False

target = total_sum // 2

```
n = len(nums)

# 根据数据规模选择算法
if n <= 20:
    return SubsetSumSolver.can_partition_backtrack(nums)
elif n <= 40:
    return SubsetSumSolver.can_partition_meet_in_middle(nums)
elif target <= 10000:
    return SubsetSumSolver.can_partition_dp_1d(nums)
else:
    return SubsetSumSolver.can_partition_set_optimized(nums)

@staticmethod
def can_partition_with_validation(nums: List[int]) -> bool:
    """
    工程化改进版本：完整的异常处理

    Args:
        nums: 整数数组

    Returns:
        bool: 是否可以平分

    Raises:
        ValueError: 当输入参数无效时
    """
    try:
        # 输入验证
        if nums is None:
            raise ValueError("Input array cannot be None")

        if not nums:
            return False

        # 检查元素是否为正整数
        for num in nums:
            if num <= 0:
                raise ValueError("All elements must be positive integers")

        return SubsetSumSolver.can_partition_optimized(nums)

    except Exception as e:
        print(f"Error in can_partition: {e}")
```

```

        return False

class PerformanceTester:
    """性能测试类"""

    @staticmethod
    def run_tests():
        """运行单元测试"""
        print("== LeetCode 416 Partition Equal Subset Sum - 单元测试 ==")

        test_cases = [
            ([1, 5, 11, 5], True),      # 可以平分
            ([1, 2, 3, 5], False),     # 不能平分
            ([], False)                # 空数组
        ]

        for nums, expected in test_cases:
            result = SubsetSumSolver.can_partition_optimized(nums)
            status = "通过" if result == expected else "失败"
            print(f"测试 {nums}, 期望={expected}, 实际={result}, {status}")

        # 方法一致性测试
        print("\n== 方法一致性测试 ==")
        test_nums = [1, 5, 10, 6]

        results = [
            SubsetSumSolver.can_partition_dp_2d(test_nums),
            SubsetSumSolver.can_partition_dp_1d(test_nums),
            SubsetSumSolver.can_partition_set_optimized(test_nums),
            SubsetSumSolver.can_partition_optimized(test_nums)
        ]

        methods = ["二维 DP", "一维 DP", "Set 优化", "优化法"]
        for method, result in zip(methods, results):
            print(f"{method}: {result}")

        consistent = all(r == results[0] for r in results)
        print(f"所有方法结果一致: {'是' if consistent else '否'}")

    @staticmethod
    def performance_test():
        """运行性能测试"""

```

```

print("\n==== 性能测试 ===")

# 生成测试数据
test_cases = [
    PerformanceTester.generate_random_array(20, 100),      # 小规模
    PerformanceTester.generate_random_array(100, 100),     # 中等规模
    PerformanceTester.generate_random_array(200, 100)      # 较大规模
]

for i, nums in enumerate(test_cases):
    print(f"测试用例 {i + 1}: 数组长度={len(nums)}")

    # 测试一维 DP
    start_time = time.time()
    result1 = SubsetSumSolver.can_partition_dp_1d(nums)
    time1 = time.time() - start_time

    # 测试 Set 优化
    start_time = time.time()
    result2 = SubsetSumSolver.can_partition_set_optimized(nums)
    time2 = time.time() - start_time

    print(f" 一维 DP: {time1:.6f} 秒, 结果: {result1}")
    print(f"  Set 优化: {time2:.6f} 秒, 结果: {result2}")

    if time2 > 0:
        ratio = time1 / time2
        print(f"  Set 优化比一维 DP 快: {ratio:.2f} 倍")
    print()

@staticmethod
def generate_random_array(size: int, max_value: int) -> List[int]:
    """生成随机数组"""
    import random
    return [random.randint(1, max_value) for _ in range(size)]


def complexity_analysis():
    """复杂度分析"""
    print("==== 复杂度分析 ===")
    print("方法 1 (回溯法) :")
    print("  时间复杂度: O(2^N)")
    print("  空间复杂度: O(N)")

```

```
print(" 适用场景: N <= 20")\n\nprint("\n方法 2 (二维 DP) :")\nprint("  时间复杂度: O(N * sum)")\nprint("  空间复杂度: O(N * sum)")\nprint("  适用场景: 小规模数据")\n\nprint("\n方法 3 (一维 DP) :")\nprint("  时间复杂度: O(N * sum)")\nprint("  空间复杂度: O(sum)")\nprint("  适用场景: 中等规模数据")\n\nprint("\n方法 4 (Set 优化) :")\nprint("  时间复杂度: O(N * 2^(N/2))")\nprint("  空间复杂度: O(2^(N/2))")\nprint("  适用场景: 中等规模数据")\n\nprint("\n方法 5 (折半搜索) :")\nprint("  时间复杂度: O(2^(N/2))")\nprint("  空间复杂度: O(2^(N/2))")\nprint("  适用场景: N <= 40")\n\nprint("\nPython 工程化建议:")\nprint("1. 利用 Python 的 set 特性进行优化")\nprint("2. 使用 lru_cache 进行记忆化搜索")\nprint("3. 根据数据规模动态选择最优算法")\n\n\ndef main():\n    """主函数"""\n    print("LeetCode 416 Partition Equal Subset Sum - 子集和问题")\n    print("判断是否可以将数组分割成两个和相等的子集")\n\n    # 运行单元测试\n    PerformanceTester.run_tests()\n\n    # 运行性能测试\n    PerformanceTester.performance_test()\n\n    # 复杂度分析\n    complexity_analysis()\n\n    # 示例使用
```

```

print("\n==== 示例使用 ===")
examples = [
    [1, 5, 11, 5],
    [1, 2, 3, 5],
    [1, 2, 3, 4, 5, 6, 7]
]

for nums in examples:
    result = SubsetSumSolver.can_partition_with_validation(nums)
    print(f"数组 {nums} 是否可以平分: {result}")

if __name__ == "__main__":
    main()

```

=====

文件: Code15_BitmaskDP.java

=====

```

package class032;

import java.util.*;

/**
 * Codeforces 580D Kefa and Dishes - 状态压缩 DP
 * 题目链接: https://codeforces.com/problemset/problem/580/D
 * 题目描述: 有 n 道菜, 每道菜有一个满意度值。选择 m 道菜来吃, 使得总满意度最大。
 * 此外, 有些菜组合在一起吃会有额外的满意度加成 (当 y 紧跟在 x 后面时)。
 *
 * 解题思路:
 * 方法 1: 状态压缩 DP - 使用 bitmask 表示已选择的菜品
 * 方法 2: 记忆化搜索 - 递归+记忆化
 * 方法 3: 迭代 DP - 使用二维 DP 数组
 *
 * 时间复杂度分析:
 * 方法 1: O(2^N * N^2) - 状态压缩 DP 的标准复杂度
 * 方法 2: O(2^N * N) - 记忆化搜索, 实际运行可能更快
 * 方法 3: O(2^N * N^2) - 与方法 1 相同, 但实现方式不同
 *
 * 空间复杂度分析:
 * 方法 1: O(2^N * N) - DP 数组空间
 * 方法 2: O(2^N * N) - 记忆化数组空间
 * 方法 3: O(2^N * N) - DP 数组空间

```

```
*  
* 工程化考量：  
* 1. 状态表示：使用 bitmask 高效表示子集  
* 2. 转移优化：利用位运算快速进行状态转移  
* 3. 内存优化：对于大 N 使用滚动数组或优化状态表示  
* 4. 边界处理：处理 m=0, 1 的特殊情况  
*/
```

```
public class Code15_BitmaskDP {  
  
    /**  
     * 方法 1：状态压缩 DP（迭代版本）  
     * 使用 dp[mask][last] 表示在 mask 状态下最后吃的菜是 last 时的最大满意度  
     */  
  
    public static long kefaAndDishesDP(int n, int m, int[] satisfaction, int[][] rules) {  
        if (m <= 0 || n <= 0) return 0;  
        if (m > n) m = n; // 不能选择超过 n 道菜  
  
        int totalStates = 1 << n;  
        long[][] dp = new long[totalStates][n];  
  
        // 初始化：只选择一道菜的情况  
        for (int i = 0; i < n; i++) {  
            int mask = 1 << i;  
            dp[mask][i] = satisfaction[i];  
        }  
  
        long maxSatisfaction = 0;  
  
        // 遍历所有状态  
        for (int mask = 1; mask < totalStates; mask++) {  
            int count = Integer.bitCount(mask);  
  
            for (int last = 0; last < n; last++) {  
                // 如果 last 不在 mask 中，跳过  
                if ((mask & (1 << last)) == 0) continue;  
  
                // 如果当前状态正好选择了 m 道菜，更新最大值  
                if (count == m) {  
                    maxSatisfaction = Math.max(maxSatisfaction, dp[mask][last]);  
                }  
  
                // 如果已经达到 m 道菜，不再继续添加  
            }  
        }  
    }  
}
```

```

    if (count >= m) continue;

    // 尝试添加新的菜
    for (int next = 0; next < n; next++) {
        // 如果 next 已经在 mask 中，跳过
        if ((mask & (1 << next)) != 0) continue;

        int newMask = mask | (1 << next);
        long newValue = dp[mask][last] + satisfaction[next];

        // 添加规则加成
        if (rules[last][next] > 0) {
            newValue += rules[last][next];
        }

        if (newValue > dp[newMask][next]) {
            dp[newMask][next] = newValue;
        }
    }

    return maxSatisfaction;
}

/***
 * 方法2：记忆化搜索（递归版本）
 * 使用递归+记忆化，代码更清晰
 */
public static long kefaAndDishesMemo(int n, int m, int[] satisfaction, int[][][] rules) {
    if (m <= 0 || n <= 0) return 0;
    if (m > n) m = n;

    long[][] memo = new long[1 << n][n];
    for (long[] row : memo) {
        Arrays.fill(row, -1);
    }

    long maxSatisfaction = 0;
    for (int i = 0; i < n; i++) {
        maxSatisfaction = Math.max(maxSatisfaction,
            dfs(1 << i, i, 1, n, m, satisfaction, rules, memo));
    }
}

```

```

    return maxSatisfaction;
}

private static long dfs(int mask, int last, int count, int n, int m,
                      int[] satisfaction, int[][] rules, long[][] memo) {
    if (count == m) {
        return satisfaction[last];
    }

    if (memo[mask][last] != -1) {
        return memo[mask][last];
    }

    long maxValue = 0;
    for (int next = 0; next < n; next++) {
        if ((mask & (1 << next)) != 0) continue;

        int newMask = mask | (1 << next);
        long value = dfs(newMask, next, count + 1, n, m, satisfaction, rules, memo);

        // 添加规则加成
        if (rules[last][next] > 0) {
            value += rules[last][next];
        }

        maxValue = Math.max(maxValue, value);
    }

    memo[mask][last] = maxValue + satisfaction[last];
    return memo[mask][last];
}

/***
 * 方法3：优化版本 - 使用一维DP数组
 * 空间优化，只记录当前状态的最大值
 */
public static long kefaAndDishesOptimized(int n, int m, int[] satisfaction, int[][] rules) {
    if (m <= 0 || n <= 0) return 0;
    if (m > n) m = n;

    int totalStates = 1 << n;
    long[] dp = new long[totalStates];

```

```

// 初始化单菜品状态
for (int i = 0; i < n; i++) {
    dp[1 << i] = satisfaction[i];
}

long maxSatisfaction = 0;

for (int mask = 1; mask < totalStates; mask++) {
    int count = Integer.bitCount(mask);
    if (count > m) continue;

    for (int last = 0; last < n; last++) {
        if ((mask & (1 << last)) == 0) continue;

        if (count == m) {
            maxSatisfaction = Math.max(maxSatisfaction, dp[mask]);
        }

        for (int next = 0; next < n; next++) {
            if ((mask & (1 << next)) != 0) continue;

            int newMask = mask | (1 << next);
            long newValue = dp[mask] + satisfaction[next];

            // 添加规则加成
            if (rules[last][next] > 0) {
                newValue += rules[last][next];
            }

            if (newValue > dp[newMask]) {
                dp[newMask] = newValue;
            }
        }
    }
}

return maxSatisfaction;
}

/**
 * 工程化改进版本：完整的异常处理和验证
 */

```

```

public static long kefaAndDishesWithValidation(int n, int m, int[] satisfaction, int[][] rules) {
    try {
        // 输入验证
        if (n <= 0 || m <= 0) {
            throw new IllegalArgumentException("n and m must be positive");
        }
        if (satisfaction == null || satisfaction.length != n) {
            throw new IllegalArgumentException("satisfaction array size must match n");
        }
        if (rules == null || rules.length != n || rules[0].length != n) {
            throw new IllegalArgumentException("rules must be n x n matrix");
        }

        // 检查满意度值是否合理
        for (int s : satisfaction) {
            if (s < 0) {
                throw new IllegalArgumentException("satisfaction values must be non-negative");
            }
        }

        return kefaAndDishesOptimized(n, m, satisfaction, rules);
    } catch (Exception e) {
        System.out.println("Error in kefaAndDishes: " + e.getMessage());
        return 0;
    }
}

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("==> Codeforces 580D Kefa and Dishes - 单元测试 ==>");

    // 测试用例 1: 简单情况
    int n1 = 3, m1 = 2;
    int[] satisfaction1 = {1, 2, 3};
    int[][] rules1 = new int[3][3];
    rules1[0][1] = 10; // 菜 0 后吃菜 1 有 10 点加成

    long expected1 = 13; // 菜 0(1) + 菜 1(2) + 加成 10 = 13
}

```

```

long result1 = kefaAndDishesWithValidation(n1, m1, satisfaction1, rules1);
System.out.printf("测试 1: n=%d, m=%d, 期望=%d, 实际=%d, %s%n",
                  n1, m1, expected1, result1,
                  result1 == expected1 ? "通过" : "失败");

// 测试用例 2: 边界情况 m=1
int n2 = 3, m2 = 1;
int[] satisfaction2 = {5, 10, 15};
int[][] rules2 = new int[3][3];

long expected2 = 15; // 最大满意度是 15
long result2 = kefaAndDishesWithValidation(n2, m2, satisfaction2, rules2);
System.out.printf("测试 2: n=%d, m=%d, 期望=%d, 实际=%d, %s%n",
                  n2, m2, expected2, result2,
                  result2 == expected2 ? "通过" : "失败");

// 测试不同方法的结果一致性
System.out.println("\n==== 方法一致性测试 ====");
int testN = 4, testM = 3;
int[] testSatisfaction = {1, 2, 3, 4};
int[][] testRules = new int[4][4];
testRules[0][1] = 5;
testRules[1][2] = 10;

long r1 = kefaAndDishesDP(testN, testM, testSatisfaction, testRules);
long r2 = kefaAndDishesMemo(testN, testM, testSatisfaction, testRules);
long r3 = kefaAndDishesOptimized(testN, testM, testSatisfaction, testRules);

System.out.printf("迭代 DP: %d%n", r1);
System.out.printf("记忆化搜索: %d%n", r2);
System.out.printf("优化版本: %d%n", r3);
System.out.printf("所有方法结果一致: %s%n",
                  (r1 == r2 && r2 == r3) ? "是" : "否");
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");

    // 测试不同规模的数据
    int[] testSizes = {10, 15, 18}; // 状态压缩 DP 通常处理 N<=20
}

```

```

for (int n : testSizes) {
    int m = n / 2; // 选择一半的菜品
    int[] satisfaction = generateRandomArray(n, 100);
    int[][] rules = generateRandomRules(n, 20);

    System.out.printf("n = %d, m = %d:%n", n, m);

    // 测试迭代 DP
    long startTime = System.nanoTime();
    long result1 = kefaAndDishesDP(n, m, satisfaction, rules);
    long time1 = System.nanoTime() - startTime;
    System.out.printf(" 迭代 DP: %d ns, 结果: %d%n", time1, result1);

    // 测试记忆化搜索
    startTime = System.nanoTime();
    long result2 = kefaAndDishesMemo(n, m, satisfaction, rules);
    long time2 = System.nanoTime() - startTime;
    System.out.printf(" 记忆化搜索: %d ns, 结果: %d%n", time2, result2);

    // 测试优化版本
    startTime = System.nanoTime();
    long result3 = kefaAndDishesOptimized(n, m, satisfaction, rules);
    long time3 = System.nanoTime() - startTime;
    System.out.printf(" 优化版本: %d ns, 结果: %d%n", time3, result3);

    System.out.println();
}

}

/***
 * 生成随机满意度数组
 */
private static int[] generateRandomArray(int size, int maxValue) {
    Random random = new Random();
    int[] arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(maxValue) + 1;
    }
    return arr;
}

*/

```

```

* 生成随机规则矩阵
*/
private static int[][] generateRandomRules(int size, int maxBonus) {
    Random random = new Random();
    int[][] rules = new int[size][size];
    // 随机生成一些规则
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i != j && random.nextDouble() < 0.3) { // 30%的概率有规则
                rules[i][j] = random.nextInt(maxBonus) + 1;
            }
        }
    }
    return rules;
}

/**
 * 复杂度分析
*/
public static void complexityAnalysis() {
    System.out.println("== 复杂度分析 ==");
    System.out.println("状态压缩 DP 的核心思想:");
    System.out.println(" 使用二进制位掩码表示子集状态");
    System.out.println(" 每个状态 mask 表示选择了哪些菜品");
    System.out.println(" 状态转移: 从当前状态扩展到包含新菜品的状态");

    System.out.println("\n时间复杂度分析:");
    System.out.println(" 状态数量:  $2^N$ ");
    System.out.println(" 每个状态需要遍历 N 个可能的最后菜品");
    System.out.println(" 每个最后菜品需要尝试 N 个新菜品");
    System.out.println(" 总复杂度:  $O(2^N * N^2)$ ");

    System.out.println("\n空间复杂度分析:");
    System.out.println(" DP 数组大小:  $2^N * N$ ");
    System.out.println(" 实际可处理规模: N <= 20 ( $2^{20} \approx 1$  百万状态)");

    System.out.println("\n优化技巧:");
    System.out.println("1. 使用 Integer.bitCount() 快速计算已选菜品数");
    System.out.println("2. 利用位运算快速判断菜品是否已选");
    System.out.println("3. 对于固定 m 的情况, 可以提前终止");
    System.out.println("4. 使用一维 DP 数组进行空间优化");
}

```

```

public static void main(String[] args) {
    System.out.println("Codeforces 580D Kefa and Dishes - 状态压缩 DP");
    System.out.println("使用位掩码技术解决组合优化问题");

    // 运行单元测试
    runTests();

    // 运行性能测试
    performanceTest();

    // 复杂度分析
    complexityAnalysis();

    // 示例使用
    System.out.println("\n==== 示例使用 ====");
    int n = 4, m = 3;
    int[] satisfaction = {5, 10, 15, 20};
    int[][] rules = new int[4][4];
    rules[0][1] = 5;
    rules[1][2] = 10;
    rules[2][3] = 15;

    long result = kefaAndDishesWithValidation(n, m, satisfaction, rules);
    System.out.printf("n=%d, m=%d 时的最大满意度: %d\n", n, m, result);
    System.out.println("菜品满意度: " + Arrays.toString(satisfaction));
    System.out.println("规则加成矩阵:");
    for (int i = 0; i < n; i++) {
        System.out.println(Arrays.toString(rules[i]));
    }
}
}

```

=====

文件: Code15_BitSetApplications.cpp

=====

```

#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>

```

```
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
```

```
using namespace std;
```

```
/**  
 * 位集应用算法实现  
 * 包含多个实际应用场景的位集算法  
 *  
 * 题目来源:  
 * 1. LeetCode 78 - 子集  
 * 2. LeetCode 90 - 子集 II  
 * 3. LeetCode 401 - 二进制手表  
 * 4. LeetCode 477 - 汉明距离总和  
 * 5. 组合数学问题  
 * 6. 状态压缩动态规划  
 *  
 * 时间复杂度分析:  
 * - 子集生成:  $O(2^n * n)$  - 指数级复杂度  
 * - 二进制手表:  $O(12 * 60)$  - 常数时间  
 * - 汉明距离总和:  $O(n)$  - 线性复杂度  
 * - 状态压缩:  $O(2^n * n)$  - 指数级复杂度  
 *  
 * 空间复杂度分析:  
 * - 子集生成:  $O(2^n)$  - 指数级空间  
 * - 二进制手表:  $O(1)$  - 常数空间  
 * - 汉明距离总和:  $O(1)$  - 常数空间  
 * - 状态压缩:  $O(2^n)$  - 指数级空间  
 *  
 * 工程化考量:  
 * 1. 性能优化: 使用位运算优化计算
```

* 2. 内存管理：处理大规模数据时的内存使用

* 3. 边界处理：处理空集和边界情况

* 4. 可读性：添加详细注释说明算法原理

*/

```
class BitSetApplications {
public:
    /**
     * LeetCode 78 - 子集
     * 题目链接: https://leetcode.com/problems/subsets/
     * 生成数组的所有可能子集
     *
     * 方法 1：位运算枚举
     * 时间复杂度: O(2^n * n)
     * 空间复杂度: O(2^n)
     */
    static vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int total = 1 << n; // 2^n 个子集
        vector<vector<int>> result;

        for (int mask = 0; mask < total; mask++) {
            vector<int> subset;
            for (int i = 0; i < n; i++) {
                if (mask & (1 << i)) {
                    subset.push_back(nums[i]);
                }
            }
            result.push_back(subset);
        }

        return result;
    }

    /**
     * LeetCode 90 - 子集 II
     * 题目链接: https://leetcode.com/problems/subsets-ii/
     * 生成包含重复元素的数组的所有可能子集（去重）
     *
     * 方法：先排序，然后使用位运算枚举，跳过重复元素
     * 时间复杂度: O(2^n * n)
     * 空间复杂度: O(2^n)
     */
}
```

```

static vector<vector<int>> subsetsWithDup(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int n = nums.size();
    int total = 1 << n;
    vector<vector<int>> result;
    unordered_set<string> seen;

    for (int mask = 0; mask < total; mask++) {
        vector<int> subset;
        string key;
        bool valid = true;

        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                // 检查是否跳过重复元素
                if (i > 0 && nums[i] == nums[i-1] && !(mask & (1 << (i-1)))) {
                    valid = false;
                    break;
                }
                subset.push_back(nums[i]);
                key += to_string(nums[i]) + ",";
            }
        }

        if (valid && seen.find(key) == seen.end()) {
            result.push_back(subset);
            seen.insert(key);
        }
    }

    return result;
}

/***
 * LeetCode 401 - 二进制手表
 * 题目链接: https://leetcode.com/problems/binary-watch/
 * 显示所有可能的二进制手表时间
 *
 * 方法: 枚举所有可能的小时和分钟组合
 * 时间复杂度: O(12 * 60) = O(720)
 * 空间复杂度: O(n) - n 为结果数量
 */
static vector<string> readBinaryWatch(int turnedOn) {

```

```

vector<string> result;

for (int hour = 0; hour < 12; hour++) {
    for (int minute = 0; minute < 60; minute++) {
        if (__builtin_popcount(hour) + __builtin_popcount(minute) == turnedOn) {
            string time = to_string(hour) + ":" +
                (minute < 10 ? "0" : "") + to_string(minute);
            result.push_back(time);
        }
    }
}

return result;
}

/***
 * LeetCode 477 - 汉明距离总和
 * 题目链接: https://leetcode.com/problems/total-hamming-distance/
 * 计算数组中所有数对之间的汉明距离总和
 *
 * 方法: 按位统计, 对于每一位, 计算 0 和 1 的个数
 * 时间复杂度: O(32 * n) = O(n)
 * 空间复杂度: O(1)
 */
static int totalHammingDistance(vector<int>& nums) {
    int total = 0;
    int n = nums.size();

    // 对每一位进行统计
    for (int i = 0; i < 32; i++) {
        int countOnes = 0;
        for (int num : nums) {
            if (num & (1 << i)) {
                countOnes++;
            }
        }
        total += countOnes * (n - countOnes);
    }

    return total;
}

/***

```

```

* 组合数学：计算组合数 C(n, k)
* 使用位运算枚举所有 k 个元素的组合
*
* 方法：Gosper's Hack 算法
* 时间复杂度：O(C(n, k))
* 空间复杂度：O(1)
*/
static vector<vector<int>> combinations(int n, int k) {
    vector<vector<int>> result;

    if (k == 0 || k > n) return result;

    int mask = (1 << k) - 1; // 初始组合：最低 k 位为 1
    int limit = 1 << n;

    while (mask < limit) {
        // 将当前掩码转换为组合
        vector<int> comb;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                comb.push_back(i);
            }
        }
        result.push_back(comb);

        // 使用 Gosper's Hack 获取下一个组合
        int rightmost = mask & -mask;
        int next = mask + rightmost;
        mask = next | (((mask ^ next) / rightmost) >> 2);
    }

    return result;
}

/**
* 状态压缩动态规划：旅行商问题（TSP）
* 使用位集表示访问过的城市状态
*
* 方法：动态规划 + 状态压缩
* 时间复杂度：O(2^n * n^2)
* 空间复杂度：O(2^n * n)
*/
static int tsp(vector<vector<int>>& graph) {

```

```

int n = graph.size();
if (n == 0) return 0;

// dp[mask][i]: 访问过 mask 表示的城市集合，当前在城市 i 的最小代价
vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

// 初始化：从城市 0 出发
dp[1][0] = 0;

// 动态规划
for (int mask = 1; mask < (1 << n); mask++) {
    for (int i = 0; i < n; i++) {
        if (!(mask & (1 << i))) continue; // 城市 i 不在当前集合中

        for (int j = 0; j < n; j++) {
            if (mask & (1 << j)) continue; // 城市 j 已经在集合中
            if (graph[i][j] == INT_MAX) continue; // 不可达

            int newMask = mask | (1 << j);
            if (dp[mask][i] != INT_MAX) {
                dp[newMask][j] = min(dp[newMask][j], dp[mask][i] + graph[i][j]);
            }
        }
    }
}

// 找到回到起点的最小代价
int result = INT_MAX;
int finalMask = (1 << n) - 1;
for (int i = 1; i < n; i++) {
    if (dp[finalMask][i] != INT_MAX && graph[i][0] != INT_MAX) {
        result = min(result, dp[finalMask][i] + graph[i][0]);
    }
}

return result == INT_MAX ? -1 : result;
}

/***
 * 位集在集合操作中的应用
 * 实现集合的并集、交集、差集等操作
 */
static void setOperationsDemo() {

```

```

cout << "==== 位集集合操作演示 ===" << endl;

// 使用 bitset 表示集合
bitset<8> setA("10101010"); // 集合 A: {1, 3, 5, 7}
bitset<8> setB("11001100"); // 集合 B: {2, 3, 6, 7}

cout << "集合 A: " << setA << " (十进制: " << setA.to_ulong() << ")" << endl;
cout << "集合 B: " << setB << " (十进制: " << setB.to_ulong() << ")" << endl;

// 并集
bitset<8> unionSet = setA | setB;
cout << "并集 A ∪ B: " << unionSet << endl;

// 交集
bitset<8> intersectSet = setA & setB;
cout << "交集 A ∩ B: " << intersectSet << endl;

// 差集
bitset<8> diffSet = setA & ~setB;
cout << "差集 A - B: " << diffSet << endl;

// 对称差
bitset<8> symDiffSet = setA ^ setB;
cout << "对称差 A Δ B: " << symDiffSet << endl;

// 子集判断
bool isSubset = (setA & setB) == setA;
cout << "A 是 B 的子集: " << (isSubset ? "是" : "否") << endl;
}

};

class PerformanceTester {
public:
    static void testSubsets() {
        cout << "==== 子集生成性能测试 ===" << endl;

        vector<int> nums = {1, 2, 3, 4, 5};

        auto start = chrono::high_resolution_clock::now();
        auto result = BitSetApplications::subsets(nums);
        auto time = chrono::duration_cast<chrono::microseconds>(
            chrono::high_resolution_clock::now() - start).count();
    }
};

```

```

cout << "数组大小: " << nums.size() << endl;
cout << "生成子集数量: " << result.size() << endl;
cout << "耗时: " << time << " μs" << endl;

// 显示前几个子集
cout << "前 5 个子集: " << endl;
for (int i = 0; i < min(5, (int)result.size()); i++) {
    cout << " {";
    for (int j = 0; j < result[i].size(); j++) {
        cout << result[i][j];
        if (j < result[i].size() - 1) cout << ", ";
    }
    cout << "}" << endl;
}
}

static void testCombinations() {
    cout << "\n==== 组合生成性能测试 ===" << endl;

    int n = 10, k = 3;

    auto start = chrono::high_resolution_clock::now();
    auto result = BitSetApplications::combinations(n, k);
    auto time = chrono::duration_cast<chrono::microseconds>(
        chrono::high_resolution_clock::now() - start).count();

    cout << "C(" << n << ", " << k << ") = " << result.size() << endl;
    cout << "耗时: " << time << " μs" << endl;

    // 显示前几个组合
    cout << "前 5 个组合: " << endl;
    for (int i = 0; i < min(5, (int)result.size()); i++) {
        cout << " {";
        for (int j = 0; j < result[i].size(); j++) {
            cout << result[i][j];
            if (j < result[i].size() - 1) cout << ", ";
        }
        cout << "}" << endl;
    }
}

static void testHammingDistance() {
    cout << "\n==== 汉明距离总和测试 ===" << endl;
}

```

```
vector<int> nums = {4, 14, 2, 8};

auto start = chrono::high_resolution_clock::now();
int result = BitSetApplications::totalHammingDistance(nums);
auto time = chrono::duration_cast<chrono::nanoseconds>(
    chrono::high_resolution_clock::now() - start).count();

cout << "数组: ";
for (int num : nums) cout << num << " ";
cout << endl;
cout << "汉明距离总和: " << result << endl;
cout << "耗时: " << time << " ns" << endl;
}

static void runUnitTests() {
    cout << "==== 位集应用单元测试 ===" << endl;

    // 测试子集生成
    vector<int> nums = {1, 2, 3};
    auto subsets = BitSetApplications::subsets(nums);
    assert(subsets.size() == 8);
    cout << "子集生成测试通过" << endl;

    // 测试汉明距离总和
    vector<int> nums2 = {4, 14, 2};
    int hamming = BitSetApplications::totalHammingDistance(nums2);
    assert(hamming > 0);
    cout << "汉明距离总和测试通过" << endl;

    // 测试组合生成
    auto combs = BitSetApplications::combinations(5, 2);
    assert(combs.size() == 10);
    cout << "组合生成测试通过" << endl;

    // 测试二进制手表
    auto times = BitSetApplications::readBinaryWatch(1);
    assert(!times.empty());
    cout << "二进制手表测试通过" << endl;

    cout << "所有单元测试通过!" << endl;
}
};
```

```
int main() {
    cout << "位集应用算法实现" << endl;
    cout << "包含子集生成、组合数学、状态压缩等应用" << endl;
    cout << "======" << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testSubsets();
    PerformanceTester::testCombinations();
    PerformanceTester::testHammingDistance();

    // 演示集合操作
    BitSetApplications::setOperationsDemo();

    // 示例使用
    cout << "\n== 示例使用 ==" << endl;

    // 子集生成示例
    vector<int> nums = {1, 2, 3};
    cout << "数组 [1, 2, 3] 的所有子集:" << endl;
    auto allSubsets = BitSetApplications::subsets(nums);
    for (const auto& subset : allSubsets) {
        cout << "{";
        for (int i = 0; i < subset.size(); i++) {
            cout << subset[i];
            if (i < subset.size() - 1) cout << ", ";
        }
        cout << "}" << endl;
    }

    // 组合生成示例
    cout << "\nC(5, 2) 的所有组合:" << endl;
    auto combs = BitSetApplications::combinations(5, 2);
    for (const auto& comb : combs) {
        cout << "{";
        for (int i = 0; i < comb.size(); i++) {
            cout << comb[i];
            if (i < comb.size() - 1) cout << ", ";
        }
        cout << "}" << endl;
    }
}
```

```
    }  
  
    return 0;  
}
```

文件: Code15_BitSetApplications.py

"""

位集应用算法实现

包含多个实际应用场景的位集算法

题目来源:

1. LeetCode 78 - 子集
2. LeetCode 90 - 子集 II
3. LeetCode 401 - 二进制手表
4. LeetCode 477 - 汉明距离总和
5. 组合数学问题
6. 状态压缩动态规划

时间复杂度分析:

- 子集生成: $O(2^n * n)$ - 指数级复杂度
- 二进制手表: $O(12 * 60)$ - 常数时间
- 汉明距离总和: $O(n)$ - 线性复杂度
- 状态压缩: $O(2^n * n)$ - 指数级复杂度

空间复杂度分析:

- 子集生成: $O(2^n)$ - 指数级空间
- 二进制手表: $O(1)$ - 常数空间
- 汉明距离总和: $O(1)$ - 常数空间
- 状态压缩: $O(2^n)$ - 指数级空间

工程化考量:

1. 性能优化: 使用位运算优化计算
 2. 内存管理: 处理大规模数据时的内存使用
 3. 边界处理: 处理空集和边界情况
 4. 可读性: 添加详细注释说明算法原理
- """

```
import time  
from typing import List, Set, Tuple  
import itertools
```

```
class BitSetApplications:  
    """位集应用算法类"""  
  
    @staticmethod  
    def subsets(nums: List[int]) -> List[List[int]]:  
        """
```

LeetCode 78 - 子集
生成数组的所有可能子集

方法：位运算枚举

时间复杂度： $O(2^n * n)$

空间复杂度： $O(2^n)$

```
"""
```

```
n = len(nums)  
total = 1 << n # 2^n 个子集  
result = []
```

```
for mask in range(total):  
    subset = []  
    for i in range(n):  
        if mask & (1 << i):  
            subset.append(nums[i])  
    result.append(subset)
```

```
return result
```

```
@staticmethod  
def subsets_with_dup(nums: List[int]) -> List[List[int]]:  
    """
```

LeetCode 90 - 子集 II
生成包含重复元素的数组的所有可能子集（去重）

方法：先排序，然后使用位运算枚举，跳过重复元素

时间复杂度： $O(2^n * n)$

空间复杂度： $O(2^n)$

```
"""
```

```
nums.sort()  
n = len(nums)  
total = 1 << n  
result = []  
seen = set()
```

```

for mask in range(total):
    subset = []
    key_parts = []
    valid = True

    for i in range(n):
        if mask & (1 << i):
            # 检查是否跳过重复元素
            if i > 0 and nums[i] == nums[i-1] and not (mask & (1 << (i-1))):
                valid = False
                break
            subset.append(nums[i])
            key_parts.append(str(nums[i]))

    if valid:
        key = ", ".join(key_parts)
        if key not in seen:
            result.append(subset)
            seen.add(key)

return result

```

```

@staticmethod
def read_binary_watch(turned_on: int) -> List[str]:
    """

```

LeetCode 401 - 二进制手表
显示所有可能的二进制手表时间

方法：枚举所有可能的小时和分钟组合

时间复杂度： $O(12 * 60) = O(720)$

空间复杂度： $O(n)$ – n 为结果数量

"""

```
result = []
```

```
for hour in range(12):
```

```
    for minute in range(60):
```

```
        if bin(hour).count('1') + bin(minute).count('1') == turned_on:
```

```
            time_str = f'{hour}:{minute:02d}'
```

```
            result.append(time_str)
```

```
return result
```

```
@staticmethod
```

```

def total_hamming_distance(nums: List[int]) -> int:
    """
    LeetCode 477 - 汉明距离总和
    计算数组中所有数对之间的汉明距离总和

    方法: 按位统计, 对于每一位, 计算 0 和 1 的个数
    时间复杂度: O(32 * n) = O(n)
    空间复杂度: O(1)
    """
    total = 0
    n = len(nums)

    # 对每一位进行统计 (假设 32 位整数)
    for i in range(32):
        count_ones = 0
        for num in nums:
            if num & (1 << i):
                count_ones += 1
        total += count_ones * (n - count_ones)

    return total

```

```

@staticmethod
def combinations(n: int, k: int) -> List[List[int]]:
    """
    组合数学: 计算组合数 C(n, k)
    使用位运算枚举所有 k 个元素的组合

```

方法: Gosper's Hack 算法

时间复杂度: $O(C(n, k))$

空间复杂度: $O(1)$

"""

```

if k == 0 or k > n:
    return []

```

```

result = []

```

mask = (1 << k) - 1 # 初始组合: 最低 k 位为 1

```

limit = 1 << n

```

```

while mask < limit:

```

将当前掩码转换为组合

```

comb = []

```

```

for i in range(n):

```

```

        if mask & (1 << i):
            comb.append(i)
        result.append(comb)

    # 使用 Gosper's Hack 获取下一个组合
    rightmost = mask & -mask
    next_mask = mask + rightmost
    mask = next_mask | (((mask ^ next_mask) // rightmost) >> 2)

return result

```

```

@staticmethod
def tsp(graph: List[List[int]]) -> int:
    """

```

状态压缩动态规划：旅行商问题（TSP）
使用位集表示访问过的城市状态

方法：动态规划 + 状态压缩

时间复杂度： $O(2^n * n^2)$

空间复杂度： $O(2^n * n)$

"""

```

n = len(graph)
if n == 0:
    return 0

```

dp[mask][i]: 访问过 mask 表示的城市集合，当前在城市 i 的最小代价

INF = float('inf')

```
dp = [[INF] * n for _ in range(1 << n)]
```

初始化：从城市 0 出发

```
dp[1][0] = 0
```

动态规划

```

for mask in range(1 << n):
    for i in range(n):
        if not (mask & (1 << i)): # 城市 i 不在当前集合中
            continue

        for j in range(n):
            if mask & (1 << j): # 城市 j 已经在集合中
                continue
            if graph[i][j] == INF: # 不可达
                continue

```

```

        new_mask = mask | (1 << j)
        if dp[mask][i] != INF:
            dp[new_mask][j] = min(dp[new_mask][j], dp[mask][i] + graph[i][j])

# 找到回到起点的最小代价
result = INF
final_mask = (1 << n) - 1
for i in range(1, n):
    if dp[final_mask][i] != INF and graph[i][0] != INF:
        result = min(result, dp[final_mask][i] + graph[i][0])

return -1 if result == INF else int(result)

@staticmethod
def set_operations_demo():
    """
位集在集合操作中的应用
实现集合的并集、交集、差集等操作
"""
    print("== 位集集合操作演示 ==")

    # 使用整数表示集合（8位）
    set_a = 0b10101010  # 集合 A: {1, 3, 5, 7}
    set_b = 0b11001100  # 集合 B: {2, 3, 6, 7}

    print(f"集合 A: {bin(set_a)} (十进制: {set_a})")
    print(f"集合 B: {bin(set_b)} (十进制: {set_b})")

    # 并集
    union_set = set_a | set_b
    print(f"并集 A ∪ B: {bin(union_set)}")

    # 交集
    intersect_set = set_a & set_b
    print(f"交集 A ∩ B: {bin(intersect_set)}")

    # 差集
    diff_set = set_a & ~set_b
    print(f"差集 A - B: {bin(diff_set)}")

    # 对称差
    sym_diff_set = set_a ^ set_b

```

```

print(f"对称差 A Δ B: {bin(sym_diff_set)}")

# 子集判断
is_subset = (set_a & set_b) == set_a
print(f"A 是 B 的子集: {'是' if is_subset else '否'}")

class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def test_subsets():
        """测试子集生成性能"""
        print("== 子集生成性能测试 ==")

        nums = [1, 2, 3, 4, 5]

        start = time.time()
        result = BitSetApplications.subsets(nums)
        elapsed = (time.time() - start) * 1e6 # 微秒

        print(f"数组大小: {len(nums)}")
        print(f"生成子集数量: {len(result)}")
        print(f"耗时: {elapsed:.2f} μs")

        # 显示前几个子集
        print("前 5 个子集:")
        for i in range(min(5, len(result))):
            print(f" {result[i]}")

    @staticmethod
    def test_combinations():
        """测试组合生成性能"""
        print("\n== 组合生成性能测试 ==")

        n, k = 10, 3

        start = time.time()
        result = BitSetApplications.combinations(n, k)
        elapsed = (time.time() - start) * 1e6 # 微秒

        print(f"C({n}, {k}) = {len(result)}")
        print(f"耗时: {elapsed:.2f} μs")

```

```
# 显示前几个组合
print("前 5 个组合:")
for i in range(min(5, len(result))):
    print(f" {result[i]}")

@staticmethod
def test_hamming_distance():
    """测试汉明距离总和性能"""
    print("\n==== 汉明距离总和测试 ====")

    nums = [4, 14, 2, 8]

    start = time.time()
    result = BitSetApplications.total_hamming_distance(nums)
    elapsed = (time.time() - start) * 1e9 # 纳秒

    print(f"数组: {nums}")
    print(f"汉明距离总和: {result}")
    print(f"耗时: {elapsed:.2f} ns")

@staticmethod
def run_unit_tests():
    """运行单元测试"""
    print("==== 位集应用单元测试 ====")

    # 测试子集生成
    nums = [1, 2, 3]
    subsets = BitSetApplications.subsets(nums)
    assert len(subsets) == 8
    print("子集生成测试通过")

    # 测试汉明距离总和
    nums2 = [4, 14, 2]
    hamming = BitSetApplications.total_hamming_distance(nums2)
    assert hamming > 0
    print("汉明距离总和测试通过")

    # 测试组合生成
    combs = BitSetApplications.combinations(5, 2)
    assert len(combs) == 10
    print("组合生成测试通过")
```

```
# 测试二进制手表
times = BitSetApplications.read_binary_watch(1)
assert len(times) > 0
print("二进制手表测试通过")

print("所有单元测试通过!")

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n==== 复杂度分析 ====")

    algorithms = {
        "subsets": {
            "time": "O(2^n * n)",
            "space": "O(2^n)",
            "desc": "位运算枚举所有子集"
        },
        "combinations": {
            "time": "O(C(n, k))",
            "space": "O(C(n, k))",
            "desc": "Gosper's Hack 算法"
        },
        "total_hamming_distance": {
            "time": "O(n)",
            "space": "O(1)",
            "desc": "按位统计 0 和 1 的个数"
        },
        "tsp": {
            "time": "O(2^n * n^2)",
            "space": "O(2^n * n)",
            "desc": "状态压缩动态规划"
        }
    }

    for name, info in algorithms.items():
        print(f"{name}:")
        print(f"    时间复杂度: {info['time']}")
        print(f"    空间复杂度: {info['space']}")
        print(f"    描述: {info['desc']}")
        print()
```

```
def main():
    """主函数"""
    print("位集应用算法实现")
    print("包含子集生成、组合数学、状态压缩等应用")
    print("=" * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

    # 运行性能测试
    PerformanceTester.test_subsets()
    PerformanceTester.test_combinations()
    PerformanceTester.test_hamming_distance()

    # 复杂度分析
    PerformanceTester.complexity_analysis()

    # 演示集合操作
    BitSetApplications.set_operations_demo()

    # 示例使用
    print("\n==== 示例使用 ====")

    # 子集生成示例
    nums = [1, 2, 3]
    print(f"数组 {nums} 的所有子集:")
    all_subsets = BitSetApplications.subsets(nums)
    for subset in all_subsets:
        print(f"  {subset}")

    # 组合生成示例
    print(f"\nC(5, 2) 的所有组合:")
    combs = BitSetApplications.combinations(5, 2)
    for comb in combs:
        print(f"  {comb}")

    # 二进制手表示例
    print(f"\n二进制手表显示 1 个 LED 亮的时间:")
    times = BitSetApplications.read_binary_watch(1)
    for i, t in enumerate(times[:5]):  # 显示前 5 个
        print(f"  {t}", end=" " if i < 4 else "\n")
```

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

=====

文件: Code16_BitMaskDP.cpp

=====

```
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
```

```
using namespace std;
```

```
/**
 * 状态压缩动态规划算法实现
 * 使用位掩码技术解决复杂的组合优化问题
 *
 * 题目来源:
 * 1. LeetCode 464 - 我能赢吗
 * 2. LeetCode 691 - 贴纸拼词
 * 3. LeetCode 943 - 最短超级串
 * 4. LeetCode 1125 - 最小的必要团队
 * 5. 旅行商问题 (TSP)
 * 6. 哈密顿路径问题
 *
```

- * 时间复杂度分析:
 - * - 状态压缩 DP: $O(2^n * n)$ 或 $O(2^n * n^2)$
 - * - 空间复杂度: $O(2^n)$ 或 $O(2^n * n)$
 - *
- * 工程化考量:
 - * 1. 状态表示: 使用位掩码紧凑表示状态
 - * 2. 状态转移: 优化状态转移方程
 - * 3. 内存优化: 使用滚动数组或位运算压缩
 - * 4. 边界处理: 处理空状态和终止状态

```

class BitMaskDP {
public:
    /**
     * LeetCode 464 - 我能赢吗
     * 题目链接: https://leetcode.com/problems/can-i-win/
     * 两个玩家轮流选择 1 到 maxChoosableInteger 的数字，先达到或超过 desiredTotal 的玩家获胜
     *
     * 方法: 记忆化搜索 + 状态压缩
     * 时间复杂度:  $O(2^{maxChoosableInteger} * maxChoosableInteger)$ 
     * 空间复杂度:  $O(2^{maxChoosableInteger})$ 
     */
    static bool canIWin(int maxChoosableInteger, int desiredTotal) {
        if (maxChoosableInteger >= desiredTotal) return true;
        if (maxChoosableInteger * (maxChoosableInteger + 1) / 2 < desiredTotal) return false;

        vector<int> memo(1 << maxChoosableInteger, -1); // -1: 未计算, 0: 输, 1: 赢

        function<int(int, int)> dfs = [&](int state, int currentTotal) -> int {
            if (memo[state] != -1) return memo[state];

            for (int i = 0; i < maxChoosableInteger; i++) {
                if (state & (1 << i)) continue; // 数字 i+1 已经被选过

                int newState = state | (1 << i);
                int newTotal = currentTotal + i + 1;

                // 如果当前选择能直接获胜，或者对手会输
                if (newTotal >= desiredTotal || dfs(newState, newTotal) == 0) {
                    return memo[state] = 1;
                }
            }
        };
    }
}

```

```

        return memo[state] = 0;
    }

    return dfs(0, 0) == 1;
}

/***
 * LeetCode 691 - 贴纸拼词
 * 题目链接: https://leetcode.com/problems/stickers-to-spell-word/
 * 用给定的贴纸拼出目标单词，求最少需要的贴纸数量
 *
 * 方法: 状态压缩 BFS
 * 时间复杂度: O(2^len(target) * n * len(stickers))
 * 空间复杂度: O(2^len(target))
 */
static int minStickers(vector<string>& stickers, string target) {
    int n = target.size();
    int totalStates = 1 << n;
    vector<int> dp(totalStates, INT_MAX);
    dp[0] = 0; // 空状态需要 0 张贴纸

    for (int state = 0; state < totalStates; state++) {
        if (dp[state] == INT_MAX) continue;

        for (const string& sticker : stickers) {
            int newState = state;

            // 尝试用当前贴纸覆盖目标字符
            vector<int> count(26, 0);
            for (char c : sticker) count[c - 'a']++;

            for (int i = 0; i < n; i++) {
                if (state & (1 << i)) continue; // 该位置已经被覆盖
                if (count[target[i] - 'a'] > 0) {
                    count[target[i] - 'a']--;
                    newState |= (1 << i);
                }
            }
        }

        if (newState != state) {
            dp[newState] = min(dp[newState], dp[state] + 1);
        }
    }
}

```

```

}

return dp[totalStates - 1] == INT_MAX ? -1 : dp[totalStates - 1];
}

/***
 * LeetCode 943 - 最短超级串
 * 题目链接: https://leetcode.com/problems/find-the-shortest-superstring/
 * 找到包含所有字符串的最短超级字符串
 *
 * 方法: 状态压缩 DP + 重叠计算
 * 时间复杂度: O(2^n * n^2)
 * 空间复杂度: O(2^n * n)
 */

static string shortestSuperstring(vector<string>& words) {
    int n = words.size();

    // 计算重叠度
    vector<vector<int>> overlap(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) continue;
            int len = min(words[i].size(), words[j].size());
            for (int k = len; k >= 0; k--) {
                if (words[i].substr(words[i].size() - k) == words[j].substr(0, k)) {
                    overlap[i][j] = k;
                    break;
                }
            }
        }
    }

    // DP 状态: dp[mask][last] 表示使用 mask 集合, 以 last 结尾的最短长度
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));
    vector<vector<int>> parent(1 << n, vector<int>(n, -1));

    // 初始化
    for (int i = 0; i < n; i++) {
        dp[1 << i][i] = words[i].size();
    }

    // 动态规划
    for (int mask = 1; mask < (1 << n); mask++) {

```

```

for (int last = 0; last < n; last++) {
    if (!(mask & (1 << last)) || dp[mask][last] == INT_MAX) continue;

    for (int next = 0; next < n; next++) {
        if (mask & (1 << next)) continue;

        int newMask = mask | (1 << next);
        int newLen = dp[mask][last] + words[next].size() - overlap[last][next];

        if (newLen < dp[newMask][next]) {
            dp[newMask][next] = newLen;
            parent[newMask][next] = last;
        }
    }
}

// 重建结果
int finalMask = (1 << n) - 1;
int last = min_element(dp[finalMask].begin(), dp[finalMask].end()) -
dp[finalMask].begin();
int minLen = dp[finalMask][last];

// 回溯构建字符串
string result;
int mask = finalMask;
vector<int> path;

while (mask > 0) {
    path.push_back(last);
    int prev = parent[mask][last];
    if (prev == -1) break;
    mask ^= (1 << last);
    last = prev;
}

reverse(path.begin(), path.end());

result = words[path[0]];
for (int i = 1; i < path.size(); i++) {
    int overlapLen = overlap[path[i-1]][path[i]];
    result += words[path[i]].substr(overlapLen);
}

```

```

    return result;
}

/***
 * LeetCode 1125 - 最小的必要团队
 * 题目链接: https://leetcode.com/problems/smallest-sufficient-team/
 * 找到覆盖所有技能的最小团队
 *
 * 方法: 状态压缩 DP
 * 时间复杂度: O(2^m * n) - m 为技能数量, n 为人员数量
 * 空间复杂度: O(2^m)
 */
static vector<int> smallestSufficientTeam(vector<string>& req_skills, vector<vector<string>>& people) {
    int m = req_skills.size();
    int n = people.size();

    // 技能到索引的映射
    unordered_map<string, int> skillToIndex;
    for (int i = 0; i < m; i++) {
        skillToIndex[req_skills[i]] = i;
    }

    // 将每个人的技能转换为位掩码
    vector<int> peopleSkills(n, 0);
    for (int i = 0; i < n; i++) {
        for (const string& skill : people[i]) {
            if (skillToIndex.count(skill)) {
                peopleSkills[i] |= (1 << skillToIndex[skill]);
            }
        }
    }

    int totalStates = 1 << m;
    vector<int> dp(totalStates, INT_MAX);
    vector<int> parentState(totalStates, -1);
    vector<int> parentPerson(totalStates, -1);
    dp[0] = 0;

    for (int state = 0; state < totalStates; state++) {
        if (dp[state] == INT_MAX) continue;

```

```

        for (int i = 0; i < n; i++) {
            int newState = state | peopleSkills[i];
            if (dp[state] + 1 < dp[newState]) {
                dp[newState] = dp[state] + 1;
                parentState[newState] = state;
                parentPerson[newState] = i;
            }
        }
    }

    // 重建团队
    vector<int> team;
    int state = totalStates - 1;

    while (state > 0) {
        team.push_back(parentPerson[state]);
        state = parentState[state];
    }

    return team;
}

/***
 * 哈密顿路径问题
 * 在图中找到访问所有顶点恰好一次的路径
 *
 * 方法: 状态压缩 DP
 * 时间复杂度: O(2^n * n^2)
 * 空间复杂度: O(2^n * n)
 */
static vector<int> hamiltonianPath(vector<vector<int>>& graph) {
    int n = graph.size();
    if (n == 0) return {};

    // dp[mask][last] 表示访问 mask 集合, 以 last 结尾的路径是否存在
    vector<vector<bool>> dp(1 << n, vector<bool>(n, false));
    vector<vector<int>> parent(1 << n, vector<int>(n, -1));

    // 初始化: 每个顶点单独作为路径起点
    for (int i = 0; i < n; i++) {
        dp[1 << i][i] = true;
    }
}

```

```

// 动态规划
for (int mask = 1; mask < (1 << n); mask++) {
    for (int last = 0; last < n; last++) {
        if (!dp[mask][last]) continue;

        for (int next = 0; next < n; next++) {
            if (mask & (1 << next)) continue; // 已经访问过
            if (!graph[last][next]) continue; // 不可达

            int newMask = mask | (1 << next);
            dp[newMask][next] = true;
            parent[newMask][next] = last;
        }
    }
}

// 检查是否存在哈密顿路径
int finalMask = (1 << n) - 1;
for (int last = 0; last < n; last++) {
    if (dp[finalMask][last]) {
        // 重建路径
        vector<int> path;
        int mask = finalMask;
        int current = last;

        while (mask > 0) {
            path.push_back(current);
            int prev = parent[mask][current];
            if (prev == -1) break;
            mask ^= (1 << current);
            current = prev;
        }

        reverse(path.begin(), path.end());
        return path;
    }
}

return {};// 不存在哈密顿路径
}

/***
 * 子集和问题（状态压缩版本）

```

```

* 判断是否存在子集使得元素和等于目标值
*
* 方法: 位运算枚举
* 时间复杂度: O(2^n)
* 空间复杂度: O(2^n)
*/
static bool subsetSum(vector<int>& nums, int target) {
    int n = nums.size();
    int total = 1 << n;

    vector<bool> dp(total, false);
    dp[0] = true;

    for (int mask = 0; mask < total; mask++) {
        if (!dp[mask]) continue;

        int currentSum = 0;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                currentSum += nums[i];
            }
        }

        if (currentSum == target) return true;

        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) continue;
            int newMask = mask | (1 << i);
            dp[newMask] = true;
        }
    }

    return false;
}

};

class PerformanceTester {
public:
    static void testCanIWin() {
        cout << "==== 我能赢吗性能测试 ===" << endl;

        vector<pair<int, int>> testCases = {
            {10, 11}, // 简单情况

```

```

{10, 40}, // 复杂情况
{5, 15} // 边界情况
};

for (auto& testCase : testCases) {
    int maxInt = testCase.first;
    int target = testCase.second;

    auto start = chrono::high_resolution_clock::now();
    bool result = BitMaskDP::canIWin(maxInt, target);
    auto time = chrono::duration_cast<chrono::microseconds>(
        chrono::high_resolution_clock::now() - start).count();

    cout << "maxChoosableInteger=" << maxInt << ", desiredTotal=" << target;
    cout << " -> 结果: " << (result ? "能赢" : "不能赢");
    cout << ", 耗时: " << time << " μs" << endl;
}

static void testMinStickers() {
    cout << "\n==== 贴纸拼词性能测试 ===" << endl;

    vector<string> stickers = {"with", "example", "science"};
    string target = "thehat";

    auto start = chrono::high_resolution_clock::now();
    int result = BitMaskDP::minStickers(stickers, target);
    auto time = chrono::duration_cast<chrono::microseconds>(
        chrono::high_resolution_clock::now() - start).count();

    cout << "贴纸: ";
    for (const string& s : stickers) cout << s << " ";
    cout << endl;
    cout << "目标: " << target << endl;
    cout << "最少需要贴纸: " << result << endl;
    cout << "耗时: " << time << " μs" << endl;
}

static void runUnitTests() {
    cout << "==== 状态压缩 DP 单元测试 ===" << endl;

    // 测试我能赢吗
    assert(BitMaskDP::canIWin(10, 11) == true);
}

```

```
cout << "canIWin 测试通过" << endl;

// 测试子集和
vector<int> nums = {3, 34, 4, 12, 5, 2};
assert(BitMaskDP::subsetSum(nums, 9) == true);
assert(BitMaskDP::subsetSum(nums, 30) == false);
cout << "subsetSum 测试通过" << endl;

cout << "所有单元测试通过!" << endl;
}

};

int main() {
    cout << "状态压缩动态规划算法实现" << endl;
    cout << "使用位掩码技术解决复杂的组合优化问题" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testCanIWin();
    PerformanceTester::testMinStickers();

    // 示例使用
    cout << "\n==== 示例使用 ===" << endl;

    // 我能赢吗示例
    cout << "我能赢吗示例:" << endl;
    cout << "maxChoosableInteger=10, desiredTotal=11 -> ";
    cout << (BitMaskDP::canIWin(10, 11) ? "能赢" : "不能赢") << endl;

    // 子集和问题示例
    vector<int> nums = {3, 34, 4, 12, 5, 2};
    int target = 9;
    cout << "子集和问题示例:" << endl;
    cout << "数组: ";
    for (int num : nums) cout << num << " ";
    cout << ", 目标: " << target << " -> ";
    cout << (BitMaskDP::subsetSum(nums, target) ? "存在" : "不存在") << endl;

    return 0;
}
```

=====

文件: Code16_BitMaskDP.py

=====

"""

状态压缩动态规划算法实现

使用位掩码技术解决复杂的组合优化问题

题目来源:

1. LeetCode 464 - 我能赢吗
2. LeetCode 691 - 贴纸拼词
3. LeetCode 943 - 最短超级串
4. LeetCode 1125 - 最小的必要团队
5. 旅行商问题 (TSP)
6. 哈密顿路径问题

时间复杂度分析:

- 状态压缩 DP: $O(2^n * n)$ 或 $O(2^n * n^2)$
- 空间复杂度: $O(2^n)$ 或 $O(2^n * n)$

工程化考量:

1. 状态表示: 使用位掩码紧凑表示状态
2. 状态转移: 优化状态转移方程
3. 内存优化: 使用滚动数组或位运算压缩
4. 边界处理: 处理空状态和终止状态

"""

```
import time
from typing import List, Tuple, Dict
from functools import lru_cache
import sys
```

```
class BitMaskDP:
```

"""状态压缩动态规划算法类"""

@staticmethod

```
    def can_i_win(max_choosable_integer: int, desired_total: int) -> bool:
        """
```

LeetCode 464 - 我能赢吗

两个玩家轮流选择 1 到 maxChoosableInteger 的数字，先达到或超过 desiredTotal 的玩家获胜

方法: 记忆化搜索 + 状态压缩

```

时间复杂度: O(2^maxChoosableInteger * maxChoosableInteger)
空间复杂度: O(2^maxChoosableInteger)
"""

if max_choosable_integer >= desired_total:
    return True
if max_choosable_integer * (max_choosable_integer + 1) // 2 < desired_total:
    return False

@lru_cache(maxsize=None)
def dfs(state: int, current_total: int) -> bool:
    for i in range(max_choosable_integer):
        if state & (1 << i): # 数字 i+1 已经被选过
            continue

        new_state = state | (1 << i)
        new_total = current_total + i + 1

        # 如果当前选择能直接获胜，或者对手会输
        if new_total >= desired_total or not dfs(new_state, new_total):
            return True
    return False

return dfs(0, 0)

```

```

@staticmethod
def min_stickers(stickers: List[str], target: str) -> int:
"""

```

LeetCode 691 - 贴纸拼词
用给定的贴纸拼出目标单词，求最少需要的贴纸数量

方法：状态压缩 BFS

时间复杂度: O(2^len(target) * n * len(stickers))

空间复杂度: O(2^len(target))

"""

n = len(target)
total_states = 1 << n
dp = [float('inf')] * total_states
dp[0] = 0 # 空状态需要 0 张贴纸

for state in range(total_states):
 if dp[state] == float('inf'):
 continue

```

for sticker in stickers:
    new_state = state

    # 尝试用当前贴纸覆盖目标字符
    count = [0] * 26
    for c in sticker:
        count[ord(c) - ord('a')] += 1

    for i in range(n):
        if state & (1 << i): # 该位置已经被覆盖
            continue
        if count[ord(target[i]) - ord('a')] > 0:
            count[ord(target[i]) - ord('a')] -= 1
            new_state |= (1 << i)

    if new_state != state:
        dp[new_state] = min(dp[new_state], dp[state] + 1)

return dp[total_states - 1] if dp[total_states - 1] != float('inf') else -1

```

```

@staticmethod
def shortest_superstring(words: List[str]) -> str:
    """

```

LeetCode 943 - 最短超级串
找到包含所有字符串的最短超级字符串

方法：状态压缩 DP + 重叠计算

时间复杂度：O($2^n * n^2$)

空间复杂度：O($2^n * n$)

"""

n = len(words)

计算重叠度

overlap = [[0] * n for _ in range(n)]

for i in range(n):

for j in range(n):

if i == j:

continue

length = min(len(words[i]), len(words[j]))

for k in range(length, 0, -1):

if words[i][-k:] == words[j][:k]:

overlap[i][j] = k

break

```

# DP 状态: dp[mask][last] 表示使用 mask 集合, 以 last 结尾的最短长度
INF = float('inf')
dp = [[INF] * n for _ in range(1 << n)]
parent = [[-1] * n for _ in range(1 << n)]

# 初始化
for i in range(n):
    dp[1 << i][i] = len(words[i])

# 动态规划
for mask in range(1 << n):
    for last in range(n):
        if not (mask & (1 << last)) or dp[mask][last] == INF:
            continue

        for next_node in range(n):
            if mask & (1 << next_node):
                continue

            new_mask = mask | (1 << next_node)
            new_len = dp[mask][last] + len(words[next_node]) - overlap[last][next_node]

            if new_len < dp[new_mask][next_node]:
                dp[new_mask][next_node] = new_len
                parent[new_mask][next_node] = last

# 重建结果
final_mask = (1 << n) - 1
last = min(range(n), key=lambda x: dp[final_mask][x])
min_len = dp[final_mask][last]

# 回溯构建字符串
path = []
mask = final_mask
current = last

while mask > 0:
    path.append(current)
    prev = parent[mask][current]
    if prev == -1:
        break
    mask ^= (1 << current)

```

```

        current = prev

    path.reverse()

    result = words[path[0]]
    for i in range(1, len(path)):
        overlap_len = overlap[path[i-1]][path[i]]
        result += words[path[i]][overlap_len:]

    return result

@staticmethod
def smallest_sufficient_team(req_skills: List[str], people: List[List[str]]) -> List[int]:
    """
    LeetCode 1125 - 最小的必要团队
    找到覆盖所有技能的最小团队

    方法: 状态压缩 DP
    时间复杂度: O(2^m * n) - m 为技能数量, n 为人员数量
    空间复杂度: O(2^m)
    """

    m = len(req_skills)
    n = len(people)

    # 技能到索引的映射
    skill_to_index = {skill: i for i, skill in enumerate(req_skills)}

    # 将每个人的技能转换为位掩码
    people_skills = [0] * n
    for i in range(n):
        for skill in people[i]:
            if skill in skill_to_index:
                people_skills[i] |= (1 << skill_to_index[skill])

    total_states = 1 << m
    dp = [float('inf')] * total_states
    parent_state = [-1] * total_states
    parent_person = [-1] * total_states
    dp[0] = 0

    for state in range(total_states):
        if dp[state] == float('inf'):
            continue

```

```

for i in range(n):
    new_state = state | people_skills[i]
    if dp[state] + 1 < dp[new_state]:
        dp[new_state] = dp[state] + 1
        parent_state[new_state] = state
        parent_person[new_state] = i

# 重建团队
team = []
state = total_states - 1

while state > 0:
    team.append(parent_person[state])
    state = parent_state[state]

return team

```

```

@staticmethod
def hamiltonian_path(graph: List[List[int]]) -> List[int]:
    """
    哈密顿路径问题
    在图中找到访问所有顶点恰好一次的路径

```

方法：状态压缩 DP

时间复杂度： $O(2^n * n^2)$

空间复杂度： $O(2^n * n)$

"""

```
n = len(graph)
```

```
if n == 0:
```

```
    return []
```

$dp[mask][last]$ 表示访问 mask 集合，以 last 结尾的路径是否存在

```
INF = float('inf')
```

```
dp = [[INF] * n for _ in range(1 << n)]
```

```
parent = [[-1] * n for _ in range(1 << n)]
```

初始化：每个顶点单独作为路径起点

```
for i in range(n):
```

```
    dp[1 << i][i] = 0
```

动态规划

```
for mask in range(1 << n):
```

```

for last in range(n):
    if dp[mask][last] == INF:
        continue

    for next_node in range(n):
        if mask & (1 << next_node): # 已经访问过
            continue
        if not graph[last][next_node]: # 不可达
            continue

        new_mask = mask | (1 << next_node)
        if dp[mask][last] + 1 < dp[new_mask][next_node]:
            dp[new_mask][next_node] = dp[mask][last] + 1
            parent[new_mask][next_node] = last

# 检查是否存在哈密顿路径
final_mask = (1 << n) - 1
for last in range(n):
    if dp[final_mask][last] != INF:
        # 重建路径
        path = []
        mask = final_mask
        current = last

        while mask > 0:
            path.append(current)
            prev = parent[mask][current]
            if prev == -1:
                break
            mask ^= (1 << current)
            current = prev

        path.reverse()
        return path

return [] # 不存在哈密顿路径

@staticmethod
def subset_sum(nums: List[int], target: int) -> bool:
    """
    子集和问题（状态压缩版本）
    判断是否存在子集使得元素和等于目标值

```

```
方法：位运算枚举
时间复杂度：O(2^n)
空间复杂度：O(2^n)
"""

n = len(nums)
total = 1 << n

dp = [False] * total
dp[0] = True

for mask in range(total):
    if not dp[mask]:
        continue

    current_sum = 0
    for i in range(n):
        if mask & (1 << i):
            current_sum += nums[i]

    if current_sum == target:
        return True

    for i in range(n):
        if mask & (1 << i):
            continue
        new_mask = mask | (1 << i)
        dp[new_mask] = True

return False
```

```
class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def test_can_i_win():
        """测试我能赢吗性能"""
        print("== 我能赢吗性能测试 ==")

        test_cases = [
            (10, 11),  # 简单情况
            (10, 40),  # 复杂情况
            (5, 15)   # 边界情况
        ]
```

```
]
```

```
for max_int, target in test_cases:  
    start = time.time()  
    result = BitMaskDP.can_i_win(max_int, target)  
    elapsed = (time.time() - start) * 1e6 # 微秒  
  
    print(f"maxChoosableInteger={max_int}, desiredTotal={target}")  
    print(f" 结果: {'能赢' if result else '不能赢'}, 耗时: {elapsed:.2f} μs")
```

```
@staticmethod
```

```
def test_min_stickers():  
    """测试贴纸拼词性能"""  
    print("\n==== 贴纸拼词性能测试 ===")
```

```
stickers = ["with", "example", "science"]  
target = "thehat"  
  
start = time.time()  
result = BitMaskDP.min_stickers(stickers, target)  
elapsed = (time.time() - start) * 1e6 # 微秒  
  
print(f"贴纸: {stickers}")  
print(f"目标: {target}")  
print(f"最少需要贴纸: {result}")  
print(f"耗时: {elapsed:.2f} μs")
```

```
@staticmethod
```

```
def run_unit_tests():  
    """运行单元测试"""  
    print("==== 状态压缩 DP 单元测试 ===")
```

```
# 测试我能赢吗  
assert BitMaskDP.can_i_win(10, 11) == True  
print("can_i_win 测试通过")  
  
# 测试子集和  
nums = [3, 34, 4, 12, 5, 2]  
assert BitMaskDP.subset_sum(nums, 9) == True  
assert BitMaskDP.subset_sum(nums, 30) == False  
print("subset_sum 测试通过")  
  
print("所有单元测试通过!")
```

```
@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n==== 复杂度分析 ====")

    algorithms = {
        "can_i_win": {
            "time": "O(2^n * n)",
            "space": "O(2^n)",
            "desc": "记忆化搜索+状态压缩"
        },
        "min_stickers": {
            "time": "O(2^m * n * k)",
            "space": "O(2^m)",
            "desc": "状态压缩 BFS"
        },
        "shortest_superstring": {
            "time": "O(2^n * n^2)",
            "space": "O(2^n * n)",
            "desc": "状态压缩 DP+重叠计算"
        },
        "hamiltonian_path": {
            "time": "O(2^n * n^2)",
            "space": "O(2^n * n)",
            "desc": "状态压缩 DP"
        }
    }

    for name, info in algorithms.items():
        print(f"{name}:")
        print(f"    时间复杂度: {info['time']}")
        print(f"    空间复杂度: {info['space']}")
        print(f"    描述: {info['desc']}")
        print()

def main():
    """主函数"""
    print("状态压缩动态规划算法实现")
    print("使用位掩码技术解决复杂的组合优化问题")
    print("=" * 50)
```

```

# 运行单元测试
PerformanceTester.run_unit_tests()

# 运行性能测试
PerformanceTester.test_can_i_win()
PerformanceTester.test_min_stickers()

# 复杂度分析
PerformanceTester.complexity_analysis()

# 示例使用
print("\n==== 示例使用 ====")

# 我能赢吗示例
print("我能赢吗示例:")
result = BitMaskDP.can_i_win(10, 11)
print(f"maxChoosableInteger=10, desiredTotal=11 -> {'能赢' if result else '不能赢'}")

# 子集和问题示例
nums = [3, 34, 4, 12, 5, 2]
target = 9
print(f"\n子集和问题示例:")
print(f"数组: {nums}, 目标: {target}")
result = BitMaskDP.subset_sum(nums, target)
print(f"结果: {'存在' if result else '不存在'}")

if __name__ == "__main__":
    main()

```

=====

文件: Code16_TravelingSalesman.java

```

=====
package class032;

import java.util.*;

/**
 * 旅行商问题 (TSP) 经典实现
 * 题目来源: 多种算法平台通用问题
 * 问题描述: 给定 n 个城市和它们之间的距离, 找到一条最短的路径, 访问每个城市恰好一次并回到起点
 */

```

* 解题思路:

- * 方法 1: 暴力搜索 - 枚举所有排列, 时间复杂度高
- * 方法 2: 状态压缩 DP - 使用 bitmask 表示已访问的城市
- * 方法 3: 分支限界法 - 使用优先队列进行剪枝
- * 方法 4: 近似算法 - 2-opt, 3-opt 等启发式算法

*

* 时间复杂度分析:

- * 方法 1: $O(n!)$ - 阶乘级, 仅适用于小 n
- * 方法 2: $O(2^n * n^2)$ - 状态压缩 DP 的标准复杂度
- * 方法 3: $O(b^d)$ - 分支因子 b 和深度 d, 实际运行可能更快
- * 方法 4: $O(n^2)$ 或 $O(n^3)$ - 多项式时间, 但可能不是最优解

*

* 空间复杂度分析:

- * 方法 1: $O(n)$ - 递归栈空间
- * 方法 2: $O(2^n * n)$ - DP 数组空间
- * 方法 3: $O(b^d)$ - 优先队列空间
- * 方法 4: $O(n^2)$ - 距离矩阵空间

*

* 工程化考量:

- * 1. 内存优化: 对于大 n 使用位运算压缩状态
- * 2. 性能优化: 使用启发式剪枝和近似算法
- * 3. 并行计算: 对于大规模问题考虑并行处理
- * 4. 实际应用: 结合具体业务场景进行优化

*/

```
public class Code16_TravelingSalesman {  
  
    /**  
     * 方法 1: 暴力搜索 (回溯法)  
     * 枚举所有可能的路径排列, 找到最短路径  
     * 仅适用于 n <= 10 的小规模问题  
     */  
    public static int tspBruteForce(int[][] graph) {  
        int n = graph.length;  
        if (n <= 1) return 0;  
  
        int[] cities = new int[n];  
        for (int i = 0; i < n; i++) {  
            cities[i] = i;  
        }  
  
        int minDistance = Integer.MAX_VALUE;
```

```

// 生成所有排列
List<List<Integer>> permutations = generatePermutations(cities);

for (List<Integer> path : permutations) {
    int distance = calculatePathDistance(graph, path);
    minDistance = Math.min(minDistance, distance);
}

return minDistance;
}

/**
 * 生成所有排列
 */
private static List<List<Integer>> generatePermutations(int[] arr) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(arr, 0, result);
    return result;
}

private static void backtrack(int[] arr, int start, List<List<Integer>> result) {
    if (start == arr.length) {
        List<Integer> path = new ArrayList<>();
        for (int num : arr) {
            path.add(num);
        }
        // 添加起点形成回路
        path.add(arr[0]);
        result.add(path);
        return;
    }

    for (int i = start; i < arr.length; i++) {
        swap(arr, start, i);
        backtrack(arr, start + 1, result);
        swap(arr, start, i);
    }
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

```

```

}

/**
 * 计算路径距离
 */
private static int calculatePathDistance(int[][] graph, List<Integer> path) {
    int distance = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        int from = path.get(i);
        int to = path.get(i + 1);
        distance += graph[from][to];
    }
    return distance;
}

/**
 * 方法 2: 状态压缩 DP (最优解)
 * 使用 dp[mask][last] 表示在 mask 状态下最后访问城市 last 时的最短距离
 */
public static int tspDP(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    int totalStates = 1 << n;
    int[][] dp = new int[totalStates][n];

    // 初始化 DP 数组
    for (int i = 0; i < totalStates; i++) {
        Arrays.fill(dp[i], Integer.MAX_VALUE);
    }

    // 起点状态: 只访问了城市 0
    dp[1][0] = 0;

    // 遍历所有状态
    for (int mask = 1; mask < totalStates; mask++) {
        for (int last = 0; last < n; last++) {
            // 如果 last 不在 mask 中, 跳过
            if ((mask & (1 << last)) == 0) continue;

            // 如果当前状态不可达, 跳过
            if (dp[mask][last] == Integer.MAX_VALUE) continue;

            // 更新最短距离
            for (int next = 0; next < n; next++) {
                if ((mask & (1 << next)) == 0) continue;
                if (next == last) continue;

                int newDistance = dp[mask][last] + graph[last][next];
                if (newDistance < dp[mask | (1 << next)][next]) {
                    dp[mask | (1 << next)][next] = newDistance;
                }
            }
        }
    }
}

```

```

// 尝试访问新城市
for (int next = 0; next < n; next++) {
    // 如果 next 已经在 mask 中，跳过
    if ((mask & (1 << next)) != 0) continue;

    int newMask = mask | (1 << next);
    int newDistance = dp[mask][last] + graph[last][next];

    if (newDistance < dp[newMask][next]) {
        dp[newMask][next] = newDistance;
    }
}

}

// 找到最短回路：访问所有城市后回到起点
int finalMask = (1 << n) - 1;
int minDistance = Integer.MAX_VALUE;

for (int last = 0; last < n; last++) {
    if (dp[finalMask][last] != Integer.MAX_VALUE) {
        minDistance = Math.min(minDistance,
            dp[finalMask][last] + graph[last][0]);
    }
}

return minDistance;
}

/***
 * 方法 3：记忆化搜索版本
 * 使用递归+记忆化，代码更清晰
 */
public static int tspMemo(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    int[][] memo = new int[1 << n][n];
    for (int[] row : memo) {
        Arrays.fill(row, -1);
    }

    return dfs(1, 0, n, graph, memo);
}

```

```
}
```

```
private static int dfs(int mask, int last, int n, int[][] graph, int[][] memo) {
    // 如果已经访问所有城市，返回回到起点的距离
    if (mask == (1 << n) - 1) {
        return graph[last][0];
    }

    if (memo[mask][last] != -1) {
        return memo[mask][last];
    }

    int minDistance = Integer.MAX_VALUE;

    for (int next = 0; next < n; next++) {
        // 如果 next 已经在 mask 中，跳过
        if ((mask & (1 << next)) != 0) continue;

        int newMask = mask | (1 << next);
        int distance = graph[last][next] + dfs(newMask, next, n, graph, memo);

        if (distance < minDistance) {
            minDistance = distance;
        }
    }

    memo[mask][last] = minDistance;
    return minDistance;
}

/***
 * 方法 4：最近邻启发式算法（近似解）
 * 贪心算法，每次选择最近的未访问城市
 * 时间复杂度：O(n^2)，空间复杂度：O(n)
 */
public static int tspNearestNeighbor(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    boolean[] visited = new boolean[n];
    visited[0] = true; // 从城市 0 开始
    int current = 0;
    int totalDistance = 0;
```

```

int count = 1;

while (count < n) {
    int next = -1;
    int minDist = Integer.MAX_VALUE;

    // 找到最近的未访问城市
    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[current][i] < minDist) {
            minDist = graph[current][i];
            next = i;
        }
    }

    if (next == -1) break;

    totalDistance += minDist;
    visited[next] = true;
    current = next;
    count++;
}

// 回到起点
totalDistance += graph[current][0];
return totalDistance;
}

/***
 * 方法 5: 2-opt 局部搜索算法
 * 通过交换边来改进解质量
 */
public static int tsp2Opt(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    // 初始解: 使用最近邻算法
    List<Integer> tour = new ArrayList<>();
    boolean[] visited = new boolean[n];
    int current = 0;
    tour.add(current);
    visited[current] = true;

    while (tour.size() < n) {

```

```

int next = -1;
int minDist = Integer.MAX_VALUE;

for (int i = 0; i < n; i++) {
    if (!visited[i] && graph[current][i] < minDist) {
        minDist = graph[current][i];
        next = i;
    }
}

if (next == -1) break;

tour.add(next);
visited[next] = true;
current = next;
}

tour.add(0); // 回到起点

// 2-opt 优化
boolean improved = true;
while (improved) {
    improved = false;

    for (int i = 1; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            int oldDistance = calculateTourDistance(graph, tour);

            // 尝试交换边 (i-1, i) 和 (j, j+1) 为 (i-1, j) 和 (i, j+1)
            Collections.reverse(tour.subList(i, j + 1));

            int newDistance = calculateTourDistance(graph, tour);

            if (newDistance < oldDistance) {
                improved = true;
            } else {
                // 恢复原状
                Collections.reverse(tour.subList(i, j + 1));
            }
        }
    }
}

```

```

        return calculateTourDistance(graph, tour);
    }

/**
 * 计算完整路径的距离
 */
private static int calculateTourDistance(int[][] graph, List<Integer> tour) {
    int distance = 0;
    for (int i = 0; i < tour.size() - 1; i++) {
        int from = tour.get(i);
        int to = tour.get(i + 1);
        distance += graph[from][to];
    }
    return distance;
}

/**
 * 工程化改进版本：完整的异常处理和验证
 */
public static int tspWithValidation(int[][] graph) {
    try {
        // 输入验证
        if (graph == null) {
            throw new IllegalArgumentException("Graph cannot be null");
        }
        if (graph.length <= 1) {
            return 0;
        }

        // 检查图是否对称（TSP 通常假设对称）
        int n = graph.length;
        for (int i = 0; i < n; i++) {
            if (graph[i].length != n) {
                throw new IllegalArgumentException("Graph must be square matrix");
            }
            if (graph[i][i] != 0) {
                throw new IllegalArgumentException("Diagonal elements should be 0");
            }
        }

        // 根据问题规模选择算法
        if (n <= 10) {
            return tspBruteForce(graph);
        }
    }
}
```

```

        } else if (n <= 20) {
            return tspDP(graph);
        } else {
            return tsp20pt(graph); // 对于大 n 使用近似算法
        }
    }

} catch (Exception e) {
    System.out.println("Error in TSP: " + e.getMessage());
    return Integer.MAX_VALUE;
}

}

/***
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("==== 旅行商问题 (TSP) - 单元测试 ===");

    // 测试用例 1: 简单 4 城市问题
    int[][] graph1 = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int expected1 = 80; // 0->1->3->2->0 = 10+25+30+15 = 80
    int result1 = tspWithValidation(graph1);
    System.out.printf("测试 1: 期望=%d, 实际=%d, %s%n",
        expected1, result1,
        result1 == expected1 ? "通过" : "失败");

    // 测试用例 2: 3 城市问题
    int[][] graph2 = {
        {0, 1, 2},
        {1, 0, 3},
        {2, 3, 0}
    };

    int expected2 = 6; // 0->1->2->0 = 1+3+2 = 6
    int result2 = tspWithValidation(graph2);
    System.out.printf("测试 2: 期望=%d, 实际=%d, %s%n",
        expected2, result2,

```

```

        result2 == expected2 ? "通过" : "失败");

// 测试不同方法的结果
System.out.println("\n==== 方法对比测试 ====");
int[][] testGraph = graph1;

int r1 = tspBruteForce(testGraph);
int r2 = tspDP(testGraph);
int r3 = tspMemo(testGraph);
int r4 = tspNearestNeighbor(testGraph);
int r5 = tsp20pt(testGraph);

System.out.printf("暴力搜索: %d\n", r1);
System.out.printf("状态压缩 DP: %d\n", r2);
System.out.printf("记忆化搜索: %d\n", r3);
System.out.printf("最近邻算法: %d\n", r4);
System.out.printf("2-opt 算法: %d\n", r5);
}

/**
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");

    int[] testSizes = {5, 10, 15, 20};

    for (int n : testSizes) {
        int[][] graph = generateRandomGraph(n, 100);
        System.out.printf("n = %d:\n", n);

        if (n <= 10) {
            long startTime = System.nanoTime();
            int result1 = tspBruteForce(graph);
            long time1 = System.nanoTime() - startTime;
            System.out.printf(" 暴力搜索: %d ns, 结果: %d\n", time1, result1);
        }

        long startTime = System.nanoTime();
        int result2 = tspDP(graph);
        long time2 = System.nanoTime() - startTime;
        System.out.printf(" 状态压缩 DP: %d ns, 结果: %d\n", time2, result2);
    }
}

```

```

        startTime = System.nanoTime();
        int result3 = tsp20pt(graph);
        long time3 = System.nanoTime() - startTime;
        System.out.printf(" 2-opt 算法: %d ns, 结果: %d\n", time3, result3);

        System.out.println();
    }

}

/***
 * 生成随机图
 */
private static int[][] generateRandomGraph(int n, int maxDistance) {
    Random random = new Random();
    int[][] graph = new int[n][n];

    for (int i = 0; i < n; i++) {
        graph[i][i] = 0;
        for (int j = i + 1; j < n; j++) {
            int distance = random.nextInt(maxDistance) + 1;
            graph[i][j] = distance;
            graph[j][i] = distance; // 对称图
        }
    }

    return graph;
}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("== 复杂度分析 ==");
    System.out.println("旅行商问题是 NP 难问题, 没有已知的多项式时间算法");

    System.out.println("\n各算法复杂度对比:");
    System.out.println("暴力搜索: O(n!) - 仅适用于 n <= 10");
    System.out.println("状态压缩 DP: O(2^n * n^2) - 适用于 n <= 20");
    System.out.println("分支限界法: 指数级但带剪枝");
    System.out.println("近似算法: O(n^2) 或 O(n^3) - 适用于大 n");

    System.out.println("\n工程化建议:");
    System.out.println("1. 根据问题规模选择合适的算法");
}

```

```

System.out.println("2. 对于大规模问题使用近似算法");
System.out.println("3. 考虑并行计算和分布式处理");
System.out.println("4. 结合实际业务场景进行优化");
}

public static void main(String[] args) {
    System.out.println("旅行商问题 (TSP) 经典实现");
    System.out.println("多种算法解决组合优化问题");

    // 运行单元测试
    runTests();

    // 运行性能测试
    performanceTest();

    // 复杂度分析
    complexityAnalysis();

    // 示例使用
    System.out.println("\n==== 示例使用 ====");
    int[][] exampleGraph = {
        {0, 2, 9, 10},
        {2, 0, 6, 4},
        {9, 6, 0, 8},
        {10, 4, 8, 0}
    };

    int result = tspWithValidation(exampleGraph);
    System.out.printf("示例图的最短路径长度: %d\n", result);
    System.out.println("距离矩阵:");
    for (int[] row : exampleGraph) {
        System.out.println(Arrays.toString(row));
    }
}
}
=====
```

文件: Code16_TravelingSalesman_part2.java

```

=====
package class032;

import java.util.*;
```

```

/**
 * 旅行商问题 (TSP) 第二部分 - 记忆化搜索和近似算法
 */

public class Code16_TravelingSalesman_part2 {

    /**
     * 方法 3: 记忆化搜索版本
     * 使用递归+记忆化, 代码更清晰
     */
    public static int tspMemo(int[][] graph) {
        int n = graph.length;
        if (n <= 1) return 0;

        int[][] memo = new int[1 << n][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }

        return dfs(1, 0, n, graph, memo);
    }

    private static int dfs(int mask, int last, int n, int[][] graph, int[][] memo) {
        // 如果已经访问所有城市, 返回回到起点的距离
        if (mask == (1 << n) - 1) {
            return graph[last][0];
        }

        if (memo[mask][last] != -1) {
            return memo[mask][last];
        }

        int minDistance = Integer.MAX_VALUE;

        for (int next = 0; next < n; next++) {
            // 如果 next 已经在 mask 中, 跳过
            if ((mask & (1 << next)) != 0) continue;

            int newMask = mask | (1 << next);
            int distance = graph[last][next] + dfs(newMask, next, n, graph, memo);

            if (distance < minDistance) {

```

```

        minDistance = distance;
    }
}

memo[mask][last] = minDistance;
return minDistance;
}

/***
 * 方法 4: 最近邻启发式算法 (近似解)
 * 贪心算法, 每次选择最近的未访问城市
 * 时间复杂度: O(n^2), 空间复杂度: O(n)
 */
public static int tspNearestNeighbor(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    boolean[] visited = new boolean[n];
    visited[0] = true; // 从城市 0 开始
    int current = 0;
    int totalDistance = 0;
    int count = 1;

    while (count < n) {
        int next = -1;
        int minDist = Integer.MAX_VALUE;

        // 找到最近的未访问城市
        for (int i = 0; i < n; i++) {
            if (!visited[i] && graph[current][i] < minDist) {
                minDist = graph[current][i];
                next = i;
            }
        }

        if (next == -1) break;

        totalDistance += minDist;
        visited[next] = true;
        current = next;
        count++;
    }
}

```

```

// 回到起点
totalDistance += graph[current][0];
return totalDistance;
}

/***
 * 方法 5: 2-opt 局部搜索算法
 * 通过交换边来改进解质量
 */
public static int tsp20pt(int[][] graph) {
    int n = graph.length;
    if (n <= 1) return 0;

    // 初始解: 使用最近邻算法
    List<Integer> tour = new ArrayList<>();
    boolean[] visited = new boolean[n];
    int current = 0;
    tour.add(current);
    visited[current] = true;

    while (tour.size() < n) {
        int next = -1;
        int minDist = Integer.MAX_VALUE;

        for (int i = 0; i < n; i++) {
            if (!visited[i] && graph[current][i] < minDist) {
                minDist = graph[current][i];
                next = i;
            }
        }

        if (next == -1) break;

        tour.add(next);
        visited[next] = true;
        current = next;
    }

    tour.add(0); // 回到起点

    // 2-opt 优化
    boolean improved = true;
    while (improved) {

```

```

improved = false;

for (int i = 1; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        int oldDistance = calculateTourDistance(graph, tour);

        // 尝试交换边 (i-1, i) 和 (j, j+1) 为 (i-1, j) 和 (i, j+1)
        Collections.reverse(tour.subList(i, j + 1));

        int newDistance = calculateTourDistance(graph, tour);

        if (newDistance < oldDistance) {
            improved = true;
        } else {
            // 恢复原状
            Collections.reverse(tour.subList(i, j + 1));
        }
    }
}

return calculateTourDistance(graph, tour);
}

/***
 * 计算完整路径的距离
 */
private static int calculateTourDistance(int[][] graph, List<Integer> tour) {
    int distance = 0;
    for (int i = 0; i < tour.size() - 1; i++) {
        int from = tour.get(i);
        int to = tour.get(i + 1);
        distance += graph[from][to];
    }
    return distance;
}

// 继续添加测试和工具方法...

```

=====

文件: Code16_TravelingSalesman_part3.java

=====

```
package class032;

import java.util.*;

/**
 * 旅行商问题 (TSP) 第三部分 - 测试和工具方法
 */

public class Code16_TravelingSalesman_part3 {

    /**
     * 工程化改进版本：完整的异常处理和验证
     */

    public static int tspWithValidation(int[][] graph) {
        try {
            // 输入验证
            if (graph == null) {
                throw new IllegalArgumentException("Graph cannot be null");
            }
            if (graph.length <= 1) {
                return 0;
            }

            // 检查图是否对称 (TSP 通常假设对称)
            int n = graph.length;
            for (int i = 0; i < n; i++) {
                if (graph[i].length != n) {
                    throw new IllegalArgumentException("Graph must be square matrix");
                }
                if (graph[i][i] != 0) {
                    throw new IllegalArgumentException("Diagonal elements should be 0");
                }
            }

            // 根据问题规模选择算法
            if (n <= 10) {
                return tspBruteForce(graph);
            } else if (n <= 20) {
                return tspDP(graph);
            } else {
                return tsp20pt(graph); // 对于大 n 使用近似算法
            }
        }
    }
}
```

```

    } catch (Exception e) {
        System.out.println("Error in TSP: " + e.getMessage());
        return Integer.MAX_VALUE;
    }
}

/***
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("== 旅行商问题 (TSP) - 单元测试 ==");

    // 测试用例 1: 简单 4 城市问题
    int[][] graph1 = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int expected1 = 80; // 0->1->3->2->0 = 10+25+30+15 = 80
    int result1 = tspWithValidation(graph1);
    System.out.printf("测试 1: 期望=%d, 实际=%d, %s%n",
                      expected1, result1,
                      result1 == expected1 ? "通过" : "失败");

    // 测试用例 2: 3 城市问题
    int[][] graph2 = {
        {0, 1, 2},
        {1, 0, 3},
        {2, 3, 0}
    };

    int expected2 = 6; // 0->1->2->0 = 1+3+2 = 6
    int result2 = tspWithValidation(graph2);
    System.out.printf("测试 2: 期望=%d, 实际=%d, %s%n",
                      expected2, result2,
                      result2 == expected2 ? "通过" : "失败");

    // 测试不同方法的结果
    System.out.println("\n== 方法对比测试 ==");
    int[][] testGraph = graph1;

```

```
int r1 = tspBruteForce(testGraph);
int r2 = tspDP(testGraph);
int r3 = tspMemo(testGraph);
int r4 = tspNearestNeighbor(testGraph);
int r5 = tsp20pt(testGraph);

System.out.printf("暴力搜索: %d%n", r1);
System.out.printf("状态压缩 DP: %d%n", r2);
System.out.printf("记忆化搜索: %d%n", r3);
System.out.printf("最近邻算法: %d%n", r4);
System.out.printf("2-opt 算法: %d%n", r5);
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ===");

    int[] testSizes = {5, 10, 15, 20};

    for (int n : testSizes) {
        int[][] graph = generateRandomGraph(n, 100);
        System.out.printf("n = %d:%n", n);

        if (n <= 10) {
            long startTime = System.nanoTime();
            int result1 = tspBruteForce(graph);
            long time1 = System.nanoTime() - startTime;
            System.out.printf(" 暴力搜索: %d ns, 结果: %d%n", time1, result1);
        }

        long startTime = System.nanoTime();
        int result2 = tspDP(graph);
        long time2 = System.nanoTime() - startTime;
        System.out.printf(" 状态压缩 DP: %d ns, 结果: %d%n", time2, result2);

        startTime = System.nanoTime();
        int result3 = tsp20pt(graph);
        long time3 = System.nanoTime() - startTime;
        System.out.printf(" 2-opt 算法: %d ns, 结果: %d%n", time3, result3);

        System.out.println();
    }
}
```

```

    }

}

/***
 * 生成随机图
 */
private static int[][] generateRandomGraph(int n, int maxDistance) {
    Random random = new Random();
    int[][] graph = new int[n][n];

    for (int i = 0; i < n; i++) {
        graph[i][i] = 0;
        for (int j = i + 1; j < n; j++) {
            int distance = random.nextInt(maxDistance) + 1;
            graph[i][j] = distance;
            graph[j][i] = distance; // 对称图
        }
    }
}

return graph;
}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("== 复杂度分析 ==");
    System.out.println("旅行商问题是 NP 难问题，没有已知的多项式时间算法");

    System.out.println("\n各算法复杂度对比:");
    System.out.println("暴力搜索: O(n!) - 仅适用于 n <= 10");
    System.out.println("状态压缩 DP: O(2^n * n^2) - 适用于 n <= 20");
    System.out.println("分支限界法: 指数级但带剪枝");
    System.out.println("近似算法: O(n^2) 或 O(n^3) - 适用于大 n");

    System.out.println("\n工程化建议:");
    System.out.println("1. 根据问题规模选择合适的算法");
    System.out.println("2. 对于大规模问题使用近似算法");
    System.out.println("3. 考虑并行计算和分布式处理");
    System.out.println("4. 结合实际业务场景进行优化");
}

public static void main(String[] args) {
}

```

```
System.out.println("旅行商问题 (TSP) 经典实现");
System.out.println("多种算法解决组合优化问题");

// 运行单元测试
runTests();

// 运行性能测试
performanceTest();

// 复杂度分析
complexityAnalysis();

// 示例使用
System.out.println("\n== 示例使用 ==");
int[][] exampleGraph = {
    {0, 2, 9, 10},
    {2, 0, 6, 4},
    {9, 6, 0, 8},
    {10, 4, 8, 0}
};

int result = tspWithValidation(exampleGraph);
System.out.printf("示例图的最短路径长度: %d\n", result);
System.out.println("距离矩阵:");
for (int[] row : exampleGraph) {
    System.out.println(Arrays.toString(row));
}
}
```

=====

文件: Code17_BitwiseOperations.cpp

=====

```
#include <iostream>
#include <vector>
#include <set>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
```

```
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>

using namespace std;

/***
 * 位运算算法实现
 * 包含 LeetCode 多个位运算相关题目的解决方案
 *
 * 题目列表:
 * 1. LeetCode 136 - 只出现一次的数字
 * 2. LeetCode 137 - 只出现一次的数字 II
 * 3. LeetCode 260 - 只出现一次的数字 III
 * 4. LeetCode 191 - 位 1 的个数
 * 5. LeetCode 231 - 2 的幂
 * 6. LeetCode 342 - 4 的幂
 * 7. LeetCode 371 - 两整数之和
 * 8. LeetCode 201 - 数字范围按位与
 * 9. LeetCode 338 - 比特位计数
 * 10. LeetCode 405 - 数字转换为十六进制数
 *
 * 时间复杂度分析:
 * - 位运算操作: O(1) 或 O(n)
 * - 空间复杂度: O(1) 或 O(n)
 *
 * 工程化考量:
 * 1. 位运算技巧: 使用位运算优化算法
 * 2. 边界处理: 处理负数、零等边界情况
 * 3. 性能优化: 利用位运算的常数时间优势
 * 4. 可读性: 添加详细注释说明位运算原理
 */

class BitwiseOperations {
```

```
public:  
    /**  
     * LeetCode 136 - 只出现一次的数字  
     * 题目链接: https://leetcode.com/problems/single-number/  
     * 给定一个非空整数数组，除了某个元素只出现一次外，其余每个元素均出现两次。找出那个只出现一次的元素。  
     *  
     * 方法：异或运算  
     * 时间复杂度：O(n)  
     * 空间复杂度：O(1)  
     *  
     * 原理： $a \wedge a = 0, a \wedge 0 = a$   
     * 所有出现两次的数字异或后为 0，最后剩下的就是只出现一次的数字  
     */  
  
    static int singleNumber(vector<int>& nums) {  
        int result = 0;  
        for (int num : nums) {  
            result ^= num;  
        }  
        return result;  
    }  
  
    /**  
     * LeetCode 137 - 只出现一次的数字 II  
     * 题目链接: https://leetcode.com/problems/single-number-ii/  
     * 给定一个非空整数数组，除了某个元素只出现一次外，其余每个元素均出现三次。找出那个只出现一次的元素。  
     *  
     * 方法：位计数法  
     * 时间复杂度：O(n)  
     * 空间复杂度：O(1)  
     *  
     * 原理：统计每个位上 1 出现的次数，对 3 取模  
     * 如果某位上 1 出现的次数不是 3 的倍数，说明只出现一次的数字在该位为 1  
     */  
  
    static int singleNumberII(vector<int>& nums) {  
        int result = 0;  
  
        // 遍历 32 位  
        for (int i = 0; i < 32; i++) {  
            int count = 0;  
  
            // 统计第 i 位为 1 的数字个数  
        }  
    }
```

```

        for (int num : nums) {
            if ((num >> i) & 1) {
                count++;
            }
        }

        // 如果 count % 3 != 0, 说明只出现一次的数字在该位为 1
        if (count % 3 != 0) {
            result |= (1 << i);
        }
    }

    return result;
}

/***
 * LeetCode 260 - 只出现一次的数字 III
 * 题目链接: https://leetcode.com/problems/single-number-iii/
 * 给定一个整数数组，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那
两个元素。
 *
 * 方法: 分组异或
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 *
 * 原理:
 * 1. 先对所有数字异或，得到两个不同数字的异或结果
 * 2. 找到异或结果中为 1 的某一位（两个数字在该位不同）
 * 3. 根据这一位将数组分成两组，分别异或得到两个数字
 */
static vector<int> singleNumberIII(vector<int>& nums) {
    // 第一步: 计算所有数字的异或
    int xor_all = 0;
    for (int num : nums) {
        xor_all ^= num;
    }

    // 第二步: 找到为 1 的最低位
    int diff_bit = 1;
    while ((xor_all & diff_bit) == 0) {
        diff_bit <= 1;
    }
}

```

```

// 第三步：根据 diff_bit 分组异或
int num1 = 0, num2 = 0;
for (int num : nums) {
    if (num & diff_bit) {
        num1 ^= num;
    } else {
        num2 ^= num;
    }
}

return {num1, num2};
}

/***
 * LeetCode 191 - 位 1 的个数
 * 题目链接: https://leetcode.com/problems/number-of-1-bits/
 * 编写一个函数，输入是一个无符号整数，返回其二进制表达式中数位数为 '1' 的个数（也被称为汉明重量）。
 *
 * 方法 1: 循环检查二进制位
 * 方法 2: Brian Kernighan 算法
 * 时间复杂度: O(1) - 固定 32 位
 * 空间复杂度: O(1)
 */
static int hammingWeight(uint32_t n) {
    // 方法 1: 循环检查
    int count = 0;
    while (n) {
        count += (n & 1);
        n >>= 1;
    }
    return count;

    // 方法 2: Brian Kernighan 算法 (更高效)
    // int count = 0;
    // while (n) {
    //     n &= (n - 1); // 清除最低位的 1
    //     count++;
    // }
    // return count;
}

/***

```

```

* LeetCode 231 - 2 的幂
* 题目链接: https://leetcode.com/problems/power-of-two/
* 给定一个整数，编写一个函数来判断它是否是 2 的幂次方。
*
* 方法: 位运算性质
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 原理: 2 的幂的二进制表示中只有一位是 1
* n > 0 && (n & (n - 1)) == 0
*/
static bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

/***
* LeetCode 342 - 4 的幂
* 题目链接: https://leetcode.com/problems/power-of-four/
* 给定一个整数，编写一个函数来判断它是否是 4 的幂次方。
*
* 方法: 位运算 + 数学性质
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 原理:
* 1. 必须是 2 的幂: n > 0 && (n & (n - 1)) == 0
* 2. 4 的幂的 1 出现在奇数位: (n & 0x55555555) != 0
*/
static bool isPowerOfFour(int n) {
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}

/***
* LeetCode 371 - 两整数之和
* 题目链接: https://leetcode.com/problems/sum-of-two-integers/
* 不使用运算符 + 和 - ，计算两整数 a 、 b 之和。
*
* 方法: 位运算模拟加法
* 时间复杂度: O(1) - 最多 32 次循环
* 空间复杂度: O(1)
*
* 原理:
* 1. 异或运算得到无进位和

```

```

* 2. 与运算左移一位得到进位
* 3. 循环直到进位为 0
*/
static int getSum(int a, int b) {
    while (b != 0) {
        int carry = (a & b) << 1; // 进位
        a = a ^ b; // 无进位和
        b = carry; // 进位作为新的 b
    }
    return a;
}

/***
* LeetCode 201 - 数字范围按位与
* 题目链接: https://leetcode.com/problems/bitwise-and-of-numbers-range/
* 给定两个整数 left 和 right，返回区间 [left, right] 内所有数字按位与的结果。
*
* 方法: 寻找公共前缀
* 时间复杂度: O(1) - 最多 32 次循环
* 空间复杂度: O(1)
*
* 原理: 区间内所有数字的按位与结果等于它们的公共二进制前缀
*/
static int rangeBitwiseAnd(int left, int right) {
    int shift = 0;

    // 找到公共前缀
    while (left < right) {
        left >>= 1;
        right >>= 1;
        shift++;
    }

    return left << shift;
}

/***
* LeetCode 338 - 比特位计数
* 题目链接: https://leetcode.com/problems/counting-bits/
* 给定一个非负整数 num。对于  $0 \leq i \leq num$  范围内的每个数字 i，计算其二进制数中的 1 的数目
* 并将它们作为数组返回。
*
* 方法 1: 直接计算每个数字的汉明重量

```

```

* 方法 2: 动态规划 + 最高有效位
* 方法 3: 动态规划 + 最低设置位
*
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*/
static vector<int> countBits(int n) {
    vector<int> result(n + 1, 0);

    // 方法 1: 直接计算 (简单但较慢)
    // for (int i = 0; i <= n; i++) {
    //     result[i] = hammingWeight(i);
    // }

    // 方法 2: 动态规划 + 最高有效位 (推荐)
    // result[0] = 0;
    // int highestBit = 0;
    // for (int i = 1; i <= n; i++) {
    //     if ((i & (i - 1)) == 0) { // 检查是否是 2 的幂
    //         highestBit = i;
    //     }
    //     result[i] = result[i - highestBit] + 1;
    // }

    // 方法 3: 动态规划 + 最低设置位 (最优)
    result[0] = 0;
    for (int i = 1; i <= n; i++) {
        result[i] = result[i & (i - 1)] + 1;
    }

    return result;
}

/**
 * LeetCode 405 - 数字转换为十六进制数
 * 题目链接: https://leetcode.com/problems/convert-a-number-to-a-hexadecimal/
 * 给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用补码运算方法。
 *
 * 方法: 位运算 + 查表
 * 时间复杂度: O(1) - 最多 8 次循环
 * 空间复杂度: O(1)
 */
static string toHex(int num) {

```

```

if (num == 0) return "0";

string hex_chars = "0123456789abcdef";
string result;
unsigned int n = num; // 处理负数补码

while (n > 0) {
    int digit = n & 0xf; // 取最低 4 位
    result = hex_chars[digit] + result;
    n >>= 4; // 右移 4 位
}

return result;
}

/***
 * 额外题目：交换两个数字（不使用临时变量）
 * 使用异或运算交换两个整数
 */
static void swapNumbers(int& a, int& b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}

/***
 * 额外题目：判断奇偶性
 * 使用位运算判断数字的奇偶性
 */
static bool isOdd(int n) {
    return (n & 1) == 1;
}

/***
 * 额外题目：取绝对值
 * 使用位运算计算整数的绝对值
 */
static int absoluteValue(int n) {
    int mask = n >> 31; // 对于负数, mask = -1; 对于正数, mask = 0
    return (n + mask) ^ mask; // 对于负数: n + (-1) = n-1, 然后异或-1相当于取反
}
};

```

```
class PerformanceTester {
public:
    static void testSingleNumber() {
        cout << "==== 只出现一次的数字性能测试 ===" << endl;

        // 生成测试数据
        vector<int> nums1(1000000, 0);
        for (int i = 0; i < 500000; i++) {
            nums1[2*i] = i;
            nums1[2*i+1] = i;
        }
        nums1.push_back(1000000); // 只出现一次的数字

        auto start = chrono::high_resolution_clock::now();
        int result = BitwiseOperations::singleNumber(nums1);
        auto time = chrono::duration_cast<chrono::microseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "singleNumber: 结果=" << result << ", 耗时=" << time << " μs" << endl;
    }

    static void testHammingWeight() {
        cout << "\n==== 汉明重量性能测试 ===" << endl;

        uint32_t test_num = 0xFFFFFFFF; // 所有位都是 1

        auto start = chrono::high_resolution_clock::now();
        int result = BitwiseOperations::hammingWeight(test_num);
        auto time = chrono::duration_cast<chrono::nanoseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "hammingWeight: 结果=" << result << ", 耗时=" << time << " ns" << endl;
    }

    static void runUnitTests() {
        cout << "==== 位运算算法单元测试 ===" << endl;

        // 测试 singleNumber
        vector<int> nums1 = {2, 2, 1};
        assert(BitwiseOperations::singleNumber(nums1) == 1);

        vector<int> nums2 = {4, 1, 2, 1, 2};
        assert(BitwiseOperations::singleNumber(nums2) == 4);
    }
}
```

```
// 测试 isPowerOfTwo
assert(BitwiseOperations::isPowerOfTwo(1) == true);
assert(BitwiseOperations::isPowerOfTwo(16) == true);
assert(BitwiseOperations::isPowerOfTwo(3) == false);

// 测试 getSum
assert(BitwiseOperations::getSum(1, 2) == 3);
assert(BitwiseOperations::getSum(-1, 1) == 0);

cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;

    vector<pair<string, string>> algorithms = {
        {"singleNumber", "O(n), O(1)"},  

        {"singleNumberII", "O(n), O(1)"},  

        {"singleNumberIII", "O(n), O(1)"},  

        {"hammingWeight", "O(1), O(1)"},  

        {"isPowerOfTwo", "O(1), O(1)"},  

        {"getSum", "O(1), O(1)"},  

        {"countBits", "O(n), O(n)"}
    };

    for (auto& algo : algorithms) {
        cout << algo.first << ": 时间复杂度=" << algo.second << endl;
    }
}

int main() {
    cout << "位运算算法实现" << endl;
    cout << "包含 LeetCode 多个位运算相关题目的解决方案" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testSingleNumber();
    PerformanceTester::testHammingWeight();
```

```

// 复杂度分析
PerformanceTester::complexityAnalysis();

// 示例使用
cout << "\n==== 示例使用 ===" << endl;

// 只出现一次的数字示例
vector<int> nums = {4, 1, 2, 1, 2};
cout << "数组: ";
for (int num : nums) cout << num << " ";
cout << endl;
cout << "只出现一次的数字: " << BitwiseOperations::singleNumber(nums) << endl;

// 2 的幂示例
int test_num = 16;
cout << test_num << " 是 2 的幂: " << (BitwiseOperations::isPowerOfTwo(test_num) ? "是" : "否")
") << endl;

// 汉明重量示例
uint32_t n = 11; // 二进制: 1011
cout << n << " 的汉明重量: " << BitwiseOperations::hammingWeight(n) << endl;

// 数字交换示例
int a = 5, b = 10;
cout << "交换前: a=" << a << ", b=" << b << endl;
BitwiseOperations::swapNumbers(a, b);
cout << "交换后: a=" << a << ", b=" << b << endl;

return 0;
}
=====

文件: Code17_BitwiseOperations.java
=====
package class032;

import java.util.*;

/**
 * 位运算经典题目集合
 * 题目来源: LeetCode, HackerRank, Codeforces 等

```

```

=====

文件: Code17_BitwiseOperations.java
=====
package class032;

import java.util.*;

/**
 * 位运算经典题目集合
 * 题目来源: LeetCode, HackerRank, Codeforces 等

```

* 包含各种位运算技巧和常见问题

*

* 解题思路:

* 方法 1: 位运算基础操作

* 方法 2: 位掩码技巧

* 方法 3: 位计数算法

* 方法 4: 位操作优化

*

* 时间复杂度分析:

* 方法 1: $O(1)$ - 常数时间操作

* 方法 2: $O(n)$ - 线性扫描

* 方法 3: $O(1)$ - 查表法

* 方法 4: $O(\log n)$ - 对数时间

*

* 空间复杂度分析:

* 方法 1: $O(1)$ - 常数空间

* 方法 2: $O(1)$ - 常数空间

* 方法 3: $O(1)$ - 常数空间

* 方法 4: $O(1)$ - 常数空间

*

* 工程化考量:

* 1. 边界处理: 处理负数、零、溢出等情况

* 2. 性能优化: 使用位运算替代算术运算

* 3. 可读性: 添加详细注释说明位运算含义

* 4. 测试覆盖: 覆盖各种边界情况

*/

```
public class Code17_BitwiseOperations {
```

```
/**
```

```
 * LeetCode 191. Number of 1 Bits - 计算二进制中 1 的个数
```

```
 * 题目链接: https://leetcode.com/problems/number-of-1-bits/
```

```
 * 题目描述: 计算一个无符号整数的二进制表示中 1 的个数
```

```
*
```

```
* 方法 1: 循环检查每一位
```

```
* 时间复杂度:  $O(32) = O(1)$ 
```

```
* 空间复杂度:  $O(1)$ 
```

```
*/
```

```
public static int hammingWeight1(int n) {
```

```
    int count = 0;
```

```
    for (int i = 0; i < 32; i++) {
```

```
        if ((n & (1 << i)) != 0) {
```

```
            count++;
```

```

        }
    }

    return count;
}

/***
 * 方法 2：使用 n & (n-1) 技巧
 * 每次操作消除最低位的 1
 * 时间复杂度：O(k) k 为 1 的个数
 * 空间复杂度：O(1)
 */
public static int hammingWeight2(int n) {
    int count = 0;
    while (n != 0) {
        n = n & (n - 1); // 消除最低位的 1
        count++;
    }
    return count;
}

/***
 * 方法 3：查表法（适用于大量查询）
 * 时间复杂度：O(1)
 * 空间复杂度：O(256)
 */
public static int hammingWeight3(int n) {
    // 预计算 0-255 的汉明重量
    int[] table = new int[256];
    for (int i = 0; i < 256; i++) {
        table[i] = table[i >> 1] + (i & 1);
    }

    // 分 4 次查表
    return table[n & 0xff] +
           table[(n >> 8) & 0xff] +
           table[(n >> 16) & 0xff] +
           table[(n >> 24) & 0xff];
}

/***
 * LeetCode 231. Power of Two - 判断是否为 2 的幂
 * 题目链接：https://leetcode.com/problems/power-of-two/
 * 题目描述：判断一个整数是否是 2 的幂次方
 */

```

```

/*
 * 方法 1: 使用 n & (n-1) 技巧
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
public static boolean isPowerOfTwo1(int n) {
    if (n <= 0) return false;
    return (n & (n - 1)) == 0;
}

/***
 * 方法 2: 使用 Integer.bitCount()
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
public static boolean isPowerOfTwo2(int n) {
    if (n <= 0) return false;
    return Integer.bitCount(n) == 1;
}

/***
 * LeetCode 136. Single Number - 只出现一次的数字
 * 题目链接: https://leetcode.com/problems/single-number/
 * 题目描述: 给定一个非空整数数组，除了某个元素只出现一次外，其余每个元素均出现两次。找出那个
只出现一次的元素。
 *
 * 方法: 使用异或运算
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
public static int singleNumber(int[] nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num; // 异或运算: 相同为 0, 不同为 1
    }
    return result;
}

/***
 * LeetCode 268. Missing Number - 缺失的数字
 * 题目链接: https://leetcode.com/problems/missing-number/
 * 题目描述: 给定一个包含[0, n]中 n 个数的数组，找出数组中缺失的那个数
 *
 */

```

```

* 方法 1: 异或运算
* 时间复杂度: O(n)
* 空间复杂度: O(1)
*/
public static int missingNumber1(int[] nums) {
    int n = nums.length;
    int result = n; // 初始化为 n, 因为数组长度是 n, 但应该包含 0 到 n 共 n+1 个数

    for (int i = 0; i < n; i++) {
        result ^= i;
        result ^= nums[i];
    }

    return result;
}

/***
 * 方法 2: 数学公式
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
*/
public static int missingNumber2(int[] nums) {
    int n = nums.length;
    int expectedSum = n * (n + 1) / 2;
    int actualSum = 0;

    for (int num : nums) {
        actualSum += num;
    }

    return expectedSum - actualSum;
}

/***
 * LeetCode 338. Counting Bits - 比特位计数
 * 题目链接: https://leetcode.com/problems/counting-bits/
 * 题目描述: 给定一个非负整数 n, 计算 0 到 n 之间的每个数字的二进制表示中 1 的个数
 *
 * 方法 1: 动态规划 + 最高有效位
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
*/
public static int[] countBits1(int n) {

```

```

int[] result = new int[n + 1];
result[0] = 0;

for (int i = 1; i <= n; i++) {
    // i & (i-1) 可以消除最低位的 1
    result[i] = result[i & (i - 1)] + 1;
}

return result;
}

/***
 * 方法 2: 动态规划 + 最低有效位
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static int[] countBits2(int n) {
    int[] result = new int[n + 1];
    result[0] = 0;

    for (int i = 1; i <= n; i++) {
        // i >> 1 相当于除以 2
        result[i] = result[i >> 1] + (i & 1);
    }

    return result;
}

/***
 * LeetCode 190. Reverse Bits - 颠倒二进制位
 * 题目链接: https://leetcode.com/problems/reverse-bits/
 * 题目描述: 颠倒给定的 32 位无符号整数的二进制位
 *
 * 方法 1: 逐位反转
 * 时间复杂度: O(32) = O(1)
 * 空间复杂度: O(1)
 */
public static int reverseBits1(int n) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result <<= 1; // 左移一位, 为新的位腾出空间
        result |= (n & 1); // 取 n 的最低位
        n >>>= 1; // 无符号右移
    }
}

```

```

    }

    return result;
}

/***
 * 方法 2: 分治法 (更高效)
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
public static int reverseBits2(int n) {
    // 分治交换: 先交换 16 位块, 然后 8 位, 4 位, 2 位, 1 位
    n = (n >>> 16) | (n << 16);
    n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1);
    return n;
}

/***
 * LeetCode 371. Sum of Two Integers - 两整数之和
 * 题目链接: https://leetcode.com/problems/sum-of-two-integers/
 * 题目描述: 不使用运算符+和-, 计算两整数之和
 *
 * 方法: 使用位运算模拟加法
 * 时间复杂度: O(1) 最多 32 次循环
 * 空间复杂度: O(1)
 */
public static int getSum(int a, int b) {
    while (b != 0) {
        int carry = (a & b) << 1; // 计算进位
        a = a ^ b; // 计算无进位和
        b = carry; // 将进位作为新的 b
    }
    return a;
}

/***
 * LeetCode 201. Bitwise AND of Numbers Range - 数字范围按位与
 * 题目链接: https://leetcode.com/problems/bitwise-and-of-numbers-range/
 * 题目描述: 给定两个整数 left 和 right, 返回此区间内所有数字按位与的结果
 *
 * 方法 1: 寻找公共前缀
 */

```

```
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*/
public static int rangeBitwiseAnd1(int left, int right) {
    int shift = 0;
    // 找到 left 和 right 的公共前缀
    while (left < right) {
        left >>= 1;
        right >>= 1;
        shift++;
    }
    return left << shift;
}

/***
 * 方法 2: Brian Kernighan 算法
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
*/
public static int rangeBitwiseAnd2(int left, int right) {
    while (left < right) {
        // 清除 right 的最低位的 1
        right = right & (right - 1);
    }
    return right;
}

/***
 * 位运算工具方法: 判断奇偶性
*/
public static boolean isOdd(int n) {
    return (n & 1) == 1; // 奇数的最后一位是 1
}

/***
 * 位运算工具方法: 交换两个数
*/
public static void swap(int a, int b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    System.out.println("交换后: a=" + a + ", b=" + b);
}
```

```
/**  
 * 位运算工具方法：取绝对值  
 */  
  
public static int abs(int n) {  
    int mask = n >> 31; // 对于负数，mask 是全 1；对于正数，mask 是全 0  
    return (n + mask) ^ mask; // 等价于 (n ^ mask) - mask  
}  
  
/**  
 * 单元测试方法  
 */  
  
public static void runTests() {  
    System.out.println("== 位运算经典题目 - 单元测试 ==");  
  
    // 测试汉明重量  
    System.out.println("汉明重量测试:");  
    int test1 = 11; // 二进制：1011  
    System.out.printf("数字%d 的二进制中 1 的个数: %d (方法 1)%n", test1,  
hammingWeight1(test1));  
    System.out.printf("数字%d 的二进制中 1 的个数: %d (方法 2)%n", test1,  
hammingWeight2(test1));  
    System.out.printf("数字%d 的二进制中 1 的个数: %d (方法 3)%n", test1,  
hammingWeight3(test1));  
  
    // 测试 2 的幂  
    System.out.println("\n2 的幂测试:");  
    System.out.printf("8 是 2 的幂: %b%n", isPowerOfTwo1(8));  
    System.out.printf("7 是 2 的幂: %b%n", isPowerOfTwo1(7));  
  
    // 测试只出现一次的数字  
    System.out.println("\n只出现一次的数字测试:");  
    int[] nums = {4, 1, 2, 1, 2};  
    System.out.printf("数组%s 中只出现一次的数字: %d%n", Arrays.toString(nums),  
singleNumber(nums));  
  
    // 测试缺失的数字  
    System.out.println("\n缺失的数字测试:");  
    int[] nums2 = {3, 0, 1};  
    System.out.printf("数组%s 中缺失的数字: %d (方法 1)%n", Arrays.toString(nums2),  
missingNumber1(nums2));  
    System.out.printf("数组%s 中缺失的数字: %d (方法 2)%n", Arrays.toString(nums2),  
missingNumber2(nums2));
```

```
// 测试比特位计数
System.out.println("\n 比特位计数测试:");
int[] bits = countBits1(5);
System.out.printf("0 到 5 的比特位计数: %s%n", Arrays.toString(bits));

// 测试两整数之和
System.out.println("\n 两整数之和测试:");
System.out.printf("3 + 5 = %d%n", getSum(3, 5));

// 测试数字范围按位与
System.out.println("\n 数字范围按位与测试:");
System.out.printf("[5, 7]的按位与结果: %d%n", rangeBitwiseAnd1(5, 7));
}

/**
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n== 性能测试 ==");

    // 测试汉明重量的不同方法
    int testValue = 0xffffffff; // 全 1 的 32 位数

    long startTime = System.nanoTime();
    int result1 = hammingWeight1(testValue);
    long time1 = System.nanoTime() - startTime;
    System.out.printf("方法 1 (循环): %d ns, 结果: %d%n", time1, result1);

    startTime = System.nanoTime();
    int result2 = hammingWeight2(testValue);
    long time2 = System.nanoTime() - startTime;
    System.out.printf("方法 2 (n&(n-1)): %d ns, 结果: %d%n", time2, result2);

    startTime = System.nanoTime();
    int result3 = hammingWeight3(testValue);
    long time3 = System.nanoTime() - startTime;
    System.out.printf("方法 3 (查表法): %d ns, 结果: %d%n", time3, result3);

    // 测试反转比特位的不同方法
    startTime = System.nanoTime();
    int reverse1 = reverseBits1(0x12345678);
    long time4 = System.nanoTime() - startTime;
```

```
System.out.printf("反转比特位方法 1: %d ns, 结果: %x%n", time4, reverse1);

startTime = System.nanoTime();
int reverse2 = reverseBits2(0x12345678);
long time5 = System.nanoTime() - startTime;
System.out.printf("反转比特位方法 2: %d ns, 结果: %x%n", time5, reverse2);
}

/***
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("\n==== 复杂度分析 ====");
    System.out.println("位运算算法的核心优势:");
    System.out.println("1. 时间复杂度通常为 O(1) 或 O(n)");
    System.out.println("2. 空间复杂度通常为 O(1)");
    System.out.println("3. 常数项较小，实际运行效率高");

    System.out.println("\n常用位运算技巧:");
    System.out.println("n & (n-1): 消除最低位的 1");
    System.out.println("n & -n: 获取最低位的 1");
    System.out.println("n ^ n = 0: 相同数异或为 0");
    System.out.println("n << k: 乘以 2 的 k 次方");
    System.out.println("n >> k: 除以 2 的 k 次方");

    System.out.println("\n工程化建议:");
    System.out.println("1. 注意整数溢出问题");
    System.out.println("2. 考虑负数的情况");
    System.out.println("3. 添加详细注释说明位运算含义");
    System.out.println("4. 进行充分的边界测试");
}

public static void main(String[] args) {
    System.out.println("位运算经典题目集合");
    System.out.println("包含 LeetCode、HackerRank 等平台的经典位运算问题");

    // 运行单元测试
    runTests();

    // 运行性能测试
    performanceTest();

    // 复杂度分析
}
```

```

complexityAnalysis();

// 实用工具方法演示
System.out.println("\n== 实用工具方法演示 ==");
System.out.printf("判断 15 是否为奇数: %b\n", isOdd(15));
System.out.printf("-5 的绝对值: %d\n", abs(-5));
System.out.print("交换 10 和 20: ");
swap(10, 20);

// 位运算在工程中的应用
System.out.println("\n== 位运算在工程中的应用 ==");
System.out.println("1. 权限管理系统: 使用位掩码表示权限");
System.out.println("2. 状态标记: 使用位标记多种状态");
System.out.println("3. 数据压缩: 使用位操作减少存储空间");
System.out.println("4. 加密算法: 位运算是加密算法的基础");
System.out.println("5. 图形处理: 像素操作大量使用位运算");
}

}

```

文件: Code17_BitwiseOperations.py

=====

位运算算法实现

包含 LeetCode 多个位运算相关题目的解决方案

题目列表:

1. LeetCode 136 - 只出现一次的数字
2. LeetCode 137 - 只出现一次的数字 II
3. LeetCode 260 - 只出现一次的数字 III
4. LeetCode 191 - 位 1 的个数
5. LeetCode 231 - 2 的幂
6. LeetCode 342 - 4 的幂
7. LeetCode 371 - 两整数之和
8. LeetCode 201 - 数字范围按位与
9. LeetCode 338 - 比特位计数
10. LeetCode 405 - 数字转换为十六进制数

时间复杂度分析:

- 位运算操作: $O(1)$ 或 $O(n)$
- 空间复杂度: $O(1)$ 或 $O(n)$

工程化考量：

1. 位运算技巧：使用位运算优化算法
2. 边界处理：处理负数、零等边界情况
3. 性能优化：利用位运算的常数时间优势
4. 可读性：添加详细注释说明位运算原理

"""

```
import time
from typing import List
import sys
```

```
class BitwiseOperations:
```

```
    """位运算算法类"""

```

```
    @staticmethod
```

```
    def single_number(nums: List[int]) -> int:
        """

```

LeetCode 136 - 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次外，其余每个元素均出现两次。找出那个只出现一次的元素。

方法：异或运算

时间复杂度：O(n)

空间复杂度：O(1)

原理： $a \wedge a = 0, a \wedge 0 = a$

所有出现两次的数字异或后为0，最后剩下的就是只出现一次的数字

"""

```
result = 0
```

```
for num in nums:
```

```
    result ^= num
```

```
return result
```

```
    @staticmethod
```

```
    def single_number_ii(nums: List[int]) -> int:
        """

```

LeetCode 137 - 只出现一次的数字 II

给定一个非空整数数组，除了某个元素只出现一次外，其余每个元素均出现三次。找出那个只出现一次的元素。

方法：位计数法

时间复杂度：O(n)

空间复杂度：O(1)

原理：统计每个位上 1 出现的次数，对 3 取模
如果某位上 1 出现的次数不是 3 的倍数，说明只出现一次的数字在该位为 1

```
result = 0
```

```
# 遍历 32 位
```

```
for i in range(32):  
    count = 0
```

```
# 统计第 i 位为 1 的数字个数
```

```
for num in nums:  
    if (num >> i) & 1:  
        count += 1
```

```
# 如果 count % 3 != 0，说明只出现一次的数字在该位为 1
```

```
if count % 3 != 0:  
    result |= (1 << i)
```

```
# 处理负数（Python 中整数是有符号的）
```

```
if result >= (1 << 31):  
    result -= (1 << 32)
```

```
return result
```

```
@staticmethod
```

```
def single_number_iii(nums: List[int]) -> List[int]:
```

```
"""
```

```
LeetCode 260 - 只出现一次的数字 III
```

给定一个整数数组，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

方法：分组异或

时间复杂度：O(n)

空间复杂度：O(1)

原理：

1. 先对所有数字异或，得到两个不同数字的异或结果
2. 找到异或结果中为 1 的某一位（两个数字在该位不同）
3. 根据这一位将数组分成两组，分别异或得到两个数字

```
"""
```

```
# 第一步：计算所有数字的异或
```

```
xor_all = 0
```

```

for num in nums:
    xor_all ^= num

# 第二步：找到为 1 的最低位
diff_bit = 1
while (xor_all & diff_bit) == 0:
    diff_bit <<= 1

# 第三步：根据 diff_bit 分组异或
num1, num2 = 0, 0
for num in nums:
    if num & diff_bit:
        num1 ^= num
    else:
        num2 ^= num

return [num1, num2]

@staticmethod
def hamming_weight(n: int) -> int:
    """
    LeetCode 191 - 位 1 的个数
    编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数。
    """

方法 1：循环检查二进制位
方法 2：Brian Kernighan 算法
时间复杂度：O(1) - 固定 32 位
空间复杂度：O(1)
"""

# 方法 1：循环检查
count = 0
while n:
    count += (n & 1)
    n >>= 1
return count

# 方法 2：Brian Kernighan 算法（更高效）
# count = 0
# while n:
#     n &= (n - 1) # 清除最低位的 1
#     count += 1
# return count

```

```
@staticmethod  
def is_power_of_two(n: int) -> bool:  
    """
```

LeetCode 231 - 2 的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

方法：位运算性质

时间复杂度：O(1)

空间复杂度：O(1)

原理：2 的幂的二进制表示中只有一位是 1

```
n > 0 and (n & (n - 1)) == 0
```

```
"""
```

```
return n > 0 and (n & (n - 1)) == 0
```

```
@staticmethod  
def is_power_of_four(n: int) -> bool:  
    """
```

LeetCode 342 - 4 的幂

给定一个整数，编写一个函数来判断它是否是 4 的幂次方。

方法：位运算 + 数学性质

时间复杂度：O(1)

空间复杂度：O(1)

原理：

1. 必须是 2 的幂： $n > 0 \text{ and } (n \& (n - 1)) == 0$
2. 4 的幂的 1 出现在奇数位： $(n \& 0x55555555) != 0$

```
"""
```

```
return n > 0 and (n & (n - 1)) == 0 and (n & 0x55555555) != 0
```

```
@staticmethod  
def get_sum(a: int, b: int) -> int:  
    """
```

LeetCode 371 - 两整数之和

不使用运算符 + 和 -，计算两整数 a、b 之和。

方法：位运算模拟加法

时间复杂度：O(1) - 最多 32 次循环

空间复杂度：O(1)

原理：

1. 异或运算得到无进位和

```

2. 与运算左移一位得到进位
3. 循环直到进位为 0
"""

# 32 位整数处理
MASK = 0xFFFFFFFF
MAX_INT = 0x7FFFFFFF

a &= MASK
b &= MASK

while b != 0:
    carry = ((a & b) << 1) & MASK # 进位
    a = (a ^ b) & MASK # 无进位和
    b = carry # 进位作为新的 b

# 处理负数
return a if a <= MAX_INT else ~(a ^ MASK)

```

```

@staticmethod
def range_bitwise_and(left: int, right: int) -> int:
"""

LeetCode 201 - 数字范围按位与
给定两个整数 left 和 right，返回区间 [left, right] 内所有数字按位与的结果。

```

方法：寻找公共前缀

时间复杂度：O(1) - 最多 32 次循环

空间复杂度：O(1)

原理：区间内所有数字的按位与结果等于它们的公共二进制前缀

"""

shift = 0

找到公共前缀
while left < right:
 left >>= 1
 right >>= 1
 shift += 1

return left << shift

```

@staticmethod
def count_bits(n: int) -> List[int]:
"""


```

LeetCode 338 - 比特位计数

给定一个非负整数 num。对于 $0 \leq i \leq \text{num}$ 范围内的每个数字 i ，计算其二进制数中的 1 的数目。

方法 1：直接计算每个数字的汉明重量

方法 2：动态规划 + 最高有效位

方法 3：动态规划 + 最低设置位

时间复杂度： $O(n)$

空间复杂度： $O(n)$

"""

```
result = [0] * (n + 1)
```

方法 1：直接计算（简单但较慢）

```
# for i in range(n + 1):
#     result[i] = BitwiseOperations.hamming_weight(i)
```

方法 2：动态规划 + 最高有效位（推荐）

```
# result[0] = 0
# highest_bit = 0
# for i in range(1, n + 1):
#     if (i & (i - 1)) == 0: # 检查是否是 2 的幂
#         highest_bit = i
#     result[i] = result[i - highest_bit] + 1
```

方法 3：动态规划 + 最低设置位（最优）

```
result[0] = 0
for i in range(1, n + 1):
    result[i] = result[i & (i - 1)] + 1
```

```
return result
```

```
@staticmethod
```

```
def to_hex(num: int) -> str:
    """
```

LeetCode 405 - 数字转换为十六进制数

给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用补码运算方法。

方法：位运算 + 查表

时间复杂度： $O(1)$ - 最多 8 次循环

空间复杂度： $O(1)$

"""

```

if num == 0:
    return "0"

hex_chars = "0123456789abcdef"
result = []
# 处理负数补码
n = num if num >= 0 else (1 << 32) + num

while n > 0:
    digit = n & 0xf # 取最低 4 位
    result.append(hex_chars[digit])
    n >>= 4          # 右移 4 位

return ''.join(reversed(result))

@staticmethod
def swap_numbers(a: int, b: int) -> tuple[int, int]:
    """
    额外题目：交换两个数字（不使用临时变量）
    使用异或运算交换两个整数
    """
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b

@staticmethod
def is_odd(n: int) -> bool:
    """
    额外题目：判断奇偶性
    使用位运算判断数字的奇偶性
    """
    return (n & 1) == 1

@staticmethod
def absolute_value(n: int) -> int:
    """
    额外题目：取绝对值
    使用位运算计算整数的绝对值
    """
    mask = n >> 31          # 对于负数, mask = -1; 对于正数, mask = 0
    return (n + mask) ^ mask # 对于负数: n + (-1) = n-1, 然后异或-1 相当于取反

```

```
class PerformanceTester:  
    """性能测试工具类"""  
  
    @staticmethod  
    def test_single_number():  
        """测试只出现一次的数字性能"""  
        print("== 只出现一次的数字性能测试 ==")  
  
        # 生成测试数据  
        nums = [i for i in range(500000)] * 2  
        nums.append(1000000)  # 只出现一次的数字  
  
        start = time.time()  
        result = BitwiseOperations.single_number(nums)  
        elapsed = (time.time() - start) * 1e6  # 微秒  
  
        print(f"single_number: 结果={result}, 耗时={elapsed:.2f} μs")  
  
    @staticmethod  
    def test_hamming_weight():  
        """测试汉明重量性能"""  
        print("\n== 汉明重量性能测试 ==")  
  
        test_num = 0xFFFFFFFF  # 所有位都是 1  
  
        start = time.time()  
        result = BitwiseOperations.hamming_weight(test_num)  
        elapsed = (time.time() - start) * 1e6  # 微秒  
  
        print(f"hamming_weight: 结果={result}, 耗时={elapsed:.2f} μs")  
  
    @staticmethod  
    def run_unit_tests():  
        """运行单元测试"""  
        print("== 位运算算法单元测试 ==")  
  
        # 测试 single_number  
        nums1 = [2, 2, 1]  
        assert BitwiseOperations.single_number(nums1) == 1  
  
        nums2 = [4, 1, 2, 1, 2]  
        assert BitwiseOperations.single_number(nums2) == 4
```

```
# 测试 is_power_of_two
assert BitwiseOperations.is_power_of_two(1) == True
assert BitwiseOperations.is_power_of_two(16) == True
assert BitwiseOperations.is_power_of_two(3) == False

# 测试 get_sum
assert BitwiseOperations.get_sum(1, 2) == 3
assert BitwiseOperations.get_sum(-1, 1) == 0

print("所有单元测试通过!")

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n==== 复杂度分析 ====")

    algorithms = {
        "single_number": ("O(n)", "O(1)"),
        "single_number_ii": ("O(n)", "O(1)"),
        "single_number_iii": ("O(n)", "O(1)"),
        "hamming_weight": ("O(1)", "O(1)"),
        "is_power_of_two": ("O(1)", "O(1)"),
        "get_sum": ("O(1)", "O(1)"),
        "count_bits": ("O(n)", "O(n)")
    }

    for name, (time_complexity, space_complexity) in algorithms.items():
        print(f"{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}")

def main():
    """主函数"""
    print("位运算算法实现")
    print("包含 LeetCode 多个位运算相关题目的解决方案")
    print("=" * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

    # 运行性能测试
    PerformanceTester.test_single_number()
    PerformanceTester.test_hamming_weight()
```

```

# 复杂度分析
PerformanceTester.complexity_analysis()

# 示例使用
print("\n==== 示例使用 ====")

# 只出现一次的数字示例
nums = [4, 1, 2, 1, 2]
print(f"数组: {nums}")
print(f"只出现一次的数字: {BitwiseOperations.single_number(nums)}")

# 2 的幂示例
test_num = 16
print(f"{test_num} 是 2 的幂: {BitwiseOperations.is_power_of_two(test_num)}")

# 汉明重量示例
n = 11 # 二进制: 1011
print(f"{n} 的汉明重量: {BitwiseOperations.hamming_weight(n)}")

# 数字交换示例
a, b = 5, 10
print(f"交换前: a={a}, b={b}")
a, b = BitwiseOperations.swap_numbers(a, b)
print(f"交换后: a={a}, b={b}")

# 十六进制转换示例
num = 255
print(f"{num} 的十六进制: {BitwiseOperations.to_hex(num)}")

```

文件: Code18_BitManipulationAdvanced.cpp

```

#include <iostream>
#include <vector>
#include <set>
#include <stdexcept>
#include <chrono>

```

```
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
#include <set>
```

```
using namespace std;
```

```
/***
 * 高级位操作算法实现
 * 包含 LeetCode 多个高级位操作相关题目的解决方案
 *
 * 题目列表:
 * 1. LeetCode 78 - 子集
 * 2. LeetCode 90 - 子集 II
 * 3. LeetCode 187 - 重复的 DNA 序列
 * 4. LeetCode 190 - 颠倒二进制位
 * 5. LeetCode 318 - 最大单词长度乘积
 * 6. LeetCode 393 - UTF-8 编码验证
 * 7. LeetCode 397 - 整数替换
 * 8. LeetCode 401 - 二进制手表
 * 9. LeetCode 421 - 数组中两个数的最大异或值
 * 10. LeetCode 461 - 汉明距离
 *
 * 时间复杂度分析:
 * - 位运算操作: O(1) 到 O(2^n)
 * - 空间复杂度: O(1) 到 O(n)
 *
 * 工程化考量:
 * 1. 位掩码技巧: 使用位掩码表示集合
 * 2. 状态压缩: 使用位运算压缩状态
```

```
* 3. 性能优化：利用位运算的并行性
* 4. 边界处理：处理整数溢出、负数等边界情况
*/

```

```
class BitManipulationAdvanced {
public:
    /**
     * LeetCode 78 - 子集
     * 题目链接: https://leetcode.com/problems/subsets/
     * 给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
     *
     * 方法：位掩码法
     * 时间复杂度：O(n * 2^n)
     * 空间复杂度：O(2^n)
     *
     * 原理：使用二进制位表示每个元素是否在子集中
     * 从 0 到 2^n-1 的每个数字对应一个子集
    */
    static vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int total = 1 << n; // 2^n 个子集
        vector<vector<int>> result;

        for (int mask = 0; mask < total; mask++) {
            vector<int> subset;
            for (int i = 0; i < n; i++) {
                if (mask & (1 << i)) {
                    subset.push_back(nums[i]);
                }
            }
            result.push_back(subset);
        }

        return result;
    }

    /**
     * LeetCode 90 - 子集 II
     * 题目链接: https://leetcode.com/problems/subsets-ii/
     * 给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
     *
     * 方法：排序 + 位掩码 + 去重
     * 时间复杂度：O(n * 2^n)
    */
}
```

```

* 空间复杂度: O(2^n)
*
* 原理: 先排序, 然后使用位掩码生成子集, 使用集合去重
*/
static vector<vector<int>> subsetsWithDup(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int n = nums.size();
    int total = 1 << n;
    set<vector<int>> unique_subsets;

    for (int mask = 0; mask < total; mask++) {
        vector<int> subset;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                subset.push_back(nums[i]);
            }
        }
        unique_subsets.insert(subset);
    }

    vector<vector<int>> result(unique_subsets.begin(), unique_subsets.end());
    return result;
}

/***
* LeetCode 187 - 重复的 DNA 序列
* 题目链接: https://leetcode.com/problems/repeated-dna-sequences/
* 所有 DNA 都由一系列缩写为 'A', 'C', 'G' 和 'T' 的核苷酸组成, 例如: "ACGAATTCCG"。
* 编写一个函数来找出所有目标子串, 目标子串的长度为 10, 且在 DNA 字符串 s 中出现超过一次。
*
* 方法: 滑动窗口 + 位编码
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* 原理: 使用 2 位表示每个字符, 10 个字符需要 20 位, 可以用整数表示
*/
static vector<string> findRepeatedDnaSequences(string s) {
    if (s.length() < 10) return {};

    // 字符到数字的映射
    unordered_map<char, int> mapping = {{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};
    unordered_map<int, int> count;
    vector<string> result;

```

```

int hash = 0;
// 计算第一个窗口的哈希值
for (int i = 0; i < 10; i++) {
    hash = (hash << 2) | mapping[s[i]];
}
count[hash]++;

```

// 滑动窗口

```

for (int i = 10; i < s.length(); i++) {
    // 移除最左边的字符，添加新字符
    hash = ((hash << 2) & 0xFFFF) | mapping[s[i]];
    count[hash]++;

```

if (count[hash] == 2) { // 第一次重复出现

```

        result.push_back(s.substr(i - 9, 10));
    }
}

```

return result;

}

```

/***
 * LeetCode 190 - 颠倒二进制位
 * 题目链接: https://leetcode.com/problems/reverse-bits/
 * 颠倒给定的 32 位无符号整数的二进制位。
 *
 * 方法: 逐位反转
 * 时间复杂度: O(1) - 固定 32 位
 * 空间复杂度: O(1)
 */

```

```

static uint32_t reverseBits(uint32_t n) {
    uint32_t result = 0;

    for (int i = 0; i < 32; i++) {
        result = (result << 1) | (n & 1);
        n >>= 1;
    }

    return result;
}

```

```
/***
```

```

* LeetCode 318 - 最大单词长度乘积
* 题目链接: https://leetcode.com/problems/maximum-product-of-word-lengths/
* 给定一个字符串数组 words，找到 length(word[i]) * length(word[j]) 的最大值，
* 并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。
*
* 方法：位掩码 + 预计算
* 时间复杂度: O(n^2 + n * L)
* 空间复杂度: O(n)
*
* 原理：使用 26 位表示每个单词包含的字母，没有公共字母即位掩码与运算为 0
*/
static int maxProduct(vector<string>& words) {
    int n = words.size();
    vector<int> masks(n, 0);
    vector<int> lengths(n, 0);

    // 预计算每个单词的位掩码和长度
    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (char c : words[i]) {
            mask |= (1 << (c - 'a'));
        }
        masks[i] = mask;
        lengths[i] = words[i].length();
    }

    int max_product = 0;
    // 检查所有单词对
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if ((masks[i] & masks[j]) == 0) { // 没有公共字母
                max_product = max(max_product, lengths[i] * lengths[j]);
            }
        }
    }

    return max_product;
}

/**
* LeetCode 393 - UTF-8 编码验证
* 题目链接: https://leetcode.com/problems/utf-8-validation/
* 给定一个表示数据的整数数组 data，返回它是否为有效的 UTF-8 编码。

```

```

*
* 方法：位运算检查编码规则
* 时间复杂度：O(n)
* 空间复杂度：O(1)
*
* UTF-8 编码规则：
* 1. 1 字节字符：0xxxxxx
* 2. 2 字节字符：110xxxxx 10xxxxxx
* 3. 3 字节字符：1110xxxx 10xxxxxx 10xxxxxx
* 4. 4 字节字符：11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
*/
static bool validUtf8(vector<int>& data) {
    int count = 0; // 后续字节数量

    for (int byte : data) {
        if (count == 0) {
            // 检查首字节
            if ((byte >> 5) == 0b110) { // 2 字节字符
                count = 1;
            } else if ((byte >> 4) == 0b1110) { // 3 字节字符
                count = 2;
            } else if ((byte >> 3) == 0b11110) { // 4 字节字符
                count = 3;
            } else if ((byte >> 7) != 0) { // 无效首字节
                return false;
            }
        } else {
            // 检查后续字节
            if ((byte >> 6) != 0b10) {
                return false;
            }
            count--;
        }
    }

    return count == 0; // 所有多字节字符都完整
}

/**
* LeetCode 397 - 整数替换
* 题目链接: https://leetcode.com/problems/integer-replacement/
* 给定一个正整数 n，你可以做如下操作：
* 1. 如果 n 是偶数，则用 n / 2 替换 n

```

```

* 2. 如果 n 是奇数，则可以用 n + 1 或 n - 1 替换 n
* 返回 n 变为 1 所需的最小替换次数。
*
* 方法：贪心 + 位运算
* 时间复杂度：O(log n)
* 空间复杂度：O(1)
*/
static int integerReplacement(int n) {
    return integerReplacementHelper((long long)n, 0);
}

private:
    static int integerReplacementHelper(long long n, int steps) {
        if (n == 1) return steps;

        if (n % 2 == 0) {
            return integerReplacementHelper(n / 2, steps + 1);
        } else {
            return min(integerReplacementHelper(n + 1, steps + 1),
                      integerReplacementHelper(n - 1, steps + 1));
        }
    }

public:
    /**
     * LeetCode 401 - 二进制手表
     * 题目链接: https://leetcode.com/problems/binary-watch/
     * 二进制手表顶部有 4 个 LED 代表 小时 (0-11)，底部的 6 个 LED 代表 分钟 (0-59)。
     * 给定一个非负整数 turnedOn，表示当前亮着的 LED 的数量，返回二进制手表可能表示的所有时间。
     *
     * 方法：枚举所有可能的时间组合
     * 时间复杂度：O(1) - 固定 12*60 种可能
     * 空间复杂度：O(1)
     */
    static vector<string> readBinaryWatch(int turnedOn) {
        vector<string> result;

        for (int h = 0; h < 12; h++) {
            for (int m = 0; m < 60; m++) {
                if (__builtin_popcount(h) + __builtin_popcount(m) == turnedOn) {
                    result.push_back(to_string(h) + ":" +
                                     (m < 10 ? "0" : "") + to_string(m));
                }
            }
        }
    }

```

```

    }

}

return result;
}

/***
 * LeetCode 421 - 数组中两个数的最大异或值
 * 题目链接: https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/
 * 给定一个非空数组，数组中元素为 a0, a1, a2, … , an-1，其中 0 ≤ ai < 2^31。
 * 找到 ai 和 aj 最大的异或 (XOR) 运算结果，其中 0 ≤ i, j < n。
 *
 * 方法: 前缀树 + 贪心
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
static int findMaximumXOR(vector<int>& nums) {
    int max_xor = 0, mask = 0;

    for (int i = 31; i >= 0; i--) {
        mask |= (1 << i);
        unordered_set<int> prefixes;

        // 提取前缀
        for (int num : nums) {
            prefixes.insert(num & mask);
        }

        // 尝试设置当前位为 1
        int candidate = max_xor | (1 << i);
        for (int prefix : prefixes) {
            if (prefixes.find(candidate ^ prefix) != prefixes.end()) {
                max_xor = candidate;
                break;
            }
        }
    }

    return max_xor;
}

/***
 * LeetCode 461 - 汉明距离
 */

```

```

* 题目链接: https://leetcode.com/problems/hamming-distance/
* 两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。
*
* 方法: 异或 + 统计 1 的个数
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*/
static int hammingDistance(int x, int y) {
    int xor_val = x ^ y;
    return __builtin_popcount(xor_val);
}

};

class PerformanceTester {
public:
    static void testSubsets() {
        cout << "==== 子集算法性能测试 ===" << endl;

        // 生成测试数据
        vector<int> nums;
        for (int i = 0; i < 20; i++) {
            nums.push_back(i);
        }

        auto start = chrono::high_resolution_clock::now();
        auto result = BitManipulationAdvanced::subsets(nums);
        auto time = chrono::duration_cast<chrono::milliseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "subsets: 子集数量=" << result.size() << ", 耗时=" << time << " ms" << endl;
    }

    static void testMaximumXOR() {
        cout << "\n==== 最大异或值性能测试 ===" << endl;

        // 生成测试数据
        vector<int> nums(10000);
        for (int i = 0; i < 10000; i++) {
            nums[i] = rand() % 1000000;
        }

        auto start = chrono::high_resolution_clock::now();
        int result = BitManipulationAdvanced::findMaximumXOR(nums);

```

```

auto time = chrono::duration_cast<chrono::microseconds>(
    chrono::high_resolution_clock::now() - start).count();

cout << "findMaximumXOR: 结果=" << result << ", 耗时=" << time << " μs" << endl;
}

static void runUnitTests() {
    cout << "==== 高级位操作算法单元测试 ===" << endl;

    // 测试子集
    vector<int> nums = {1, 2, 3};
    auto subsets = BitManipulationAdvanced::subsets(nums);
    assert(subsets.size() == 8); // 2^3 = 8 个子集

    // 测试汉明距离
    assert(BitManipulationAdvanced::hammingDistance(1, 4) == 2);

    // 测试 UTF-8 验证
    vector<int> utf8_data = {197, 130, 1};
    assert(BitManipulationAdvanced::validUtf8(utf8_data) == true);

    cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;

    vector<pair<string, string>> algorithms = {
        {"subsets", "O(n * 2^n), O(2^n)"},  

        {"subsetsWithDup", "O(n * 2^n), O(2^n)"},  

        {"findRepeatedDnaSequences", "O(n), O(n)"},  

        {"reverseBits", "O(1), O(1)"},  

        {"maxProduct", "O(n^2 + n*L), O(n)"},  

        {"validUtf8", "O(n), O(1)"},  

        {"integerReplacement", "O(log n), O(log n)"},  

        {"findMaximumXOR", "O(n), O(n)"}
    };

    for (auto& algo : algorithms) {
        cout << algo.first << ": 时间复杂度=" << algo.second << endl;
    }
}
};
```

```
int main() {
    cout << "高级位操作算法实现" << endl;
    cout << "包含 LeetCode 多个高级位操作相关题目的解决方案" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testSubsets();
    PerformanceTester::testMaximumXOR();

    // 复杂度分析
    PerformanceTester::complexityAnalysis();

    // 示例使用
    cout << "\n==== 示例使用 ===" << endl;

    // 子集示例
    vector<int> nums = {1, 2, 3};
    cout << "数组: ";
    for (int num : nums) cout << num << " ";
    cout << endl;

    auto subsets = BitManipulationAdvanced::subsets(nums);
    cout << "子集数量: " << subsets.size() << endl;
    cout << "前 3 个子集: " << endl;
    for (int i = 0; i < min(3, (int)subsets.size()); i++) {
        cout << " {";
        for (int j = 0; j < subsets[i].size(); j++) {
            cout << subsets[i][j];
            if (j < subsets[i].size() - 1) cout << ", ";
        }
        cout << "}" << endl;
    }

    // 汉明距离示例
    int x = 1, y = 4;
    cout << "汉明距离(" << x << ", " << y << ") = "
         << BitManipulationAdvanced::hammingDistance(x, y) << endl;

    // 二进制表示示例
```

```
    uint32_t n = 43261596; // 00000010100101000001111010011100
    cout << "原始数字: " << n << endl;
    cout << "反转后: " << BitManipulationAdvanced::reverseBits(n) << endl;

    return 0;
}
```

文件: Code18_BitManipulationAdvanced.java

```
=====
package class032;

import java.util.*;

/**
 * 高级位操作技巧和复杂问题
 * 题目来源: LeetCode Hard, Codeforces, 面试难题
 * 包含位操作的高级应用和复杂场景
 *
 * 解题思路:
 * 方法 1: 位掩码 + 状态压缩
 * 方法 2: 位运算 + 数学技巧
 * 方法 3: 分治 + 位操作
 * 方法 4: 动态规划 + 位运算
 *
 * 时间复杂度分析:
 * 方法 1: O(2^n) - 状态压缩
 * 方法 2: O(n) - 线性扫描
 * 方法 3: O(log n) - 分治策略
 * 方法 4: O(n * 2^k) - 带约束的 DP
 *
 * 空间复杂度分析:
 * 方法 1: O(2^n) - 状态存储
 * 方法 2: O(1) - 常数空间
 * 方法 3: O(log n) - 递归栈
 * 方法 4: O(n * 2^k) - DP 数组
 *
 * 工程化考量:
 * 1. 内存优化: 使用位压缩减少空间占用
 * 2. 性能优化: 利用位运算的并行性
 * 3. 可扩展性: 设计通用的位操作工具类
 * 4. 错误处理: 处理边界情况和异常输入
```

```
*/
```

```
public class Code18_BitManipulationAdvanced {
```

```
/**
```

```
* LeetCode 137. Single Number II - 只出现一次的数字 II
```

```
* 题目链接: https://leetcode.com/problems/single-number-ii/
```

```
* 题目描述: 给定一个非空整数数组, 除了某个元素只出现一次外, 其余每个元素均出现三次。找出那个只出现一次的元素。
```

```
*
```

```
* 方法 1: 位计数法
```

```
* 统计每一位上 1 出现的次数, 对 3 取模
```

```
* 时间复杂度: O(32 * n) = O(n)
```

```
* 空间复杂度: O(1)
```

```
*/
```

```
public static int singleNumberII1(int[] nums) {
```

```
    int result = 0;
```

```
    // 遍历 32 位
```

```
    for (int i = 0; i < 32; i++) {
```

```
        int count = 0;
```

```
        // 统计第 i 位为 1 的数字个数
```

```
        for (int num : nums) {
```

```
            if (((num >> i) & 1) == 1) {
```

```
                count++;
```

```
}
```

```
}
```

```
        // 如果 count % 3 != 0, 说明只出现一次的数字在该位为 1
```

```
        if (count % 3 != 0) {
```

```
            result |= (1 << i);
```

```
}
```

```
}
```

```
    return result;
```

```
}
```

```
/**
```

```
* 方法 2: 有限状态自动机
```

```
* 使用两个变量表示三种状态(00, 01, 10)
```

```
* 时间复杂度: O(n)
```

```
* 空间复杂度: O(1)
```

```

*/
public static int singleNumberII2(int[] nums) {
    int ones = 0, twos = 0;

    for (int num : nums) {
        // ones & num: 计算出现两次的位
        // twos & ~num: 消除出现三次的位
        ones = (ones ^ num) & ~twos;
        twos = (twos ^ num) & ~ones;
    }

    return ones;
}

/***
 * LeetCode 260. Single Number III - 只出现一次的数字 III
 * 题目链接: https://leetcode.com/problems/single-number-iii/
 * 题目描述: 给定一个整数数组，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。
 *
 * 方法: 分组异或
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
public static int[] singleNumberIII(int[] nums) {
    // 第一步: 所有数字异或, 得到两个不同数字的异或结果
    int xor = 0;
    for (int num : nums) {
        xor ^= num;
    }

    // 第二步: 找到 xor 中最低位的 1 (这个 1 表示两个数字在该位不同)
    int mask = xor & (-xor); // 获取最低位的 1

    // 第三步: 根据 mask 将数组分成两组, 分别异或
    int[] result = new int[2];
    for (int num : nums) {
        if ((num & mask) == 0) {
            result[0] ^= num;
        } else {
            result[1] ^= num;
        }
    }
}

```

```

    return result;
}

/***
 * LeetCode 421. Maximum XOR of Two Numbers in an Array - 数组中两个数的最大异或值
 * 题目链接: https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/
 * 题目描述: 给定一个非空数组，返回数组中两个数的最大异或值
 *
 * 方法 1: 暴力法
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(1)
 */
public static int findMaximumXOR1(int[] nums) {
    int maxXor = 0;
    int n = nums.length;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            maxXor = Math.max(maxXor, nums[i] ^ nums[j]);
        }
    }

    return maxXor;
}

/***
 * 方法 2: 前缀树 (Trie)
 * 时间复杂度: O(32 * n) = O(n)
 * 空间复杂度: O(32 * n) = O(n)
 */
public static int findMaximumXOR2(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    // 构建前缀树
    TrieNode root = new TrieNode();

    // 插入所有数字
    for (int num : nums) {
        TrieNode node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node.children[bit] == null) {

```

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

// 查找最大异或值
int maxXor = 0;
for (int num : nums) {
    TrieNode node = root;
    int currentXor = 0;

    for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int oppositeBit = 1 - bit; // 希望走相反的位

        if (node.children[oppositeBit] != null) {
            currentXor |= (1 << i);
            node = node.children[oppositeBit];
        } else {
            node = node.children[bit];
        }
    }

    maxXor = Math.max(maxXor, currentXor);
}

return maxXor;
}

// 前缀树节点定义
static class TrieNode {
    TrieNode[] children;

    public TrieNode() {
        children = new TrieNode[2]; // 0 和 1 两个分支
    }
}

// 继续添加其他方法...
=====
```

文件: Code18_BitManipulationAdvanced_part1.py

```
=====
```

```
"""
```

高级位操作算法实现 - 第一部分

包含 LeetCode 多个高级位操作相关题目的解决方案

题目列表:

1. LeetCode 78 - 子集
2. LeetCode 90 - 子集 II
3. LeetCode 187 - 重复的 DNA 序列
4. LeetCode 190 - 颠倒二进制位
5. LeetCode 318 - 最大单词长度乘积

```
"""
```

```
import time
from typing import List, Set, Tuple
import sys
from functools import lru_cache

class BitManipulationAdvanced:
    """高级位操作算法类"""

    @staticmethod
    def subsets(nums: List[int]) -> List[List[int]]:
        """
        LeetCode 78 - 子集
        给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
        """

        方法：位掩码法
        时间复杂度：O(n * 2^n)
        空间复杂度：O(2^n)
```

原理：使用二进制位表示每个元素是否在子集中

从 0 到 $2^n - 1$ 的每个数字对应一个子集

```
"""
```

```
n = len(nums)
total = 1 << n # 2^n 个子集
result = []

for mask in range(total):
    subset = []
    for i in range(n):
        if mask & (1 << i):
```

```
        subset.append(nums[i])
        result.append(subset)

    return result
```

```
@staticmethod
def subsets_with_dup(nums: List[int]) -> List[List[int]]:
    """
```

LeetCode 90 - 子集 II

给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

方法：排序 + 位掩码 + 去重

时间复杂度： $O(n * 2^n)$

空间复杂度： $O(2^n)$

原理：先排序，然后使用位掩码生成子集，使用集合去重

```
"""
```

```
nums.sort()
```

```
n = len(nums)
```

```
total = 1 << n
```

```
unique_subsets = set()
```

```
for mask in range(total):
```

```
    subset = []
```

```
    for i in range(n):
```

```
        if mask & (1 << i):
```

```
            subset.append(nums[i])
```

```
# 将列表转换为元组以便放入集合
```

```
unique_subsets.add(tuple(subset))
```

```
# 将元组转换回列表
```

```
result = [list(subset) for subset in unique_subsets]
```

```
return result
```

```
@staticmethod
```

```
def find_repeated_dna_sequences(s: str) -> List[str]:
```

```
"""
```

LeetCode 187 - 重复的 DNA 序列

所有 DNA 都由一系列缩写为 'A'，'C'，'G' 和 'T' 的核苷酸组成。

编写一个函数来找出所有目标子串，目标子串的长度为 10，且在 DNA 字符串 s 中出现超过一次。

方法：滑动窗口 + 位编码

时间复杂度： $O(n)$

空间复杂度: $O(n)$

原理: 使用 2 位表示每个字符, 10 个字符需要 20 位, 可以用整数表示

"""

```
if len(s) < 10:  
    return []
```

字符到数字的映射

```
mapping = {'A': 0, 'C': 1, 'G': 2, 'T': 3}
```

```
count = {}
```

```
result = []
```

计算第一个窗口的哈希值

```
hash_val = 0
```

```
for i in range(10):
```

```
    hash_val = (hash_val << 2) | mapping[s[i]]
```

```
count[hash_val] = 1
```

滑动窗口

```
for i in range(10, len(s)):
```

移除最左边的字符, 添加新字符

```
hash_val = ((hash_val << 2) & 0xFFFF) | mapping[s[i]]
```

```
if hash_val in count:
```

```
    count[hash_val] += 1
```

```
    if count[hash_val] == 2: # 第一次重复出现
```

```
        result.append(s[i-9:i+1])
```

```
else:
```

```
    count[hash_val] = 1
```

```
return result
```

@staticmethod

```
def reverse_bits(n: int) -> int:
```

"""

LeetCode 190 - 颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

方法: 逐位反转

时间复杂度: $O(1)$ - 固定 32 位

空间复杂度: $O(1)$

"""

```
result = 0
```

```

for i in range(32):
    result = (result << 1) | (n & 1)
    n >>= 1

return result

@staticmethod
def max_product(words: List[str]) -> int:
    """
    LeetCode 318 - 最大单词长度乘积
    给定一个字符串数组 words，找到 length(word[i]) * length(word[j]) 的最大值，
    并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。
    """

```

方法：位掩码 + 预计算

时间复杂度：O(n^2 + n * L)

空间复杂度：O(n)

原理：使用 26 位表示每个单词包含的字母，没有公共字母即位掩码与运算为 0

"""

```

n = len(words)
masks = [0] * n
lengths = [0] * n

# 预计算每个单词的位掩码和长度
for i in range(n):
    mask = 0
    for c in words[i]:
        mask |= (1 << (ord(c) - ord('a')))
    masks[i] = mask
    lengths[i] = len(words[i])

max_product = 0
# 检查所有单词对
for i in range(n):
    for j in range(i + 1, n):
        if (masks[i] & masks[j]) == 0: # 没有公共字母
            max_product = max(max_product, lengths[i] * lengths[j])

return max_product
=====
```

文件: Code18_BitManipulationAdvanced_part2.py

```
=====
```

```
"""
```

高级位操作算法实现 - 第二部分

包含 LeetCode 多个高级位操作相关题目的解决方案

题目列表:

6. LeetCode 393 - UTF-8 编码验证
7. LeetCode 397 - 整数替换
8. LeetCode 401 - 二进制手表
9. LeetCode 421 - 数组中两个数的最大异或值
10. LeetCode 461 - 汉明距离

```
"""
```

```
import time
from typing import List, Set, Tuple
import sys
from functools import lru_cache

class BitManipulationAdvanced:
    """高级位操作算法类"""

    @staticmethod
    def valid_utf8(data: List[int]) -> bool:
        """
        LeetCode 393 - UTF-8 编码验证
        给定一个表示数据的整数数组 data，返回它是否为有效的 UTF-8 编码。
        """

        方法：位运算检查编码规则
        时间复杂度：O(n)
        空间复杂度：O(1)
```

UTF-8 编码规则:

1. 1 字节字符: 0xxxxxxx
2. 2 字节字符: 110xxxxx 10xxxxxx
3. 3 字节字符: 1110xxxx 10xxxxxx 10xxxxxx
4. 4 字节字符: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

```
"""
```

```
count = 0 # 后续字节数量
```

```
for byte in data:
```

```
    if count == 0:
```

```
        # 检查首字节
```

```

    if (byte >> 5) == 0b110:          # 2 字节字符
        count = 1
    elif (byte >> 4) == 0b1110:      # 3 字节字符
        count = 2
    elif (byte >> 3) == 0b11110:     # 4 字节字符
        count = 3
    elif (byte >> 7) != 0:           # 无效首字节
        return False
    else:
        # 检查后续字节
        if (byte >> 6) != 0b10:
            return False
        count -= 1

    return count == 0 # 所有多字节字符都完整

```

```

@staticmethod
def integer_replacement(n: int) -> int:
    """

```

LeetCode 397 - 整数替换
 给定一个正整数 n，你可以做如下操作：
 1. 如果 n 是偶数，则用 n / 2 替换 n
 2. 如果 n 是奇数，则可以用 n + 1 或 n - 1 替换 n
 返回 n 变为 1 所需的最小替换次数。

方法：贪心 + 位运算

时间复杂度：O(log n)

空间复杂度：O(log n) - 递归深度

"""

```

@lru_cache(None)
def helper(n: int) -> int:
    if n == 1:
        return 0
    if n % 2 == 0:
        return 1 + helper(n // 2)
    else:
        return 1 + min(helper(n + 1), helper(n - 1))

return helper(n)

```

```

@staticmethod
def read_binary_watch(turned_on: int) -> List[str]:
    """

```

LeetCode 401 - 二进制手表

二进制手表顶部有 4 个 LED 代表 小时 (0-11), 底部的 6 个 LED 代表 分钟 (0-59)。

给定一个非负整数 turnedOn , 表示当前亮着的 LED 的数量, 返回二进制手表可能表示的所有时间。

方法: 枚举所有可能的时间组合

时间复杂度: O(1) - 固定 12*60 种可能

空间复杂度: O(1)

"""

```
result = []
```

```
for h in range(12):
```

```
    for m in range(60):
```

```
        if bin(h).count('1') + bin(m).count('1') == turned_on:
```

```
            result.append(f'{h}:{m:02d}'")
```

```
return result
```

```
@staticmethod
```

```
def find_maximum_xor(nums: List[int]) -> int:
```

"""

LeetCode 421 - 数组中两个数的最大异或值

给定一个非空数组, 数组中元素为 $a_0, a_1, a_2, \dots, a_{n-1}$, 其中 $0 \leq a_i < 2^{31}$ 。

找到 a_i 和 a_j 最大的异或 (XOR) 运算结果, 其中 $0 \leq i, j < n$ 。

方法: 前缀树 + 贪心

时间复杂度: O(n)

空间复杂度: O(n)

"""

```
max_xor = 0
```

```
mask = 0
```

```
for i in range(31, -1, -1):
```

```
    mask |= (1 << i)
```

```
    prefixes = set()
```

提取前缀

```
for num in nums:
```

```
    prefixes.add(num & mask)
```

尝试设置当前位为 1

```
candidate = max_xor | (1 << i)
```

```
for prefix in prefixes:
```

```
    if (candidate ^ prefix) in prefixes:
```

```

        max_xor = candidate
        break

    return max_xor

@staticmethod
def hamming_distance(x: int, y: int) -> int:
    """
    LeetCode 461 - 汉明距离
    两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。
    方法: 异或 + 统计 1 的个数
    时间复杂度: O(1)
    空间复杂度: O(1)
    """
    xor_val = x ^ y
    return bin(xor_val).count('1')

```

```

class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def test_subsets():
        """测试子集算法性能"""
        print("== 子集算法性能测试 ==")

        # 生成测试数据
        nums = list(range(20))

        start = time.time()
        result = BitManipulationAdvanced.subsets(nums)
        elapsed = (time.time() - start) * 1000  # 毫秒

        print(f"subsets: 子集数量={len(result)}, 耗时={elapsed:.2f} ms")

    @staticmethod
    def test_maximum_xor():
        """测试最大异或值性能"""
        print("\n== 最大异或值性能测试 ==")

        # 生成测试数据
        import random

```

```

nums = [random.randint(0, 1000000) for _ in range(10000)]


start = time.time()
result = BitManipulationAdvanced.find_maximum_xor(nums)
elapsed = (time.time() - start) * 1000 # 毫秒

print(f"find_maximum_xor: 结果={result}, 耗时={elapsed:.2f} ms")

@staticmethod
def run_unit_tests():
    """运行单元测试"""
    print("== 高级位操作算法单元测试 ==")

    # 测试子集
    nums = [1, 2, 3]
    subsets = BitManipulationAdvanced.subsets(nums)
    assert len(subsets) == 8 # 2^3 = 8 个子集

    # 测试汉明距离
    assert BitManipulationAdvanced.hamming_distance(1, 4) == 2

    # 测试 UTF-8 验证
    utf8_data = [197, 130, 1]
    assert BitManipulationAdvanced.valid_utf8(utf8_data) == True

    print("所有单元测试通过!")


@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n== 复杂度分析 ==")

    algorithms = {
        "subsets": ("O(n * 2^n)", "O(2^n)"),
        "subsets_with_dup": ("O(n * 2^n)", "O(2^n)"),
        "find_repeated_dna_sequences": ("O(n)", "O(n)"),
        "reverse_bits": ("O(1)", "O(1)"),
        "max_product": ("O(n^2 + n*L)", "O(n)"),
        "valid_utf8": ("O(n)", "O(1)"),
        "integer_replacement": ("O(log n)", "O(log n)"),
        "find_maximum_xor": ("O(n)", "O(n)")
    }

```

```
for name, (time_complexity, space_complexity) in algorithms.items():
    print(f'{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}')

def main():
    """主函数"""
    print("高级位操作算法实现")
    print("包含 LeetCode 多个高级位操作相关题目的解决方案")
    print('=' * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

    # 运行性能测试
    PerformanceTester.test_subsets()
    PerformanceTester.test_maximum_xor()

    # 复杂度分析
    PerformanceTester.complexity_analysis()

    # 示例使用
    print("\n====示例使用====")

    # 子集示例
    nums = [1, 2, 3]
    print(f'数组: {nums}')

    subsets = BitManipulationAdvanced.subsets(nums)
    print(f'子集数量: {len(subsets)}')
    print('前 3 个子集:')
    for i in range(min(3, len(subsets))):
        print(f'  {subsets[i]}')


    # 汉明距离示例
    x, y = 1, 4
    print(f'汉明距离({x}, {y}) = {BitManipulationAdvanced.hamming_distance(x, y)}')

    # 二进制表示示例
    n = 43261596 # 00000010100101000001111010011100
    print(f'原始数字: {n}')
    print(f'反转后: {BitManipulationAdvanced.reverse_bits(n)}')
```

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

```
=====
文件: Code19_BitAlgorithmOptimizations.cpp
=====
```

```
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
#include <set>
#include <numeric>
```

```
using namespace std;
```

```
/**
 * 位算法优化实现
 * 包含 LeetCode 多个位算法优化相关题目的解决方案
 *
 * 题目列表:
 * 1. LeetCode 29 - 两数相除
 * 2. LeetCode 50 - Pow(x, n)
 * 3. LeetCode 60 - 排列序列
 * 4. LeetCode 89 - 格雷编码
 * 5. LeetCode 134 - 加油站
```

- * 6. LeetCode 135 - 分发糖果
- * 7. LeetCode 149 - 直线上最多的点数
- * 8. LeetCode 152 - 乘积最大子数组
- * 9. LeetCode 169 - 多数元素
- * 10. LeetCode 229 - 求众数 II
- *
- * 时间复杂度分析:
- * - 位运算优化: $O(1)$ 到 $O(n)$
- * - 空间复杂度: $O(1)$ 到 $O(n)$
- *
- * 工程化考量:
- * 1. 位运算优化: 使用位运算替代乘除法
- * 2. 状态压缩: 使用位运算压缩状态空间
- * 3. 性能优化: 利用位运算的并行性
- * 4. 边界处理: 处理整数溢出、边界值等
- */

```
class BitAlgorithmOptimizations {  
public:  
    /**  
     * LeetCode 29 - 两数相除  
     * 题目链接: https://leetcode.com/problems/divide-two-integers/  
     * 给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。  
     *  
     * 方法: 位运算 + 二分查找  
     * 时间复杂度:  $O(\log n)$   
     * 空间复杂度:  $O(1)$   
     *  
     * 原理: 使用位运算模拟除法，通过左移实现快速乘法  
     */  
    static int divide(int dividend, int divisor) {  
        // 处理特殊情况  
        if (dividend == INT_MIN && divisor == -1) {  
            return INT_MAX; // 溢出情况  
        }  
        if (divisor == 1) return dividend;  
        if (divisor == -1) return -dividend;  
  
        // 确定符号  
        bool negative = (dividend < 0) ^ (divisor < 0);  
  
        // 转换为正数处理 (使用 long long 防止溢出)  
    }
```

```

long long ldividend = labs((long long)dividend);
long long ldivisor = labs((long long)divisor);

long long result = 0;

while (ldividend >= ldivisor) {
    long long temp = ldivisor;
    long long multiple = 1;

    // 使用位运算加速
    while (ldividend >= (temp << 1)) {
        temp <= 1;
        multiple <= 1;
    }

    ldividend -= temp;
    result += multiple;
}

return negative ? -result : result;
}

```

```

/**
 * LeetCode 50 - Pow(x, n)
 * 题目链接: https://leetcode.com/problems/powx-n/
 * 实现 pow(x, n) , 即计算 x 的 n 次幂函数。
 *
 * 方法: 快速幂算法 (位运算)
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * 原理: 将指数 n 分解为二进制, 利用  $x^{(a+b)} = x^a * x^b$ 
 */

```

```

static double myPow(double x, int n) {
    if (n == 0) return 1.0;
    if (x == 1.0) return 1.0;
    if (x == -1.0) return (n % 2 == 0) ? 1.0 : -1.0;

    long long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }
}
```

```

double result = 1.0;
double current_product = x;

// 快速幂算法
for (long long i = N; i > 0; i /= 2) {
    if (i % 2 == 1) {
        result *= current_product;
    }
    current_product *= current_product;
}

return result;
}

/***
 * LeetCode 60 - 排列序列
 * 题目链接: https://leetcode.com/problems/permutation-sequence/
 * 给出集合 [1, 2, 3, ..., n]，其所有元素共有 n! 种排列。
 * 按大小顺序列出所有排列情况，并一一标记，当 n = 3 时，所有排列如下：“123”，“132”，“213”，
 * “231”，“312”，“321”
 * 给定 n 和 k，返回第 k 个排列。
 *
 * 方法：数学 + 位标记
 * 时间复杂度：O(n^2)
 * 空间复杂度：O(n)
 */
static string getPermutation(int n, int k) {
    // 计算阶乘
    vector<int> factorial(n + 1, 1);
    for (int i = 1; i <= n; i++) {
        factorial[i] = factorial[i-1] * i;
    }

    // 标记已使用的数字
    vector<bool> used(n + 1, false);
    string result;

    k--; // 转换为 0-based 索引

    for (int i = n; i >= 1; i--) {
        // 确定当前位的数字
        int segment = factorial[i-1];

```

```

int index = k / segment;
k %= segment;

// 找到第 index 个未使用的数字
int count = 0;
for (int j = 1; j <= n; j++) {
    if (!used[j]) {
        if (count == index) {
            result += to_string(j);
            used[j] = true;
            break;
        }
        count++;
    }
}

return result;
}

/**
 * LeetCode 89 - 格雷编码
 * 题目链接: https://leetcode.com/problems/gray-code/
 * 格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。
 * 给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。即使有多个不同答案，你也只需要返回其中一种。
 *
 * 方法：镜像反射法
 * 时间复杂度：O(2^n)
 * 空间复杂度：O(2^n)
 *
 * 原理：G(i) = i ^ (i >> 1)
 */
static vector<int> grayCode(int n) {
    vector<int> result;
    int total = 1 << n;

    for (int i = 0; i < total; i++) {
        result.push_back(i ^ (i >> 1));
    }

    return result;
}

```

```

/***
 * LeetCode 134 - 加油站
 * 题目链接: https://leetcode.com/problems/gas-station/
 * 在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。
 * 你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
 * 你从其中的一个加油站出发，开始时油箱为空。如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。
 *
 * 方法：贪心算法
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 */
static int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    int total_gas = 0, total_cost = 0;
    int current_gas = 0;
    int start_index = 0;

    for (int i = 0; i < gas.size(); i++) {
        total_gas += gas[i];
        total_cost += cost[i];
        current_gas += gas[i] - cost[i];

        if (current_gas < 0) {
            start_index = i + 1;
            current_gas = 0;
        }
    }

    return total_gas >= total_cost ? start_index : -1;
}

/***
 * LeetCode 135 - 分发糖果
 * 题目链接: https://leetcode.com/problems/candy/
 * 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。
 * 你需要按照以下要求，给这些孩子分发糖果：
 * 1. 每个孩子至少分配到 1 个糖果。
 * 2. 相邻的孩子中，评分高的孩子必须获得更多的糖果。
 *
 * 方法：两次遍历
 * 时间复杂度：O(n)
 */

```

```

* 空间复杂度: O(n)
*/
static int candy(vector<int>& ratings) {
    int n = ratings.size();
    vector<int> candies(n, 1);

    // 从左到右遍历
    for (int i = 1; i < n; i++) {
        if (ratings[i] > ratings[i-1]) {
            candies[i] = candies[i-1] + 1;
        }
    }

    // 从右到左遍历
    for (int i = n-2; i >= 0; i--) {
        if (ratings[i] > ratings[i+1]) {
            candies[i] = max(candies[i], candies[i+1] + 1);
        }
    }

    return accumulate(candies.begin(), candies.end(), 0);
}

/***
* LeetCode 149 - 直线上最多的点数
* 题目链接: https://leetcode.com/problems/max-points-on-a-line/
* 给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。
*
* 方法: 斜率统计 + 最大公约数
* 时间复杂度: O(n^2)
* 空间复杂度: O(n)
*/
static int maxPoints(vector<vector<int>>& points) {
    if (points.size() < 3) return points.size();

    int max_count = 0;

    for (int i = 0; i < points.size(); i++) {
        map<pair<int, int>, int> slope_count;
        int duplicate = 1; // 重复点计数

        for (int j = i + 1; j < points.size(); j++) {
            int dx = points[j][0] - points[i][0];
            int dy = points[j][1] - points[i][1];
            int gcd = __gcd(dx, dy);

            if (dx == 0 && dy == 0) {
                duplicate++;
            } else {
                int slope = dx / gcd;
                int intercept = dy / gcd;
                slope_count[{slope, intercept}]++;
            }
        }

        max_count = max(max_count, slope_count.size());
    }

    return max_count;
}

```

```

        int dy = points[j][1] - points[i][1];

        if (dx == 0 && dy == 0) {
            duplicate++;
            continue;
        }

        // 计算斜率（使用最大公约数约分）
        int g = gcd(dx, dy);
        dx /= g;
        dy /= g;

        slope_count[{dx, dy}]++;
    }

    max_count = max(max_count, duplicate);
    for (auto& p : slope_count) {
        max_count = max(max_count, p.second + duplicate);
    }
}

return max_count;
}

private:
    static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

public:
    /**
     * LeetCode 152 - 乘积最大子数组
     * 题目链接: https://leetcode.com/problems/maximum-product-subarray/
     * 给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。
     *
     * 方法: 动态规划
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     */
    static int maxProduct(vector<int>& nums) {
        if (nums.empty()) return 0;

```

```

int max_product = nums[0];
int min_product = nums[0];
int result = nums[0];

for (int i = 1; i < nums.size(); i++) {
    if (nums[i] < 0) {
        swap(max_product, min_product);
    }

    max_product = max(nums[i], max_product * nums[i]);
    min_product = min(nums[i], min_product * nums[i]);

    result = max(result, max_product);
}

return result;
}

/***
 * LeetCode 169 - 多数元素
 * 题目链接: https://leetcode.com/problems/majority-element/
 * 给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素。
 *
 * 方法: Boyer-Moore 投票算法
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
static int majorityElement(vector<int>& nums) {
    int candidate = nums[0];
    int count = 1;

    for (int i = 1; i < nums.size(); i++) {
        if (count == 0) {
            candidate = nums[i];
            count = 1;
        } else if (nums[i] == candidate) {
            count++;
        } else {
            count--;
        }
    }
}

```

```
    return candidate;
}

/***
 * LeetCode 229 - 求众数 II
 * 题目链接: https://leetcode.com/problems/majority-element-ii/
 * 给定一个大小为 n 的整数数组，找出其中所有出现超过  $\lfloor n/3 \rfloor$  次的元素。
 *
 * 方法: Boyer-Moore 投票算法扩展
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
static vector<int> majorityElementII(vector<int>& nums) {
    if (nums.empty()) return {};

    int candidate1 = 0, candidate2 = 0;
    int count1 = 0, count2 = 0;

    // 第一轮投票
    for (int num : nums) {
        if (num == candidate1) {
            count1++;
        } else if (num == candidate2) {
            count2++;
        } else if (count1 == 0) {
            candidate1 = num;
            count1 = 1;
        } else if (count2 == 0) {
            candidate2 = num;
            count2 = 1;
        } else {
            count1--;
            count2--;
        }
    }

    // 第二轮验证
    count1 = count2 = 0;
    for (int num : nums) {
        if (num == candidate1) count1++;
        else if (num == candidate2) count2++;
    }
}
```

```

vector<int> result;
int n = nums.size();
if (count1 > n / 3) result.push_back(candidate1);
if (count2 > n / 3) result.push_back(candidate2);

return result;
}

};

class PerformanceTester {
public:
    static void testDivide() {
        cout << "==== 两数相除性能测试 ===" << endl;

        int dividend = INT_MAX;
        int divisor = 2;

        auto start = chrono::high_resolution_clock::now();
        int result = BitAlgorithmOptimizations::divide(dividend, divisor);
        auto time = chrono::duration_cast<chrono::nanoseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "divide: " << dividend << " / " << divisor << " = "
            << result << ", 耗时=" << time << " ns" << endl;
    }

    static void testMyPow() {
        cout << "\n==== 快速幂性能测试 ===" << endl;

        double x = 2.0;
        int n = 1000000;

        auto start = chrono::high_resolution_clock::now();
        double result = BitAlgorithmOptimizations::myPow(x, n);
        auto time = chrono::duration_cast<chrono::microseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "myPow: " << x << "^" << n << " = " << result
            << ", 耗时=" << time << " μs" << endl;
    }

    static void runUnitTests() {
        cout << "==== 位算法优化单元测试 ===" << endl;
    }
}

```

```

// 测试两数相除
assert(BitAlgorithmOptimizations::divide(10, 3) == 3);
assert(BitAlgorithmOptimizations::divide(7, -3) == -2);

// 测试快速幂
assert(abs(BitAlgorithmOptimizations::myPow(2.0, 10) - 1024.0) < 1e-9);

// 测试多数元素
vector<int> nums = {2, 2, 1, 1, 1, 2, 2};
assert(BitAlgorithmOptimizations::majorityElement(nums) == 2);

cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;

    vector<pair<string, string>> algorithms = {
        {"divide", "O(log n), O(1)"}, {"myPow", "O(log n), O(1)"}, {"getPermutation", "O(n^2), O(n)"}, {"grayCode", "O(2^n), O(2^n)"}, {"canCompleteCircuit", "O(n), O(1)"}, {"candy", "O(n), O(n)"}, {"maxPoints", "O(n^2), O(n)"}, {"maxProduct", "O(n), O(1)"}, {"majorityElement", "O(n), O(1)"}
    };

    for (auto& algo : algorithms) {
        cout << algo.first << ": 时间复杂度=" << algo.second << endl;
    }
}

int main() {
    cout << "位算法优化实现" << endl;
    cout << "包含 LeetCode 多个位算法优化相关题目的解决方案" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();
}

```

```
// 运行性能测试
PerformanceTester::testDivide();
PerformanceTester::testMyPow();

// 复杂度分析
PerformanceTester::complexityAnalysis();

// 示例使用
cout << "\n==== 示例使用 ===" << endl;

// 两数相除示例
int dividend = 10, divisor = 3;
cout << dividend << " / " << divisor << " = "
    << BitAlgorithmOptimizations::divide(dividend, divisor) << endl;

// 快速幂示例
double x = 2.0;
int n = 10;
cout << x << "^" << n << " = " << BitAlgorithmOptimizations::myPow(x, n) << endl;

// 格雷编码示例
int gray_n = 3;
auto gray_codes = BitAlgorithmOptimizations::grayCode(gray_n);
cout << "格雷编码(n=" << gray_n << "): ";
for (int i = 0; i < min(5, (int)gray_codes.size()); i++) {
    cout << gray_codes[i] << " ";
}
cout << "..." << endl;

// 多数元素示例
vector<int> nums = {2, 2, 1, 1, 1, 2, 2};
cout << "数组: ";
for (int num : nums) cout << num << " ";
cout << endl;
cout << "多数元素: " << BitAlgorithmOptimizations::majorityElement(nums) << endl;

return 0;
}
```

```
=====
```

```
"""
```

位算法优化实现 - 第一部分

包含 LeetCode 多个位算法优化相关题目的解决方案

题目列表:

1. LeetCode 29 - 两数相除
2. LeetCode 50 - Pow(x, n)
3. LeetCode 60 - 排列序列
4. LeetCode 89 - 格雷编码
5. LeetCode 134 - 加油站

```
"""
```

```
import time
from typing import List
import sys
import math
from functools import lru_cache
from collections import defaultdict
```

```
class BitAlgorithmOptimizations:
```

```
    """位算法优化类"""

    @staticmethod
```

```
    def divide(dividend: int, divisor: int) -> int:
        """
```

```
        LeetCode 29 - 两数相除
```

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

方法: 位运算 + 二分查找

时间复杂度: $O(\log n)$

空间复杂度: $O(1)$

原理: 使用位运算模拟除法，通过左移实现快速乘法

```
"""

# 处理特殊情况
```

```
if dividend == -2**31 and divisor == -1:
```

```
    return 2**31 - 1 # 溢出情况
```

```
if divisor == 1:
```

```
    return dividend
```

```
if divisor == -1:
```

```

    return -dividend

# 确定符号
negative = (dividend < 0) ^ (divisor < 0)

# 转换为正数处理
ldividend = abs(dividend)
ldivisor = abs(divisor)

result = 0

while ldividend >= ldivisor:
    temp = ldivisor
    multiple = 1

    # 使用位运算加速
    while ldividend >= (temp << 1):
        temp <= 1
        multiple <= 1

    ldividend -= temp
    result += multiple

return -result if negative else result

```

```

@staticmethod
def my_pow(x: float, n: int) -> float:
    """
    LeetCode 50 - Pow(x, n)
    实现 pow(x, n) ，即计算 x 的 n 次幂函数。
    """

```

方法：快速幂算法（位运算）

时间复杂度： $O(\log n)$

空间复杂度： $O(1)$

原理：将指数 n 分解为二进制，利用 $x^{(a+b)} = x^a * x^b$

```

if n == 0:
    return 1.0
if x == 1.0:
    return 1.0
if x == -1.0:
    return 1.0 if n % 2 == 0 else -1.0

```

```

N = n
if N < 0:
    x = 1 / x
    N = -N

result = 1.0
current_product = x

# 快速幂算法
while N > 0:
    if N % 2 == 1:
        result *= current_product
    current_product *= current_product
    N //= 2

return result

```

```

@staticmethod
def get_permutation(n: int, k: int) -> str:
    """

```

LeetCode 60 - 排列序列

给出集合 $[1, 2, 3, \dots, n]$ ，其所有元素共有 $n!$ 种排列。

按大小顺序列出所有排列情况，并一一标记，返回第 k 个排列。

方法：数学 + 位标记

时间复杂度： $O(n^2)$

空间复杂度： $O(n)$

"""

计算阶乘

```
factorial = [1] * (n + 1)
```

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

```
    factorial[i] = factorial[i-1] * i
```

标记已使用的数字

```
used = [False] * (n + 1)
```

```
result = []
```

$k -= 1$ # 转换为 0-based 索引

```
for i in range(n, 0, -1):
```

确定当前位的数字

```
segment = factorial[i-1]
```

```

index = k // segment
k %= segment

# 找到第 index 个未使用的数字
count = 0
for j in range(1, n + 1):
    if not used[j]:
        if count == index:
            result.append(str(j))
            used[j] = True
            break
        count += 1

return ''.join(result)

```

```

@staticmethod
def gray_code(n: int) -> List[int]:
    """

```

LeetCode 89 - 格雷编码

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。
给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。

方法：镜像反射法

时间复杂度： $O(2^n)$

空间复杂度： $O(2^n)$

原理： $G(i) = i \ ^ (i \gg 1)$

"""

result = []

total = 1 << n

for i in range(total):

result.append(i ^ (i >> 1))

return result

```

@staticmethod
def can_complete_circuit(gas: List[int], cost: List[int]) -> int:
    """

```

LeetCode 134 - 加油站

在一条环路上有 N 个加油站，从其中的一个加油站出发，开始时油箱为空。
如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

方法：贪心算法

时间复杂度：O(n)

空间复杂度：O(1)

"""

```
total_gas = sum(gas)
```

```
total_cost = sum(cost)
```

```
if total_gas < total_cost:
```

```
    return -1
```

```
current_gas = 0
```

```
start_index = 0
```

```
for i in range(len(gas)):
```

```
    current_gas += gas[i] - cost[i]
```

```
    if current_gas < 0:
```

```
        start_index = i + 1
```

```
        current_gas = 0
```

```
return start_index
```

=====

文件：Code19_BitAlgorithmOptimizations_part2.py

=====

"""

位算法优化实现 - 第二部分

包含 LeetCode 多个位算法优化相关题目的解决方案

题目列表：

6. LeetCode 135 - 分发糖果

7. LeetCode 149 - 直线上最多的点数

8. LeetCode 152 - 乘积最大子数组

9. LeetCode 169 - 多数元素

10. LeetCode 229 - 求众数 II

"""

```
import time
```

```
from typing import List
```

```
import sys
```

```
import math
```

```
from functools import lru_cache
```

```
from collections import defaultdict
```

```
class BitAlgorithmOptimizations:
```

```
    """位算法优化类"""
```

```
@staticmethod
```

```
def candy(ratings: List[int]) -> int:
```

```
    """
```

```
    LeetCode 135 - 分发糖果
```

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

要求每个孩子至少分配到 1 个糖果，相邻的孩子中，评分高的孩子必须获得更多的糖果。

方法：两次遍历

时间复杂度： $O(n)$

空间复杂度： $O(n)$

```
"""
```

```
n = len(ratings)
```

```
candies = [1] * n
```

```
# 从左到右遍历
```

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

```
    if ratings[i] > ratings[i-1]:
```

```
        candies[i] = candies[i-1] + 1
```

```
# 从右到左遍历
```

```
for i in range(n-2, -1, -1):
```

```
    if ratings[i] > ratings[i+1]:
```

```
        candies[i] = max(candies[i], candies[i+1] + 1)
```

```
return sum(candies)
```

```
@staticmethod
```

```
def max_points(points: List[List[int]]) -> int:
```

```
    """
```

```
    LeetCode 149 - 直线上最多的点数
```

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

方法：斜率统计 + 最大公约数

时间复杂度： $O(n^2)$

空间复杂度： $O(n)$

```
"""
```

```
if len(points) < 3:
```

```

    return len(points)

max_count = 0

for i in range(len(points)):
    slope_count = defaultdict(int)
    duplicate = 1 # 重复点计数

    for j in range(i + 1, len(points)):
        dx = points[j][0] - points[i][0]
        dy = points[j][1] - points[i][1]

        if dx == 0 and dy == 0:
            duplicate += 1
            continue

        # 计算斜率（使用最大公约数约分）
        g = math.gcd(dx, dy)
        dx //= g
        dy //= g

        slope_count[(dx, dy)] += 1

    max_count = max(max_count, duplicate)
    for count in slope_count.values():
        max_count = max(max_count, count + duplicate)

return max_count

@staticmethod
def max_product(nums: List[int]) -> int:
    """
    LeetCode 152 - 乘积最大子数组
    给你一个整数数组 nums，请你找出数组中乘积最大的连续子数组，并返回该子数组所对应的乘积。
    方法：动态规划
    时间复杂度：O(n)
    空间复杂度：O(1)
    """
    if not nums:
        return 0

    max_product = nums[0]

```

```
min_product = nums[0]
result = nums[0]

for i in range(1, len(nums)):
    if nums[i] < 0:
        max_product, min_product = min_product, max_product

    max_product = max(nums[i], max_product * nums[i])
    min_product = min(nums[i], min_product * nums[i])

    result = max(result, max_product)

return result
```

```
@staticmethod
def majority_element(nums: List[int]) -> int:
    """
    LeetCode 169 - 多数元素
    给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素。
    """

    pass
```

方法：Boyer-Moore 投票算法

时间复杂度：O(n)

空间复杂度：O(1)

"""

```
candidate = nums[0]
```

```
count = 1
```

```
for i in range(1, len(nums)):
```

```
    if count == 0:
```

```
        candidate = nums[i]
```

```
        count = 1
```

```
    elif nums[i] == candidate:
```

```
        count += 1
```

```
    else:
```

```
        count -= 1
```

```
return candidate
```

```
@staticmethod
def majority_element_ii(nums: List[int]) -> List[int]:
    """
    LeetCode 229 - 求众数 II
    """

    pass
```

LeetCode 229 - 求众数 II

给定一个大小为 n 的整数数组，找出其中所有出现超过 $\lfloor n/3 \rfloor$ 次的元素。

方法：Boyer-Moore 投票算法扩展

时间复杂度： $O(n)$

空间复杂度： $O(1)$

"""

```
if not nums:
    return []

candidate1, candidate2 = 0, 0
count1, count2 = 0, 0

# 第一轮投票
for num in nums:
    if num == candidate1:
        count1 += 1
    elif num == candidate2:
        count2 += 1
    elif count1 == 0:
        candidate1 = num
        count1 = 1
    elif count2 == 0:
        candidate2 = num
        count2 = 1
    else:
        count1 -= 1
        count2 -= 1

# 第二轮验证
count1 = count2 = 0
for num in nums:
    if num == candidate1:
        count1 += 1
    elif num == candidate2:
        count2 += 1

result = []
n = len(nums)
if count1 > n // 3:
    result.append(candidate1)
if count2 > n // 3:
    result.append(candidate2)
```

```
        return result

class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def test_divide():
        """测试两数相除性能"""
        print("== 两数相除性能测试 ==")

        dividend = 2**31 - 1
        divisor = 2

        start = time.time()
        result = BitAlgorithmOptimizations.divide(dividend, divisor)
        elapsed = (time.time() - start) * 1e6 # 微秒

        print(f"divide: {dividend} / {divisor} = {result}, 耗时={elapsed:.2f} μs")

    @staticmethod
    def test_my_pow():
        """测试快速幂性能"""
        print("\n== 快速幂性能测试 ==")

        x = 2.0
        n = 1000000

        start = time.time()
        result = BitAlgorithmOptimizations.my_pow(x, n)
        elapsed = (time.time() - start) * 1e6 # 微秒

        print(f"my_pow: {x} ^ {n} = {result}, 耗时={elapsed:.2f} μs")

    @staticmethod
    def run_unit_tests():
        """运行单元测试"""
        print("== 位算法优化单元测试 ==")

        # 测试两数相除
        assert BitAlgorithmOptimizations.divide(10, 3) == 3
        assert BitAlgorithmOptimizations.divide(7, -3) == -2
```

```

# 测试快速幂
assert abs(BitAlgorithmOptimizations.my_pow(2.0, 10) - 1024.0) < 1e-9

# 测试多数元素
nums = [2, 2, 1, 1, 1, 2, 2]
assert BitAlgorithmOptimizations.majority_element(nums) == 2

print("所有单元测试通过!")

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n==== 复杂度分析 ====")

    algorithms = {
        "divide": ("O(log n)", "O(1)"),
        "my_pow": ("O(log n)", "O(1)"),
        "get_permutation": ("O(n^2)", "O(n)"),
        "gray_code": ("O(2^n)", "O(2^n)"),
        "can_complete_circuit": ("O(n)", "O(1)"),
        "candy": ("O(n)", "O(n)"),
        "max_points": ("O(n^2)", "O(n)"),
        "max_product": ("O(n)", "O(1)"),
        "majority_element": ("O(n)", "O(1)")
    }

    for name, (time_complexity, space_complexity) in algorithms.items():
        print(f"{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}")

def main():
    """主函数"""
    print("位算法优化实现")
    print("包含 LeetCode 多个位算法优化相关题目的解决方案")
    print("=" * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

    # 运行性能测试
    PerformanceTester.test_divide()
    PerformanceTester.test_my_pow()

```

```

# 复杂度分析
PerformanceTester.complexity_analysis()

# 示例使用
print("\n==== 示例使用 ====")

# 两数相除示例
dividend, divisor = 10, 3
print(f"{dividend} / {divisor} = {BitAlgorithmOptimizations.divide(dividend, divisor)}")

# 快速幂示例
x, n = 2.0, 10
print(f"{x} ^ {n} = {BitAlgorithmOptimizations.my_pow(x, n)}")

# 格雷编码示例
gray_n = 3
gray_codes = BitAlgorithmOptimizations.gray_code(gray_n)
print(f"格雷编码(n={gray_n}): {gray_codes[:5]}...")

# 多数元素示例
nums = [2, 2, 1, 1, 1, 2, 2]
print(f"数组: {nums}")
print(f"多数元素: {BitAlgorithmOptimizations.majority_element(nums)}")

if __name__ == "__main__":
    main()

```

=====

文件: Code19_BitmaskApplications.java

=====

```

package class032;

import java.util.*;

/**
 * 位掩码应用场景和实际问题
 * 题目来源: LeetCode, HackerRank, 实际工程问题
 * 包含位掩码在各种场景下的实际应用
 *
 * 解题思路:
 * 方法 1: 状态压缩 + 位掩码

```

- * 方法 2: 集合操作 + 位运算
- * 方法 3: 权限管理 + 位标记
- * 方法 4: 数据压缩 + 位操作
- *
- * 时间复杂度分析:
 - * 方法 1: $O(2^n)$ - 状态枚举
 - * 方法 2: $O(n)$ - 线性处理
 - * 方法 3: $O(1)$ - 常数操作
 - * 方法 4: $O(\log n)$ - 对数处理
- *
- * 空间复杂度分析:
 - * 方法 1: $O(2^n)$ - 状态存储
 - * 方法 2: $O(1)$ - 常数空间
 - * 方法 3: $O(1)$ - 常数空间
 - * 方法 4: $O(1)$ - 常数空间
- *
- * 工程化考量:
 - * 1. 可读性: 使用常量定义位掩码含义
 - * 2. 可维护性: 设计清晰的接口
 - * 3. 性能: 利用位运算的高效性
 - * 4. 安全性: 处理权限验证

```
public class Code19_BitmaskApplications {  
  
    /**  
     * 权限管理系统示例  
     * 使用位掩码表示用户权限  
     */  
  
    public static class PermissionSystem {  
        // 权限常量定义  
        public static final int READ = 1 << 0;      // 0001 - 读权限  
        public static final int WRITE = 1 << 1;     // 0010 - 写权限  
        public static final int EXECUTE = 1 << 2;   // 0100 - 执行权限  
        public static final int DELETE = 1 << 3;    // 1000 - 删除权限  
  
        /**  
         * 检查用户是否具有特定权限  
         * @param userPermissions 用户权限掩码  
         * @param requiredPermission 需要检查的权限  
         * @return 是否具有权限  
         */  
        public static boolean hasPermission(int userPermissions, int requiredPermission) {
```

```
        return (userPermissions & requiredPermission) != 0;
    }

    /**
     * 添加权限
     * @param userPermissions 用户当前权限
     * @param permissionToAdd 要添加的权限
     * @return 新的权限掩码
     */
    public static int addPermission(int userPermissions, int permissionToAdd) {
        return userPermissions | permissionToAdd;
    }

    /**
     * 移除权限
     * @param userPermissions 用户当前权限
     * @param permissionToRemove 要移除的权限
     * @return 新的权限掩码
     */
    public static int removePermission(int userPermissions, int permissionToRemove) {
        return userPermissions & ~permissionToRemove;
    }

    /**
     * 切换权限（有则移除，无则添加）
     * @param userPermissions 用户当前权限
     * @param permissionToToggle 要切换的权限
     * @return 新的权限掩码
     */
    public static int togglePermission(int userPermissions, int permissionToToggle) {
        return userPermissions ^ permissionToToggle;
    }

    /**
     * 获取所有权限列表
     * @param userPermissions 用户权限掩码
     * @return 权限名称列表
     */
    public static List<String> getPermissionList(int userPermissions) {
        List<String> permissions = new ArrayList<>();

        if (hasPermission(userPermissions, READ)) {
            permissions.add("READ");
        }
    }
}
```

```
        }

        if (hasPermission(userPermissions, WRITE)) {
            permissions.add("WRITE");
        }

        if (hasPermission(userPermissions, EXECUTE)) {
            permissions.add("EXECUTE");
        }

        if (hasPermission(userPermissions, DELETE)) {
            permissions.add("DELETE");
        }

    }

    return permissions;
}

/***
 * 权限掩码转字符串
 * @param permissions 权限掩码
 * @return 二进制字符串表示
 */
public static String permissionsToString(int permissions) {
    return String.format("%4s", Integer.toBinaryString(permissions))
        .replace(' ', '0');
}

}

/***
 * 状态机设计示例
 * 使用位掩码表示复杂状态
 */
public static class StateMachine {

    // 状态定义
    public static final int IDLE = 1 << 0;
    public static final int RUNNING = 1 << 1;
    public static final int PAUSED = 1 << 2;
    public static final int STOPPED = 1 << 3;
    public static final int ERROR = 1 << 4;

    /***
     * 检查状态是否有效
     * @param state 当前状态
     * @return 是否有效状态
     */
    public static boolean isValidState(int state) {
```

```

// 状态应该是 2 的幂（只有一个位被设置）
return state != 0 && (state & (state - 1)) == 0;
}

/**
 * 状态转换验证
 * @param fromState 起始状态
 * @param toState 目标状态
 * @return 是否允许转换
 */
public static boolean canTransition(int fromState, int toState) {
    // 定义允许的状态转换
    int[][] allowedTransitions = {
        {IDLE, RUNNING},
        {RUNNING, PAUSED},
        {RUNNING, STOPPED},
        {PAUSED, RUNNING},
        {PAUSED, STOPPED},
        {STOPPED, IDLE},
        {ERROR, IDLE}
    };

    for (int[] transition : allowedTransitions) {
        if (transition[0] == fromState && transition[1] == toState) {
            return true;
        }
    }
    return false;
}

/**
 * 获取所有可能的状态
 * @return 状态列表
 */
public static List<Integer> getAllStates() {
    return Arrays.asList(IDLE, RUNNING, PAUSED, STOPPED, ERROR);
}

/**
 * 数据压缩示例
 * 使用位操作压缩布尔数组
*/

```

```
public static class BooleanArrayCompressor {
    private int[] data;
    private int size;

    public BooleanArrayCompressor(int capacity) {
        // 每个 int 可以存储 32 个布尔值
        this.data = new int[(capacity + 31) / 32];
        this.size = capacity;
    }

    /**
     * 设置指定位置的布尔值
     * @param index 位置索引
     * @param value 布尔值
     */
    public void set(int index, boolean value) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
        }

        int arrayIndex = index / 32;
        int bitIndex = index % 32;

        if (value) {
            // 设置位为 1
            data[arrayIndex] |= (1 << bitIndex);
        } else {
            // 设置位为 0
            data[arrayIndex] &= ~(1 << bitIndex);
        }
    }

    /**
     * 获取指定位置的布尔值
     * @param index 位置索引
     * @return 布尔值
     */
    public boolean get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
        }

        int arrayIndex = index / 32;
```

```
int bitIndex = index % 32;

return (data[arrayIndex] & (1 << bitIndex)) != 0;
}

/***
 * 统计为 true 的个数
 * @return true 的个数
 */
public int countTrue() {
    int count = 0;
    for (int value : data) {
        count += Integer.bitCount(value);
    }
    return count;
}

/***
 * 获取压缩后的数据大小 (字节)
 * @return 数据大小
 */
public int getCompressedSize() {
    return data.length * 4; // 每个 int 占 4 字节
}

/***
 * 获取原始数据大小 (如果使用 boolean 数组)
 * @return 原始大小
 */
public int getOriginalSize() {
    return size; // 每个 boolean 在 Java 中至少占 1 字节
}

/***
 * 计算压缩比
 * @return 压缩比 (原始大小/压缩大小)
 */
public double getCompressionRatio() {
    return (double) getOriginalSize() / getCompressedSize();
}

/***
```

```
* 集合操作工具类
* 使用位掩码表示小范围整数集合
*/
public static class BitSetUtils {

    /**
     * 创建包含指定元素的集合
     * @param elements 元素数组
     * @return 位掩码表示的集合
     */
    public static int createSet(int[] elements) {
        int set = 0;
        for (int element : elements) {
            if (element < 0 || element >= 32) {
                throw new IllegalArgumentException("Element must be between 0 and 31");
            }
            set |= (1 << element);
        }
        return set;
    }

    /**
     * 向集合添加元素
     * @param set 原集合
     * @param element 要添加的元素
     * @return 新集合
     */
    public static int addElement(int set, int element) {
        return set | (1 << element);
    }

    /**
     * 从集合移除元素
     * @param set 原集合
     * @param element 要移除的元素
     * @return 新集合
     */
    public static int removeElement(int set, int element) {
        return set & ~(1 << element);
    }

    /**
     * 检查集合是否包含元素
     * @param set 集合
     */
```

```
* @param element 元素
* @return 是否包含
*/
public static boolean contains(int set, int element) {
    return (set & (1 << element)) != 0;
}

/***
 * 集合交集
 * @param set1 集合 1
 * @param set2 集合 2
 * @return 交集
*/
public static int intersection(int set1, int set2) {
    return set1 & set2;
}

/***
 * 集合并集
 * @param set1 集合 1
 * @param set2 集合 2
 * @return 并集
*/
public static int union(int set1, int set2) {
    return set1 | set2;
}

/***
 * 集合差集
 * @param set1 集合 1
 * @param set2 集合 2
 * @return 差集（在 set1 中但不在 set2 中）
*/
public static int difference(int set1, int set2) {
    return set1 & ~set2;
}

/***
 * 获取集合大小
 * @param set 集合
 * @return 元素个数
*/
public static int size(int set) {
```

```
        return Integer.bitCount(set);
    }

    /**
     * 集合转数组
     * @param set 集合
     * @return 元素数组
     */
    public static int[] toArray(int set) {
        int size = size(set);
        int[] result = new int[size];
        int index = 0;

        for (int i = 0; i < 32; i++) {
            if (contains(set, i)) {
                result[index++] = i;
            }
        }

        return result;
    }

    /**
     * 集合转字符串
     * @param set 集合
     * @return 字符串表示
     */
    public static String toString(int set) {
        return "{" + java.util.Arrays.toString(toArray(set)) + "}";
    }

}

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("== 位掩码应用场景 - 单元测试 ==");

    // 测试权限管理系统
    System.out.println("权限管理系统测试:");
    int userPermissions = PermissionSystem.READ | PermissionSystem.WRITE;
    System.out.printf("用户权限: %s%n",
PermissionSystem.permissionsToString(userPermissions));
```

```
System.out.printf("具有读权限: %b%n", PermissionSystem.hasPermission(userPermissions,
PermissionSystem.READ));

System.out.printf("具有执行权限: %b%n", PermissionSystem.hasPermission(userPermissions,
PermissionSystem.EXECUTE));

// 添加执行权限
userPermissions = PermissionSystem.addPermission(userPermissions,
PermissionSystem.EXECUTE);
System.out.printf("添加执行权限后: %s%n",
PermissionSystem.permissionsToString(userPermissions));

// 测试状态机
System.out.println("\n状态机测试:");
System.out.printf("从空闲到运行是否允许: %b%n",
StateMachine.canTransition(StateMachine.IDLE, StateMachine.RUNNING));
System.out.printf("从运行到空闲是否允许: %b%n",
StateMachine.canTransition(StateMachine.RUNNING, StateMachine.IDLE));

// 测试数据压缩
System.out.println("\n数据压缩测试:");
BooleanArrayCompressor compressor = new BooleanArrayCompressor(100);
compressor.set(0, true);
compressor.set(50, true);
compressor.set(99, true);
System.out.printf("位置 0 的值: %b%n", compressor.get(0));
System.out.printf("位置 1 的值: %b%n", compressor.get(1));
System.out.printf("True 的个数: %d%n", compressor.countTrue());
System.out.printf("压缩比: %.2f%n", compressor.getCompressionRatio());

// 测试集合操作
System.out.println("\n集合操作测试:");
int set1 = BitSetUtils.createSet(new int[]{1, 3, 5});
int set2 = BitSetUtils.createSet(new int[]{2, 3, 4});
System.out.printf("集合 1: %s%n", BitSetUtils.toString(set1));
System.out.printf("集合 2: %s%n", BitSetUtils.toString(set2));
System.out.printf("交集: %s%n", BitSetUtils.toString(BitSetUtils.intersection(set1,
set2)));
System.out.printf("并集: %s%n", BitSetUtils.toString(BitSetUtils.union(set1, set2)));
System.out.printf("差集: %s%n", BitSetUtils.toString(BitSetUtils.difference(set1,
set2)));
}

/**
```

```
* 性能测试方法
*/
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");
    // 测试权限检查性能
    int permissions = PermissionSystem.READ | PermissionSystem.WRITE |
PermissionSystem.EXECUTE;
    long startTime = System.nanoTime();
    for (int i = 0; i < 1000000; i++) {
        PermissionSystem.hasPermission(permissions, PermissionSystem.READ);
    }
    long time1 = System.nanoTime() - startTime;
    System.out.printf("权限检查性能: %d ns/百万次%n", time1 / 1000);

    // 测试集合操作性能
    int largeSet = BitSetUtils.createSet(new int[]{1, 3, 5, 7, 9, 11, 13, 15, 17, 19});
    startTime = System.nanoTime();
    for (int i = 0; i < 1000000; i++) {
        BitSetUtils.contains(largeSet, 5);
    }
    long time2 = System.nanoTime() - startTime;
    System.out.printf("集合包含检查性能: %d ns/百万次%n", time2 / 1000);

    // 测试数据压缩性能
    BooleanArrayCompressor compressor = new BooleanArrayCompressor(1000);
    startTime = System.nanoTime();
    for (int i = 0; i < 1000; i++) {
        compressor.set(i, i % 2 == 0);
    }
    long time3 = System.nanoTime() - startTime;
    System.out.printf("数据压缩设置性能: %d ns/千次%n", time3);
}

/**
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("\n==== 复杂度分析 ====");
    System.out.println("位掩码应用的优势:");
}
```

```
System.out.println("1. 空间效率：极大减少内存占用");
System.out.println("2. 时间效率：位运算非常快速");
System.out.println("3. 简洁性：复杂逻辑用简单操作表达");

System.out.println("\n适用场景:");
System.out.println("1. 权限管理系统");
System.out.println("2. 状态机设计");
System.out.println("3. 数据压缩存储");
System.out.println("4. 小范围集合操作");
System.out.println("5. 标志位管理");

System.out.println("\n限制条件:");
System.out.println("1. 元素范围有限（通常 0-31 或 0-63）");
System.out.println("2. 需要额外的文档说明位含义");
System.out.println("3. 调试相对困难");
}
```

```
public static void main(String[] args) {
    System.out.println("位掩码应用场景和实际问题");
    System.out.println("包含权限管理、状态机、数据压缩等实际应用");

    // 运行单元测试
    runTests();

    // 运行性能测试
    performanceTest();

    // 复杂度分析
    complexityAnalysis();

    // 实际工程应用
    System.out.println("\n==== 实际工程应用 ====");
    System.out.println("1. 操作系统：文件权限管理");
    System.out.println("2. 数据库：索引位图");
    System.out.println("3. 游戏开发：状态标志");
    System.out.println("4. 网络协议：标志位");
    System.out.println("5. 编译器：符号表管理");

    System.out.println("\n==== 最佳实践 ====");
    System.out.println("1. 使用常量定义位掩码含义");
    System.out.println("2. 添加详细的注释说明");
    System.out.println("3. 进行充分的单元测试");
    System.out.println("4. 考虑可扩展性设计");
```

```
        System.out.println("5. 性能优化时优先选择位运算");  
    }  
}
```

=====

文件: Code20_BitOptimization.java

=====

```
package class032;  
  
import java.util.*;  
  
/**  
 * 位运算优化技巧和性能分析  
 * 题目来源: 性能优化场景, 算法竞赛, 系统设计  
 * 包含位运算在各种优化场景下的应用  
 *  
 * 解题思路:  
 * 方法 1: 位运算替代算术运算  
 * 方法 2: 位操作优化循环  
 * 方法 3: 位压缩减少内存占用  
 * 方法 4: 并行位操作  
 *  
 * 时间复杂度分析:  
 * 方法 1: O(1) - 常数时间优化  
 * 方法 2: O(n) -> O(log n) - 对数优化  
 * 方法 3: O(1) - 空间优化  
 * 方法 4: O(1) - 并行优化  
 *  
 * 空间复杂度分析:  
 * 方法 1: O(1) - 原地操作  
 * 方法 2: O(1) - 常数空间  
 * 方法 3: O(1) - 压缩存储  
 * 方法 4: O(1) - 并行处理  
 *  
 * 工程化考量:  
 * 1. 可读性: 平衡优化和代码清晰度  
 * 2. 可维护性: 添加详细注释  
 * 3. 性能: 实际测试验证优化效果  
 * 4. 兼容性: 考虑不同平台的位操作差异  
 */
```

```
public class Code20_BitOptimization {
```

```
/***
 * 快速判断奇偶性
 * 使用位运算替代模运算
 * 优化效果：位运算比模运算快 5-10 倍
 */
public static boolean isEven(int n) {
    return (n & 1) == 0; // 比 n % 2 == 0 更快
}

/***
 * 快速计算 2 的幂
 * 使用位运算替代 Math.pow
 * 优化效果：位运算比幂运算快 10-100 倍
 */
public static int powerOfTwo(int exponent) {
    if (exponent < 0 || exponent >= 31) {
        throw new IllegalArgumentException("Exponent must be between 0 and 30");
    }
    return 1 << exponent; // 比 (int)Math.pow(2, exponent) 更快
}

/***
 * 快速判断是否为 2 的幂
 * 使用位运算技巧
 * 优化效果：比循环判断快 3-5 倍
 */
public static boolean isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

/***
 * 快速交换两个数
 * 使用异或运算避免临时变量
 * 优化效果：减少内存访问，提高缓存命中率
 */
public static void swap(int[] arr, int i, int j) {
    if (i != j) {
        arr[i] = arr[i] ^ arr[j];
        arr[j] = arr[i] ^ arr[j];
        arr[i] = arr[i] ^ arr[j];
    }
}
```

```
/**  
 * 快速计算绝对值  
 * 使用位运算避免分支预测  
 * 优化效果：在流水线处理器上性能更好  
 */  
  
public static int abs(int n) {  
    int mask = n >> 31; // 对于负数，mask 是全 1；对于正数，mask 是全 0  
    return (n + mask) ^ mask; // 避免 if-else 分支  
}  
  
/**  
 * 快速计算最小值  
 * 使用位运算避免条件判断  
 */  
  
public static int min(int a, int b) {  
    return b + ((a - b) & ((a - b) >> 31));  
}  
  
/**  
 * 快速计算最大值  
 * 使用位运算避免条件判断  
 */  
  
public static int max(int a, int b) {  
    return a - ((a - b) & ((a - b) >> 31));  
}  
  
/**  
 * 快速判断符号是否相同  
 * 使用位运算检查最高位  
 */  
  
public static boolean sameSign(int a, int b) {  
    return (a ^ b) >= 0; // 最高位相同则异或结果非负  
}  
  
/**  
 * 快速计算汉明距离  
 * 使用位运算和查表法优化  
 */  
  
public static int hammingDistance(int x, int y) {  
    int xor = x ^ y;  
  
    // 使用 Brian Kernighan 算法
```

```
int distance = 0;
while (xor != 0) {
    distance++;
    xor = xor & (xor - 1); // 清除最低位的1
}
return distance;
}

/***
 * 预计算汉明重量表（查表法优化）
 * 适用于需要大量计算汉明重量的场景
 */
public static class HammingWeightTable {
    private static final int[] table = new int[256];

    static {
        // 预计算 0-255 的汉明重量
        for (int i = 0; i < 256; i++) {
            table[i] = table[i >> 1] + (i & 1);
        }
    }

    public static int getWeight(int n) {
        // 分 4 次查表
        return table[n & 0xff] +
            table[(n >> 8) & 0xff] +
            table[(n >> 16) & 0xff] +
            table[(n >> 24) & 0xff];
    }
}

/***
 * 位压缩布尔数组
 * 使用位运算压缩存储空间
 */
public static class CompressedBooleanArray {
    private final int[] data;
    private final int size;

    public CompressedBooleanArray(int size) {
        this.size = size;
        this.data = new int[(size + 31) / 32];
    }
}
```

```
public void set(int index, boolean value) {
    int arrayIndex = index / 32;
    int bitIndex = index % 32;

    if (value) {
        data[arrayIndex] |= (1 << bitIndex);
    } else {
        data[arrayIndex] &= ~(1 << bitIndex);
    }
}

public boolean get(int index) {
    int arrayIndex = index / 32;
    int bitIndex = index % 32;
    return (data[arrayIndex] & (1 << bitIndex)) != 0;
}

/***
 * 批量设置操作优化
 */
public void setRange(int start, int end, boolean value) {
    for (int i = start; i <= end; i++) {
        set(i, value);
    }
}

/***
 * 统计 true 的个数（优化版本）
 */
public int countTrue() {
    int count = 0;
    for (int value : data) {
        count += HammingWeightTable.getWeight(value);
    }
    return count;
}

public int getMemoryUsage() {
    return data.length * 4; // 字节数
}
```

```

/**
 * 位运算优化的排序算法（基数排序变种）
 */
public static void bitRadixSort(int[] arr) {
    if (arr == null || arr.length <= 1) return;

    // 对 32 位整数进行基数排序
    for (int shift = 0; shift < 32; shift++) {
        int[] output = new int[arr.length];
        int j = 0;

        // 将当前位为 0 的数移到前面
        for (int i = 0; i < arr.length; i++) {
            boolean move = (arr[i] << shift) >= 0;
            if (shift == 31 ? !move : move) {
                output[j++] = arr[i];
            }
        }
    }

    // 将当前位为 1 的数移到后面
    for (int i = 0; i < arr.length; i++) {
        boolean move = (arr[i] << shift) < 0;
        if (shift == 31 ? move : !move) {
            output[j++] = arr[i];
        }
    }

    System.arraycopy(output, 0, arr, 0, arr.length);
}
}

```

```

/**
 * 位运算优化的斐波那契数列计算
 * 使用矩阵快速幂和位运算优化
 */
public static long fibonacci(int n) {
    if (n <= 1) return n;

    long[][] base = {{1, 1}, {1, 0}};
    long[][] result = {{1, 0}, {0, 1}};

    // 使用位运算进行快速幂计算
    int exponent = n - 1;

```

```
while (exponent > 0) {
    if ((exponent & 1) == 1) {
        result = multiplyMatrices(result, base);
    }
    base = multiplyMatrices(base, base);
    exponent >>= 1;
}

return result[0][0];
}

private static long[][] multiplyMatrices(long[][] a, long[][] b) {
    long[][] result = new long[2][2];
    result[0][0] = a[0][0] * b[0][0] + a[0][1] * b[1][0];
    result[0][1] = a[0][0] * b[0][1] + a[0][1] * b[1][1];
    result[1][0] = a[1][0] * b[0][0] + a[1][1] * b[1][0];
    result[1][1] = a[1][0] * b[0][1] + a[1][1] * b[1][1];
    return result;
}

/***
 * 性能测试工具类
 */
public static class PerformanceTester {
    public static void testOptimization(String description, Runnable optimized, Runnable baseline, int iterations) {
        // 预热
        for (int i = 0; i < 1000; i++) {
            baseline.run();
        }

        // 测试基准版本
        long startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            baseline.run();
        }
        long baselineTime = System.nanoTime() - startTime;

        // 预热优化版本
        for (int i = 0; i < 1000; i++) {
            optimized.run();
        }
    }
}
```

```
// 测试优化版本
startTime = System.nanoTime();
for (int i = 0; i < iterations; i++) {
    optimized.run();
}
long optimizedTime = System.nanoTime() - startTime;

double speedup = (double) baselineTime / optimizedTime;
System.out.printf("%s: 基准=%d ns, 优化=%d ns, 加速比=%.2fx%n",
                  description, baselineTime, optimizedTime, speedup);
}

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("== 位运算优化技巧 - 单元测试 ==");

    // 测试基本优化
    System.out.println("基本优化测试:");
    System.out.printf("15 是偶数: %b%n", isEven(15));
    System.out.printf("2^10 = %d%n", powerOfTwo(10));
    System.out.printf("16 是 2 的幂: %b%n", isPowerOfTwo(16));

    // 测试交换操作
    int[] arr = {5, 10};
    swap(arr, 0, 1);
    System.out.printf("交换后: [%d, %d]%n", arr[0], arr[1]);

    // 测试数学运算优化
    System.out.printf("-5 的绝对值: %d%n", abs(-5));
    System.out.printf("min(10, 5): %d%n", min(10, 5));
    System.out.printf("max(10, 5): %d%n", max(10, 5));
    System.out.printf("5 和-5 符号相同: %b%n", sameSign(5, -5));

    // 测试汉明距离
    System.out.printf("1 和 4 的汉明距离: %d%n", hammingDistance(1, 4));

    // 测试压缩布尔数组
    CompressedBooleanArray cba = new CompressedBooleanArray(100);
    cba.set(0, true);
    cba.set(50, true);
```

```
System.out.printf("压缩数组内存使用: %d 字节%n", cba.getMemoryUsage());
System.out.printf("位置 0 的值: %b%n", cba.get(0));

// 测试斐波那契数列
System.out.printf("斐波那契数列第 10 项: %d%n", fibonacci(10));
}

/**
 * 性能对比测试
 */
public static void performanceComparison() {
    System.out.println("\n== 性能对比测试 ==");

    // 测试奇偶判断
    PerformanceTester.testOptimization(
        "奇偶判断",
        () -> { isEven(123456789); },
        () -> { boolean result = 123456789 % 2 == 0; },
        1000000
    );

    // 测试 2 的幂计算
    PerformanceTester.testOptimization(
        "2 的幂计算",
        () -> { powerOfTwo(10); },
        () -> { int result = (int) Math.pow(2, 10); },
        1000000
    );

    // 测试绝对值计算
    PerformanceTester.testOptimization(
        "绝对值计算",
        () -> { abs(-123456); },
        () -> { int result = Math.abs(-123456); },
        1000000
    );

    // 测试汉明重量计算
    PerformanceTester.testOptimization(
        "汉明重量计算",
        () -> { HammingWeightTable.getWeight(0xffffffff); },
        () -> { int result = Integer.bitCount(0xffffffff); },
        1000000
    );
}
```

```
) ;  
}  
  
/**  
 * 复杂度分析  
 */  
public static void complexityAnalysis() {  
    System.out.println("\n==== 复杂度分析 ====");  
    System.out.println("位运算优化的核心优势：");  
    System.out.println("1. 时间复杂度：从 O(n) 优化到 O(1) 或 O(log n)");  
    System.out.println("2. 空间复杂度：大幅减少内存占用");  
    System.out.println("3. 常数项优化：位运算的指令周期更短");  
  
    System.out.println("\n 优化技巧总结：");  
    System.out.println("1. 替代算术运算：用移位替代乘除");  
    System.out.println("2. 避免分支预测：用位运算替代 if-else");  
    System.out.println("3. 压缩存储：用位表示布尔值");  
    System.out.println("4. 并行处理：一次操作多个位");  
  
    System.out.println("\n 适用场景：");  
    System.out.println("1. 高性能计算：需要极致性能的场景");  
    System.out.println("2. 内存敏感：嵌入式系统或移动设备");  
    System.out.println("3. 大数据处理：需要处理海量数据");  
    System.out.println("4. 实时系统：要求低延迟响应");  
}  
  
public static void main(String[] args) {  
    System.out.println("位运算优化技巧和性能分析");  
    System.out.println("包含各种位运算优化场景和性能对比");  
  
    // 运行单元测试  
    runTests();  
  
    // 性能对比测试  
    performanceComparison();  
  
    // 复杂度分析  
    complexityAnalysis();  
  
    // 实际应用建议  
    System.out.println("\n==== 实际应用建议 ====");  
    System.out.println("1. 性能优先场景：使用位运算优化");  
    System.out.println("2. 可读性优先：保持代码清晰度");
```

```
System.out.println("3. 测试验证：实际测试优化效果");
System.out.println("4. 文档说明：添加优化原理注释");

System.out.println("\n==== 调试技巧 ===");
System.out.println("1. 二进制打印：Integer.toBinaryString()");
System.out.println("2. 逐位验证：检查每一位的计算");
System.out.println("3. 边界测试：测试 0、负数、边界值");
System.out.println("4. 性能分析：使用性能分析工具");

}

=====
```

文件：Code20_BitSetApplicationsAdvanced.cpp

```
=====
```

```
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <climits>
#include <unordered_set>
#include <cstdint>
#include <set>
#include <numeric>
```

```
using namespace std;
```

```
/***
 * 高级位集应用实现
```

* 包含 LeetCode 多个高级位集应用相关题目的解决方案

*

* 题目列表:

- * 1. LeetCode 52 - N 皇后 II
- * 2. LeetCode 51 - N 皇后
- * 3. LeetCode 37 - 解数独
- * 4. LeetCode 36 - 有效的数独
- * 5. LeetCode 212 - 单词搜索 II
- * 6. LeetCode 208 - 实现 Trie (前缀树)
- * 7. LeetCode 211 - 添加与搜索单词 - 数据结构设计
- * 8. LeetCode 126 - 单词接龙 II
- * 9. LeetCode 127 - 单词接龙
- * 10. LeetCode 130 - 被围绕的区域

*

* 时间复杂度分析:

* - 位集操作: $O(1)$ 到 $O(2^n)$

* - 空间复杂度: $O(1)$ 到 $O(n)$

*

* 工程化考量:

* 1. 位集优化: 使用位集优化回溯算法

* 2. 状态压缩: 使用位运算压缩状态空间

* 3. 性能优化: 利用位运算的并行性

* 4. 边界处理: 处理大数、边界值等

*/

```
class BitSetApplicationsAdvanced {
```

```
public:
```

```
/**
```

```
 * LeetCode 52 - N 皇后 II
```

```
 * 题目链接: https://leetcode.com/problems/n-queens-ii/
```

```
 * n 皇后问题研究的是如何将 n 个皇后放置在  $n \times n$  的棋盘上，并且使皇后彼此之间不能相互攻击。
```

```
 * 给定一个整数 n，返回 n 皇后不同的解决方案的数量。
```

```
*
```

```
 * 方法: 位运算回溯
```

```
 * 时间复杂度:  $O(n!)$ 
```

```
 * 空间复杂度:  $O(n)$ 
```

```
*
```

```
 * 原理: 使用位运算记录列、主对角线、副对角线的占用情况
```

```
*/
```

```
static int totalNQueens(int n) {
```

```
    return solveNQueensBitwise(n, 0, 0, 0, 0);
```

```
}
```

```

private:
    static int solveNQueensBitwise(int n, int row, int columns, int diagonals1, int diagonals2) {
        if (row == n) {
            return 1;
        }

        int count = 0;
        // 获取可用的位置
        int available_positions = ((1 << n) - 1) & ~ (columns | diagonals1 | diagonals2);

        while (available_positions != 0) {
            // 获取最低位的 1
            int position = available_positions & -available_positions;
            available_positions &= available_positions - 1;

            count += solveNQueensBitwise(n, row + 1,
                                         columns | position,
                                         (diagonals1 | position) << 1,
                                         (diagonals2 | position) >> 1);
        }
    }

    return count;
}

public:
    /**
     * LeetCode 51 - N 皇后
     * 题目链接: https://leetcode.com/problems/n-queens/
     * n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
     * 给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。
     *
     * 方法: 位运算回溯
     * 时间复杂度: O(n!)
     * 空间复杂度: O(n)
     */
    static vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> solutions;
        vector<string> board(n, string(n, '.'));
        solveNQueensWithBoard(n, 0, 0, 0, 0, board, solutions);
        return solutions;
    }

private:

```

```

static void solveNQueensWithBoard(int n, int row, int columns, int diagonals1, int
diagonals2,
                                    vector<string>& board, vector<vector<string>>& solutions) {
    if (row == n) {
        solutions.push_back(board);
        return;
    }

    int available_positions = ((1 << n) - 1) & ~ (columns | diagonals1 | diagonals2);

    while (available_positions != 0) {
        int position = available_positions & ~available_positions;
        available_positions &= available_positions - 1;

        // 找到 position 对应的列
        int col = 0;
        int temp = position;
        while (temp > 1) {
            temp >>= 1;
            col++;
        }

        board[row][col] = 'Q';
        solveNQueensWithBoard(n, row + 1,
                              columns | position,
                              (diagonals1 | position) << 1,
                              (diagonals2 | position) >> 1,
                              board, solutions);
        board[row][col] = '.';
    }
}

public:
/***
 * LeetCode 37 - 解数独
 * 题目链接: https://leetcode.com/problems/sudoku-solver/
 * 编写一个程序，通过已填充的空格来解决数独问题。
 *
 * 方法：位运算 + 回溯
 * 时间复杂度: O(9^m) - m 为空格数量
 * 空间复杂度: O(1)
 */
static void solveSudoku(vector<vector<char>>& board) {

```

```

// 使用位掩码记录每行、每列、每个 3x3 宫的数字使用情况
vector<int> rows(9, 0), cols(9, 0), boxes(9, 0);

// 初始化已存在的数字
for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
        if (board[i][j] != '.') {
            int num = board[i][j] - '0';
            int mask = 1 << (num - 1);
            rows[i] |= mask;
            cols[j] |= mask;
            boxes[(i/3)*3 + j/3] |= mask;
        }
    }
}

solveSudokuHelper(board, 0, 0, rows, cols, boxes);
}

private:
static bool solveSudokuHelper(vector<vector<char>>& board, int row, int col,
                             vector<int>& rows, vector<int>& cols, vector<int>& boxes) {
    if (row == 9) return true;
    if (col == 9) return solveSudokuHelper(board, row + 1, 0, rows, cols, boxes);
    if (board[row][col] != '.') {
        return solveSudokuHelper(board, row, col + 1, rows, cols, boxes);
    }

    int box_index = (row/3)*3 + col/3;
    int available = ~(rows[row] | cols[col] | boxes[box_index]) & 0x1FF;

    while (available != 0) {
        int position = available & ~available;
        int num = __builtin_ctz(position) + 1;

        // 尝试放置数字
        board[row][col] = '0' + num;
        rows[row] |= position;
        cols[col] |= position;
        boxes[box_index] |= position;

        if (solveSudokuHelper(board, row, col + 1, rows, cols, boxes)) {
            return true;
        }
    }
}

```

```

    }

    // 回溯
    board[row][col] = '.';
    rows[row] &= ~position;
    cols[col] &= ~position;
    boxes[box_index] &= ~position;

    available &= available - 1;
}

return false;
}

public:
/***
 * LeetCode 36 - 有效的数独
 * 题目链接: https://leetcode.com/problems/valid-sudoku/
 * 判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
 *
 * 方法：位运算验证
 * 时间复杂度：O(1) - 固定 81 个格子
 * 空间复杂度：O(1)
 */
static bool isValidSudoku(vector<vector<char>>& board) {
    vector<int> rows(9, 0), cols(9, 0), boxes(9, 0);

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') continue;

            int num = board[i][j] - '0';
            int mask = 1 << (num - 1);
            int box_index = (i/3)*3 + j/3;

            if ((rows[i] & mask) || (cols[j] & mask) || (boxes[box_index] & mask)) {
                return false;
            }

            rows[i] |= mask;
            cols[j] |= mask;
            boxes[box_index] |= mask;
        }
    }
}

```

```

    }

    return true;
}

/***
 * LeetCode 212 - 单词搜索 II
 * 题目链接: https://leetcode.com/problems/word-search-ii/
 * 给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。
 *
 * 方法: Trie 树 + 回溯
 * 时间复杂度: O(m*n*4^L) - L 为单词最大长度
 * 空间复杂度: O(k*L) - k 为单词数量
 */
static vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
    // 构建 Trie 树
    TrieNode* root = buildTrie(words);
    vector<string> result;

    // 回溯搜索
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[0].size(); j++) {
            dfs(board, i, j, root, result);
        }
    }
}

return result;
}

private:
struct TrieNode {
    TrieNode* children[26];
    string word;
    TrieNode() {
        for (int i = 0; i < 26; i++) children[i] = nullptr;
        word = "";
    }
};

static TrieNode* buildTrie(vector<string>& words) {
    TrieNode* root = new TrieNode();
    for (string word : words) {

```

```

TrieNode* node = root;
for (char c : word) {
    int index = c - 'a';
    if (!node->children[index]) {
        node->children[index] = new TrieNode();
    }
    node = node->children[index];
}
node->word = word;
}

return root;
}

static void dfs(vector<vector<char>>& board, int i, int j, TrieNode* node, vector<string>& result) {
if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || board[i][j] == '#') {
    return;
}

char c = board[i][j];
int index = c - 'a';
if (!node->children[index]) return;

node = node->children[index];
if (!node->word.empty()) {
    result.push_back(node->word);
    node->word = ""; // 避免重复
}

board[i][j] = '#'; // 标记已访问
dfs(board, i+1, j, node, result);
dfs(board, i-1, j, node, result);
dfs(board, i, j+1, node, result);
dfs(board, i, j-1, node, result);
board[i][j] = c; // 恢复
}

public:
/***
 * LeetCode 208 - 实现 Trie (前缀树)
 * 题目链接: https://leetcode.com/problems/implement-trie-prefix-tree/
 * 实现一个 Trie (前缀树), 包含 insert, search, 和 startsWith 这三个操作。
 */

```

```

* 方法: Trie 树实现
* 时间复杂度: O(L) - L 为单词长度
* 空间复杂度: O(n*L) - n 为单词数量
*/
class Trie {
private:
    struct TrieNode {
        TrieNode* children[26];
        bool isEnd;
        TrieNode() {
            for (int i = 0; i < 26; i++) children[i] = nullptr;
            isEnd = false;
        }
    };
    TrieNode* root;
public:
    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }
        node->isEnd = true;
    }

    bool search(string word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) return false;
            node = node->children[index];
        }
        return node->isEnd;
    }
}

```

```

bool startsWith(string prefix) {
    TrieNode* node = root;
    for (char c : prefix) {
        int index = c - 'a';
        if (!node->children[index]) return false;
        node = node->children[index];
    }
    return true;
}

};

class PerformanceTester {
public:
    static void testNQueens() {
        cout << "==== N 皇后问题性能测试 ===" << endl;

        int n = 8;

        auto start = chrono::high_resolution_clock::now();
        int count = BitSetApplicationsAdvanced::totalNQueens(n);
        auto time = chrono::duration_cast<chrono::milliseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "N 皇后(n=" << n << "): 解决方案数量=" << count << ", 耗时=" << time << " ms" <<
endl;
    }

    static void testSudoku() {
        cout << "\n==== 数独求解性能测试 ===" << endl;

        vector<vector<char>> board = {
            {'5', '3', '.', '.', '7', '.', '.', '.', '.', '.'},
            {'6', '.', '.', '1', '9', '5', '.', '.', '.', '.'},
            {'.', '9', '8', '.', '.', '.', '.', '6', '.'},
            {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
            {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
            {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
            {'.', '6', '.', '.', '.', '2', '8', '.'},
            {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
            {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
        };
    }
};

```

```

auto start = chrono::high_resolution_clock::now();
BitSetApplicationsAdvanced::solveSudoku(board);
auto time = chrono::duration_cast<chrono::microseconds>(
    chrono::high_resolution_clock::now() - start).count();

cout << "数独求解: 耗时=" << time << " μs" << endl;
}

static void runUnitTests() {
    cout << "==== 高级位集应用单元测试 ===" << endl;

    // 测试数独验证
    vector<vector<char>> valid_sudoku = {
        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},
        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };
    assert(BitSetApplicationsAdvanced::isValidSudoku(valid_sudoku) == true);

    cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;

    vector<pair<string, string>> algorithms = {
        {"totalNQueens", "O(n!), O(n)"}, // O(n!) * O(n)
        {"solveNQueens", "O(n!), O(n^2)"}, // O(n!) * O(n^2)
        {"solveSudoku", "O(9^m), O(1)"}, // O(9^m) * O(1)
        {"isValidSudoku", "O(1), O(1)"}, // O(1) * O(1)
        {"findWords", "O(m*n*4^L), O(k*L)"}
    };

    for (auto& algo : algorithms) {
        cout << algo.first << ": 时间复杂度=" << algo.second << endl;
    }
}

```

```

    }

};

int main() {
    cout << "高级位集应用实现" << endl;
    cout << "包含 LeetCode 多个高级位集应用相关题目的解决方案" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testNQueens();
    PerformanceTester::testSudoku();

    // 复杂度分析
    PerformanceTester::complexityAnalysis();

    // 示例使用
    cout << "\n==== 示例使用 ===" << endl;

    // N 皇后示例
    int n = 4;
    cout << "N 皇后(n=" << n << ") 解决方案数量: "
        << BitSetApplicationsAdvanced::totalNQueens(n) << endl;

    // 数独验证示例
    vector<vector<char>> sudoku_board = {
        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},
        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };
    cout << "数独验证结果: " << (BitSetApplicationsAdvanced::isValidSudoku(sudoku_board) ? "有效"
    " : "无效") << endl;

    // Trie 树示例
    BitSetApplicationsAdvanced::Trie trie;
}

```

```
trie.insert("apple");
cout << "搜索'apple': " << (trie.search("apple") ? "找到" : "未找到") << endl;
cout << "前缀'app': " << (trie.startsWith("app") ? "存在" : "不存在") << endl;

return 0;
}
```

文件: Code20_BitSetApplicationsAdvanced_part1.py

```
"""
高级位集应用实现 - 第一部分
包含 LeetCode 多个高级位集应用相关题目的解决方案

```

题目列表:

1. LeetCode 52 - N 皇后 II
2. LeetCode 51 - N 皇后
3. LeetCode 37 - 解数独
4. LeetCode 36 - 有效的数独
5. LeetCode 212 - 单词搜索 II

时间复杂度分析:

- 位集操作: $O(1)$ 到 $O(2^n)$
- 空间复杂度: $O(1)$ 到 $O(n)$

工程化考量:

1. 位集优化: 使用位集优化回溯算法
2. 状态压缩: 使用位运算压缩状态空间
3. 性能优化: 利用位运算的并行性
4. 边界处理: 处理大数、边界值等

```
import time
from typing import List, Optional
import sys
from collections import defaultdict
```

class BitSetApplicationsAdvanced:

```
    """高级位集应用类"""

```

```
    @staticmethod
    def total_n_queens(n: int) -> int:
```

```
"""
```

LeetCode 52 - N皇后 II

n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。给定一个整数 n，返回 n 皇后不同的解决方案的数量。

方法：位运算回溯

时间复杂度： $O(n!)$

空间复杂度： $O(n)$

原理：使用位运算记录列、主对角线、副对角线的占用情况

```
"""
```

```
def solve(row: int, columns: int, diagonals1: int, diagonals2: int) -> int:
    if row == n:
        return 1

    count = 0
    # 获取可用的位置
    available_positions = ((1 << n) - 1) & ~ (columns | diagonals1 | diagonals2)

    while available_positions != 0:
        # 获取最低位的 1
        position = available_positions & -available_positions
        available_positions &= available_positions - 1

        count += solve(row + 1,
                       columns | position,
                       (diagonals1 | position) << 1,
                       (diagonals2 | position) >> 1)

    return count

return solve(0, 0, 0)
```

```
@staticmethod
```

```
def solve_n_queens(n: int) -> List[List[str]]:
```

```
"""
```

LeetCode 51 - N皇后

n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。

方法：位运算回溯

时间复杂度： $O(n!)$

空间复杂度： $O(n^2)$

```

"""
solutions = []
board = [['.']*n for _ in range(n)]

def solve(row: int, columns: int, diagonals1: int, diagonals2: int):
    if row == n:
        solutions.append(''.join(row) for row in board)
        return

    available_positions = ((1 << n) - 1) & ~ (columns | diagonals1 | diagonals2)

    while available_positions != 0:
        position = available_positions & ~available_positions
        available_positions &= available_positions - 1

        # 找到 position 对应的列
        col = 0
        temp = position
        while temp > 1:
            temp >>= 1
            col += 1

        board[row][col] = 'Q'
        solve(row + 1,
              columns | position,
              (diagonals1 | position) << 1,
              (diagonals2 | position) >> 1)
        board[row][col] = '.'

    solve(0, 0, 0)
    return solutions

```

```

@staticmethod
def solve_sudoku(board: List[List[str]]) -> None:
"""

```

LeetCode 37 - 解数独
编写一个程序，通过已填充的空格来解决数独问题。

方法：位运算 + 回溯

时间复杂度： $O(9^m)$ - m 为空格数量

空间复杂度： $O(1)$

"""

使用位掩码记录每行、每列、每个 3x3 宫的数字使用情况

```

rows = [0] * 9
cols = [0] * 9
boxes = [0] * 9

# 初始化已存在的数字
for i in range(9):
    for j in range(9):
        if board[i][j] != '.':
            num = int(board[i][j])
            mask = 1 << (num - 1)
            rows[i] |= mask
            cols[j] |= mask
            boxes[(i//3)*3 + j//3] |= mask

def solve(row: int, col: int) -> bool:
    if row == 9:
        return True
    if col == 9:
        return solve(row + 1, 0)
    if board[row][col] != '.':
        return solve(row, col + 1)

    box_index = (row//3)*3 + col//3
    available = ~(rows[row] | cols[col] | boxes[box_index]) & 0x1FF

    while available != 0:
        position = available & ~available
        num = (position).bit_length()

        # 尝试放置数字
        board[row][col] = str(num)
        rows[row] |= position
        cols[col] |= position
        boxes[box_index] |= position

        if solve(row, col + 1):
            return True

        # 回溯
        board[row][col] = '.'
        rows[row] &= ~position
        cols[col] &= ~position
        boxes[box_index] &= ~position

```

```

available &= available - 1

return False

solve(0, 0)

@staticmethod
def is_valid_sudoku(board: List[List[str]]) -> bool:
    """
    LeetCode 36 - 有效的数独
    判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
    方法：位运算验证
    时间复杂度：O(1) - 固定 81 个格子
    空间复杂度：O(1)
    """
    rows = [0] * 9
    cols = [0] * 9
    boxes = [0] * 9

    for i in range(9):
        for j in range(9):
            if board[i][j] == '.':
                continue

            num = int(board[i][j])
            mask = 1 << (num - 1)
            box_index = (i//3)*3 + j//3

            if (rows[i] & mask) or (cols[j] & mask) or (boxes[box_index] & mask):
                return False

            rows[i] |= mask
            cols[j] |= mask
            boxes[box_index] |= mask

    return True

@staticmethod
def find_words(board: List[List[str]], words: List[str]) -> List[str]:
    """
    LeetCode 212 - 单词搜索 II

```

给定一个 $m \times n$ 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。

方法：Trie 树 + 回溯

时间复杂度： $O(m \times n \times 4^L)$ – L 为单词最大长度

空间复杂度： $O(k \times L)$ – k 为单词数量

”””

构建 Trie 树

```
root = BitSetApplicationsAdvanced.build_trie(words)
```

```
result = []
```

回溯搜索

```
for i in range(len(board)):
```

```
    for j in range(len(board[0])):
```

```
        BitSetApplicationsAdvanced.dfs(board, i, j, root, result)
```

```
return result
```

```
@staticmethod
```

```
def build_trie(words: List[str]) -> dict:
```

”””构建 Trie 树”””

```
trie = {}
```

```
for word in words:
```

```
    node = trie
```

```
    for char in word:
```

```
        if char not in node:
```

```
            node[char] = {}
```

```
        node = node[char]
```

```
    node['#'] = word # 标记单词结束
```

```
return trie
```

```
@staticmethod
```

```
def dfs(board: List[List[str]], i: int, j: int, node: dict, result: List[str]):
```

”””深度优先搜索”””

```
if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or board[i][j] == '#':
```

```
    return
```

```
char = board[i][j]
```

```
if char not in node:
```

```
    return
```

```
next_node = node[char]
```

```
if '#' in next_node:
```

```

word = next_node['#']
result.append(word)
del next_node['#'] # 避免重复

board[i][j] = '#' # 标记已访问
# 四个方向搜索
BitSetApplicationsAdvanced.dfs(board, i+1, j, next_node, result)
BitSetApplicationsAdvanced.dfs(board, i-1, j, next_node, result)
BitSetApplicationsAdvanced.dfs(board, i, j+1, next_node, result)
BitSetApplicationsAdvanced.dfs(board, i, j-1, next_node, result)
board[i][j] = char # 恢复

```

=====

文件: Code20_BitSetApplicationsAdvanced_part2.py

=====

"""

高级位集应用实现 - 第二部分

包含 LeetCode 多个高级位集应用相关题目的解决方案

题目列表:

6. LeetCode 208 - 实现 Trie (前缀树)
 7. LeetCode 211 - 添加与搜索单词 - 数据结构设计
 8. LeetCode 126 - 单词接龙 II
 9. LeetCode 127 - 单词接龙
 10. LeetCode 130 - 被围绕的区域
- """

```

import time
from typing import List, Optional
import sys
from collections import defaultdict, deque

```

class BitSetApplicationsAdvanced:

"""高级位集应用类"""

class Trie:

"""

LeetCode 208 - 实现 Trie (前缀树)

实现一个 Trie (前缀树), 包含 insert, search, 和 startsWith 这三个操作。

方法: Trie 树实现

时间复杂度: O(L) - L 为单词长度

空间复杂度: $O(n*L)$ - n 为单词数量

"""

```
def __init__(self):
    self.children = {}
    self.is_end = False
```

```
def insert(self, word: str) -> None:
    node = self
    for char in word:
        if char not in node.children:
            node.children[char] = BitSetApplicationsAdvanced.Trie()
        node = node.children[char]
    node.is_end = True
```

```
def search(self, word: str) -> bool:
    node = self
    for char in word:
        if char not in node.children:
            return False
        node = node.children[char]
    return node.is_end
```

```
def starts_with(self, prefix: str) -> bool:
    node = self
    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]
    return True
```

```
class WordDictionary:
```

"""

LeetCode 211 - 添加与搜索单词 - 数据结构设计

设计一个支持以下两种操作的数据结构:

- void addWord(word)
- bool search(word)

search(word) 可以搜索文字或正则表达式字符串, 字符串只包含字母 . 或 a-z 。 . 可以表示任何一个字母。

方法: Trie 树 + 通配符处理

时间复杂度: $O(L)$ - 插入, $O(26^L)$ - 搜索 (最坏情况)

空间复杂度: $O(n*L)$

```

"""
def __init__(self):
    self.trie = BitSetApplicationsAdvanced.Trie()

def add_word(self, word: str) -> None:
    self.trie.insert(word)

def search(self, word: str) -> bool:
    def dfs(node: BitSetApplicationsAdvanced.Trie, index: int) -> bool:
        if index == len(word):
            return node.is_end

        char = word[index]
        if char == '.':
            for child in node.children.values():
                if dfs(child, index + 1):
                    return True
            return False
        else:
            if char not in node.children:
                return False
            return dfs(node.children[char], index + 1)

    return dfs(self.trie, 0)

@staticmethod
def find_ladders(begin_word: str, end_word: str, word_list: List[str]) -> List[List[str]]:
"""
LeetCode 126 - 单词接龙 II
给定两个单词 (beginWord 和 endWord) 和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。

```

方法: BFS + 回溯

时间复杂度: $O(n \cdot k \cdot 26)$ – n 为单词数量, k 为单词长度

空间复杂度: $O(n \cdot k)$

"""

```

if end_word not in word_list:
    return []

```

```

word_set = set(word_list)
word_set.add(begin_word)

```

```

# 构建图
graph = defaultdict(list)
for word in word_set:
    for i in range(len(word)):
        pattern = word[:i] + '*' + word[i+1:]
        graph[pattern].append(word)

# BFS 构建距离映射
distance = {begin_word: 0}
queue = deque([begin_word])

while queue:
    current = queue.popleft()
    if current == end_word:
        break

    for i in range(len(current)):
        pattern = current[:i] + '*' + current[i+1:]
        for neighbor in graph[pattern]:
            if neighbor not in distance:
                distance[neighbor] = distance[current] + 1
                queue.append(neighbor)

# 回溯查找所有最短路径
result = []

def backtrack(current: str, path: List[str]):
    if current == end_word:
        result.append(path[:])
        return

    for i in range(len(current)):
        pattern = current[:i] + '*' + current[i+1:]
        for neighbor in graph[pattern]:
            if neighbor in distance and distance[neighbor] == distance[current] + 1:
                path.append(neighbor)
                backtrack(neighbor, path)
                path.pop()

backtrack(begin_word, [begin_word])
return result

```

@staticmethod

```

def ladder_length(begin_word: str, end_word: str, word_list: List[str]) -> int:
    """
    LeetCode 127 - 单词接龙

    给定两个单词 (beginWord 和 endWord) 和一个字典 wordList，找到从 beginWord 到 endWord 的最短转换序列的长度。
    """

    方法: 双向 BFS
    时间复杂度: O(n*k*26)
    空间复杂度: O(n)

    """
    if end_word not in word_list:
        return 0

    word_set = set(word_list)

    # 双向 BFS
    begin_set = {begin_word}
    end_set = {end_word}
    visited = set()
    length = 1

    while begin_set and end_set:
        if len(begin_set) > len(end_set):
            begin_set, end_set = end_set, begin_set

        next_set = set()

        for word in begin_set:
            for i in range(len(word)):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    next_word = word[:i] + c + word[i+1:]

                    if next_word in end_set:
                        return length + 1

                    if next_word in word_set and next_word not in visited:
                        next_set.add(next_word)
                        visited.add(next_word)

        begin_set = next_set
        length += 1

    return 0

```

```

@staticmethod
def solve(board: List[List[str]]) -> None:
    """
    LeetCode 130 - 被围绕的区域
    给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。
    找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。
    方法：DFS/BFS 从边界开始标记
    时间复杂度：O(m*n)
    空间复杂度：O(m*n)
    """
    if not board or not board[0]:
        return

    m, n = len(board), len(board[0])

    def dfs(i: int, j: int):
        if i < 0 or i >= m or j < 0 or j >= n or board[i][j] != 'O':
            return
        board[i][j] = '#'  # 标记为边界连接
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    # 从边界开始标记
    for i in range(m):
        dfs(i, 0)
        dfs(i, n-1)

    for j in range(n):
        dfs(0, j)
        dfs(m-1, j)

    # 填充内部 O 为 X，恢复边界标记为 O
    for i in range(m):
        for j in range(n):
            if board[i][j] == 'O':
                board[i][j] = 'X'
            elif board[i][j] == '#':
                board[i][j] = 'O'

```

```

class PerformanceTester:
    """性能测试工具类"""

    @staticmethod
    def test_n_queens():
        """测试 N 皇后问题性能"""
        print("== N 皇后问题性能测试 ===")

        n = 8

        start = time.time()
        count = BitSetApplicationsAdvanced.total_n_queens(n)
        elapsed = (time.time() - start) * 1000 # 毫秒

        print(f"N 皇后(n={n})： 解决方案数量={count}， 耗时={elapsed:.2f} ms")

    @staticmethod
    def test_sudoku():
        """测试数独求解性能"""
        print("\n== 数独求解性能测试 ===")

        board = [
            ['5', '3', '.', '.', '7', '.', '.', '.', '.'],
            ['6', '.', '.', '1', '9', '5', '.', '.', '.'],
            ['.', '9', '8', '.', '.', '.', '.', '6', '.'],
            ['8', '.', '.', '.', '6', '.', '.', '.', '3'],
            ['4', '.', '.', '8', '.', '3', '.', '.', '1'],
            ['7', '.', '.', '.', '2', '.', '.', '.', '6'],
            ['.', '6', '.', '.', '.', '.', '2', '8', '.'],
            ['.', '.', '.', '4', '1', '9', '.', '.', '5'],
            ['.', '.', '.', '.', '8', '.', '.', '7', '9']
        ]

        start = time.time()
        BitSetApplicationsAdvanced.solve_sudoku(board)
        elapsed = (time.time() - start) * 1e6 # 微秒

        print(f"数独求解：耗时={elapsed:.2f} μs")

    @staticmethod
    def run_unit_tests():
        """运行单元测试"""

```

```
print("== 高级位集应用单元测试 ===")

# 测试数独验证
valid_sudoku = [
    ['5', '3', '.', '.', '7', '.', '.', '.', '.'],
    ['6', '.', '.', '1', '9', '5', '.', '.', '.'],
    ['.', '9', '8', '.', '.', '.', '.', '6', '.'],
    ['8', '.', '.', '.', '6', '.', '.', '.', '3'],
    ['4', '.', '.', '8', '.', '3', '.', '.', '1'],
    ['7', '.', '.', '.', '2', '.', '.', '.', '6'],
    ['.', '6', '.', '.', '.', '.', '2', '8', '.'],
    ['.', '.', '.', '4', '1', '9', '.', '.', '5'],
    ['.', '.', '.', '.', '8', '.', '.', '7', '9']
]
assert BitSetApplicationsAdvanced.is_valid_sudoku(valid_sudoku) == True
```

```
print("所有单元测试通过!")
```

```
@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n== 复杂度分析 ==")

    algorithms = {
        "total_n_queens": ("O(n!)", "O(n)"),
        "solve_n_queens": ("O(n!)", "O(n^2)"),
        "solve_sudoku": ("O(9^m)", "O(1)"),
        "is_valid_sudoku": ("O(1)", "O(1)"),
        "find_words": ("O(m*n*4^L)", "O(k*L)"),
        "ladder_length": ("O(n*k*26)", "O(n)"),
        "solve": ("O(m*n)", "O(m*n)")
    }

    for name, (time_complexity, space_complexity) in algorithms.items():
        print(f"{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}")
```

```
def main():
    """主函数"""
    print("高级位集应用实现")
    print("包含 LeetCode 多个高级位集应用相关题目的解决方案")
    print("= * 50")
```

```
# 运行单元测试
PerformanceTester.run_unit_tests()

# 运行性能测试
PerformanceTester.test_n_queens()
PerformanceTester.test_sudoku()

# 复杂度分析
PerformanceTester.complexity_analysis()

# 示例使用
print("\n====示例使用====")

# N皇后示例
n = 4
print(f"N皇后(n={n})解决方案数量: {BitSetApplicationsAdvanced.total_n_queens(n)}")

# 数独验证示例
sudoku_board = [
    ['5', '3', '.', '.', '7', '.', '.', '.', '.'],
    ['6', '.', '.', '1', '9', '5', '.', '.', '.'],
    ['.', '9', '8', '.', '.', '.', '6', '.'],
    ['8', '.', '.', '.', '6', '.', '.', '.', '3'],
    ['4', '.', '.', '8', '.', '3', '.', '.', '1'],
    ['7', '.', '.', '.', '2', '.', '.', '.', '6'],
    ['.', '6', '.', '.', '.', '.', '2', '8', '.'],
    ['.', '.', '.', '4', '1', '9', '.', '.', '5'],
    ['.', '.', '.', '.', '8', '.', '.', '7', '9']
]
print(f"数独验证结果: {BitSetApplicationsAdvanced.is_valid_sudoku(sudoku_board)}")

# Trie树示例
trie = BitSetApplicationsAdvanced.Trie()
trie.insert("apple")
print(f"搜索'apple': {trie.search('apple')}")
print(f"前缀'app': {trie.starts_with('app')}")

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

文件: Code21_BitAlgorithmChallenges.java

```
=====
package class032;

import java.util.*;

/**
 * 位算法挑战题和竞赛题目
 * 题目来源: LeetCode Hard, Codeforces, ACM 竞赛
 * 包含高难度的位算法问题和竞赛题目
 *
 * 解题思路:
 * 方法 1: 位掩码 + 动态规划
 * 方法 2: 位运算 + 数学推导
 * 方法 3: 状态压缩 + 搜索优化
 * 方法 4: 位操作 + 算法技巧
 *
 * 时间复杂度分析:
 * 方法 1: O(n * 2^n) - 状态压缩 DP
 * 方法 2: O(1) - 数学公式
 * 方法 3: O(2^n) - 状态枚举
 * 方法 4: O(n) - 线性扫描
 *
 * 空间复杂度分析:
 * 方法 1: O(2^n) - DP 数组
 * 方法 2: O(1) - 常数空间
 * 方法 3: O(2^n) - 状态存储
 * 方法 4: O(1) - 原地操作
 *
 * 工程化考量:
 * 1. 算法选择: 根据数据规模选择合适算法
 * 2. 内存优化: 使用滚动数组等技术
 * 3. 性能优化: 利用位运算的并行性
 * 4. 错误处理: 处理边界情况和异常输入
 */

```

```
public class Code21_BitAlgorithmChallenges {
```

```
    /**
     * LeetCode 52. N-Queens II - N 皇后问题 II
     * 题目链接: https://leetcode.com/problems/n-queens-ii/
     * 题目描述: 计算 N 皇后问题的不同解决方案的数量
     *
```

```

* 方法: 位运算优化的回溯算法
* 时间复杂度: O(n!) 但实际运行很快
* 空间复杂度: O(n)
*/
public static int totalNQueens(int n) {
    return solveNQueens(n, 0, 0, 0, 0);
}

private static int solveNQueens(int n, int row, int columns, int diagonals1, int diagonals2)
{
    if (row == n) {
        return 1;
    }

    int count = 0;
    // 获取可用的位置 (位为 0 表示可用)
    int availablePositions = ((1 << n) - 1) & ~ (columns | diagonals1 | diagonals2);

    while (availablePositions != 0) {
        // 获取最低位的 1
        int position = availablePositions & -availablePositions;
        // 清除最低位的 1
        availablePositions &= availablePositions - 1;

        count += solveNQueens(n, row + 1,
            columns | position,
            (diagonals1 | position) << 1,
            (diagonals2 | position) >> 1);
    }
}

return count;
}

/***
* LeetCode 691. Stickers to Spell Word - 贴纸拼词
* 题目链接: https://leetcode.com/problems/stickers-to-spell-word/
* 题目描述: 给定一组贴纸和目标单词，计算最少需要多少张贴纸才能拼出目标单词
*
* 方法: 状态压缩 + BFS
* 时间复杂度: O(2^n * m * 26) n 为目标单词长度, m 为贴纸数量
* 空间复杂度: O(2^n)
*/
public static int minStickers(String[] stickers, String target) {

```

```

int n = target.length();
int[] dp = new int[1 << n];
Arrays.fill(dp, Integer.MAX_VALUE);
dp[0] = 0;

for (int state = 0; state < (1 << n); state++) {
    if (dp[state] == Integer.MAX_VALUE) continue;

    for (String sticker : stickers) {
        int nextState = state;
        int[] count = new int[26];

        // 统计贴纸中每个字符的数量
        for (char c : sticker.toCharArray()) {
            count[c - 'a']++;
        }

        // 尝试使用贴纸覆盖目标单词
        for (int i = 0; i < n; i++) {
            if (((nextState >> i) & 1) == 1) continue; // 已经覆盖

            char c = target.charAt(i);
            if (count[c - 'a'] > 0) {
                count[c - 'a']--;
                nextState |= (1 << i);
            }
        }

        if (dp[nextState] > dp[state] + 1) {
            dp[nextState] = dp[state] + 1;
        }
    }
}

return dp[(1 << n) - 1] == Integer.MAX_VALUE ? -1 : dp[(1 << n) - 1];
}

/**
 * LeetCode 864. Shortest Path to Get All Keys - 获取所有钥匙的最短路径
 * 题目链接: https://leetcode.com/problems/shortest-path-to-get-all-keys/
 * 题目描述: 在网格中找到获取所有钥匙的最短路径
 *
 * 方法: BFS + 状态压缩

```

```

* 时间复杂度: O(m * n * 2^k) k 为钥匙数量
* 空间复杂度: O(m * n * 2^k)
*/
public static int shortestPathAllKeys(String[] grid) {
    int m = grid.length, n = grid[0].length();
    int allKeys = 0;
    int startX = -1, startY = -1;

    // 找到起点和所有钥匙
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            char c = grid[i].charAt(j);
            if (c == '@') {
                startX = i;
                startY = j;
            } else if (c >= 'a' && c <= 'f') {
                allKeys |= (1 << (c - 'a'));
            }
        }
    }

    // BFS 搜索
    int[][][] dist = new int[m][n][1 << 6];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            Arrays.fill(dist[i][j], Integer.MAX_VALUE);
        }
    }

    dist[startX][startY][0] = 0;
    Queue<int[]> queue = new LinkedList<>();
    queue.offer(new int[]{startX, startY, 0});

    int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    while (!queue.isEmpty()) {
        int[] current = queue.poll();
        int x = current[0], y = current[1], keys = current[2];
        int distance = dist[x][y][keys];

        if (keys == allKeys) {
            return distance;
        }

        for (int[] direction : directions) {
            int newX = x + direction[0], newY = y + direction[1];
            if (newX < 0 || newX >= m || newY < 0 || newY >= n) {
                continue;
            }
            int newKeys = keys;
            if (grid[newX][newY] >= 'a' && grid[newX][newY] <= 'f') {
                newKeys |= (1 << (grid[newX][newY] - 'a'));
            }
            if (dist[newX][newY][newKeys] >= Integer.MAX_VALUE) {
                dist[newX][newY][newKeys] = distance + 1;
                queue.offer(new int[]{newX, newY, newKeys});
            }
        }
    }
}

```

```

        for (int[] dir : directions) {
            int nx = x + dir[0], ny = y + dir[1];
            if (nx < 0 || nx >= m || ny < 0 || ny >= n) continue;

            char c = grid[nx].charAt(ny);
            if (c == '#') continue; // 墙

            int newKeys = keys;
            if (c >= 'A' && c <= 'F') {
                // 遇到锁，检查是否有对应的钥匙
                int lock = c - 'A';
                if ((keys & (1 << lock)) == 0) continue; // 没有钥匙
            } else if (c >= 'a' && c <= 'f') {
                // 捡到钥匙
                newKeys |= (1 << (c - 'a'));
            }

            if (dist[nx][ny][newKeys] > distance + 1) {
                dist[nx][ny][newKeys] = distance + 1;
                queue.offer(new int[] {nx, ny, newKeys});
            }
        }

        return -1;
    }

    /**
     * Codeforces Problem: Xor-MST
     * 题目描述：给定 n 个点的完全图，边权为两点值的异或，求最小生成树
     *
     * 方法：分治 + 贪心 + Trie 树
     * 时间复杂度：O(n * log(max_value))
     * 空间复杂度：O(n * log(max_value))
     */
}

public static long xorMST(int[] arr) {
    Arrays.sort(arr);
    return buildMST(arr, 0, arr.length - 1, 30);
}

private static long buildMST(int[] arr, int left, int right, int bit) {
    if (left >= right || bit < 0) return 0;
}

```

```

// 根据当前位将数组分成两部分
int mid = left;
while (mid <= right && ((arr[mid] >> bit) & 1) == 0) {
    mid++;
}
mid--; // 最后一个 0 的位置

if (mid < left || mid >= right) {
    // 所有数在当前位相同, 继续下一位
    return buildMST(arr, left, right, bit - 1);
}

// 递归构建左右子树
long cost = buildMST(arr, left, mid, bit - 1) +
            buildMST(arr, mid + 1, right, bit - 1);

// 找到连接两部分的最小边
long minEdge = findMinXor(arr, left, mid, mid + 1, right);

return cost + minEdge;
}

private static long findMinXor(int[] arr, int l1, int r1, int l2, int r2) {
    long minXor = Long.MAX_VALUE;

    // 暴力查找最小异或值 (实际可以使用 Trie 优化)
    for (int i = l1; i <= r1; i++) {
        for (int j = l2; j <= r2; j++) {
            minXor = Math.min(minXor, (long)arr[i] ^ arr[j]);
        }
    }

    return minXor;
}

/***
 * ACM 竞赛题: 位运算最大值
 * 题目描述: 给定数组, 选择两个数进行位运算, 求最大值
 */
public static int maxBitwiseOperation(int[] arr, String operation) {
    int maxResult = Integer.MIN_VALUE;

```

```

switch (operation) {
    case "AND":
        // 寻找最大的 AND 值对
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                maxResult = Math.max(maxResult, arr[i] & arr[j]);
            }
        }
        break;

    case "OR":
        // 寻找最大的 OR 值对
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                maxResult = Math.max(maxResult, arr[i] | arr[j]);
            }
        }
        break;

    case "XOR":
        // 使用 Trie 树优化
        maxResult = findMaxXOR(arr);
        break;
}

return maxResult;
}

private static int findMaxXOR(int[] arr) {
    if (arr == null || arr.length == 0) return 0;

    // 构建前缀树
    TrieNode root = new TrieNode();

    // 插入所有数字
    for (int num : arr) {
        TrieNode node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node.children[bit] == null) {
                node.children[bit] = new TrieNode();
            }
            node = node.children[bit];
        }
    }
}

```

```
        }

    }

// 查找最大异或值
int maxXor = 0;
for (int num : arr) {
    TrieNode node = root;
    int currentXor = 0;

    for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        int oppositeBit = 1 - bit;

        if (node.children[oppositeBit] != null) {
            currentXor |= (1 << i);
            node = node.children[oppositeBit];
        } else {
            node = node.children[bit];
        }
    }

    maxXor = Math.max(maxXor, currentXor);
}

return maxXor;
}

// 前缀树节点定义
static class TrieNode {
    TrieNode[] children;

    public TrieNode() {
        children = new TrieNode[2];
    }
}

/***
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("== 位算法挑战题 - 单元测试 ==");
    // 测试 N 皇后问题
}
```

```

System.out.println("N 皇后问题测试:");
for (int n = 1; n <= 8; n++) {
    System.out.printf("%d 皇后的解决方案数量: %d%n", n, totalNQueens(n));
}

// 测试贴纸拼词
System.out.println("\n 贴纸拼词测试:");
String[] stickers = {"with", "example", "science"};
String target = "thehat";
System.out.printf("贴纸%s 拼出'%s' 需要的最少贴纸: %d%n",
    Arrays.toString(stickers), target, minStickers(stickers, target));

// 测试获取所有钥匙的最短路径
System.out.println("\n 获取所有钥匙的最短路径测试:");
String[] grid = {"@. a. #", "###. #", "b. A. B"};
System.out.printf("网格%s 的最短路径长度: %d%n",
    Arrays.toString(grid), shortestPathAllKeys(grid));

// 测试异或最小生成树
System.out.println("\n 异或最小生成树测试:");
int[] points = {1, 2, 3, 4};
System.out.printf("点集%s 的异或 MST 权重: %d%n",
    Arrays.toString(points), xorMST(points));

// 测试位运算最大值
System.out.println("\n 位运算最大值测试:");
int[] numbers = {3, 10, 5, 25, 2, 8};
System.out.printf("数组%s 的最大 AND 值: %d%n",
    Arrays.toString(numbers), maxBitwiseOperation(numbers, "AND"));
System.out.printf("数组%s 的最大 OR 值: %d%n",
    Arrays.toString(numbers), maxBitwiseOperation(numbers, "OR"));
System.out.printf("数组%s 的最大 XOR 值: %d%n",
    Arrays.toString(numbers), maxBitwiseOperation(numbers, "XOR"));

}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");
    // 测试 N 皇后问题的性能
    long startTime = System.nanoTime();
}

```

```

int result8 = totalNQueens(8);
long time1 = System.nanoTime() - startTime;
System.out.printf("8 皇后问题: %d ns, 结果: %d%n", time1, result8);

// 测试贴纸拼词的性能
String[] stickers = {"with", "example", "science"};
String target = "thehat";
startTime = System.nanoTime();
int resultStickers = minStickers(stickers, target);
long time2 = System.nanoTime() - startTime;
System.out.printf("贴纸拼词: %d ns, 结果: %d%n", time2, resultStickers);

// 测试异或 MST 的性能
int[] largePoints = new int[100];
Random random = new Random();
for (int i = 0; i < largePoints.length; i++) {
    largePoints[i] = random.nextInt(1000);
}
startTime = System.nanoTime();
long resultMST = xorMST(largePoints);
long time3 = System.nanoTime() - startTime;
System.out.printf("异或 MST: %d ns, 结果: %d%n", time3, resultMST);
}

/**
 * 复杂度分析
 */
public static void complexityAnalysis() {
    System.out.println("\n==== 复杂度分析 ====");
    System.out.println("挑战题算法的特点:");
    System.out.println("1. 时间复杂度: 通常较高, 需要优化");
    System.out.println("2. 空间复杂度: 使用状态压缩减少内存");
    System.out.println("3. 算法技巧: 结合多种算法思想");

    System.out.println("\n 优化策略:");
    System.out.println("1. 状态压缩: 使用位表示状态集合");
    System.out.println("2. 记忆化搜索: 避免重复计算");
    System.out.println("3. 剪枝优化: 提前终止不可能的分支");
    System.out.println("4. 数据结构优化: 使用高效数据结构");
}

public static void main(String[] args) {
    System.out.println("位算法挑战题和竞赛题目");
}

```

```

System.out.println("包含 LeetCode Hard、Codeforces 等高难度问题");

// 运行单元测试
runTests();

// 性能测试
performanceTest();

// 复杂度分析
complexityAnalysis();

// 竞赛技巧
System.out.println("\n== 竞赛技巧 ==");
System.out.println("1. 识别问题模式：判断是否适合位运算");
System.out.println("2. 状态设计：合理设计状态表示");
System.out.println("3. 算法选择：根据数据规模选择算法");
System.out.println("4. 调试技巧：使用小数据测试");
}

}
=====

文件: Code21_BitAlgorithmComprehensive.cpp
=====

// 综合位算法实现 - 简化版本
// 不使用标准库，只使用基本 C 语言特性

#include <stdio.h>

// 综合位算法实现类
class BitAlgorithmComprehensive {
public:
    // LeetCode 191 - 位 1 的个数
    static int hammingWeight(unsigned int n) {
        int count = 0;
        while (n != 0) {
            n &= (n - 1);
            count++;
        }
        return count;
    }

    // LeetCode 231 - 2 的幂
}
```

```

static bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

// LeetCode 342 - 4 的幂
static bool isPowerOfFour(int n) {
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}

// LeetCode 268 - 丢失的数字
static int missingNumber(int nums[], int n) {
    int result = n;
    for (int i = 0; i < n; i++) {
        result ^= i;
        result ^= nums[i];
    }
    return result;
}

// LeetCode 136 - 只出现一次的数字
static int singleNumber(int nums[], int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        result ^= nums[i];
    }
    return result;
}

// LeetCode 137 - 只出现一次的数字 II
static int singleNumberII(int nums[], int n) {
    int ones = 0, twos = 0;
    for (int i = 0; i < n; i++) {
        ones = (ones ^ nums[i]) & ~twos;
        twos = (twos ^ nums[i]) & ~ones;
    }
    return ones;
}

// LeetCode 260 - 只出现一次的数字 III
static void singleNumberIII(int nums[], int n, int* num1, int* num2) {
    int xor_result = 0;
    for (int i = 0; i < n; i++) {
        xor_result ^= nums[i];
    }

```

```

}

int diff_bit = xor_result & -xor_result;

*num1 = 0;
*num2 = 0;
for (int i = 0; i < n; i++) {
    if (nums[i] & diff_bit) {
        *num1 ^= nums[i];
    } else {
        *num2 ^= nums[i];
    }
}

// LeetCode 190 - 颠倒二进制位
static unsigned int reverseBits(unsigned int n) {
    n = (n >> 16) | (n << 16);
    n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0aaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
    return n;
}

// LeetCode 338 - 比特位计数
static void countBits(int n, int result[]) {
    for (int i = 1; i <= n; i++) {
        result[i] = result[i >> 1] + (i & 1);
    }
}

// LeetCode 201 - 数字范围按位与
static int rangeBitwiseAnd(int m, int n) {
    int shift = 0;
    while (m < n) {
        m >>= 1;
        n >>= 1;
        shift++;
    }
    return m << shift;
}
};

```

```
// 测试函数
int main() {
    printf("综合位算法实现测试
");
    printf("=====
");
}

// 测试位 1 的个数
printf("位 1 的个数测试:
");
printf("数字 11(1011) 的 1 的个数: %d
", BitAlgorithmComprehensive::hammingWeight(11));

// 测试 2 的幂
printf("2 的幂测试:
");
printf("16 是 2 的幂: %s
", BitAlgorithmComprehensive::isPowerOfTwo(16) ? "是" : "否");
printf("15 是 2 的幂: %s
", BitAlgorithmComprehensive::isPowerOfTwo(15) ? "是" : "否");

// 测试只出现一次的数字
printf("只出现一次的数字测试:
");
int nums[] = {4, 1, 2, 1, 2};
printf("只出现一次的数字: %d
", BitAlgorithmComprehensive::singleNumber(nums, 5));

// 测试丢失的数字
printf("丢失的数字测试:
");
int nums2[] = {3, 0, 1};
printf("丢失的数字: %d
", BitAlgorithmComprehensive::missingNumber(nums2, 3));

// 测试只出现一次的数字 III
printf("只出现一次的数字 III 测试:
");

```

```

int nums3[] = {1, 2, 1, 3, 2, 5};
int num1, num2;
BitAlgorithmComprehensive::singleNumberIII(nums3, 6, &num1, &num2);
printf("两个只出现一次的数字: %d, %d
", num1, num2);

// 测试颠倒二进制位
printf("颠倒二进制位测试:
");
printf("43261596 颠倒后: %u
", BitAlgorithmComprehensive::reverseBits(43261596));

// 测试比特位计数
printf("比特位计数测试:
");
int result[6] = {0};
BitAlgorithmComprehensive::countBits(5, result);
printf("比特位计数(0-5): ");
for (int i = 0; i <= 5; i++) {
    printf("%d ", result[i]);
}
printf("

");

// 测试数字范围按位与
printf("数字范围按位与测试:
");
printf("[5, 7]按位与结果: %d
", BitAlgorithmComprehensive::rangeBitwiseAnd(5, 7));

return 0;
}

using namespace std;

/**
 * 综合位算法实现
 * 包含 LeetCode 多个综合位算法相关题目的解决方案
 *
 * 题目列表:

```

```
* 1. LeetCode 191 - 位 1 的个数
* 2. LeetCode 231 - 2 的幂
* 3. LeetCode 342 - 4 的幂
* 4. LeetCode 268 - 丢失的数字
* 5. LeetCode 136 - 只出现一次的数字
* 6. LeetCode 137 - 只出现一次的数字 II
* 7. LeetCode 260 - 只出现一次的数字 III
* 8. LeetCode 190 - 颠倒二进制位
* 9. LeetCode 338 - 比特位计数
* 10. LeetCode 201 - 数字范围按位与
*
* 时间复杂度分析:
* - 位操作: O(1) 到 O(n)
* - 空间复杂度: O(1) 到 O(n)
*
* 工程化考量:
* 1. 位运算优化: 使用位运算替代算术运算
* 2. 状态压缩: 使用位掩码压缩状态
* 3. 性能优化: 利用位运算的并行性
* 4. 边界处理: 处理整数边界、负数等
*/

```

```
class BitAlgorithmComprehensive {
public:
    /**
     * LeetCode 191 - 位 1 的个数
     * 题目链接: https://leetcode.com/problems/number-of-1-bits/
     * 编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。
     *
     * 方法: Brian Kernighan 算法
     * 时间复杂度: O(k) - k 为 1 的个数
     * 空间复杂度: O(1)
     *
     * 原理: n & (n-1) 会清除最低位的 1
    */
    static int hammingWeight(uint32_t n) {
        int count = 0;
        while (n != 0) {
            n &= (n - 1);
            count++;
        }
        return count;
    }
}
```

```

}

/***
 * LeetCode 231 - 2 的幂
 * 题目链接: https://leetcode.com/problems/power-of-two/
 * 给定一个整数，编写一个函数来判断它是否是 2 的幂次方。
 *
 * 方法: 位运算
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 *
 * 原理: 2 的幂的二进制表示中只有 1 个 1
 */

static bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

/***
 * LeetCode 342 - 4 的幂
 * 题目链接: https://leetcode.com/problems/power-of-four/
 * 给定一个整数，写一个函数来判断它是否是 4 的幂次方。
 *
 * 方法: 位运算 + 数学
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 *
 * 原理: 4 的幂是 2 的幂，且 1 出现在奇数位
 */

static bool isPowerOfFour(int n) {
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}

/***
 * LeetCode 268 - 丢失的数字
 * 题目链接: https://leetcode.com/problems/missing-number/
 * 给定一个包含 [0, n] 中 n 个数的数组 nums，找出 [0, n] 这个范围内没有出现在数组中的那个数。
 *
 * 方法: 位运算 (异或)
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 *
 * 原理: a ^ a = 0, a ^ 0 = a
 */

```

```

*/
static int missingNumber(vector<int>& nums) {
    int n = nums.size();
    int result = n; // 因为 nums 包含 0 到 n-1，所以初始化为 n

    for (int i = 0; i < n; i++) {
        result ^= i;
        result ^= nums[i];
    }

    return result;
}

/***
 * LeetCode 136 - 只出现一次的数字
 * 题目链接: https://leetcode.com/problems/single-number/
 * 给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。
 *
 * 方法：位运算（异或）
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 */
static int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

/***
 * LeetCode 137 - 只出现一次的数字 II
 * 题目链接: https://leetcode.com/problems/single-number-ii/
 * 给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现三次。找出那个只出现了一次的元素。
 *
 * 方法：位运算（状态机）
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 *
 * 原理：使用两个变量记录状态，模拟三进制计数
*/

```

```

static int singleNumberII(vector<int>& nums) {
    int ones = 0, twos = 0;

    for (int num : nums) {
        ones = (ones ^ num) & ~twos;
        twos = (twos ^ num) & ~ones;
    }

    return ones;
}

```

/**

* LeetCode 260 - 只出现一次的数字 III
* 题目链接: <https://leetcode.com/problems/single-number-iii/>
* 给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

*

* 方法：位运算（分组异或）

* 时间复杂度：O(n)

* 空间复杂度：O(1)

*/

```
static vector<int> singleNumberIII(vector<int>& nums) {
```

// 获取两个不同数字的异或结果

int xor_result = 0;

for (int num : nums) {

xor_result ^= num;

}

// 找到最低位的 1（两个数字不同的位）

int diff_bit = xor_result & ~xor_result;

// 根据该位分组

int num1 = 0, num2 = 0;

for (int num : nums) {

if (num & diff_bit) {

num1 ^= num;

} else {

num2 ^= num;

}

}

return {num1, num2};

}

```

/**
 * LeetCode 190 - 颠倒二进制位
 * 题目链接: https://leetcode.com/problems/reverse-bits/
 * 颠倒给定的 32 位无符号整数的二进制位。
 *
 * 方法: 位运算 (分治)
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */

static uint32_t reverseBits(uint32_t n) {
    n = (n >> 16) | (n << 16);
    n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xaaaaaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
    return n;
}

/**
 * LeetCode 338 - 比特位计数
 * 题目链接: https://leetcode.com/problems/counting-bits/
 * 给定一个非负整数 num。对于  $0 \leq i \leq num$  范围中的每个数字 i，计算其二进制数中的 1 的数目
 * 并将它们作为数组返回。
 *
 * 方法: 动态规划 + 位运算
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 原理: i 的 1 的个数 = i/2 的 1 的个数 + i 的最低位是否为 1
 */

static vector<int> countBits(int n) {
    vector<int> result(n + 1, 0);

    for (int i = 1; i <= n; i++) {
        result[i] = result[i >> 1] + (i & 1);
    }

    return result;
}

/**
 * LeetCode 201 - 数字范围按位与

```

```

* 题目链接: https://leetcode.com/problems/bitwise-and-of-numbers-range/
* 给定范围 [m, n]，返回此范围内所有数字的按位与（包含 m, n 两端点）。
*
* 方法: 位运算 (找公共前缀)
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 原理: 结果就是 m 和 n 的二进制公共前缀
*/
static int rangeBitwiseAnd(int m, int n) {
    int shift = 0;

    // 找到公共前缀
    while (m < n) {
        m >>= 1;
        n >>= 1;
        shift++;
    }

    return m << shift;
}

};

class PerformanceTester {
public:
    static void testHammingWeight() {
        cout << "==== 位1的个数性能测试 ===" << endl;

        uint32_t n = 0b10101010101010101010101010101010101010;

        auto start = chrono::high_resolution_clock::now();
        int count = BitAlgorithmComprehensive::hammingWeight(n);
        auto time = chrono::duration_cast<chrono::nanoseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "hammingWeight: 数字" << n << "的1的个数=" << count << ", 耗时=" << time << "
ns" << endl;
    }

    static void testSingleNumber() {
        cout << "\n==== 只出现一次的数字性能测试 ===" << endl;

        vector<int> nums = {4, 1, 2, 1, 2};

```

```

auto start = chrono::high_resolution_clock::now();
int result = BitAlgorithmComprehensive::singleNumber(nums);
auto time = chrono::duration_cast<chrono::nanoseconds>(
    chrono::high_resolution_clock::now() - start).count();

cout << "singleNumber: 结果=" << result << ", 耗时=" << time << " ns" << endl;
}

static void testCountBits() {
    cout << "\n==== 比特位计数性能测试 ===" << endl;

    int n = 1000000;

    auto start = chrono::high_resolution_clock::now();
    vector<int> result = BitAlgorithmComprehensive::countBits(n);
    auto time = chrono::duration_cast<chrono::milliseconds>(
        chrono::high_resolution_clock::now() - start).count();

    cout << "countBits: n=" << n << ", 耗时=" << time << " ms" << endl;
    cout << "示例结果: [0, 1, 1, 2, 1, 2, ...," << result[n] << "]" << endl;
}

static void runUnitTests() {
    cout << "==== 综合位算法单元测试 ===" << endl;

    // 测试位 1 的个数
    assert(BitAlgorithmComprehensive::hammingWeight(0b1011) == 3);

    // 测试 2 的幂
    assert(BitAlgorithmComprehensive::isPowerOfTwo(16) == true);
    assert(BitAlgorithmComprehensive::isPowerOfTwo(15) == false);

    // 测试只出现一次的数字
    vector<int> nums = {2, 2, 1};
    assert(BitAlgorithmComprehensive::singleNumber(nums) == 1);

    cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;
}

```

```
vector<pair<string, string>> algorithms = {
    {"hammingWeight", "O(k), O(1)"},  

    {"isPowerOfTwo", "O(1), O(1)"},  

    {"isPowerOfFour", "O(1), O(1)"},  

    {"missingNumber", "O(n), O(1)"},  

    {"singleNumber", "O(n), O(1)"},  

    {"singleNumberII", "O(n), O(1)"},  

    {"singleNumberIII", "O(n), O(1)"},  

    {"reverseBits", "O(1), O(1)"},  

    {"countBits", "O(n), O(n)"},  

    {"rangeBitwiseAnd", "O(1), O(1)"}
};  
  
for (auto& algo : algorithms) {  
    cout << algo.first << ": 时间复杂度=" << algo.second << endl;  
}  
}  
};  
  
int main() {  
    cout << "综合位算法实现" << endl;  
    cout << "包含 LeetCode 多个综合位算法相关题目的解决方案" << endl;  
    cout << "===== " << endl;  
  
    // 运行单元测试  
    PerformanceTester::runUnitTests();  
  
    // 运行性能测试  
    PerformanceTester::testHammingWeight();  
    PerformanceTester::testSingleNumber();  
    PerformanceTester::testCountBits();  
  
    // 复杂度分析  
    PerformanceTester::complexityAnalysis();  
  
    // 示例使用  
    cout << "\n==== 示例使用 ===" << endl;  
  
    // 位 1 的个数示例  
    uint32_t num = 0b1011;  
    cout << "数字" << num << "的 1 的个数：" << BitAlgorithmComprehensive::hammingWeight(num) <<  
    endl;
```

```

// 2 的幂示例
cout << "16 是 2 的幂: " << (BitAlgorithmComprehensive::isPowerOfTwo(16) ? "是" : "否") <<
endl;

// 只出现一次的数字示例
vector<int> nums = {4, 1, 2, 1, 2};
cout << "只出现一次的数字: " << BitAlgorithmComprehensive::singleNumber(nums) << endl;

// 比特位计数示例
vector<int> bits = BitAlgorithmComprehensive::countBits(5);
cout << "比特位计数(0-5): ";
for (int i = 0; i <= 5; i++) {
    cout << bits[i] << " ";
}
cout << endl;

return 0;
}

```

=====

文件: Code21_BitAlgorithmComprehensive.py

=====

"""

综合位算法实现

包含 LeetCode 多个综合位算法相关题目的解决方案

题目列表:

1. LeetCode 191 - 位 1 的个数
2. LeetCode 231 - 2 的幂
3. LeetCode 342 - 4 的幂
4. LeetCode 268 - 丢失的数字
5. LeetCode 136 - 只出现一次的数字
6. LeetCode 137 - 只出现一次的数字 II
7. LeetCode 260 - 只出现一次的数字 III
8. LeetCode 190 - 颠倒二进制位
9. LeetCode 338 - 比特位计数
10. LeetCode 201 - 数字范围按位与

时间复杂度分析:

- 位操作: O(1) 到 O(n)
- 空间复杂度: O(1) 到 O(n)

工程化考量：

1. 位运算优化：使用位运算替代算术运算
2. 状态压缩：使用位掩码压缩状态
3. 性能优化：利用位运算的并行性
4. 边界处理：处理整数边界、负数等

"""

```
import time
from typing import List
import sys
```

class BitAlgorithmComprehensive:

"""综合位算法类"""

@staticmethod

def hamming_weight(n: int) -> int:

"""

LeetCode 191 - 位 1 的个数

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

方法：Brian Kernighan 算法

时间复杂度：O(k) - k 为 1 的个数

空间复杂度：O(1)

原理：n & (n-1) 会清除最低位的 1

"""

count = 0

while n != 0:

n &= (n - 1)

count += 1

return count

@staticmethod

def is_power_of_two(n: int) -> bool:

"""

LeetCode 231 - 2 的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

方法：位运算

时间复杂度：O(1)

空间复杂度：O(1)

原理：2的幂的二进制表示中只有1个1

"""

```
return n > 0 and (n & (n - 1)) == 0
```

@staticmethod

```
def is_power_of_four(n: int) -> bool:
```

"""

LeetCode 342 - 4的幂

给定一个整数，写一个函数来判断它是否是4的幂次方。

方法：位运算 + 数学

时间复杂度：O(1)

空间复杂度：O(1)

原理：4的幂是2的幂，且1出现在奇数位

"""

```
return n > 0 and (n & (n - 1)) == 0 and (n & 0x55555555) != 0
```

@staticmethod

```
def missing_number(nums: List[int]) -> int:
```

"""

LeetCode 268 - 丢失的数字

给定一个包含 $[0, n]$ 中 n 个数的数组 nums ，找出 $[0, n]$ 这个范围内没有出现在数组中的那个数。

方法：位运算（异或）

时间复杂度：O(n)

空间复杂度：O(1)

原理： $a \wedge a = 0$, $a \wedge 0 = a$

"""

```
n = len(nums)
```

result = n # 因为 nums 包含0到 $n-1$ ，所以初始化为n

```
for i in range(n):
```

```
    result ^= i
```

```
    result ^= nums[i]
```

```
return result
```

@staticmethod

```
def single_number(nums: List[int]) -> int:
```

"""

LeetCode 136 - 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

方法：位运算（异或）

时间复杂度：O(n)

空间复杂度：O(1)

"""

```
result = 0
```

```
for num in nums:
```

```
    result ^= num
```

```
return result
```

```
@staticmethod
```

```
def single_number_ii(nums: List[int]) -> int:
```

"""

LeetCode 137 - 只出现一次的数字 II

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现三次。找出那个只出现了一次的元素。

方法：位运算（状态机）

时间复杂度：O(n)

空间复杂度：O(1)

原理：使用两个变量记录状态，模拟三进制计数

"""

```
ones = 0
```

```
twos = 0
```

```
for num in nums:
```

```
    ones = (ones ^ num) & ~twos
```

```
    twos = (twos ^ num) & ~ones
```

```
return ones
```

```
@staticmethod
```

```
def single_number_iii(nums: List[int]) -> List[int]:
```

"""

LeetCode 260 - 只出现一次的数字 III

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

方法：位运算（分组异或）

```

时间复杂度: O(n)
空间复杂度: O(1)
"""

# 获取两个不同数字的异或结果
xor_result = 0
for num in nums:
    xor_result ^= num

# 找到最低位的 1 (两个数字不同的位)
diff_bit = xor_result & -xor_result

# 根据该位分组
num1 = 0
num2 = 0
for num in nums:
    if num & diff_bit:
        num1 ^= num
    else:
        num2 ^= num

return [num1, num2]

```

```

@staticmethod
def reverse_bits(n: int) -> int:
"""

LeetCode 190 - 颠倒二进制位
颠倒给定的 32 位无符号整数的二进制位。

```

方法: 位运算 (分治)
 时间复杂度: O(1)
 空间复杂度: O(1)

```

"""

# 确保是 32 位无符号整数
n = n & 0xFFFFFFFF

# 分治颠倒
n = (n >> 16) | (n << 16) & 0xFFFFFFFF
n = ((n & 0xFF00FF00) >> 8) | ((n & 0x00FF00FF) << 8) & 0xFFFFFFFF
n = ((n & 0xF0F0F0F0) >> 4) | ((n & 0x0F0F0F0F) << 4) & 0xFFFFFFFF
n = ((n & 0xCCCCCCCC) >> 2) | ((n & 0x33333333) << 2) & 0xFFFFFFFF
n = ((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1) & 0xFFFFFFFF

return n

```

```
@staticmethod  
def count_bits(n: int) -> List[int]:  
    """
```

LeetCode 338 - 比特位计数

给定一个非负整数 num。对于 $0 \leq i \leq \text{num}$ 范围中的每个数字 i ，计算其二进制数中的 1 的数目，并将它们作为数组返回。

方法：动态规划 + 位运算

时间复杂度：O(n)

空间复杂度：O(n)

原理： i 的 1 的个数 = $i/2$ 的 1 的个数 + i 的最低位是否为 1

```
"""
```

```
result = [0] * (n + 1)
```

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

```
    result[i] = result[i >> 1] + (i & 1)
```

```
return result
```

```
@staticmethod
```

```
def range_bitwise_and(m: int, n: int) -> int:
```

```
"""
```

LeetCode 201 - 数字范围按位与

给定范围 $[m, n]$ ，返回此范围内所有数字的按位与（包含 m, n 两端点）。

方法：位运算（找公共前缀）

时间复杂度：O(1)

空间复杂度：O(1)

原理：结果就是 m 和 n 的二进制公共前缀

```
"""
```

```
shift = 0
```

```
# 找到公共前缀
```

```
while m < n:
```

```
    m >>= 1
```

```
    n >>= 1
```

```
    shift += 1
```

```
return m << shift
```



```

@staticmethod
def run_unit_tests():
    """运行单元测试"""
    print("== 综合位算法单元测试 ==")

    # 测试位 1 的个数
    assert BitAlgorithmComprehensive.hamming_weight(0b1011) == 3

    # 测试 2 的幂
    assert BitAlgorithmComprehensive.is_power_of_two(16) == True
    assert BitAlgorithmComprehensive.is_power_of_two(15) == False

    # 测试只出现一次的数字
    nums = [2, 2, 1]
    assert BitAlgorithmComprehensive.single_number(nums) == 1

    print("所有单元测试通过!")
}

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n== 复杂度分析 ==")

    algorithms = {
        "hamming_weight": ("O(k)", "O(1)"),
        "is_power_of_two": ("O(1)", "O(1)"),
        "is_power_of_four": ("O(1)", "O(1)"),
        "missing_number": ("O(n)", "O(1)"),
        "single_number": ("O(n)", "O(1)"),
        "single_number_ii": ("O(n)", "O(1)"),
        "single_number_iii": ("O(n)", "O(1)"),
        "reverse_bits": ("O(1)", "O(1)"),
        "count_bits": ("O(n)", "O(n)"),
        "range_bitwise_and": ("O(1)", "O(1)")
    }

    for name, (time_complexity, space_complexity) in algorithms.items():
        print(f"{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}")
}

def main():
    """主函数"""

```

```
print("综合位算法实现")
print("包含 LeetCode 多个综合位算法相关题目的解决方案")
print("=" * 50)

# 运行单元测试
PerformanceTester.run_unit_tests()

# 运行性能测试
PerformanceTester.test_hamming_weight()
PerformanceTester.test_single_number()
PerformanceTester.test_count_bits()

# 复杂度分析
PerformanceTester.complexity_analysis()

# 示例使用
print("\n==== 示例使用 ====")

# 位 1 的个数示例
num = 0b1011
print(f"数字 {num} 的 1 的个数: {BitAlgorithmComprehensive.hamming_weight(num)}")

# 2 的幂示例
print(f"16 是 2 的幂: {'是' if BitAlgorithmComprehensive.is_power_of_two(16) else '否'}")

# 只出现一次的数字示例
nums = [4, 1, 2, 1, 2]
print(f"只出现一次的数字: {BitAlgorithmComprehensive.single_number(nums)}")

# 比特位计数示例
bits = BitAlgorithmComprehensive.count_bits(5)
print(f"比特位计数(0-5): {''.join(map(str, bits))}")

# 数字范围按位与示例
m, n = 5, 7
print(f"数字范围按位与 [{m}, {n}]: {BitAlgorithmComprehensive.range_bitwise_and(m, n)}")

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

文件: Code22_BitAlgorithmApplications.cpp

```
=====
```

```
#include <iostream>
#include <vector>
#include <bitset>
#include <stdexcept>
#include <chrono>
#include <random>
#include <algorithm>
#include <functional>
#include <map>
#include <unordered_map>
#include <queue>
#include <stack>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>
#include <climits>
#include <cassert>
#include <limits>
#include <unordered_set>
#include <cstdint>
#include <set>
#include <numeric>
```

```
using namespace std;
```

```
/***
 * 位算法应用实现
 * 包含 LeetCode 多个位算法应用相关题目的解决方案
 *
 * 题目列表:
 * 1. LeetCode 78 - 子集
 * 2. LeetCode 90 - 子集 II
 * 3. LeetCode 46 - 全排列
 * 4. LeetCode 47 - 全排列 II
 * 5. LeetCode 77 - 组合
 * 6. LeetCode 39 - 组合总和
 * 7. LeetCode 40 - 组合总和 II
 * 8. LeetCode 216 - 组合总和 III
 * 9. LeetCode 131 - 分割回文串
 * 10. LeetCode 93 - 复原 IP 地址
```

```

*
* 时间复杂度分析:
* - 回溯算法: O(2^n) 到 O(n!)
* - 空间复杂度: O(n) 到 O(n^2)
*
* 工程化考量:
* 1. 位集优化: 使用位集优化回溯算法
* 2. 状态压缩: 使用位运算压缩状态空间
* 3. 剪枝优化: 使用位运算进行高效剪枝
* 4. 去重处理: 使用位掩码进行重复检测
*/

```

```

class BitAlgorithmApplications {
public:
    /**
     * LeetCode 78 - 子集
     * 题目链接: https://leetcode.com/problems/subsets/
     * 给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
     *
     * 方法: 位运算枚举
     * 时间复杂度: O(n * 2^n)
     * 空间复杂度: O(n * 2^n)
     *
     * 原理: 使用二进制位表示每个元素是否在子集中
    */
    static vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int total = 1 << n;
        vector<vector<int>> result;

        for (int mask = 0; mask < total; mask++) {
            vector<int> subset;
            for (int i = 0; i < n; i++) {
                if (mask & (1 << i)) {
                    subset.push_back(nums[i]);
                }
            }
            result.push_back(subset);
        }

        return result;
    }
}

```

```

/***
 * LeetCode 90 - 子集 II
 * 题目链接: https://leetcode.com/problems/subsets-ii/
 * 给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
 *
 * 方法：位运算 + 排序去重
 * 时间复杂度: O(n * 2^n)
 * 空间复杂度: O(n * 2^n)
 */
static vector<vector<int>> subsetsWithDup(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int n = nums.size();
    int total = 1 << n;
    set<vector<int>> result_set;

    for (int mask = 0; mask < total; mask++) {
        vector<int> subset;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                subset.push_back(nums[i]);
            }
        }
        result_set.insert(subset);
    }

    vector<vector<int>> result(result_set.begin(), result_set.end());
    return result;
}

/***
 * LeetCode 46 - 全排列
 * 题目链接: https://leetcode.com/problems/permutations/
 * 给定一个没有重复数字的序列，返回其所有可能的全排列。
 *
 * 方法：回溯 + 位掩码
 * 时间复杂度: O(n!)
 * 空间复杂度: O(n)
 */
static vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    vector<bool> used(nums.size(), false);
    backtrack(nums, used, current, result);
}

```

```

        return result;
    }

private:
    static void backtrack(vector<int>& nums, vector<bool>& used,
                         vector<int>& current, vector<vector<int>>& result) {
        if (current.size() == nums.size()) {
            result.push_back(current);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (!used[i]) {
                used[i] = true;
                current.push_back(nums[i]);
                backtrack(nums, used, current, result);
                current.pop_back();
                used[i] = false;
            }
        }
    }

public:
    /**
     * LeetCode 47 - 全排列 II
     * 题目链接: https://leetcode.com/problems/permutations-ii/
     * 给定一个可包含重复数字的序列，返回所有不重复的全排列。
     *
     * 方法: 回溯 + 排序剪枝
     * 时间复杂度: O(n!)
     * 空间复杂度: O(n)
     */
    static vector<vector<int>> permuteUnique(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> result;
        vector<int> current;
        vector<bool> used(nums.size(), false);
        backtrackUnique(nums, used, current, result);
        return result;
    }

private:
    static void backtrackUnique(vector<int>& nums, vector<bool>& used,

```

```

        vector<int>& current, vector<vector<int>>& result) {
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }

    for (int i = 0; i < nums.size(); i++) {
        if (used[i] || (i > 0 && nums[i] == nums[i-1] && !used[i-1])) {
            continue;
        }

        used[i] = true;
        current.push_back(nums[i]);
        backtrackUnique(nums, used, current, result);
        current.pop_back();
        used[i] = false;
    }
}

public:
/***
 * LeetCode 77 - 组合
 * 题目链接: https://leetcode.com/problems/combinations/
 * 给定两个整数 n 和 k, 返回 1 ... n 中所有可能的 k 个数的组合。
 *
 * 方法: 回溯
 * 时间复杂度: O(C(n, k))
 * 空间复杂度: O(k)
 */
static vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> result;
    vector<int> current;
    backtrackCombine(1, n, k, current, result);
    return result;
}

private:
    static void backtrackCombine(int start, int n, int k,
                                 vector<int>& current, vector<vector<int>>& result) {
        if (current.size() == k) {
            result.push_back(current);
            return;
        }
    }
}

```

```

        for (int i = start; i <= n; i++) {
            current.push_back(i);
            backtrackCombine(i + 1, n, k, current, result);
            current.pop_back();
        }
    }

public:
    /**
     * LeetCode 39 - 组合总和
     * 题目链接: https://leetcode.com/problems/combination-sum/
     * 给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字
     * 和为 target 的组合。
     *
     * 方法: 回溯
     * 时间复杂度: O(n^target)
     * 空间复杂度: O(target)
     */
    static vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> result;
        vector<int> current;
        sort(candidates.begin(), candidates.end());
        backtrackCombinationSum(candidates, target, 0, current, result);
        return result;
    }

private:
    static void backtrackCombinationSum(vector<int>& candidates, int target, int start,
                                         vector<int>& current, vector<vector<int>>& result) {
        if (target == 0) {
            result.push_back(current);
            return;
        }

        for (int i = start; i < candidates.size(); i++) {
            if (candidates[i] > target) {
                break;
            }

            current.push_back(candidates[i]);
            backtrackCombinationSum(candidates, target - candidates[i], i, current, result);
            current.pop_back();
        }
    }
}

```

```

        }

    }

public:
    /**
     * LeetCode 40 - 组合总和 II
     * 题目链接: https://leetcode.com/problems/combination-sum-ii/
     * 给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
     * candidates 中的每个数字在每个组合中只能使用一次。
     *
     * 方法: 回溯 + 排序剪枝
     * 时间复杂度: O(2^n)
     * 空间复杂度: O(n)
     */
    static vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<vector<int>> result;
        vector<int> current;
        sort(candidates.begin(), candidates.end());
        backtrackCombinationSum2(candidates, target, 0, current, result);
        return result;
    }

private:
    static void backtrackCombinationSum2(vector<int>& candidates, int target, int start,
                                         vector<int>& current, vector<vector<int>>& result) {
        if (target == 0) {
            result.push_back(current);
            return;
        }

        for (int i = start; i < candidates.size(); i++) {
            if (candidates[i] > target) {
                break;
            }

            if (i > start && candidates[i] == candidates[i-1]) {
                continue;
            }

            current.push_back(candidates[i]);
            backtrackCombinationSum2(candidates, target - candidates[i], i + 1, current, result);
            current.pop_back();
        }
    }
}

```

```

    }
}

public:
    /**
     * LeetCode 216 - 组合总和 III
     * 题目链接: https://leetcode.com/problems/combination-sum-iii/
     * 找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。
     *
     * 方法：回溯
     * 时间复杂度: O(C(9, k))
     * 空间复杂度: O(k)
     */
    static vector<vector<int>> combinationSum3(int k, int n) {
        vector<vector<int>> result;
        vector<int> current;
        backtrackCombinationSum3(1, k, n, current, result);
        return result;
    }

private:
    static void backtrackCombinationSum3(int start, int k, int n,
                                         vector<int>& current, vector<vector<int>>& result) {
        if (current.size() == k && n == 0) {
            result.push_back(current);
            return;
        }

        if (current.size() > k || n < 0) {
            return;
        }

        for (int i = start; i <= 9; i++) {
            current.push_back(i);
            backtrackCombinationSum3(i + 1, k, n - i, current, result);
            current.pop_back();
        }
    }
}

public:
    /**
     * LeetCode 131 - 分割回文串

```

```

* 题目链接: https://leetcode.com/problems/palindrome-partitioning/
* 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。
*
* 方法: 回溯 + 动态规划预处理
* 时间复杂度: O(n * 2^n)
* 空间复杂度: O(n^2)
*/
static vector<vector<string>> partition(string s) {
    int n = s.length();
    vector<vector<bool>> dp(n, vector<bool>(n, false));
    vector<vector<string>> result;
    vector<string> current;

    // 预处理回文信息
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            if (s[i] == s[j] && (i - j <= 2 || dp[j+1][i-1])) {
                dp[j][i] = true;
            }
        }
    }

    backtrackPartition(s, 0, dp, current, result);
    return result;
}

private:
    static void backtrackPartition(string& s, int start, vector<vector<bool>>& dp,
                                   vector<string>& current, vector<vector<string>>& result) {
        if (start == s.length()) {
            result.push_back(current);
            return;
        }

        for (int i = start; i < s.length(); i++) {
            if (dp[start][i]) {
                current.push_back(s.substr(start, i - start + 1));
                backtrackPartition(s, i + 1, dp, current, result);
                current.pop_back();
            }
        }
    }
}

```

```

public:
    /**
     * LeetCode 93 - 复原 IP 地址
     * 题目链接: https://leetcode.com/problems/restore-ip-addresses/
     * 给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。
     *
     * 方法: 回溯
     * 时间复杂度: O(1) - 固定长度
     * 空间复杂度: O(1)
     */
    static vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        vector<string> current;
        backtrackRestoreIP(s, 0, current, result);
        return result;
    }

private:
    static void backtrackRestoreIP(string& s, int start, vector<string>& current, vector<string>& result) {
        if (current.size() == 4 && start == s.length()) {
            result.push_back(current[0] + "." + current[1] + "." + current[2] + "." +
                current[3]);
            return;
        }

        if (current.size() == 4 || start == s.length()) {
            return;
        }

        for (int len = 1; len <= 3 && start + len <= s.length(); len++) {
            string segment = s.substr(start, len);

            // 检查段是否有效
            if (segment.length() > 1 && segment[0] == '0') {
                continue;
            }

            int num = stoi(segment);
            if (num > 255) {
                continue;
            }
        }
    }

```

```

        current.push_back(segment);
        backtrackRestoreIP(s, start + len, current, result);
        current.pop_back();
    }
}

};

class PerformanceTester {
public:
    static void testSubsets() {
        cout << "==== 子集问题性能测试 ===" << endl;

        vector<int> nums = {1, 2, 3, 4};

        auto start = chrono::high_resolution_clock::now();
        auto result = BitAlgorithmApplications::subsets(nums);
        auto time = chrono::duration_cast<chrono::microseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "子集问题: 输入大小=" << nums.size() << ", 结果数量=" << result.size()
            << ", 耗时=" << time << " μs" << endl;
    }

    static void testPermutations() {
        cout << "\n==== 全排列性能测试 ===" << endl;

        vector<int> nums = {1, 2, 3, 4, 5};

        auto start = chrono::high_resolution_clock::now();
        auto result = BitAlgorithmApplications::permute(nums);
        auto time = chrono::duration_cast<chrono::milliseconds>(
            chrono::high_resolution_clock::now() - start).count();

        cout << "全排列: 输入大小=" << nums.size() << ", 结果数量=" << result.size()
            << ", 耗时=" << time << " ms" << endl;
    }

    static void runUnitTests() {
        cout << "==== 位算法应用单元测试 ===" << endl;

        // 测试子集
        vector<int> nums = {1, 2, 3};
        auto subsets_result = BitAlgorithmApplications::subsets(nums);
    }
}

```

```

assert(subsets_result.size() == 8);

// 测试全排列
auto permute_result = BitAlgorithmApplications::permute(nums);
assert(permute_result.size() == 6);

cout << "所有单元测试通过!" << endl;
}

static void complexityAnalysis() {
    cout << "\n==== 复杂度分析 ===" << endl;

    vector<pair<string, string>> algorithms = {
        {"subsets", "O(n*2^n), O(n*2^n)"},  

        {"subsetsWithDup", "O(n*2^n), O(n*2^n)"},  

        {"permute", "O(n!), O(n)"},  

        {"permuteUnique", "O(n!), O(n)"},  

        {"combine", "O(C(n, k)), O(k)"},  

        {"combinationSum", "O(n^target), O(target)"},  

        {"combinationSum2", "O(2^n), O(n)"},  

        {"combinationSum3", "O(C(9, k)), O(k)"},  

        {"partition", "O(n*2^n), O(n^2)"},  

        {"restoreIpAddresses", "O(1), O(1)"}
    };

    for (auto& algo : algorithms) {
        cout << algo.first << ": 时间复杂度=" << algo.second << endl;
    }
}

int main() {
    cout << "位算法应用实现" << endl;
    cout << "包含 LeetCode 多个位算法应用相关题目的解决方案" << endl;
    cout << "===== " << endl;

    // 运行单元测试
    PerformanceTester::runUnitTests();

    // 运行性能测试
    PerformanceTester::testSubsets();
    PerformanceTester::testPermutations();
}

```

```

// 复杂度分析
PerformanceTester::complexityAnalysis();

// 示例使用
cout << "\n==> 示例使用 ==>" << endl;

// 子集示例
vector<int> nums = {1, 2, 3};
auto subsets_result = BitAlgorithmApplications::subsets(nums);
cout << "子集示例([1, 2, 3]): 共" << subsets_result.size() << "个子集" << endl;

// 全排列示例
auto permute_result = BitAlgorithmApplications::permute(nums);
cout << "全排列示例([1, 2, 3]): 共" << permute_result.size() << "个排列" << endl;

// 组合示例
auto combine_result = BitAlgorithmApplications::combine(4, 2);
cout << "组合示例(C(4, 2)): 共" << combine_result.size() << "个组合" << endl;

return 0;
}

```

=====

文件: Code22_BitAlgorithmApplications.java

=====

```

package class032;

import java.util.*;

/**
 * 位算法实际应用和工程场景
 * 题目来源: 实际工程问题, 系统设计, 性能优化
 * 包含位算法在真实场景中的应用案例
 *
 * 解题思路:
 * 方法 1: 位运算优化数据库查询
 * 方法 2: 位压缩存储大规模数据
 * 方法 3: 位操作加速图像处理
 * 方法 4: 位掩码实现权限系统
 *
 * 时间复杂度分析:
 * 方法 1: O(1) - 常数时间查询

```

- * 方法 2: $O(n)$ - 线性处理
- * 方法 3: $O(w \times h)$ - 图像尺寸相关
- * 方法 4: $O(1)$ - 权限检查
- *
- * 空间复杂度分析:
 - * 方法 1: $O(1)$ - 原地操作
 - * 方法 2: $O(n/32)$ - 压缩存储
 - * 方法 3: $O(w \times h)$ - 图像存储
 - * 方法 4: $O(1)$ - 常数空间
- *
- * 工程化考量:
 - * 1. 实际性能: 真实环境测试
 - * 2. 可维护性: 清晰的接口设计
 - * 3. 扩展性: 支持功能扩展
 - * 4. 兼容性: 跨平台兼容
- */

```
public class Code22_BitAlgorithmApplications {  
  
    /**  
     * 应用 1: 权限管理系统  
     * 使用位掩码实现细粒度权限控制  
     * 实际应用: 用户权限管理, 角色权限控制  
     */  
  
    public static class PermissionSystem {  
        // 权限定义  
        public static final int READ = 1 << 0;      // 0001  
        public static final int WRITE = 1 << 1;     // 0010  
        public static final int EXECUTE = 1 << 2; // 0100  
        public static final int DELETE = 1 << 3; // 1000  
  
        // 组合权限  
        public static final int READ_WRITE = READ | WRITE;  
        public static final int FULL_ACCESS = READ | WRITE | EXECUTE | DELETE;  
  
        private int userPermissions;  
  
        public PermissionSystem(int initialPermissions) {  
            this.userPermissions = initialPermissions;  
        }  
  
        /**  
         * 检查是否具有某个权限  
         */  
    }
```

```
/*
public boolean hasPermission(int permission) {
    return (userPermissions & permission) == permission;
}

/***
 * 添加权限
 */
public void addPermission(int permission) {
    userPermissions |= permission;
}

/***
 * 移除权限
 */
public void removePermission(int permission) {
    userPermissions &= ~permission;
}

/***
 * 切换权限状态
 */
public void togglePermission(int permission) {
    userPermissions ^= permission;
}

/***
 * 检查权限组合
 */
public boolean hasAllPermissions(int... permissions) {
    int combined = 0;
    for (int perm : permissions) {
        combined |= perm;
    }
    return (userPermissions & combined) == combined;
}

/***
 * 检查至少有一个权限
 */
public boolean hasAnyPermission(int... permissions) {
    for (int perm : permissions) {
        if ((userPermissions & perm) != 0) {
```

```
        return true;
    }
}

return false;
}

/***
 * 获取权限列表
 */
public List<String> getPermissionList() {
    List<String> permissions = new ArrayList<>();
    if (hasPermission(READ)) permissions.add("READ");
    if (hasPermission(WRITE)) permissions.add("WRITE");
    if (hasPermission(EXECUTE)) permissions.add("EXECUTE");
    if (hasPermission(DELETE)) permissions.add("DELETE");
    return permissions;
}

/***
 * 权限序列化（存储到数据库）
 */
public int serializePermissions() {
    return userPermissions;
}

/***
 * 权限反序列化（从数据库加载）
 */
public void deserializePermissions(int permissions) {
    this.userPermissions = permissions;
}

@Override
public String toString() {
    return String.format("权限位掩码: %04d (二进制: %s",
                        userPermissions,
                        String.format("%4s",
                        Integer.toBinaryString(userPermissions)).replace(' ', '0')));
}

/***
 * 应用 2: 布隆过滤器
*/
```

```
* 使用位数组实现高效的存在性检查
* 实际应用：缓存系统，垃圾邮件过滤，URL 去重
*/
public static class BloomFilter {
    private final int[] bitArray;
    private final int size;
    private final int[] hashSeeds;

    public BloomFilter(int size, int numHashes) {
        this.size = size;
        this.bitArray = new int[(size + 31) / 32];
        this.hashSeeds = new int[numHashes];

        // 初始化哈希种子
        Random random = new Random(42); // 固定种子保证可重复性
        for (int i = 0; i < numHashes; i++) {
            hashSeeds[i] = random.nextInt();
        }
    }

    /**
     * 添加元素
     */
    public void add(String element) {
        for (int i = 0; i < hashSeeds.length; i++) {
            int hash = hash(element, hashSeeds[i]);
            int index = Math.abs(hash % size);
            setBit(index);
        }
    }

    /**
     * 检查元素是否存在（可能有误判）
     */
    public boolean mightContain(String element) {
        for (int i = 0; i < hashSeeds.length; i++) {
            int hash = hash(element, hashSeeds[i]);
            int index = Math.abs(hash % size);
            if (!getBit(index)) {
                return false;
            }
        }
        return true;
    }
}
```

```
}
```

```
/**  
 * 计算误判率（理论值）  
 */  
public double getFalsePositiveRate(int numElements) {  
    double k = hashSeeds.length;  
    double m = size;  
    double n = numElements;  
    return Math.pow(1 - Math.exp(-k * n / m), k);  
}
```

```
private int hash(String element, int seed) {  
    int hash = seed;  
    for (char c : element.toCharArray()) {  
        hash = hash * 31 + c;  
    }  
    return hash;  
}
```

```
private void setBit(int index) {  
    int arrayIndex = index / 32;  
    int bitIndex = index % 32;  
    bitArray[arrayIndex] |= (1 << bitIndex);  
}
```

```
private boolean getBit(int index) {  
    int arrayIndex = index / 32;  
    int bitIndex = index % 32;  
    return (bitArray[arrayIndex] & (1 << bitIndex)) != 0;  
}
```

```
/**  
 * 获取内存使用情况  
 */  
public int getMemoryUsage() {  
    return bitArray.length * 4; // 字节数  
}
```

```
/**  
 * 清空过滤器  
 */  
public void clear() {
```

```
        Arrays.fill(bitArray, 0);
    }
}

/***
 * 应用 3: 图像处理 - 二值图像压缩
 * 使用位运算加速二值图像处理
 * 实际应用: OCR 预处理, 图像二值化, 边缘检测
 */
public static class BinaryImageProcessor {
    private final int width;
    private final int height;
    private final int[] bitData;

    public BinaryImageProcessor(int width, int height) {
        this.width = width;
        this.height = height;
        this.bitData = new int[(width * height + 31) / 32];
    }

    /**
     * 设置像素值
     */
    public void setPixel(int x, int y, boolean value) {
        if (x < 0 || x >= width || y < 0 || y >= height) {
            throw new IllegalArgumentException("坐标超出范围");
        }

        int index = y * width + x;
        int arrayIndex = index / 32;
        int bitIndex = index % 32;

        if (value) {
            bitData[arrayIndex] |= (1 << bitIndex);
        } else {
            bitData[arrayIndex] &= ~(1 << bitIndex);
        }
    }

    /**
     * 获取像素值
     */
    public boolean getPixel(int x, int y) {
```

```

        int index = y * width + x;
        int arrayIndex = index / 32;
        int bitIndex = index % 32;
        return (bitData[arrayIndex] & (1 << bitIndex)) != 0;
    }

/***
 * 图像膨胀操作（形态学处理）
 */
public void dilate() {
    BinaryImageProcessor result = new BinaryImageProcessor(width, height);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (getPixel(x, y)) {
                // 设置 3x3 邻域
                for (int dy = -1; dy <= 1; dy++) {
                    for (int dx = -1; dx <= 1; dx++) {
                        int nx = x + dx, ny = y + dy;
                        if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                            result.setPixel(nx, ny, true);
                        }
                    }
                }
            }
        }
    }

    // 复制结果
    System.arraycopy(result.bitData, 0, bitData, 0, bitData.length);
}

/***
 * 图像腐蚀操作（形态学处理）
 */
public void erode() {
    BinaryImageProcessor result = new BinaryImageProcessor(width, height);

    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            boolean allTrue = true;

            // 检查 3x3 邻域

```

```
        for (int dy = -1; dy <= 1; dy++) {
            for (int dx = -1; dx <= 1; dx++) {
                if (!getPixel(x + dx, y + dy)) {
                    allTrue = false;
                    break;
                }
            }
            if (!allTrue) break;
        }

        if (allTrue) {
            result.setPixel(x, y, true);
        }
    }

    // 复制结果
    System.arraycopy(result.bitData, 0, bitData, 0, bitData.length);
}

/***
 * 图像反转
 */
public void invert() {
    for (int i = 0; i < bitData.length; i++) {
        bitData[i] = ~bitData[i];
    }
}

/***
 * 统计前景像素数量
 */
public int countForegroundPixels() {
    int count = 0;
    for (int value : bitData) {
        count += Integer.bitCount(value);
    }
    return count;
}

/***
 * 获取压缩比
*/

```

```
public double getCompressionRatio() {
    int originalSize = width * height; // 每个像素 1 字节
    int compressedSize = bitData.length * 4; // 每个 int32 位存储 32 个像素
    return (double) originalSize / compressedSize;
}

/**
 * 转换为二维布尔数组（用于显示）
 */
public boolean[][] toBooleanArray() {
    boolean[][] result = new boolean[height][width];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            result[y][x] = getPixel(x, y);
        }
    }
    return result;
}

}

/***
 * 应用 4：数据库查询优化 - 位索引
 * 使用位运算加速数据库查询
 * 实际应用：大数据分析，实时查询，OLAP 系统
 */
public static class BitmapIndex {
    private final Map<String, int[]> index;
    private final int recordCount;

    public BitmapIndex(int recordCount) {
        this.recordCount = recordCount;
        this.index = new HashMap<>();
    }

}

/***
 * 为某个值创建位图索引
 */
public void addValue(String value, int recordIndex) {
    if (recordIndex < 0 || recordIndex >= recordCount) {
        throw new IllegalArgumentException("记录索引超出范围");
    }

    int[] bitmap = index.computeIfAbsent(value, k -> new int[(recordCount + 31) / 32]);
}
```

```

        int arrayIndex = recordIndex / 32;
        int bitIndex = recordIndex % 32;
        bitmap[arrayIndex] |= (1 << bitIndex);
    }

/***
 * 查询等于某个值的记录
 */
public List<Integer> queryEquals(String value) {
    int[] bitmap = index.get(value);
    if (bitmap == null) {
        return Collections.emptyList();
    }

    List<Integer> result = new ArrayList<>();
    for (int i = 0; i < recordCount; i++) {
        int arrayIndex = i / 32;
        int bitIndex = i % 32;
        if ((bitmap[arrayIndex] & (1 << bitIndex)) != 0) {
            result.add(i);
        }
    }
    return result;
}

/***
 * 查询在多个值中的记录 (OR 操作)
 */
public List<Integer> queryIn(String... values) {
    if (values.length == 0) {
        return Collections.emptyList();
    }

    int[] combined = new int[(recordCount + 31) / 32];
    for (String value : values) {
        int[] bitmap = index.get(value);
        if (bitmap != null) {
            for (int i = 0; i < combined.length; i++) {
                combined[i] |= bitmap[i];
            }
        }
    }
}

```

```
List<Integer> result = new ArrayList<>();
for (int i = 0; i < recordCount; i++) {
    int arrayIndex = i / 32;
    int bitIndex = i % 32;
    if ((combined[arrayIndex] & (1 << bitIndex)) != 0) {
        result.add(i);
    }
}
return result;
}

/**
 * 查询同时满足多个值的记录（AND 操作）
 */
public List<Integer> queryAnd(String... values) {
    if (values.length == 0) {
        return Collections.emptyList();
    }

    int[] combined = null;
    for (String value : values) {
        int[] bitmap = index.get(value);
        if (bitmap == null) {
            return Collections.emptyList(); // 某个值不存在
        }

        if (combined == null) {
            combined = bitmap.clone();
        } else {
            for (int i = 0; i < combined.length; i++) {
                combined[i] &= bitmap[i];
            }
        }
    }

    List<Integer> result = new ArrayList<>();
    for (int i = 0; i < recordCount; i++) {
        int arrayIndex = i / 32;
        int bitIndex = i % 32;
        if ((combined[arrayIndex] & (1 << bitIndex)) != 0) {
            result.add(i);
        }
    }
}
```

```
        return result;
    }

/**
 * 获取索引大小
 */
public int getIndexSize() {
    int size = 0;
    for (int[] bitmap : index.values()) {
        size += bitmap.length * 4; // 每个 int 占 4 字节
    }
    return size;
}

/**
 * 获取索引统计信息
 */
public Map<String, Object> getStatistics() {
    Map<String, Object> stats = new HashMap<>();
    stats.put("记录数量", recordCount);
    stats.put("不同值数量", index.size());
    stats.put("索引大小(字节)", getIndexSize());

    int totalBitsSet = 0;
    for (int[] bitmap : index.values()) {
        for (int value : bitmap) {
            totalBitsSet += Integer.bitCount(value);
        }
    }
    stats.put("设置的位总数", totalBitsSet);

    return stats;
}

/**
 * 单元测试方法
 */
public static void runTests() {
    System.out.println("==== 位算法实际应用 - 单元测试 ====");
    // 测试权限系统
    System.out.println("权限系统测试:");
}
```

```

PermissionSystem ps = new PermissionSystem(PermissionSystem.READ);
System.out.println("初始权限: " + ps);

ps.addPermission(PermissionSystem.WRITE);
System.out.println("添加 WRITE 权限后: " + ps);
System.out.println("是否有 READ 权限: " + ps.hasPermission(PermissionSystem.READ));
System.out.println("是否有 EXECUTE 权限: " + ps.hasPermission(PermissionSystem.EXECUTE));

// 测试布隆过滤器
System.out.println("\n 布隆过滤器测试:");
BloomFilter bf = new BloomFilter(1000, 3);
bf.add("hello");
bf.add("world");
System.out.println("包含'hello': " + bf.mightContain("hello"));
System.out.println("包含'test': " + bf.mightContain("test"));
System.out.println("误判率(100 元素): " + bf.getFalsePositiveRate(100));
System.out.println("内存使用: " + bf.getMemoryUsage() + " 字节");

// 测试图像处理
System.out.println("\n 图像处理测试:");
BinaryImageProcessor image = new BinaryImageProcessor(10, 10);
image.setPixel(5, 5, true);
image.setPixel(5, 6, true);
System.out.println("前景像素数量: " + image.countForegroundPixels());
System.out.println("压缩比: " + image.getCompressionRatio());

// 测试位图索引
System.out.println("\n 位图索引测试:");
BitmapIndex index = new BitmapIndex(100);
index.addValue("男", 1);
index.addValue("男", 3);
index.addValue("女", 2);
index.addValue("女", 4);

System.out.println("性别=男的记录: " + index.queryEquals("男"));
System.out.println("性别=男的记录: " + index.queryEquals("女"));
System.out.println("索引统计: " + index.getStatistics());
}

/**
 * 性能测试方法
 */
public static void performanceTest() {

```

```
System.out.println("\n== 性能测试 ==");

// 测试权限系统性能
long startTime = System.nanoTime();
PermissionSystem ps = new PermissionSystem(PermissionSystem.FULL_ACCESS);
for (int i = 0; i < 1000000; i++) {
    ps.hasPermission(PermissionSystem.READ);
}
long time1 = System.nanoTime() - startTime;
System.out.printf("权限检查性能: %d ns/百万次%n", time1 / 1000);

// 测试布隆过滤器性能
startTime = System.nanoTime();
BloomFilter bf = new BloomFilter(10000, 5);
for (int i = 0; i < 1000; i++) {
    bf.add("element" + i);
}
for (int i = 0; i < 1000; i++) {
    bf.mightContain("element" + i);
}
long time2 = System.nanoTime() - startTime;
System.out.printf("布隆过滤器性能: %d ns%n", time2);

// 测试位图索引性能
startTime = System.nanoTime();
BitmapIndex index = new BitmapIndex(100000);
for (int i = 0; i < 100000; i++) {
    index.addValue("value" + (i % 10), i);
}
List<Integer> result = index.queryEquals("value5");
long time3 = System.nanoTime() - startTime;
System.out.printf("位图索引查询性能: %d ns, 结果数量: %d%n", time3, result.size());
}

/***
 * 实际应用场景分析
 */
public static void applicationAnalysis() {
    System.out.println("\n== 实际应用场景分析 ==");

    System.out.println("1. 权限管理系统应用场景:");
    System.out.println("    - 用户角色权限控制");
    System.out.println("    - API 访问权限管理");
}
```

```
System.out.println(" - 文件系统权限控制");  
  
System.out.println("\n2. 布隆过滤器应用场景:");  
System.out.println(" - 缓存穿透防护");  
System.out.println(" - 垃圾邮件过滤");  
System.out.println(" - 大规模数据去重");  
  
System.out.println("\n3. 图像处理应用场景:");  
System.out.println(" - OCR 文字识别预处理");  
System.out.println(" - 医学图像分析");  
System.out.println(" - 工业视觉检测");  
  
System.out.println("\n4. 位图索引应用场景:");  
System.out.println(" - 大数据分析查询");  
System.out.println(" - 数据仓库 OLAP 系统");  
System.out.println(" - 实时报表生成");  
  
System.out.println("\n5. 性能优势:");  
System.out.println(" - 内存使用减少 80-90%");  
System.out.println(" - 查询速度提升 10-100 倍");  
System.out.println(" - 适合大规模数据处理");  
}
```

```
public static void main(String[] args) {  
    System.out.println("位算法实际应用和工程场景");  
    System.out.println("包含权限管理、布隆过滤器、图像处理、数据库索引等实际应用");  
  
    // 运行单元测试  
    runTests();  
  
    // 性能测试  
    performanceTest();  
  
    // 应用场景分析  
    applicationAnalysis();  
  
    // 工程化建议  
    System.out.println("\n==== 工程化建议 ====");  
    System.out.println("1. 选择合适的应用场景");  
    System.out.println("2. 进行充分的性能测试");  
    System.out.println("3. 考虑内存和 CPU 的平衡");  
    System.out.println("4. 设计清晰的 API 接口");  
    System.out.println("5. 编写详细的文档说明");
```

```
    }  
}
```

```
=====文件: Code22_BitAlgorithmApplications.py=====
```

```
"""
```

位算法应用实现

包含 LeetCode 多个位算法应用相关题目的解决方案

题目列表:

1. LeetCode 78 - 子集
2. LeetCode 90 - 子集 II
3. LeetCode 46 - 全排列
4. LeetCode 47 - 全排列 II
5. LeetCode 77 - 组合
6. LeetCode 39 - 组合总和
7. LeetCode 40 - 组合总和 II
8. LeetCode 216 - 组合总和 III
9. LeetCode 131 - 分割回文串
10. LeetCode 93 - 复原 IP 地址

时间复杂度分析:

- 回溯算法: $O(2^n)$ 到 $O(n!)$
- 空间复杂度: $O(n)$ 到 $O(n^2)$

工程化考量:

1. 位集优化: 使用位集优化回溯算法
2. 状态压缩: 使用位运算压缩状态空间
3. 剪枝优化: 使用位运算进行高效剪枝
4. 去重处理: 使用位掩码进行重复检测

```
"""
```

```
import time  
from typing import List  
import sys  
  
class BitAlgorithmApplications:  
    """位算法应用类"""\n  
    @staticmethod  
    def subsets(nums: List[int]) -> List[List[int]]:
```

```
"""
```

LeetCode 78 - 子集

给定一组不含重复元素的整数数组 `nums`, 返回该数组所有可能的子集（幂集）。

方法：位运算枚举

时间复杂度： $O(n * 2^n)$

空间复杂度： $O(n * 2^n)$

原理：使用二进制位表示每个元素是否在子集中

```
"""
```

```
n = len(nums)
```

```
total = 1 << n
```

```
result = []
```

```
for mask in range(total):
```

```
    subset = []
```

```
    for i in range(n):
```

```
        if mask & (1 << i):
```

```
            subset.append(nums[i])
```

```
    result.append(subset)
```

```
return result
```

```
@staticmethod
```

```
def subsets_with_dup(nums: List[int]) -> List[List[int]]:
```

```
"""
```

LeetCode 90 - 子集 II

给定一个可能包含重复元素的整数数组 `nums`, 返回该数组所有可能的子集（幂集）。

方法：位运算 + 排序去重

时间复杂度： $O(n * 2^n)$

空间复杂度： $O(n * 2^n)$

```
"""
```

```
nums.sort()
```

```
n = len(nums)
```

```
total = 1 << n
```

```
result_set = set()
```

```
for mask in range(total):
```

```
    subset = []
```

```
    for i in range(n):
```

```
        if mask & (1 << i):
```

```
            subset.append(nums[i])
```

```

# 使用元组作为集合的键
result_set.add(tuple(subset))

return [list(subset) for subset in result_set]

@staticmethod
def permute(nums: List[int]) -> List[List[int]]:
    """
    LeetCode 46 - 全排列
    给定一个没有重复数字的序列，返回其所有可能的全排列。
    方法：回溯 + 位掩码
    时间复杂度：O(n!)
    空间复杂度：O(n)
    """
    result = []
    current = []
    used = [False] * len(nums)

    def backtrack():
        if len(current) == len(nums):
            result.append(current[:])
            return

        for i in range(len(nums)):
            if not used[i]:
                used[i] = True
                current.append(nums[i])
                backtrack()
                current.pop()
                used[i] = False

    backtrack()
    return result

```

```

@staticmethod
def permute_unique(nums: List[int]) -> List[List[int]]:
    """
    LeetCode 47 - 全排列 II
    给定一个可包含重复数字的序列，返回所有不重复的全排列。
    
```

方法：回溯 + 排序剪枝
 时间复杂度：O(n!)

```

空间复杂度: O(n)
"""

nums.sort()
result = []
current = []
used = [False] * len(nums)

def backtrack():
    if len(current) == len(nums):
        result.append(current[:])
        return

    for i in range(len(nums)):
        if used[i] or (i > 0 and nums[i] == nums[i-1] and not used[i-1]):
            continue

        used[i] = True
        current.append(nums[i])
        backtrack()
        current.pop()
        used[i] = False

backtrack()
return result

```

```

@staticmethod
def combine(n: int, k: int) -> List[List[int]]:
    """

    LeetCode 77 - 组合
    给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

```

方法: 回溯

时间复杂度: $O(C(n, k))$

空间复杂度: $O(k)$

"""

```

result = []
current = []

def backtrack(start: int):
    if len(current) == k:
        result.append(current[:])
        return

    for i in range(start, n+1):
        current.append(i)
        backtrack(i+1)
        current.pop()

```

```

        for i in range(start, n + 1):
            current.append(i)
            backtrack(i + 1)
            current.pop()

    backtrack(1)
    return result

@staticmethod
def combination_sum(candidates: List[int], target: int) -> List[List[int]]:
    """
    LeetCode 39 - 组合总和
    给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
    """

```

方法：回溯

时间复杂度： $O(n^{\text{target}})$

空间复杂度： $O(\text{target})$

"""

result = []

current = []

candidates.sort()

def backtrack(start: int, remaining: int):

if remaining == 0:

result.append(current[:])

return

for i in range(start, len(candidates)):

if candidates[i] > remaining:

break

current.append(candidates[i])

backtrack(i, remaining - candidates[i])

current.pop()

backtrack(0, target)

return result

@staticmethod

def combination_sum2(candidates: List[int], target: int) -> List[List[int]]:

"""

LeetCode 40 - 组合总和 II

给定一个数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

方法：回溯 + 排序剪枝

时间复杂度： $O(2^n)$

空间复杂度： $O(n)$

"""

```
result = []
```

```
current = []
```

```
candidates.sort()
```

```
def backtrack(start: int, remaining: int):
```

```
    if remaining == 0:
```

```
        result.append(current[:])
```

```
        return
```

```
    for i in range(start, len(candidates)):
```

```
        if candidates[i] > remaining:
```

```
            break
```

```
        if i > start and candidates[i] == candidates[i-1]:
```

```
            continue
```

```
        current.append(candidates[i])
```

```
        backtrack(i + 1, remaining - candidates[i])
```

```
        current.pop()
```

```
backtrack(0, target)
```

```
return result
```

```
@staticmethod
```

```
def combination_sum3(k: int, n: int) -> List[List[int]]:
```

```
"""
```

LeetCode 216 - 组合总和 III

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

方法：回溯

时间复杂度： $O(C(9, k))$

空间复杂度： $O(k)$

```
"""
```

```
result = []
```

```

current = []

def backtrack(start: int, remaining: int):
    if len(current) == k and remaining == 0:
        result.append(current[:])
        return

    if len(current) > k or remaining < 0:
        return

    for i in range(start, 10):
        current.append(i)
        backtrack(i + 1, remaining - i)
        current.pop()

backtrack(1, n)
return result

```

```

@staticmethod
def partition(s: str) -> List[List[str]]:
    """
    LeetCode 131 - 分割回文串
    给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。
    
```

方法：回溯 + 动态规划预处理

时间复杂度： $O(n * 2^n)$

空间复杂度： $O(n^2)$

"""

n = len(s)

预处理回文信息

dp = [[False] * n for _ in range(n)]

for i in range(n):

for j in range(i + 1):

if s[i] == s[j] and (i - j <= 2 or dp[j+1][i-1]):

dp[j][i] = True

result = []

current = []

def backtrack(start: int):

if start == n:

result.append(current[:])

```

        return

    for i in range(start, n):
        if dp[start][i]:
            current.append(s[start:i+1])
            backtrack(i + 1)
            current.pop()

    backtrack(0)
    return result

```

```

@staticmethod
def restore_ip_addresses(s: str) -> List[str]:
    """

```

LeetCode 93 - 复原 IP 地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

方法：回溯

时间复杂度：O(1) - 固定长度

空间复杂度：O(1)

"""

```
result = []
```

```
current = []
```

```
def backtrack(start: int):
```

```
    if len(current) == 4 and start == len(s):
        result.append('.'.join(current))
        return
```

```
    if len(current) == 4 or start == len(s):
        return
```

```
    for length in range(1, 4):
```

```
        if start + length > len(s):
            break
```

```
        segment = s[start:start+length]
```

检查段是否有效

```
        if len(segment) > 1 and segment[0] == '0':
            continue
```

```
        num = int(segment)
```

```
if num > 255:  
    continue  
  
    current.append(segment)  
    backtrack(start + length)  
    current.pop()  
  
backtrack(0)  
return result  
  
  
class PerformanceTester:  
    """性能测试工具类"""  
  
    @staticmethod  
    def test_subsets():  
        """测试子集问题性能"""  
        print("== 子集问题性能测试 ==")  
  
        nums = [1, 2, 3, 4]  
  
        start = time.time()  
        result = BitAlgorithmApplications.subsets(nums)  
        elapsed = (time.time() - start) * 1e6 # 微秒  
  
        print(f"子集问题: 输入大小={len(nums)}, 结果数量={len(result)}, 耗时={elapsed:.2f} μs")  
  
    @staticmethod  
    def test_permutations():  
        """测试全排列性能"""  
        print("\n== 全排列性能测试 ==")  
  
        nums = [1, 2, 3, 4, 5]  
  
        start = time.time()  
        result = BitAlgorithmApplications.permute(nums)  
        elapsed = (time.time() - start) * 1000 # 毫秒  
  
        print(f"全排列: 输入大小={len(nums)}, 结果数量={len(result)}, 耗时={elapsed:.2f} ms")  
  
    @staticmethod  
    def run_unit_tests():  
        """运行单元测试"""
```

```

print("== 位算法应用单元测试 ==")

# 测试子集
nums = [1, 2, 3]
subsets_result = BitAlgorithmApplications.subsets(nums)
assert len(subsets_result) == 8

# 测试全排列
permute_result = BitAlgorithmApplications.permute(nums)
assert len(permute_result) == 6

print("所有单元测试通过!")

@staticmethod
def complexity_analysis():
    """复杂度分析"""
    print("\n== 复杂度分析 ==")

    algorithms = {
        "subsets": ("O(n*2^n)", "O(n*2^n)"),
        "subsets_with_dup": ("O(n*2^n)", "O(n*2^n)"),
        "permute": ("O(n!)", "O(n!)"),
        "permute_unique": ("O(n!)", "O(n!)"),
        "combine": ("O(C(n, k))", "O(k)"),
        "combination_sum": ("O(n^target)", "O(target)"),
        "combination_sum2": ("O(2^n)", "O(n)"),
        "combination_sum3": ("O(C(9, k))", "O(k)"),
        "partition": ("O(n*2^n)", "O(n^2)"),
        "restore_ip_addresses": ("O(1)", "O(1)")
    }

    for name, (time_complexity, space_complexity) in algorithms.items():
        print(f"{name}: 时间复杂度={time_complexity}, 空间复杂度={space_complexity}")

def main():
    """主函数"""
    print("位算法应用实现")
    print("包含 LeetCode 多个位算法应用相关题目的解决方案")
    print("==" * 50)

    # 运行单元测试
    PerformanceTester.run_unit_tests()

```

```
# 运行性能测试
PerformanceTester. test_subsets()
PerformanceTester. test_permutations()

# 复杂度分析
PerformanceTester. complexity_analysis()

# 示例使用
print("\n==== 示例使用 ====")

# 子集示例
nums = [1, 2, 3]
subsets_result = BitAlgorithmApplications. subsets(nums)
print(f"子集示例([1, 2, 3]): 共{len(subsets_result)}个子集")

# 全排列示例
permute_result = BitAlgorithmApplications. permute(nums)
print(f"全排列示例([1, 2, 3]): 共{len(permute_result)}个排列")

# 组合示例
combine_result = BitAlgorithmApplications. combine(4, 2)
print(f"组合示例(C(4, 2)): 共{len(combine_result)}个组合")

# 组合总和示例
candidates = [2, 3, 6, 7]
target = 7
combination_result = BitAlgorithmApplications. combination_sum(candidates, target)
print(f"组合总和示例({candidates}, target={target}): 共{len(combination_result)}个解")

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