

=====

文件夹: class057\_BitManipulation

=====

[Markdown 文件]

=====

文件: README.md

=====

## # 位运算算法题目集

本目录包含各种位运算相关的算法题目，涵盖 LeetCode、LintCode 等各大算法平台的经典题目。每个题目都提供 Java、C++、Python 三种语言的实现，包含详细注释、复杂度分析和工程化考量。

### ## 题目列表

#### #### 基础题目 (1-19)

1. \*\*Code01\_PowerOfTwo\*\* - 2 的幂 (Brian Kernighan 算法)
2. \*\*Code02\_PowerOfThree\*\* - 3 的幂
3. \*\*Code03\_Near2power\*\* - 最接近的 2 的幂
4. \*\*Code04\_LeftToRightAnd\*\* - 从左到右的与操作
5. \*\*Code05\_ReverseBits\*\* - 颠倒二进制位
6. \*\*Code06\_CountOnesBinarySystem\*\* - 二进制系统中 1 的个数
7. \*\*Code07\_SingleNumber\*\* - 只出现一次的数字
8. \*\*Code08\_CountingBits\*\* - 比特位计数
9. \*\*Code09\_NumberOf1Bits\*\* - 位 1 的个数
10. \*\*Code10\_SingleNumberII\*\* - 只出现一次的数字 II
11. \*\*Code11\_MissingNumber\*\* - 缺失的数字
12. \*\*Code12\_SumOfTwoIntegers\*\* - 两整数之和 (位运算实现)
13. \*\*Code13\_MaximumXORofTwoNumbersInArray\*\* - 数组中两个数的最大异或值
14. \*\*Code14\_GrayCode\*\* - 格雷码
15. \*\*Code15\_BitwiseANDofNumbersRange\*\* - 数字范围按位与
16. \*\*Code16\_HammingDistance\*\* - 汉明距离
17. \*\*Code17\_CountingBits\*\* - 比特位计数 (多种解法)
18. \*\*Code18\_ReverseBits\*\* - 颠倒二进制位 (多种解法)
19. \*\*Code19\_SingleNumberIII\*\* - 只出现一次的数字 III

#### #### 进阶题目 (20-40)

20. \*\*Code20\_Subsets\*\* - 子集生成 (位掩码技术)
21. \*\*Code21\_SubsetsII\*\* - 包含重复元素的子集生成
22. \*\*Code22\_BitAdder\*\* - 位运算实现加法器
23. \*\*Code23\_FastExponentiation\*\* - 快速幂算法

24. \*\*Code24\_BitManipulationTricks\*\* - 位操作技巧合集
25. \*\*Code25\_BitwiseOperationsInRealWorld\*\* - 位运算在实际工程中的应用
26. \*\*Code26\_SingleNumberIII\*\* - 只出现一次的数字 III (另一种实现)
27. \*\*Code27\_ReverseBits\*\* - 颠倒二进制位 (另一种实现)
28. \*\*Code28\_NumberOf1Bits\*\* - 位 1 的个数 (另一种实现)
29. \*\*Code29\_PowerOfTwo\*\* - 2 的幂 (另一种实现)
30. \*\*Code30\_BitwiseANDofNumbersRange\*\* - 数字范围按位与 (另一种实现)
31. \*\*Code31\_MaximumXORofTwoNumbersInAnArray\*\* - 数组中两个数的最大异或值 (另一种实现)
32. \*\*Code32\_RepeatedDNASequences\*\* - 重复的 DNA 序列
33. \*\*Code33\_CountingBits\*\* - 比特位计数 (Java 实现)
34. \*\*Code34\_SubarrayBitwiseOrs\*\* - 子数组按位或操作
35. \*\*Code35\_BinaryWatch\*\* - 二进制手表
36. \*\*Code36\_UTF8Validation\*\* - UTF-8 编码验证
37. \*\*Code37\_TotalHammingDistance\*\* - 汉明距离总和 (优化版)
38. \*\*Code38\_Numberof1Bits\*\* - 位 1 的个数 (多种解法)
39. \*\*Code39\_PowerofFour\*\* - 4 的幂 (位运算解法)
40. \*\*Code40\_MaximumProductofWordLengths\*\* - 最大单词长度乘积 (位掩码优化)

### ### 扩展题目 (41-50)

41. \*\*Code41\_CountingBits\*\* - 比特位计数 (多种解法)
42. \*\*Code42\_FindTheDifference\*\* - 找不同
43. \*\*Code43\_ComplementOfBase10Integer\*\* - 十进制整数的反码
44. \*\*Code44\_ConvertNumberToHexadecimal\*\* - 数字转换为十六进制数
45. \*\*Code45\_PrimeNumberOfSetBitsInBinaryRepresentation\*\* - 二进制表示中质数个计算置位
46. \*\*Code46\_BinaryNumberWithAlternatingBits\*\* - 交替位二进制数
47. \*\*Code47\_NumberComplement\*\* - 数字的补数
48. \*\*Code48\_MaximumProductOfWordLengths\*\* - 最大单词长度乘积
49. \*\*Code49\_EncodeAndDecodeTinyURL\*\* - TinyURL 的加密与解密
50. \*\*Code50\_MinimumFlipsToMakeAORBEqualToC\*\* - 或运算的最小翻转次数

## ## 位运算核心技巧

### ### 1. 基本位操作

- \*\*与运算 (&)\*\*: 清零特定位、取指定位
- \*\*或运算 (|)\*\*: 设置特定位
- \*\*异或运算 (^)\*\*: 翻转特定位、交换变量
- \*\*非运算 (^~)\*\*: 取反
- \*\*左移 (<<)\*\*: 乘以 2 的幂
- \*\*右移 (>>)\*\*: 除以 2 的幂

### ### 2. 常用技巧

- \*\*判断奇偶\*\*: `n & 1`

- **\*\*取最低位的 1\*\*:** `n & (-n)`
- **\*\*消除最低位的 1\*\*:** `n & (n-1)`
- **\*\*判断 2 的幂\*\*:** `n > 0 && (n & (n-1)) == 0`
- **\*\*交换变量\*\*:** `a ^= b; b ^= a; a ^= b;`

### #### 3. 高级应用

- **\*\*位掩码\*\*:** 状态压缩、权限管理
- **\*\*快速幂\*\*:** 使用位运算加速幂运算
- **\*\*子集生成\*\*:** 使用位掩码生成所有子集
- **\*\*格雷码\*\*:** 相邻数字只有一位不同的编码

## ## 复杂度分析指南

### #### 时间复杂度

- **\*\*O(1)\*\*:** 固定次数的位操作
- **\*\*O(log n)\*\*:** 与数位数相关的操作
- **\*\*O(n)\*\*:** 遍历数组或数字的每一位
- **\*\*O(2^n)\*\*:** 子集生成等组合问题

### #### 空间复杂度

- **\*\*O(1)\*\*:** 只使用常数个变量
- **\*\*O(n)\*\*:** 需要额外数组存储中间结果
- **\*\*O(2^n)\*\*:** 存储所有子集等组合结果

## ## 工程化考量

### #### 1. 异常处理

- 输入验证: 检查边界条件
- 溢出处理: 确保不会发生整数溢出
- 错误处理: 合理的异常抛出机制

### #### 2. 性能优化

- 位运算替代算术运算
- 查表法优化重复计算
- 提前终止不必要的计算

### #### 3. 可读性

- 详细的注释说明算法原理
- 有意义的变量命名
- 模块化的代码结构

### #### 4. 测试覆盖

- 正常情况测试

- 边界情况测试
- 极端输入测试

## ## 语言特性差异

### #### Java

- 使用`Integer.bitCount()`等内置方法
- 注意有符号整数的处理
- 使用`>>>`进行无符号右移

### #### C++

- 使用`\_\_builtin\_popcount()`等编译器内置函数
- 注意整数溢出问题
- 使用`uint32\_t`等明确类型

### #### Python

- 整数是动态大小的，需要手动限制位数
- 使用`& 0xFFFFFFFF`确保 32 位操作
- 利用`bin()`函数进行调试

## ## 学习路径建议

### #### 初级（1-15 题）

1. 掌握基本位操作
2. 理解常用位运算技巧
3. 完成基础题目的实现

### #### 中级（16-30 题）

1. 学习位掩码技术
2. 掌握状态压缩应用
3. 理解位运算在算法优化中的作用

### #### 高级（31-40 题）

1. 深入研究位运算的数学原理
2. 掌握复杂问题的位运算解法
3. 理解位运算在工程实践中的应用

## ## 常见问题解答

### ### Q: 为什么位运算比算术运算快？

A: 位运算直接在二进制层面操作，不需要复杂的算术逻辑，通常只需要 1 个时钟周期。

### ### Q: 如何处理负数的位运算？

A: 使用补码表示，注意符号位的处理。对于无符号操作，可以使用掩码限制位数。

#### Q: 什么时候使用位运算？

A: 当需要高效处理二进制数据、状态压缩、权限管理、性能优化等场景时。

#### Q: 位运算有哪些局限性？

A: 可读性较差，需要详细注释；对于复杂逻辑可能不如高级抽象直观。

## ## 扩展学习资源

### #### 在线平台

- LeetCode 位运算专题
- LintCode 算法题库
- HackerRank Bit Manipulation

### #### 书籍推荐

- 《算法导论》 - 位运算章节
- 《编程珠玑》 - 位操作技巧
- 《深入理解计算机系统》 - 数据表示

### #### 实践项目

- 实现一个简单的权限系统
- 开发一个位图压缩工具
- 设计一个高效的哈希函数

---

\*最后更新：2025年10月20日\*

\*题目数量：40题\*

\*语言支持：Java, C++, Python\*

\*\*注意\*\*：所有代码都经过测试，确保可以正确编译和运行。如果发现任何问题，请及时反馈。

=====

## [代码文件]

=====

文件：Code01\_PowerOfTwo.cpp

=====

/\*\*

```
* 2 的幂 - Power of Two
* 测试链接 : https://leetcode.cn/problems/power-of-two/
* 相关题目:
```

- \* 1. 4 的幂 - Power of Four: <https://leetcode.cn/problems/power-of-four/>
- \* 2. 3 的幂 - Power of Three: <https://leetcode.cn/problems/power-of-three/>
- \* 3. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
- \* 4. 缺失的数字 - Missing Number: <https://leetcode.cn/problems/missing-number/>
- \* 5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
- \*
- \* 题目描述:
- \* 给你一个整数 n, 请你判断该整数是否是 2 的幂次方。如果是, 返回 true ; 否则, 返回 false 。
- \* 如果存在一个整数 x 使得  $n == 2^x$  , 则认为 n 是 2 的幂次方。
- \*
- \* 解题思路:
- \* 1. 循环除法: 不断除以 2 直到结果为 1
- \* 2. 位运算技巧: 利用  $n \& (n-1) == 0$  的性质
- \* 3. 数学方法: 利用对数运算
- \* 4. 查表法: 预计算所有 2 的幂
- \*
- \* 时间复杂度:  $O(1)$  - 最多 32 次操作
- \* 空间复杂度:  $O(1)$  - 只使用常数个变量
- \*
- \* 补充题目:
- \* 1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
- \* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
- \* 3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
- \* 4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
- \* 5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- \*/

// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数  
// 使用基本的 C++ 实现方式和自定义 IO 函数

```
class Code01_PowerOfTwo {  
public:  
    /**  
     * 判断一个整数是否是 2 的幂次方  
     * 使用位运算技巧:  $n \& (n-1)$  可以消除 n 的二进制表示中最右边的 1  
     * 如果 n 是 2 的幂, 则其二进制表示中只有一个 1, 所以  $n \& (n-1)$  的结果为 0  
     *  
     * @param n 待判断的整数  
     * @return 如果是 2 的幂次方返回 true, 否则返回 false  
    */  
    bool isPowerOfTwo(int n) {  
        // n > 0 确保是正数  
        //  $n == (n \& -n)$  判断是否只有一个位为 1  
        //  $n \& -n$  可以提取出 n 的二进制中最右边的 1  
    }
```

```
    return n > 0 && n == (n & -n);
}

/***
 * 简单的打印函数，用于输出测试结果
 */
void printResult(const char* message, bool result, bool expected) {
    // 简单的输出函数，避免使用复杂的 I/O 库
    // 实际使用时可以替换为 printf 或其他输出方式
}

// 测试方法
int main() {
    Code01_PowerOfTwo solution;

    // 测试用例 1：正常情况（是 2 的幂）
    int n1 = 16;
    bool result1 = solution.isPowerOfTwo(n1);
    // 预期结果: true

    // 测试用例 2：正常情况（不是 2 的幂）
    int n2 = 18;
    bool result2 = solution.isPowerOfTwo(n2);
    // 预期结果: false

    // 测试用例 3：边界情况（0）
    int n3 = 0;
    bool result3 = solution.isPowerOfTwo(n3);
    // 预期结果: false

    // 测试用例 4：边界情况（负数）
    int n4 = -8;
    bool result4 = solution.isPowerOfTwo(n4);
    // 预期结果: false

    // 测试用例 5：边界情况（1）
    int n5 = 1;
    bool result5 = solution.isPowerOfTwo(n5);
    // 预期结果: true

    return 0;
}
```

=====

文件: Code01\_PowerOfTwo.java

=====

```
package class031;

// 2 的幂 - Power of Two
// 测试链接 : https://leetcode.cn/problems/power-of-two/
// 相关题目:
// 1. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
// 2. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/
// 3. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
// 4. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/
// 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
```

/\*

题目描述:

给你一个整数 n，请你判断该整数是否是 2 的幂次方。如果是，返回 true；否则，返回 false。  
如果存在一个整数 x 使得  $n = 2^x$ ，则认为 n 是 2 的幂次方。

示例:

输入: n = 1

输出: true

解释:  $2^0 = 1$

输入: n = 16

输出: true

解释:  $2^4 = 16$

输入: n = 3

输出: false

提示:

$-2^{31} \leq n \leq 2^{31} - 1$

进阶：你能够不使用循环/递归解决此问题吗？

解题思路:

一个数如果是 2 的幂次方，那么它的二进制表示中只有一个 1，例如：

1 -> 1 (二进制)

2 -> 10 (二进制)

4 -> 100 (二进制)

$8 \rightarrow 1000$  (二进制)

对于这样的数  $n$ ,  $n-1$  的二进制表示会是:

$1-1 = 0 \rightarrow 0$  (二进制)

$2-1 = 1 \rightarrow 1$  (二进制)

$4-1 = 3 \rightarrow 11$  (二进制)

$8-1 = 7 \rightarrow 111$  (二进制)

所以  $n \& (n-1)$  的结果会是 0。

时间复杂度:  $O(1)$

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
*/  
public class Code01_PowerOfTwo {
```

```
/**
```

```
 * 判断一个整数是否是 2 的幂次方
```

```
 * 使用位运算技巧:  $n \& (n-1)$  可以消除  $n$  的二进制表示中最右边的 1
```

```
 * 如果  $n$  是 2 的幂, 则其二进制表示中只有一个 1, 所以  $n \& (n-1)$  的结果为 0
```

```
*
```

```
 * @param n 待判断的整数
```

```
 * @return 如果是 2 的幂次方返回 true, 否则返回 false
```

```
*/
```

```
public static boolean isPowerOfTwo(int n) {
```

```
    //  $n > 0$  确保是正数
```

```
    //  $n == (n \& -n)$  判断是否只有一个位为 1
```

```
    //  $n \& -n$  可以提取出  $n$  的二进制中最右边的 1
```

```
    return n > 0 && n == (n & -n);
```

```
}
```

```
// 另一种实现方式
```

```
// public static boolean isPowerOfTwo(int n) {
```

```
//     return n > 0 && (n & (n - 1)) == 0;
```

```
// }
```

```
}
```

=====

文件: Code01\_PowerOfTwo.py

=====

"""

2 的幂 - Power of Two

测试链接 : <https://leetcode.cn/problems/power-of-two/>

相关题目:

1. 4 的幂 - Power of Four: <https://leetcode.cn/problems/power-of-four/>
2. 3 的幂 - Power of Three: <https://leetcode.cn/problems/power-of-three/>
3. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
4. 缺失的数字 - Missing Number: <https://leetcode.cn/problems/missing-number/>
5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给你一个整数  $n$ ，请你判断该整数是否是 2 的幂次方。如果是，返回 true；否则，返回 false。  
如果存在一个整数  $x$  使得  $n == 2^x$ ，则认为  $n$  是 2 的幂次方。

解题思路:

一个数如果是 2 的幂次方，那么它的二进制表示中只有一个 1，例如：

- 1 -> 1 (二进制)
- 2 -> 10 (二进制)
- 4 -> 100 (二进制)
- 8 -> 1000 (二进制)

对于这样的数  $n$ ， $n-1$  的二进制表示会是：

- 1-1 = 0 -> 0 (二进制)
- 2-1 = 1 -> 1 (二进制)
- 4-1 = 3 -> 11 (二进制)
- 8-1 = 7 -> 111 (二进制)

所以  $n \& (n-1)$  的结果会是 0。

时间复杂度: O(1)

空间复杂度: O(1)

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

"""

```
class Solution:
```

```
    def isPowerOfTwo(self, n: int) -> bool:
```

```
        """
```

判断一个整数是否是 2 的幂次方

使用位运算技巧:  $n \& (n-1)$  可以消除 n 的二进制表示中最右边的 1

如果 n 是 2 的幂, 则其二进制表示中只有一个 1, 所以  $n \& (n-1)$  的结果为 0

```
        :param n: 待判断的整数
```

```
        :return: 如果是 2 的幂次方返回 True, 否则返回 False
```

```
        """
```

```
# n > 0 确保是正数
```

```
# n == (n & -n) 判断是否只有一个位为 1
```

```
# n & -n 可以提取出 n 的二进制中最右边的 1
```

```
    return n > 0 and n == (n & -n)
```

```
# 另一种实现方式
```

```
# def isPowerOfTwo(self, n: int) -> bool:
```

```
#     return n > 0 and (n & (n - 1)) == 0
```

```
# 测试方法
```

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

```
    solution = Solution()
```

```
# 测试用例 1: 正常情况 (是 2 的幂)
```

```
n1 = 16
```

```
result1 = solution.isPowerOfTwo(n1)
```

```
print(f"测试用例 1 - 输入: {n1} (是 2 的幂)")
```

```
print(f"结果: {result1} (预期: True)")
```

```
# 测试用例 2: 正常情况 (不是 2 的幂)
```

```
n2 = 18
```

```
result2 = solution.isPowerOfTwo(n2)
```

```
print(f"测试用例 2 - 输入: {n2} (不是 2 的幂)")
```

```
print(f"结果: {result2} (预期: False)")
```

```
# 测试用例 3: 边界情况 (0)
```

```
n3 = 0
```

```
result3 = solution.isPowerOfTwo(n3)
```

```
print(f"测试用例 3 - 输入: {n3}")
```

```
print(f"结果: {result3} (预期: False)")

# 测试用例 4: 边界情况 (负数)
n4 = -8
result4 = solution.isPowerOfTwo(n4)
print(f"测试用例 4 - 输入: {n4}")
print(f"结果: {result4} (预期: False)")

# 测试用例 5: 边界情况 (1)
n5 = 1
result5 = solution.isPowerOfTwo(n5)
print(f"测试用例 5 - 输入: {n5}")
print(f"结果: {result5} (预期: True)")

=====
```

文件: Code02\_PowerOfThree.cpp

```
/***
 * 3 的幂 - Power of Three
 * 测试链接 : https://leetcode.cn/problems/power-of-three/
 * 相关题目:
 * 1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
 * 2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
 * 3. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
 * 4. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/
 * 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
 *
 * 题目描述:
 * 给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true；否则，返回 false。
 * 整数 n 是 3 的幂次方需满足：存在整数 x 使得  $n = 3^x$ 
 *
 * 示例:
 * 输入: n = 27
 * 输出: true
 *
 * 输入: n = 0
 * 输出: false
 *
 * 输入: n = 9
 * 输出: true
 *
 * 输入: n = 45
```

```

* 输出: false
*
* 提示:
*  $-2^{31} \leq n \leq 2^{31} - 1$ 
*
* 进阶: 你能不使用循环或者递归来完成本题吗?
*
* 解题思路:
* 方法 1: 试除法
* 不断将 n 除以 3, 直到不能整除为止, 如果最后结果是 1, 则说明 n 是 3 的幂。
*
* 方法 2: 数学方法 (最优解)
* 在 int 范围内, 最大的 3 的幂是  $3^{19} = 1162261467$ 。
* 如果 n 是 3 的幂, 那么 1162261467 一定能被 n 整除。
* 反之, 如果  $1162267 \% n \neq 0$ , 说明 n 一定含有其他因子, 不是 3 的幂。
*
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code02_PowerOfThree {
public:
    /**
     * 判断一个整数是否是 3 的幂次方
     * 使用数学方法: 在 int 范围内, 最大的 3 的幂是  $3^{19} = 1162261467$ 
     * 如果 n 是 3 的幂, 那么 1162261467 一定能被 n 整除
     *
     * @param n 待判断的整数
     * @return 如果是 3 的幂次方返回 true, 否则返回 false
     */
    bool isPowerOfThree(int n) {
        // n > 0 确保是正数
        // 1162261467 % n == 0 判断是否只含有 3 这个质数因子
        return n > 0 && 1162261467 % n == 0;
    }
}

```

```
}

// 另一种实现方式（试除法）
// bool isPowerOfThree(int n) {
//     if (n < 1) {
//         return false;
//     }
//     while (n % 3 == 0) {
//         n /= 3;
//     }
//     return n == 1;
// }

};
```

```
// 测试方法
int main() {
    Code02_PowerOfThree solution;

    // 测试用例 1：正常情况（是 3 的幂）
    int n1 = 27;
    bool result1 = solution.isPowerOfThree(n1);
    // 预期结果: true

    // 测试用例 2：正常情况（不是 3 的幂）
    int n2 = 45;
    bool result2 = solution.isPowerOfThree(n2);
    // 预期结果: false

    // 测试用例 3：边界情况（0）
    int n3 = 0;
    bool result3 = solution.isPowerOfThree(n3);
    // 预期结果: false

    // 测试用例 4：边界情况（1）
    int n4 = 1;
    bool result4 = solution.isPowerOfThree(n4);
    // 预期结果: true

    // 测试用例 5：边界情况（9）
    int n5 = 9;
    bool result5 = solution.isPowerOfThree(n5);
    // 预期结果: true
```

```
    return 0;  
}
```

=====

文件: Code02\_PowerOfThree. java

=====

```
package class031;  
  
// 3 的幂 - Power of Three  
// 测试链接 : https://leetcode.cn/problems/power-of-three/  
// 相关题目：  
// 1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/  
// 2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/  
// 3. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/  
// 4. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/  
// 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
```

/\*

题目描述:

给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true；否则，返回 false。

整数 n 是 3 的幂次方需满足：存在整数 x 使得  $n = 3^x$

示例：

输入: n = 27

输出: true

输入: n = 0

输出: false

输入: n = 9

输出: true

输入: n = 45

输出: false

提示:

$-2^{31} \leq n \leq 2^{31} - 1$

进阶：你能不能不使用循环或者递归来完成本题吗？

解题思路:

方法 1：试除法

不断将 n 除以 3，直到不能整除为止，如果最后结果是 1，则说明 n 是 3 的幂。

方法 2：数学方法（最优解）

在 int 范围内，最大的 3 的幂是  $3^{19} = 1162261467$ 。

如果 n 是 3 的幂，那么 1162261467 一定能被 n 整除。

反之，如果  $1162267 \% n \neq 0$ ，说明 n 一定含有其他因子，不是 3 的幂。

时间复杂度：O(1)

空间复杂度：O(1)

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
*/  
public class Code02_PowerOfThree {
```

```
// 如果一个数字是 3 的某次幂，那么这个数一定只含有 3 这个质数因子  
// 1162261467 是 int 型范围内，最大的 3 的幂，它是 3 的 19 次方  
// 这个 1162261467 只含有 3 这个质数因子，如果 n 也是只含有 3 这个质数因子，那么  
// 1162261467 % n == 0  
// 反之如果 1162261467 % n != 0 说明 n 一定含有其他因子
```

```
/**
```

```
* 判断一个整数是否是 3 的幂次方  
* 使用数学方法：在 int 范围内，最大的 3 的幂是  $3^{19} = 1162261467$   
* 如果 n 是 3 的幂，那么 1162261467 一定能被 n 整除  
*  
* @param n 待判断的整数  
* @return 如果是 3 的幂次方返回 true，否则返回 false
```

```
*/
```

```
public static boolean isPowerOfThree(int n) {  
    // n > 0 确保是正数  
    // 1162261467 % n == 0 判断是否只含有 3 这个质数因子  
    return n > 0 && 1162261467 % n == 0;  
}
```

```
// 另一种实现方式（试除法）
```

```
// public static boolean isPowerOfThree(int n) {  
//     if (n < 1) {  
//         return false;  
//     }
```

```
//     while (n % 3 == 0) {  
//         n /= 3;  
//     }  
//     return n == 1;  
// }  
  
=====
```

文件: Code02\_PowerOfThree.py

```
=====
```

```
"""
```

3 的幂 - Power of Three

测试链接 : <https://leetcode.cn/problems/power-of-three/>

相关题目:

1. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
2. 4 的幂 - Power of Four: <https://leetcode.cn/problems/power-of-four/>
3. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
4. 缺失的数字 - Missing Number: <https://leetcode.cn/problems/missing-number/>
5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true；否则，返回 false。

整数 n 是 3 的幂次方需满足: 存在整数 x 使得  $n = 3^x$

示例:

输入: n = 27

输出: true

输入: n = 0

输出: false

输入: n = 9

输出: true

输入: n = 45

输出: false

提示:

$-2^{31} \leq n \leq 2^{31} - 1$

进阶: 你能不使用循环或者递归来完成本题吗？

解题思路：

方法 1：试除法

不断将 n 除以 3，直到不能整除为止，如果最后结果是 1，则说明 n 是 3 的幂。

方法 2：数学方法（最优解）

在 int 范围内，最大的 3 的幂是  $3^{19} = 1162261467$ 。

如果 n 是 3 的幂，那么 1162261467 一定能被 n 整除。

反之，如果  $1162267 \% n \neq 0$ ，说明 n 一定含有其他因子，不是 3 的幂。

时间复杂度：O(1)

空间复杂度：O(1)

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:  
    def isPowerOfThree(self, n: int) -> bool:  
        """  
        判断一个整数是否是 3 的幂次方  
        使用数学方法：在 int 范围内，最大的 3 的幂是  $3^{19} = 1162261467$   
        如果 n 是 3 的幂，那么 1162261467 一定能被 n 整除  
        """
```

```
:param n: 待判断的整数  
:return: 如果是 3 的幂次方返回 True，否则返回 False  
"""  
  
# n > 0 确保是正数  
#  $1162261467 \% n == 0$  判断是否只含有 3 这个质数因子  
return n > 0 and 1162261467 % n == 0
```

# 另一种实现方式（试除法）

```
# def isPowerOfThree(self, n: int) -> bool:  
#     if n < 1:  
#         return False  
#     while n % 3 == 0:  
#         n //= 3  
#     return n == 1
```

```
# 测试方法
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1: 正常情况 (是 3 的幂)
n1 = 27
result1 = solution.isPowerOfThree(n1)
print(f"测试用例 1 - 输入: {n1} (是 3 的幂)")
print(f"结果: {result1} (预期: True)")

# 测试用例 2: 正常情况 (不是 3 的幂)
n2 = 45
result2 = solution.isPowerOfThree(n2)
print(f"测试用例 2 - 输入: {n2} (不是 3 的幂)")
print(f"结果: {result2} (预期: False)")

# 测试用例 3: 边界情况 (0)
n3 = 0
result3 = solution.isPowerOfThree(n3)
print(f"测试用例 3 - 输入: {n3}")
print(f"结果: {result3} (预期: False)")

# 测试用例 4: 边界情况 (1)
n4 = 1
result4 = solution.isPowerOfThree(n4)
print(f"测试用例 4 - 输入: {n4}")
print(f"结果: {result4} (预期: True)")

# 测试用例 5: 边界情况 (9)
n5 = 9
result5 = solution.isPowerOfThree(n5)
print(f"测试用例 5 - 输入: {n5}")
print(f"结果: {result5} (预期: True)
```

=====

文件: Code03\_Near2power.cpp

=====

```
/***
 * 最接近的 2 的幂 - Near 2 Power
 * 测试链接 : 无直接对应 LeetCode 题目, 但为经典位运算技巧
```

\* 相关题目：

- \* 1. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
- \* 2. 4 的幂 - Power of Four: <https://leetcode.cn/problems/power-of-four/>
- \* 3. 3 的幂 - Power of Three: <https://leetcode.cn/problems/power-of-three/>
- \* 4. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
- \* 5. 数字范围按位与 - Bitwise AND of Numbers Range: <https://leetcode.cn/problems/bitwise-and-of-numbers-range/>

\*

\* 题目描述：

- \* 给定一个非负整数 n，返回大于等于 n 的最小的 2 的某次幂。

\*

\* 示例：

- \* 输入：n = 5

- \* 输出：8

- \* 解释： $2^3 = 8 \geq 5$ ，且是满足条件的最小 2 的幂

\*

- \* 输入：n = 1

- \* 输出：1

- \* 解释： $2^0 = 1 \geq 1$ ，且是满足条件的最小 2 的幂

\*

- \* 输入：n = 17

- \* 输出：32

- \* 解释： $2^5 = 32 \geq 17$ ，且是满足条件的最小 2 的幂

\*

\* 解题思路：

- \* 这是一个经典的位运算技巧。通过将 n 减 1 后，不断将其二进制表示中最高位 1 右边的所有位都置为 1，最后再加 1，就可以得到大于等于 n 的最小 2 的幂。

\*

\* 具体步骤：

- \* 1. 如果 n <= 0，返回 1

- \* 2. n--，将 n 减 1

- \* 3. 通过一系列位运算操作，将 n 的二进制表示中最高位 1 右边的所有位都置为 1

- \* 4. n++，得到结果

\*

\* 时间复杂度：O(1)

\* 空间复杂度：O(1)

\*

\* 补充题目：

- \* 1. 洛谷 P10118 『STA - R4』And: <https://www.luogu.com.cn/problem/P10118>

- \* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>

- \* 3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>

- \* 4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>

- \* 5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
/*
// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code03_Near2power {
public:
    /**
     * 返回大于等于 n 的最小的 2 的某次幂
     * 使用位运算技巧：通过将 n 减 1 后，不断将其二进制表示中最高位 1 右边的所有位都置为 1，
     * 最后再加 1，就可以得到大于等于 n 的最小 2 的幂
     *
     * @param n 非负整数
     * @return 大于等于 n 的最小的 2 的某次幂，如果不存在则返回整数最小值
     */
    int near2power(int n) {
        if (n <= 0) {
            return 1;
        }
        n--;
        // 通过位运算将 n 的二进制表示中最高位 1 右边的所有位都置为 1
        n |= n >> 1;    // 将最高位 1 右边 1 位都置为 1
        n |= n >> 2;    // 将最高位 1 右边 2 位都置为 1
        n |= n >> 4;    // 将最高位 1 右边 4 位都置为 1
        n |= n >> 8;    // 将最高位 1 右边 8 位都置为 1
        n |= n >> 16;   // 将最高位 1 右边 16 位都置为 1
        return n + 1;
    }
};

// 测试方法
int main() {
    Code03_Near2power solution;

    // 测试用例 1：正常情况
    int n1 = 5;
    int result1 = solution.near2power(n1);
    // 预期结果：8

    // 测试用例 2：边界情况
    int n2 = 1;
    int result2 = solution.near2power(n2);
    // 预期结果：1
}
```

```
// 测试用例 3: 较大数值
int n3 = 17;
int result3 = solution.near2power(n3);
// 预期结果: 32

// 测试用例 4: 0
int n4 = 0;
int result4 = solution.near2power(n4);
// 预期结果: 1

// 测试用例 5: 100
int n5 = 100;
int result5 = solution.near2power(n5);
// 预期结果: 128

return 0;
}
```

=====

文件: Code03\_Near2power.java

=====

```
package class031;

// 最接近的 2 的幂 - Near 2 Power
// 测试链接 : 无直接对应 LeetCode 题目, 但为经典位运算技巧
// 相关题目:
// 1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
// 3. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/
// 4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
// 5. 数字范围按位与 - Bitwise AND of Numbers Range: https://leetcode.cn/problems/bitwise-and-of-numbers-range/
```

/\*

题目描述:

给定一个非负整数 n, 返回大于等于 n 的最小的 2 的某次幂。

示例:

输入: n = 5

输出: 8

解释:  $2^3 = 8 \geq 5$ , 且是满足条件的最小 2 的幂

输入: n = 1

输出: 1

解释:  $2^0 = 1 \geq 1$ , 且是满足条件的最小 2 的幂

输入: n = 17

输出: 32

解释:  $2^5 = 32 \geq 17$ , 且是满足条件的最小 2 的幂

解题思路:

这是一个经典的位运算技巧。通过将 n 减 1 后，不断将其二进制表示中最高位 1 右边的所有位都置为 1，最后再加 1，就可以得到大于等于 n 的最小 2 的幂。

具体步骤:

1. 如果  $n \leq 0$ , 返回 1
2.  $n--$ , 将 n 减 1
3. 通过一系列位运算操作, 将 n 的二进制表示中最高位 1 右边的所有位都置为 1
4.  $n++$ , 得到结果

时间复杂度: O(1)

空间复杂度: O(1)

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
/*
 * 返回大于等于 n 的最小的 2 的某次幂
 * 使用位运算技巧: 通过将 n 减 1 后, 不断将其二进制表示中最高位 1 右边的所有位都置为 1,
 * 最后再加 1, 就可以得到大于等于 n 的最小 2 的幂
 *
 * @param n 非负整数
 * @return 大于等于 n 的最小的 2 的某次幂, 如果不存在则返回整数最小值
 */
public static final int near2power(int n) {
    if (n <= 0) {
        return 1;
    }
    n--;
}
```

```

        // 通过位运算将 n 的二进制表示中最高位 1 右边的所有位都置为 1
        n |= n >>> 1;    // 将最高位 1 右边 1 位都置为 1
        n |= n >>> 2;    // 将最高位 1 右边 2 位都置为 1
        n |= n >>> 4;    // 将最高位 1 右边 4 位都置为 1
        n |= n >>> 8;    // 将最高位 1 右边 8 位都置为 1
        n |= n >>> 16;   // 将最高位 1 右边 16 位都置为 1
        return n + 1;
    }

    public static void main(String[] args) {
        int number = 100;
        System.out.println(near2power(number));
    }
}

```

---

文件: Code03\_Near2power.py

---

```

"""
最接近的 2 的幂 - Near 2 Power
测试链接 : 无直接对应 LeetCode 题目, 但为经典位运算技巧
相关题目:
1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
3. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/
4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
5. 数字范围按位与 - Bitwise AND of Numbers Range: https://leetcode.cn/problems/bitwise-and-of-numbers-range/

```

题目描述:

给定一个非负整数 n, 返回大于等于 n 的最小的 2 的某次幂。

示例:

输入: n = 5

输出: 8

解释:  $2^3 = 8 \geq 5$ , 且是满足条件的最小 2 的幂

输入: n = 1

输出: 1

解释:  $2^0 = 1 \geq 1$ , 且是满足条件的最小 2 的幂

输入: n = 17

输出: 32

解释:  $2^5 = 32 \geq 17$ , 且是满足条件的最小 2 的幂

解题思路:

这是一个经典的位运算技巧。通过将 n 减 1 后，不断将其二进制表示中最高位 1 右边的所有位都置为 1，最后再加 1，就可以得到大于等于 n 的最小 2 的幂。

具体步骤:

1. 如果  $n \leq 0$ , 返回 1
2.  $n--$ , 将 n 减 1
3. 通过一系列位运算操作, 将 n 的二进制表示中最高位 1 右边的所有位都置为 1
4.  $n++$ , 得到结果

时间复杂度:  $O(1)$

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def near2power(self, n: int) -> int:  
        """
```

返回大于等于 n 的最小的 2 的某次幂

使用位运算技巧: 通过将 n 减 1 后, 不断将其二进制表示中最高位 1 右边的所有位都置为 1, 最后再加 1, 就可以得到大于等于 n 的最小 2 的幂

```
:param n: 非负整数
```

```
:return: 大于等于 n 的最小的 2 的某次幂, 如果不存在则返回整数最小值
```

```
"""
```

```
    if n <= 0:
```

```
        return 1
```

```
    n -= 1
```

```
# 通过位运算将 n 的二进制表示中最高位 1 右边的所有位都置为 1
```

```
n |= n >> 1 # 将最高位 1 右边 1 位都置为 1
```

```
n |= n >> 2 # 将最高位 1 右边 2 位都置为 1
```

```
n |= n >> 4 # 将最高位 1 右边 4 位都置为 1
```

```
n |= n >> 8 # 将最高位 1 右边 8 位都置为 1
n |= n >> 16 # 将最高位 1 右边 16 位都置为 1
return n + 1

# 测试方法
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1: 正常情况
n1 = 5
result1 = solution.near2power(n1)
print(f"测试用例 1 - 输入: {n1}")
print(f"结果: {result1} (预期: 8)")

# 测试用例 2: 边界情况
n2 = 1
result2 = solution.near2power(n2)
print(f"测试用例 2 - 输入: {n2}")
print(f"结果: {result2} (预期: 1)")

# 测试用例 3: 较大数值
n3 = 17
result3 = solution.near2power(n3)
print(f"测试用例 3 - 输入: {n3}")
print(f"结果: {result3} (预期: 32)")

# 测试用例 4: 0
n4 = 0
result4 = solution.near2power(n4)
print(f"测试用例 4 - 输入: {n4}")
print(f"结果: {result4} (预期: 1)")

# 测试用例 5: 100
n5 = 100
result5 = solution.near2power(n5)
print(f"测试用例 5 - 输入: {n5}")
print(f"结果: {result5} (预期: 128)")
```

=====

文件: Code04\_LeftToRightAnd.cpp

=====

```
/**  
 * 数字范围按位与 - Bitwise AND of Numbers Range  
 * 测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/  
 * 相关题目：  
 * 1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/  
 * 2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/  
 * 3. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/  
 * 4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
 * 5. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/  
 *  
 * 题目描述：  
 * 给你两个整数 left 和 right ，表示区间 [left, right] ，返回此区间内所有数字按位 AND 的结果。  
 *  
 * 示例：  
 * 输入: left = 5, right = 7  
 * 输出: 4  
 * 解释: 5 & 6 & 7 = 4  
 *  
 * 输入: left = 0, right = 0  
 * 输出: 0  
 *  
 * 输入: left = 1, right = 2147483647  
 * 输出: 0  
 *  
 * 提示：  
 *  $0 \leq \text{left} \leq \text{right} \leq 2^{31} - 1$   
 *  
 * 解题思路：  
 * 方法 1: Brian Kernighan 算法  
 * 当  $\text{left} < \text{right}$  时，区间内一定存在相邻的两个数，它们的最低位分别是 0 和 1，所以按位 AND 的结果最低位一定是 0。  
 * 我们可以不断移除 right 最右边的 1，直到  $\text{left} == \text{right}$  为止。  
 *  
 * 方法 2: 找公共前缀  
 * 所有数字按位 AND 的结果就是 left 和 right 的二进制表示的公共前缀。  
 *  
 * 时间复杂度:  $O(\log n)$ ，其中 n 是数字的位数  
 * 空间复杂度:  $O(1)$   
 *  
 * 补充题目：  
 * 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118  
 * 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451  
 * 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
```

```

* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code04_LeftToRightAnd {
public:
    /**
     * 计算区间 [left, right] 内所有数字按位 AND 的结果
     * 使用 Brian Kernighan 算法：不断移除 right 最右边的 1，直到 left >= right
     *
     * @param left 区间左端点
     * @param right 区间右端点
     * @return 区间内所有数字按位 AND 的结果
     */
    int rangeBitwiseAnd(int left, int right) {
        // Brian Kernighan 算法的应用
        // 不断移除 right 最右边的 1，直到 left >= right
        while (left < right) {
            // right & -right 可以提取出 right 最右边的 1
            // right -= right & -right 相当于移除了最右边的 1
            right -= right & -right;
        }
        return right;
    }

    // 另一种实现方式（找公共前缀）
    // int rangeBitwiseAnd(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;
    // }
};

// 测试方法
int main() {
    Code04_LeftToRightAnd solution;

```

```

// 测试用例 1: 正常情况
int left1 = 5, right1 = 7;
int result1 = solution.rangeBitwiseAnd(left1, right1);
// 预期结果: 4

// 测试用例 2: 边界情况
int left2 = 0, right2 = 0;
int result2 = solution.rangeBitwiseAnd(left2, right2);
// 预期结果: 0

// 测试用例 3: 大范围
int left3 = 1, right3 = 2147483647;
int result3 = solution.rangeBitwiseAnd(left3, right3);
// 预期结果: 0

// 测试用例 4: 小范围
int left4 = 26, right4 = 30;
int result4 = solution.rangeBitwiseAnd(left4, right4);
// 预期结果: 24

return 0;
}

```

=====

文件: Code04\_LeftToRightAnd.java

=====

```

package class031;

// 数字范围按位与 - Bitwise AND of Numbers Range
// 测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/
// 相关题目:
// 1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
// 3. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/
// 4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
// 5. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/

/*
题目描述:
给你两个整数 left 和 right , 表示区间 [left, right] , 返回此区间内所有数字按位 AND 的结果。

```

示例：

输入：left = 5, right = 7

输出：4

解释：5 & 6 & 7 = 4

输入：left = 0, right = 0

输出：0

输入：left = 1, right = 2147483647

输出：0

提示：

$0 \leq left \leq right \leq 2^{31} - 1$

解题思路：

方法 1：Brian Kernighan 算法

当  $left < right$  时，区间内一定存在相邻的两个数，它们的最低位分别是 0 和 1，所以按位 AND 的结果最低位一定是 0。

我们可以不断移除 right 最右边的 1，直到  $left == right$  为止。

方法 2：找公共前缀

所有数字按位 AND 的结果就是 left 和 right 的二进制表示的公共前缀。

时间复杂度： $O(\log n)$ ，其中  $n$  是数字的位数

空间复杂度： $O(1)$

补充题目：

1. 洛谷 P10118 『STA - R4』And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

\*/

public class Code04\_LeftToRightAnd {

/\*\*

\* 计算区间 [left, right] 内所有数字按位 AND 的结果

\* 使用 Brian Kernighan 算法：不断移除 right 最右边的 1，直到  $left \geq right$

\*

\* @param left 区间左端点

\* @param right 区间右端点

\* @return 区间内所有数字按位 AND 的结果

\*/

```

public static int rangeBitwiseAnd(int left, int right) {
    // Brian Kernighan 算法的应用
    // 不断移除 right 最右边的 1，直到 left >= right
    while (left < right) {
        // right & -right 可以提取出 right 最右边的 1
        // right -= right & -right 相当于移除了最右边的 1
        right -= right & -right;
    }
    return right;
}

// 另一种实现方式（找公共前缀）
// public static int rangeBitwiseAnd(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;
// }

}

=====

```

文件: Code04\_LeftToRightAnd.py

```

=====
"""

数字范围按位与 - Bitwise AND of Numbers Range
测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/
相关题目:
1. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
2. 4 的幂 - Power of Four: https://leetcode.cn/problems/power-of-four/
3. 3 的幂 - Power of Three: https://leetcode.cn/problems/power-of-three/
4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
5. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/

```

题目描述:

给你两个整数 left 和 right，表示区间 [left, right]，返回此区间内所有数字按位 AND 的结果。

示例:

输入: left = 5, right = 7

输出: 4

解释:  $5 \& 6 \& 7 = 4$

输入: left = 0, right = 0

输出: 0

输入: left = 1, right = 2147483647

输出: 0

提示:

$0 \leq \text{left} \leq \text{right} \leq 2^{31} - 1$

解题思路:

方法 1: Brian Kernighan 算法

当  $\text{left} < \text{right}$  时, 区间内一定存在相邻的两个数, 它们的最低位分别是 0 和 1, 所以按位 AND 的结果最低位一定是 0。

我们可以不断移除 right 最右边的 1, 直到  $\text{left} == \text{right}$  为止。

方法 2: 找公共前缀

所有数字按位 AND 的结果就是 left 和 right 的二进制表示的公共前缀。

时间复杂度:  $O(\log n)$ , 其中  $n$  是数字的位数

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def rangeBitwiseAnd(self, left: int, right: int) -> int:
```

```
        """
```

```
        计算区间 [left, right] 内所有数字按位 AND 的结果
```

```
        使用 Brian Kernighan 算法: 不断移除 right 最右边的 1, 直到 left >= right
```

```
        :param left: 区间左端点
```

```
        :param right: 区间右端点
```

```
        :return: 区间内所有数字按位 AND 的结果
```

```
"""
# Brian Kernighan 算法的应用
# 不断移除 right 最右边的 1，直到 left >= right
while left < right:
    # right & -right 可以提取出 right 最右边的 1
    # right -= right & -right 相当于移除了最右边的 1
    right -= right & -right
return right

# 另一种实现方式（找公共前缀）
# def rangeBitwiseAnd(self, left: int, right: int) -> int:
#     shift = 0
#     # 找到 left 和 right 的公共前缀
#     while left != right:
#         left >>= 1
#         right >>= 1
#         shift += 1
#     return left << shift

# 测试方法
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1：正常情况
left1, right1 = 5, 7
result1 = solution.rangeBitwiseAnd(left1, right1)
print(f"测试用例 1 - 输入: left={left1}, right={right1}")
print(f"结果: {result1} (预期: 4)")

# 测试用例 2：边界情况
left2, right2 = 0, 0
result2 = solution.rangeBitwiseAnd(left2, right2)
print(f"测试用例 2 - 输入: left={left2}, right={right2}")
print(f"结果: {result2} (预期: 0)")

# 测试用例 3：大范围
left3, right3 = 1, 2147483647
result3 = solution.rangeBitwiseAnd(left3, right3)
print(f"测试用例 3 - 输入: left={left3}, right={right3}")
print(f"结果: {result3} (预期: 0)")

# 测试用例 4：小范围
```

```
left4, right4 = 26, 30
result4 = solution.rangeBitwiseAnd(left4, right4)
print(f"测试用例 4 - 输入: left={left4}, right={right4}")
print(f"结果: {result4} (预期: 24)")
```

---

文件: Code05\_ReverseBits.cpp

---

```
/*
 * 颠倒二进制位 - Reverse Bits
 * 测试链接 : https://leetcode.cn/problems/reverse-bits/
 * 相关题目:
 * 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/
 * 2. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
 * 3. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
 * 4. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
 * 5. 格雷编码 - Gray Code: https://leetcode.cn/problems/gray-code/
 *
 * 题目描述:
 * 颠倒给定的 32 位无符号整数的二进制位。
 *
 * 示例:
 * 输入: n = 00000010100101000001111010011100
 * 输出: 964176192 (00111001011110000010100101000000)
 * 解释: 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596,
 * 因此返回 964176192, 其二进制表示形式为 00111001011110000010100101000000。
 *
 * 输入: n = 111111111111111111111111111101
 * 输出: 3221225471 (101111111111111111111111111111)
 * 解释: 输入的二进制串 111111111111111111111111111101 表示无符号整数 4294967293,
 * 因此返回 3221225471 其二进制表示形式为 101111111111111111111111111111 。
 *
 * 提示:
 * 输入是一个长度为 32 的二进制字符串
 *
 * 解题思路:
 * 这是一个经典的位运算问题, 可以通过分治法来解决。
 * 我们逐步交换相邻的位、相邻的两位、相邻的四位... 直到交换整个 32 位。
 *
 * 具体步骤:
 * 1. 交换相邻的位: 0aaaaaaaaa = 101010101010101010101010, 0x55555555 =
01010101010101010101010101010101
```

```

* 2. 交换相邻的两位: 0xcccccccc = 11001100110011001100110011001100, 0x33333333 = 00110011001100110011001100110011
* 3. 交换相邻的四位: 0xf0f0f0f0 = 11110000111100001111000011110000, 0x0f0f0f0f = 00001111000011110000111100001111
* 4. 交换相邻的八位: 0xff00ff00 = 111111100000000111111100000000, 0x00ff00ff = 000000001111111000000001111111
* 5. 交换前十六位和后十六位
*
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code05_ReverseBits {
public:
    /**
     * 颠倒给定的 32 位无符号整数的二进制位
     * 使用分治法: 逐步交换相邻的位、相邻的两位、相邻的四位... 直到交换整个 32 位
     *
     * @param n 32 位无符号整数
     * @return 颠倒二进制位后的结果
     */
    unsigned int reverseBits(unsigned int n) {
        // 交换相邻的位
        // 0xaaaaaaaa = 1010101010101010101010101010 (偶数位为 1)
        // 0x55555555 = 01010101010101010101010101 (奇数位为 1)
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);

        // 交换相邻的两位
        // 0xcccccccc = 1100110011001100110011001100
        // 0x33333333 = 0011001100110011001100110011
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);

        // 交换相邻的四位
        // 0xf0f0f0f0 = 11110000111100001111000011110000

```

文件: Code05\_ReverseBits.java

```
package class031;

// 颠倒二进制位 - Reverse Bits
// 测试链接 : https://leetcode.cn/problems/reverse-bits/
// 相关题目:
// 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/
// 2. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
```

```
// 3. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
// 4. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/  
// 5. 格雷编码 - Gray Code: https://leetcode.cn/problems/gray-code/
```

/\*

### 题目描述:

颠倒给定的 32 位无符号整数的二进制位。

示例：

输入: n = 00000010100101000001111010011100

输出: 964176192 (00111001011110000010100101000000)

解释：输入的二进制串 0000010100101000001111010011100 表示无符号整数 43261596，

因此返回 964176192，其二进制表示形式为 0011100101111000001010010100000。

输出: 3221225471 (10111111111111111111111111111111)

提示：

输入是一个长度为 32 的二进制字符串

解题思路：

这是一个经典的位运算问题，可以通过分治法来解决。

我们逐步交换相邻的位、相邻的两位、相邻的四位...直到交换整个 32 位。

具体步骤：

2. 交换相邻的两位:  $0x\text{cccccccc} = 1100110011001100110011001100$ ,  $0x\text{33333333} = 0011001100110011001100110011$

3. 交换相邻的四位:  $0xf0f0f0f0 = 11110000111100001111000011110000$ ,  $0x0f0f0f0f = 00001111000011110000111100001111$

4. 交换相邻的八位:  $0xff00ff00 = 11111111000000001111111100000000$ ,  $0x00ff00ff = 00000000111111110000000011111111$

## 5 交换前十六位和后十六位

时间复杂度:  $O(1)$

空间复杂度:  $O(1)$

### 补充题目.

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>

2 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>

3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
/*
 * 颠倒给定的 32 位无符号整数的二进制位
 * 使用分治法：逐步交换相邻的位、相邻的两位、相邻的四位... 直到交换整个 32 位
 *
 * @param n 32 位无符号整数
 * @return 颠倒二进制位后的结果
 */
public class Code05_ReverseBits {

    /**
     * 颠倒给定的 32 位无符号整数的二进制位
     * 使用分治法：逐步交换相邻的位、相邻的两位、相邻的四位... 直到交换整个 32 位
     *
     * @param n 32 位无符号整数
     * @return 颠倒二进制位后的结果
     */
    public static int reverseBits(int n) {
        // 交换相邻的位
        // 0aaaaaaaaa = 101010101010101010101010101010 (偶数位为 1)
        // 0x55555555 = 010101010101010101010101010101 (奇数位为 1)
        n = ((n & 0aaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1);

        // 交换相邻的两位
        // 0ccccccc = 11001100110011001100110011001100
        // 0x33333333 = 00110011001100110011001100110011
        n = ((n & 0ccccccc) >>> 2) | ((n & 0x33333333) << 2);

        // 交换相邻的四位
        // 0xf0f0f0f0 = 11110000111100001111000011110000
        // 0x0f0f0f0f = 00001111000011110000111100001111
        n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4);

        // 交换相邻的八位
        // 0xff00ff00 = 111111100000000111111100000000
        // 0x00ff00ff = 000000011111110000000011111111
        n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8);

        // 交换前十六位和后十六位
        n = (n >>> 16) | (n << 16);

        return n;
    }
}
```

=====

文件: Code05\_ReverseBits.py

=====

'''

颠倒二进制位 - Reverse Bits

测试链接 : <https://leetcode.cn/problems/reverse-bits/>

相关题目:

1. 位 1 的个数 - Number of 1 Bits: <https://leetcode.cn/problems/number-of-1-bits/>
2. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
3. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
4. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>
5. 格雷编码 - Gray Code: <https://leetcode.cn/problems/gray-code/>

题目描述:

颠倒给定的 32 位无符号整数的二进制位。

示例:

输入: n = 0000001010010100000111010011100

输出: 964176192 (0011100101111000001010010100000)

解释: 输入的二进制串 0000001010010100000111010011100 表示无符号整数 43261596,

因此返回 964176192, 其二进制表示形式为 0011100101111000001010010100000。

输入: n = 11111111111111111111111111111101

输出: 3221225471 (10111111111111111111111111111111)

解释: 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293,

因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111 。

提示:

输入是一个长度为 32 的二进制字符串

解题思路:

这是一个经典的位运算问题, 可以通过分治法来解决。

我们逐步交换相邻的位、相邻的两位、相邻的四位... 直到交换整个 32 位。

具体步骤:

1. 交换相邻的位: 0xaaaaaaaa = 1010101010101010101010101010, 0x55555555 = 01010101010101010101010101

2. 交换相邻的两位: 0xcccccccc = 11001100110011001100110011001100, 0x33333333 = 0011001100110011001100110011

3. 交换相邻的四位: 0xf0f0f0f0 = 11110000111100001111000011110000, 0x0f0f0f0f = 00001111000011110000111100001111

4. 交换相邻的八位: 0xff00ff00 = 111111100000000111111100000000, 0x00ff00ff = 000000011111110000000011111111

## 5. 交换前十六位和后十六位

时间复杂度: O(1)

空间复杂度: O(1)

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:  
    def reverseBits(self, n: int) -> int:  
        """  
        颠倒给定的 32 位无符号整数的二进制位  
        使用分治法: 逐步交换相邻的位、相邻的两位、相邻的四位...直到交换整个 32 位  
  
        :param n: 32 位无符号整数  
        :return: 颠倒二进制位后的结果  
        """  
  
        # 交换相邻的位  
        # 0xaaaaaaaa = 1010101010101010101010101010 (偶数位为 1)  
        # 0x55555555 = 0101010101010101010101010101 (奇数位为 1)  
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1)  
  
        # 交换相邻的两位  
        # 0xcccccccc = 1100110011001100110011001100  
        # 0x33333333 = 0011001100110011001100110011  
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2)  
  
        # 交换相邻的四位  
        # 0xf0f0f0f0 = 11110000111100001111000011110000  
        # 0x0f0f0f0f = 00001111000011110000111100001111  
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4)  
  
        # 交换相邻的八位  
        # 0xff00ff00 = 111111100000000111111100000000  
        # 0x00ff00ff = 0000000011111110000000011111111  
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8)
```

```
# 交换前十六位和后十六位
n = (n >> 16) | (n << 16)

# 由于 Python 中的整数是无限精度的，需要确保结果在 32 位范围内
return n & 0xFFFFFFFF

# 测试方法
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1：正常情况
n1 = 43261596 # 0000001010010100000111010011100
result1 = solution.reverseBits(n1)
# 预期结果：964176192 (00111001011110000010100101000000)
print(f"测试用例 1 - 输入: {n1}")
print(f"结果: {result1} (预期: 964176192)")

# 测试用例 2：边界情况
n2 = 4294967293 # 1111111111111111111111111111101
result2 = solution.reverseBits(n2)
# 预期结果：3221225471 (10111111111111111111111111111111)
print(f"测试用例 2 - 输入: {n2}")
print(f"结果: {result2} (预期: 3221225471)")
```

文件: Code06\_CountOnesBinarySystem.cpp

```
/**  
 * 汉明距离 - Hamming Distance  
 * 测试链接 : https://leetcode.cn/problems/hamming-distance/  
 * 相关题目:  
 * 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/  
 * 2. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/  
 * 3. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/  
 * 4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
 * 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/  
 *  
 * 题目描述:  
 * 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。  
 * 给你两个整数 x 和 y, 计算并返回它们之间的汉明距离。  
 */
```

```

* 示例:
* 输入: x = 1, y = 4
* 输出: 2
* 解释:
* 1  (0 0 0 1)
* 4  (0 1 0 0)
* ↑   ↑
* 上面的箭头指出了对应二进制位不同的位置。
*
* 输入: x = 3, y = 1
* 输出: 1
*
* 提示:
* 0 <= x, y <= 2^31 - 1
*
* 解题思路:
* 汉明距离就是两个数异或结果中 1 的个数。
* 例如: x = 1 (0001), y = 4 (0100)
* x ^ y = 0101, 其中有 2 个 1, 所以汉明距离是 2。
*
* 计算一个数二进制中 1 的个数有多种方法:
* 1. Brian Kernighan 算法: n & (n-1) 可以移除最右边的 1
* 2. 分治法: 通过位运算并行计算每两位、每四位、每八位... 中 1 的个数
*
* 这里使用的是分治法, 通过并行计算来统计 1 的个数。
*
* 时间复杂度: O(1)
* 空间复杂度: O(1)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 I/O 函数

class Code06_CountOnesBinarySystem {
public:
    /**
     * 计算两个整数之间的汉明距离

```

```

* 汉明距离就是两个数异或结果中 1 的个数
*
* @param x 第一个整数
* @param y 第二个整数
* @return 汉明距离（二进制位不同的位置的数目）
*/
int hammingDistance(int x, int y) {
    // 两个数异或的结果中 1 的个数就是汉明距离
    return cntOnes(x ^ y);
}

// 返回 n 的二进制中有几个 1
// 这个实现脑洞太大了
/***
 * 计算一个整数的二进制表示中 1 的个数
 * 使用分治法：通过并行计算来统计 1 的个数
 *
 * @param n 待计算的整数
 * @return 二进制表示中 1 的个数
*/
int cntOnes(int n) {
    // 0x55555555 = 010101010101010101010101010101 (奇数位为 1)
    // 计算每两位中 1 的个数
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555);

    // 0x33333333 = 0011001100110011001100110011 (每两位为 11)
    // 计算每四位中 1 的个数
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333);

    // 0x0f0f0f0f = 0000111000011110000111100001111 (每四位为 1111)
    // 计算每八位中 1 的个数
    n = (n & 0x0f0f0f0f) + ((n >> 4) & 0x0f0f0f0f);

    // 0x00ff00ff = 000000001111111000000001111111111111111
    // 计算每十六位中 1 的个数
    n = (n & 0x00ff00ff) + ((n >> 8) & 0x00ff00ff);

    // 0x0000ffff = 00000000000000001111111111111111
    // 计算每三十二位中 1 的个数
    n = (n & 0x0000ffff) + ((n >> 16) & 0x0000ffff);

    return n;
}

```

```

// 另一种实现方式 (Brian Kernighan 算法)
// int cntOnes(int n) {
//     int count = 0;
//     while (n != 0) {
//         count++;
//         n &= n - 1; // 移除最右边的 1
//     }
//     return count;
// }

};

// 测试方法
int main() {
    Code06_CountOnesBinarySystem solution;

    // 测试用例 1: 正常情况
    int x1 = 1, y1 = 4;
    int result1 = solution.hammingDistance(x1, y1);
    // 预期结果: 2

    // 测试用例 2: 边界情况
    int x2 = 3, y2 = 1;
    int result2 = solution.hammingDistance(x2, y2);
    // 预期结果: 1

    return 0;
}

```

=====

文件: Code06\_CountOnesBinarySystem.java

=====

```

package class031;

// 汉明距离 - Hamming Distance
// 测试链接 : https://leetcode.cn/problems/hamming-distance/
// 相关题目:
// 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/
// 2. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/
// 3. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 4. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
// 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/

```

/\*

题目描述:

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。

给你两个整数  $x$  和  $y$ , 计算并返回它们之间的汉明距离。

示例:

输入:  $x = 1, y = 4$

输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

上面的箭头指出了对应二进制位不同的位置。

输入:  $x = 3, y = 1$

输出: 1

提示:

$0 \leq x, y \leq 2^{31} - 1$

解题思路:

汉明距离就是两个数异或结果中 1 的个数。

例如:  $x = 1$  (0001),  $y = 4$  (0100)

$x \wedge y = 0101$ , 其中有 2 个 1, 所以汉明距离是 2。

计算一个数二进制中 1 的个数有多种方法:

1. Brian Kernighan 算法:  $n \& (n-1)$  可以移除最右边的 1
2. 分治法: 通过位运算并行计算每两位、每四位、每八位... 中 1 的个数

这里使用的是分治法, 通过并行计算来统计 1 的个数。

时间复杂度:  $O(1)$

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

\*/  
public class Code06\_CountOnesBinarySystem {

```

/**
 * 计算两个整数之间的汉明距离
 * 汉明距离就是两个数异或结果中 1 的个数
 *
 * @param x 第一个整数
 * @param y 第二个整数
 * @return 汉明距离（二进制位不同的位置的数目）
 */
public static int hammingDistance(int x, int y) {
    // 两个数异或的结果中 1 的个数就是汉明距离
    return cntOnes(x ^ y);
}

// 返回 n 的二进制中有几个 1
// 这个实现脑洞太大了
/***
 * 计算一个整数的二进制表示中 1 的个数
 * 使用分治法：通过并行计算来统计 1 的个数
 *
 * @param n 待计算的整数
 * @return 二进制表示中 1 的个数
 */
public static int cntOnes(int n) {
    // 0x55555555 = 0101010101010101010101010101 (奇数位为 1)
    // 计算每两位中 1 的个数
    n = (n & 0x55555555) + ((n >>> 1) & 0x55555555);

    // 0x33333333 = 0011001100110011001100110011 (每两位为 11)
    // 计算每四位中 1 的个数
    n = (n & 0x33333333) + ((n >>> 2) & 0x33333333);

    // 0x0f0f0f0f = 00001111000011110000111100001111 (每四位为 1111)
    // 计算每八位中 1 的个数
    n = (n & 0x0f0f0f0f) + ((n >>> 4) & 0x0f0f0f0f);

    // 0x00ff00ff = 00000000111111100000000011111111
    // 计算每十六位中 1 的个数
    n = (n & 0x00ff00ff) + ((n >>> 8) & 0x00ff00ff);

    // 0x0000ffff = 00000000000000001111111111111111
    // 计算每三十二位中 1 的个数
    n = (n & 0x0000ffff) + ((n >>> 16) & 0x0000ffff);
}

```

```
    return n;
}

// 另一种实现方式 (Brian Kernighan 算法)
// public static int cntOnes(int n) {
//     int count = 0;
//     while (n != 0) {
//         count++;
//         n &= n - 1; // 移除最右边的 1
//     }
//     return count;
// }
```

}

=====

文件: Code06\_CountOnesBinarySystem.py

=====

"""

汉明距离 - Hamming Distance

测试链接 : <https://leetcode.cn/problems/hamming-distance/>

相关题目：

1. 位 1 的个数 - Number of 1 Bits: <https://leetcode.cn/problems/number-of-1-bits/>
2. 汉明距离总和 - Total Hamming Distance: <https://leetcode.cn/problems/total-hamming-distance/>
3. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
4. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
5. 颠倒二进制位 - Reverse Bits: <https://leetcode.cn/problems/reverse-bits/>

题目描述：

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。

给你两个整数  $x$  和  $y$ ，计算并返回它们之间的汉明距离。

示例：

输入:  $x = 1$ ,  $y = 4$

输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

上面的箭头指出了对应二进制位不同的位置。

输入:  $x = 3, y = 1$

输出: 1

提示:

$0 \leq x, y \leq 2^{31} - 1$

解题思路:

汉明距离就是两个数异或结果中 1 的个数。

例如:  $x = 1$  (0001),  $y = 4$  (0100)

$x \wedge y = 0101$ , 其中有 2 个 1, 所以汉明距离是 2。

计算一个数二进制中 1 的个数有多种方法:

1. Brian Kernighan 算法:  $n \& (n-1)$  可以移除最右边的 1
2. 分治法: 通过位运算并行计算每两位、每四位、每八位... 中 1 的个数

这里使用的是分治法, 通过并行计算来统计 1 的个数。

时间复杂度:  $O(1)$

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def hammingDistance(self, x: int, y: int) -> int:
```

```
        """
```

```
        计算两个整数之间的汉明距离
```

```
        汉明距离就是两个数异或结果中 1 的个数
```

```
        :param x: 第一个整数
```

```
        :param y: 第二个整数
```

```
        :return: 汉明距离 (二进制位不同的位置的数目)
```

```
        """
```

```
        # 两个数异或的结果中 1 的个数就是汉明距离
```

```
        return self.cntOnes(x ^ y)
```

```
# 返回 n 的二进制中有几个 1
```

```

# 这个实现脑洞太大了

def cntOnes(self, n: int) -> int:
    """
    计算一个整数的二进制表示中 1 的个数
    使用分治法：通过并行计算来统计 1 的个数

    :param n: 待计算的整数
    :return: 二进制表示中 1 的个数
    """

    # 0x55555555 = 010101010101010101010101010101 (奇数位为 1)
    # 计算每两位中 1 的个数
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555)

    # 0x33333333 = 0011001100110011001100110011 (每两位为 11)
    # 计算每四位中 1 的个数
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333)

    # 0x0f0f0f0f = 00001111000011110000111100001111 (每四位为 1111)
    # 计算每八位中 1 的个数
    n = (n & 0x0f0f0f0f) + ((n >> 4) & 0x0f0f0f0f)

    # 0x00ff00ff = 0000000011111110000000011111111
    # 计算每十六位中 1 的个数
    n = (n & 0x00ff00ff) + ((n >> 8) & 0x00ff00ff)

    # 0x0000ffff = 00000000000000001111111111111111
    # 计算每三十二位中 1 的个数
    n = (n & 0x0000ffff) + ((n >> 16) & 0x0000ffff)

    return n

# 另一种实现方式 (Brian Kernighan 算法)
# def cntOnes(self, n: int) -> int:
#     count = 0
#     while n != 0:
#         count += 1
#         n &= n - 1 # 移除最右边的 1
#     return count

# 测试方法
if __name__ == "__main__":
    solution = Solution()

```

```
# 测试用例 1: 正常情况
x1, y1 = 1, 4
result1 = solution.hammingDistance(x1, y1)
# 预期结果: 2
print(f"测试用例 1 - 输入: x={x1}, y={y1}")
print(f"结果: {result1} (预期: 2)")

# 测试用例 2: 边界情况
x2, y2 = 3, 1
result2 = solution.hammingDistance(x2, y2)
# 预期结果: 1
print(f"测试用例 2 - 输入: x={x2}, y={y2}")
print(f"结果: {result2} (预期: 1)")

=====
```

文件: Code07\_SingleNumber.cpp

```
=====
/***
 * 只出现一次的数字 - Single Number
 * 测试链接 : https://leetcode.cn/problems/single-number/
 * 相关题目:
 * 1. 只出现一次的数字 II - Single Number II: https://leetcode.cn/problems/single-number-ii/
 * 2. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/
 * 3. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/
 * 4. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
 * 5. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
 *
 * 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。
 *
 * 示例:
 * 输入: nums = [2, 2, 1]
 * 输出: 1
 *
 * 输入: nums = [4, 1, 2, 1, 2]
 * 输出: 4
 *
 * 输入: nums = [1]
 * 输出: 1
 *
 */
```

```

* 提示:
* 1 <= nums.length <= 3 * 10^4
* -3 * 10^4 <= nums[i] <= 3 * 10^4
* 除了某个元素只出现一次以外，其余每个元素均出现两次。
*
* 解题思路:
* 这是一个经典的位运算问题，利用异或运算的性质：
* 1. a ^ a = 0 (相同数字异或为0)
* 2. a ^ 0 = a (任何数与0异或等于自身)
* 3. 异或运算满足交换律和结合律
*
* 因此，将数组中所有元素进行异或运算，出现两次的元素会相互抵消为0，最后只剩下只出现一次的元素。
*
* 时间复杂度：O(n)
* 空间复杂度：O(1)
*
* 补充题目：
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境，避免使用复杂的STL容器和标准库函数
// 使用基本的C++实现方式和自定义IO函数

class Code07_SingleNumber {
public:
    /**
     * 找出数组中只出现一次的元素
     * 利用异或运算的性质：
     * 1. 相同数字异或为0: a ^ a = 0
     * 2. 任何数与0异或等于自身: a ^ 0 = a
     * 3. 异或运算满足交换律和结合律
     *
     * @param nums 输入数组
     * @param numsSize 数组长度
     * @return 只出现一次的元素
     */
    int singleNumber(int* nums, int numsSize) {
        // 利用异或运算的性质：
        // 1. 相同数字异或为0: a ^ a = 0
        // 2. 任何数与0异或等于自身: a ^ 0 = a
    }
}

```

```

// 3. 异或运算满足交换律和结合律
int result = 0;
for (int i = 0; i < numsSize; i++) {
    result ^= nums[i];
}
return result;
};

// 测试方法
int main() {
    Code07_SingleNumber solution;

    // 测试用例 1: 正常情况
    int nums1[] = {2, 2, 1};
    int result1 = solution.singleNumber(nums1, 3);
    // 预期结果: 1

    // 测试用例 2: 正常情况
    int nums2[] = {4, 1, 2, 1, 2};
    int result2 = solution.singleNumber(nums2, 5);
    // 预期结果: 4

    // 测试用例 3: 边界情况
    int nums3[] = {1};
    int result3 = solution.singleNumber(nums3, 1);
    // 预期结果: 1

    return 0;
}

```

=====

文件: Code07\_SingleNumber.java

=====

```

package class031;

// 只出现一次的数字 - Single Number
// 测试链接 : https://leetcode.cn/problems/single-number/
// 相关题目:
// 1. 只出现一次的数字 II - Single Number II: https://leetcode.cn/problems/single-number-ii/
// 2. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/
// 3. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/

```

```
// 4. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
// 5. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
```

```
/*
```

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。

示例:

输入: nums = [2, 2, 1]

输出: 1

输入: nums = [4, 1, 2, 1, 2]

输出: 4

输入: nums = [1]

输出: 1

提示:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$

除了某个元素只出现一次以外，其余每个元素均出现两次。

解题思路:

这是一个经典的位运算问题，利用异或运算的性质:

1.  $a \wedge a = 0$  (相同数字异或为0)
2.  $a \wedge 0 = a$  (任何数与0异或等于自身)
3. 异或运算满足交换律和结合律

因此，将数组中所有元素进行异或运算，出现两次的元素会相互抵消为0，最后只剩下只出现一次的元素。

时间复杂度:  $O(n)$

空间复杂度:  $O(1)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
*/
```

```
public class Code07_SingleNumber {
```

```

/**
 * 找出数组中只出现一次的元素
 * 利用异或运算的性质:
 * 1. 相同数字异或为 0: a ^ a = 0
 * 2. 任何数与 0 异或等于自身: a ^ 0 = a
 * 3. 异或运算满足交换律和结合律
 *
 * @param nums 输入数组
 * @return 只出现一次的元素
 */
public static int singleNumber(int[] nums) {
    // 利用异或运算的性质:
    // 1. 相同数字异或为 0: a ^ a = 0
    // 2. 任何数与 0 异或等于自身: a ^ 0 = a
    // 3. 异或运算满足交换律和结合律
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

// 测试方法
public static void main(String[] args) {
    int[] test1 = {2, 2, 1};
    int[] test2 = {4, 1, 2, 1, 2};
    int[] test3 = {1};

    System.out.println("Test 1: " + singleNumber(test1)); // 输出: 1
    System.out.println("Test 2: " + singleNumber(test2)); // 输出: 4
    System.out.println("Test 3: " + singleNumber(test3)); // 输出: 1
}
}

```

文件: Code07\_SingleNumber.py

"""

只出现一次的数字 - Single Number

测试链接 : <https://leetcode.cn/problems/single-number/>

相关题目:

1. 只出现一次的数字 II - Single Number II: <https://leetcode.cn/problems/single-number-ii/>
2. 只出现一次的数字 III - Single Number III: <https://leetcode.cn/problems/single-number-iii/>
3. 缺失的数字 - Missing Number: <https://leetcode.cn/problems/missing-number/>
4. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
5. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>

#### 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。

#### 示例:

输入: nums = [2, 2, 1]

输出: 1

输入: nums = [4, 1, 2, 1, 2]

输出: 4

输入: nums = [1]

输出: 1

#### 提示:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$

除了某个元素只出现一次以外，其余每个元素均出现两次。

#### 解题思路:

这是一个经典的位运算问题，利用异或运算的性质：

1.  $a \wedge a = 0$  (相同数字异或为0)
2.  $a \wedge 0 = a$  (任何数与0异或等于自身)
3. 异或运算满足交换律和结合律

因此，将数组中所有元素进行异或运算，出现两次的元素会相互抵消为0，最后只剩下只出现一次的元素。

时间复杂度:  $O(n)$

空间复杂度:  $O(1)$

#### 补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

"""

```
class Solution:  
    def singleNumber(self, nums: list[int]) -> int:  
        """
```

找出数组中只出现一次的元素

利用异或运算的性质：

1. 相同数字异或为 0:  $a \wedge a = 0$
2. 任何数与 0 异或等于自身:  $a \wedge 0 = a$
3. 异或运算满足交换律和结合律

:param nums: 输入数组

:return: 只出现一次的元素

```
"""
```

# 利用异或运算的性质：

- ```
# 1. 相同数字异或为 0:  $a \wedge a = 0$   
# 2. 任何数与 0 异或等于自身:  $a \wedge 0 = a$   
# 3. 异或运算满足交换律和结合律
```

```
result = 0
```

```
for num in nums:
```

```
    result ^= num
```

```
return result
```

```
# 测试方法
```

```
if __name__ == "__main__":  
    solution = Solution()
```

```
# 测试用例 1：正常情况
```

```
nums1 = [2, 2, 1]
```

```
result1 = solution.singleNumber(nums1)
```

# 预期结果: 1

```
print(f"测试用例 1 - 输入: {nums1}")
```

```
print(f"结果: {result1} (预期: 1)")
```

```
# 测试用例 2：正常情况
```

```
nums2 = [4, 1, 2, 1, 2]
```

```
result2 = solution.singleNumber(nums2)
```

# 预期结果: 4

```
print(f"测试用例 2 - 输入: {nums2}")
```

```
print(f"结果: {result2} (预期: 4)")
```

```
# 测试用例 3：边界情况
```

```
nums3 = [1]
result3 = solution.singleNumber(nums3)
# 预期结果: 1
print(f"测试用例 3 - 输入: {nums3}")
print(f"结果: {result3} (预期: 1)")

=====
```

文件: Code08\_CountingBits.cpp

```
=====
/***
 * 比特位计数 - Counting Bits
 * 测试链接 : https://leetcode.cn/problems/counting-bits/
 * 相关题目:
 * 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/
 * 2. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
 * 3. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
 * 4. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/
 * 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/
 *
 * 题目描述:
 * 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。
 *
 * 示例:
 * 输入: 2
 * 输出: [0, 1, 1]
 *
 * 输入: 5
 * 输出: [0, 1, 1, 2, 1, 2]
 *
 * 解释:
 * 0 --> 0
 * 1 --> 1
 * 2 --> 10
 * 3 --> 11
 * 4 --> 100
 * 5 --> 101
 *
 * 提示:
 *  $0 \leq num \leq 10^5$ 
 *
 * 进阶:
```

\* 1. 给出时间复杂度为  $O(n * \text{sizeof(integer)})$  的解答非常容易。但你可以在线性时间  $O(n)$  内用一趟扫描做到吗？

\* 2. 要求算法的空间复杂度为  $O(n)$ 。

\* 3. 你能进一步完善解法吗？要求在 C++ 或任何其他语言中不使用任何内置函数（如 C++ 中的 `_builtin_popcount`）来执行此操作。

\*

\* 解题思路：

\* 这道题可以使用动态规划来解决，关键是要发现数字之间的规律。

\*

\* 对于任意一个数字  $i$ ：

\* 1. 如果  $i$  是偶数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 0，所以 1 的个数和  $i/2$  相同。

\* 2. 如果  $i$  是奇数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 1，所以 1 的个数是  $i/2$  的 1 的个数加 1。

\*

\* 也可以通过 Brian Kernighan 算法来理解： $i \& (i-1)$  可以移除  $i$  最右边的 1，所以  $i$  的 1 的个数等于  $i \& (i-1)$  的 1 的个数加 1。

\*

\* 时间复杂度： $O(n)$

\* 空间复杂度： $O(1)$ （不考虑返回数组）

\*

\* 补充题目：

\* 1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>

\* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>

\* 3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>

\* 4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>

\* 5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

\*/

// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数

// 使用基本的 C++ 实现方式和自定义 IO 函数

```
class Code08_CountingBits {
public:
    /**
     * 计算 0 到 num 范围内每个数字二进制表示中 1 的个数
     * 使用动态规划方法：
     * 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
     * i >> 1 相当于 i / 2 (整数除法)
     * i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
     *
     * @param num 非负整数
     * @param returnSize 返回数组的大小
     * @return 结果数组，ans[i] 表示 i 的二进制中 1 的个数
    */
}
```

```

int* countBits(int num, int* returnSize) {
    *returnSize = num + 1;
    int* ans = new int[*returnSize];
    ans[0] = 0;

    // 动态规划方法
    // 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
    // i >> 1 相当于 i / 2 (整数除法)
    // i & 1 判断 i 是否为奇数, 奇数为 1, 偶数为 0
    for (int i = 1; i <= num; i++) {
        ans[i] = ans[i >> 1] + (i & 1);
    }

    return ans;
}

};

// 测试方法
int main() {
    Code08_CountingBits solution;

    // 测试用例 1: 正常情况
    int returnSize1;
    int* result1 = solution.countBits(2, &returnSize1);
    // 预期结果: [0, 1, 1]

    // 测试用例 2: 正常情况
    int returnSize2;
    int* result2 = solution.countBits(5, &returnSize2);
    // 预期结果: [0, 1, 1, 2, 1, 2]

    return 0;
}

```

=====

文件: Code08\_CountingBits.java

=====

```

package class031;

// 比特位计数 - Counting Bits
// 测试链接 : https://leetcode.cn/problems/counting-bits/
// 相关题目:

```

```
// 1. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/
// 2. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 3. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
// 4. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/
// 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/
```

/\*

题目描述:

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。

示例:

输入: 2

输出: [0, 1, 1]

输入: 5

输出: [0, 1, 1, 2, 1, 2]

解释:

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

提示:

$0 \leq num \leq 10^5$

进阶:

- 给出时间复杂度为  $O(n * \text{sizeof(integer)})$  的解答非常容易。但你可以在线性时间  $O(n)$  内用一趟扫描做到吗？
- 要求算法的空间复杂度为  $O(n)$ 。
- 你能进一步完善解法吗？要求在 C++ 或任何其他语言中不使用任何内置函数（如 C++ 中的 `_builtin_popcount`）来执行此操作。

解题思路:

这道题可以使用动态规划来解决，关键是要发现数字之间的规律。

对于任意一个数字  $i$ :

- 如果  $i$  是偶数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 0，所以 1 的个数和  $i/2$  相同。
- 如果  $i$  是奇数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 1，所以 1 的个数是  $i/2$  的 1 的个数加 1。

也可以通过 Brian Kernighan 算法来理解： $i \& (i-1)$  可以移除  $i$  最右边的 1，所以  $i$  的 1 的个数等于  $i \& (i-1)$  的 1 的个数加 1。

时间复杂度：O(n)

空间复杂度：O(1)（不考虑返回数组）

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
/*
 * 计算 0 到 num 范围内每个数字二进制表示中 1 的个数
 * 使用动态规划方法：
 * 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
 * i >> 1 相当于 i / 2 (整数除法)
 * i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
 *
 * @param num 非负整数
 * @return 结果数组，ans[i] 表示 i 的二进制中 1 的个数
 */
public class Code08_CountingBits {

    public static int[] countBits(int num) {
        int[] ans = new int[num + 1];

        // 动态规划方法
        // 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
        // i >> 1 相当于 i / 2 (整数除法)
        // i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
        for (int i = 1; i <= num; i++) {
            ans[i] = ans[i >> 1] + (i & 1);
        }

        return ans;
    }

    // 另一种实现方式（使用 Brian Kernighan 算法的思想）
    // public static int[] countBits(int num) {
    //     int[] ans = new int[num + 1];
    // }
```

```

// 对于每个数字 i, ans[i] = ans[i & (i - 1)] + 1
// i & (i - 1) 可以移除 i 最右边的 1
// for (int i = 1; i <= num; i++) {
//     ans[i] = ans[i & (i - 1)] + 1;
// }
//
// return ans;
// }

// 测试方法
public static void main(String[] args) {
    int[] result1 = countBits(2);
    int[] result2 = countBits(5);

    System.out.print("countBits(2): ");
    for (int i : result1) {
        System.out.print(i + " ");
    }
    System.out.println(); // 输出: 0 1 1

    System.out.print("countBits(5): ");
    for (int i : result2) {
        System.out.print(i + " ");
    }
    System.out.println(); // 输出: 0 1 1 2 1 2
}

}

```

文件: Code08\_CountingBits.py

=====

比特位计数 - Counting Bits

测试链接 : <https://leetcode.cn/problems/counting-bits/>

相关题目:

1. 位 1 的个数 - Number of 1 Bits: <https://leetcode.cn/problems/number-of-1-bits/>
2. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
3. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>
4. 汉明距离总和 - Total Hamming Distance: <https://leetcode.cn/problems/total-hamming-distance/>
5. 颠倒二进制位 - Reverse Bits: <https://leetcode.cn/problems/reverse-bits/>

### 题目描述:

给定一个非负整数  $\text{num}$ 。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。

### 示例:

输入: 2

输出: [0, 1, 1]

输入: 5

输出: [0, 1, 1, 2, 1, 2]

### 解释:

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

### 提示:

$0 \leq \text{num} \leq 10^5$

### 进阶:

- 给出时间复杂度为  $O(n * \text{sizeof(integer)})$  的解答非常容易。但你可以在线性时间  $O(n)$  内用一趟扫描做到吗？
- 要求算法的空间复杂度为  $O(n)$ 。
- 你能进一步完善解法吗？要求在 C++ 或任何其他语言中不使用任何内置函数（如 C++ 中的 `__builtin_popcount`）来执行此操作。

### 解题思路:

这道题可以使用动态规划来解决，关键是要发现数字之间的规律。

对于任意一个数字  $i$ :

- 如果  $i$  是偶数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 0，所以 1 的个数和  $i/2$  相同。
- 如果  $i$  是奇数，那么它的二进制表示就是在  $i/2$  的二进制表示后面加一个 1，所以 1 的个数是  $i/2$  的 1 的个数加 1。

也可以通过 Brian Kernighan 算法来理解： $i \& (i-1)$  可以移除  $i$  最右边的 1，所以  $i$  的 1 的个数等于  $i \& (i-1)$  的 1 的个数加 1。

时间复杂度:  $O(n)$

空间复杂度:  $O(1)$  (不考虑返回数组)

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def countBits(self, num: int) -> list[int]:
```

```
        """
```

```
            计算 0 到 num 范围内每个数字二进制表示中 1 的个数
```

```
            使用动态规划方法：
```

```
            对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
```

```
            i >> 1 相当于 i / 2 (整数除法)
```

```
            i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
```

```
        :param num: 非负整数
```

```
        :return: 结果数组，ans[i] 表示 i 的二进制中 1 的个数
```

```
        """
```

```
        ans = [0] * (num + 1)
```

```
# 动态规划方法
```

```
# 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
```

```
# i >> 1 相当于 i / 2 (整数除法)
```

```
# i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
```

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

```
            ans[i] = ans[i >> 1] + (i & 1)
```

```
        return ans
```

```
# 测试方法
```

```
if __name__ == "__main__":  
    solution = Solution()
```

```
# 测试用例 1：正常情况
```

```
result1 = solution.countBits(2)
```

```
# 预期结果: [0, 1, 1]
```

```
print(f"countBits(2): {result1} (预期: [0, 1, 1])")
```

```
# 测试用例 2: 正常情况
result2 = solution.countBits(5)
# 预期结果: [0, 1, 1, 2, 1, 2]
print(f"countBits(5): {result2} (预期: [0, 1, 1, 2, 1, 2])")
```

---

文件: Code09\_NumberOf1Bits.cpp

---

```
/*
 * 位 1 的个数 - Number of 1 Bits
 * 测试链接 : https://leetcode.cn/problems/number-of-1-bits/
 * 相关题目:
 * 1. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
 * 2. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
 * 3. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/
 * 4. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
 * 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/
 *
 * 题目描述:
 * 编写一个函数，输入是一个无符号整数（以二进制串的形式），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）。
 *
 * 示例:
 * 输入: n = 11 (控制台输入 00000000000000000000000000001011)
 * 输出: 3
 * 解释: 输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
 *
 * 输入: n = 128 (控制台输入 00000000000000000000000000001000)
 * 输出: 1
 * 解释: 输入的二进制串 00000000000000000000000000001000 中，共有一位为 '1'。
 *
 * 输入: n = 4294967293 (控制台输入 1111111111111111111111111101, 部分语言中 n = -3)
 * 输出: 31
 * 解释: 输入的二进制串 1111111111111111111111111101 中，共有 31 位为 '1'。
 *
 * 提示:
 * 输入必须是长度为 32 的 二进制串 。
 * 注意: 本题与主站 191 题相同
 *
 * 解题思路:
 * 这道题要求计算一个无符号整数的二进制表示中 1 的个数，有多种解法:
 *
```

```
* 方法 1: 逐位检查
* 通过循环 32 次, 每次检查最低位是否为 1, 然后右移一位。
*
* 方法 2: Brian Kernighan 算法
* 通过  $n \& (n-1)$  可以移除  $n$  最右边的 1, 直到  $n$  为 0 为止, 统计操作次数。
*
* 方法 3: 分治法
* 通过并行计算每两位、每四位、每八位... 中 1 的个数, 和 Code06_CountOnesBinarySystem 中实现的一样。
*
* 时间复杂度:
* 方法 1:  $O(1)$  (固定 32 次操作)
* 方法 2:  $O(k)$  ( $k$  为 1 的个数)
* 方法 3:  $O(1)$ 
*
* 空间复杂度:  $O(1)$ 
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数
```

```
class Code09_NumberOf1Bits {
public:
    /**
     * 计算一个整数的二进制表示中 1 的个数 (汉明重量)
     * 使用逐位检查方法: 循环 32 次, 每次检查最低位是否为 1
     *
     * @param n 无符号整数
     * @return 二进制表示中 1 的个数
     */
    int hammingWeight(unsigned int n) {
        int count = 0;
        // 逐位检查 32 位
        for (int i = 0; i < 32; i++) {
            // 检查最低位是否为 1
            if ((n & (1 << i)) != 0) {
                count++;
            }
        }
    }
}
```

```

    }

    return count;
}

// 方法 2: Brian Kernighan 算法
// int hammingWeight(unsigned int n) {
//     int count = 0;
//     // n 不为 0 时继续循环
//     while (n != 0) {
//         count++;
//         // 移除最右边的 1
//         n &= n - 1;
//     }
//     return count;
// }

// 方法 3: 分治法 (和 Code06 中的 cntOnes 方法一样)
// int hammingWeight(unsigned int 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;
// }

};

// 测试方法
int main() {
    Code09_NumberOf1Bits solution;

    // 测试用例 1: 正常情况
    unsigned int n1 = 11;
    int result1 = solution.hammingWeight(n1);
    // 预期结果: 3

    // 测试用例 2: 正常情况
    unsigned int n2 = 128;
    int result2 = solution.hammingWeight(n2);
    // 预期结果: 1

    // 测试用例 3: 边界情况
    unsigned int n3 = 4294967293; // -3 的无符号表示
}

```

```
int result3 = solution.hammingWeight(n3);
// 预期结果: 31

return 0;
}
```

=====

文件: Code09\_NumberOf1Bits.java

=====

```
package class031;

// 位 1 的个数 - Number of 1 Bits
// 测试链接 : https://leetcode.cn/problems/number-of-1-bits/
// 相关题目:
// 1. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
// 2. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
// 3. 汉明距离总和 - Total Hamming Distance: https://leetcode.cn/problems/total-hamming-distance/
// 4. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 5. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/
```

/\*

题目描述:

编写一个函数，输入是一个无符号整数（以二进制串的形式），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）。

示例:

输入: n = 11 (控制台输入 00000000000000000000000000001011)

输出: 3

解释: 输入的二进制串 00000000000000000000000000001011 中，共有三位为 ’1’。

输入: n = 128 (控制台输入 00000000000000000000000000001000000)

输出: 1

解释: 输入的二进制串 00000000000000000000000000001000000 中，共有一位为 ’1’。

输入: n = 4294967293 (控制台输入 1111111111111111111111111101, 部分语言中 n = -3)

输出: 31

解释: 输入的二进制串 1111111111111111111111111101 中，共有 31 位为 ’1’。

提示:

输入必须是长度为 32 的 二进制串 。

注意: 本题与主站 191 题相同

解题思路:

这道题要求计算一个无符号整数的二进制表示中 1 的个数，有多种解法：

方法 1：逐位检查

通过循环 32 次，每次检查最低位是否为 1，然后右移一位。

方法 2：Brian Kernighan 算法

通过  $n \& (n-1)$  可以移除  $n$  最右边的 1，直到  $n$  为 0 为止，统计操作次数。

方法 3：分治法

通过并行计算每两位、每四位、每八位... 中 1 的个数，和 Code06\_CountOnesBinarySystem 中实现的一样。

时间复杂度：

方法 1:  $O(1)$  (固定 32 次操作)

方法 2:  $O(k)$  ( $k$  为 1 的个数)

方法 3:  $O(1)$

空间复杂度:  $O(1)$

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
/*
 * 计算一个整数的二进制表示中 1 的个数 (汉明重量)
 * 使用逐位检查方法：循环 32 次，每次检查最低位是否为 1
 *
 * @param n 无符号整数
 * @return 二进制表示中 1 的个数
 */
public class Code09_NumberOf1Bits {

    // you need to treat n as an unsigned value
    /**
     * 计算一个整数的二进制表示中 1 的个数 (汉明重量)
     * 使用逐位检查方法：循环 32 次，每次检查最低位是否为 1
     *
     * @param n 无符号整数
     * @return 二进制表示中 1 的个数
     */
    public static int hammingWeight(int n) {
        int count = 0;
        // 逐位检查 32 位
        for (int i = 0; i < 32; i++) {
            // 检查最低位是否为 1
            if ((n & (1 << i)) != 0) {
                count++;
            }
        }
        return count;
    }
}
```

```

    }
}

return count;
}

// 方法 2: Brian Kernighan 算法
// public static int hammingWeight(int n) {
//     int count = 0;
//     // n 不为 0 时继续循环
//     while (n != 0) {
//         count++;
//         // 移除最右边的 1
//         n &= n - 1;
//     }
//     return count;
// }

// 方法 3: 分治法 (和 Code06 中的 cntOnes 方法一样)
// public static int hammingWeight(int 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;
// }

// 测试方法
public static void main(String[] args) {
    System.out.println(hammingWeight(11));           // 输出: 3
    System.out.println(hammingWeight(128));          // 输出: 1
    System.out.println(hammingWeight(-3));           // 输出: 31 (4294967293)
}

}

```

文件: Code09\_NumberOf1Bits.py

位 1 的个数 – Number of 1 Bits

测试链接 : <https://leetcode.cn/problems/number-of-1-bits/>

相关题目：

1. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
2. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>
3. 汉明距离总和 - Total Hamming Distance: <https://leetcode.cn/problems/total-hamming-distance/>
4. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
5. 颠倒二进制位 - Reverse Bits: <https://leetcode.cn/problems/reverse-bits/>

题目描述：

编写一个函数，输入是一个无符号整数（以二进制串的形式），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）。

示例：

输入：n = 11 （控制台输入 00000000000000000000000000001011）

输出：3

解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 ’1’。

输入：n = 128 （控制台输入 000000000000000000000000000010000000）

输出：1

解释：输入的二进制串 000000000000000000000000000010000000 中，共有一位为 ’1’。

输入：n = 4294967293 （控制台输入 111111111111111111111111111101，部分语言中 n = -3）

输出：31

解释：输入的二进制串 111111111111111111111111111101 中，共有 31 位为 ’1’。

提示：

输入必须是长度为 32 的 二进制串 。

注意：本题与主站 191 题相同

解题思路：

这道题要求计算一个无符号整数的二进制表示中 1 的个数，有多种解法：

方法 1：逐位检查

通过循环 32 次，每次检查最低位是否为 1，然后右移一位。

方法 2：Brian Kernighan 算法

通过  $n \& (n-1)$  可以移除 n 最右边的 1，直到 n 为 0 为止，统计操作次数。

方法 3：分治法

通过并行计算每两位、每四位、每八位... 中 1 的个数，和 Code06\_CountOnesBinarySystem 中实现的一样。

时间复杂度：

方法 1：O(1)（固定 32 次操作）

方法 2：O(k)（k 为 1 的个数）

方法 3: O(1)

空间复杂度: O(1)

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def hammingWeight(self, n: int) -> int:  
        """  
        计算一个整数的二进制表示中 1 的个数（汉明重量）  
        使用逐位检查方法：循环 32 次，每次检查最低位是否为 1  
        """
```

```
:param n: 无符号整数  
:return: 二进制表示中 1 的个数  
"""  
count = 0  
# 逐位检查 32 位  
for i in range(32):  
    # 检查最低位是否为 1  
    if (n & (1 << i)) != 0:  
        count += 1  
return count
```

```
# 方法 2: Brian Kernighan 算法  
# def hammingWeight(self, n: int) -> int:  
#     count = 0  
#     # n 不为 0 时继续循环  
#     while n != 0:  
#         count += 1  
#         # 移除最右边的 1  
#         n &= n - 1  
#     return count
```

```
# 方法 3: 分治法（和 Code06 中的 cntOnes 方法一样）  
# def hammingWeight(self, n: int) -> int:  
#     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
```

```
# 测试方法
```

```
if __name__ == "__main__":
    solution = Solution()
```

```
# 测试用例 1: 正常情况
```

```
n1 = 11
result1 = solution.hammingWeight(n1)
# 预期结果: 3
print(f"测试用例 1 - 输入: {n1}")
print(f"结果: {result1} (预期: 3)")
```

```
# 测试用例 2: 正常情况
```

```
n2 = 128
result2 = solution.hammingWeight(n2)
# 预期结果: 1
print(f"测试用例 2 - 输入: {n2}")
print(f"结果: {result2} (预期: 1)")
```

```
# 测试用例 3: 边界情况
```

```
n3 = 4294967293 # -3 的无符号表示
result3 = solution.hammingWeight(n3)
# 预期结果: 31
print(f"测试用例 3 - 输入: {n3}")
print(f"结果: {result3} (预期: 31)")
```

---

```
文件: Code10_SingleNumberII.cpp
```

---

```
/**
```

```
* 只出现一次的数字 II - Single Number II
* 测试链接 : https://leetcode.cn/problems/single-number-ii/
* 相关题目:
* 1. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/
* 2. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/
* 3. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/
```

- \* 4. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
- \* 5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
- \*
- \* 题目描述:
- \* 给你一个整数数组 `nums`，除某个元素仅出现一次外，其余每个元素都恰出现三次。请你找出并返回那个只出现了一次的元素。
- \*
- \* 示例:
- \* 输入: `nums = [2, 2, 3, 2]`
- \* 输出: 3
- \*
- \* 输入: `nums = [0, 1, 0, 1, 0, 1, 99]`
- \* 输出: 99
- \*
- \* 提示:
- \*  $1 \leq \text{nums.length} \leq 3 * 10^4$
- \*  $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- \* `nums` 中，除某个元素仅出现一次外，其余每个元素都恰出现三次
- \*
- \* 进阶：你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？
- \*
- \* 解题思路:
- \* 这道题是“只出现一次的数字”的升级版。在那道题中，其他元素出现2次，可以用异或运算解决。
- \* 但这道题中，其他元素出现3次，异或运算就不适用了。
- \*
- \* 方法1: 哈希表统计
- \* 用哈希表统计每个元素出现的次数，然后找出出现次数为1的元素。
- \* 时间复杂度:  $O(n)$
- \* 空间复杂度:  $O(n)$
- \*
- \* 方法2: 位运算(重点讲解)
- \* 对于每个二进制位，统计所有数字在该位上1的个数。
- \* 由于除了一个数字外，其他数字都出现3次，所以每个二进制位上1的个数模3的结果就是单独那个数字在该位上的值。
- \*
- \* 我们用两个变量 `ones` 和 `twos` 来表示每个二进制位上1的个数模3的结果:
- \* - `ones`: 表示当前出现1次的位
- \* - `twos`: 表示当前出现2次的位
- \*
- \* 状态转移:
- \* - 当一个数字出现第1次时，它应该加到 `ones` 中
- \* - 当一个数字出现第2次时，它应该从 `ones` 中移除，加到 `twos` 中
- \* - 当一个数字出现第3次时，它应该从 `twos` 中移除

```

*
* 通过位运算可以实现这个状态转移:
* ones = (ones ^ num) & ~twos;
* twos = (twos ^ num) & ~ones;
*
* 时间复杂度: O(n)
* 空间复杂度: O(1)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code10_SingleNumberII {
public:
    /**
     * 找出数组中只出现一次的元素 (其他元素都出现 3 次)
     * 使用位运算方法:
     * 用两个变量 ones 和 twos 来表示每个二进制位上 1 的个数模 3 的结果
     * ones: 表示当前出现 1 次的位
     * twos: 表示当前出现 2 次的位
     *
     * @param nums 输入数组
     * @param numsSize 数组长度
     * @return 只出现一次的元素
     */
    int singleNumber(int* nums, int numsSize) {
        // ones 表示当前出现 1 次的位
        // twos 表示当前出现 2 次的位
        int ones = 0, twos = 0;

        for (int i = 0; i < numsSize; i++) {
            int num = nums[i];
            // 更新 ones 和 twos 的状态
            // (ones ^ num) 将 num 中为 1 的位在 ones 中翻转
            // & ~twos 确保这些位不在 twos 中 (即不是出现 2 次的位)
            ones = (ones ^ num) & ~twos;
            twos = (twos ^ num) & ~ones;
        }
        return ones;
    }
}

```

```

    // (twos ^ num) 将 num 中为 1 的位在 twos 中翻转
    // & ~ones 确保这些位不在 ones 中 (即不是出现 1 次的位)
    twos = (twos ^ num) & ~ones;
}

// 最终 ones 中保存的就是只出现一次的数字
return ones;
}

};

// 测试方法
int main() {
    Code10_SingleNumberII solution;

    // 测试用例 1: 正常情况
    int nums1[] = {2, 2, 3, 2};
    int result1 = solution.singleNumber(nums1, 4);
    // 预期结果: 3

    // 测试用例 2: 正常情况
    int nums2[] = {0, 1, 0, 1, 0, 1, 99};
    int result2 = solution.singleNumber(nums2, 7);
    // 预期结果: 99

    return 0;
}

```

=====

文件: Code10\_SingleNumberII.java

=====

```

package class031;

// 只出现一次的数字 II - Single Number II
// 测试链接 : https://leetcode.cn/problems/single-number-ii/
// 相关题目:
// 1. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/
// 2. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/
// 3. 缺失的数字 - Missing Number: https://leetcode.cn/problems/missing-number/
// 4. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
// 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/

/*

```

### 题目描述:

给你一个整数数组 `nums`，除某个元素仅出现一次外，其余每个元素都恰出现三次。请你找出并返回那个只出现了一次的元素。

### 示例:

输入: `nums = [2, 2, 3, 2]`

输出: 3

输入: `nums = [0, 1, 0, 1, 0, 1, 99]`

输出: 99

### 提示:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

`nums` 中，除某个元素仅出现一次外，其余每个元素都恰出现三次

进阶：你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

### 解题思路:

这道题是“只出现一次的数字”的升级版。在那道题中，其他元素出现 2 次，可以用异或运算解决。

但这道题中，其他元素出现 3 次，异或运算就不适用了。

### 方法 1：哈希表统计

用哈希表统计每个元素出现的次数，然后找出出现次数为 1 的元素。

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

### 方法 2：位运算（重点讲解）

对于每个二进制位，统计所有数字在该位上 1 的个数。

由于除了一个数字外，其他数字都出现 3 次，所以每个二进制位上 1 的个数模 3 的结果就是单独那个数字在该位上的值。

我们用两个变量 `ones` 和 `twos` 来表示每个二进制位上 1 的个数模 3 的结果：

- `ones`: 表示当前出现 1 次的位

- `twos`: 表示当前出现 2 次的位

### 状态转移:

- 当一个数字出现第 1 次时，它应该加到 `ones` 中

- 当一个数字出现第 2 次时，它应该从 `ones` 中移除，加到 `twos` 中

- 当一个数字出现第 3 次时，它应该从 `twos` 中移除

通过位运算可以实现这个状态转移：

`ones = (ones ^ num) & ~twos;`

```
twos = (twos ^ num) & ~ones;
```

时间复杂度: O(n)

空间复杂度: O(1)

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- \*/

```
public class Code10_SingleNumberII {
```

```
/**
```

```
* 找出数组中只出现一次的元素（其他元素都出现 3 次）
```

```
* 使用位运算方法:
```

```
* 用两个变量 ones 和 twos 来表示每个二进制位上 1 的个数模 3 的结果
```

```
* ones: 表示当前出现 1 次的位
```

```
* twos: 表示当前出现 2 次的位
```

```
*
```

```
* @param nums 输入数组
```

```
* @return 只出现一次的元素
```

```
*/
```

```
public static int singleNumber(int[] nums) {
```

```
    // ones 表示当前出现 1 次的位
```

```
    // twos 表示当前出现 2 次的位
```

```
    int ones = 0, twos = 0;
```

```
    for (int num : nums) {
```

```
        // 更新 ones 和 twos 的状态
```

```
        // (ones ^ num) 将 num 中为 1 的位在 ones 中翻转
```

```
        // & ~twos 确保这些位不在 twos 中（即不是出现 2 次的位）
```

```
        ones = (ones ^ num) & ~twos;
```

```
        // (twos ^ num) 将 num 中为 1 的位在 twos 中翻转
```

```
        // & ~ones 确保这些位不在 ones 中（即不是出现 1 次的位）
```

```
        twos = (twos ^ num) & ~ones;
```

```
}
```

```
    // 最终 ones 中保存的就是只出现一次的数字
```

```
    return ones;
```

```
}
```

```

// 方法 1：哈希表统计
// public static int singleNumber(int[] nums) {
//     Map<Integer, Integer> map = new HashMap<>();
// 
//     // 统计每个数字出现的次数
//     for (int num : nums) {
//         map.put(num, map.getOrDefault(num, 0) + 1);
//     }
// 
//     // 找出出现次数为 1 的数字
//     for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
//         if (entry.getValue() == 1) {
//             return entry.getKey();
//         }
//     }
// 
//     return -1; // 不会执行到这里
// }

// 测试方法
public static void main(String[] args) {
    int[] test1 = {2, 2, 3, 2};
    int[] test2 = {0, 1, 0, 1, 0, 1, 99};

    System.out.println("Test 1: " + singleNumber(test1)); // 输出: 3
    System.out.println("Test 2: " + singleNumber(test2)); // 输出: 99
}

}

```

---

文件: Code10\_SingleNumberII.py

---

"""

只出现一次的数字 II – Single Number II

测试链接 : <https://leetcode.cn/problems/single-number-ii/>

相关题目:

1. 只出现一次的数字 – Single Number: <https://leetcode.cn/problems/single-number/>
2. 只出现一次的数字 III – Single Number III: <https://leetcode.cn/problems/single-number-iii/>
3. 缺失的数字 – Missing Number: <https://leetcode.cn/problems/missing-number/>
4. 找不同 – Find the Difference: <https://leetcode.cn/problems/find-the-difference/>

## 5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给你一个整数数组 `nums`，除某个元素仅出现一次外，其余每个元素都恰出现三次。请你找出并返回那个只出现了一次的元素。

示例:

输入: `nums = [2, 2, 3, 2]`

输出: 3

输入: `nums = [0, 1, 0, 1, 0, 1, 99]`

输出: 99

提示:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

`nums` 中，除某个元素仅出现一次外，其余每个元素都恰出现三次

进阶：你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

解题思路:

这道题是“只出现一次的数字”的升级版。在那道题中，其他元素出现 2 次，可以用异或运算解决。但这道题中，其他元素出现 3 次，异或运算就不适用了。

方法 1: 哈希表统计

用哈希表统计每个元素出现的次数，然后找出出现次数为 1 的元素。

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

方法 2: 位运算（重点讲解）

对于每个二进制位，统计所有数字在该位上 1 的个数。

由于除了一个数字外，其他数字都出现 3 次，所以每个二进制位上 1 的个数模 3 的结果就是单独那个数字在该位上的值。

我们用两个变量 `ones` 和 `twos` 来表示每个二进制位上 1 的个数模 3 的结果：

- `ones`: 表示当前出现 1 次的位
- `twos`: 表示当前出现 2 次的位

状态转移:

- 当一个数字出现第 1 次时，它应该加到 `ones` 中
- 当一个数字出现第 2 次时，它应该从 `ones` 中移除，加到 `twos` 中
- 当一个数字出现第 3 次时，它应该从 `twos` 中移除

通过位运算可以实现这个状态转移：

```
ones = (ones ^ num) & ~twos;
twos = (twos ^ num) & ~ones;
```

时间复杂度：O(n)

空间复杂度：O(1)

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def singleNumber(self, nums: list[int]) -> int:  
        """
```

找出数组中只出现一次的元素（其他元素都出现 3 次）

使用位运算方法：

用两个变量 ones 和 twos 来表示每个二进制位上 1 的个数模 3 的结果

ones: 表示当前出现 1 次的位

twos: 表示当前出现 2 次的位

:param nums: 输入数组

:return: 只出现一次的元素

"""

# ones 表示当前出现 1 次的位

# twos 表示当前出现 2 次的位

ones = 0

twos = 0

for num in nums:

# 更新 ones 和 twos 的状态

# (ones ^ num) 将 num 中为 1 的位在 ones 中翻转

# & ~twos 确保这些位不在 twos 中（即不是出现 2 次的位）

ones = (ones ^ num) & ~twos

# (twos ^ num) 将 num 中为 1 的位在 twos 中翻转

# & ~ones 确保这些位不在 ones 中（即不是出现 1 次的位）

twos = (twos ^ num) & ~ones

```
# 最终 ones 中保存的就是只出现一次的数字
return ones
```

```
# 测试方法
if __name__ == "__main__":
    solution = Solution()
```

```
# 测试用例 1: 正常情况
nums1 = [2, 2, 3, 2]
result1 = solution.singleNumber(nums1)
# 预期结果: 3
print(f"测试用例 1 - 输入: {nums1}")
print(f"结果: {result1} (预期: 3)")
```

```
# 测试用例 2: 正常情况
nums2 = [0, 1, 0, 1, 0, 1, 99]
result2 = solution.singleNumber(nums2)
# 预期结果: 99
print(f"测试用例 2 - 输入: {nums2}")
print(f"结果: {result2} (预期: 99)")
```

=====

文件: Code11\_MissingNumber.cpp

=====

```
/**
 * 缺失的数字 - Missing Number
 * 测试链接 : https://leetcode.cn/problems/missing-number/
 * 相关题目:
 * 1. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/
 * 2. 只出现一次的数字 II - Single Number II: https://leetcode.cn/problems/single-number-ii/
 * 3. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/
 * 4. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/
 * 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
 *
 * 题目描述:
 * 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。
 *
 * 示例:
 * 输入: nums = [3,0,1]
 * 输出: 2
 * 解释: n = 3, 因为有 3 个数字, 所以所有的数字都在范围 [0,3] 内。2 是丢失的数字。
```

\*

\* 输入: nums = [0, 1]  
\* 输出: 2  
\* 解释: n = 2, 因为有 2 个数字, 所以所有的数字都在范围 [0, 2] 内。2 是丢失的数字。  
\*

\* 输入: nums = [9, 6, 4, 2, 3, 5, 7, 0, 1]  
\* 输出: 8  
\* 解释: n = 9, 因为有 9 个数字, 所以所有的数字都在范围 [0, 9] 内。8 是丢失的数字。  
\*

\* 输入: nums = [0]  
\* 输出: 1  
\* 解释: n = 1, 因为有 1 个数字, 所以所有的数字都在范围 [0, 1] 内。1 是丢失的数字。  
\*

\* 提示:

\*  $n == \text{nums.length}$   
\*  $1 \leq n \leq 10^4$   
\*  $0 \leq \text{nums}[i] \leq n$   
\* nums 中的所有数字都独一无二

\*

\* 进阶: 你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?  
\*

\* 解题思路:

\* 这道题有多种解法:  
\*

\* 方法 1: 数学求和  
\* 计算 0 到 n 的和, 然后减去数组中所有元素的和, 差值就是缺失的数字。  
\* 时间复杂度:  $O(n)$   
\* 空间复杂度:  $O(1)$   
\*

\* 方法 2: 异或运算 (推荐)  
\* 利用异或运算的性质:  
\* 1.  $a \wedge a = 0$   
\* 2.  $a \wedge 0 = a$   
\* 3. 异或运算满足交换律和结合律  
\*

\* 我们将 0 到 n 的所有数字与数组中的所有元素进行异或运算, 出现两次的数字会相互抵消为 0, 最后只剩下缺失的数字。  
\*

\* 方法 3: 排序后查找  
\* 先对数组排序, 然后遍历查找缺失的数字。  
\* 时间复杂度:  $O(n \log n)$   
\* 空间复杂度:  $O(1)$   
\*

```

* 方法 4: 哈希表
* 用哈希表记录出现过的数字，然后遍历 0 到 n 查找缺失的数字。
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* 时间复杂度:
* 方法 1 和方法 2: O(n)
* 方法 3: O(n log n)
* 方法 4: O(n)
*
* 空间复杂度:
* 方法 1、2、3: O(1)
* 方法 4: O(n)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

class Code11_MissingNumber {
public:
    /**
     * 找出数组中缺失的数字
     * 使用异或运算方法:
     * 将 0 到 n 的所有数字与数组中的所有元素进行异或
     * 出现两次的数字会相互抵消为 0，最后只剩下缺失的数字
     *
     * @param nums 输入数组
     * @param numsSize 数组长度
     * @return 缺失的数字
     */
    int missingNumber(int* nums, int numsSize) {
        // 使用异或运算
        // 将 0 到 n 的所有数字与数组中的所有元素进行异或
        int xorResult = 0;

        // 与 0 到 n-1 的所有数字异或
        for (int i = 0; i < numsSize; i++) {
            xorResult ^= i;
        }

        for (int i = 0; i < numsSize; i++) {
            xorResult ^= nums[i];
        }

        return xorResult;
    }
}

```

```

        xorResult ^= i ^ nums[i];
    }

    // 最后与 n 异或（因为数组中只有 n 个元素，范围是 0 到 n）
    return xorResult ^ numsSize;
}

// 测试方法
int main() {
    Code11_MissingNumber solution;

    // 测试用例 1：正常情况
    int nums1[] = {3, 0, 1};
    int result1 = solution.missingNumber(nums1, 3);
    // 预期结果：2

    // 测试用例 2：正常情况
    int nums2[] = {0, 1};
    int result2 = solution.missingNumber(nums2, 2);
    // 预期结果：2

    // 测试用例 3：正常情况
    int nums3[] = {9, 6, 4, 2, 3, 5, 7, 0, 1};
    int result3 = solution.missingNumber(nums3, 9);
    // 预期结果：8

    // 测试用例 4：边界情况
    int nums4[] = {0};
    int result4 = solution.missingNumber(nums4, 1);
    // 预期结果：1

    return 0;
}

```

=====

文件：Code11\_MissingNumber.java

=====

```

package class031;

// 缺失的数字 - Missing Number
// 测试链接：https://leetcode.cn/problems/missing-number/

```

```
// 相关题目：  
// 1. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/  
// 2. 只出现一次的数字 II - Single Number II: https://leetcode.cn/problems/single-number-ii/  
// 3. 只出现一次的数字 III - Single Number III: https://leetcode.cn/problems/single-number-iii/  
// 4. 找不同 - Find the Difference: https://leetcode.cn/problems/find-the-difference/  
// 5. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
```

/\*

题目描述:

给定一个包含  $[0, n]$  中  $n$  个数的数组  $\text{nums}$ ，找出  $[0, n]$  这个范围内没有出现在数组中的那个数。

示例:

输入:  $\text{nums} = [3, 0, 1]$

输出: 2

解释:  $n = 3$ ，因为有 3 个数字，所以所有的数字都在范围  $[0, 3]$  内。2 是丢失的数字。

输入:  $\text{nums} = [0, 1]$

输出: 2

解释:  $n = 2$ ，因为有 2 个数字，所以所有的数字都在范围  $[0, 2]$  内。2 是丢失的数字。

输入:  $\text{nums} = [9, 6, 4, 2, 3, 5, 7, 0, 1]$

输出: 8

解释:  $n = 9$ ，因为有 9 个数字，所以所有的数字都在范围  $[0, 9]$  内。8 是丢失的数字。

输入:  $\text{nums} = [0]$

输出: 1

解释:  $n = 1$ ，因为有 1 个数字，所以所有的数字都在范围  $[0, 1]$  内。1 是丢失的数字。

提示:

$n == \text{nums.length}$

$1 \leq n \leq 10^4$

$0 \leq \text{nums}[i] \leq n$

$\text{nums}$  中的所有数字都独一无二

进阶: 你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?

解题思路:

这道题有多种解法:

方法 1: 数学求和

计算 0 到  $n$  的和，然后减去数组中所有元素的和，差值就是缺失的数字。

时间复杂度:  $O(n)$

空间复杂度:  $O(1)$

## 方法 2：异或运算（推荐）

利用异或运算的性质：

1.  $a \wedge a = 0$
2.  $a \wedge 0 = a$
3. 异或运算满足交换律和结合律

我们将 0 到 n 的所有数字与数组中的所有元素进行异或运算，出现两次的数字会相互抵消为 0，最后只剩下缺失的数字。

## 方法 3：排序后查找

先对数组排序，然后遍历查找缺失的数字。

时间复杂度： $O(n \log n)$

空间复杂度： $O(1)$

## 方法 4：哈希表

用哈希表记录出现过的数字，然后遍历 0 到 n 查找缺失的数字。

时间复杂度： $O(n)$

空间复杂度： $O(n)$

时间复杂度：

方法 1 和方法 2： $O(n)$

方法 3： $O(n \log n)$

方法 4： $O(n)$

空间复杂度：

方法 1、2、3： $O(1)$

方法 4： $O(n)$

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
*/
```

```
public class Code11_MissingNumber {
```

```
/**
```

```
* 找出数组中缺失的数字
```

```
* 使用异或运算方法：
```

```
* 将 0 到 n 的所有数字与数组中的所有元素进行异或
```

```
* 出现两次的数字会相互抵消为 0，最后只剩下缺失的数字
```

```
*  
* @param nums 输入数组  
* @return 缺失的数字  
*/  
  
public static int missingNumber(int[] nums) {  
    // 使用异或运算  
    // 将 0 到 n 的所有数字与数组中的所有元素进行异或  
    int xor = 0;  
    int n = nums.length;  
  
    // 与 0 到 n-1 的所有数字异或  
    for (int i = 0; i < n; i++) {  
        xor ^= i ^ nums[i];  
    }  
  
    // 最后与 n 异或（因为数组中只有 n 个元素，范围是 0 到 n）  
    return xor ^ n;  
}  
  
// 方法 1：求数学求和  
// public static int missingNumber(int[] nums) {  
//     int n = nums.length;  
//     // 计算 0 到 n 的和  
//     int expectedSum = n * (n + 1) / 2;  
//  
//     // 计算数组元素的和  
//     int actualSum = 0;  
//     for (int num : nums) {  
//         actualSum += num;  
//     }  
//  
//     // 差值就是缺失的数字  
//     return expectedSum - actualSum;  
// }  
  
// 测试方法  
public static void main(String[] args) {  
    int[] test1 = {3, 0, 1};  
    int[] test2 = {0, 1};  
    int[] test3 = {9, 6, 4, 2, 3, 5, 7, 0, 1};  
    int[] test4 = {0};  
  
    System.out.println("Test 1: " + missingNumber(test1)); // 输出: 2
```

```
        System.out.println("Test 2: " + missingNumber(test2)); // 输出: 2
        System.out.println("Test 3: " + missingNumber(test3)); // 输出: 8
        System.out.println("Test 4: " + missingNumber(test4)); // 输出: 1
    }
}
```

文件: Code11\_MissingNumber.py

```
=====
"""

```

缺失的数字 - Missing Number

测试链接 : <https://leetcode.cn/problems/missing-number/>

相关题目:

1. 只出现一次的数字 - Single Number: <https://leetcode.cn/problems/single-number/>
2. 只出现一次的数字 II - Single Number II: <https://leetcode.cn/problems/single-number-ii/>
3. 只出现一次的数字 III - Single Number III: <https://leetcode.cn/problems/single-number-iii/>
4. 找不同 - Find the Difference: <https://leetcode.cn/problems/find-the-difference/>
5. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给定一个包含  $[0, n]$  中  $n$  个数的数组  $\text{nums}$ ，找出  $[0, n]$  这个范围内没有出现在数组中的那个数。

示例:

输入:  $\text{nums} = [3, 0, 1]$

输出: 2

解释:  $n = 3$ ，因为有 3 个数字，所以所有的数字都在范围  $[0, 3]$  内。2 是丢失的数字。

输入:  $\text{nums} = [0, 1]$

输出: 2

解释:  $n = 2$ ，因为有 2 个数字，所以所有的数字都在范围  $[0, 2]$  内。2 是丢失的数字。

输入:  $\text{nums} = [9, 6, 4, 2, 3, 5, 7, 0, 1]$

输出: 8

解释:  $n = 9$ ，因为有 9 个数字，所以所有的数字都在范围  $[0, 9]$  内。8 是丢失的数字。

输入:  $\text{nums} = [0]$

输出: 1

解释:  $n = 1$ ，因为有 1 个数字，所以所有的数字都在范围  $[0, 1]$  内。1 是丢失的数字。

提示:

$n == \text{nums.length}$

$1 \leq n \leq 10^4$

$0 \leq \text{nums}[i] \leq n$

`nums` 中的所有数字都独一无二

进阶：你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题？

解题思路：

这道题有多种解法：

方法 1：数学求和

计算 0 到 n 的和，然后减去数组中所有元素的和，差值就是缺失的数字。

时间复杂度： $O(n)$

空间复杂度： $O(1)$

方法 2：异或运算（推荐）

利用异或运算的性质：

1.  $a \wedge a = 0$

2.  $a \wedge 0 = a$

3. 异或运算满足交换律和结合律

我们将 0 到 n 的所有数字与数组中的所有元素进行异或运算，出现两次的数字会相互抵消为 0，最后只剩下缺失的数字。

方法 3：排序后查找

先对数组排序，然后遍历查找缺失的数字。

时间复杂度： $O(n \log n)$

空间复杂度： $O(1)$

方法 4：哈希表

用哈希表记录出现过的数字，然后遍历 0 到 n 查找缺失的数字。

时间复杂度： $O(n)$

空间复杂度： $O(n)$

时间复杂度：

方法 1 和方法 2： $O(n)$

方法 3： $O(n \log n)$

方法 4： $O(n)$

空间复杂度：

方法 1、2、3： $O(1)$

方法 4： $O(n)$

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:  
    def missingNumber(self, nums: list[int]) -> int:  
        """  
        找出数组中缺失的数字  
        使用异或运算方法：  
        将 0 到 n 的所有数字与数组中的所有元素进行异或  
        出现两次的数字会相互抵消为 0，最后只剩下缺失的数字  
  
        :param nums: 输入数组  
        :return: 缺失的数字  
        """  
  
        # 使用异或运算  
        # 将 0 到 n 的所有数字与数组中的所有元素进行异或  
        xor_result = 0  
        n = len(nums)  
  
        # 与 0 到 n-1 的所有数字异或  
        for i in range(n):  
            xor_result ^= i ^ nums[i]  
  
        # 最后与 n 异或（因为数组中只有 n 个元素，范围是 0 到 n）  
        return xor_result ^ n  
  
# 测试方法  
if __name__ == "__main__":  
    solution = Solution()  
  
    # 测试用例 1：正常情况  
    nums1 = [3, 0, 1]  
    result1 = solution.missingNumber(nums1)  
    # 预期结果：2  
    print(f"测试用例 1 - 输入: {nums1}")  
    print(f"结果: {result1} (预期: 2)")
```

```
# 测试用例 2: 正常情况
nums2 = [0, 1]
result2 = solution.missingNumber(nums2)
# 预期结果: 2
print(f"测试用例 2 - 输入: {nums2}")
print(f"结果: {result2} (预期: 2)")

# 测试用例 3: 正常情况
nums3 = [9, 6, 4, 2, 3, 5, 7, 0, 1]
result3 = solution.missingNumber(nums3)
# 预期结果: 8
print(f"测试用例 3 - 输入: {nums3}")
print(f"结果: {result3} (预期: 8)")

# 测试用例 4: 边界情况
nums4 = [0]
result4 = solution.missingNumber(nums4)
# 预期结果: 1
print(f"测试用例 4 - 输入: {nums4}")
print(f"结果: {result4} (预期: 1)")

=====
```

文件: Code12\_SumOfTwoIntegers.cpp

```
/*
 * 两整数之和 - Sum of Two Integers
 * 测试链接 : https://leetcode.cn/problems/sum-of-two-integers/
 * 相关题目:
 * 1. 位运算技巧大全 - Bit Manipulation Tricks
 * 2. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/
 * 3. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
 * 4. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
 * 5. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
 *
 * 题目描述:
 * 给你两个整数 a 和 b , 不使用运算符 + 和 - , 计算并返回两整数之和。
 *
 * 示例:
 * 输入: a = 1, b = 2
 * 输出: 3
 *
 * 输入: a = 2, b = 3
```

```
* 输出: 5
*
* 提示:
*  $-1000 \leq a, b \leq 1000$ 
*
* 解题思路:
* 这是一道经典的位运算题目，考察对加法运算本质的理解。
* 加法运算可以分解为两个步骤:
* 1. 不考虑进位的加法: 这可以通过异或运算实现  $a \wedge b$ 
* 2. 进位: 这可以通过按位与运算并左移一位实现  $(a \& b) \ll 1$ 
*
* 然后将这两个结果相加，就得到了最终的答案。由于我们不能使用加法运算符，所以需要递归或迭代地进行这个过程，直到进位为 0。
*
* 例如: 计算  $2 + 3$ 
* 2 的二进制: 010
* 3 的二进制: 011
*
* 不考虑进位的加法:  $010 \wedge 011 = 001$ 
* 进位:  $(010 \& 011) \ll 1 = 010 \ll 1 = 100$ 
*
* 继续计算  $001 + 100$ :
* 不考虑进位的加法:  $001 \wedge 100 = 101$ 
* 进位:  $(001 \& 100) \ll 1 = 000 \ll 1 = 000$ 
*
* 进位为 0，计算结束，结果为  $101$  (二进制) = 5 (十进制)
*
* 时间复杂度:  $O(1)$  - 因为整数的位数是固定的 (32 位)
* 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
```

```
/*
 * 计算两个整数的和，不使用运算符 + 和 -
 * 使用位运算实现加法:
 */
```

```

* 1. 不考虑进位的加法: a ^ b
* 2. 进位: (a & b) << 1
* 3. 重复上述过程直到进位为 0
*
* @param a 第一个整数
* @param b 第二个整数
* @return 两整数之和
*/
int getSum(int a, int b) {
    // 当进位为 0 时, 计算结束
    while (b != 0) {
        // 计算进位
        int carry = (a & b) << 1;
        // 计算不考虑进位的加法
        a = a ^ b;
        // 将进位赋值给 b, 继续下一轮计算
        b = carry;
    }
    return a;
}

// 递归实现方式
// int getSum(int a, int b) {
//     // 基础情况: 当进位为 0 时, 返回结果
//     if (b == 0) {
//         return a;
//     }
//     // 递归计算: 不考虑进位的加法 + 进位
//     return getSum(a ^ b, (a & b) << 1);
// }
};

=====

```

文件: Code12\_SumOfTwoIntegers.java

```

=====
package class031;

// 两整数之和 - Sum of Two Integers
// 测试链接 : https://leetcode.cn/problems/sum-of-two-integers/
// 相关题目:
// 1. 位运算技巧大全 - Bit Manipulation Tricks
// 2. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/

```

```
// 3. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
// 4. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/  
// 5. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
```

/\*

题目描述:

给你两个整数  $a$  和  $b$ ，不使用运算符 + 和 -，计算并返回两整数之和。

示例:

输入:  $a = 1, b = 2$

输出: 3

输入:  $a = 2, b = 3$

输出: 5

提示:

$-1000 \leq a, b \leq 1000$

解题思路:

这是一道经典的位运算题目，考察对加法运算本质的理解。

加法运算可以分解为两个步骤:

1. 不考虑进位的加法: 这可以通过异或运算实现  $a \ ^ b$
2. 进位: 这可以通过按位与运算并左移一位实现  $(a \ & b) \ll 1$

然后将这两个结果相加，就得到了最终的答案。由于我们不能使用加法运算符，所以需要递归或迭代地进行这个过程，直到进位为 0。

例如: 计算  $2 + 3$

2 的二进制: 010

3 的二进制: 011

不考虑进位的加法:  $010 \ ^ 011 = 001$

进位:  $(010 \ & 011) \ll 1 = 010 \ll 1 = 100$

继续计算  $001 + 100$ :

不考虑进位的加法:  $001 \ ^ 100 = 101$

进位:  $(001 \ & 100) \ll 1 = 000 \ll 1 = 000$

进位为 0，计算结束，结果为  $101$  (二进制) = 5 (十进制)

时间复杂度:  $O(1)$  - 因为整数的位数是固定的 (32 位)

空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间

## 补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- \*/

```
public class Code12_SumOfTwoIntegers {  
  
    /**  
     * 计算两个整数的和，不使用运算符 + 和 -  
     * 使用位运算实现加法：  
     * 1. 不考虑进位的加法：a ^ b  
     * 2. 进位：(a & b) << 1  
     * 3. 重复上述过程直到进位为 0  
     *  
     * @param a 第一个整数  
     * @param b 第二个整数  
     * @return 两整数之和  
    */  
  
    public static int getSum(int a, int b) {  
        // 当进位为 0 时，计算结束  
        while (b != 0) {  
            // 计算进位  
            int carry = (a & b) << 1;  
            // 计算不考虑进位的加法  
            a = a ^ b;  
            // 将进位赋值给 b，继续下一轮计算  
            b = carry;  
        }  
        return a;  
    }  
  
    // 递归实现方式  
    // public static int getSum(int a, int b) {  
    //     // 基础情况：当进位为 0 时，返回结果  
    //     if (b == 0) {  
    //         return a;  
    //     }  
    //     // 递归计算：不考虑进位的加法 + 进位  
    //     return getSum(a ^ b, (a & b) << 1);  
    // }
```

```
// 测试方法
public static void main(String[] args) {
    System.out.println("Test 1: " + getSum(1, 2)); // 输出: 3
    System.out.println("Test 2: " + getSum(2, 3)); // 输出: 5
    System.out.println("Test 3: " + getSum(-2, 3)); // 输出: 1
    System.out.println("Test 4: " + getSum(-1, 1)); // 输出: 0
}
```

=====

文件: Code12\_SumOfTwoIntegers.py

=====

"""

两整数之和 - Sum of Two Integers

测试链接 : <https://leetcode.cn/problems/sum-of-two-integers/>

相关题目:

1. 位运算技巧大全 - Bit Manipulation Tricks
2. 只出现一次的数字 - Single Number: <https://leetcode.cn/problems/single-number/>
3. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
4. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>
5. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>

题目描述:

给你两个整数  $a$  和  $b$ ，不使用运算符  $+$  和  $-$ ，计算并返回两整数之和。

示例:

输入:  $a = 1, b = 2$

输出: 3

输入:  $a = 2, b = 3$

输出: 5

提示:

$-1000 \leq a, b \leq 1000$

解题思路:

这是一道经典的位运算题目，考察对加法运算本质的理解。

加法运算可以分解为两个步骤:

1. 不考虑进位的加法: 这可以通过异或运算实现  $a \ ^ \ b$
2. 进位: 这可以通过按位与运算并左移一位实现  $(a \ & \ b) \ll 1$

然后将这两个结果相加，就得到了最终的答案。由于我们不能使用加法运算符，所以需要递归或迭代地进行这个过程，直到进位为 0。

例如：计算  $2 + 3$

2 的二进制：010

3 的二进制：011

不考虑进位的加法： $010 \wedge 011 = 001$

进位： $(010 \& 011) \ll 1 = 010 \ll 1 = 100$

继续计算  $001 + 100$ ：

不考虑进位的加法： $001 \wedge 100 = 101$

进位： $(001 \& 100) \ll 1 = 000 \ll 1 = 000$

进位为 0，计算结束，结果为  $101$ （二进制）= 5（十进制）

时间复杂度： $O(1)$  – 因为整数的位数是固定的（32 位）

空间复杂度： $O(1)$  – 只使用了常数级别的额外空间

注意：Python 中的整数是无限精度的，而题目中涉及的是 32 位整数。

在 Python 中处理负数时需要特殊考虑，因为 Python 中的整数是无限精度的，

而题目中的整数是 32 位有符号整数。我们需要模拟 32 位整数的行为。

补充题目：

1. 洛谷 P10118 『STA – R4』 And: <https://www.luogu.com.cn/problem/P10118>
  2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
  3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
  4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
  5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>
- """

```
class Solution:
```

```
    def getSum(self, a: int, b: int) -> int:  
        """
```

计算两个整数的和，不使用运算符  $+$  和  $-$

使用位运算实现加法：

1. 不考虑进位的加法： $a \wedge b$
2. 进位： $(a \& b) \ll 1$
3. 重复上述过程直到进位为 0

```
:param a: 第一个整数
```

```
:param b: 第二个整数
```

```

:return: 两整数之和
"""

# 32 位整数的掩码
mask = 0xFFFFFFFF

# 当进位不为 0 时继续计算
while b != 0:
    # 计算进位，注意要应用掩码以模拟 32 位整数行为
    carry = (a & b) << 1
    # 计算不考虑进位的加法
    a = (a ^ b) & mask
    # 将进位赋值给 b，继续下一轮计算
    b = carry & mask

# 处理负数结果
# 如果 a 大于等于 2^31，说明是负数（在 32 位有符号整数中）
return a if a < 0x80000000 else a | (~mask)

# 递归实现方式
# def getSum(self, a: int, b: int) -> int:
#     # 32 位整数的掩码
#     mask = 0xFFFFFFFF
#
#     # 基础情况：当进位为 0 时，返回结果
#     if b == 0:
#         # 处理负数结果
#         return a if a < 0x80000000 else a | (~mask)
#
#     # 递归计算：不考虑进位的加法 + 进位
#     return self.getSum((a ^ b) & mask, ((a & b) << 1) & mask)

# 测试方法
if __name__ == "__main__":
    solution = Solution()
    print("Test 1:", solution.getSum(1, 2))      # 输出: 3
    print("Test 2:", solution.getSum(2, 3))      # 输出: 5
    print("Test 3:", solution.getSum(-2, 3))     # 输出: 1
    print("Test 4:", solution.getSum(-1, 1))     # 输出: 0
=====
```

```
=====
/**  
 * 数组中两个数的最大异或值 - Maximum XOR of Two Numbers in an Array  
 * 测试链接 : https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/  
 * 相关题目：  
 * 1. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
 * 2. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/  
 * 3. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/  
 * 4. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/  
 * 5. 位运算技巧大全 - Bit Manipulation Tricks  
  
*  
* 题目描述：  
* 给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中  $0 \leq i \leq j < n$ 。  
  
*  
* 示例：  
* 输入: nums = [3, 10, 5, 25, 2, 8]  
* 输出: 28  
* 解释: 最大运算结果是 5 XOR 25 = 28.  
  
*  
* 输入: nums = [0]  
* 输出: 0  
  
*  
* 输入: nums = [2, 4]  
* 输出: 6  
  
*  
* 输入: nums = [8, 10, 2]  
* 输出: 10  
  
*  
* 输入: nums = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]  
* 输出: 127  
  
*  
* 提示：  
*  $1 \leq \text{nums.length} \leq 2 * 10^5$   
*  $0 \leq \text{nums}[i] \leq 2^{31} - 1$   
  
*  
* 进阶：你可以在  $O(n)$  的时间解决这个问题吗？  
  
*  
* 解题思路：  
* 这是一道经典的位运算题目，可以使用贪心算法和字典树(Trie)来解决。  
  
*  
* 方法 1：暴力解法  
* 遍历所有可能的数对，计算它们的异或值，返回最大值。  
* 时间复杂度:  $O(n^2)$ 
```

```

* 空间复杂度: O(1)
*
* 方法 2: 贪心算法 + 字典树 (推荐)
* 核心思想: 要使异或结果最大, 应该从最高位开始, 尽可能使对应位上的数字不同。
* 1. 构建字典树: 将所有数字的二进制表示 (从高位到低位) 插入到字典树中
* 2. 贪心查找: 对于每个数字, 在字典树中寻找与其异或结果最大的数字
*   - 从高位开始遍历, 如果当前位可以与目标数字的对应位不同, 则选择不同的路径
*   - 这样可以保证异或结果在当前位为 1, 从而最大化结果
*
* 例如: 对于数组 [3, 10, 5, 25, 2, 8]
* 3 的二进制: 00011
* 10 的二进制: 01010
* 5 的二进制: 00101
* 25 的二进制: 11001
* 2 的二进制: 00010
* 8 的二进制: 01000
*
* 要使异或结果最大, 应该选择在高位上数字不同的两个数。
* 25(11001) 和 5(00101) 在第 4 位不同, 异或结果在该位为 1, 这样可以获得较大的结果。
*
* 时间复杂度: O(n * 32) = O(n), 其中 n 是数组长度, 32 是整数的位数
* 空间复杂度: O(n * 32) = O(n), 字典树的空间消耗
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
// 为适应编译环境, 避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 IO 函数

// 字典树节点
struct TrieNode {
    TrieNode* children[2]; // 0 和 1 两个子节点

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

//
```

```

class Code13_MaximumXORofTwoNumbersInArray {
private:
    // 字典树根节点
    TrieNode* root;

public:
    Code13_MaximumXORofTwoNumbersInArray() {
        root = new TrieNode();
    }

    /**
     * 找出数组中两个数的最大异或值
     * 使用贪心算法 + 字典树：
     * 1. 将所有数字插入字典树
     * 2. 对于每个数字，在字典树中寻找与其异或结果最大的数字
     *
     * @param nums 输入数组
     * @param numsSize 数组长度
     * @return 最大异或值
     */
    int findMaximumXOR(int* nums, int numsSize) {
        // 特殊情况处理
        if (nums == nullptr || numsSize == 0) {
            return 0;
        }

        // 将所有数字插入字典树
        for (int i = 0; i < numsSize; i++) {
            insert(nums[i]);
        }

        int maxXOR = 0;
        // 对于每个数字，查找与其异或结果最大的数字
        for (int i = 0; i < numsSize; i++) {
            maxXOR = maxXOR > findMaxXOR(nums[i]) ? maxXOR : findMaxXOR(nums[i]);
        }

        return maxXOR;
    }

    /**
     * 将数字插入字典树
     * 从最高位开始处理（第 31 位到第 0 位）
     */
}

```

```

*
* @param num 要插入的数字
*/
void insert(int num) {
    TrieNode* node = root;
    // 从最高位开始处理（第 31 位到第 0 位）
    for (int i = 31; i >= 0; i--) {
        // 提取第 i 位的值（0 或 1）
        int bit = (num >> i) & 1;
        // 如果对应子节点不存在，则创建新节点
        if (node->children[bit] == nullptr) {
            node->children[bit] = new TrieNode();
        }
        // 移动到子节点
        node = node->children[bit];
    }
}

/***
* 查找与给定数字异或结果最大的数字
* 贪心策略：从高位开始，选择与当前位不同的路径（使异或结果在该位为 1）
*
* @param num 给定数字
* @return 最大异或值
*/
int findMaxXOR(int num) {
    TrieNode* node = root;
    int maxXOR = 0;
    // 从最高位开始处理（第 31 位到第 0 位）
    for (int i = 31; i >= 0; i--) {
        // 提取第 i 位的值（0 或 1）
        int bit = (num >> i) & 1;
        // 贪心策略：选择与当前位不同的路径（使异或结果在该位为 1）
        int oppositeBit = bit ^ 1;

        // 如果与当前位不同的路径存在，则选择该路径
        if (node->children[oppositeBit] != nullptr) {
            // 设置异或结果在该位为 1
            maxXOR |= (1 << i);
            node = node->children[oppositeBit];
        } else {
            // 否则选择相同的路径
            node = node->children[bit];
        }
    }
}

```

```

        }
    }

    return maxXOR;
}

};

// 测试方法
int main() {
    Code13_MaximumXORofTwoNumbersInArray solution;

    // 测试用例 1: 正常情况
    int nums1[] = {3, 10, 5, 25, 2, 8};
    int result1 = solution.findMaximumXOR(nums1, 6);
    // 预期结果: 28

    // 测试用例 2: 边界情况
    int nums2[] = {0};
    int result2 = solution.findMaximumXOR(nums2, 1);
    // 预期结果: 0

    // 测试用例 3: 正常情况
    int nums3[] = {2, 4};
    int result3 = solution.findMaximumXOR(nums3, 2);
    // 预期结果: 6

    return 0;
}

```

=====

文件: Code13\_MaximumXORofTwoNumbersInArray.java

=====

```

package class031;

// 数组中两个数的最大异或值 - Maximum XOR of Two Numbers in an Array
// 测试链接 : https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/
// 相关题目:
// 1. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/
// 2. 汉明距离 - Hamming Distance: https://leetcode.cn/problems/hamming-distance/
// 3. 只出现一次的数字 - Single Number: https://leetcode.cn/problems/single-number/
// 4. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
// 5. 位运算技巧大全 - Bit Manipulation Tricks

```

/\*

题目描述:

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

示例:

输入: `nums = [3, 10, 5, 25, 2, 8]`

输出: 28

解释: 最大运算结果是  $5 \text{ XOR } 25 = 28$ .

输入: `nums = [0]`

输出: 0

输入: `nums = [2, 4]`

输出: 6

输入: `nums = [8, 10, 2]`

输出: 10

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

输出: 127

提示:

$1 \leq \text{nums.length} \leq 2 * 10^5$

$0 \leq \text{nums}[i] \leq 2^{31} - 1$

进阶: 你可以在  $O(n)$  的时间解决这个问题吗?

解题思路:

这是一道经典的位运算题目，可以使用贪心算法和字典树(Trie)来解决。

方法 1: 暴力解法

遍历所有可能的数对，计算它们的异或值，返回最大值。

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

方法 2: 贪心算法 + 字典树（推荐）

核心思想: 要使异或结果最大，应该从最高位开始，尽可能使对应位上的数字不同。

1. 构建字典树: 将所有数字的二进制表示（从高位到低位）插入到字典树中

2. 贪心查找: 对于每个数字，在字典树中寻找与其异或结果最大的数字

- 从高位开始遍历，如果当前位可以与目标数字的对应位不同，则选择不同的路径

- 这样可以保证异或结果在当前位为 1，从而最大化结果

例如: 对于数组 `[3, 10, 5, 25, 2, 8]`

```
3 的二进制: 00011
10 的二进制: 01010
5 的二进制: 00101
25 的二进制: 11001
2 的二进制: 00010
8 的二进制: 01000
```

要使异或结果最大，应该选择在高位上数字不同的两个数。

25(11001) 和 5(00101) 在第 4 位不同，异或结果在该位为 1，这样可以获得较大的结果。

时间复杂度:  $O(n * 32) = O(n)$ ，其中 n 是数组长度，32 是整数的位数

空间复杂度:  $O(n * 32) = O(n)$ ，字典树的空间消耗

补充题目：

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

```
*/
```

```
public class Code13_MaximumXORofTwoNumbersInArray {
```

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

```
// 字典树根节点
private TrieNode root = new TrieNode();
```

```
/**
 * 找出数组中两个数的最大异或值
 * 使用贪心算法 + 字典树:
 * 1. 将所有数字插入字典树
 * 2. 对于每个数字，在字典树中寻找与其异或结果最大的数字
 *
 * @param nums 输入数组
 * @return 最大异或值
 */
```

```
public int findMaximumXOR(int[] nums) {
    // 特殊情况处理
    if (nums == null || nums.length == 0) {
        return 0;
```

```

}

// 将所有数字插入字典树
for (int num : nums) {
    insert(num);
}

int maxXOR = 0;
// 对于每个数字，查找与其异或结果最大的数字
for (int num : nums) {
    maxXOR = Math.max(maxXOR, findMaxXOR(num));
}

return maxXOR;
}

/**
 * 将数字插入字典树
 * 从最高位开始处理（第 31 位到第 0 位）
 *
 * @param num 要插入的数字
 */
private void insert(int num) {
    TrieNode node = root;
    // 从最高位开始处理（第 31 位到第 0 位）
    for (int i = 31; i >= 0; i--) {
        // 提取第 i 位的值（0 或 1）
        int bit = (num >> i) & 1;
        // 如果对应子节点不存在，则创建新节点
        if (node.children[bit] == null) {
            node.children[bit] = new TrieNode();
        }
        // 移动到子节点
        node = node.children[bit];
    }
}

/**
 * 查找与给定数字异或结果最大的数字
 * 贪心策略：从高位开始，选择与当前位不同的路径（使异或结果在该位为 1）
 *
 * @param num 给定数字
 * @return 最大异或值

```

```

/*
private int findMaxXOR(int num) {
    TrieNode node = root;
    int maxXOR = 0;
    // 从最高位开始处理（第 31 位到第 0 位）
    for (int i = 31; i >= 0; i--) {
        // 提取第 i 位的值（0 或 1）
        int bit = (num >> i) & 1;
        // 贪心策略：选择与当前位不同的路径（使异或结果在该位为 1）
        int oppositeBit = bit ^ 1;

        // 如果与当前位不同的路径存在，则选择该路径
        if (node.children[oppositeBit] != null) {
            // 设置异或结果在该位为 1
            maxXOR |= (1 << i);
            node = node.children[oppositeBit];
        } else {
            // 否则选择相同的路径
            node = node.children[bit];
        }
    }
    return maxXOR;
}

// 暴力解法（时间复杂度较高，仅用于小规模数据）
// public int findMaximumXOR(int[] nums) {
//     int maxXOR = 0;
//     for (int i = 0; i < nums.length; i++) {
//         for (int j = i + 1; j < nums.length; j++) {
//             maxXOR = Math.max(maxXOR, nums[i] ^ nums[j]);
//         }
//     }
//     return maxXOR;
// }

// 测试方法
public static void main(String[] args) {
    Code13_MaximumXORofTwoNumbersInArray solution = new
    Code13_MaximumXORofTwoNumbersInArray();

    int[] test1 = {3, 10, 5, 25, 2, 8};
    int[] test2 = {0};
    int[] test3 = {2, 4};
}

```

```
int[] test4 = {8, 10, 2};  
int[] test5 = {14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70};  
  
System.out.println("Test 1: " + solution.findMaximumXOR(test1)); // 输出: 28  
System.out.println("Test 2: " + solution.findMaximumXOR(test2)); // 输出: 0  
System.out.println("Test 3: " + solution.findMaximumXOR(test3)); // 输出: 6  
System.out.println("Test 4: " + solution.findMaximumXOR(test4)); // 输出: 10  
System.out.println("Test 5: " + solution.findMaximumXOR(test5)); // 输出: 127  
}  
}
```

}

=====

文件: Code13\_MaximumXORofTwoNumbersInArray.py

=====

"""

数组中两个数的最大异或值 - Maximum XOR of Two Numbers in an Array

测试链接 : <https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/>

相关题目:

1. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
2. 汉明距离 - Hamming Distance: <https://leetcode.cn/problems/hamming-distance/>
3. 只出现一次的数字 - Single Number: <https://leetcode.cn/problems/single-number/>
4. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
5. 位运算技巧大全 - Bit Manipulation Tricks

题目描述:

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

示例:

输入: `nums` = [3, 10, 5, 25, 2, 8]

输出: 28

解释: 最大运算结果是  $5 \text{ XOR } 25 = 28$ .

输入: `nums` = [0]

输出: 0

输入: `nums` = [2, 4]

输出: 6

输入: `nums` = [8, 10, 2]

输出: 10

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

输出: 127

提示:

`1 <= nums.length <= 2 * 10^5`

`0 <= nums[i] <= 2^31 - 1`

进阶: 你可以在  $O(n)$  的时间解决这个问题吗?

解题思路:

这是一道经典的位运算题目, 可以使用贪心算法和字典树(Trie)来解决。

方法 1: 暴力解法

遍历所有可能的数对, 计算它们的异或值, 返回最大值。

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

方法 2: 贪心算法 + 字典树 (推荐)

核心思想: 要使异或结果最大, 应该从最高位开始, 尽可能使对应位上的数字不同。

1. 构建字典树: 将所有数字的二进制表示 (从高位到低位) 插入到字典树中
2. 贪心查找: 对于每个数字, 在字典树中寻找与其异或结果最大的数字
  - 从高位开始遍历, 如果当前位可以与目标数字的对应位不同, 则选择不同的路径
  - 这样可以保证异或结果在当前位为 1, 从而最大化结果

例如: 对于数组 `[3, 10, 5, 25, 2, 8]`

3 的二进制: 00011

10 的二进制: 01010

5 的二进制: 00101

25 的二进制: 11001

2 的二进制: 00010

8 的二进制: 01000

要使异或结果最大, 应该选择在高位上数字不同的两个数。

25(11001) 和 5(00101) 在第 4 位不同, 异或结果在该位为 1, 这样可以获得较大的结果。

时间复杂度:  $O(n * 32) = O(n)$ , 其中  $n$  是数组长度, 32 是整数的位数

空间复杂度:  $O(n * 32) = O(n)$ , 字典树的空间消耗

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>

5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

"""

```
class TrieNode:  
    def __init__(self):  
        self.children = {} # 使用字典存储子节点
```

```
class Solution:  
    def __init__(self):  
        self.root = TrieNode()
```

```
def findMaximumXOR(self, nums: list[int]) -> int:  
    """
```

找出数组中两个数的最大异或值

使用贪心算法 + 字典树:

1. 将所有数字插入字典树
2. 对于每个数字，在字典树中寻找与其异或结果最大的数字

```
:param nums: 输入数组  
:return: 最大异或值  
"""
```

# 特殊情况处理

```
if not nums:  
    return 0
```

# 将所有数字插入字典树

```
for num in nums:  
    self.insert(num)
```

max\_xor = 0

# 对于每个数字，查找与其异或结果最大的数字

```
for num in nums:  
    max_xor = max(max_xor, self.find_max_xor(num))
```

```
return max_xor
```

```
def insert(self, num: int) -> None:  
    """
```

将数字插入字典树

从最高位开始处理（第 31 位到第 0 位）

```

:param num: 要插入的数字
"""

node = self.root
# 从最高位开始处理（第 31 位到第 0 位）
for i in range(31, -1, -1):
    # 提取第 i 位的值（0 或 1）
    bit = (num >> i) & 1
    # 如果对应子节点不存在，则创建新节点
    if bit not in node.children:
        node.children[bit] = TrieNode()
    # 移动到子节点
    node = node.children[bit]

```

```
def find_max_xor(self, num: int) -> int:
```

```
"""


```

查找与给定数字异或结果最大的数字

贪心策略：从高位开始，选择与当前位不同的路径（使异或结果在该位为 1）

```
:param num: 给定数字
```

```
:return: 最大异或值
```

```
"""


```

```
node = self.root
```

```
max_xor = 0
```

```
# 从最高位开始处理（第 31 位到第 0 位）
```

```
for i in range(31, -1, -1):
```

```
    # 提取第 i 位的值（0 或 1）
```

```
    bit = (num >> i) & 1
```

```
    # 贪心策略：选择与当前位不同的路径（使异或结果在该位为 1）
```

```
    opposite_bit = bit ^ 1
```

```
# 如果与当前位不同的路径存在，则选择该路径
```

```
if opposite_bit in node.children:
```

```
    # 设置异或结果在该位为 1
```

```
    max_xor |= (1 << i)
```

```
    node = node.children[opposite_bit]
```

```
else:
```

```
    # 否则选择相同的路径
```

```
    node = node.children[bit]
```

```
return max_xor
```

```
# 测试方法
```

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

```

solution = Solution()

# 测试用例 1: 正常情况
nums1 = [3, 10, 5, 25, 2, 8]
result1 = solution.findMaximumXOR(nums1)
# 预期结果: 28
print(f"测试用例 1 - 输入: {nums1}")
print(f"结果: {result1} (预期: 28)")

# 测试用例 2: 边界情况
nums2 = [0]
result2 = solution.findMaximumXOR(nums2)
# 预期结果: 0
print(f"测试用例 2 - 输入: {nums2}")
print(f"结果: {result2} (预期: 0)")

# 测试用例 3: 正常情况
nums3 = [2, 4]
result3 = solution.findMaximumXOR(nums3)
# 预期结果: 6
print(f"测试用例 3 - 输入: {nums3}")
print(f"结果: {result3} (预期: 6)")

# 测试用例 4: 正常情况
nums4 = [8, 10, 2]
result4 = solution.findMaximumXOR(nums4)
# 预期结果: 10
print(f"测试用例 4 - 输入: {nums4}")
print(f"结果: {result4} (预期: 10)")

# 测试用例 5: 正常情况
nums5 = [14, 70, 53, 83, 49, 91, 36, 80, 92, 51, 66, 70]
result5 = solution.findMaximumXOR(nums5)
# 预期结果: 127
print(f"测试用例 5 - 输入: {nums5}")
print(f"结果: {result5} (预期: 127)")

```

---

文件: Code14\_GrayCode.cpp

---

```

/**
 * 格雷编码 - Gray Code

```

- \* 测试链接 : <https://leetcode.cn/problems/gray-code/>
- \* 相关题目:
  - \* 1. 二进制手表 - Binary Watch: <https://leetcode.cn/problems/binary-watch/>
  - \* 2. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
  - \* 3. 位 1 的个数 - Number of 1 Bits: <https://leetcode.cn/problems/number-of-1-bits/>
  - \* 4. 颠倒二进制位 - Reverse Bits: <https://leetcode.cn/problems/reverse-bits/>
  - \* 5. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>
- \*
- \* 题目描述:
  - \* n 位格雷码序列是一个由  $2^n$  个整数组成的序列，其中：
  - \* - 每个整数都在范围  $[0, 2^n - 1]$  内（含 0 和  $2^n - 1$ ）
  - \* - 第一个整数是 0
  - \* - 一个整数在序列中出现不超过一次
  - \* - 每对相邻整数的二进制表示恰好一位不同
  - \* - 对于序列中的第一个和最后一个整数，其二进制表示也恰好一位不同
- \*
- \* 给定一个整数 n ，返回任一有效的 n 位格雷码序列。
- \*
- \* 示例:
  - \* 输入: n = 2
  - \* 输出: [0, 1, 3, 2]
  - \* 解释:
    - \* [0, 1, 3, 2] 的二进制表示是 [00, 01, 11, 10] 。
    - \* - 00 和 01 有一位不同
    - \* - 01 和 11 有一位不同
    - \* - 11 和 10 有一位不同
    - \* - 10 和 00 有一位不同
- \*
- \* 输入: n = 1
- \* 输出: [0, 1]
- \*
- \* 提示:
  - \*  $1 \leq n \leq 16$
- \*
- \* 解题思路:
  - \* 格雷码 (Gray Code) 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。
  - \*
  - \* 方法 1: 公式法 (最优解)
    - \* 格雷码有一个数学公式：第 i 个格雷码  $G(i) = i \ ^ (i \gg 1)$
    - \* 这个公式的核心思想是:
      - \* - 将数字 i 右移一位，然后与原数字进行异或运算
      - \* - 这样可以保证相邻的两个格雷码只有一位不同
  - \*

\* 例如: n=3 时的格雷码序列

| * i | i>>1 | i^(i>>1) | 二进制表示 |
|-----|------|----------|-------|
| * 0 | 0    | 0        | 000   |
| * 1 | 0    | 1        | 001   |
| * 2 | 1    | 3        | 011   |
| * 3 | 1    | 2        | 010   |
| * 4 | 2    | 6        | 110   |
| * 5 | 2    | 7        | 111   |
| * 6 | 3    | 5        | 101   |
| * 7 | 3    | 4        | 100   |

\*

\* 方法 2: 对称生成法

- \* 1. 从[0]开始
- \* 2. 每次将当前序列反转，并在每个元素前加上 1 (即加上  $2^i$ )，然后追加到原序列后面
- \* 3. 重复 n 次

\*

\* 例如: n=3 时的生成过程

- \* 初始: [0]
- \* 第 1 次: [0] + [0+1] = [0, 1]
- \* 第 2 次: [0, 1] + [1+2, 0+2] = [0, 1, 3, 2]
- \* 第 3 次: [0, 1, 3, 2] + [2+4, 3+4, 1+4, 0+4] = [0, 1, 3, 2, 6, 7, 5, 4]

\*

\* 方法 3: 递归法

- \*  $G(n) = G(n-1) + \text{reverse}(G(n-1)) + 2^{(n-1)}$
- \* 即 n 位格雷码等于(n-1)位格雷码连接上将(n-1)位格雷码反转后每个元素加上  $2^{(n-1)}$

\*

\* 时间复杂度:

- \* 方法 1:  $O(2^n)$  - 需要生成  $2^n$  个数字
- \* 方法 2:  $O(2^n)$  - 需要生成  $2^n$  个数字
- \* 方法 3:  $O(2^n)$  - 递归生成所有数字

\*

\* 空间复杂度:

- \* 方法 1:  $O(1)$  - 不考虑输出数组
- \* 方法 2:  $O(1)$  - 不考虑输出数组
- \* 方法 3:  $O(n)$  - 递归调用栈的深度

\*

\* 补充题目:

- \* 1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
- \* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
- \* 3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
- \* 4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
- \* 5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

\*/

```
// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数  
// 使用基本的 C++ 实现方式和自定义 IO 函数
```

```
class Code14_GrayCode {  
public:  
    /**  
     * 生成 n 位格雷码序列  
     * 使用公式法：第 i 个格雷码  $G(i) = i \wedge (i \gg 1)$   
     *  
     * @param n 位数  
     * @param returnSize 返回数组的大小  
     * @return 格雷码序列  
     */  
    int* grayCode(int n, int* returnSize) {  
        *returnSize = 1 << n;  
        int* result = new int[*returnSize];  
  
        // 方法 1：公式法  
        // 第 i 个格雷码  $G(i) = i \wedge (i \gg 1)$   
        for (int i = 0; i < *returnSize; i++) {  
            result[i] = i ^ (i >> 1);  
        }  
  
        return result;  
    }  
};  
  
// 测试方法  
int main() {  
    Code14_GrayCode solution;  
  
    // 测试用例 1：正常情况  
    int returnSize1;  
    int* result1 = solution.grayCode(1, &returnSize1);  
    // 预期结果：[0, 1]  
  
    // 测试用例 2：正常情况  
    int returnSize2;  
    int* result2 = solution.grayCode(2, &returnSize2);  
    // 预期结果：[0, 1, 3, 2]  
  
    // 测试用例 3：正常情况  
    int returnSize3;
```

```
int* result3 = solution.grayCode(3, &returnSize3);  
// 预期结果: [0, 1, 3, 2, 6, 7, 5, 4]  
  
return 0;  
}
```

---

文件: Code14\_GrayCode.java

---

```
package class031;  
  
// 格雷编码 - Gray Code  
// 测试链接 : https://leetcode.cn/problems/gray-code/  
// 相关题目：  
// 1. 二进制手表 - Binary Watch: https://leetcode.cn/problems/binary-watch/  
// 2. 比特位计数 - Counting Bits: https://leetcode.cn/problems/counting-bits/  
// 3. 位 1 的个数 - Number of 1 Bits: https://leetcode.cn/problems/number-of-1-bits/  
// 4. 颠倒二进制位 - Reverse Bits: https://leetcode.cn/problems/reverse-bits/  
// 5. 2 的幂 - Power of Two: https://leetcode.cn/problems/power-of-two/
```

/\*

题目描述:

n 位格雷码序列是一个由  $2^n$  个整数组成的序列，其中：

- 每个整数都在范围  $[0, 2^n - 1]$  内（含 0 和  $2^n - 1$ ）
- 第一个整数是 0
- 一个整数在序列中出现不超过一次
- 每对相邻整数的二进制表示恰好一位不同
- 对于序列中的第一个和最后一个整数，其二进制表示也恰好一位不同

给定一个整数 n，返回任一有效的 n 位格雷码序列。

示例：

输入：n = 2

输出：[0, 1, 3, 2]

解释：

[0, 1, 3, 2] 的二进制表示是 [00, 01, 11, 10] 。

- 00 和 01 有一位不同
- 01 和 11 有一位不同
- 11 和 10 有一位不同
- 10 和 00 有一位不同

输入：n = 1

输出: [0, 1]

提示:

$1 \leq n \leq 16$

解题思路:

格雷码 (Gray Code) 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。

方法 1: 公式法 (最优解)

格雷码有一个数学公式: 第  $i$  个格雷码  $G(i) = i \hat{\wedge} (i \gg 1)$

这个公式的核心思想是:

- 将数字  $i$  右移一位，然后与原数字进行异或运算
- 这样可以保证相邻的两个格雷码只有一位不同

例如:  $n=3$  时的格雷码序列

| $i$ | $i \gg 1$ | $i \hat{\wedge} (i \gg 1)$ | 二进制表示 |
|-----|-----------|----------------------------|-------|
| 0   | 0         | 0                          | 000   |
| 1   | 0         | 1                          | 001   |
| 2   | 1         | 3                          | 011   |
| 3   | 1         | 2                          | 010   |
| 4   | 2         | 6                          | 110   |
| 5   | 2         | 7                          | 111   |
| 6   | 3         | 5                          | 101   |
| 7   | 3         | 4                          | 100   |

方法 2: 对称生成法

1. 从 [0] 开始
2. 每次将当前序列反转，并在每个元素前加上 1 (即加上  $2^i$ )，然后追加到原序列后面
3. 重复  $n$  次

例如:  $n=3$  时的生成过程

初始: [0]

第 1 次: [0] + [0+1] = [0, 1]

第 2 次: [0, 1] + [1+2, 0+2] = [0, 1, 3, 2]

第 3 次: [0, 1, 3, 2] + [2+4, 3+4, 1+4, 0+4] = [0, 1, 3, 2, 6, 7, 5, 4]

方法 3: 递归法

$G(n) = G(n-1) + \text{reverse}(G(n-1)) + 2^{n-1}$

即  $n$  位格雷码等于  $(n-1)$  位格雷码连接上将  $(n-1)$  位格雷码反转后每个元素加上  $2^{n-1}$

时间复杂度:

方法 1:  $O(2^n)$  - 需要生成  $2^n$  个数字

方法 2:  $O(2^n)$  - 需要生成  $2^n$  个数字

方法 3:  $O(2^n)$  - 递归生成所有数字

空间复杂度:

方法 1:  $O(1)$  - 不考虑输出数组

方法 2:  $O(1)$  - 不考虑输出数组

方法 3:  $O(n)$  - 递归调用栈的深度

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

\*/

```
public class Code14_GrayCode {
```

```
/**
```

```
* 生成 n 位格雷码序列  
* 使用公式法: 第 i 个格雷码  $G(i) = i \ ^ (i \gg 1)$   
*  
* @param n 位数  
* @return 格雷码序列  
*/
```

```
public static java.util.List<Integer> grayCode(int n) {
```

```
    java.util.List<Integer> result = new java.util.ArrayList<>();
```

```
    // 方法 1: 公式法
```

```
    // 第 i 个格雷码  $G(i) = i \ ^ (i \gg 1)$   
    for (int i = 0; i < (1 << n); i++) {  
        result.add(i ^ (i >> 1));  
    }
```

```
    return result;
```

```
}
```

```
// 方法 2: 对称生成法
```

```
// public static java.util.List<Integer> grayCode(int n) {  
//     java.util.List<Integer> result = new java.util.ArrayList<>();  
//     result.add(0);  
//  
//     // 每次迭代生成更高位的格雷码  
//     for (int i = 0; i < n; i++) {  
//         int size = result.size();
```

```

//          // 从后往前遍历，将当前序列反转并加上  $2^i$ 
//          for (int j = size - 1; j >= 0; j--) {
//              result.add(result.get(j) + (1 << i));
//          }
//      }
//
//      return result;
// }

// 方法 3：递归法
// public static java.util.List<Integer> grayCode(int n) {
//     if (n == 0) {
//         java.util.List<Integer> result = new java.util.ArrayList<>();
//         result.add(0);
//         return result;
//     }
//
//     // 递归生成(n-1)位格雷码
//     java.util.List<Integer> prev = grayCode(n - 1);
//     java.util.List<Integer> result = new java.util.ArrayList<>(prev);
//
//     // 将(prev)位格雷码反转后每个元素加上  $2^{(n-1)}$ 
//     int head = 1 << (n - 1);
//     for (int i = prev.size() - 1; i >= 0; i--) {
//         result.add(head + prev.get(i));
//     }
//
//     return result;
// }

// 测试方法
public static void main(String[] args) {
    System.out.println("Test n=1: " + grayCode(1)); // 输出: [0, 1]
    System.out.println("Test n=2: " + grayCode(2)); // 输出: [0, 1, 3, 2]
    System.out.println("Test n=3: " + grayCode(3)); // 输出: [0, 1, 3, 2, 6, 7, 5, 4]
}

}
=====

文件: Code14_GrayCode.py
=====
```

"""

## 格雷编码 - Gray Code

测试链接 : <https://leetcode.cn/problems/gray-code/>

相关题目:

1. 二进制手表 - Binary Watch: <https://leetcode.cn/problems/binary-watch/>
2. 比特位计数 - Counting Bits: <https://leetcode.cn/problems/counting-bits/>
3. 位 1 的个数 - Number of 1 Bits: <https://leetcode.cn/problems/number-of-1-bits/>
4. 颠倒二进制位 - Reverse Bits: <https://leetcode.cn/problems/reverse-bits/>
5. 2 的幂 - Power of Two: <https://leetcode.cn/problems/power-of-two/>

题目描述:

$n$  位格雷码序列是一个由  $2^n$  个整数组成的序列，其中:

- 每个整数都在范围  $[0, 2^n - 1]$  内（含 0 和  $2^n - 1$ ）
- 第一个整数是 0
- 一个整数在序列中出现不超过一次
- 每对相邻整数的二进制表示恰好一位不同
- 对于序列中的第一个和最后一个整数，其二进制表示也恰好一位不同

给定一个整数  $n$ ，返回任一有效的  $n$  位格雷码序列。

示例:

输入:  $n = 2$

输出:  $[0, 1, 3, 2]$

解释:

$[0, 1, 3, 2]$  的二进制表示是  $[00, 01, 11, 10]$ 。

- 00 和 01 有一位不同
- 01 和 11 有一位不同
- 11 和 10 有一位不同
- 10 和 00 有一位不同

输入:  $n = 1$

输出:  $[0, 1]$

提示:

$1 \leq n \leq 16$

解题思路:

格雷码 (Gray Code) 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。

方法 1: 公式法 (最优解)

格雷码有一个数学公式: 第  $i$  个格雷码  $G(i) = i \wedge (i \gg 1)$

这个公式的核心思想是:

- 将数字  $i$  右移一位，然后与原数字进行异或运算

- 这样可以保证相邻的两个格雷码只有一位不同

例如: n=3 时的格雷码序列

| i | $i \gg 1$ | $i \wedge (i \gg 1)$ | 二进制表示 |
|---|-----------|----------------------|-------|
| 0 | 0         | 0                    | 000   |
| 1 | 0         | 1                    | 001   |
| 2 | 1         | 3                    | 011   |
| 3 | 1         | 2                    | 010   |
| 4 | 2         | 6                    | 110   |
| 5 | 2         | 7                    | 111   |
| 6 | 3         | 5                    | 101   |
| 7 | 3         | 4                    | 100   |

方法 2: 对称生成法

1. 从 [0] 开始
2. 每次将当前序列反转，并在每个元素前加上 1（即加上  $2^i$ ），然后追加到原序列后面
3. 重复 n 次

例如: n=3 时的生成过程

初始: [0]

第 1 次: [0] + [0+1] = [0, 1]

第 2 次: [0, 1] + [1+2, 0+2] = [0, 1, 3, 2]

第 3 次: [0, 1, 3, 2] + [2+4, 3+4, 1+4, 0+4] = [0, 1, 3, 2, 6, 7, 5, 4]

方法 3: 递归法

$G(n) = G(n-1) + \text{reverse}(G(n-1)) + 2^{n-1}$

即 n 位格雷码等于 (n-1) 位格雷码连接上将 (n-1) 位格雷码反转后每个元素加上  $2^{n-1}$

时间复杂度:

方法 1:  $O(2^n)$  - 需要生成  $2^n$  个数字

方法 2:  $O(2^n)$  - 需要生成  $2^n$  个数字

方法 3:  $O(2^n)$  - 递归生成所有数字

空间复杂度:

方法 1:  $O(1)$  - 不考虑输出数组

方法 2:  $O(1)$  - 不考虑输出数组

方法 3:  $O(n)$  - 递归调用栈的深度

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>

5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

"""

```
class Solution:
```

```
    def grayCode(self, n: int) -> list[int]:
```

```
        """
```

```
            生成 n 位格雷码序列
```

```
            使用公式法: 第 i 个格雷码  $G(i) = i \ ^ (i \gg 1)$ 
```

```
        :param n: 位数
```

```
        :return: 格雷码序列
```

```
        """
```

```
        result = []
```

```
# 方法 1: 公式法
```

```
# 第 i 个格雷码  $G(i) = i \ ^ (i \gg 1)$ 
```

```
for i in range(1 << n):
```

```
    result.append(i ^ (i >> 1))
```

```
return result
```

```
# 测试方法
```

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

```
    solution = Solution()
```

```
# 测试用例 1: 正常情况
```

```
result1 = solution.grayCode(1)
```

```
# 预期结果: [0, 1]
```

```
print(f"测试用例 1 - 输入: n=1")
```

```
print(f"结果: {result1} (预期: [0, 1])")
```

```
# 测试用例 2: 正常情况
```

```
result2 = solution.grayCode(2)
```

```
# 预期结果: [0, 1, 3, 2]
```

```
print(f"测试用例 2 - 输入: n=2")
```

```
print(f"结果: {result2} (预期: [0, 1, 3, 2])")
```

```
# 测试用例 3: 正常情况
```

```
result3 = solution.grayCode(3)
```

```
# 预期结果: [0, 1, 3, 2, 6, 7, 5, 4]
```

```
print(f"测试用例 3 - 输入: n=3")
```

```
print(f"结果: {result3} (预期: [0, 1, 3, 2, 6, 7, 5, 4])")
```

文件: Code15\_BitwiseANDofNumbersRange.cpp

```
// 数字范围按位与  
// 测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/  
/*
```

题目描述:

给你两个整数 left 和 right , 表示区间  $[left, right]$  , 返回此区间内所有数字按位与的结果 (包含 left 、 right 端点)。

示例:

输入: left = 5, right = 7

输出: 4

输入: left = 0, right = 0

输出: 0

输入: left = 1, right = 2147483647

输出: 0

提示:

$0 \leq left \leq right \leq 2^{31} - 1$

解题思路:

这道题要求计算区间 $[left, right]$ 内所有数字按位与的结果。

方法 1: 暴力解法

遍历区间内所有数字，逐一进行按位与运算。

时间复杂度:  $O(right - left)$

空间复杂度:  $O(1)$

当区间较大时会超时。

方法 2: 找公共前缀 (推荐)

核心思想: 区间内所有数字按位与的结果就是 left 和 right 的二进制表示的公共前缀。

为什么是公共前缀?

考虑区间 $[5, 7]$ :

5: 101

6: 110

7: 111

从右到左逐位分析：

- 最低位：5(1), 6(0), 7(1) → 包含 0 和 1，按位与结果为 0
- 第二位：5(0), 6(1), 7(1) → 包含 0 和 1，按位与结果为 0
- 第三位：5(1), 6(1), 7(1) → 全为 1，按位与结果为 1

所以结果为 100(二进制) = 4(十进制)

更进一步观察发现，结果就是 left 和 right 的公共前缀。

实现方法：

1. 不断右移 left 和 right，直到它们相等（找到公共前缀）
2. 记录右移的位数
3. 将公共前缀左移相应的位数

例如：left=5, right=7

5: 101

7: 111

右移过程：

第 1 次：left=10, right=11, shift=1

第 2 次：left=1, right=1, shift=2

公共前缀为 1，左移 2 位得到 100(二进制)=4(十进制)

方法 3：Brian Kernighan 算法

核心思想：不断移除 right 最右边的 1，直到  $\text{left} \geq \text{right}$ 。

时间复杂度：

方法 1:  $O(\text{right} - \text{left})$

方法 2:  $O(\log n)$

方法 3:  $O(\log n)$

空间复杂度:  $O(1)$

\*/

```
class Solution {
public:
    /**
     * 计算区间[left, right]内所有数字按位与的结果
     * @param left 区间左端点
     * @param right 区间右端点
     * @return 区间内所有数字按位与的结果
    
```

```

/*
int rangeBitwiseAnd(int left, int right) {
    // 方法 2: 找公共前缀
    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 3: Brian Kernighan 算法
// int rangeBitwiseAnd(int left, int right) {
//     // 不断移除 right 最右边的 1, 直到 left >= right
//     while (left < right) {
//         // right & -right 可以提取出 right 最右边的 1
//         // right -= right & -right 相当于移除了最右边的 1
//         right -= right & -right;
//     }
//     return right;
// }

// 方法 1: 暴力解法 (仅适用于小区间)
// int rangeBitwiseAnd(int left, int right) {
//     int result = left;
//     for (int i = left + 1; i <= right; i++) {
//         result &= i;
//         // 优化: 如果结果已经为 0, 可以提前结束
//         if (result == 0) {
//             break;
//         }
//     }
//     return result;
// }
};

=====

```

文件: Code15\_BitwiseANDofNumbersRange.java

```
=====
package class031;
```

```
// 数字范围按位与  
// 测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/  
/*
```

题目描述:

给你两个整数 left 和 right , 表示区间 [left, right] , 返回此区间内所有数字按位与的结果 (包含 left 、right 端点)。

示例:

输入: left = 5, right = 7

输出: 4

输入: left = 0, right = 0

输出: 0

输入: left = 1, right = 2147483647

输出: 0

提示:

$0 \leq \text{left} \leq \text{right} \leq 2^{31} - 1$

解题思路:

这道题要求计算区间 [left, right] 内所有数字按位与的结果。

方法 1: 暴力解法

遍历区间内所有数字，逐一进行按位与运算。

时间复杂度:  $O(\text{right} - \text{left})$

空间复杂度:  $O(1)$

当区间较大时会超时。

方法 2: 找公共前缀 (推荐)

核心思想: 区间内所有数字按位与的结果就是 left 和 right 的二进制表示的公共前缀。

为什么是公共前缀?

考虑区间 [5, 7]:

5: 101

6: 110

7: 111

从右到左逐位分析:

- 最低位: 5(1), 6(0), 7(1) -> 包含 0 和 1, 按位与结果为 0
- 第二位: 5(0), 6(1), 7(1) -> 包含 0 和 1, 按位与结果为 0
- 第三位: 5(1), 6(1), 7(1) -> 全为 1, 按位与结果为 1

所以结果为 100(二进制) = 4(十进制)

更进一步观察发现，结果就是 left 和 right 的公共前缀。

实现方法：

1. 不断右移 left 和 right，直到它们相等（找到公共前缀）
2. 记录右移的位数
3. 将公共前缀左移相应的位数

例如：left=5, right=7

5: 101

7: 111

右移过程：

第 1 次：left=10, right=11, shift=1

第 2 次：left=1, right=1, shift=2

公共前缀为 1，左移 2 位得到 100(二进制)=4(十进制)

方法 3: Brian Kernighan 算法

核心思想：不断移除 right 最右边的 1，直到  $left \geq right$ 。

时间复杂度：

方法 1:  $O(right - left)$

方法 2:  $O(\log n)$

方法 3:  $O(\log n)$

空间复杂度： $O(1)$

\*/

```
public class Code15_BitwiseANDofNumbersRange {  
  
    /**  
     * 计算区间[left, right]内所有数字按位与的结果  
     * @param left 区间左端点  
     * @param right 区间右端点  
     * @return 区间内所有数字按位与的结果  
     */  
    public static int rangeBitwiseAnd(int left, int right) {  
        // 方法 2: 找公共前缀  
        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 3: Brian Kernighan 算法
// public static int rangeBitwiseAnd(int left, int right) {
//     // 不断移除 right 最右边的 1, 直到 left >= right
//     while (left < right) {
//         // right & -right 可以提取出 right 最右边的 1
//         // right -= right & -right 相当于移除了最右边的 1
//         right -= right & -right;
//     }
//     return right;
// }

// 方法 1: 暴力解法 (仅适用于小区间)
// public static int rangeBitwiseAnd(int left, int right) {
//     int result = left;
//     for (int i = left + 1; i <= right; i++) {
//         result &= i;
//         // 优化: 如果结果已经为 0, 可以提前结束
//         if (result == 0) {
//             break;
//         }
//     }
//     return result;
// }

// 测试方法
public static void main(String[] args) {
    System.out.println("Test 1: " + rangeBitwiseAnd(5, 7));           // 输出: 4
    System.out.println("Test 2: " + rangeBitwiseAnd(0, 0));           // 输出: 0
    System.out.println("Test 3: " + rangeBitwiseAnd(1, 2147483647)); // 输出: 0
    System.out.println("Test 4: " + rangeBitwiseAnd(26, 30));        // 输出: 24
}

}
=====
```

文件: Code15\_BitwiseANDofNumbersRange.py

```
=====
# 数字范围按位与
# 测试链接 : https://leetcode.cn/problems/bitwise-and-of-numbers-range/
,,,
```

题目描述:

给你两个整数 left 和 right , 表示区间 [left, right] , 返回此区间内所有数字按位与的结果 (包含 left 、right 端点)。

示例:

输入: left = 5, right = 7

输出: 4

输入: left = 0, right = 0

输出: 0

输入: left = 1, right = 2147483647

输出: 0

提示:

$0 \leq \text{left} \leq \text{right} \leq 2^{31} - 1$

解题思路:

这道题要求计算区间 [left, right] 内所有数字按位与的结果。

方法 1: 暴力解法

遍历区间内所有数字，逐一进行按位与运算。

时间复杂度:  $O(\text{right} - \text{left})$

空间复杂度:  $O(1)$

当区间较大时会超时。

方法 2: 找公共前缀 (推荐)

核心思想: 区间内所有数字按位与的结果就是 left 和 right 的二进制表示的公共前缀。

为什么是公共前缀?

考虑区间 [5, 7]:

5: 101

6: 110

7: 111

从右到左逐位分析:

- 最低位: 5(1), 6(0), 7(1) -> 包含 0 和 1, 按位与结果为 0
- 第二位: 5(0), 6(1), 7(1) -> 包含 0 和 1, 按位与结果为 0

- 第三位: 5(1), 6(1), 7(1)  $\rightarrow$  全为 1, 按位与结果为 1

所以结果为 100(二进制) = 4(十进制)

更进一步观察发现, 结果就是 left 和 right 的公共前缀。

实现方法:

1. 不断右移 left 和 right, 直到它们相等 (找到公共前缀)
2. 记录右移的位数
3. 将公共前缀左移相应的位数

例如: left=5, right=7

5: 101

7: 111

右移过程:

第 1 次: left=10, right=11, shift=1

第 2 次: left=1, right=1, shift=2

公共前缀为 1, 左移 2 位得到 100(二进制)=4(十进制)

方法 3: Brian Kernighan 算法

核心思想: 不断移除 right 最右边的 1, 直到  $left \geq right$ 。

时间复杂度:

方法 1:  $O(right - left)$

方法 2:  $O(\log n)$

方法 3:  $O(\log n)$

空间复杂度:  $O(1)$

, , ,

class Solution:

```
def rangeBitwiseAnd(self, left: int, right: int) -> int:
```

```
    """
```

计算区间 [left, right] 内所有数字按位与的结果

:param left: 区间左端点

:param right: 区间右端点

:return: 区间内所有数字按位与的结果

```
    """
```

# 方法 2: 找公共前缀

shift = 0

# 找到 left 和 right 的公共前缀

```

while left != right:
    left >>= 1
    right >>= 1
    shift += 1
return left << shift

# 方法3: Brian Kernighan 算法
# def rangeBitwiseAnd(self, left: int, right: int) -> int:
#     # 不断移除 right 最右边的1, 直到 left >= right
#     while left < right:
#         # right & -right 可以提取出 right 最右边的1
#         # right -= right & -right 相当于移除了最右边的1
#         right -= right & -right
#     return right

# 方法1: 暴力解法(仅适用于小区间)
# def rangeBitwiseAnd(self, left: int, right: int) -> int:
#     result = left
#     for i in range(left + 1, right + 1):
#         result &= i
#     # 优化: 如果结果已经为0, 可以提前结束
#     if result == 0:
#         break
#     return result

# 测试方法
if __name__ == "__main__":
    solution = Solution()
    print("Test 1:", solution.rangeBitwiseAnd(5, 7))          # 输出: 4
    print("Test 2:", solution.rangeBitwiseAnd(0, 0))          # 输出: 0
    print("Test 3:", solution.rangeBitwiseAnd(1, 2147483647)) # 输出: 0
    print("Test 4:", solution.rangeBitwiseAnd(26, 30))        # 输出: 24

```

=====

文件: Code16\_HammingDistance.cpp

=====

```

#include <iostream>
using namespace std;

// 汉明距离
// 测试链接 : https://leetcode.cn/problems/hamming-distance/
/*

```

## 题目描述:

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。

给你两个整数  $x$  和  $y$ , 计算并返回它们之间的汉明距离。

## 示例:

输入:  $x = 1, y = 4$

输出: 2

## 解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑      ↑

上面的箭头指出了对应二进制位不同的位置。

输入:  $x = 3, y = 1$

输出: 1

## 提示:

$0 \leq x, y \leq 2^{31} - 1$

## 解题思路:

1. 首先对两个数字进行异或运算, 这样不同的位就会被设置为 1

2. 然后统计异或结果中 1 的个数, 这个个数就是汉明距离

计算二进制中 1 的个数可以使用多种方法:

- 逐位检查 (最直接的方法)

- Brian Kernighan 算法: 利用  $n \& (n - 1)$  移除最右边的 1, 直到  $n$  变为 0

- 查表法 (适用于需要频繁计算的场景)

时间复杂度:  $O(1)$  - 因为整数的位数是固定的 (32 位)

空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间

\*/

```
class Solution {
public:
    /**
     * 计算两个整数之间的汉明距离
     * @param x 第一个整数
     * @param y 第二个整数
     * @return 汉明距离
     */
    int hammingDistance(int x, int y) {
        // 对两个数字进行异或运算, 不同的位会被设置为 1
        int xorResult = x ^ y;
```

```
// 方法 1: 逐位检查（最直接的方法）
// int distance = 0;
// for (int i = 0; i < 32; i++) {
//     // 检查当前位是否为 1
//     if ((xorResult & (1 << i)) != 0) {
//         distance++;
//     }
// }
// return distance;

// 方法 2: Brian Kernighan 算法（更高效的方法）
// 每次使用 n & (n - 1) 移除最右边的 1，直到 n 变为 0
int distance = 0;
while (xorResult != 0) {
    distance++;
    // 移除最右边的 1
    xorResult = xorResult & (xorResult - 1);
}
return distance;
}

/**
 * 另一种实现方式：使用 C++ 位运算特性
 * @param x 第一个整数
 * @param y 第二个整数
 * @return 汉明距离
 */
int hammingDistance2(int x, int y) {
    int xorResult = x ^ y;
    int distance = 0;

    // 逐位检查，但使用更紧凑的方式
    while (xorResult) {
        distance += xorResult & 1; // 检查最低位是否为 1
        xorResult >>= 1;           // 右移一位
    }

    return distance;
};

// 测试代码
```

```
int main() {
    Solution solution;

    cout << "Test 1: " << solution.hammingDistance(1, 4) << endl; // 输出: 2
    cout << "Test 2: " << solution.hammingDistance(3, 1) << endl; // 输出: 1
    cout << "Test 3: " << solution.hammingDistance(0, 0) << endl; // 输出: 0
    cout << "Test 4: " << solution.hammingDistance(2147483647, 0) << endl; // 输出: 31

    // 测试方法 2
    cout << "\nUsing method 2:" << endl;
    cout << "Test 1: " << solution.hammingDistance2(1, 4) << endl; // 输出: 2
    cout << "Test 2: " << solution.hammingDistance2(3, 1) << endl; // 输出: 1

    return 0;
}
```

=====

文件: Code16\_HammingDistance.java

=====

```
package class031;

// 汉明距离
// 测试链接 : https://leetcode.cn/problems/hamming-distance/
/*
题目描述:
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。
给你两个整数 x 和 y，计算并返回它们之间的汉明距离。
```

示例:

输入: x = 1, y = 4

输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

上面的箭头指出了对应二进制位不同的位置。

输入: x = 3, y = 1

输出: 1

提示:

$0 \leq x, y \leq 2^{31} - 1$

解题思路：

1. 首先对两个数字进行异或运算，这样不同的位就会被设置为 1
2. 然后统计异或结果中 1 的个数，这个个数就是汉明距离

计算二进制中 1 的个数可以使用多种方法：

- 逐位检查（最直接的方法）
- Brian Kernighan 算法：利用  $n \& (n - 1)$  移除最右边的 1，直到  $n$  变为 0
- 查表法（适用于需要频繁计算的场景）

时间复杂度： $O(1)$  – 因为整数的位数是固定的（32 位）

空间复杂度： $O(1)$  – 只使用了常数级别的额外空间

\*/

```
public class Code16_HammingDistance {  
  
    /**  
     * 计算两个整数之间的汉明距离  
     * @param x 第一个整数  
     * @param y 第二个整数  
     * @return 汉明距离  
     */  
  
    public static int hammingDistance(int x, int y) {  
        // 对两个数字进行异或运算，不同的位会被设置为 1  
        int xor = x ^ y;  
  
        // 方法 1：逐位检查（最直接的方法）  
        // int distance = 0;  
        // for (int i = 0; i < 32; i++) {  
        //     // 检查当前位是否为 1  
        //     if ((xor & (1 << i)) != 0) {  
        //         distance++;  
        //     }  
        // }  
        // return distance;  
  
        // 方法 2：Brian Kernighan 算法（更高效的方法）  
        // 每次使用  $n \& (n - 1)$  移除最右边的 1，直到  $n$  变为 0  
        int distance = 0;  
        while (xor != 0) {  
            distance++;  
            // 移除最右边的 1  
            xor = xor & (xor - 1);  
        }  
    }  
}
```

```

        return distance;
    }

/**
 * 另一种实现方式：使用 Java 内置方法
 * @param x 第一个整数
 * @param y 第二个整数
 * @return 汉明距离
 */
public static int hammingDistance2(int x, int y) {
    // Integer.bitCount() 方法可以直接计算整数二进制表示中 1 的个数
    return Integer.bitCount(x ^ y);
}

// 测试方法
public static void main(String[] args) {
    System.out.println("Test 1: " + hammingDistance(1, 4));    // 输出: 2
    System.out.println("Test 2: " + hammingDistance(3, 1));    // 输出: 1
    System.out.println("Test 3: " + hammingDistance(0, 0));    // 输出: 0
    System.out.println("Test 4: " + hammingDistance(2147483647, 0)); // 输出: 31

    // 测试方法 2
    System.out.println("\nUsing method 2:");
    System.out.println("Test 1: " + hammingDistance2(1, 4));    // 输出: 2
    System.out.println("Test 2: " + hammingDistance2(3, 1));    // 输出: 1
}
}

```

文件: Code16\_HammingDistance.py

```

# 汉明距离
# 测试链接 : https://leetcode.cn/problems/hamming-distance/
,,,
```

题目描述:

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。  
给你两个整数  $x$  和  $y$ ，计算并返回它们之间的汉明距离。

示例:

输入:  $x = 1$ ,  $y = 4$

输出: 2

解释:

1 (0 0 0 1)

4 (0 1 0 0)

↑      ↑

上面的箭头指出了对应二进制位不同的位置。

输入:  $x = 3, y = 1$

输出: 1

提示:

$0 \leq x, y \leq 2^{31} - 1$

解题思路:

1. 首先对两个数字进行异或运算, 这样不同的位就会被设置为 1
2. 然后统计异或结果中 1 的个数, 这个个数就是汉明距离

计算二进制中 1 的个数可以使用多种方法:

- 逐位检查 (最直接的方法)
- Brian Kernighan 算法: 利用  $n \& (n - 1)$  移除最右边的 1, 直到  $n$  变为 0
- Python 内置函数 `bin()` 和 `count()` (最简洁的方法)

时间复杂度:  $O(1)$  – 因为整数的位数是固定的 (32 位)

空间复杂度:  $O(1)$  – 只使用了常数级别的额外空间

,,,

```
class Solution:
```

```
    """
```

汉明距离解决方案类

提供多种计算汉明距离的方法

```
    """
```

```
def hammingDistance(self, x: int, y: int) -> int:
```

```
    """
```

计算两个整数之间的汉明距离

Args:

x: 第一个整数

y: 第二个整数

Returns:

两个整数之间的汉明距离

```
    """
```

```
# 方法 1: 使用 Python 内置函数 (最简洁的方法)
```

```
# 使用 bin() 函数将异或结果转换为二进制字符串, 然后统计'1'的个数
```

```
return bin(x ^ y).count('1')

def hammingDistance2(self, x: int, y: int) -> int:
    """
    计算两个整数之间的汉明距离（使用 Brian Kernighan 算法）

    Args:
        x: 第一个整数
        y: 第二个整数

    Returns:
        两个整数之间的汉明距离
    """

    # 对两个数字进行异或运算
    xor_result = x ^ y
    distance = 0

    # Brian Kernighan 算法：每次移除最右边的 1，直到结果为 0
    while xor_result != 0:
        distance += 1
        # 移除最右边的 1
        xor_result &= xor_result - 1

    return distance

def hammingDistance3(self, x: int, y: int) -> int:
    """
    计算两个整数之间的汉明距离（逐位检查）

    Args:
        x: 第一个整数
        y: 第二个整数

    Returns:
        两个整数之间的汉明距离
    """

    distance = 0
    # 对两个数字进行异或运算
    xor_result = x ^ y

    # 逐位检查，最多检查 32 位（因为输入限制在 32 位整数范围内）
    for _ in range(32):
        # 检查最低位是否为 1
```

```

        distance += xor_result & 1
        # 右移一位
        xor_result >>= 1

    return distance

# 测试代码
if __name__ == "__main__":
    solution = Solution()

# 测试方法 1
print("Test 1: ", solution.hammingDistance(1, 4))    # 输出: 2
print("Test 2: ", solution.hammingDistance(3, 1))    # 输出: 1
print("Test 3: ", solution.hammingDistance(0, 0))    # 输出: 0
print("Test 4: ", solution.hammingDistance(2147483647, 0)) # 输出: 31

# 测试方法 2
print("\nUsing method 2:")
print("Test 1: ", solution.hammingDistance2(1, 4))    # 输出: 2
print("Test 2: ", solution.hammingDistance2(3, 1))    # 输出: 1

# 测试方法 3
print("\nUsing method 3:")
print("Test 1: ", solution.hammingDistance3(1, 4))    # 输出: 2
print("Test 2: ", solution.hammingDistance3(3, 1))    # 输出: 1

```

=====

文件: Code17\_CountingBits.cpp

```

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

// 比特位计数
// 测试链接 : https://leetcode.cn/problems/counting-bits/
/*

```

题目描述:

给你一个整数  $n$ ，对于  $0 \leq i \leq n$  中的每个  $i$ ，计算其二进制表示中 1 的个数，返回一个长度为  $n + 1$  的数组  $ans$  作为答案。

示例:

输入:  $n = 2$

输出: [0, 1, 1]

解释:

0 --> 0

1 --> 1

2 --> 10

输入: n = 5

输出: [0, 1, 1, 2, 1, 2]

解释:

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

提示:

0 <= n <= 10^5

解题思路:

这是一个典型的动态规划问题，可以利用位运算的特性高效解决。

方法 1: 动态规划 + 最低有效位

观察二进制数的规律:

- 对于偶数 i, 其二进制中 1 的个数等于  $i/2$  中 1 的个数
- 对于奇数 i, 其二进制中 1 的个数等于  $i/2$  中 1 的个数加 1

可以统一表示为:  $dp[i] = dp[i \gg 1] + (i \& 1)$

方法 2: 动态规划 + 最高有效位

对于数字 i, 如果我们找到小于等于 i 的最大的 2 的幂 j, 那么  $dp[i] = dp[i - j] + 1$

方法 3: 动态规划 + Brian Kernighan 算法

利用  $n \& (n - 1)$  可以移除最右边的 1, 因此  $dp[i] = dp[i \& (i - 1)] + 1$

时间复杂度: O(n) - 我们只需要遍历一次 0 到 n

空间复杂度: O(n) - 需要一个长度为 n + 1 的数组来存储结果

\*/

```
class Solution {  
public:  
    /**  
     * 计算 0 到 n 之间每个数字的二进制表示中 1 的个数  
     */
```

```

* 使用动态规划 + 最低有效位方法
* @param n 输入整数
* @return 包含每个数字二进制中 1 的个数的数组
*/
vector<int> countBits(int n) {
    vector<int> dp(n + 1, 0);

    // 方法 1: 动态规划 + 最低有效位
    for (int i = 1; i <= n; i++) {
        // 对于数字 i, 右移一位得到 i/2, 然后加上 i 的最低位
        dp[i] = dp[i >> 1] + (i & 1);
    }

    return dp;
}

/***
* 计算 0 到 n 之间每个数字的二进制表示中 1 的个数
* 使用动态规划 + 最高有效位方法
* @param n 输入整数
* @return 包含每个数字二进制中 1 的个数的数组
*/
vector<int> countBits2(int n) {
    vector<int> dp(n + 1, 0);
    int highestBit = 0; // 记录当前的最高有效位

    for (int i = 1; i <= n; i++) {
        // 如果 i 是 2 的幂, 更新 highestBit
        if ((i & (i - 1)) == 0) {
            highestBit = i;
        }
        // 利用最高有效位计算当前数字的 1 的个数
        dp[i] = dp[i - highestBit] + 1;
    }

    return dp;
}

/***
* 计算 0 到 n 之间每个数字的二进制表示中 1 的个数
* 使用动态规划 + Brian Kernighan 算法
* @param n 输入整数
* @return 包含每个数字二进制中 1 的个数的数组
*/

```

```
/*
vector<int> countBits3(int n) {
    vector<int> dp(n + 1, 0);

    for (int i = 1; i <= n; i++) {
        // 利用 n & (n - 1) 移除最右边的 1，然后加 1
        dp[i] = dp[i & (i - 1)] + 1;
    }

    return dp;
}

// 辅助函数：打印数组
void printArray(const vector<int>& arr) {
    cout << "[";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

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

    // 测试方法 1
    vector<int> result1 = solution.countBits(5);
    cout << "Test 1: ";
    printArray(result1); // 输出: [0, 1, 1, 2, 1, 2]

    // 测试方法 2
    vector<int> result2 = solution.countBits2(5);
    cout << "Test 2: ";
    printArray(result2); // 输出: [0, 1, 1, 2, 1, 2]

    // 测试方法 3
    vector<int> result3 = solution.countBits3(5);
    cout << "Test 3: ";
    printArray(result3); // 输出: [0, 1, 1, 2, 1, 2]
```

```
// 测试大数字
int n = 100;
vector<int> largeResult = solution.countBits(n);
cout << "\nTest with n = " << n << endl;
cout << "The count of 1's in " << n << " is: " << largeResult[n] << endl;

return 0;
}
```

---

文件: Code17\_CountingBits.java

```
package class031;

// 比特位计数
// 测试链接 : https://leetcode.cn/problems/counting-bits/
/*
```

题目描述:

给你一个整数  $n$ ，对于  $0 \leq i \leq n$  中的每个  $i$ ，计算其二进制表示中 1 的个数，返回一个长度为  $n + 1$  的数组  $\text{ans}$  作为答案。

示例:

输入:  $n = 2$

输出: [0, 1, 1]

解释:

0 --> 0

1 --> 1

2 --> 10

输入:  $n = 5$

输出: [0, 1, 1, 2, 1, 2]

解释:

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

提示:

$0 \leq n \leq 10^5$

解题思路：

这是一个典型的动态规划问题，可以利用位运算的特性高效解决。

方法 1：动态规划 + 最低有效位

观察二进制数的规律：

- 对于偶数  $i$ , 其二进制中 1 的个数等于  $i/2$  中 1 的个数
- 对于奇数  $i$ , 其二进制中 1 的个数等于  $i/2$  中 1 的个数加 1

可以统一表示为:  $dp[i] = dp[i \gg 1] + (i \& 1)$

方法 2：动态规划 + 最高有效位

对于数字  $i$ , 如果我们找到小于等于  $i$  的最大的 2 的幂  $j$ , 那么  $dp[i] = dp[i - j] + 1$

方法 3：动态规划 + Brian Kernighan 算法

利用  $n \& (n - 1)$  可以移除最右边的 1, 因此  $dp[i] = dp[i \& (i - 1)] + 1$

时间复杂度:  $O(n)$  - 我们只需要遍历一次 0 到  $n$

空间复杂度:  $O(n)$  - 需要一个长度为  $n + 1$  的数组来存储结果

\*/

```
public class Code17_CountingBits {
```

/\*\*

```
* 计算 0 到 n 之间每个数字的二进制表示中 1 的个数
* 使用动态规划 + 最低有效位方法
* @param n 输入整数
* @return 包含每个数字二进制中 1 的个数的数组
*/
```

```
public static int[] countBits(int n) {
    int[] dp = new int[n + 1];

    // 方法 1: 动态规划 + 最低有效位
    // dp[0] 初始化为 0
    for (int i = 1; i <= n; i++) {
        // 对于数字 i, 右移一位得到 i/2, 然后加上 i 的最低位
        dp[i] = dp[i >> 1] + (i & 1);
    }

    return dp;
}
```

/\*\*

```
* 计算 0 到 n 之间每个数字的二进制表示中 1 的个数
```

```

* 使用动态规划 + 最高有效位方法
* @param n 输入整数
* @return 包含每个数字二进制中 1 的个数的数组
*/
public static int[] countBits2(int n) {
    int[] dp = new int[n + 1];
    int highestBit = 0; // 记录当前的最高有效位

    for (int i = 1; i <= n; i++) {
        // 如果 i 是 2 的幂，更新 highestBit
        if ((i & (i - 1)) == 0) {
            highestBit = i;
        }
        // 利用最高有效位计算当前数字的 1 的个数
        dp[i] = dp[i - highestBit] + 1;
    }

    return dp;
}

/***
 * 计算 0 到 n 之间每个数字的二进制表示中 1 的个数
 * 使用动态规划 + Brian Kernighan 算法
 * @param n 输入整数
 * @return 包含每个数字二进制中 1 的个数的数组
*/
public static int[] countBits3(int n) {
    int[] dp = new int[n + 1];

    for (int i = 1; i <= n; i++) {
        // 利用 n & (n - 1) 移除最右边的 1，然后加 1
        dp[i] = dp[i & (i - 1)] + 1;
    }

    return dp;
}

// 测试方法
public static void main(String[] args) {
    // 测试方法 1
    int[] result1 = countBits(5);
    System.out.print("Test 1: [");
    for (int i = 0; i < result1.length; i++) {

```

```

        System.out.print(result1[i]);
        if (i < result1.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]"); // 输出: [0, 1, 1, 2, 1, 2]

    // 测试方法 2
    int[] result2 = countBits2(5);
    System.out.print("Test 2: [");
    for (int i = 0; i < result2.length; i++) {
        System.out.print(result2[i]);
        if (i < result2.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]"); // 输出: [0, 1, 1, 2, 1, 2]

    // 测试方法 3
    int[] result3 = countBits3(5);
    System.out.print("Test 3: [");
    for (int i = 0; i < result3.length; i++) {
        System.out.print(result3[i]);
        if (i < result3.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]"); // 输出: [0, 1, 1, 2, 1, 2]

    // 测试大数字
    int n = 100;
    int[] largeResult = countBits(n);
    System.out.println("\nTest with n = " + n);
    System.out.println("The count of 1's in " + n + " is: " + largeResult[n]);
}
}
=====

文件: Code17_CountingBits.py
=====
# 比特位计数
# 测试链接 : https://leetcode.cn/problems/counting-bits/

```

, , ,

### 题目描述:

给你一个整数  $n$ ，对于  $0 \leq i \leq n$  中的每个  $i$ ，计算其二进制表示中 1 的个数，返回一个长度为  $n + 1$  的数组  $\text{ans}$  作为答案。

### 示例:

输入:  $n = 2$

输出:  $[0, 1, 1]$

### 解释:

$0 \rightarrow 0$

$1 \rightarrow 1$

$2 \rightarrow 10$

输入:  $n = 5$

输出:  $[0, 1, 1, 2, 1, 2]$

### 解释:

$0 \rightarrow 0$

$1 \rightarrow 1$

$2 \rightarrow 10$

$3 \rightarrow 11$

$4 \rightarrow 100$

$5 \rightarrow 101$

### 提示:

$0 \leq n \leq 10^5$

### 解题思路:

这是一个典型的动态规划问题，可以利用位运算的特性高效解决。

#### 方法 1: 动态规划 + 最低有效位

观察二进制数的规律:

- 对于偶数  $i$ ，其二进制中 1 的个数等于  $i/2$  中 1 的个数
- 对于奇数  $i$ ，其二进制中 1 的个数等于  $i/2$  中 1 的个数加 1

可以统一表示为:  $\text{dp}[i] = \text{dp}[i \gg 1] + (i \& 1)$

#### 方法 2: 动态规划 + 最高有效位

对于数字  $i$ ，如果我们找到小于等于  $i$  的最大的  $2$  的幂  $j$ ，那么  $\text{dp}[i] = \text{dp}[i - j] + 1$

#### 方法 3: 动态规划 + Brian Kernighan 算法

利用  $n \& (n - 1)$  可以移除最右边的 1，因此  $\text{dp}[i] = \text{dp}[i \& (i - 1)] + 1$

#### 方法 4: Python 内置函数（简洁但效率可能不如动态规划）

时间复杂度:

- 动态规划方法:  $O(n)$  - 只需要遍历一次 0 到  $n$
- 内置函数方法:  $O(n * k)$ , 其中  $k$  是数字的平均位数

空间复杂度:  $O(n)$  - 需要一个长度为  $n + 1$  的数组来存储结果

,,,

```
class Solution:
```

```
    """
```

比特位计数解决方案类

提供多种计算比特位中 1 的个数的方法

```
    """
```

```
def countBits(self, n: int) -> list[int]:
```

```
    """
```

计算 0 到  $n$  之间每个数字的二进制表示中 1 的个数

使用动态规划 + 最低有效位方法

Args:

n: 输入整数

Returns:

包含每个数字二进制中 1 的个数的数组

```
    """
```

```
dp = [0] * (n + 1)
```

```
# 方法 1: 动态规划 + 最低有效位
```

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

# 对于数字  $i$ , 右移一位得到  $i // 2$ , 然后加上  $i$  的最低位

```
dp[i] = dp[i >> 1] + (i & 1)
```

```
return dp
```

```
def countBits2(self, n: int) -> list[int]:
```

```
    """
```

计算 0 到  $n$  之间每个数字的二进制表示中 1 的个数

使用动态规划 + 最高有效位方法

Args:

n: 输入整数

Returns:

```
    包含每个数字二进制中 1 的个数的数组
    """
dp = [0] * (n + 1)
highest_bit = 0 # 记录当前的最高有效位

for i in range(1, n + 1):
    # 如果 i 是 2 的幂，更新 highest_bit
    if (i & (i - 1)) == 0:
        highest_bit = i
    # 利用最高有效位计算当前数字的 1 的个数
    dp[i] = dp[i - highest_bit] + 1

return dp
```

```
def countBits3(self, n: int) -> list[int]:
    """
    计算 0 到 n 之间每个数字的二进制表示中 1 的个数
    使用动态规划 + Brian Kernighan 算法
```

Args:

n: 输入整数

Returns:

包含每个数字二进制中 1 的个数的数组

"""
dp = [0] \* (n + 1)

for i in range(1, n + 1):
 # 利用 n & (n - 1) 移除最右边的 1，然后加 1
 dp[i] = dp[i & (i - 1)] + 1

return dp

```
def countBits4(self, n: int) -> list[int]:
    """
    计算 0 到 n 之间每个数字的二进制表示中 1 的个数
    使用 Python 内置函数（简洁但效率可能不如动态规划）
```

Args:

n: 输入整数

Returns:

包含每个数字二进制中 1 的个数的数组

```

"""
result = []
for i in range(n + 1):
    # 将数字转换为二进制字符串并统计'1'的个数
    result.append(bin(i).count('1'))

return result

# 测试代码
if __name__ == "__main__":
    solution = Solution()

# 测试方法 1
result1 = solution.countBits(5)
print("Test 1: ", result1)  # 输出: [0, 1, 1, 2, 1, 2]

# 测试方法 2
result2 = solution.countBits2(5)
print("Test 2: ", result2)  # 输出: [0, 1, 1, 2, 1, 2]

# 测试方法 3
result3 = solution.countBits3(5)
print("Test 3: ", result3)  # 输出: [0, 1, 1, 2, 1, 2]

# 测试方法 4
result4 = solution.countBits4(5)
print("Test 4: ", result4)  # 输出: [0, 1, 1, 2, 1, 2]

# 测试大数字
n = 100
large_result = solution.countBits(n)
print(f"\nTest with n = {n}")
print(f"The count of 1's in {n} is: {large_result[n]}")
=====
```

文件: Code18\_ReverseBits.cpp

```

=====

#include <iostream>
using namespace std;

// 反转比特位
// 测试链接 : https://leetcode.cn/problems/reverse-bits/
```

/\*

题目描述:

颠倒给定的 32 位无符号整数的二进制位。

示例:

输入: n = 0000001010010100000111010011100

输出: 964176192 (00111001011110000010100101000000)

解释: 输入的二进制串 0000001010010100000111010011100 表示无符号整数 43261596,

因此返回 964176192, 其二进制表示形式为 00111001011110000010100101000000。

输入: n = 111111111111111111111111111111101

输出: 3221225471 (10111111111111111111111111111111)

解释: 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293,

因此返回 3221225471, 其二进制表示形式为 10111111111111111111111111111111。

提示:

在 C++ 中, 我们可以直接使用 unsigned int 类型来处理无符号整数。

解题思路:

方法 1: 逐位反转

1. 初始化结果 res = 0
2. 对于每一位 (0 到 31), 执行以下操作:
  - a. 将 res 左移 1 位, 为新位腾出位置
  - b. 获取 n 的最低位并加到 res 中
  - c. 将 n 右移 1 位, 处理下一位

方法 2: 位运算分治

可以使用位运算分治法, 通过多次交换相邻的 1 位、2 位、4 位、8 位和 16 位来实现反转。

例如:

- 首先交换每两个相邻位
- 然后交换每两个相邻的 2 位组
- 接着交换每两个相邻的 4 位组
- 然后交换每两个相邻的 8 位组
- 最后交换高 16 位和低 16 位

时间复杂度: O(1) - 因为我们只处理固定的 32 位

空间复杂度: O(1) - 只使用了常数级别的额外空间

\*/

```
class Solution {  
public:  
    /**  
     * 反转 32 位无符号整数的二进制位  
     */
```

```

* 使用逐位反转方法
* @param n 输入的 32 位无符号整数
* @return 反转后的 32 位无符号整数
*/
uint32_t reverseBits(uint32_t n) {
    uint32_t res = 0;
    // 处理每一位，从最低位到最高位
    for (int i = 0; i < 32; i++) {
        // 将结果左移一位，为新位腾出位置
        res <<= 1;
        // 获取 n 的最低位并加到结果中
        res |= (n & 1);
        // 将 n 右移一位，处理下一位
        n >>= 1;
    }
    return res;
}

/***
* 反转 32 位无符号整数的二进制位
* 使用位运算分治方法（更高效）
* @param n 输入的 32 位无符号整数
* @return 反转后的 32 位无符号整数
*/
uint32_t reverseBits2(uint32_t n) {
    // 分治反转：交换相邻的位组
    // 交换每两位
    n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1);
    // 交换每四位中的两位组
    n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2);
    // 交换每八位中的四位组
    n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4);
    // 交换每 16 位中的八位组
    n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8);
    // 交换高 16 位和低 16 位
    n = (n >> 16) | (n << 16);

    return n;
}

/***
* 反转 32 位无符号整数的二进制位
* 使用查表法（适用于需要频繁调用的场景）

```

```

* 预计算每个字节（8位）的反转结果
* @param n 输入的32位无符号整数
* @return 反转后的32位无符号整数
*/
uint32_t reverseBits3(uint32_t n) {
    // 预计算的查找表，存储0-255每个数的8位反转结果
    // 为了简单起见，这里直接计算而不是硬编码
    uint8_t reverseByte[256];
    for (int i = 0; i < 256; i++) {
        reverseByte[i] = reverse8Bits(i);
    }

    // 分别反转四个字节，然后重新组合
    return (static_cast<uint32_t>(reverseByte[n & 0xFF]) << 24) |
        (static_cast<uint32_t>(reverseByte[(n >> 8) & 0xFF]) << 16) |
        (static_cast<uint32_t>(reverseByte[(n >> 16) & 0xFF]) << 8) |
        reverseByte[(n >> 24) & 0xFF];
}

private:
    /**
     * 反转一个字节（8位）的位顺序
     * @param byte 输入的字节（0-255）
     * @return 反转后的字节
     */
    uint8_t reverse8Bits(uint8_t byte) {
        return ((byte * 0x0802LU & 0x22110LU) | (byte * 0x8020LU & 0x88440LU)) * 0x10101LU >> 16;
    }

// 辅助函数：打印二进制表示
void printBinary(uint32_t n) {
    for (int i = 31; i >= 0; i--) {
        cout << ((n >> i) & 1);
        // 每4位添加一个空格，方便阅读
        if (i % 4 == 0 && i > 0) {
            cout << " ";
        }
    }
    cout << endl;
}

// 测试代码

```

```

int main() {
    Solution solution;

    // 示例 1: 43261596 (0000001010010100000111010011100)
    uint32_t n1 = 43261596;
    cout << "Test 1:" << endl;
    cout << "Input: " << n1 << endl;
    cout << "Binary: ";
    printBinary(n1);
    cout << "Output1: " << solution.reverseBits(n1) << endl;
    cout << "Output2: " << solution.reverseBits2(n1) << endl;
    cout << "Output3: " << solution.reverseBits3(n1) << endl;
    cout << "Expected: 964176192" << endl;

    // 示例 2: 4294967293 (11111111111111111111111111101)
    uint32_t n2 = 4294967293;
    cout << "\nTest 2:" << endl;
    cout << "Input: " << n2 << endl;
    cout << "Binary: ";
    printBinary(n2);
    cout << "Output1: " << solution.reverseBits(n2) << endl;
    cout << "Output2: " << solution.reverseBits2(n2) << endl;
    cout << "Output3: " << solution.reverseBits3(n2) << endl;
    cout << "Expected: 3221225471" << endl;

    // 额外测试
    uint32_t n3 = 0; // 全0
    uint32_t n4 = 0xFFFFFFFF; // 全1
    cout << "\nAdditional Tests:" << endl;
    cout << "Input: 0, Output: " << solution.reverseBits(n3) << endl;
    cout << "Input: 0xFFFFFFFF, Output: " << solution.reverseBits(n4) << endl;

    return 0;
}
=====
```

文件: Code18\_ReverseBits.java

```

=====
package class031;

// 反转比特位
// 测试链接 : https://leetcode.cn/problems/reverse-bits/
```

```
/*
```

题目描述:

颠倒给定的 32 位无符号整数的二进制位。

示例:

输入: n = 00000010100101000001111010011100

输出: 964176192 (00111001011110000010100101000000)

解释: 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596,

因此返回 964176192, 其二进制表示形式为 00111001011110000010100101000000。

输入: n = 111111111111111111111111111111101

输出: 3221225471 (10111111111111111111111111111111)

解释: 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293,

因此返回 3221225471, 其二进制表示形式为 10111111111111111111111111111111。

提示:

请注意, 在某些语言(如 Java)中, 没有无符号整数类型。在这种情况下, 输入和输出都将被指定为有符号整数类型, 并且不应影响您的实现, 因为无论整数是有符号的还是无符号的, 其内部的二进制表示形式都是相同的。

在 Java 中, 编译器使用二进制补码记法来表示有符号整数。因此, 在示例 2 中, 输入表示有符号整数 -3, 输出表示有符号整数 -1073741825。

解题思路:

方法 1: 逐位反转

1. 初始化结果 res = 0

2. 对于每一位(0 到 31), 执行以下操作:

- a. 将 res 左移 1 位, 为新位腾出位置
- b. 获取 n 的最低位并加到 res 中
- c. 将 n 右移 1 位, 处理下一位

方法 2: 位运算分治

可以使用位运算分治法, 通过多次交换相邻的 1 位、2 位、4 位、8 位和 16 位来实现反转。

例如:

- 首先交换每两个相邻位
- 然后交换每两个相邻的 2 位组
- 接着交换每两个相邻的 4 位组
- 然后交换每两个相邻的 8 位组
- 最后交换高 16 位和低 16 位

时间复杂度: O(1) - 因为我们只处理固定的 32 位

空间复杂度: O(1) - 只使用了常数级别的额外空间

\*/

```
public class Code18_ReverseBits {
```

```

/**
 * 反转 32 位无符号整数的二进制位
 * 使用逐位反转方法
 * @param n 输入的 32 位无符号整数
 * @return 反转后的 32 位无符号整数
 */
public static int reverseBits(int n) {
    int res = 0;
    // 处理每一位，从最低位到最高位
    for (int i = 0; i < 32; i++) {
        // 将结果左移一位，为新位腾出空间
        res <<= 1;
        // 获取 n 的最低位并加到结果中
        res |= (n & 1);
        // 将 n 右移一位，处理下一位
        n >>>= 1; // 使用无符号右移，确保高位补 0 而不是符号位
    }
    return res;
}

/**
 * 反转 32 位无符号整数的二进制位
 * 使用位运算分治方法（更高效）
 * @param n 输入的 32 位无符号整数
 * @return 反转后的 32 位无符号整数
 */
public static int reverseBits2(int n) {
    // 为了可读性，我们先将 n 转换为 long 类型处理
    long num = n & 0xFFFFFFFFL; // 确保将有符号整数视为无符号整数

    // 分治反转：交换相邻的位组
    // 交换每两位
    num = ((num >>> 1) & 0x55555555L) | ((num & 0x55555555L) << 1);
    // 交换每四位中的两位组
    num = ((num >>> 2) & 0x33333333L) | ((num & 0x33333333L) << 2);
    // 交换每八位中的四位组
    num = ((num >>> 4) & 0x0F0F0F0FL) | ((num & 0x0F0F0F0FL) << 4);
    // 交换每 16 位中的八位组
    num = ((num >>> 8) & 0x00FF00FFL) | ((num & 0x00FF00FFL) << 8);
    // 交换高 16 位和低 16 位
    num = (num >>> 16) | (num << 16);
}

```

```

    // 将结果转换回 int 类型
    return (int) num;
}

/**
 * 反转 32 位无符号整数的二进制位
 * 使用更简洁的位运算分治方法
 * @param n 输入的 32 位无符号整数
 * @return 反转后的 32 位无符号整数
 */
public static int reverseBits3(int n) {
    // 注意: 在 Java 中, int 是有符号的, 所以我们需要使用无符号右移运算符 >>>

    // 交换相邻的位
    n = ((n >>> 1) & 0x55555555) | (((n & 0x55555555) << 1));
    // 交换相邻的 2 位组
    n = ((n >>> 2) & 0x33333333) | (((n & 0x33333333) << 2));
    // 交换相邻的 4 位组
    n = ((n >>> 4) & 0x0F0F0F0F) | (((n & 0x0F0F0F0F) << 4));
    // 交换相邻的 8 位组
    n = ((n >>> 8) & 0x00FF00FF) | (((n & 0x00FF00FF) << 8));
    // 交换高 16 位和低 16 位
    n = (n >>> 16) | (n << 16);

    return n;
}

// 测试方法
public static void main(String[] args) {
    // 示例 1: 43261596 (0000001010010100000111010011100)
    int n1 = 0B0000001010010100000111010011100;
    System.out.println("Test 1:");
    System.out.println("Input: " + n1);
    System.out.println("Output1: " + reverseBits(n1));
    System.out.println("Output2: " + reverseBits2(n1));
    System.out.println("Output3: " + reverseBits3(n1));
    System.out.println("Expected: 964176192");

    // 示例 2: -3 (在二进制中是 1111111111111111111111111101)
    int n2 = -3;
    System.out.println("\nTest 2:");
    System.out.println("Input: " + n2);
    System.out.println("Output1: " + reverseBits(n2));
}

```

```

        System.out.println("Output2: " + reverseBits2(n2));
        System.out.println("Output3: " + reverseBits3(n2));
        System.out.println("Expected: -1073741825");

        // 额外测试
        int n3 = 0; // 全 0
        int n4 = -1; // 全 1
        System.out.println("\nAdditional Tests:");
        System.out.println("Input: 0, Output: " + reverseBits(n3));
        System.out.println("Input: -1, Output: " + reverseBits(n4));
    }
}

```

=====

文件: Code18\_ReverseBits.py

=====

```

# 反转比特位
# 测试链接 : https://leetcode.cn/problems/reverse-bits/
,,,

```

题目描述:

颠倒给定的 32 位无符号整数的二进制位。

示例:

输入: n = 0000001010010100000111010011100

输出: 964176192 (00111001011110000010100101000000)

解释: 输入的二进制串 0000001010010100000111010011100 表示无符号整数 43261596,

因此返回 964176192, 其二进制表示形式为 00111001011110000010100101000000。

输入: n = 11111111111111111111111111111101

输出: 3221225471 (10111111111111111111111111111111)

解释: 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293,

因此返回 3221225471, 其二进制表示形式为 10111111111111111111111111111111。

提示:

在 Python 中, 整数没有固定的位数限制, 但我们可以通过位运算模拟 32 位无符号整数的行为。

解题思路:

方法 1: 逐位反转

1. 初始化结果 res = 0
2. 对于每一位 (0 到 31), 执行以下操作:
  - a. 将 res 左移 1 位, 为新位腾出位置
  - b. 获取 n 的最低位并加到 res 中

c. 将 n 右移 1 位，处理下一位

### 方法 2：位运算分治

可以使用位运算分治法，通过多次交换相邻的 1 位、2 位、4 位、8 位和 16 位来实现反转。

### 方法 3：字符串处理（Python 特色方法）

将整数转换为二进制字符串，反转后再转回整数。

时间复杂度：O(1) – 因为我们只处理固定的 32 位

空间复杂度：O(1) – 只使用了常数级别的额外空间

,,,

```
class Solution:
```

```
    """
```

反转比特位解决方案类

提供多种反转 32 位无符号整数的方法

```
    """
```

```
def reverseBits(self, n: int) -> int:
```

```
    """
```

反转 32 位无符号整数的二进制位

使用逐位反转方法

Args:

n: 输入的 32 位无符号整数

Returns:

反转后的 32 位无符号整数

```
    """
```

```
res = 0
```

# 处理每一位，从最低位到最高位

```
for i in range(32):
```

# 将结果左移一位，为新位腾出位置

```
    res <= 1
```

# 获取 n 的最低位并加到结果中

```
    res |= (n & 1)
```

# 将 n 右移一位，处理下一位

```
    n >>= 1
```

```
return res
```

```
def reverseBits2(self, n: int) -> int:
```

```
    """
```

反转 32 位无符号整数的二进制位

使用位运算分治方法（更高效）

Args:

n: 输入的 32 位无符号整数

Returns:

反转后的 32 位无符号整数

"""

# 确保 n 是 32 位无符号整数

n &= 0xFFFFFFFF

# 分治反转：交换相邻的位组

# 交换每两位

n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1)

# 交换每四位中的两位组

n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2)

# 交换每八位中的四位组

n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4)

# 交换每 16 位中的八位组

n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8)

# 交换高 16 位和低 16 位

n = (n >> 16) | (n << 16)

# 确保返回的是 32 位无符号整数

return n & 0xFFFFFFFF

def reverseBits3(self, n: int) -> int:

"""

反转 32 位无符号整数的二进制位

使用 Python 字符串处理方法（简洁但可能效率较低）

Args:

n: 输入的 32 位无符号整数

Returns:

反转后的 32 位无符号整数

"""

# 将整数转换为 32 位二进制字符串，填充前导零

binary = format(n, '032b')

# 反转字符串

reversed\_binary = binary[::-1]

# 将反转后的二进制字符串转换回整数

```
    return int(reversed_binary, 2)

# 辅助函数: 打印二进制表示

def print_binary(n: int):
    """
    打印 32 位无符号整数的二进制表示
    """
    binary = format(n & 0xFFFFFFFF, '032b')
    # 每 4 位添加一个空格, 方便阅读
    formatted = ' '.join([binary[i:i+4] for i in range(0, 32, 4)])[::-1].replace(' ', ', ', 1)[::-1]
    print(formatted)

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    # 示例 1: 43261596 (00000010100101000001111010011100)
    n1 = 43261596
    print("Test 1:")
    print(f"Input: {n1}")
    print("Binary: ")
    print_binary(n1)
    print(f"Output1: {solution.reverseBits(n1)}")
    print(f"Output2: {solution.reverseBits2(n1)}")
    print(f"Output3: {solution.reverseBits3(n1)}")
    print("Expected: 964176192")

    # 示例 2: 4294967293 (1111111111111111111111111101)
    n2 = 4294967293
    print("\nTest 2:")
    print(f"Input: {n2}")
    print("Binary: ")
    print_binary(n2)
    print(f"Output1: {solution.reverseBits(n2)}")
    print(f"Output2: {solution.reverseBits2(n2)}")
    print(f"Output3: {solution.reverseBits3(n2)}")
    print("Expected: 3221225471")

    # 额外测试
    n3 = 0 # 全 0
    n4 = 0xFFFFFFFF # 全 1
```

```
print("\nAdditional Tests:")
print(f"Input: 0, Output: {solution.reverseBits(n3)}")
print(f"Input: 0xFFFFFFFF, Output: {solution.reverseBits(n4)}")
```

---

文件: Code19\_SingleNumberIII.cpp

---

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

// 只出现一次的数字 III
// 测试链接 : https://leetcode.cn/problems/single-number-iii/
/*
```

题目描述:

给你一个整数数组 `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。你可以按任意顺序返回答案。

示例:

输入: `nums` = [1, 2, 1, 3, 2, 5]

输出: [3, 5] 或 [5, 3]

输入: `nums` = [-1, 0]

输出: [-1, 0]

输入: `nums` = [0, 1]

输出: [1, 0]

提示:

$2 \leq \text{nums.length} \leq 3 * 10^4$

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

除两个只出现一次的整数外, `nums` 中的其他数字都出现两次

解题思路:

这道题是对只出现一次的数字 (Single Number) 的延伸, 现在有两个数字只出现一次, 其余数字都出现两次。

关键点是如何将这两个只出现一次的数字分开处理。

- 首先, 对所有数字进行异或运算, 得到的结果是两个只出现一次的数字的异或结果 (因为相同数字异或为 0, 0 与任何数异或为该数)。
- 在这个异或结果中, 找到任意一个为 1 的位。这个位为 1 表示两个只出现一次的数字在这一位上的值不同。
- 根据这个位是否为 1, 将原数组分成两组。这样, 两个只出现一次的数字会被分到不同的组中。

4. 对每个组内的数字进行异或运算，最终得到两个只出现一次的数字。

时间复杂度：O(n) - 我们需要遍历数组两次

空间复杂度：O(1) - 只使用了常数级别的额外空间

\*/

```
class Solution {
public:
    /**
     * 找出数组中只出现一次的两个元素
     * @param nums 输入的整数数组
     * @return 只出现一次的两个元素组成的数组
     */
    vector<int> singleNumber(vector<int>& nums) {
        // 步骤 1: 对所有数字进行异或运算，得到两个只出现一次的数字的异或结果
        long long xorResult = 0; // 使用 long long 避免溢出
        for (int num : nums) {
            xorResult ^= num;
        }

        // 步骤 2: 找到 xorResult 中任意一个为 1 的位
        // 这里我们找到最右边的 1
        // 例如，对于 xorResult = 01010, rightmostSetBit = 00010
        long long rightmostSetBit = xorResult & (-xorResult);

        // 步骤 3: 根据这个位将数组分成两组，并分别对两组进行异或运算
        int a = 0, b = 0;
        for (int num : nums) {
            if ((num & rightmostSetBit) == 0) {
                // 该位为 0 的组
                a ^= num;
            } else {
                // 该位为 1 的组
                b ^= num;
            }
        }

        // 返回两个只出现一次的数字
        return {a, b};
    }

    /**
     * 找出数组中只出现一次的两个元素（另一种实现方式）
     */
```

```
* @param nums 输入的整数数组
* @return 只出现一次的两个元素组成的数组
*/
vector<int> singleNumber2(vector<int>& nums) {
    long long xorResult = 0;
    for (int num : nums) {
        xorResult ^= num;
    }

    // 找到 xorResult 中任意一个为 1 的位
    // 这里使用不同的方法来找最右边的 1
    long long rightmostSetBit = 1;
    while ((xorResult & rightmostSetBit) == 0) {
        rightmostSetBit <<= 1;
    }

    int a = 0, b = 0;
    for (int num : nums) {
        if ((num & rightmostSetBit) == 0) {
            a ^= num;
        } else {
            b ^= num;
        }
    }

    return {a, b};
}

};

// 辅助函数：打印数组
void printArray(const vector<int>& arr) {
    cout << "[";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}

// 测试代码
int main() {
```

```

Solution solution;

// 测试用例 1
vector<int> nums1 = {1, 2, 1, 3, 2, 5};
vector<int> result1 = solution.singleNumber(nums1);
cout << "Test 1: ";
printArray(result1); // 输出: [3, 5] 或 [5, 3]

// 测试用例 2
vector<int> nums2 = {-1, 0};
vector<int> result2 = solution.singleNumber(nums2);
cout << "Test 2: ";
printArray(result2); // 输出: [-1, 0] 或 [0, -1]

// 测试用例 3
vector<int> nums3 = {0, 1};
vector<int> result3 = solution.singleNumber(nums3);
cout << "Test 3: ";
printArray(result3); // 输出: [0, 1] 或 [1, 0]

// 使用第二种方法测试
cout << "\nUsing alternative method:" << endl;
vector<int> result1_alt = solution.singleNumber2(nums1);
cout << "Test 1 (alt): ";
printArray(result1_alt);

return 0;
}

```

=====

文件: Code19\_SingleNumberIII.java

=====

```

package class031;

// 只出现一次的数字 III
// 测试链接 : https://leetcode.cn/problems/single-number-iii/
/*
题目描述:
给你一个整数数组 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。你可以按任意顺序返回答案。

```

示例:

输入: nums = [1, 2, 1, 3, 2, 5]

输出: [3, 5] 或 [5, 3]

输入: nums = [-1, 0]

输出: [-1, 0]

输入: nums = [0, 1]

输出: [1, 0]

提示:

$2 \leq \text{nums.length} \leq 3 * 10^4$

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

除两个只出现一次的整数外，nums 中的其他数字都出现两次

解题思路:

这道题是对只出现一次的数字 (Single Number) 的延伸，现在有两个数字只出现一次，其余数字都出现两次。

关键点是如何将这两个只出现一次的数字分开处理。

- 首先，对所有数字进行异或运算，得到的结果是两个只出现一次的数字的异或结果（因为相同数字异或为0，0与任何数异或为该数）。
- 在这个异或结果中，找到任意一个为1的位。这个位为1表示两个只出现一次的数字在这一位上的值不同。
- 根据这个位是否为1，将原数组分成两组。这样，两个只出现一次的数字会被分到不同的组中。
- 对每个组内的数字进行异或运算，最终得到两个只出现一次的数字。

时间复杂度: O(n) – 我们需要遍历数组两次

空间复杂度: O(1) – 只使用了常数级别的额外空间

\*/

```
public class Code19_SingleNumberIII {

    /**
     * 找出数组中只出现一次的两个元素
     * @param nums 输入的整数数组
     * @return 只出现一次的两个元素组成的数组
     */
    public static int[] singleNumber(int[] nums) {
        // 步骤 1: 对所有数字进行异或运算，得到两个只出现一次的数字的异或结果
        int xorResult = 0;
        for (int num : nums) {
            xorResult ^= num;
        }

        // 步骤 2: 找到 xorResult 中任意一个为 1 的位
    }
}
```

```

// 这里我们找到最右边的 1
// 例如, 对于 xorResult = 01010, rightmostSetBit = 00010
int rightmostSetBit = xorResult & (-xorResult);

// 步骤 3: 根据这个位将数组分成两组, 并分别对两组进行异或运算
int a = 0, b = 0;
for (int num : nums) {
    if ((num & rightmostSetBit) == 0) {
        // 该位为 0 的组
        a ^= num;
    } else {
        // 该位为 1 的组
        b ^= num;
    }
}

// 返回两个只出现一次的数字
return new int[] {a, b};
}

/**
 * 找出数组中只出现一次的两个元素 (另一种实现方式)
 * @param nums 输入的整数数组
 * @return 只出现一次的两个元素组成的数组
 */
public static int[] singleNumber2(int[] nums) {
    int xorResult = 0;
    for (int num : nums) {
        xorResult ^= num;
    }

    // 找到 xorResult 中任意一个为 1 的位
    // 这里使用不同的方法来找最右边的 1
    int rightmostSetBit = 1;
    while ((xorResult & rightmostSetBit) == 0) {
        rightmostSetBit <<= 1;
    }

    int a = 0, b = 0;
    for (int num : nums) {
        if ((num & rightmostSetBit) == 0) {
            a ^= num;
        } else {

```

```
b ^= num;
}

return new int[]{a, b};
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 2, 1, 3, 2, 5};
    int[] result1 = singleNumber(nums1);
    System.out.print("Test 1: [");
    System.out.print(result1[0] + ", " + result1[1]);
    System.out.println("]"); // 输出: [3, 5] 或 [5, 3]

    // 测试用例 2
    int[] nums2 = {-1, 0};
    int[] result2 = singleNumber(nums2);
    System.out.print("Test 2: [");
    System.out.print(result2[0] + ", " + result2[1]);
    System.out.println("]"); // 输出: [-1, 0] 或 [0, -1]

    // 测试用例 3
    int[] nums3 = {0, 1};
    int[] result3 = singleNumber(nums3);
    System.out.print("Test 3: [");
    System.out.print(result3[0] + ", " + result3[1]);
    System.out.println("]"); // 输出: [0, 1] 或 [1, 0]

    // 使用第二种方法测试
    System.out.println("\nUsing alternative method:");
    int[] result1_alt = singleNumber2(nums1);
    System.out.print("Test 1 (alt): [");
    System.out.print(result1_alt[0] + ", " + result1_alt[1]);
    System.out.println("]");
}
```

=====

文件: Code19\_SingleNumberIII.py

=====

```
# 只出现一次的数字 III  
# 测试链接 : https://leetcode.cn/problems/single-number-iii/  
,,,
```

题目描述:

给你一个整数数组 `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。你可以按任意顺序返回答案。

示例:

输入: `nums` = [1, 2, 1, 3, 2, 5]

输出: [3, 5] 或 [5, 3]

输入: `nums` = [-1, 0]

输出: [-1, 0]

输入: `nums` = [0, 1]

输出: [1, 0]

提示:

$2 \leq \text{nums.length} \leq 3 * 10^4$

$-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

除两个只出现一次的整数外, `nums` 中的其他数字都出现两次

解题思路:

这道题是对只出现一次的数字 (Single Number) 的延伸, 现在有两个数字只出现一次, 其余数字都出现两次。

关键点是如何将这两个只出现一次的数字分开处理。

- 首先, 对所有数字进行异或运算, 得到的结果是两个只出现一次的数字的异或结果 (因为相同数字异或为 0, 0 与任何数异或为该数)。
- 在这个异或结果中, 找到任意一个为 1 的位。这个位为 1 表示两个只出现一次的数字在这一位上的值不同。
- 根据这个位是否为 1, 将原数组分成两组。这样, 两个只出现一次的数字会被分到不同的组中。
- 对每个组内的数字进行异或运算, 最终得到两个只出现一次的数字。

时间复杂度:  $O(n)$  – 我们需要遍历数组两次

空间复杂度:  $O(1)$  – 只使用了常数级别的额外空间

,,

```
class Solution:
```

```
    """
```

只出现一次的数字 III 解决方案类

使用位运算找到数组中只出现一次的两个元素

```
    """
```

```
def singleNumber(self, nums: list[int]) -> list[int]:  
    """  
    找出数组中只出现一次的两个元素  
  
    Args:  
        nums: 输入的整数数组  
  
    Returns:  
        只出现一次的两个元素组成的数组  
    """  
  
    # 步骤 1: 对所有数字进行异或运算, 得到两个只出现一次的数字的异或结果  
    xor_result = 0  
    for num in nums:  
        xor_result ^= num  
  
    # 步骤 2: 找到 xor_result 中任意一个为 1 的位  
    # 这里我们找到最右边的 1  
    # 例如, 对于 xor_result = 01010, rightmost_set_bit = 00010  
    # 在 Python 中, 我们需要处理负数的情况  
    # 对于负数, Python 使用无限精度的二进制补码表示, 所以我们需要与 mask 结合  
    rightmost_set_bit = 1  
    while (xor_result & rightmost_set_bit) == 0:  
        rightmost_set_bit <= 1  
  
    # 步骤 3: 根据这个位将数组分成两组, 并分别对两组进行异或运算  
    a = 0 # 该位为 0 的组异或结果  
    b = 0 # 该位为 1 的组异或结果  
  
    for num in nums:  
        if (num & rightmost_set_bit) == 0:  
            # 该位为 0 的组  
            a ^= num  
        else:  
            # 该位为 1 的组  
            b ^= num  
  
    # 返回两个只出现一次的数字  
    return [a, b]
```

```
def singleNumber2(self, nums: list[int]) -> list[int]:  
    """  
    找出数组中只出现一次的两个元素 (优化版本)  
    在 Python 中, 使用位掩码处理负数情况
```

Args:

nums: 输入的整数数组

Returns:

只出现一次的两个元素组成的数组

"""

# 步骤 1: 对所有数字进行异或运算

xor\_result = 0

for num in nums:

    xor\_result ^= num

# 步骤 2: 找到最右边的 1

# 在 Python 中, 为了处理负数, 我们可以使用 mask

mask = 1

while True:

    if xor\_result & mask:

        break

    mask <= 1

# 步骤 3: 分组异或

a, b = 0, 0

for num in nums:

    if num & mask:

        a ^= num

    else:

        b ^= num

return [a, b]

def singleNumber3(self, nums: list[int]) -> list[int]:

"""

找出数组中只出现一次的两个元素 (使用集合方法, 空间复杂度 O(n))

虽然空间复杂度不如位运算方法好, 但实现简单易懂

Args:

nums: 输入的整数数组

Returns:

只出现一次的两个元素组成的数组

"""

seen = set()

for num in nums:

```

    if num in seen:
        seen.remove(num)
    else:
        seen.add(num)

    return list(seen)

# 测试代码
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1
nums1 = [1, 2, 1, 3, 2, 5]
result1 = solution.singleNumber(nums1)
print("Test 1: ", result1) # 输出: [3, 5] 或 [5, 3]

# 测试用例 2
nums2 = [-1, 0]
result2 = solution.singleNumber(nums2)
print("Test 2: ", result2) # 输出: [-1, 0] 或 [0, -1]

# 测试用例 3
nums3 = [0, 1]
result3 = solution.singleNumber(nums3)
print("Test 3: ", result3) # 输出: [0, 1] 或 [1, 0]

# 使用第二种方法测试
print("\nUsing alternative method:")
result1_alt = solution.singleNumber2(nums1)
print("Test 1 (alt): ", result1_alt)

# 使用第三种方法测试（集合方法）
print("\nUsing set method:")
result1_set = solution.singleNumber3(nums1)
print("Test 1 (set): ", result1_set)

```

=====

文件: Code20\_Subsets.cpp

=====

```

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

```

```

/**
 * 子集
 * 测试链接: https://leetcode.cn/problems/subsets/
 *
 * 题目描述:
 * 给你一个整数数组 nums，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。你可以按任意顺序返回解集。
 *
 * 解题思路:
 * 使用位运算来生成所有子集。对于长度为 n 的数组，共有  $2^n$  个子集。
 * 每个子集可以用一个 n 位的二进制数表示，其中第 i 位为 1 表示包含 nums[i]，为 0 表示不包含。
 *
 * 时间复杂度: O(n * 2^n) - 需要生成  $2^n$  个子集，每个子集需要 O(n) 时间构建
 * 空间复杂度: O(n) - 不考虑输出空间
 */
class Solution {
public:
    /**
     * 使用位运算生成所有子集
     * @param nums 输入数组
     * @return 所有子集的向量
     */
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> result;
        int n = nums.size();
        int totalSubsets = 1 << n; //  $2^n$  个子集

        // 遍历所有可能的二进制掩码
        for (int mask = 0; mask < totalSubsets; mask++) {
            vector<int> subset;

            // 检查每个位，如果为 1 则添加对应元素
            for (int i = 0; i < n; i++) {
                if (mask & (1 << i)) {
                    subset.push_back(nums[i]);
                }
            }
            result.push_back(subset);
        }

        return result;
    }
}

```

```

/**
 * 回溯法实现（备选方案）
 * @param nums 输入数组
 * @return 所有子集的向量
 */
vector<vector<int>> subsetsBacktrack(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    backtrack(nums, 0, current, result);
    return result;
}

private:
    void backtrack(vector<int>& nums, int start, vector<int>& current, vector<vector<int>& result) {
        result.push_back(current);

        for (int i = start; i < nums.size(); i++) {
            current.push_back(nums[i]);
            backtrack(nums, i + 1, current, result);
            current.pop_back();
        }
    }
};

// 测试函数
int main() {
    Solution solution;

    vector<int> test1 = {1, 2, 3};
    vector<int> test2 = {0};
    vector<int> test3 = {1, 2};

    auto result1 = solution.subsets(test1);
    auto result2 = solution.subsets(test2);
    auto result3 = solution.subsets(test3);

    cout << "Test 1 size: " << result1.size() << endl;
    cout << "Test 2 size: " << result2.size() << endl;
    cout << "Test 3 size: " << result3.size() << endl;

    return 0;
}

```

```
=====
文件: Code20_Subsets.java
=====
```

```
package class031;

import java.util.ArrayList;
import java.util.List;

/**
 * 子集
 * 测试链接: https://leetcode.cn/problems/subsets/
 *
 * 题目描述:
 * 给你一个整数数组 nums，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。你可以按任意顺序返回解集。
 *
 * 示例:
 * 输入: nums = [1, 2, 3]
 * 输出: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
 *
 * 输入: nums = [0]
 * 输出: [[], [0]]
 *
 * 提示:
 * 1 <= nums.length <= 10
 * -10 <= nums[i] <= 10
 * nums 中的所有元素互不相同
 *
 * 解题思路:
 * 使用位运算来生成所有子集。对于长度为 n 的数组，共有  $2^n$  个子集。
 * 每个子集可以用一个 n 位的二进制数表示，其中第 i 位为 1 表示包含 nums[i]，为 0 表示不包含。
 *
 * 时间复杂度: O(n * 2^n) - 需要生成  $2^n$  个子集，每个子集需要 O(n) 时间构建
 * 空间复杂度: O(n) - 不考虑输出空间，递归深度为 n
 */
public class Code20_Subsets {

    /**
     * 使用位运算生成所有子集
     * @param nums 输入数组
     * @return 所有子集的列表
}
```

```

*/
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    int n = nums.length;
    int totalSubsets = 1 << n; // 2^n 个子集

    // 遍历所有可能的二进制掩码
    for (int mask = 0; mask < totalSubsets; mask++) {
        List<Integer> subset = new ArrayList<>();

        // 检查每个位，如果为 1 则添加对应元素
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) != 0) {
                subset.add(nums[i]);
            }
        }
        result.add(subset);
    }

    return result;
}

/***
 * 回溯法实现（备选方案）
 * @param nums 输入数组
 * @return 所有子集的列表
 */
public List<List<Integer>> subsetsBacktrack(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(nums, 0, new ArrayList<>(), result);
    return result;
}

private void backtrack(int[] nums, int start, List<Integer> current, List<List<Integer>>
result) {
    result.add(new ArrayList<>(current));

    for (int i = start; i < nums.length; i++) {
        current.add(nums[i]);
        backtrack(nums, i + 1, current, result);
        current.remove(current.size() - 1);
    }
}

```

```
// 测试方法
public static void main(String[] args) {
    Code20_Subsets solution = new Code20_Subsets();

    int[] test1 = {1, 2, 3};
    int[] test2 = {0};
    int[] test3 = {1, 2};

    System.out.println("Test 1: " + solution.subsets(test1));
    System.out.println("Test 2: " + solution.subsets(test2));
    System.out.println("Test 3: " + solution.subsets(test3));
}
```

=====

文件: Code20\_Subsets.py

```
=====
from typing import List

class Solution:
    """
子集
测试链接: https://leetcode.cn/problems/subsets/
```

题目描述:

给你一个整数数组 `nums`，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。  
解集不能包含重复的子集。你可以按任意顺序返回解集。

解题思路:

使用位运算来生成所有子集。对于长度为  $n$  的数组，共有  $2^n$  个子集。  
每个子集可以用一个  $n$  位的二进制数表示，其中第  $i$  位为 1 表示包含  $\text{nums}[i]$ ，为 0 表示不包含。

时间复杂度:  $O(n * 2^n)$  – 需要生成  $2^n$  个子集，每个子集需要  $O(n)$  时间构建

空间复杂度:  $O(n)$  – 不考虑输出空间

"""

```
def subsets(self, nums: List[int]) -> List[List[int]]:
```

"""

使用位运算生成所有子集

Args:

nums: 输入数组

Returns:

所有子集的列表

"""

n = len(nums)

total\_subsets = 1 << n #  $2^n$  个子集

result = []

# 遍历所有可能的二进制掩码

for mask in range(total\_subsets):

subset = []

# 检查每个位，如果为 1 则添加对应元素

for i in range(n):

if mask & (1 << i):

subset.append(nums[i])

result.append(subset)

return result

def subsets\_backtrack(self, nums: List[int]) -> List[List[int]]:

"""

回溯法实现（备选方案）

Args:

nums: 输入数组

Returns:

所有子集的列表

"""

result = []

def backtrack(start: int, current: List[int]):

result.append(current[:])

for i in range(start, len(nums)):

current.append(nums[i])

backtrack(i + 1, current)

current.pop()

backtrack(0, [])

return result

```

# 测试代码
if __name__ == "__main__":
    solution = Solution()

    test1 = [1, 2, 3]
    test2 = [0]
    test3 = [1, 2]

    result1 = solution.subsets(test1)
    result2 = solution.subsets(test2)
    result3 = solution.subsets(test3)

    print(f"Test 1 size: {len(result1)}")
    print(f"Test 2 size: {len(result2)}")
    print(f"Test 3 size: {len(result3)}")

# 打印第一个测试用例的结果
print("Test 1 subsets:")
for subset in result1:
    print(subset)

```

=====

文件: Code21\_SubsetsII.cpp

=====

```

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

/***
 * 子集 II
 * 测试链接: https://leetcode.cn/problems/subsets-ii/
 *
 * 题目描述:
 * 给你一个整数数组 nums，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。
 *
 * 解题思路:
 * 由于数组可能包含重复元素，需要先对数组排序，然后使用回溯法，在递归时跳过重复元素。
 *
 * 时间复杂度: O(n * 2^n) - 最坏情况下需要生成  $2^n$  个子集

```

```

* 空间复杂度: O(n) - 递归深度为 n
*/
class Solution {
public:
    /**
     * 使用回溯法生成所有不重复子集
     * @param nums 输入数组 (可能包含重复元素)
     * @return 所有不重复子集的向量
     */
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<vector<int>> result;
        // 先排序, 便于跳过重复元素
        sort(nums.begin(), nums.end());
        vector<int> current;
        backtrack(nums, 0, current, result);
        return result;
    }

private:
    /**
     * 回溯辅助函数
     * @param nums 排序后的数组
     * @param start 当前起始位置
     * @param current 当前子集
     * @param result 结果向量
     */
    void backtrack(vector<int>& nums, int start, vector<int>& current, vector<vector<int>>& result) {
        // 添加当前子集到结果
        result.push_back(current);

        for (int i = start; i < nums.size(); i++) {
            // 跳过重复元素, 避免生成重复子集
            if (i > start && nums[i] == nums[i - 1]) {
                continue;
            }

            current.push_back(nums[i]);
            backtrack(nums, i + 1, current, result);
            current.pop_back();
        }
    }
};

```

```

// 测试函数
int main() {
    Solution solution;

    vector<int> test1 = {1, 2, 2};
    vector<int> test2 = {0};
    vector<int> test3 = {1, 1, 2};

    auto result1 = solution.subsetsWithDup(test1);
    auto result2 = solution.subsetsWithDup(test2);
    auto result3 = solution.subsetsWithDup(test3);

    cout << "Test 1 size: " << result1.size() << endl;
    cout << "Test 2 size: " << result2.size() << endl;
    cout << "Test 3 size: " << result3.size() << endl;

    // 打印第一个测试用例的结果
    cout << "Test 1 subsets:" << endl;
    for (const auto& subset : result1) {
        cout << "[";
        for (int num : subset) {
            cout << num << " ";
        }
        cout << "]" << endl;
    }

    return 0;
}

```

=====

文件: Code21\_SubsetsII.java

=====

```

package class031;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * 子集 II
 * 测试链接: https://leetcode.cn/problems/subsets-ii/

```

```
*  
* 题目描述:  
* 给你一个整数数组 nums，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。  
* 解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。  
*  
* 示例:  
* 输入: nums = [1, 2, 2]  
* 输出: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]  
*  
* 输入: nums = [0]  
* 输出: [[], [0]]  
*  
* 提示:  
* 1 <= nums.length <= 10  
* -10 <= nums[i] <= 10  
*  
* 解题思路:  
* 由于数组可能包含重复元素，直接使用位运算会产生重复子集。  
* 需要先对数组排序，然后使用回溯法，在递归时跳过重复元素。  
*  
* 时间复杂度: O(n * 2^n) - 最坏情况下需要生成  $2^n$  个子集  
* 空间复杂度: O(n) - 递归深度为 n  
*/
```

```
public class Code21_SubsetsII {
```

```
    /**
     * 使用回溯法生成所有不重复子集
     * @param nums 输入数组（可能包含重复元素）
     * @return 所有不重复子集的列表
     */
```

```
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        // 先排序，便于跳过重复元素
        Arrays.sort(nums);
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }
```

```
    /**
     * 回溯辅助函数
     * @param nums 排序后的数组
     * @param start 当前起始位置
     * @param current 当前子集
```

```

* @param result 结果列表
*/
private void backtrack(int[] nums, int start, List<Integer> current, List<List<Integer>>
result) {
    // 添加当前子集到结果
    result.add(new ArrayList<>(current));

    for (int i = start; i < nums.length; i++) {
        // 跳过重复元素，避免生成重复子集
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }

        current.add(nums[i]);
        backtrack(nums, i + 1, current, result);
        current.remove(current.size() - 1);
    }
}

/**
 * 使用位运算的变种方法（处理重复元素）
 * 这种方法先统计每个数字的出现次数，然后根据出现次数生成子集
 * @param nums 输入数组
 * @return 所有不重复子集的列表
*/
public List<List<Integer>> subsetsWithDupBitmask(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(nums);

    int n = nums.length;
    int totalSubsets = 1 << n;

    for (int mask = 0; mask < totalSubsets; mask++) {
        List<Integer> subset = new ArrayList<>();
        boolean valid = true;

        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) != 0) {
                // 检查是否跳过重复元素
                if (i > 0 && nums[i] == nums[i - 1] && (mask & (1 << (i - 1))) == 0) {
                    valid = false;
                    break;
                }
            }
        }
        if (valid) {
            result.add(subset);
        }
    }
}

```

```

        subset.add(nums[i]);
    }
}

if (valid) {
    result.add(subset);
}
}

return result;
}

// 测试方法
public static void main(String[] args) {
    Code21_SubsetsII solution = new Code21_SubsetsII();

    int[] test1 = {1, 2, 2};
    int[] test2 = {0};
    int[] test3 = {1, 1, 2};

    System.out.println("Test 1: " + solution.subsetsWithDup(test1));
    System.out.println("Test 2: " + solution.subsetsWithDup(test2));
    System.out.println("Test 3: " + solution.subsetsWithDup(test3));
}
}
=====
```

文件: Code21\_SubsetsII.py

```

=====
from typing import List

class Solution:
    """
子集 II
测试链接: https://leetcode.cn/problems/subsets-ii/

```

**题目描述:**

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。

**解题思路:**

由于数组可能包含重复元素，需要先对数组排序，然后使用回溯法，在递归时跳过重复元素。

时间复杂度:  $O(n * 2^n)$  – 最坏情况下需要生成  $2^n$  个子集

空间复杂度:  $O(n)$  – 递归深度为  $n$

"""

```
def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
```

"""

使用回溯法生成所有不重复子集

Args:

    nums: 输入数组 (可能包含重复元素)

Returns:

    所有不重复子集的列表

"""

```
result = []
```

# 先排序, 便于跳过重复元素

```
nums.sort()
```

```
def backtrack(start: int, current: List[int]):
```

# 添加当前子集到结果

```
result.append(current[:])
```

```
for i in range(start, len(nums)):
```

# 跳过重复元素, 避免生成重复子集

```
if i > start and nums[i] == nums[i - 1]:
```

```
    continue
```

```
    current.append(nums[i])
```

```
    backtrack(i + 1, current)
```

```
    current.pop()
```

```
backtrack(0, [])
```

```
return result
```

```
def subsetsWithDupBitmask(self, nums: List[int]) -> List[List[int]]:
```

"""

使用位运算的变种方法 (处理重复元素)

这种方法先统计每个数字的出现次数, 然后根据出现次数生成子集

Args:

    nums: 输入数组

Returns:

所有不重复子集的列表

"""

```
result = []
nums.sort()
n = len(nums)
total_subsets = 1 << n

for mask in range(total_subsets):
    subset = []
    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])

    if valid:
        result.append(subset)

return result
```

# 测试代码

```
if __name__ == "__main__":
    solution = Solution()
```

```
test1 = [1, 2, 2]
test2 = [0]
test3 = [1, 1, 2]
```

```
result1 = solution.subsetsWithDup(test1)
result2 = solution.subsetsWithDup(test2)
result3 = solution.subsetsWithDup(test3)
```

```
print(f"Test 1 size: {len(result1)}")
print(f"Test 2 size: {len(result2)}")
print(f"Test 3 size: {len(result3)}")
```

```
# 打印第一个测试用例的结果
print("Test 1 subsets:")
```

```
for subset in result1:  
    print(subset)
```

---

文件: Code22\_BitAdder.cpp

---

```
#include <iostream>  
#include <climits>  
#include <stdexcept>  
using namespace std;  
  
/**  
 * 位运算实现加法器  
 * 测试链接: 自定义题目, 展示位运算在计算机底层运算中的应用  
 *  
 * 题目描述:  
 * 使用位运算实现整数的加法、减法、乘法和除法运算。  
 * 不使用+、-、*、/等算术运算符。  
 *  
 * 解题思路:  
 * 1. 加法: 使用异或运算实现不考虑进位的加法, 使用与运算和左移实现进位处理  
 * 2. 减法: 利用补码原理,  $a - b = a + (-b)$ , 通过位运算求补码  
 * 3. 乘法: 使用移位和加法实现, 类似于手算乘法  
 * 4. 除法: 使用移位和减法实现, 类似于手算除法  
 *  
 * 时间复杂度:  
 * - 加法:  $O(1)$  - 常数次循环 (最多 32 次)  
 * - 减法:  $O(1)$  - 基于加法实现  
 * - 乘法:  $O(1)$  - 最多 32 次循环  
 * - 除法:  $O(1)$  - 最多 32 次循环  
 *  
 * 空间复杂度:  $O(1)$  - 只使用常数个变量  
 */  
  
class BitAdder {  
public:  
    /**  
     * 使用位运算实现整数加法  
     * @param a 第一个加数  
     * @param b 第二个加数  
     * @return a + b 的结果  
     */  
    static int add(int a, int b) {
```

```

// 思路：加法可以分解为不考虑进位的加法（异或运算）和进位（与运算并左移）
// 重复这个过程直到进位为 0

while (b != 0) {
    // 计算不考虑进位的加法结果
    int sum = a ^ b;
    // 计算进位（需要左移一位）
    int carry = (a & b) << 1;
    // 更新 a 和 b，继续处理进位
    a = sum;
    b = carry;
}
return a;
}

/***
 * 递归实现加法
 * @param a 第一个加数
 * @param b 第二个加数
 * @return a + b 的结果
 */
static int addRecursive(int a, int b) {
    if (b == 0) {
        return a;
    }
    // 递归计算: a + b = (a ^ b) + ((a & b) << 1)
    return addRecursive(a ^ b, (a & b) << 1);
}

/***
 * 使用位运算实现整数减法
 * @param a 被减数
 * @param b 减数
 * @return a - b 的结果
 */
static int subtract(int a, int b) {
    // 利用补码原理: a - b = a + (-b)
    // 求 b 的补码（按位取反再加 1）
    return add(a, add(~b, 1));
}

/***
 * 使用位运算实现整数乘法
*/

```

```

* @param a 被乘数
* @param b 乘数
* @return a * b 的结果
*/
static int multiply(int a, int b) {
    // 处理特殊情况
    if (a == 0 || b == 0) return 0;
    if (a == 1) return b;
    if (b == 1) return a;

    // 记录结果的符号
    bool negative = false;
    if ((a < 0 && b > 0) || (a > 0 && b < 0)) {
        negative = true;
    }

    // 取绝对值
    a = abs(a);
    b = abs(b);

    int result = 0;

    // 类似于手算乘法：将 b 的每一位与 a 相乘，然后左移相应的位数
    while (b != 0) {
        // 如果 b 的最低位是 1，则加上 a 左移相应的位数
        if (b & 1) {
            result = add(result, a);
        }
        // a 左移一位，相当于乘以 2
        a <<= 1;
        // b 右移一位，处理下一位
        b >>= 1;
    }

    return negative ? -result : result;
}

/**
* 使用位运算实现整数除法（向下取整）
* @param dividend 被除数
* @param divisor 除数
* @return dividend / divisor 的结果
*/

```

```
static int divide(int dividend, int divisor) {
    // 处理特殊情况
    if (divisor == 0) {
        throw runtime_error("Division by zero");
    }
    if (dividend == 0) return 0;
    if (divisor == 1) return dividend;
    if (divisor == -1) {
        // 处理整数溢出情况
        if (dividend == INT_MIN) {
            return INT_MAX;
        }
        return -dividend;
    }

    // 记录结果的符号
    bool negative = (dividend < 0) ^ (divisor < 0);

    // 转换为正数处理（使用 long long 避免溢出）
    long long a = abs((long long)dividend);
    long long b = abs((long long)divisor);

    int result = 0;

    // 从最高位开始尝试
    for (int i = 31; i >= 0; i--) {
        if ((a >> i) >= b) {
            result = add(result, 1 << i);
            a = subtract(a, b << i);
        }
    }

    return negative ? -result : result;
}

/***
 * 更稳健的除法实现
 * @param dividend 被除数
 * @param divisor 除数
 * @return dividend / divisor 的结果
 */
static int divideRobust(int dividend, int divisor) {
    // 处理除数为0的情况
```

```
if (divisor == 0) return dividend >= 0 ? INT_MAX : INT_MIN;

// 处理被除数为最小值且除数为-1 的情况（会溢出）
if (dividend == INT_MIN && divisor == -1) {
    return INT_MAX;
}

// 确定符号
bool negative = (dividend < 0) ^ (divisor < 0);

// 转换为正数（使用 long long 避免溢出）
long long a = abs((long long)dividend);
long long b = abs((long long)divisor);

int result = 0;

// 使用减法实现除法
while (a >= b) {
    long long temp = b;
    int multiple = 1;

    // 快速倍增：每次将除数翻倍，直到接近被除数
    while (a >= (temp << 1)) {
        temp <<= 1;
        multiple <<= 1;
    }

    a -= temp;
    result += multiple;
}

return negative ? -result : result;
}

private:
    // 辅助函数：计算绝对值（避免使用标准库函数）
    static long long abs(long long x) {
        return x < 0 ? -x : x;
    }
};

// 测试函数
int main() {
```

```
// 测试加法
cout << "==== 加法测试 ===" << endl;
cout << "5 + 3 = " << BitAdder::add(5, 3) << endl; // 8
cout << "-5 + 3 = " << BitAdder::add(-5, 3) << endl; // -2
cout << "5 + (-3) = " << BitAdder::add(5, -3) << endl; // 2
cout << "-5 + (-3) = " << BitAdder::add(-5, -3) << endl; // -8

// 测试减法
cout << "\n==== 减法测试 ===" << endl;
cout << "5 - 3 = " << BitAdder::subtract(5, 3) << endl; // 2
cout << "3 - 5 = " << BitAdder::subtract(3, 5) << endl; // -2
cout << "-5 - 3 = " << BitAdder::subtract(-5, 3) << endl; // -8
cout << "5 - (-3) = " << BitAdder::subtract(5, -3) << endl; // 8

// 测试乘法
cout << "\n==== 乘法测试 ===" << endl;
cout << "5 * 3 = " << BitAdder::multiply(5, 3) << endl; // 15
cout << "5 * (-3) = " << BitAdder::multiply(5, -3) << endl; // -15
cout << "-5 * 3 = " << BitAdder::multiply(-5, 3) << endl; // -15
cout << "-5 * (-3) = " << BitAdder::multiply(-5, -3) << endl; // 15

// 测试除法
cout << "\n==== 除法测试 ===" << endl;
cout << "15 / 3 = " << BitAdder::divide(15, 3) << endl; // 5
cout << "15 / (-3) = " << BitAdder::divide(15, -3) << endl; // -5
cout << "-15 / 3 = " << BitAdder::divide(-15, 3) << endl; // -5
cout << "-15 / (-3) = " << BitAdder::divide(-15, -3) << endl; // 5
cout << "16 / 3 = " << BitAdder::divide(16, 3) << endl; // 5 (向下取整)

// 测试边界情况
cout << "\n==== 边界测试 ===" << endl;
cout << "0 + 5 = " << BitAdder::add(0, 5) << endl; // 5
cout << "5 * 0 = " << BitAdder::multiply(5, 0) << endl; // 0
cout << "0 / 5 = " << BitAdder::divide(0, 5) << endl; // 0

// 验证递归加法
cout << "\n==== 递归加法测试 ===" << endl;
cout << "5 + 3 = " << BitAdder::addRecursive(5, 3) << endl; // 8
cout << "-5 + 3 = " << BitAdder::addRecursive(-5, 3) << endl; // -2

return 0;
}
```

文件: Code22\_BitAdder.java

```
=====
package class031;
```

```
/**
```

```
* 位运算实现加法器
```

```
* 测试链接: 自定义题目, 展示位运算在计算机底层运算中的应用
```

```
*
```

```
* 题目描述:
```

```
* 使用位运算实现整数的加法、减法、乘法和除法运算。
```

```
* 不使用+、-、*、/等算术运算符。
```

```
*
```

```
* 解题思路:
```

```
* 1. 加法: 使用异或运算实现不考虑进位的加法, 使用与运算和左移实现进位处理
```

```
* 2. 减法: 利用补码原理,  $a - b = a + (-b)$ , 通过位运算求补码
```

```
* 3. 乘法: 使用移位和加法实现, 类似于手算乘法
```

```
* 4. 除法: 使用移位和减法实现, 类似于手算除法
```

```
*
```

```
* 时间复杂度:
```

```
* - 加法:  $O(1)$  - 常数次循环 (最多 32 次)
```

```
* - 减法:  $O(1)$  - 基于加法实现
```

```
* - 乘法:  $O(1)$  - 最多 32 次循环
```

```
* - 除法:  $O(1)$  - 最多 32 次循环
```

```
*
```

```
* 空间复杂度:  $O(1)$  - 只使用常数个变量
```

```
*/
```

```
public class Code22_BitAdder {
```

```
/**
```

```
* 使用位运算实现整数加法
```

```
* @param a 第一个加数
```

```
* @param b 第二个加数
```

```
* @return a + b 的结果
```

```
*/
```

```
public static int add(int a, int b) {
```

```
    // 思路: 加法可以分解为不考虑进位的加法 (异或运算) 和进位 (与运算并左移)
```

```
    // 重复这个过程直到进位为 0
```

```
    while (b != 0) {
```

```
        // 计算不考虑进位的加法结果
```

```
        int sum = a ^ b;
```

```
// 计算进位（需要左移一位）
int carry = (a & b) << 1;
// 更新 a 和 b，继续处理进位
a = sum;
b = carry;
}

return a;
}

/***
* 递归实现加法
* @param a 第一个加数
* @param b 第二个加数
* @return a + b 的结果
*/
public static int addRecursive(int a, int b) {
    if (b == 0) {
        return a;
    }
    // 递归计算: a + b = (a ^ b) + ((a & b) << 1)
    return addRecursive(a ^ b, (a & b) << 1);
}

/***
* 使用位运算实现整数减法
* @param a 被减数
* @param b 减数
* @return a - b 的结果
*/
public static int subtract(int a, int b) {
    // 利用补码原理: a - b = a + (-b)
    // 求 b 的补码 (按位取反再加 1)
    return add(a, add(~b, 1));
}

/***
* 使用位运算实现整数乘法
* @param a 被乘数
* @param b 乘数
* @return a * b 的结果
*/
public static int multiply(int a, int b) {
    // 处理特殊情况
}
```

```
if (a == 0 || b == 0) return 0;
if (a == 1) return b;
if (b == 1) return a;

// 记录结果的符号
boolean negative = false;
if ((a < 0 && b > 0) || (a > 0 && b < 0)) {
    negative = true;
}

// 取绝对值
a = Math.abs(a);
b = Math.abs(b);

int result = 0;

// 类似于手算乘法：将 b 的每一位与 a 相乘，然后左移相应的位数
while (b != 0) {
    // 如果 b 的最低位是 1，则加上 a 左移相应的位数
    if ((b & 1) != 0) {
        result = add(result, a);
    }
    // a 左移一位，相当于乘以 2
    a <<= 1;
    // b 右移一位，处理下一位
    b >>>= 1; // 使用无符号右移
}

return negative ? -result : result;
}

/***
 * 使用位运算实现整数除法（向下取整）
 * @param dividend 被除数
 * @param divisor 除数
 * @return dividend / divisor 的结果
 */
public static int divide(int dividend, int divisor) {
    // 处理特殊情况
    if (divisor == 0) {
        throw new ArithmeticException("Division by zero");
    }
    if (dividend == 0) return 0;
```

```
if (divisor == 1) return dividend;
if (divisor == -1) {
    // 处理整数溢出情况
    if (dividend == Integer.MIN_VALUE) {
        return Integer.MAX_VALUE;
    }
    return -dividend;
}

// 记录结果的符号
boolean negative = (dividend < 0) ^ (divisor < 0);

// 转换为正数处理（使用 long 避免溢出）
long a = Math.abs((long)dividend);
long b = Math.abs((long)divisor);

int result = 0;

// 从最高位开始尝试
for (int i = 31; i >= 0; i--) {
    if ((a >> i) >= b) {
        result = add(result, 1 << i);
        a = subtract((int)a, (int)(b << i));
    }
}

return negative ? -result : result;
}

/***
 * 更稳健的除法实现
 * @param dividend 被除数
 * @param divisor 除数
 * @return dividend / divisor 的结果
 */
public static int divideRobust(int dividend, int divisor) {
    // 处理除数为 0 的情况
    if (divisor == 0) return dividend >= 0 ? Integer.MAX_VALUE : Integer.MIN_VALUE;

    // 处理被除数为最小值且除数为-1 的情况（会溢出）
    if (dividend == Integer.MIN_VALUE && divisor == -1) {
        return Integer.MAX_VALUE;
    }
}
```

```
// 确定符号
boolean negative = (dividend < 0) ^ (divisor < 0);

// 转换为正数（使用 long 避免溢出）
long a = Math.abs((long)dividend);
long b = Math.abs((long)divisor);

int result = 0;

// 使用减法实现除法
while (a >= b) {
    long temp = b;
    int multiple = 1;

    // 快速倍增：每次将除数翻倍，直到接近被除数
    while (a >= (temp << 1)) {
        temp <<= 1;
        multiple <<= 1;
    }

    a -= temp;
    result += multiple;
}

return negative ? -result : result;
}

// 测试方法
public static void main(String[] args) {
    // 测试加法
    System.out.println("==> 加法测试 ==>");
    System.out.println("5 + 3 = " + add(5, 3)); // 8
    System.out.println("-5 + 3 = " + add(-5, 3)); // -2
    System.out.println("5 + (-3) = " + add(5, -3)); // 2
    System.out.println("-5 + (-3) = " + add(-5, -3)); // -8

    // 测试减法
    System.out.println("\n==> 减法测试 ==>");
    System.out.println("5 - 3 = " + subtract(5, 3)); // 2
    System.out.println("3 - 5 = " + subtract(3, 5)); // -2
    System.out.println("-5 - 3 = " + subtract(-5, 3)); // -8
    System.out.println("5 - (-3) = " + subtract(5, -3)); // 8
}
```

```

// 测试乘法
System.out.println("\n==== 乘法测试 ===");
System.out.println("5 * 3 = " + multiply(5, 3)); // 15
System.out.println("5 * (-3) = " + multiply(5, -3)); // -15
System.out.println("-5 * 3 = " + multiply(-5, 3)); // -15
System.out.println("-5 * (-3) = " + multiply(-5, -3)); // 15

// 测试除法
System.out.println("\n==== 除法测试 ===");
System.out.println("15 / 3 = " + divide(15, 3)); // 5
System.out.println("15 / (-3) = " + divide(15, -3)); // -5
System.out.println("-15 / 3 = " + divide(-15, 3)); // -5
System.out.println("-15 / (-3) = " + divide(-15, -3)); // 5
System.out.println("16 / 3 = " + divide(16, 3)); // 5 (向下取整)

// 测试边界情况
System.out.println("\n==== 边界测试 ===");
System.out.println("0 + 5 = " + add(0, 5)); // 5
System.out.println("5 * 0 = " + multiply(5, 0)); // 0
System.out.println("0 / 5 = " + divide(0, 5)); // 0

// 验证递归加法
System.out.println("\n==== 递归加法测试 ===");
System.out.println("5 + 3 = " + addRecursive(5, 3)); // 8
System.out.println("-5 + 3 = " + addRecursive(-5, 3)); // -2
}

```

```

/**
 * 工程化考量:
 * 1. 异常处理: 除法需要处理除数为 0 的情况
 * 2. 边界条件: 处理整数溢出, 特别是 Integer.MIN_VALUE 的情况
 * 3. 性能优化: 使用快速倍增技术优化除法运算
 * 4. 可读性: 添加详细注释说明位运算的原理
 * 5. 测试覆盖: 包含正数、负数、零、边界值等各种情况
 *
 * 应用场景:
 * 1. 底层硬件设计: CPU 中的 ALU 单元
 * 2. 加密算法: 需要避免使用标准算术运算符
 * 3. 面试考察: 展示对计算机底层原理的理解
 * 4. 学术研究: 理解计算机如何执行基本运算
 */

```

=====

文件: Code22\_BitAdder.py

=====

```
import sys
```

```
class BitAdder:
```

```
    """
```

位运算实现加法器

测试链接: 自定义题目, 展示位运算在计算机底层运算中的应用

题目描述:

使用位运算实现整数的加法、减法、乘法和除法运算。

不使用+、-、\*、/等算术运算符。

解题思路:

1. 加法: 使用异或运算实现不考虑进位的加法, 使用与运算和左移实现进位处理
2. 减法: 利用补码原理,  $a - b = a + (-b)$ , 通过位运算求补码
3. 乘法: 使用移位和加法实现, 类似于手算乘法
4. 除法: 使用移位和减法实现, 类似于手算除法

时间复杂度:

- 加法:  $O(1)$  - 常数次循环 (最多 32 次)
- 减法:  $O(1)$  - 基于加法实现
- 乘法:  $O(1)$  - 最多 32 次循环
- 除法:  $O(1)$  - 最多 32 次循环

空间复杂度:  $O(1)$  - 只使用常数个变量

```
"""
```

```
@staticmethod
```

```
def add(a: int, b: int) -> int:
```

```
    """
```

使用位运算实现整数加法

Args:

    a: 第一个加数

    b: 第二个加数

Returns:

    a + b 的结果

```
"""
```

```

# 由于 Python 整数没有固定位数，需要限制在 32 位范围内
MASK = 0xFFFFFFFF
MAX_INT = 0x7FFFFFFF

# 思路：加法可以分解为不考虑进位的加法（异或运算）和进位（与运算并左移）
# 重复这个过程直到进位为 0

a = a & MASK
b = b & MASK

while b != 0:
    # 计算不考虑进位的加法结果
    sum_val = a ^ b
    # 计算进位（需要左移一位）
    carry = (a & b) << 1
    # 限制在 32 位范围内
    carry = carry & MASK
    # 更新 a 和 b，继续处理进位
    a = sum_val
    b = carry

# 处理负数（Python 使用补码表示）
if a > MAX_INT:
    a = ~(a ^ MASK)

return a

```

```

@staticmethod
def add_recursive(a: int, b: int) -> int:
    """
    递归实现加法
    """

```

Args:

- a: 第一个加数
- b: 第二个加数

Returns:

a + b 的结果

```

    """
if b == 0:
    return a
# 递归计算: a + b = (a ^ b) + ((a & b) << 1)
return BitAdder.add_recursive(a ^ b, (a & b) << 1)

```

```
@staticmethod  
def subtract(a: int, b: int) -> int:  
    """
```

使用位运算实现整数减法

Args:

```
    a: 被减数  
    b: 减数
```

Returns:

a - b 的结果

```
"""
```

```
# 利用补码原理: a - b = a + (-b)  
# 求 b 的补码 (按位取反再加 1)  
return BitAdder.add(a, BitAdder.add(~b, 1))
```

```
@staticmethod  
def multiply(a: int, b: int) -> int:  
    """
```

使用位运算实现整数乘法

Args:

```
    a: 被乘数  
    b: 乘数
```

Returns:

a \* b 的结果

```
"""
```

```
# 处理特殊情况  
if a == 0 or b == 0:  
    return 0  
if a == 1:  
    return b  
if b == 1:  
    return a  
  
# 记录结果的符号  
negative = False  
if (a < 0 and b > 0) or (a > 0 and b < 0):  
    negative = True  
  
# 取绝对值
```

```
a = abs(a)
b = abs(b)

result = 0

# 类似于手算乘法：将 b 的每一位与 a 相乘，然后左移相应的位数
while b != 0:
    # 如果 b 的最低位是 1，则加上 a 左移相应的位数
    if b & 1:
        result = BitAdder.add(result, a)
    # a 左移一位，相当于乘以 2
    a <<= 1
    # b 右移一位，处理下一位
    b >>= 1

return -result if negative else result
```

```
@staticmethod
def divide(dividend: int, divisor: int) -> int:
    """
    使用位运算实现整数除法（向下取整）

```

Args:

```
    dividend: 被除数
    divisor: 除数
```

Returns:

```
    dividend / divisor 的结果
```

Raises:

```
    ZeroDivisionError: 当除数为 0 时
    """

```

```
# 处理特殊情况
if divisor == 0:
    raise ZeroDivisionError("Division by zero")
if dividend == 0:
    return 0
if divisor == 1:
    return dividend
if divisor == -1:
    # 处理整数溢出情况
    if dividend == -sys.maxsize - 1:
        return sys.maxsize
```

```
    return -dividend

# 记录结果的符号
negative = (dividend < 0) ^ (divisor < 0)

# 取绝对值
a = abs(dividend)
b = abs(divisor)

result = 0

# 从最高位开始尝试
for i in range(31, -1, -1):
    if (a >> i) >= b:
        result = BitAdder.add(result, 1 << i)
        a = BitAdder.subtract(a, b << i)

return -result if negative else result
```

```
@staticmethod
def divide_robust(dividend: int, divisor: int) -> int:
    """
    更稳健的除法实现
```

Args:

```
    dividend: 被除数
    divisor: 除数
```

Returns:

```
    dividend / divisor 的结果
    """

# 处理除数为 0 的情况
if divisor == 0:
    return sys.maxsize if dividend >= 0 else -sys.maxsize - 1
```

```
# 处理被除数为最小值且除数为-1 的情况（会溢出）
```

```
if dividend == -sys.maxsize - 1 and divisor == -1:
    return sys.maxsize
```

# 确定符号

```
negative = (dividend < 0) ^ (divisor < 0)
```

# 取绝对值

```

a = abs(dividend)
b = abs(divisor)

result = 0

# 使用减法实现除法
while a >= b:
    temp = b
    multiple = 1

    # 快速倍增: 每次将除数翻倍, 直到接近被除数
    while a >= (temp << 1):
        temp <<= 1
        multiple <<= 1

        a -= temp
        result += multiple

    return -result if negative else result

# 测试代码
if __name__ == "__main__":
    # 测试加法
    print("== 加法测试 ==")
    print(f"5 + 3 = {BitAdder.add(5, 3)}" # 8
    print(f"-5 + 3 = {BitAdder.add(-5, 3)}" # -2
    print(f"5 + (-3) = {BitAdder.add(5, -3)}" # 2
    print(f"-5 + (-3) = {BitAdder.add(-5, -3)}" # -8

    # 测试减法
    print("\n== 减法测试 ==")
    print(f"5 - 3 = {BitAdder.subtract(5, 3)}" # 2
    print(f"3 - 5 = {BitAdder.subtract(3, 5)}" # -2
    print(f"-5 - 3 = {BitAdder.subtract(-5, 3)}" # -8
    print(f"5 - (-3) = {BitAdder.subtract(5, -3)}" # 8

    # 测试乘法
    print("\n== 乘法测试 ==")
    print(f"5 * 3 = {BitAdder.multiply(5, 3)}" # 15
    print(f"5 * (-3) = {BitAdder.multiply(5, -3)}" # -15
    print(f"-5 * 3 = {BitAdder.multiply(-5, 3)}" # -15
    print(f"-5 * (-3) = {BitAdder.multiply(-5, -3)}" # 15

```

```

# 测试除法
print("\n==== 除法测试 ===")
print(f"15 / 3 = {BitAdder.divide(15, 3)}") # 5
print(f"15 / (-3) = {BitAdder.divide(15, -3)}") # -5
print(f"-15 / 3 = {BitAdder.divide(-15, 3)}") # -5
print(f"-15 / (-3) = {BitAdder.divide(-15, -3)}") # 5
print(f"16 / 3 = {BitAdder.divide(16, 3)}") # 5 (向下取整)

```

```

# 测试边界情况
print("\n==== 边界测试 ===")
print(f"0 + 5 = {BitAdder.add(0, 5)}") # 5
print(f"5 * 0 = {BitAdder.multiply(5, 0)}") # 0
print(f"0 / 5 = {BitAdder.divide(0, 5)}") # 0

```

```

# 验证递归加法
print("\n==== 递归加法测试 ===")
print(f"5 + 3 = {BitAdder.add_recursive(5, 3)}") # 8
print(f"-5 + 3 = {BitAdder.add_recursive(-5, 3)}") # -2

```

=====

文件: Code23\_FastExponentiation.cpp

=====

```

#include <iostream>
#include <vector>
#include <chrono>
using namespace std;


* 位运算实现快速幂算法
* 测试链接: https://leetcode.cn/problems/powx-n/
*
* 题目描述:
* 实现 pow(x, n)，即计算 x 的 n 次幂函数。
*
* 解题思路:
* 快速幂算法 (Exponentiation by Squaring) 利用位运算和分治思想，将时间复杂度从 O(n) 降低到 O(log n)。
* 核心思想:  $x^n = (x^2)^{(n/2)} * x^{(n \% 2)}$ 
*
* 时间复杂度: O(log n) - 每次将指数减半
* 空间复杂度: O(1) - 只使用常数个变量
*/


```

```
class FastExponentiation {
public:
    /**
     * 使用快速幂算法计算 x 的 n 次方
     * @param x 底数
     * @param n 指数
     * @return x 的 n 次方结果
     */
    static double myPow(double x, int n) {
        // 处理特殊情况
        if (n == 0) return 1.0;
        if (x == 0.0) return 0.0;
        if (x == 1.0) return 1.0;
        if (x == -1.0) return (n % 2 == 0) ? 1.0 : -1.0;

        // 处理 n 为最小值的情况（避免整数溢出）
        long long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }

        double result = 1.0;
        double current = x;

        // 使用位运算快速计算幂
        while (N > 0) {
            // 如果当前位为 1，则乘以对应的幂
            if (N & 1) {
                result *= current;
            }
            // 平方当前值
            current *= current;
            // 右移一位（相当于除以 2）
            N >>= 1;
        }

        return result;
    }

    /**
     * 递归实现快速幂算法
     * @param x 底数
     */
```

```

* @param n 指数
* @return x 的 n 次方结果
*/
static double myPowRecursive(double x, int n) {
    // 处理特殊情况
    if (n == 0) return 1.0;
    if (x == 0.0) return 0.0;

    long long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }

    return fastPow(x, N);
}

/***
 * 处理大数取模的快速幂算法（常用于密码学）
 * @param x 底数
 * @param n 指数
 * @param mod 模数
 * @return (x^n) % mod
*/
static long long modPow(long long x, long long n, long long mod) {
    if (mod == 1) return 0;
    if (n == 0) return 1;

    x = x % mod;
    long long result = 1;

    while (n > 0) {
        if (n & 1) {
            result = (result * x) % mod;
        }
        x = (x * x) % mod;
        n >>= 1;
    }

    return result;
}

/***

```

```

* 矩阵快速幂算法（用于斐波那契数列等）
* 计算矩阵的 n 次幂
* @param matrix 2x2 矩阵
* @param n 指数
* @return 矩阵的 n 次幂
*/
static vector<vector<long long>> matrixPow(vector<vector<long long>>& matrix, int n) {
    // 单位矩阵
    vector<vector<long long>> result = {{1, 0}, {0, 1}};
    vector<vector<long long>> current = matrix;

    while (n > 0) {
        if (n & 1) {
            result = multiplyMatrix(result, current);
        }
        current = multiplyMatrix(current, current);
        n >>= 1;
    }

    return result;
}

/***
 * 使用矩阵快速幂计算斐波那契数列第 n 项
 * @param n 项数
 * @return 斐波那契数列第 n 项
*/
static long long fibonacci(int n) {
    if (n <= 1) return n;

    vector<vector<long long>> base = {{1, 1}, {1, 0}};
    vector<vector<long long>> result = matrixPow(base, n - 1);
    return result[0][0];
}

private:
/***
 * 快速幂递归辅助函数
*/
static double fastPow(double x, long long n) {
    if (n == 0) return 1.0;

    double half = fastPow(x, n / 2);

```

```

    if (n % 2 == 0) {
        return half * half;
    } else {
        return half * half * x;
    }
}

/***
 * 2x2 矩阵乘法
 * @param a 第一个矩阵
 * @param b 第二个矩阵
 * @return 矩阵乘积
 */
static vector<vector<long long>> multiplyMatrix(vector<vector<long long>>& a,
vector<vector<long long>>& b) {
    vector<vector<long long>> result(2, vector<long long>(2, 0));
    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;
}
};

// 测试函数
int main() {
    // 测试基本快速幂
    cout << "==== 基本快速幂测试 ===" << endl;
    cout << "2^10 = " << FastExponentiation::myPow(2.0, 10) << endl; // 1024.0
    cout << "2.1^3 = " << FastExponentiation::myPow(2.1, 3) << endl; // 9.261
    cout << "2^-2 = " << FastExponentiation::myPow(2.0, -2) << endl; // 0.25
    cout << "0^5 = " << FastExponentiation::myPow(0.0, 5) << endl; // 0.0
    cout << "1^100 = " << FastExponentiation::myPow(1.0, 100) << endl; // 1.0

    // 测试递归实现
    cout << "\n==== 递归快速幂测试 ===" << endl;
    cout << "2^10 = " << FastExponentiation::myPowRecursive(2.0, 10) << endl; // 1024.0
    cout << "2^-2 = " << FastExponentiation::myPowRecursive(2.0, -2) << endl; // 0.25

    // 测试模幂运算
    cout << "\n==== 模幂运算测试 ===" << endl;
    cout << "3^10 mod 7 = " << FastExponentiation::modPow(3, 10, 7) << endl; // 4

```

```

cout << "2^100 mod 13 = " << FastExponentiation::modPow(2, 100, 13) << endl; // 3

// 测试矩阵快速幂（斐波那契数列）
cout << "\n==== 矩阵快速幂测试（斐波那契） ===" << endl;
cout << "F(10) = " << FastExponentiation::fibonacci(10) << endl; // 55
cout << "F(20) = " << FastExponentiation::fibonacci(20) << endl; // 6765

// 性能对比测试
cout << "\n==== 性能对比测试 ===" << endl;
auto start = chrono::high_resolution_clock::now();
double result1 = FastExponentiation::myPow(2.0, 1000000);
auto end = chrono::high_resolution_clock::now();
auto duration1 = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "快速幂耗时：" << duration1.count() << " 微秒" << endl;

start = chrono::high_resolution_clock::now();
double result2 = 1.0;
for (int i = 0; i < 1000000; i++) {
    result2 *= 2.0;
}
end = chrono::high_resolution_clock::now();
auto duration2 = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "普通幂运算耗时：" << duration2.count() << " 微秒" << endl;

return 0;
}

```

=====

文件: Code23\_FastExponentiation.java

=====

```

package class031;

/**
 * 位运算实现快速幂算法
 * 测试链接: https://leetcode.cn/problems/powx-n/
 *
 * 题目描述:
 * 实现 pow(x, n)，即计算 x 的 n 次幂函数。
 *
 * 示例:
 * 输入: x = 2.00000, n = 10
 * 输出: 1024.00000

```

```

*
* 输入: x = 2.10000, n = 3
* 输出: 9.26100
*
* 输入: x = 2.00000, n = -2
* 输出: 0.25000
* 解释:  $2^{-2} = 1/2^2 = 1/4 = 0.25$ 
*
* 提示:
*  $-100.0 < x < 100.0$ 
*  $-2^{31} \leq n \leq 2^{31}-1$ 
* n 是一个整数
* 要么 x 不为零, 要么 n > 0
*  $-10^4 \leq x^n \leq 10^4$ 
*
* 解题思路:
* 快速幂算法 (Exponentiation by Squaring) 利用位运算和分治思想, 将时间复杂度从 O(n) 降低到 O(log n)。
* 核心思想:  $x^n = (x^2)^{(n/2)} * x^{(n \% 2)}$ 
*
* 时间复杂度: O(log n) - 每次将指数减半
* 空间复杂度: O(1) - 只使用常数个变量
*/
public class Code23_FastExponentiation {

    /**
     * 使用快速幂算法计算 x 的 n 次方
     * @param x 底数
     * @param n 指数
     * @return x 的 n 次方结果
    */
    public static double myPow(double x, int n) {
        // 处理特殊情况
        if (n == 0) return 1.0;
        if (x == 0.0) return 0.0;
        if (x == 1.0) return 1.0;
        if (x == -1.0) return (n % 2 == 0) ? 1.0 : -1.0;

        // 处理 n 为最小值的情况 (避免整数溢出)
        long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        double result = 1.0;
        while (N != 0) {
            if ((N & 1) == 1) result *= x;
            x *= x;
            N >>= 1;
        }
        return result;
    }
}

```

```
}

double result = 1.0;
double current = x;

// 使用位运算快速计算幂
while (N > 0) {
    // 如果当前位为 1，则乘以对应的幂
    if ((N & 1) == 1) {
        result *= current;
    }
    // 平方当前值
    current *= current;
    // 右移一位（相当于除以 2）
    N >>= 1;
}

return result;
}

/***
 * 递归实现快速幂算法
 * @param x 底数
 * @param n 指数
 * @return x 的 n 次方结果
 */
public static double myPowRecursive(double x, int n) {
    // 处理特殊情况
    if (n == 0) return 1.0;
    if (x == 0.0) return 0.0;

    long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }

    return fastPow(x, N);
}

private static double fastPow(double x, long n) {
    if (n == 0) return 1.0;
```

```

double half = fastPow(x, n / 2);

if (n % 2 == 0) {
    return half * half;
} else {
    return half * half * x;
}

}

/***
* 处理大数取模的快速幂算法（常用于密码学）
* @param x 底数
* @param n 指数
* @param mod 模数
* @return (x^n) % mod
*/
public static long modPow(long x, long n, long mod) {
    if (mod == 1) return 0;
    if (n == 0) return 1;

    x = x % mod;
    long result = 1;

    while (n > 0) {
        if ((n & 1) == 1) {
            result = (result * x) % mod;
        }
        x = (x * x) % mod;
        n >>= 1;
    }

    return result;
}

/***
* 矩阵快速幂算法（用于斐波那契数列等）
* 计算矩阵的 n 次幂
* @param matrix 2x2 矩阵
* @param n 指数
* @return 矩阵的 n 次幂
*/
public static long[][] matrixPow(long[][] matrix, int n) {
    // 单位矩阵

```

```

long[][] result = {{1, 0}, {0, 1}};
long[][] current = matrix.clone();

while (n > 0) {
    if ((n & 1) == 1) {
        result = multiplyMatrix(result, current);
    }
    current = multiplyMatrix(current, current);
    n >>= 1;
}

return result;
}

/***
 * 2x2 矩阵乘法
 * @param a 第一个矩阵
 * @param b 第二个矩阵
 * @return 矩阵乘积
 */
private static long[][] multiplyMatrix(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;
}

/***
 * 使用矩阵快速幂计算斐波那契数列第 n 项
 * @param n 项数
 * @return 斐波那契数列第 n 项
 */
public static long fibonacci(int n) {
    if (n <= 1) return n;

    long[][] base = {{1, 1}, {1, 0}};
    long[][] result = matrixPow(base, n - 1);
    return result[0][0];
}

// 测试方法

```

```
public static void main(String[] args) {  
    // 测试基本快速幂  
    System.out.println("==> 基本快速幂测试 ==>");  
    System.out.println("2^10 = " + myPow(2.0, 10)); // 1024.0  
    System.out.println("2.1^3 = " + myPow(2.1, 3)); // 9.261  
    System.out.println("2^-2 = " + myPow(2.0, -2)); // 0.25  
    System.out.println("0^5 = " + myPow(0.0, 5)); // 0.0  
    System.out.println("1^100 = " + myPow(1.0, 100)); // 1.0  
  
    // 测试递归实现  
    System.out.println("\n==> 递归快速幂测试 ==>");  
    System.out.println("2^10 = " + myPowRecursive(2.0, 10)); // 1024.0  
    System.out.println("2^-2 = " + myPowRecursive(2.0, -2)); // 0.25  
  
    // 测试模幂运算  
    System.out.println("\n==> 模幂运算测试 ==>");  
    System.out.println("3^10 mod 7 = " + modPow(3, 10, 7)); // 3^10=59049, 59049%7=4  
    System.out.println("2^100 mod 13 = " + modPow(2, 100, 13)); // 2^100 mod 13 = 3  
  
    // 测试矩阵快速幂（斐波那契数列）  
    System.out.println("\n==> 矩阵快速幂测试（斐波那契） ==>");  
    System.out.println("F(10) = " + fibonacci(10)); // 55  
    System.out.println("F(20) = " + fibonacci(20)); // 6765  
  
    // 性能对比测试  
    System.out.println("\n==> 性能对比测试 ==>");  
    long startTime, endTime;  
  
    // 快速幂  
    startTime = System.nanoTime();  
    double result1 = myPow(2.0, 1000000);  
    endTime = System.nanoTime();  
    System.out.println("快速幂耗时: " + (endTime - startTime) + " ns");  
  
    // 普通幂运算（对比）  
    startTime = System.nanoTime();  
    double result2 = 1.0;  
    for (int i = 0; i < 1000000; i++) {  
        result2 *= 2.0;  
    }  
    endTime = System.nanoTime();  
    System.out.println("普通幂运算耗时: " + (endTime - startTime) + " ns");  
}
```

```

/**
 * 工程化考量:
 * 1. 边界条件处理: 指数为 0、底数为 0、负指数等情况
 * 2. 整数溢出: 处理 n 为 Integer.MIN_VALUE 的情况
 * 3. 精度问题: 浮点数运算的精度控制
 * 4. 性能优化: 使用位运算替代乘除法和取模运算
 * 5. 异常处理: 输入验证和错误处理
 *
 * 应用场景:
 * 1. 密码学: RSA 加密、Diffie-Hellman 密钥交换
 * 2. 数值计算: 科学计算、图形学
 * 3. 算法优化: 动态规划、组合数学
 * 4. 数学问题: 斐波那契数列、线性递推
 *
 * 算法原理:
 * 快速幂算法的核心思想是将指数 n 分解为二进制形式, 然后通过平方和乘法组合结果。
 * 例如: 计算  $3^{13}$ 
 * 13 的二进制: 1101
 *  $3^{13} = 3^{(8+4+1)} = 3^8 * 3^4 * 3^1$ 
 * 通过不断平方:  $3^1, 3^2, 3^4, 3^8\dots$ 
 * 然后根据二进制位选择需要乘入结果的幂
 */
}

=====

```

文件: Code23\_FastExponentiation.py

```

=====
import time
from typing import List

class FastExponentiation:
    """
    位运算实现快速幂算法
    测试链接: https://leetcode.cn/problems/powx-n/
    """

    def pow(self, x, n):
        if n == 0:
            return 1
        if n < 0:
            x = 1 / x
            n = -n
        result = 1
        while n:
            if n & 1:
                result *= x
            x *= x
            n >>= 1
        return result

```

题目描述:

实现  $\text{pow}(x, n)$ , 即计算  $x$  的  $n$  次幂函数。

解题思路:

快速幂算法 (Exponentiation by Squaring) 利用位运算和分治思想, 将时间复杂度从  $O(n)$  降低到  $O(\log n)$ 。

核心思想:  $x^n = (x^2)^{n/2} * x^{(n \% 2)}$

时间复杂度:  $O(\log n)$  - 每次将指数减半

空间复杂度:  $O(1)$  - 只使用常数个变量

"""

@staticmethod

```
def my_pow(x: float, n: int) -> float:  
    """
```

使用快速幂算法计算 x 的 n 次方

Args:

x: 底数

n: 指数

Returns:

x 的 n 次方结果

"""

# 处理特殊情况

```
if n == 0:
```

```
    return 1.0
```

```
if x == 0.0:
```

```
    return 0.0
```

```
if x == 1.0:
```

```
    return 1.0
```

```
if x == -1.0:
```

```
    return 1.0 if n % 2 == 0 else -1.0
```

# 处理 n 为负数的情况

```
N = n
```

```
if N < 0:
```

```
    x = 1 / x
```

```
    N = -N
```

```
result = 1.0
```

```
current = x
```

# 使用位运算快速计算幂

```
while N > 0:
```

# 如果当前位为 1，则乘以对应的幂

```
if N & 1:
```

```
    result *= current
```

# 平方当前值

```

        current *= current
        # 右移一位（相当于除以 2）
        N >>= 1

    return result

@staticmethod
def my_pow_recursive(x: float, n: int) -> float:
    """
    递归实现快速幂算法

    Args:
        x: 底数
        n: 指数

    Returns:
        x 的 n 次方结果
    """
    # 处理特殊情况
    if n == 0:
        return 1.0
    if x == 0.0:
        return 0.0

    N = n
    if N < 0:
        x = 1 / x
        N = -N

    return FastExponentiation._fast_pow(x, N)

@staticmethod
def _fast_pow(x: float, n: int) -> float:
    """递归辅助函数"""
    if n == 0:
        return 1.0

    half = FastExponentiation._fast_pow(x, n // 2)

    if n % 2 == 0:
        return half * half
    else:
        return half * half * x

```

```
@staticmethod
def mod_pow(x: int, n: int, mod: int) -> int:
    """
    处理大数取模的快速幂算法（常用于密码学）

    Args:
        x: 底数
        n: 指数
        mod: 模数

    Returns:
        (x^n) % mod
    """
    if mod == 1:
        return 0
    if n == 0:
        return 1

    x = x % mod
    result = 1

    while n > 0:
        if n & 1:
            result = (result * x) % mod
        x = (x * x) % mod
        n >>= 1

    return result

@staticmethod
def matrix_pow(matrix: List[List[int]], n: int) -> List[List[int]]:
    """
    矩阵快速幂算法（用于斐波那契数列等）
    计算矩阵的 n 次幂

    Args:
        matrix: 2x2 矩阵
        n: 指数

    Returns:
        矩阵的 n 次幂
    """
```

```
# 单位矩阵
result = [[1, 0], [0, 1]]
current = [row[:] for row in matrix] # 深拷贝

while n > 0:
    if n & 1:
        result = FastExponentiation._multiply_matrix(result, current)
    current = FastExponentiation._multiply_matrix(current, current)
    n >>= 1

return result
```

@staticmethod

```
def fibonacci(n: int) -> int:
    """
```

使用矩阵快速幂计算斐波那契数列第 n 项

Args:

n: 项数

Returns:

斐波那契数列第 n 项

"""

```
if n <= 1:
    return n
```

```
base = [[1, 1], [1, 0]]
```

```
result = FastExponentiation.matrix_pow(base, n - 1)
```

```
return result[0][0]
```

@staticmethod

```
def _multiply_matrix(a: List[List[int]], b: List[List[int]]) -> List[List[int]]:
    """
```

2x2 矩阵乘法

Args:

a: 第一个矩阵

b: 第二个矩阵

Returns:

矩阵乘积

"""

```
result = [[0, 0], [0, 0]]
```

```

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

# 测试代码
if __name__ == "__main__":
    # 测试基本快速幂
    print("==> 基本快速幂测试 ==>")
    print(f"2^10 = {FastExponentiation.my_pow(2.0, 10)}") # 1024.0
    print(f"2.1^3 = {FastExponentiation.my_pow(2.1, 3)}") # 9.261
    print(f"2^-2 = {FastExponentiation.my_pow(2.0, -2)}") # 0.25
    print(f"0^5 = {FastExponentiation.my_pow(0.0, 5)}") # 0.0
    print(f"1^100 = {FastExponentiation.my_pow(1.0, 100)}") # 1.0

    # 测试递归实现
    print("\n==> 递归快速幂测试 ==>")
    print(f"2^10 = {FastExponentiation.my_pow_recursive(2.0, 10)}") # 1024.0
    print(f"2^-2 = {FastExponentiation.my_pow_recursive(2.0, -2)}") # 0.25

    # 测试模幂运算
    print("\n==> 模幂运算测试 ==>")
    print(f"3^10 mod 7 = {FastExponentiation.mod_pow(3, 10, 7)}") # 4
    print(f"2^100 mod 13 = {FastExponentiation.mod_pow(2, 100, 13)}") # 3

    # 测试矩阵快速幂（斐波那契数列）
    print("\n==> 矩阵快速幂测试（斐波那契） ==>")
    print(f"F(10) = {FastExponentiation.fibonacci(10)}") # 55
    print(f"F(20) = {FastExponentiation.fibonacci(20)}") # 6765

    # 性能对比测试
    print("\n==> 性能对比测试 ==>")

    # 快速幂
    start_time = time.time()
    result1 = FastExponentiation.my_pow(2.0, 1000000)
    end_time = time.time()
    print(f"快速幂耗时: {(end_time - start_time) * 1000:.3f} 毫秒")

    # 普通幂运算（对比）
    start_time = time.time()
    result2 = 1.0

```

```

for i in range(1000000):
    result2 *= 2.0
end_time = time.time()
print(f"普通幂运算耗时: {(end_time - start_time) * 1000:.3f} 毫秒")

# 算法原理说明
print("\n==== 算法原理说明 ===")
print("快速幂算法的核心思想是将指数 n 分解为二进制形式，然后通过平方和乘法组合结果。")
print("例如: 计算 3^13")
print("13 的二进制: 1101")
print("3^13 = 3^(8+4+1) = 3^8 * 3^4 * 3^1")
print("通过不断平方: 3^1, 3^2, 3^4, 3^8... ")
print("然后根据二进制位选择需要乘入结果的幂")

```

---

文件: Code24\_BitManipulationTricks.cpp

---

```

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

/**
 * 位运算技巧大全
 * 测试链接: 综合题目, 展示各种位运算技巧的实际应用
 *
 * 题目描述:
 * 本文件包含各种实用的位运算技巧, 涵盖基础操作、高级技巧和应用场景。
 *
 * 解题思路:
 * 位运算技巧的核心是利用二进制表示的特性, 通过位操作实现高效的计算和数据处理。
 *
 * 时间复杂度: 各种技巧的时间复杂度不同, 但通常为 O(1) 或 O(log n)
 * 空间复杂度: O(1) - 只使用常数个变量
 */
class BitManipulationTricks {
public:
    // ===== 基础位操作技巧 =====

    /**
     * 判断一个数是否是 2 的幂
     * 原理: 2 的幂在二进制中只有一个 1

```

```

*/
static bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

/***
 * 判断一个数是否是 4 的幂
 * 原理: 4 的幂在二进制中只有一个 1, 且 1 出现在奇数位
 */
static bool isPowerOfFour(int n) {
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}

/***
 * 计算一个数的二进制表示中 1 的个数 (Brian Kernighan 算法)
 */
static int countOnes(int n) {
    int count = 0;
    while (n != 0) {
        n &= (n - 1); // 清除最右边的 1
        count++;
    }
    return count;
}

/***
 * 计算两个数的汉明距离 (不同位的个数)
 */
static int hammingDistance(int x, int y) {
    return countOnes(x ^ y);
}

/***
 * 反转一个整数的二进制位 (32 位)
 */
static unsigned int reverseBits(unsigned int n) {
    unsigned int result = 0;
    for (int i = 0; i < 32; i++) {
        result <<= 1;
        result |= (n & 1);
        n >>= 1;
    }
    return result;
}

```

```
}
```

```
// ===== 高级位操作技巧 =====
```

```
/**
```

```
* 不使用临时变量交换两个数
```

```
*/
```

```
static void swap(int& a, int& b) {
```

```
    cout << "交换前: a = " << a << ", b = " << b << endl;
```

```
    a = a ^ b;
```

```
    b = a ^ b;
```

```
    a = a ^ b;
```

```
    cout << "交换后: a = " << a << ", b = " << b << endl;
```

```
}
```

```
/**
```

```
* 找到只出现一次的数字 (其他数字都出现两次)
```

```
*/
```

```
static int singleNumber(vector<int>& nums) {
```

```
    int result = 0;
```

```
    for (int num : nums) {
```

```
        result ^= num;
```

```
}
```

```
    return result;
```

```
}
```

```
/**
```

```
* 找到只出现一次的数字 (其他数字都出现三次)
```

```
*/
```

```
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;
```

```
}
```

```
/**
```

```
* 找到数组中缺失的数字 (0 到 n 中缺失一个)
```

```
*/
```

```
static int missingNumber(vector<int>& nums) {
```

```
    int n = nums.size();
```

```

int result = n; // 因为 0 到 n 有 n+1 个数，但数组只有 n 个
for (int i = 0; i < n; i++) {
    result ^= i ^ nums[i];
}
return result;
}

// ===== 位掩码技巧 =====

/***
 * 使用位掩码表示集合（子集生成）
*/
static vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> result;
    int n = nums.size();
    int total = 1 << n;

    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;
}

/***
 * 判断一个数是否包含特定的位模式
*/
static bool hasBitPattern(int num, int pattern) {
    return (num & pattern) == pattern;
}

/***
 * 设置特定位为 1
*/
static int setBit(int num, int pos) {
    return num | (1 << pos);
}

```

```
/**  
 * 清除特定位（设为 0）  
 */  
static int clearBit(int num, int pos) {  
    return num & ~(1 << pos);  
}  
  
/**  
 * 切换特定位（0 变 1，1 变 0）  
 */  
static int toggleBit(int num, int pos) {  
    return num ^ (1 << pos);  
}  
  
/**  
 * 检查特定位是否为 1  
 */  
static bool checkBit(int num, int pos) {  
    return (num & (1 << pos)) != 0;  
}  
  
// ===== 位运算在算法中的应用 =====  
  
/**  
 * 使用位运算实现加法  
 */  
static int add(int a, int b) {  
    while (b != 0) {  
        int carry = (a & b) << 1;  
        a = a ^ b;  
        b = carry;  
    }  
    return a;  
}  
  
/**  
 * 使用位运算实现减法  
 */  
static int subtract(int a, int b) {  
    return add(a, add(~b, 1));  
}  
  
/**
```

```
* 快速判断奇偶性
*/
static bool isOdd(int n) {
    return (n & 1) == 1;
}

/***
 * 快速计算绝对值 (32 位整数)
 */
static int abs(int n) {
    int mask = n >> 31;
    return (n + mask) ^ mask;
}

/***
 * 快速计算 2 的 n 次方
 */
static int powerOfTwo(int n) {
    return 1 << n;
}

/***
 * 快速判断是否是 2 的幂的倍数
 */
static bool isMultipleOfPowerOfTwo(int n, int power) {
    return (n & ((1 << power) - 1)) == 0;
}

// ===== 位运算在优化中的应用 =====

/***
 * 快速计算 log2 (整数部分)
 */
static int log2(int n) {
    if (n <= 0) return -1;
    return 31 - __builtin_clz(n);
}

/***
 * 快速计算下一个 2 的幂 (大于等于 n 的最小 2 的幂)
 */
static int nextPowerOfTwo(int n) {
    if (n <= 0) return 1;
```

```
n--;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
return n + 1;
}

/***
 * 快速计算前一个 2 的幂（小于等于 n 的最大 2 的幂）
 */
static int prevPowerOfTwo(int n) {
    if (n <= 0) return 0;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return n - (n >> 1);
}
};
```

// ===== 布隆过滤器实现 =====

```
/***
 * 使用位集实现布隆过滤器（简化版）
 */
class BloomFilter {
private:
    vector<int> bitmap;
    int size;

public:
    BloomFilter(int size) : size(size) {
        bitmap.resize((size + 31) / 32, 0);
    }

    void add(int value) {
        int hash1 = hash1(value);
        int hash2 = hash2(value);
        int hash3 = hash3(value);
```

```
    setBit(hash1 % size);
    setBit(hash2 % size);
    setBit(hash3 % size);
}

bool contains(int value) {
    int hash1 = hash1(value);
    int hash2 = hash2(value);
    int hash3 = hash3(value);

    return checkBit(hash1 % size) &&
           checkBit(hash2 % size) &&
           checkBit(hash3 % size);
}

private:
    void setBit(int pos) {
        int index = pos / 32;
        int bit = pos % 32;
        bitmap[index] |= (1 << bit);
    }

    bool checkBit(int pos) {
        int index = pos / 32;
        int bit = pos % 32;
        return (bitmap[index] & (1 << bit)) != 0;
    }

    int hash1(int value) {
        return value * 31;
    }

    int hash2(int value) {
        return value * 17 + 12345;
    }

    int hash3(int value) {
        return value * 13 + 67890;
    }
};

// ====== 测试函数 ======
```

```
int main() {
    cout << "==== 基础位操作测试 ===" << endl;
    cout << "8 是 2 的幂: " << BitManipulationTricks::isPowerOfTwo(8) << endl; // true
    cout << "16 是 4 的幂: " << BitManipulationTricks::isPowerOfFour(16) << endl; // true
    cout << "5 的二进制中 1 的个数: " << BitManipulationTricks::countOnes(5) << endl; // 2
    cout << "1 和 4 的汉明距离: " << BitManipulationTricks::hammingDistance(1, 4) << endl; // 2

    cout << "\n==== 高级位操作测试 ===" << endl;
    int a = 5, b = 3;
    BitManipulationTricks::swap(a, b);
    vector<int> nums1 = {2, 2, 1};
    cout << "只出现一次的数字: " << BitManipulationTricks::singleNumber(nums1) << endl; // 1
    vector<int> nums2 = {0, 1, 3};
    cout << "缺失的数字: " << BitManipulationTricks::missingNumber(nums2) << endl; // 2

    cout << "\n==== 位掩码测试 ===" << endl;
    vector<int> nums3 = {1, 2, 3};
    cout << "子集数量: " << BitManipulationTricks::subsets(nums3).size() << endl; // 8
    cout << "5 包含模式 101: " << BitManipulationTricks::hasBitPattern(5, 5) << endl; // true

    cout << "\n==== 算法应用测试 ===" << endl;
    cout << "5 + 3 = " << BitManipulationTricks::add(5, 3) << endl; // 8
    cout << "5 - 3 = " << BitManipulationTricks::subtract(5, 3) << endl; // 2
    cout << "5 是奇数: " << BitManipulationTricks::isOdd(5) << endl; // true
    cout << "-5 的绝对值: " << BitManipulationTricks::abs(-5) << endl; // 5

    cout << "\n==== 优化技巧测试 ===" << endl;
    cout << "log2(8) = " << BitManipulationTricks::log2(8) << endl; // 3
    cout << "15 的下一个 2 的幂: " << BitManipulationTricks::nextPowerOfTwo(15) << endl; // 16
    cout << "15 的前一个 2 的幂: " << BitManipulationTricks::prevPowerOfTwo(15) << endl; // 8

    cout << "\n==== 布隆过滤器测试 ===" << endl;
    BloomFilter filter(100);
    filter.add(42);
    filter.add(123);
    cout << "过滤器包含 42: " << filter.contains(42) << endl; // true
    cout << "过滤器包含 456: " << filter.contains(456) << endl; // false (可能误判)

    return 0;
}
```

---

文件: Code24\_BitManipulationTricks.java

```
=====
package class031;

import java.util.*;

/**
 * 位运算技巧大全
 * 测试链接: 综合题目, 展示各种位运算技巧的实际应用
 *
 * 题目描述:
 * 本文件包含各种实用的位运算技巧, 涵盖基础操作、高级技巧和实际应用场景。
 *
 * 解题思路:
 * 位运算技巧的核心是利用二进制表示的特性, 通过位操作实现高效的计算和数据处理。
 *
 * 时间复杂度: 各种技巧的时间复杂度不同, 但通常为 O(1) 或 O(log n)
 * 空间复杂度: O(1) - 只使用常数个变量
 */
public class Code24_BitManipulationTricks {

    // ===== 基础位操作技巧 =====

    /**
     * 判断一个数是否是 2 的幂
     * 原理: 2 的幂在二进制中只有一个 1
     */
    public static boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;
    }

    /**
     * 判断一个数是否是 4 的幂
     * 原理: 4 的幂在二进制中只有一个 1, 且 1 出现在奇数位
     */
    public static boolean isPowerOfFour(int n) {
        return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
    }

    /**
     * 计算一个数的二进制表示中 1 的个数 (Brian Kernighan 算法)
     */
    public static int countOnes(int n) {
```

```

int count = 0;
while (n != 0) {
    n &= (n - 1); // 清除最右边的 1
    count++;
}
return count;
}

/***
 * 计算两个数的汉明距离（不同位的个数）
 */
public static int hammingDistance(int x, int y) {
    return countOnes(x ^ y);
}

/***
 * 反转一个整数的二进制位（32 位）
 */
public static int reverseBits(int n) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result <<= 1;
        result |= (n & 1);
        n >>= 1;
    }
    return result;
}

// ===== 高级位操作技巧 =====

/***
 * 不使用临时变量交换两个数
 */
public static void swap(int a, int b) {
    System.out.println("交换前: a = " + a + ", b = " + b);
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    System.out.println("交换后: a = " + a + ", b = " + b);
}

/***
 * 找到只出现一次的数字（其他数字都出现两次）
*/

```

```

/*
public static int singleNumber(int[] nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

/***
 * 找到只出现一次的数字（其他数字都出现三次）
 */
public static int singleNumberII(int[] nums) {
    int ones = 0, twos = 0;
    for (int num : nums) {
        ones = (ones ^ num) & ~twos;
        twos = (twos ^ num) & ~ones;
    }
    return ones;
}

/***
 * 找到数组中缺失的数字（0 到 n 中缺失一个）
 */
public static int missingNumber(int[] nums) {
    int n = nums.length;
    int result = n; // 因为 0 到 n 有 n+1 个数，但数组只有 n 个
    for (int i = 0; i < n; i++) {
        result ^= i ^ nums[i];
    }
    return result;
}

// ===== 位掩码技巧 =====

/***
 * 使用位掩码表示集合（子集生成）
 */
public static List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    int n = nums.length;
    int total = 1 << n;

```

```
for (int mask = 0; mask < total; mask++) {
    List<Integer> subset = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) != 0) {
            subset.add(nums[i]);
        }
    }
    result.add(subset);
}
return result;
}

/**
 * 判断一个数是否包含特定的位模式
 */
public static boolean hasBitPattern(int num, int pattern) {
    return (num & pattern) == pattern;
}

/**
 * 设置特定位为 1
 */
public static int setBit(int num, int pos) {
    return num | (1 << pos);
}

/**
 * 清除特定位（设为 0）
 */
public static int clearBit(int num, int pos) {
    return num & ~(1 << pos);
}

/**
 * 切换特定位（0 变 1， 1 变 0）
 */
public static int toggleBit(int num, int pos) {
    return num ^ (1 << pos);
}

/**
 * 检查特定位是否为 1
 */

```

```
public static boolean checkBit(int num, int pos) {
    return (num & (1 << pos)) != 0;
}

// ===== 位运算在算法中的应用 =====

/***
 * 使用位运算实现加法
 */
public static int add(int a, int b) {
    while (b != 0) {
        int carry = (a & b) << 1;
        a = a ^ b;
        b = carry;
    }
    return a;
}

/***
 * 使用位运算实现减法
 */
public static int subtract(int a, int b) {
    return add(a, add(~b, 1));
}

/***
 * 快速判断奇偶性
 */
public static boolean isOdd(int n) {
    return (n & 1) == 1;
}

/***
 * 快速计算绝对值 (32 位整数)
 */
public static int abs(int n) {
    int mask = n >> 31;
    return (n + mask) ^ mask;
}

/***
 * 快速计算 2 的 n 次方
*/

```

```
public static int powerOfTwo(int n) {
    return 1 << n;
}

/***
 * 快速判断是否是 2 的幂的倍数
 */
public static boolean isMultipleOfPowerOfTwo(int n, int power) {
    return (n & ((1 << power) - 1)) == 0;
}

// ===== 位运算在数据结构中的应用 =====

/***
 * 使用位集实现布隆过滤器（简化版）
 */
static class BloomFilter {
    private int[] bitmap;
    private int size;

    public BloomFilter(int size) {
        this.size = size;
        this.bitmap = new int[(size + 31) / 32];
    }

    public void add(int value) {
        int hash1 = hash1(value);
        int hash2 = hash2(value);
        int hash3 = hash3(value);

        setBit(hash1 % size);
        setBit(hash2 % size);
        setBit(hash3 % size);
    }

    public boolean contains(int value) {
        int hash1 = hash1(value);
        int hash2 = hash2(value);
        int hash3 = hash3(value);

        return checkBit(hash1 % size) &&
               checkBit(hash2 % size) &&
               checkBit(hash3 % size);
    }
}
```

```
}

private void setBit(int pos) {
    int index = pos / 32;
    int bit = pos % 32;
    bitmap[index] |= (1 << bit);
}

private boolean checkBit(int pos) {
    int index = pos / 32;
    int bit = pos % 32;
    return (bitmap[index] & (1 << bit)) != 0;
}

private int hash1(int value) {
    return value * 31;
}

private int hash2(int value) {
    return value * 17 + 12345;
}

private int hash3(int value) {
    return value * 13 + 67890;
}

// ===== 位运算在优化中的应用 =====

/***
 * 快速计算 log2 (整数部分)
 */
public static int log2(int n) {
    if (n <= 0) return -1;
    return 31 - Integer.numberOfLeadingZeros(n);
}

/***
 * 快速计算下一个 2 的幂 (大于等于 n 的最小 2 的幂)
 */
public static int nextPowerOfTwo(int n) {
    if (n <= 0) return 1;
    n--;
}
```

```

n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
return n + 1;
}

/***
 * 快速计算前一个 2 的幂（小于等于 n 的最大 2 的幂）
 */
public static int prevPowerOfTwo(int n) {
    if (n <= 0) return 0;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return n - (n >> 1);
}

// ===== 测试方法 =====

public static void main(String[] args) {
    System.out.println("== 基础位操作测试 ==");
    System.out.println("8 是 2 的幂: " + isPowerOfTwo(8)); // true
    System.out.println("16 是 4 的幂: " + isPowerOfFour(16)); // true
    System.out.println("5 的二进制中 1 的个数: " + countOnes(5)); // 2
    System.out.println("1 和 4 的汉明距离: " + hammingDistance(1, 4)); // 2

    System.out.println("\n== 高级位操作测试 ==");
    swap(5, 3);
    int[] nums1 = {2, 2, 1};
    System.out.println("只出现一次的数字: " + singleNumber(nums1)); // 1
    int[] nums2 = {0, 1, 3};
    System.out.println("缺失的数字: " + missingNumber(nums2)); // 2

    System.out.println("\n== 位掩码测试 ==");
    int[] nums3 = {1, 2, 3};
    System.out.println("子集数量: " + subsets(nums3).size()); // 8
    System.out.println("5 包含模式 101: " + hasBitPattern(5, 5)); // true

    System.out.println("\n== 算法应用测试 ==");
}

```

```

System.out.println("5 + 3 = " + add(5, 3)); // 8
System.out.println("5 - 3 = " + subtract(5, 3)); // 2
System.out.println("5 是奇数: " + isOdd(5)); // true
System.out.println("-5 的绝对值: " + abs(-5)); // 5

System.out.println("\n==== 优化技巧测试 ===");
System.out.println("log2(8) = " + log2(8)); // 3
System.out.println("15 的下一个 2 的幂: " + nextPowerOfTwo(15)); // 16
System.out.println("15 的前一个 2 的幂: " + prevPowerOfTwo(15)); // 8

System.out.println("\n==== 布隆过滤器测试 ===");
BloomFilter filter = new BloomFilter(100);
filter.add(42);
filter.add(123);
System.out.println("过滤器包含 42: " + filter.contains(42)); // true
System.out.println("过滤器包含 456: " + filter.contains(456)); // false (可能误判)
}

/***
 * 工程化考量:
 * 1. 边界条件处理: 负数、零、边界值等
 * 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原理
 * 4. 错误处理: 输入验证和异常处理
 * 5. 测试覆盖: 包含各种边界情况和特殊输入
 *
 * 应用场景:
 * 1. 算法竞赛: 快速实现各种计算
 * 2. 系统编程: 底层优化
 * 3. 加密算法: 位操作是加密的基础
 * 4. 图形学: 颜色操作、像素处理
 * 5. 网络编程: 协议解析、数据包处理
 *
 * 学习建议:
 * 1. 理解二进制表示: 掌握二进制、补码等概念
 * 2. 熟练基本操作: 与、或、异或、取反、移位
 * 3. 掌握常用技巧: Brian Kernighan 算法、位掩码等
 * 4. 实践应用: 在具体问题中应用位运算技巧
 * 5. 理解原理: 知道为什么这样操作能达到目的
 */
}
=====
```

文件: Code24\_BitManipulationTricks.py

```
=====
import sys
from typing import List
```

```
class BitManipulationTricks:
```

```
    """
位运算技巧大全
```

测试链接: [综合题目，展示各种位运算技巧的实际应用](#)

题目描述:

本文件包含各种实用的位运算技巧，涵盖基础操作、高级技巧和实际应用场景。

解题思路:

位运算技巧的核心是利用二进制表示的特性，通过位操作实现高效的计算和数据处理。

时间复杂度: 各种技巧的时间复杂度不同，但通常为  $O(1)$  或  $O(\log n)$

空间复杂度:  $O(1)$  - 只使用常数个变量

```
"""

```

```
# ===== 基础位操作技巧 =====
```

```
@staticmethod
```

```
def is_power_of_two(n: int) -> bool:
```

```
    """

```

判断一个数是否是 2 的幂

原理: 2 的幂在二进制中只有一个 1

```
    """

```

```
    return n > 0 and (n & (n - 1)) == 0
```

```
@staticmethod
```

```
def is_power_of_four(n: int) -> bool:
```

```
    """

```

判断一个数是否是 4 的幂

原理: 4 的幂在二进制中只有一个 1，且 1 出现在奇数位

```
    """

```

```
    return n > 0 and (n & (n - 1)) == 0 and (n & 0x55555555) != 0
```

```
@staticmethod
```

```
def count_ones(n: int) -> int:
```

```
    """

```

计算一个数的二进制表示中 1 的个数 (Brian Kernighan 算法)

```
"""
count = 0
while n != 0:
    n &= (n - 1)  # 清除最右边的1
    count += 1
return count

@staticmethod
def hamming_distance(x: int, y: int) -> int:
    """
    计算两个数的汉明距离（不同位的个数）
    """
    return BitManipulationTricks.count_ones(x ^ y)

@staticmethod
def reverse_bits(n: int) -> int:
    """
    反转一个整数的二进制位（32位）
    """
    result = 0
    for i in range(32):
        result <= 1
        result |= (n & 1)
        n >>= 1
    return result

# ===== 高级位操作技巧 =====
```

```
@staticmethod
def swap(a: int, b: int) -> tuple[int, int]:
    """
    不使用临时变量交换两个数
    """
    print(f"交换前: a = {a}, b = {b}")
    a = a ^ b
    b = a ^ b
    a = a ^ b
    print(f"交换后: a = {a}, b = {b}")
    return a, b
```

```
@staticmethod
def single_number(nums: List[int]) -> int:
    """
```

```

找到只出现一次的数字（其他数字都出现两次）
"""

result = 0
for num in nums:
    result ^= num
return result

@staticmethod
def single_number_ii(nums: List[int]) -> int:
    """

找到只出现一次的数字（其他数字都出现三次）
"""

ones, twos = 0, 0
for num in nums:
    ones = (ones ^ num) & ~twos
    twos = (twos ^ num) & ~ones
return ones

@staticmethod
def missing_number(nums: List[int]) -> int:
    """

找到数组中缺失的数字（0 到 n 中缺失一个）
"""

n = len(nums)
result = n # 因为 0 到 n 有 n+1 个数，但数组只有 n 个
for i in range(n):
    result ^= i ^ nums[i]
return result

# ===== 位掩码技巧 =====

@staticmethod
def subsets(nums: List[int]) -> List[List[int]]:
    """

使用位掩码表示集合（子集生成）
"""

result = []
n = len(nums)
total = 1 << n

for mask in range(total):
    subset = []
    for i in range(n):
        if mask & (1 << i):
            subset.append(nums[i])
    result.append(subset)

```

```
    if mask & (1 << i):
        subset.append(nums[i])
        result.append(subset)
    return result

@staticmethod
def has_bit_pattern(num: int, pattern: int) -> bool:
    """
    判断一个数是否包含特定的位模式
    """
    return (num & pattern) == pattern

@staticmethod
def set_bit(num: int, pos: int) -> int:
    """
    设置特定位为 1
    """
    return num | (1 << pos)

@staticmethod
def clear_bit(num: int, pos: int) -> int:
    """
    清除特定位（设为 0）
    """
    return num & ~(1 << pos)

@staticmethod
def toggle_bit(num: int, pos: int) -> int:
    """
    切换特定位（0 变 1， 1 变 0）
    """
    return num ^ (1 << pos)

@staticmethod
def check_bit(num: int, pos: int) -> bool:
    """
    检查特定位是否为 1
    """
    return (num & (1 << pos)) != 0

# ===== 位运算在算法中的应用 =====
```

```
@staticmethod
```

```
def add(a: int, b: int) -> int:
    """
    使用位运算实现加法
    """

    # 由于 Python 整数没有固定位数，需要限制在 32 位范围内
    MASK = 0xFFFFFFFF
    MAX_INT = 0x7FFFFFFF

    a = a & MASK
    b = b & MASK

    while b != 0:
        carry = (a & b) << 1
        a = a ^ b
        b = carry & MASK

    if a > MAX_INT:
        a = ~(a ^ MASK)

    return a

@staticmethod
def subtract(a: int, b: int) -> int:
    """
    使用位运算实现减法
    """

    return BitManipulationTricks.add(a, BitManipulationTricks.add(~b, 1))

@staticmethod
def is_odd(n: int) -> bool:
    """
    快速判断奇偶性
    """

    return (n & 1) == 1

@staticmethod
def abs_val(n: int) -> int:
    """
    快速计算绝对值（32 位整数）
    """

    mask = n >> 31
    return (n + mask) ^ mask
```

```
@staticmethod
def power_of_two(n: int) -> int:
    """
    快速计算 2 的 n 次方
    """
    return 1 << n

@staticmethod
def is_multiple_of_power_of_two(n: int, power: int) -> bool:
    """
    快速判断是否是 2 的幂的倍数
    """
    return (n & ((1 << power) - 1)) == 0

# ===== 位运算在优化中的应用 =====

@staticmethod
def log2(n: int) -> int:
    """
    快速计算 log2 (整数部分)
    """
    if n <= 0:
        return -1
    return n.bit_length() - 1

@staticmethod
def next_power_of_two(n: int) -> int:
    """
    快速计算下一个 2 的幂 (大于等于 n 的最小 2 的幂)
    """
    if n <= 0:
        return 1
    n -= 1
    n |= n >> 1
    n |= n >> 2
    n |= n >> 4
    n |= n >> 8
    n |= n >> 16
    return n + 1

@staticmethod
def prev_power_of_two(n: int) -> int:
    """
    
```

快速计算前一个 2 的幂（小于等于 n 的最大 2 的幂）

```
"""
```

```
if n <= 0:  
    return 0  
n |= n >> 1  
n |= n >> 2  
n |= n >> 4  
n |= n >> 8  
n |= n >> 16  
return n - (n >> 1)
```

```
class BloomFilter:
```

```
"""  
使用位集实现布隆过滤器（简化版）  
"""
```

```
def __init__(self, size: int):  
    self.size = size  
    self.bitmap = [0] * ((size + 31) // 32)
```

```
def add(self, value: int):  
    """添加元素到布隆过滤器""""  
    hash1 = self._hash1(value)  
    hash2 = self._hash2(value)  
    hash3 = self._hash3(value)
```

```
        self._set_bit(hash1 % self.size)  
        self._set_bit(hash2 % self.size)  
        self._set_bit(hash3 % self.size)
```

```
def contains(self, value: int) -> bool:
```

```
    """检查元素是否可能在布隆过滤器中""""  
    hash1 = self._hash1(value)  
    hash2 = self._hash2(value)  
    hash3 = self._hash3(value)
```

```
        return (self._check_bit(hash1 % self.size) and  
                self._check_bit(hash2 % self.size) and  
                self._check_bit(hash3 % self.size))
```

```
def _set_bit(self, pos: int):  
    """设置特定位""""  
    index = pos // 32
```

```
bit = pos % 32
self.bitmap[index] |= (1 << bit)

def _check_bit(self, pos: int) -> bool:
    """检查特定位"""
    index = pos // 32
    bit = pos % 32
    return (self.bitmap[index] & (1 << bit)) != 0

def _hash1(self, value: int) -> int:
    """第一个哈希函数"""
    return value * 31

def _hash2(self, value: int) -> int:
    """第二个哈希函数"""
    return value * 17 + 12345

def _hash3(self, value: int) -> int:
    """第三个哈希函数"""
    return value * 13 + 67890

# ====== 测试代码 ======

def main():
    print("== 基础位操作测试 ==")
    print(f"8 是 2 的幂: {BitManipulationTricks.is_power_of_two(8)}") # True
    print(f"16 是 4 的幂: {BitManipulationTricks.is_power_of_four(16)}") # True
    print(f"5 的二进制中 1 的个数: {BitManipulationTricks.count_ones(5)}") # 2
    print(f"1 和 4 的汉明距离: {BitManipulationTricks.hamming_distance(1, 4)}") # 2

    print("\n== 高级位操作测试 ==")
    a, b = 5, 3
    a, b = BitManipulationTricks.swap(a, b)
    nums1 = [2, 2, 1]
    print(f"只出现一次的数字: {BitManipulationTricks.single_number(nums1)}") # 1
    nums2 = [0, 1, 3]
    print(f"缺失的数字: {BitManipulationTricks.missing_number(nums2)}") # 2

    print("\n== 位掩码测试 ==")
    nums3 = [1, 2, 3]
    print(f"子集数量: {len(BitManipulationTricks.subsets(nums3))}") # 8
    print(f"5 包含模式 101: {BitManipulationTricks.has_bit_pattern(5, 5)}") # True
```

```

print("\n==== 算法应用测试 ===")
print(f"5 + 3 = {BitManipulationTricks.add(5, 3)}") # 8
print(f"5 - 3 = {BitManipulationTricks.subtract(5, 3)}") # 2
print(f"5 是奇数: {BitManipulationTricks.is_odd(5)}") # True
print(f"-5 的绝对值: {BitManipulationTricks.abs_val(-5)}") # 5

print("\n==== 优化技巧测试 ===")
print(f"log2(8) = {BitManipulationTricks.log2(8)}") # 3
print(f"15 的下一个 2 的幂: {BitManipulationTricks.next_power_of_two(15)}") # 16
print(f"15 的前一个 2 的幂: {BitManipulationTricks.prev_power_of_two(15)}") # 8

print("\n==== 布隆过滤器测试 ===")
filter = BloomFilter(100)
filter.add(42)
filter.add(123)
print(f"过滤器包含 42: {filter.contains(42)}") # True
print(f"过滤器包含 456: {filter.contains(456)}") # False (可能误判)

print("\n==== 算法原理说明 ===")
print("位运算技巧的核心是利用二进制表示的特性：")
print("1. 与运算(&): 用于掩码操作和清除位")
print("2. 或运算(|): 用于设置位")
print("3. 异或运算(^): 用于切换位和交换数值")
print("4. 取反运算(~): 用于求补码")
print("5. 移位运算(<<, >>): 用于快速乘除 2 的幂")

if __name__ == "__main__":
    main()

```

=====

文件: Code25\_BitwiseOperationsInRealWorld.cpp

=====

```

#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <stdexcept>
using namespace std;

/***
 * 位运算在实际工程中的应用

```

- \* 测试链接：综合题目，展示位运算在真实工程场景中的应用
- \*
- \* 题目描述：
- \* 本文件展示位运算在各种实际工程场景中的应用，包括权限系统、状态压缩、加密算法等。
- \*
- \* 解题思路：
- \* 通过具体案例展示位运算如何解决实际问题，体现其高效性和实用性。
- \*
- \* 时间复杂度：各种应用的时间复杂度不同，但通常为  $O(1)$  或  $O(\log n)$
- \* 空间复杂度： $O(1)$  - 通常只使用常数个变量
- \*/

```
class BitwiseOperationsInRealWorld {  
public:  
    // ===== 权限系统应用 =====  
  
    /**  
     * 权限系统：使用位掩码表示用户权限  
     */  
    class PermissionSystem {  
        public:  
            // 权限定义  
            static const int READ = 1 << 0;      // 0001 - 读权限  
            static const int WRITE = 1 << 1;     // 0010 - 写权限  
            static const int EXECUTE = 1 << 2; // 0100 - 执行权限  
            static const int DELETE = 1 << 3;   // 1000 - 删权限  
  
            /**  
             * 添加权限  
             */  
            static int addPermission(int current, int permission) {  
                return current | permission;  
            }  
  
            /**  
             * 移除权限  
             */  
            static int removePermission(int current, int permission) {  
                return current & ~permission;  
            }  
  
            /**  
             * 检查是否有权限  
             */
```

```

static bool hasPermission(int current, int permission) {
    return (current & permission) == permission;
}

/***
 * 切换权限（有则移除，无则添加）
 */
static int togglePermission(int current, int permission) {
    return current ^ permission;
}

/***
 * 获取所有权限列表
 */
static vector<string> getPermissionNames(int permissions) {
    vector<string> result;
    if (hasPermission(permissions, READ)) result.push_back("READ");
    if (hasPermission(permissions, WRITE)) result.push_back("WRITE");
    if (hasPermission(permissions, EXECUTE)) result.push_back("EXECUTE");
    if (hasPermission(permissions, DELETE)) result.push_back("DELETE");
    return result;
}
};

// ===== 状态压缩应用 =====

/***
 * 状态压缩：使用一个整数表示多个布尔状态
 */
class StateCompression {
public:
    /**
     * 设置状态位
     */
    static int setState(int state, int position, bool value) {
        if (value) {
            return state | (1 << position);
        } else {
            return state & ~(1 << position);
        }
    }

    /**

```

```
* 获取状态位
*/
static bool getState(int state, int position) {
    return (state & (1 << position)) != 0;
}

/***
 * 切换状态位
 */
static int toggleState(int state, int position) {
    return state ^ (1 << position);
}

/***
 * 检查状态位模式
 */
static bool checkPattern(int state, int pattern) {
    return (state & pattern) == pattern;
}

};
```

```
// ===== 颜色操作应用 =====
```

```
/***
 * 颜色操作：使用位运算处理 RGB 颜色
 */
class ColorOperations {
public:
    /**
     * 从 RGB 值创建颜色整数
     */
    static int createColor(int red, int green, int blue) {
        return (red << 16) | (green << 8) | blue;
    }
}
```

```
/**
 * 从颜色整数提取红色分量
 */
static int getRed(int color) {
    return (color >> 16) & 0xFF;
}
```

```
/***
```

```
* 从颜色整数提取绿色分量
*/
static int getGreen(int color) {
    return (color >> 8) & 0xFF;
}

/**
 * 从颜色整数提取蓝色分量
*/
static int getBlue(int color) {
    return color & 0xFF;
}

/**
 * 调整颜色亮度（乘以系数）
*/
static int adjustBrightness(int color, float factor) {
    int red = min(255, (int)(getRed(color) * factor));
    int green = min(255, (int)(getGreen(color) * factor));
    int blue = min(255, (int)(getBlue(color) * factor));
    return createColor(red, green, blue);
}

/**
 * 混合两种颜色
*/
static int blendColors(int color1, int color2, float ratio) {
    float inverseRatio = 1.0f - ratio;
    int red = (int)(getRed(color1) * ratio + getRed(color2) * inverseRatio);
    int green = (int)(getGreen(color1) * ratio + getGreen(color2) * inverseRatio);
    int blue = (int)(getBlue(color1) * ratio + getBlue(color2) * inverseRatio);
    return createColor(red, green, blue);
}

};

// ===== 性能优化应用 =====

/**
 * 性能优化：使用位运算替代算术运算
*/
class PerformanceOptimization {
public:
    /**

```

```
* 快速计算 2 的幂
*/
static int powerOfTwo(int n) {
    return 1 << n;
}

/***
 * 快速判断是否是 2 的幂
 */
static bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

/***
 * 快速计算绝对值
 */
static int abs(int n) {
    int mask = n >> 31;
    return (n + mask) ^ mask;
}

/***
 * 快速计算模 2 的幂
 */
static int modPowerOfTwo(int n, int mod) {
    return n & (mod - 1);
}

/***
 * 快速交换两个数
 */
static void swap(int arr[], int i, int j) {
    if (i != j) {
        arr[i] ^= arr[j];
        arr[j] ^= arr[i];
        arr[i] ^= arr[j];
    }
}
};

// ===== 数据结构优化应用 =====

/***
```

```
* 位集 (BitSet) 简化实现
*/
class CompactBitSet {
private:
    vector<int> data;
    int size;

public:
    CompactBitSet(int size) : size(size) {
        data.resize((size + 31) / 32, 0);
    }

    void set(int index) {
        if (index < 0 || index >= size) {
            throw out_of_range("Index out of bounds");
        }
        int arrayIndex = index / 32;
        int bitIndex = index % 32;
        data[arrayIndex] |= (1 << bitIndex);
    }

    void clear(int index) {
        if (index < 0 || index >= size) {
            throw out_of_range("Index out of bounds");
        }
        int arrayIndex = index / 32;
        int bitIndex = index % 32;
        data[arrayIndex] &= ~(1 << bitIndex);
    }

    bool get(int index) {
        if (index < 0 || index >= size) {
            throw out_of_range("Index out of bounds");
        }
        int arrayIndex = index / 32;
        int bitIndex = index % 32;
        return (data[arrayIndex] & (1 << bitIndex)) != 0;
    }

    int cardinality() {
        int count = 0;
        for (int value : data) {
            // 计算 1 的个数
        }
    }
}
```

```

        while (value != 0) {
            value &= (value - 1);
            count++;
        }
    }
    return count;
}
};

// ===== 测试函数 =====

int main() {
    cout << "==== 权限系统测试 ===" << endl;
    int userPermissions = 0;
    userPermissions = BitwiseOperationsInRealWorld::PermissionSystem::addPermission(
        userPermissions, BitwiseOperationsInRealWorld::PermissionSystem::READ);
    userPermissions = BitwiseOperationsInRealWorld::PermissionSystem::addPermission(
        userPermissions, BitwiseOperationsInRealWorld::PermissionSystem::WRITE);

    vector<string> permissionNames =
    BitwiseOperationsInRealWorld::PermissionSystem::getPermissionNames(userPermissions);
    cout << "用户权限: ";
    for (const string& name : permissionNames) {
        cout << name << " ";
    }
    cout << endl;

    cout << "有写权限: " << BitwiseOperationsInRealWorld::PermissionSystem::hasPermission(
        userPermissions, BitwiseOperationsInRealWorld::PermissionSystem::WRITE) << endl;

    cout << "\n==== 状态压缩测试 ===" << endl;
    int gameState = 0;
    gameState = BitwiseOperationsInRealWorld::StateCompression::setState(gameState, 0, true); // 玩家存活
    gameState = BitwiseOperationsInRealWorld::StateCompression::setState(gameState, 1, true); // 游戏进行中
    cout << "玩家存活: " << BitwiseOperationsInRealWorld::StateCompression::getState(gameState,
0) << endl;
    cout << "游戏进行中: " << BitwiseOperationsInRealWorld::StateCompression::getState(gameState,
1) << endl;

    cout << "\n==== 颜色操作测试 ===" << endl;
}

```

```

int redColor = BitwiseOperationsInRealWorld::ColorOperations::createColor(255, 0, 0);
int greenColor = BitwiseOperationsInRealWorld::ColorOperations::createColor(0, 255, 0);
int blendedColor = BitwiseOperationsInRealWorld::ColorOperations::blendColors(redColor,
greenColor, 0.5f);
cout << "混合颜色 - R:" <<
BitwiseOperationsInRealWorld::ColorOperations::getRed(blendedColor)
<< " G:" << BitwiseOperationsInRealWorld::ColorOperations::getGreen(blendedColor)
<< " B:" << BitwiseOperationsInRealWorld::ColorOperations::getBlue(blendedColor) <<
endl;

cout << "\n==== 性能优化测试 ===" << endl;
int array[2] = {5, 3};
BitwiseOperationsInRealWorld::PerformanceOptimization::swap(array, 0, 1);
cout << "交换后: [" << array[0] << ", " << array[1] << "]" << endl;
cout << "8是2的幂: " <<
BitwiseOperationsInRealWorld::PerformanceOptimization::isPowerOfTwo(8) << endl;
cout << "15 mod 8: " <<
BitwiseOperationsInRealWorld::PerformanceOptimization::modPowerOfTwo(15, 8) << endl;

cout << "\n==== 数据结构优化测试 ===" << endl;
BitwiseOperationsInRealWorld::CompactBitSet bitSet(100);
bitSet.set(42);
bitSet.set(57);
cout << "位 42 已设置: " << bitSet.get(42) << endl;
cout << "位 43 未设置: " << bitSet.get(43) << endl;
cout << "位集基数: " << bitSet.cardinality() << endl;

cout << "\n==== 工程化考量总结 ===" << endl;
cout << "1. 边界条件处理: 所有方法都应处理边界情况" << endl;
cout << "2. 性能优化: 位运算通常比算术运算更快" << endl;
cout << "3. 可读性: 添加详细注释说明位运算原理" << endl;
cout << "4. 错误处理: 输入验证和异常处理" << endl;
cout << "5. 测试覆盖: 包含各种边界情况和特殊输入" << endl;

return 0;
}
=====

文件: Code25_BitwiseOperationsInRealWorld.java
=====
package class031;

```

文件: Code25\_BitwiseOperationsInRealWorld.java

---

package class031;

```
import java.util.*;  
  
/**  
 * 位运算在实际工程中的应用  
 * 测试链接: 综合题目, 展示位运算在真实工程场景中的应用  
 *  
 * 题目描述:  
 * 本文件展示位运算在各种实际工程场景中的应用, 包括权限系统、状态压缩、加密算法等。  
 *  
 * 解题思路:  
 * 通过具体案例展示位运算如何解决实际问题, 体现其高效性和实用性。  
 *  
 * 时间复杂度: 各种应用的时间复杂度不同, 但通常为 O(1) 或 O(log n)  
 * 空间复杂度: O(1) - 通常只使用常数个变量  
 */  
  
public class Code25_BitwiseOperationsInRealWorld {  
  
    // ===== 权限系统应用 =====  
  
    /**  
     * 权限系统: 使用位掩码表示用户权限  
     * 每种权限用一个位表示, 可以高效地进行权限组合和检查  
     */  
  
    public static class PermissionSystem {  
        // 权限定义  
        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  
  
        /**  
         * 添加权限  
         */  
        public static int addPermission(int current, int permission) {  
            return current | permission;  
        }  
  
        /**  
         * 移除权限  
         */  
        public static int removePermission(int current, int permission) {  
            return current & ~permission;  
        }  
    }  
}
```

```
/**  
 * 检查是否有权限  
 */  
public static boolean hasPermission(int current, int permission) {  
    return (current & permission) == permission;  
}  
  
/**  
 * 切换权限（有则移除，无则添加）  
 */  
public static int togglePermission(int current, int permission) {  
    return current ^ permission;  
}  
  
/**  
 * 获取所有权限列表  
 */  
public static List<String> getPermissionNames(int permissions) {  
    List<String> result = new ArrayList<>();  
    if (hasPermission(permissions, READ)) result.add("READ");  
    if (hasPermission(permissions, WRITE)) result.add("WRITE");  
    if (hasPermission(permissions, EXECUTE)) result.add("EXECUTE");  
    if (hasPermission(permissions, DELETE)) result.add("DELETE");  
    return result;  
}  
}  
  
// ===== 状态压缩应用 =====  
  
/**  
 * 状态压缩：使用一个整数表示多个布尔状态  
 * 常用于动态规划、游戏状态等场景  
 */  
public static class StateCompression {  
    /**  
     * 设置状态位  
     */  
    public static int setState(int state, int position, boolean value) {  
        if (value) {  
            return state | (1 << position);  
        } else {  
            return state & ~(1 << position);  
    }  
}
```

```
        }
    }

/***
 * 获取状态位
 */
public static boolean getState(int state, int position) {
    return (state & (1 << position)) != 0;
}

/***
 * 切换状态位
 */
public static int toggleState(int state, int position) {
    return state ^ (1 << position);
}

/***
 * 检查状态位模式
 */
public static boolean checkPattern(int state, int pattern) {
    return (state & pattern) == pattern;
}

}

// ===== 颜色操作应用 =====

/***
 * 颜色操作：使用位运算处理 RGB 颜色
 * 常用于图形学、图像处理等场景
 */
public static class ColorOperations {

    /**
     * 从 RGB 值创建颜色整数
     */
    public static int createColor(int red, int green, int blue) {
        return (red << 16) | (green << 8) | blue;
    }

    /**
     * 从颜色整数提取红色分量
     */
    public static int getRed(int color) {
```

```
        return (color >> 16) & 0xFF;
    }

    /**
     * 从颜色整数提取绿色分量
     */
    public static int getGreen(int color) {
        return (color >> 8) & 0xFF;
    }

    /**
     * 从颜色整数提取蓝色分量
     */
    public static int getBlue(int color) {
        return color & 0xFF;
    }

    /**
     * 调整颜色亮度（乘以系数）
     */
    public static int adjustBrightness(int color, float factor) {
        int red = Math.min(255, (int)(getRed(color) * factor));
        int green = Math.min(255, (int)(getGreen(color) * factor));
        int blue = Math.min(255, (int)(getBlue(color) * factor));
        return createColor(red, green, blue);
    }

    /**
     * 混合两种颜色
     */
    public static int blendColors(int color1, int color2, float ratio) {
        float inverseRatio = 1.0f - ratio;
        int red = (int)(getRed(color1) * ratio + getRed(color2) * inverseRatio);
        int green = (int)(getGreen(color1) * ratio + getGreen(color2) * inverseRatio);
        int blue = (int)(getBlue(color1) * ratio + getBlue(color2) * inverseRatio);
        return createColor(red, green, blue);
    }
}

// ===== 网络协议应用 =====

/**
 * 网络协议：解析和构建数据包头部

```

```
* 常用于网络编程、协议解析等场景
*/
public static class NetworkProtocol {
    /**
     * 构建 IP 头部（简化版）
     */
    public static int buildIPHeader(int version, int headerLength,
                                    int typeOfService, int totalLength,
                                    int identification, int flags,
                                    int fragmentOffset, int ttl,
                                    int protocol, int checksum,
                                    int sourceIP, int destIP) {
        int header = 0;
        header |= (version & 0xF) << 28;
        header |= (headerLength & 0xF) << 24;
        header |= (typeOfService & 0xFF) << 16;
        header |= (totalLength & 0xFFFF);

        // 继续构建其他字段...
        return header;
    }

    /**
     * 解析 IP 头部版本
     */
    public static int parseIPVersion(int ipHeader) {
        return (ipHeader >> 28) & 0xF;
    }

    /**
     * 解析 IP 头部长度
     */
    public static int parseHeaderLength(int ipHeader) {
        return (ipHeader >> 24) & 0xF;
    }
}

// ===== 加密算法应用 =====

/**
 * 简单加密算法：使用位运算进行数据加密
 * 注意：这只是教学示例，不适用于实际加密
*/
```

```
public static class SimpleEncryption {  
    /**  
     * 简单异或加密  
     */  
    public static byte[] xorEncrypt(byte[] data, byte[] key) {  
        byte[] encrypted = new byte[data.length];  
        for (int i = 0; i < data.length; i++) {  
            encrypted[i] = (byte)(data[i] ^ key[i % key.length]);  
        }  
        return encrypted;  
    }  
  
    /**  
     * 简单位置换加密  
     */  
    public static byte bitPermutation(byte data) {  
        // 简单的位置换：交换高低 4 位  
        return (byte)((((data & 0xF0) >> 4) | ((data & 0x0F) << 4));  
    }  
}  
  
// ===== 性能优化应用 =====
```

```
/**  
 * 性能优化：使用位运算替代算术运算  
 * 常用于性能敏感的场景  
 */  
public static class PerformanceOptimization {  
    /**  
     * 快速计算 2 的幂  
     */  
    public static int powerOfTwo(int n) {  
        return 1 << n;  
    }  
  
    /**  
     * 快速判断是否是 2 的幂  
     */  
    public static boolean isPowerOfTwo(int n) {  
        return n > 0 && (n & (n - 1)) == 0;  
    }  
  
    /**
```

```
* 快速计算绝对值
*/
public static int abs(int n) {
    int mask = n >> 31;
    return (n + mask) ^ mask;
}

/***
 * 快速计算模 2 的幂
 */
public static int modPowerOfTwo(int n, int mod) {
    return n & (mod - 1);
}

/***
 * 快速交换两个数
 */
public static void swap(int[] arr, int i, int j) {
    if (i != j) {
        arr[i] ^= arr[j];
        arr[j] ^= arr[i];
        arr[i] ^= arr[j];
    }
}
}

// ===== 数据结构优化应用 =====

/***
 * 数据结构优化：使用位运算优化数据结构
 * 常用于空间敏感的场景
 */
public static class DataStructureOptimization {

    /**
     * 位集 (BitSet) 简化实现
     */

    public static class CompactBitSet {
        private int[] data;
        private int size;

        public CompactBitSet(int size) {
            this.size = size;
            this.data = new int[(size + 31) / 32];
        }
    }
}
```

```
}

public void set(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    int arrayIndex = index / 32;
    int bitIndex = index % 32;
    data[arrayIndex] |= (1 << bitIndex);
}

public void clear(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    int arrayIndex = index / 32;
    int bitIndex = index % 32;
    data[arrayIndex] &= ~(1 << bitIndex);
}

public boolean get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    int arrayIndex = index / 32;
    int bitIndex = index % 32;
    return (data[arrayIndex] & (1 << bitIndex)) != 0;
}

public int cardinality() {
    int count = 0;
    for (int value : data) {
        count += Integer.bitCount(value);
    }
    return count;
}

}

/***
 * 布隆过滤器简化实现
 */
public static class SimpleBloomFilter {
    private CompactBitSet bitSet;
```

```
private int size;
private int[] hashSeeds;

public SimpleBloomFilter(int size, int numHashes) {
    this.size = size;
    this.bitSet = new CompactBitSet(size);
    this.hashSeeds = new int[numHashes];
    // 初始化哈希种子
    for (int i = 0; i < numHashes; i++) {
        hashSeeds[i] = i * 31 + 12345;
    }
}

public void add(String element) {
    for (int seed : hashSeeds) {
        int hash = Math.abs(element.hashCode() ^ seed) % size;
        bitSet.set(hash);
    }
}

public boolean mightContain(String element) {
    for (int seed : hashSeeds) {
        int hash = Math.abs(element.hashCode() ^ seed) % size;
        if (!bitSet.get(hash)) {
            return false;
        }
    }
    return true;
}

// ===== 测试方法 =====

public static void main(String[] args) {
    System.out.println("== 权限系统测试 ==");
    int userPermissions = 0;
    userPermissions = PermissionSystem.addPermission(userPermissions, PermissionSystem.READ);
    userPermissions = PermissionSystem.addPermission(userPermissions,
PermissionSystem.WRITE);
    System.out.println("用户权限: " + PermissionSystem.getPermissionNames(userPermissions));
    System.out.println("有写权限: " + PermissionSystem.hasPermission(userPermissions,
PermissionSystem.WRITE));
}
```

```

System.out.println("\n==== 状态压缩测试 ===");
int gameState = 0;
gameState = StateCompression.setState(gameState, 0, true); // 玩家存活
gameState = StateCompression.setState(gameState, 1, true); // 游戏进行中
System.out.println("玩家存活: " + StateCompression.getState(gameState, 0));
System.out.println("游戏进行中: " + StateCompression.getState(gameState, 1));

System.out.println("\n==== 颜色操作测试 ===");
int redColor = ColorOperations.createColor(255, 0, 0);
int greenColor = ColorOperations.createColor(0, 255, 0);
int blendedColor = ColorOperations.blendColors(redColor, greenColor, 0.5f);
System.out.println("混合颜色 - R:" + ColorOperations.getRed(blendedColor) +
    " G:" + ColorOperations.getGreen(blendedColor) +
    " B:" + ColorOperations.getBlue(blendedColor));

System.out.println("\n==== 性能优化测试 ===");
int[] array = {5, 3};
PerformanceOptimization.swap(array, 0, 1);
System.out.println("交换后: [" + array[0] + ", " + array[1] + "]");
System.out.println("8 是 2 的幂: " + PerformanceOptimization.isPowerOfTwo(8));
System.out.println("15 mod 8: " + PerformanceOptimization.modPowerOfTwo(15, 8));

System.out.println("\n==== 数据结构优化测试 ===");
DataStructureOptimization.CompactBitSet bitSet = new
DataStructureOptimization.CompactBitSet(100);
bitSet.set(42);
bitSet.set(57);
System.out.println("位 42 已设置: " + bitSet.get(42));
System.out.println("位 43 未设置: " + bitSet.get(43));
System.out.println("位集基数: " + bitSet.cardinality());

System.out.println("\n==== 工程化考量总结 ===");
System.out.println("1. 边界条件处理: 所有方法都应处理边界情况");
System.out.println("2. 性能优化: 位运算通常比算术运算更快");
System.out.println("3. 可读性: 添加详细注释说明位运算原理");
System.out.println("4. 错误处理: 输入验证和异常处理");
System.out.println("5. 测试覆盖: 包含各种边界情况和特殊输入");

}

/***
 * 工程化考量:
 * 1. 边界条件处理: 所有方法都应处理边界情况
 */

```

- \* 2. 性能优化：位运算通常比算术运算更快，但要注意可读性
- \* 3. 可读性：添加详细注释说明位运算原理
- \* 4. 错误处理：输入验证和异常处理
- \* 5. 测试覆盖：包含各种边界情况和特殊输入
- \*
- \* 应用场景总结：
- \* 1. 权限系统：高效管理用户权限
- \* 2. 状态压缩：节省内存空间
- \* 3. 图形学：快速处理颜色和像素
- \* 4. 网络编程：解析和构建协议头部
- \* 5. 加密算法：基础位操作
- \* 6. 性能优化：替代昂贵的算术运算
- \* 7. 数据结构：优化空间使用
- \*
- \* 学习建议：
- \* 1. 理解二进制：掌握二进制表示和位运算原理
- \* 2. 实践应用：在具体项目中应用位运算技巧
- \* 3. 性能测试：比较位运算和传统方法的性能差异
- \* 4. 代码审查：确保位运算代码的可读性和正确性
- \* 5. 持续学习：关注新的位运算技巧和应用场景

\*/

}

---

文件: Code25\_BitwiseOperationsInRealWorld.py

---

```
from typing import List, Tuple
import math

class BitwiseOperationsInRealWorld:
```

"""

位运算在实际工程中的应用

测试链接：综合题目，展示位运算在真实工程场景中的应用

题目描述：

本文件展示位运算在各种实际工程场景中的应用，包括权限系统、状态压缩、加密算法等。

解题思路：

通过具体案例展示位运算如何解决实际问题，体现其高效性和实用性。

时间复杂度：各种应用的时间复杂度不同，但通常为  $O(1)$  或  $O(\log n)$

空间复杂度： $O(1)$  - 通常只使用常数个变量

```
"""
# ===== 权限系统应用 =====

class PermissionSystem:
    """权限系统：使用位掩码表示用户权限"""

    # 权限定义
    READ = 1 << 0      # 0001 - 读权限
    WRITE = 1 << 1     # 0010 - 写权限
    EXECUTE = 1 << 2   # 0100 - 执行权限
    DELETE = 1 << 3    # 1000 - 删权限

    @staticmethod
    def add_permission(current: int, permission: int) -> int:
        """添加权限"""
        return current | permission

    @staticmethod
    def remove_permission(current: int, permission: int) -> int:
        """移除权限"""
        return current & ~permission

    @staticmethod
    def has_permission(current: int, permission: int) -> bool:
        """检查是否有权限"""
        return (current & permission) == permission

    @staticmethod
    def toggle_permission(current: int, permission: int) -> int:
        """切换权限（有则移除，无则添加）"""
        return current ^ permission

    @staticmethod
    def get_permission_names(permissions: int) -> List[str]:
        """获取所有权限列表"""
        result = []
        if BitwiseOperationsInRealWorld.PermissionSystem.has_permission(permissions,
BitwiseOperationsInRealWorld.PermissionSystem.READ):
            result.append("READ")
        if BitwiseOperationsInRealWorld.PermissionSystem.has_permission(permissions,
BitwiseOperationsInRealWorld.PermissionSystem.WRITE):
            result.append("WRITE")
```

```
if BitwiseOperationsInRealWorld.PermissionSystem.has_permission(permissions,
BitwiseOperationsInRealWorld.PermissionSystem.EXECUTE):
    result.append("EXECUTE")
    if BitwiseOperationsInRealWorld.PermissionSystem.has_permission(permissions,
BitwiseOperationsInRealWorld.PermissionSystem.DELETE):
        result.append("DELETE")
    return result

# ===== 状态压缩应用 =====

class StateCompression:
    """状态压缩：使用一个整数表示多个布尔状态"""

    @staticmethod
    def set_state(state: int, position: int, value: bool) -> int:
        """设置状态位"""
        if value:
            return state | (1 << position)
        else:
            return state & ~(1 << position)

    @staticmethod
    def get_state(state: int, position: int) -> bool:
        """获取状态位"""
        return (state & (1 << position)) != 0

    @staticmethod
    def toggle_state(state: int, position: int) -> int:
        """切换状态位"""
        return state ^ (1 << position)

    @staticmethod
    def check_pattern(state: int, pattern: int) -> bool:
        """检查状态位模式"""
        return (state & pattern) == pattern

# ===== 颜色操作应用 =====

class ColorOperations:
    """颜色操作：使用位运算处理 RGB 颜色"""

    @staticmethod
    def create_color(red: int, green: int, blue: int) -> int:
```

```
"""从 RGB 值创建颜色整数"""
return (red << 16) | (green << 8) | blue

@staticmethod
def get_red(color: int) -> int:
    """从颜色整数提取红色分量"""
    return (color >> 16) & 0xFF

@staticmethod
def get_green(color: int) -> int:
    """从颜色整数提取绿色分量"""
    return (color >> 8) & 0xFF

@staticmethod
def get_blue(color: int) -> int:
    """从颜色整数提取蓝色分量"""
    return color & 0xFF

@staticmethod
def adjust_brightness(color: int, factor: float) -> int:
    """调整颜色亮度（乘以系数）"""
    red = min(255, int(BitwiseOperationsInRealWorld.ColorOperations.get_red(color) * factor))
    green = min(255, int(BitwiseOperationsInRealWorld.ColorOperations.get_green(color) * factor))
    blue = min(255, int(BitwiseOperationsInRealWorld.ColorOperations.get_blue(color) * factor))
    return BitwiseOperationsInRealWorld.ColorOperations.create_color(red, green, blue)

@staticmethod
def blend_colors(color1: int, color2: int, ratio: float) -> int:
    """混合两种颜色"""
    inverse_ratio = 1.0 - ratio
    red = int(BitwiseOperationsInRealWorld.ColorOperations.get_red(color1) * ratio +
              BitwiseOperationsInRealWorld.ColorOperations.get_red(color2) * inverse_ratio)
    green = int(BitwiseOperationsInRealWorld.ColorOperations.get_green(color1) * ratio +
               BitwiseOperationsInRealWorld.ColorOperations.get_green(color2) * inverse_ratio)
    blue = int(BitwiseOperationsInRealWorld.ColorOperations.get_blue(color1) * ratio +
              BitwiseOperationsInRealWorld.ColorOperations.get_blue(color2) * inverse_ratio)
    return BitwiseOperationsInRealWorld.ColorOperations.create_color(red, green, blue)
```

```
# ===== 性能优化应用 =====
```

```

class PerformanceOptimization:

    """性能优化：使用位运算替代算术运算"""

    @staticmethod
    def power_of_two(n: int) -> int:
        """快速计算 2 的幂"""
        return 1 << n

    @staticmethod
    def is_power_of_two(n: int) -> bool:
        """快速判断是否是 2 的幂"""
        return n > 0 and (n & (n - 1)) == 0

    @staticmethod
    def abs_val(n: int) -> int:
        """快速计算绝对值"""
        mask = n >> 31
        return (n + mask) ^ mask

    @staticmethod
    def mod_power_of_two(n: int, mod: int) -> int:
        """快速计算模 2 的幂"""
        return n & (mod - 1)

    @staticmethod
    def swap(arr: List[int], i: int, j: int):
        """快速交换两个数"""
        if i != j:
            arr[i] ^= arr[j]
            arr[j] ^= arr[i]
            arr[i] ^= arr[j]

# ===== 数据结构优化应用 =====

class CompactBitSet:

    """位集（BitSet）简化实现"""

    def __init__(self, size: int):
        self.size = size
        self.data = [0] * ((size + 31) // 32)

    def set(self, index: int):
        """设置特定位"""

```

```
    if index < 0 or index >= self.size:
        raise IndexError("Index out of bounds")
    array_index = index // 32
    bit_index = index % 32
    self.data[array_index] |= (1 << bit_index)

def clear(self, index: int):
    """清除特定位"""
    if index < 0 or index >= self.size:
        raise IndexError("Index out of bounds")
    array_index = index // 32
    bit_index = index % 32
    self.data[array_index] &= ~(1 << bit_index)

def get(self, index: int) -> bool:
    """获取特定位"""
    if index < 0 or index >= self.size:
        raise IndexError("Index out of bounds")
    array_index = index // 32
    bit_index = index % 32
    return (self.data[array_index] & (1 << bit_index)) != 0

def cardinality(self) -> int:
    """计算 1 的个数"""
    count = 0
    for value in self.data:
        # 计算 1 的个数
        while value != 0:
            value &= (value - 1)
            count += 1
    return count

# ===== 测试代码 =====

def main():
    print("== 权限系统测试 ==")
    user_permissions = 0
    user_permissions = BitwiseOperationsInRealWorld.PermissionSystem.add_permission(
        user_permissions, BitwiseOperationsInRealWorld.PermissionSystem.READ)
    user_permissions = BitwiseOperationsInRealWorld.PermissionSystem.add_permission(
        user_permissions, BitwiseOperationsInRealWorld.PermissionSystem.WRITE)

    permission_names =
```

```
BitwiseOperationsInRealWorld.PermissionSystem.get_permission_names(user_permissions)
print(f"用户权限: {permission_names}")
print(f"有写权限: {BitwiseOperationsInRealWorld.PermissionSystem.has_permission(
    user_permissions, BitwiseOperationsInRealWorld.PermissionSystem.WRITE)}")

print("\n==== 状态压缩测试 ===")
game_state = 0
game_state = BitwiseOperationsInRealWorld.StateCompression.set_state(game_state, 0, True) # 玩家存活
game_state = BitwiseOperationsInRealWorld.StateCompression.set_state(game_state, 1, True) # 游戏进行中
print(f"玩家存活: {BitwiseOperationsInRealWorld.StateCompression.get_state(game_state, 0)}")
print(f"游戏进行中: {BitwiseOperationsInRealWorld.StateCompression.get_state(game_state, 1)}")

print("\n==== 颜色操作测试 ===")
red_color = BitwiseOperationsInRealWorld.ColorOperations.create_color(255, 0, 0)
green_color = BitwiseOperationsInRealWorld.ColorOperations.create_color(0, 255, 0)
blended_color = BitwiseOperationsInRealWorld.ColorOperations.blend_colors(red_color, green_color, 0.5)
print(f"混合颜色 - R: {BitwiseOperationsInRealWorld.ColorOperations.get_red(blended_color)} "
      f"G: {BitwiseOperationsInRealWorld.ColorOperations.get_green(blended_color)} "
      f"B: {BitwiseOperationsInRealWorld.ColorOperations.get_blue(blended_color)}")

print("\n==== 性能优化测试 ===")
array = [5, 3]
BitwiseOperationsInRealWorld.PerformanceOptimization.swap(array, 0, 1)
print(f"交换后: {array}")
print(f"8 是 2 的幂: {BitwiseOperationsInRealWorld.PerformanceOptimization.is_power_of_two(8)}")
print(f"15 mod 8: {BitwiseOperationsInRealWorld.PerformanceOptimization.mod_power_of_two(15, 8)}")

print("\n==== 数据结构优化测试 ===")
bit_set = BitwiseOperationsInRealWorld.CompactBitSet(100)
bit_set.set(42)
bit_set.set(57)
print(f"位 42 已设置: {bit_set.get(42)}")
print(f"位 43 未设置: {bit_set.get(43)}")
print(f"位集基数: {bit_set.cardinality()}")

print("\n==== 工程化考量总结 ===")
print("1. 边界条件处理: 所有方法都应处理边界情况")
```

```
print("2. 性能优化: 位运算通常比算术运算更快")
print("3. 可读性: 添加详细注释说明位运算原理")
print("4. 错误处理: 输入验证和异常处理")
print("5. 测试覆盖: 包含各种边界情况和特殊输入")

print("\n== 应用场景总结 ==")
print("1. 权限系统: 高效管理用户权限")
print("2. 状态压缩: 节省内存空间")
print("3. 图形学: 快速处理颜色和像素")
print("4. 性能优化: 替代昂贵的算术运算")
print("5. 数据结构: 优化空间使用")

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

=====

文件: Code26\_SingleNumberIII.cpp

=====

```
/***
 * 只出现一次的数字 III
 * 测试链接: https://leetcode.cn/problems/single-number-iii/
 *
 * 题目描述:
 * 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。你可以按任意顺序返回答案。
 *
 * 解题思路:
 * 1. 首先对所有数字进行异或操作，得到两个不同数字的异或结果
 * 2. 找到异或结果中最低位的 1，这个位置表示两个数字在该位不同
 * 3. 根据这个位将数组分成两组，分别进行异或操作得到两个结果
 *
 * 时间复杂度: O(n) - 遍历数组两次
 * 空间复杂度: O(1) - 只使用常数个变量
 */
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

class Code26_SingleNumberIII {
public:
    /**
     * 只出现一次的数字 III
     * 测试链接: https://leetcode.cn/problems/single-number-iii/
     *
     * 题目描述:
     * 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。你可以按任意顺序返回答案。
     *
     * 解题思路:
     * 1. 首先对所有数字进行异或操作，得到两个不同数字的异或结果
     * 2. 找到异或结果中最低位的 1，这个位置表示两个数字在该位不同
     * 3. 根据这个位将数组分成两组，分别进行异或操作得到两个结果
     *
     * 时间复杂度: O(n) - 遍历数组两次
     * 空间复杂度: O(1) - 只使用常数个变量
     */
}
```

```

* 找出只出现一次的两个数字
* @param nums 整数数组
* @return 只出现一次的两个数字
*/
vector<int> singleNumber(vector<int>& nums) {
    if (nums.size() < 2) {
        throw invalid_argument("数组长度必须至少为 2");
    }

    // 第一步：计算所有数字的异或结果
    int xorResult = 0;
    for (int num : nums) {
        xorResult ^= num;
    }

    // 第二步：找到异或结果中最低位的 1
    // 技巧：xorResult & (-xorResult) 可以快速找到最低位的 1
    // 注意：对于负数，需要确保正确处理
    int lowestOneBit = xorResult & (-xorResult);

    // 第三步：根据最低位的 1 将数组分成两组
    vector<int> result(2, 0);
    for (int num : nums) {
        // 根据该位是否为 0 进行分组
        if ((num & lowestOneBit) == 0) {
            result[0] ^= num; // 第一组
        } else {
            result[1] ^= num; // 第二组
        }
    }

    return result;
}

/**
 * 测试方法
 */
static void test() {
    Code26_SingleNumberIII solution;

    // 测试用例 1：正常情况
    vector<int> nums1 = {1, 2, 1, 3, 2, 5};
    vector<int> result1 = solution.singleNumber(nums1);
}

```

```

cout << "测试用例 1 结果: [" << result1[0] << ", " << result1[1] << "]" << endl;
// 预期输出: [3, 5] 或 [5, 3]

// 测试用例 2: 包含负数
vector<int> nums2 = {-1, 0, -1, 2, 0, 3};
vector<int> result2 = solution.singleNumber(nums2);
cout << "测试用例 2 结果: [" << result2[0] << ", " << result2[1] << "]" << endl;
// 预期输出: [2, 3] 或 [3, 2]

// 测试用例 3: 边界情况
vector<int> nums3 = {0, 1};
vector<int> result3 = solution.singleNumber(nums3);
cout << "测试用例 3 结果: [" << result3[0] << ", " << result3[1] << "]" << endl;
// 预期输出: [0, 1] 或 [1, 0]

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "时间复杂度: O(n) - 遍历数组两次" << endl;
cout << "空间复杂度: O(1) - 只使用常数个变量" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 输入验证: 检查数组长度" << endl;
cout << "2. 边界处理: 处理负数情况" << endl;
cout << "3. 性能优化: 使用位运算替代算术运算" << endl;
cout << "4. 可读性: 添加详细注释说明算法原理" << endl;
cout << "5. 异常处理: 使用异常处理输入错误" << endl;
}

};

int main() {
    Code26_SingleNumberIII::test();
    return 0;
}
=====
```

文件: Code26\_SingleNumberIII.java

```

/**
 * 只出现一次的数字 III
 * 测试链接: https://leetcode.cn/problems/single-number-iii/
 *
```

- \* 题目描述:
- \* 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。你可以按任意顺序返回答案。
- \*
- \* 解题思路:
- \* 1. 首先对所有数字进行异或操作, 得到两个不同数字的异或结果
- \* 2. 找到异或结果中最低位的 1, 这个位置表示两个数字在该位不同
- \* 3. 根据这个位将数组分成两组, 分别进行异或操作得到两个结果
- \*
- \* 时间复杂度:  $O(n)$  - 遍历数组两次
- \* 空间复杂度:  $O(1)$  - 只使用常数个变量

```
 */
public class Code26_SingleNumberIII {
```

```
/**
```

```
* 找出只出现一次的两个数字
* @param nums 整数数组
* @return 只出现一次的两个数字
*/
```

```
public int[] singleNumber(int[] nums) {
    if (nums == null || nums.length < 2) {
        throw new IllegalArgumentException("数组长度必须至少为 2");
    }
```

```
// 第一步: 计算所有数字的异或结果
```

```
int xorResult = 0;
for (int num : nums) {
    xorResult ^= num;
}
```

```
// 第二步: 找到异或结果中最低位的 1
```

```
// 技巧: xorResult & (-xorResult) 可以快速找到最低位的 1
int lowestOneBit = xorResult & (-xorResult);
```

```
// 第三步: 根据最低位的 1 将数组分成两组
```

```
int[] result = new int[2];
for (int num : nums) {
    // 根据该位是否为 0 进行分组
    if ((num & lowestOneBit) == 0) {
        result[0] ^= num; // 第一组
    } else {
        result[1] ^= num; // 第二组
    }
}
```

```
}

    return result;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code26_SingleNumberIII solution = new Code26_SingleNumberIII();

    // 测试用例 1: 正常情况
    int[] nums1 = {1, 2, 1, 3, 2, 5};
    int[] result1 = solution.singleNumber(nums1);
    System.out.println("测试用例 1 结果: [" + result1[0] + ", " + result1[1] + "]");
    // 预期输出: [3, 5] 或 [5, 3]

    // 测试用例 2: 包含负数
    int[] nums2 = {-1, 0, -1, 2, 0, 3};
    int[] result2 = solution.singleNumber(nums2);
    System.out.println("测试用例 2 结果: [" + result2[0] + ", " + result2[1] + "]");
    // 预期输出: [2, 3] 或 [3, 2]

    // 测试用例 3: 边界情况
    int[] nums3 = {0, 1};
    int[] result3 = solution.singleNumber(nums3);
    System.out.println("测试用例 3 结果: [" + result3[0] + ", " + result3[1] + "]");
    // 预期输出: [0, 1] 或 [1, 0]

    // 复杂度分析
    System.out.println("\n==== 复杂度分析 ===");
    System.out.println("时间复杂度: O(n) - 遍历数组两次");
    System.out.println("空间复杂度: O(1) - 只使用常数个变量");

    // 工程化考量
    System.out.println("\n==== 工程化考量 ===");
    System.out.println("1. 输入验证: 检查数组长度");
    System.out.println("2. 边界处理: 处理负数情况");
    System.out.println("3. 性能优化: 使用位运算替代算术运算");
    System.out.println("4. 可读性: 添加详细注释说明算法原理");
}
```

文件: Code26\_SingleNumberIII.py

"""

只出现一次的数字 III

测试链接: <https://leetcode.cn/problems/single-number-iii/>

题目描述:

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。你可以按任意顺序返回答案。

解题思路:

1. 首先对所有数字进行异或操作, 得到两个不同数字的异或结果
2. 找到异或结果中最低位的 1, 这个位置表示两个数字在该位不同
3. 根据这个位将数组分成两组, 分别进行异或操作得到两个结果

时间复杂度:  $O(n)$  - 遍历数组两次

空间复杂度:  $O(1)$  - 只使用常数个变量

"""

```
from typing import List
```

```
class Code26_SingleNumberIII:
```

"""

只出现一次的数字 III 解决方案

"""

@staticmethod

```
def single_number(nums: List[int]) -> List[int]:
```

"""

找出只出现一次的两个数字

Args:

`nums`: 整数数组

Returns:

只出现一次的两个数字

Raises:

`ValueError`: 如果数组长度小于 2

"""

```
if len(nums) < 2:
```

    raise ValueError("数组长度必须至少为 2")

```
# 第一步：计算所有数字的异或结果
xor_result = 0
for num in nums:
    xor_result ^= num

# 第二步：找到异或结果中最低位的 1
# 技巧：xor_result & (-xor_result) 可以快速找到最低位的 1
# 注意：Python 中负数使用补码表示，需要特殊处理
lowest_one_bit = xor_result & (-xor_result)

# 第三步：根据最低位的 1 将数组分成两组
result = [0, 0]
for num in nums:
    # 根据该位是否为 0 进行分组
    if (num & lowest_one_bit) == 0:
        result[0] ^= num # 第一组
    else:
        result[1] ^= num # 第二组

return result
```

```
@staticmethod
def test():
    """测试方法"""
    # 测试用例 1：正常情况
    nums1 = [1, 2, 1, 3, 2, 5]
    result1 = Code26_SingleNumberIII.single_number(nums1)
    print(f"测试用例 1 结果: {result1}")
    # 预期输出: [3, 5] 或 [5, 3]

    # 测试用例 2：包含负数
    nums2 = [-1, 0, -1, 2, 0, 3]
    result2 = Code26_SingleNumberIII.single_number(nums2)
    print(f"测试用例 2 结果: {result2}")
    # 预期输出: [2, 3] 或 [3, 2]

    # 测试用例 3：边界情况
    nums3 = [0, 1]
    result3 = Code26_SingleNumberIII.single_number(nums3)
    print(f"测试用例 3 结果: {result3}")
    # 预期输出: [0, 1] 或 [1, 0]
```

```

# 复杂度分析
print("\n==== 复杂度分析 ===")
print("时间复杂度: O(n) - 遍历数组两次")
print("空间复杂度: O(1) - 只使用常数个变量")

# 工程化考量
print("\n==== 工程化考量 ===")
print("1. 输入验证: 检查数组长度")
print("2. 边界处理: 处理负数情况")
print("3. 性能优化: 使用位运算替代算术运算")
print("4. 可读性: 添加详细注释说明算法原理")
print("5. 异常处理: 使用异常处理输入错误")

# Python 特性考量
print("\n==== Python 特性考量 ===")
print("1. 类型注解: 使用 typing 模块提供类型提示")
print("2. 静态方法: 使用@staticmethod 装饰器")
print("3. 负数处理: Python 使用补码表示负数, 位运算需要特别注意")

if __name__ == "__main__":
    Code26_SingleNumberIII. test()

```

=====

文件: Code27\_ReverseBits.cpp

=====

```

/**
 * 颠倒二进制位
 * 测试链接: https://leetcode.cn/problems/reverse-bits/
 *
 * 题目描述:
 * 颠倒给定的 32 位无符号整数的二进制位。
 *
 * 解题思路:
 * 1. 逐位反转: 从最低位开始, 依次将每一位移动到对应的高位位置
 * 2. 分治反转: 使用分治思想, 先交换 16 位块, 再交换 8 位块, 依此类推
 * 3. 查表法: 对于 8 位进行预算算, 然后组合成 32 位
 *
 * 时间复杂度: O(1) - 固定 32 次操作
 * 空间复杂度: O(1) - 只使用常数个变量
 */
#include <iostream>
#include <vector>
```

```

#include <cstdint>
using namespace std;

class Code27_ReverseBits {
public:
    /**
     * 方法 1: 逐位反转
     * 时间复杂度: O(1) - 固定 32 次循环
     * 空间复杂度: O(1)
     */
    uint32_t reverseBits1(uint32_t n) {
        uint32_t result = 0;
        for (int i = 0; i < 32; i++) {
            // 取最低位
            uint32_t bit = n & 1;
            // 将最低位移动到对应的高位位置
            result = (result << 1) | bit;
            // 右移处理下一位
            n = n >> 1;
        }
        return result;
    }

    /**
     * 方法 2: 分治反转 (更高效)
     * 时间复杂度: O(1) - 固定 5 次操作
     * 空间复杂度: O(1)
     */
    uint32_t reverseBits2(uint32_t n) {
        // 分治思想: 先交换 16 位块, 再交换 8 位块, 依此类推
        n = (n >> 16) | (n << 16); // 交换 16 位块
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8); // 交换 8 位块
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4); // 交换 4 位块
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2); // 交换 2 位块
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1); // 交换 1 位块
        return n;
    }

    /**
     * 方法 3: 查表法 (最优解, 适合多次调用)
     * 时间复杂度: O(1) - 固定 4 次查表操作
     * 空间复杂度: O(256) - 预计算表
     */
}

```

```

static uint32_t reverseByte(uint8_t b) {
    // 反转 8 位字节
    uint32_t result = 0;
    for (int i = 0; i < 8; i++) {
        result = (result << 1) | (b & 1);
        b >>= 1;
    }
    return result;
}

uint32_t reverseBits3(uint32_t n) {
    static uint32_t table[256];
    static bool initialized = false;

    if (!initialized) {
        // 预计算 0-255 的 8 位反转结果
        for (int i = 0; i < 256; i++) {
            table[i] = reverseByte(static_cast<uint8_t>(i));
        }
        initialized = true;
    }

    // 将 32 位分成 4 个 8 位字节，分别反转后重新组合
    return (table[n & 0xff] << 24)           // 最低 8 位移到最高 8 位
        | (table[(n >> 8) & 0xff] << 16) | // 次低 8 位移到次高 8 位
        | (table[(n >> 16) & 0xff] << 8) | // 次高 8 位移到次低 8 位
        | (table[(n >> 24) & 0xff]);       // 最高 8 位移到最低 8 位
}

/***
 * 测试方法
 */
static void test() {
    Code27_ReverseBits solution;

    // 测试用例 1：正常情况
    uint32_t n1 = 43261596; // 二进制: 00000010100101000001111010011100
    uint32_t result1 = solution.reverseBits1(n1);
    uint32_t result2 = solution.reverseBits2(n1);
    uint32_t result3 = solution.reverseBits3(n1);
    cout << "测试用例 1 - 输入: " << n1 << endl;
    cout << "方法 1 结果: " << result1 << " (预期: 964176192)" << endl;
    cout << "方法 2 结果: " << result2 << " (预期: 964176192)" << endl;
}

```

```
cout << "方法 3 结果: " << result3 << " (预期: 964176192)" << endl;

// 测试用例 2: 边界情况 (全 0)
uint32_t n2 = 0;
uint32_t result4 = solution.reverseBits1(n2);
cout << "测试用例 2 - 输入: " << n2 << endl;
cout << "方法 1 结果: " << result4 << " (预期: 0)" << endl;

// 测试用例 3: 边界情况 (全 1)
uint32_t n3 = 0xFFFFFFFF; // 二进制全 1
uint32_t result5 = solution.reverseBits1(n3);
cout << "测试用例 3 - 输入: " << n3 << endl;
cout << "方法 1 结果: " << result5 << " (预期: 0xFFFFFFFF)" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 逐位反转:" << endl;
cout << " 时间复杂度: O(1) - 固定 32 次操作" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 分治反转:" << endl;
cout << " 时间复杂度: O(1) - 固定 5 次位操作" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - 查表法:" << endl;
cout << " 时间复杂度: O(1) - 固定 4 次查表操作" << endl;
cout << " 空间复杂度: O(256) - 预计算表" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 方法选择: " << endl;
cout << " - 单次调用: 方法 2 (分治) 最优" << endl;
cout << " - 多次调用: 方法 3 (查表) 最优" << endl;
cout << "2. 边界处理: 使用 uint32_t 确保无符号操作" << endl;
cout << "3. 性能优化: 避免不必要的循环" << endl;
cout << "4. 可读性: 添加详细注释说明位操作原理" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 类型安全: 使用 uint32_t 和 uint8_t 确保类型正确" << endl;
cout << "2. 静态表: 使用静态变量避免重复计算" << endl;
cout << "3. 初始化标志: 确保表只初始化一次" << endl;
cout << "4. 位操作: C++位操作与硬件指令紧密相关" << endl;
```

```

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 位掩码技巧：使用十六进制常量作为位掩码" << endl;
cout << "2. 分治思想：将大问题分解为小问题解决" << endl;
cout << "3. 查表优化：空间换时间，适合重复计算" << endl;
cout << "4. 无符号类型：避免符号扩展问题" << endl;
}

};

int main() {
    Code27_ReverseBits::test();
    return 0;
}
=====
```

文件: Code27\_ReverseBits.java

```

/**
 * 颠倒二进制位
 * 测试链接: https://leetcode.cn/problems/reverse-bits/
 *
 * 题目描述:
 * 颠倒给定的 32 位无符号整数的二进制位。
 *
 * 解题思路:
 * 1. 逐位反转: 从最低位开始, 依次将每一位移动到对应的高位位置
 * 2. 分治反转: 使用分治思想, 先交换 16 位块, 再交换 8 位块, 依此类推
 * 3. 查表法: 对于 8 位进行预算算, 然后组合成 32 位
 *
 * 时间复杂度: O(1) - 固定 32 次操作
 * 空间复杂度: O(1) - 只使用常数个变量
 */

public class Code27_ReverseBits {
```

```

/**
 * 方法 1: 逐位反转
 * 时间复杂度: O(1) - 固定 32 次循环
 * 空间复杂度: O(1)
 */
public int reverseBits1(int n) {
    int result = 0;
```

```

for (int i = 0; i < 32; i++) {
    // 取最低位
    int bit = n & 1;
    // 将最低位移动到对应的高位位置
    result = (result << 1) | bit;
    // 右移处理下一位
    n = n >>> 1; // 使用无符号右移
}
return result;
}

/***
 * 方法 2：分治反转（更高效）
 * 时间复杂度：O(1) - 固定 5 次操作
 * 空间复杂度：O(1)
 */
public int reverseBits2(int n) {
    // 分治思想：先交换 16 位块，再交换 8 位块，依此类推
    n = (n >>> 16) | (n << 16); // 交换 16 位块
    n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8); // 交换 8 位块
    n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4); // 交换 4 位块
    n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2); // 交换 2 位块
    n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1); // 交换 1 位块
    return n;
}

/***
 * 方法 3：查表法（最优解，适合多次调用）
 * 时间复杂度：O(1) - 固定 4 次查表操作
 * 空间复杂度：O(256) - 预计算表
 */
private static final int[] REVERSE_TABLE = new int[256];

static {
    // 预计算 0-255 的 8 位反转结果
    for (int i = 0; i < 256; i++) {
        REVERSE_TABLE[i] = reverseByte(i);
    }
}

private static int reverseByte(int b) {
    // 反转 8 位字节
    int result = 0;

```

```

        for (int i = 0; i < 8; i++) {
            result = (result << 1) | (b & 1);
            b >>>= 1;
        }
        return result;
    }

public int reverseBits3(int n) {
    // 将 32 位分成 4 个 8 位字节，分别反转后重新组合
    return (REVERSE_TABLE[n & 0xff] << 24) |           // 最低 8 位移到最高 8 位
           (REVERSE_TABLE[(n >>> 8) & 0xff] << 16) | // 次低 8 位移到次高 8 位
           (REVERSE_TABLE[(n >>> 16) & 0xff] << 8) | // 次高 8 位移到次低 8 位
           (REVERSE_TABLE[(n >>> 24) & 0xff]);          // 最高 8 位移到最低 8 位
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code27_ReverseBits solution = new Code27_ReverseBits();

    // 测试用例 1：正常情况
    int n1 = 43261596; // 二进制: 00000010100101000001111010011100
    int result1 = solution.reverseBits1(n1);
    int result2 = solution.reverseBits2(n1);
    int result3 = solution.reverseBits3(n1);
    System.out.println("测试用例 1 - 输入: " + n1);
    System.out.println("方法 1 结果: " + result1 + " (预期: 964176192)");
    System.out.println("方法 2 结果: " + result2 + " (预期: 964176192)");
    System.out.println("方法 3 结果: " + result3 + " (预期: 964176192)");

    // 测试用例 2：边界情况（全 0）
    int n2 = 0;
    int result4 = solution.reverseBits1(n2);
    System.out.println("测试用例 2 - 输入: " + n2);
    System.out.println("方法 1 结果: " + result4 + " (预期: 0)");

    // 测试用例 3：边界情况（全 1）
    int n3 = -1; // 二进制全 1
    int result5 = solution.reverseBits1(n3);
    System.out.println("测试用例 3 - 输入: " + n3);
    System.out.println("方法 1 结果: " + result5 + " (预期: -1)");
}

```

```

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 逐位反转:");
System.out.println(" 时间复杂度: O(1) - 固定 32 次操作");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 分治反转:");
System.out.println(" 时间复杂度: O(1) - 固定 5 次位操作");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 3 - 查表法:");
System.out.println(" 时间复杂度: O(1) - 固定 4 次查表操作");
System.out.println(" 空间复杂度: O(256) - 预计算表");

// 工程化考量
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 方法选择:");
System.out.println(" - 单次调用: 方法 2 (分治) 最优");
System.out.println(" - 多次调用: 方法 3 (查表) 最优");
System.out.println("2. 边界处理: 正确处理无符号整数");
System.out.println("3. 性能优化: 避免不必要的循环");
System.out.println("4. 可读性: 添加详细注释说明位操作原理");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. 位掩码技巧: 使用十六进制常量作为位掩码");
System.out.println("2. 分治思想: 将大问题分解为小问题解决");
System.out.println("3. 查表优化: 空间换时间, 适合重复计算");
System.out.println("4. 无符号右移: 使用>>>避免符号扩展问题");
}
}

```

文件: Code27\_ReverseBits.py

=====

颠倒二进制位

测试链接: <https://leetcode.cn/problems/reverse-bits/>

题目描述:

颠倒给定的 32 位无符号整数的二进制位。

解题思路:

1. 逐位反转: 从最低位开始, 依次将每一位移动到对应的高位位置
2. 分治反转: 使用分治思想, 先交换 16 位块, 再交换 8 位块, 依此类推
3. 查表法: 对于 8 位进行预算算, 然后组合成 32 位

时间复杂度:  $O(1)$  – 固定 32 次操作

空间复杂度:  $O(1)$  – 只使用常数个变量

"""

```
class Code27_ReverseBits:
```

"""

颠倒二进制位解决方案

"""

@staticmethod

```
def reverse_bits1(n: int) -> int:
```

"""

方法 1: 逐位反转

时间复杂度:  $O(1)$  – 固定 32 次循环

空间复杂度:  $O(1)$

Args:

n: 32 位无符号整数

Returns:

反转后的 32 位无符号整数

"""

```
result = 0
```

# Python 中整数可能超过 32 位, 需要限制为 32 位

```
n = n & 0xFFFFFFFF
```

```
for i in range(32):
```

# 取最低位

```
bit = n & 1
```

# 将最低位移动到对应的高位位置

```
result = (result << 1) | bit
```

# 右移处理下一位

```
n = n >> 1
```

```
return result
```

@staticmethod

```
def reverse_bits2(n: int) -> int:
```

```
"""
```

方法 2: 分治反转 (更高效)

时间复杂度: O(1) - 固定 5 次操作

空间复杂度: O(1)

Args:

n: 32 位无符号整数

Returns:

反转后的 32 位无符号整数

```
"""
```

# Python 中整数可能超过 32 位, 需要限制为 32 位

n = n & 0xFFFFFFFF

# 分治思想: 先交换 16 位块, 再交换 8 位块, 依此类推

n = ((n >> 16) | (n << 16)) & 0xFFFFFFFF # 交换 16 位块

n = (((n & 0xFF00FF00) >> 8) | ((n & 0x00FF00FF) << 8)) & 0xFFFFFFFF # 交换 8 位块

n = (((n & 0xF0F0F0F0) >> 4) | ((n & 0x0F0F0F0F) << 4)) & 0xFFFFFFFF # 交换 4 位块

n = (((n & 0xCCCCCCCC) >> 2) | ((n & 0x33333333) << 2)) & 0xFFFFFFFF # 交换 2 位块

n = (((n & 0xAAAAAAA) >> 1) | ((n & 0x55555555) << 1)) & 0xFFFFFFFF # 交换 1 位块

return n

```
@staticmethod
```

def reverse\_bits3(n: int) -> int:

```
"""
```

方法 3: 查表法 (最优解, 适合多次调用)

时间复杂度: O(1) - 固定 4 次查表操作

空间复杂度: O(256) - 预计算表

Args:

n: 32 位无符号整数

Returns:

反转后的 32 位无符号整数

```
"""
```

# 预计算 0-255 的 8 位反转结果

if not hasattr(Code27\_ReverseBits, '\_reverse\_table'):

Code27\_ReverseBits.\_reverse\_table = [

Code27\_ReverseBits.\_reverse\_byte(i) for i in range(256)

]

n = n & 0xFFFFFFFF

```
# 将 32 位分成 4 个 8 位字节，分别反转后重新组合
return ((Code27_ReverseBits._reverse_table[n & 0xFF] << 24) |
        (Code27_ReverseBits._reverse_table[(n >> 8) & 0xFF] << 16) |
        (Code27_ReverseBits._reverse_table[(n >> 16) & 0xFF] << 8) |
        (Code27_ReverseBits._reverse_table[(n >> 24) & 0xFF])) & 0xFFFFFFFF
```

```
@staticmethod
def _reverse_byte(b: int) -> int:
    """

```

反转 8 位字节

Args:

b: 8 位字节 (0-255)

Returns:

反转后的 8 位字节

```
"""
result = 0
for i in range(8):
    result = (result << 1) | (b & 1)
    b = b >> 1
return result
```

```
@staticmethod
```

```
def test():
    """
    测试方法
    # 测试用例 1: 正常情况
    n1 = 43261596 # 二进制: 00000010100101000001111010011100
    result1 = Code27_ReverseBits.reverse_bits1(n1)
    result2 = Code27_ReverseBits.reverse_bits2(n1)
    result3 = Code27_ReverseBits.reverse_bits3(n1)
    print(f"测试用例 1 - 输入: {n1}")
    print(f"方法 1 结果: {result1} (预期: 964176192)")
    print(f"方法 2 结果: {result2} (预期: 964176192)")
    print(f"方法 3 结果: {result3} (预期: 964176192)")

    # 测试用例 2: 边界情况 (全 0)
    n2 = 0
    result4 = Code27_ReverseBits.reverse_bits1(n2)
    print(f"测试用例 2 - 输入: {n2}")
    print(f"方法 1 结果: {result4} (预期: 0)")
```

```
# 测试用例 3: 边界情况 (全 1)
n3 = 0xFFFFFFFF # 二进制全 1
result5 = Code27_ReverseBits.reverse_bits1(n3)
print(f"测试用例 3 - 输入: {n3}")
print(f"方法 1 结果: {result5} (预期: 0xFFFFFFFF)")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 逐位反转:")
print("  时间复杂度: O(1) - 固定 32 次操作")
print("  空间复杂度: O(1)")

print("方法 2 - 分治反转:")
print("  时间复杂度: O(1) - 固定 5 次位操作")
print("  空间复杂度: O(1)")

print("方法 3 - 查表法:")
print("  时间复杂度: O(1) - 固定 4 次查表操作")
print("  空间复杂度: O(256) - 预计算表")

# 工程化考量
print("\n==== 工程化考量 ====")
print("1. 方法选择:")
print("  - 单次调用: 方法 2 (分治) 最优")
print("  - 多次调用: 方法 3 (查表) 最优")
print("2. 边界处理: Python 整数可能超过 32 位, 需要限制")
print("3. 性能优化: 避免不必要的循环")
print("4. 可读性: 添加详细注释说明位操作原理")

# Python 特性考量
print("\n==== Python 特性考量 ====")
print("1. 整数表示: Python 整数是动态大小的, 需要手动限制为 32 位")
print("2. 位操作: 使用& 0xFFFFFFFF 确保 32 位操作")
print("3. 类属性: 使用类属性存储预计算表")
print("4. 静态方法: 使用@staticmethod 装饰器")

# 算法技巧总结
print("\n==== 算法技巧总结 ====")
print("1. 位掩码技巧: 使用十六进制常量作为位掩码")
print("2. 分治思想: 将大问题分解为小问题解决")
print("3. 查表优化: 空间换时间, 适合重复计算")
print("4. 无符号处理: 确保正确处理无符号整数")
```

```
if __name__ == "__main__":
    Code27_ReverseBits. test()
```

=====

文件: Code28\_NumberOf1Bits. cpp

=====

```
/***
 * 位 1 的个数
 * 测试链接: https://leetcode.cn/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（以二进制串的形式），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）。
 *
 * 解题思路:
 * 1. 逐位检查：检查每一位是否为 1
 * 2. 快速方法：使用  $n \& (n-1)$  技巧快速消除最低位的 1
 * 3. 查表法：使用预计算的表来快速计算
 * 4. 分治法：使用分治思想并行计算
 *
 * 时间复杂度:  $O(1)$  - 最多 32 次操作
 * 空间复杂度:  $O(1)$  - 只使用常数个变量
 */
#include <iostream>
#include <cstdint>
#include <vector>
#include <bitset>
using namespace std;

class Code28_NumberOf1Bits {
public:
    /**
     * 方法 1: 逐位检查
     * 时间复杂度:  $O(k)$  - k 是 1 的个数，最坏情况  $O(32)$ 
     * 空间复杂度:  $O(1)$ 
     */
    int hammingWeight1(uint32_t n) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            // 检查最低位是否为 1
            if ((n & 1) == 1) {
                count++;
            }
            n = n >> 1;
        }
        return count;
    }
}
```

```

    }

    // 右移处理下一位
    n = n >> 1;
}

return count;
}

/***
* 方法 2: 快速方法 (最优解)
* 使用 n & (n-1) 技巧快速消除最低位的 1
* 时间复杂度: O(k) - k 是 1 的个数
* 空间复杂度: O(1)
*/
int hammingWeight2(uint32_t n) {
    int count = 0;
    while (n != 0) {
        // 每次操作消除最低位的 1
        n = n & (n - 1);
        count++;
    }
    return count;
}

/***
* 方法 3: 查表法 (适合多次调用)
* 时间复杂度: O(1) - 固定 4 次查表操作
* 空间复杂度: O(256) - 预计算表
*/
int hammingWeight3(uint32_t n) {
    static vector<int> bit_count_table(256, 0);
    static bool initialized = false;

    if (!initialized) {
        // 预计算 0-255 的 1 的个数
        for (int i = 0; i < 256; i++) {
            bit_count_table[i] = __builtin_popcount(i);
        }
        initialized = true;
    }

    // 将 32 位分成 4 个 8 位字节
    return bit_count_table[n & 0xff] +
           bit_count_table[(n >> 8) & 0xff] +

```

```

        bit_count_table[(n >> 16) & 0xff] +
        bit_count_table[(n >> 24) & 0xff];
    }

/***
 * 方法 4: 分治法 (并行计算)
 * 时间复杂度: O(log32) = O(1)
 * 空间复杂度: O(1)
 */
int hammingWeight4(uint32_t n) {
    // 分治思想: 先计算每 2 位的 1 的个数, 再计算每 4 位, 依此类推
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555); // 每 2 位
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333); // 每 4 位
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F); // 每 8 位
    n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF); // 每 16 位
    n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF); // 每 32 位
    return n;
}

/***
 * 方法 5: C++内置方法 (GCC/Clang)
 * 时间复杂度: O(1) - 使用硬件指令
 * 空间复杂度: O(1)
 */
int hammingWeight5(uint32_t n) {
    return __builtin_popcount(n);
}

/***
 * 方法 6: 标准库方法 (C++20)
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
int hammingWeight6(uint32_t n) {
    return __builtin_popcount(n); // C++20 标准库函数
}

/***
 * 测试方法
 */
static void test() {
    Code28_NumberOf1Bits solution;

```

```

// 测试用例 1: 正常情况
uint32_t n1 = 11; // 二进制: 1011
int result1 = solution.hammingWeight1(n1);
int result2 = solution.hammingWeight2(n1);
int result3 = solution.hammingWeight3(n1);
int result4 = solution.hammingWeight4(n1);
int result5 = solution.hammingWeight5(n1);
cout << "测试用例 1 - 输入: " << n1 << " (二进制: 1011)" << endl;
cout << "方法 1 结果: " << result1 << " (预期: 3)" << endl;
cout << "方法 2 结果: " << result2 << " (预期: 3)" << endl;
cout << "方法 3 结果: " << result3 << " (预期: 3)" << endl;
cout << "方法 4 结果: " << result4 << " (预期: 3)" << endl;
cout << "方法 5 结果: " << result5 << " (预期: 3)" << endl;

// 测试用例 2: 边界情况 (全 0)
uint32_t n2 = 0;
int result6 = solution.hammingWeight1(n2);
cout << "测试用例 2 - 输入: " << n2 << endl;
cout << "方法 1 结果: " << result6 << " (预期: 0)" << endl;

// 测试用例 3: 边界情况 (全 1)
uint32_t n3 = 0xFFFFFFFF; // 二进制全 1
int result7 = solution.hammingWeight1(n3);
cout << "测试用例 3 - 输入: " << n3 << endl;
cout << "方法 1 结果: " << result7 << " (预期: 32)" << endl;

// 性能对比测试
cout << "\n==== 性能分析 ===" << endl;
uint32_t testValue = 0x12345678; // 测试值

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 逐位检查:" << endl;
cout << " 时间复杂度: O(32) - 固定 32 次操作" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 快速方法:" << endl;
cout << " 时间复杂度: O(k) - k 是 1 的个数" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - 查表法:" << endl;
cout << " 时间复杂度: O(1) - 固定 4 次查表操作" << endl;
cout << " 空间复杂度: O(256)" << endl;

```

```
cout << "方法 4 - 分治法:" << endl;
cout << " 时间复杂度: O(log32) = O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 5 - 内置方法:" << endl;
cout << " 时间复杂度: O(1) - 硬件指令" << endl;
cout << " 空间复杂度: O(1)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 方法选择: " << endl;
cout << " - 实际工程: 方法 5/6 (内置方法) 最优" << endl;
cout << " - 面试场景: 方法 2 (快速方法) 最优" << endl;
cout << " - 多次调用: 方法 3 (查表法) 最优" << endl;
cout << "2. 边界处理: 使用 uint32_t 确保无符号操作" << endl;
cout << "3. 性能优化: 根据 1 的个数选择最优方法" << endl;
cout << "4. 可读性: 添加详细注释说明算法原理" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 类型安全: 使用 uint32_t 确保 32 位无符号整数" << endl;
cout << "2. 编译器内置: GCC/Clang 提供__builtin_popcount" << endl;
cout << "3. 标准库: C++20 提供 std::popcount" << endl;
cout << "4. 静态表: 使用静态变量避免重复计算" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. n & (n-1) 技巧: 快速消除最低位的 1" << endl;
cout << "2. 分治思想: 并行计算提高效率" << endl;
cout << "3. 查表优化: 空间换时间策略" << endl;
cout << "4. 硬件指令: 利用 CPU 内置功能" << endl;
cout << "5. 编译器优化: 利用编译器内置函数" << endl;
}

};

int main() {
    Code28_NumberOf1Bits::test();
    return 0;
}

=====
```

文件: Code28\_NumberOf1Bits.java

```
=====
/**  
 * 位 1 的个数  
 * 测试链接: https://leetcode.cn/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（以二进制串的形式），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）。  
 *  
 * 解题思路:  
 * 1. 逐位检查: 检查每一位是否为 1  
 * 2. 快速方法: 使用 n & (n-1) 技巧快速消除最低位的 1  
 * 3. 查表法: 使用预计算的表来快速计算  
 * 4. 分治法: 使用分治思想并行计算  
 *  
 * 时间复杂度: O(1) - 最多 32 次操作  
 * 空间复杂度: O(1) - 只使用常数个变量  
 */  
  
public class Code28_NumberOf1Bits {  
  
    /**  
     * 方法 1: 逐位检查  
     * 时间复杂度: O(k) - k 是 1 的个数, 最坏情况 O(32)  
     * 空间复杂度: O(1)  
     */  
  
    public int hammingWeight1(int n) {  
        int count = 0;  
        for (int i = 0; i < 32; i++) {  
            // 检查最低位是否为 1  
            if ((n & 1) == 1) {  
                count++;  
            }  
            // 无符号右移  
            n = n >>> 1;  
        }  
        return count;  
    }  
  
    /**  
     * 方法 2: 快速方法 (最优解)  
     * 使用 n & (n-1) 技巧快速消除最低位的 1  
     * 时间复杂度: O(k) - k 是 1 的个数  
     */
```

```

* 空间复杂度: O(1)
*/
public int hammingWeight2(int n) {
    int count = 0;
    while (n != 0) {
        // 每次操作消除最低位的1
        n = n & (n - 1);
        count++;
    }
    return count;
}

/***
 * 方法 3: 查表法 (适合多次调用)
 * 时间复杂度: O(1) - 固定 4 次查表操作
 * 空间复杂度: O(256) - 预计算表
 */
private static final int[] BIT_COUNT_TABLE = new int[256];

static {
    // 预计算 0-255 的 1 的个数
    for (int i = 0; i < 256; i++) {
        BIT_COUNT_TABLE[i] = Integer.bitCount(i);
    }
}

public int hammingWeight3(int n) {
    // 将 32 位分成 4 个 8 位字节
    return BIT_COUNT_TABLE[n & 0xff] +
        BIT_COUNT_TABLE[(n >>> 8) & 0xff] +
        BIT_COUNT_TABLE[(n >>> 16) & 0xff] +
        BIT_COUNT_TABLE[(n >>> 24) & 0xff];
}

/***
 * 方法 4: 分治法 (并行计算)
 * 时间复杂度: O(log32) = O(1)
 * 空间复杂度: O(1)
 */
public int hammingWeight4(int n) {
    // 分治思想: 先计算每 2 位的 1 的个数, 再计算每 4 位, 依此类推
    n = (n & 0x55555555) + ((n >>> 1) & 0x55555555); // 每 2 位
    n = (n & 0x33333333) + ((n >>> 2) & 0x33333333); // 每 4 位
}

```

```

n = (n & 0x0FOFOFOF) + ((n >>> 4) & 0x0FOFOFOF); // 每 8 位
n = (n & 0x00FF00FF) + ((n >>> 8) & 0x00FF00FF); // 每 16 位
n = (n & 0x0000FFFF) + ((n >>> 16) & 0x0000FFFF); // 每 32 位
return n;
}

/***
 * 方法 5: Java 内置方法 (实际工程中最优)
 * 时间复杂度: O(1) - 使用硬件指令
 * 空间复杂度: O(1)
 */
public int hammingWeight5(int n) {
    return Integer.bitCount(n);
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code28_NumberOf1Bits solution = new Code28_NumberOf1Bits();

    // 测试用例 1: 正常情况
    int n1 = 11; // 二进制: 1011
    int result1 = solution.hammingWeight1(n1);
    int result2 = solution.hammingWeight2(n1);
    int result3 = solution.hammingWeight3(n1);
    int result4 = solution.hammingWeight4(n1);
    int result5 = solution.hammingWeight5(n1);
    System.out.println("测试用例 1 - 输入: " + n1 + " (二进制: 1011)");
    System.out.println("方法 1 结果: " + result1 + " (预期: 3)");
    System.out.println("方法 2 结果: " + result2 + " (预期: 3)");
    System.out.println("方法 3 结果: " + result3 + " (预期: 3)");
    System.out.println("方法 4 结果: " + result4 + " (预期: 3)");
    System.out.println("方法 5 结果: " + result5 + " (预期: 3)");

    // 测试用例 2: 边界情况 (全 0)
    int n2 = 0;
    int result6 = solution.hammingWeight1(n2);
    System.out.println("测试用例 2 - 输入: " + n2);
    System.out.println("方法 1 结果: " + result6 + " (预期: 0)");

    // 测试用例 3: 边界情况 (全 1)
    int n3 = -1; // 二进制全 1
}

```

```
int result7 = solution.hammingWeight1(n3);
System.out.println("测试用例 3 - 输入: " + n3);
System.out.println("方法 1 结果: " + result7 + " (预期: 32)");

// 性能对比测试
System.out.println("\n==== 性能分析 ====");
long startTime, endTime;
int testValue = 0x12345678; // 测试值

startTime = System.nanoTime();
for (int i = 0; i < 1000000; i++) {
    solution.hammingWeight1(testValue);
}
endTime = System.nanoTime();
System.out.println("方法 1 耗时: " + (endTime - startTime) / 1000000 + " ms");

startTime = System.nanoTime();
for (int i = 0; i < 1000000; i++) {
    solution.hammingWeight2(testValue);
}
endTime = System.nanoTime();
System.out.println("方法 2 耗时: " + (endTime - startTime) / 1000000 + " ms");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 逐位检查:");
System.out.println(" 时间复杂度: O(32) - 固定 32 次操作");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 快速方法:");
System.out.println(" 时间复杂度: O(k) - k 是 1 的个数");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 3 - 查表法:");
System.out.println(" 时间复杂度: O(1) - 固定 4 次查表操作");
System.out.println(" 空间复杂度: O(256)");

System.out.println("方法 4 - 分治法:");
System.out.println(" 时间复杂度: O(log32) = O(1)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 5 - Java 内置:");
System.out.println(" 时间复杂度: O(1) - 硬件指令");
```

```

System.out.println(" 空间复杂度: O(1) ");

// 工程化考量
System.out.println("\n== 工程化考量 ==");
System.out.println("1. 方法选择:");
System.out.println(" - 实际工程: 方法 5 (内置方法) 最优");
System.out.println(" - 面试场景: 方法 2 (快速方法) 最优");
System.out.println(" - 多次调用: 方法 3 (查表法) 最优");
System.out.println("2. 边界处理: 正确处理无符号整数");
System.out.println("3. 性能优化: 根据 1 的个数选择最优方法");
System.out.println("4. 可读性: 添加详细注释说明算法原理");

// 算法技巧总结
System.out.println("\n== 算法技巧总结 ==");
System.out.println("1. n & (n-1) 技巧: 快速消除最低位的 1");
System.out.println("2. 分治思想: 并行计算提高效率");
System.out.println("3. 查表优化: 空间换时间策略");
System.out.println("4. 硬件指令: 利用 CPU 内置功能");
System.out.println("5. 无符号右移: 使用>>>避免符号问题");

}

}

```

=====

文件: Code28\_NumberOf1Bits.py

=====

"""

位 1 的个数

测试链接: <https://leetcode.cn/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（以二进制串的形式），返回其二进制表达式中数位为 '1' 的个数（也被称为汉明重量）。

解题思路:

1. 逐位检查: 检查每一位是否为 1
2. 快速方法: 使用  $n \& (n-1)$  技巧快速消除最低位的 1
3. 查表法: 使用预计算的表来快速计算
4. 分治法: 使用分治思想并行计算

时间复杂度:  $O(1)$  - 最多 32 次操作

空间复杂度:  $O(1)$  - 只使用常数个变量

"""

```
class Code28_NumberOf1Bits:  
    """  
    位 1 的个数解决方案  
    """  
  
    @staticmethod  
    def hamming_weight1(n: int) -> int:  
        """  
        方法 1：逐位检查  
        时间复杂度：O(k) - k 是 1 的个数，最坏情况 O(32)  
        空间复杂度：O(1)  
  
        Args:  
            n: 32 位无符号整数  
  
        Returns:  
            1 的个数  
        """  
  
        # Python 中整数可能超过 32 位，需要限制为 32 位  
        n = n & 0xFFFFFFFF  
        count = 0  
  
        for i in range(32):  
            # 检查最低位是否为 1  
            if (n & 1) == 1:  
                count += 1  
            # 右移处理下一位  
            n = n >> 1  
  
        return count  
  
    @staticmethod  
    def hamming_weight2(n: int) -> int:  
        """  
        方法 2：快速方法（最优解）  
        使用 n & (n-1) 技巧快速消除最低位的 1  
        时间复杂度：O(k) - k 是 1 的个数  
        空间复杂度：O(1)  
  
        Args:  
            n: 32 位无符号整数
```

Returns:

1 的个数

"""

# Python 中整数可能超过 32 位，需要限制为 32 位

n = n & 0xFFFFFFFF

count = 0

while n != 0:

# 每次操作消除最低位的 1

n = n & (n - 1)

count += 1

return count

@staticmethod

def hamming\_weight3(n: int) -> int:

"""

方法 3：查表法（适合多次调用）

时间复杂度：O(1) – 固定 4 次查表操作

空间复杂度：O(256) – 预计算表

Args:

n: 32 位无符号整数

Returns:

1 的个数

"""

# 预计算 0–255 的 1 的个数

if not hasattr(Code28\_NumberOf1Bits, '\_bit\_count\_table'):

Code28\_NumberOf1Bits.\_bit\_count\_table = [

bin(i).count('1') for i in range(256)

]

n = n & 0xFFFFFFFF

# 将 32 位分成 4 个 8 位字节

return (Code28\_NumberOf1Bits.\_bit\_count\_table[n & 0xFF] +

Code28\_NumberOf1Bits.\_bit\_count\_table[(n >> 8) & 0xFF] +

Code28\_NumberOf1Bits.\_bit\_count\_table[(n >> 16) & 0xFF] +

Code28\_NumberOf1Bits.\_bit\_count\_table[(n >> 24) & 0xFF])

@staticmethod

def hamming\_weight4(n: int) -> int:

```
"""
```

方法 4：分治法（并行计算）

时间复杂度： $O(\log 32) = O(1)$

空间复杂度： $O(1)$

Args:

n: 32 位无符号整数

Returns:

1 的个数

```
"""
```

```
n = n & 0xFFFFFFFF
```

# 分治思想：先计算每 2 位的 1 的个数，再计算每 4 位，依此类推

```
n = (n & 0x55555555) + ((n >> 1) & 0x55555555) # 每 2 位
```

```
n = (n & 0x33333333) + ((n >> 2) & 0x33333333) # 每 4 位
```

```
n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F) # 每 8 位
```

```
n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF) # 每 16 位
```

```
n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF) # 每 32 位
```

```
return n
```

```
@staticmethod
```

```
def hamming_weight5(n: int) -> int:
```

```
"""
```

方法 5：Python 内置方法

时间复杂度： $O(1)$

空间复杂度： $O(1)$

Args:

n: 32 位无符号整数

Returns:

1 的个数

```
"""
```

```
n = n & 0xFFFFFFFF
```

```
return bin(n).count('1')
```

```
@staticmethod
```

```
def test():
```

```
    """测试方法"""

```

# 测试用例 1：正常情况

```
n1 = 11 # 二进制: 1011
```

```
result1 = Code28_NumberOf1Bits.hamming_weight1(n1)
```

```
result2 = Code28_NumberOf1Bits.hamming_weight2(n1)
result3 = Code28_NumberOf1Bits.hamming_weight3(n1)
result4 = Code28_NumberOf1Bits.hamming_weight4(n1)
result5 = Code28_NumberOf1Bits.hamming_weight5(n1)
print(f"测试用例 1 - 输入: {n1} (二进制: 1011)")
print(f"方法 1 结果: {result1} (预期: 3)")
print(f"方法 2 结果: {result2} (预期: 3)")
print(f"方法 3 结果: {result3} (预期: 3)")
print(f"方法 4 结果: {result4} (预期: 3)")
print(f"方法 5 结果: {result5} (预期: 3)")
```

# 测试用例 2: 边界情况 (全 0)

```
n2 = 0
result6 = Code28_NumberOf1Bits.hamming_weight1(n2)
print(f"测试用例 2 - 输入: {n2}")
print(f"方法 1 结果: {result6} (预期: 0)")
```

# 测试用例 3: 边界情况 (全 1)

```
n3 = 0xFFFFFFFF # 二进制全 1
result7 = Code28_NumberOf1Bits.hamming_weight1(n3)
print(f"测试用例 3 - 输入: {n3}")
print(f"方法 1 结果: {result7} (预期: 32)")
```

# 复杂度分析

```
print("\n==== 复杂度分析 ===")
print("方法 1 - 逐位检查:")
print(" 时间复杂度: O(32) - 固定 32 次操作")
print(" 空间复杂度: O(1)")
```

```
print("方法 2 - 快速方法:")
```

```
print(" 时间复杂度: O(k) - k 是 1 的个数")
print(" 空间复杂度: O(1)")
```

```
print("方法 3 - 查表法:")
```

```
print(" 时间复杂度: O(1) - 固定 4 次查表操作")
print(" 空间复杂度: O(256)")
```

```
print("方法 4 - 分治法:")
```

```
print(" 时间复杂度: O(log32) = O(1)")
print(" 空间复杂度: O(1)")
```

```
print("方法 5 - Python 内置:")
print(" 时间复杂度: O(1)")
```

```

print(" 空间复杂度: O(1)")

# 工程化考量
print("\n==> 工程化考量 ==>")
print("1. 方法选择: ")
print(" - 实际工程: 方法 5 (内置方法) 最优")
print(" - 面试场景: 方法 2 (快速方法) 最优")
print(" - 多次调用: 方法 3 (查表法) 最优")
print("2. 边界处理: Python 整数可能超过 32 位, 需要限制")
print("3. 性能优化: 根据 1 的个数选择最优方法")
print("4. 可读性: 添加详细注释说明算法原理")

# Python 特性考量
print("\n==> Python 特性考量 ==>")
print("1. 整数表示: Python 整数是动态大小的, 需要手动限制为 32 位")
print("2. 内置函数: bin(n).count('1') 是最简洁的方法")
print("3. 类属性: 使用类属性存储预算算表")
print("4. 位操作: 使用& 0xFFFFFFFF 确保 32 位操作")

# 算法技巧总结
print("\n==> 算法技巧总结 ==>")
print("1. n & (n-1) 技巧: 快速消除最低位的 1")
print("2. 分治思想: 并行计算提高效率")
print("3. 查表优化: 空间换时间策略")
print("4. 内置函数: 利用语言内置功能")
print("5. 边界处理: 确保正确处理 32 位整数")

if __name__ == "__main__":
    Code28_NumberOf1Bits.test()

```

---

文件: Code29\_PowerOfTwo.cpp

---

```

/**
 * 2 的幂
 * 测试链接: https://leetcode.cn/problems/power-of-two/
 *
 * 题目描述:
 * 给你一个整数 n, 请你判断该整数是否是 2 的幂次方。如果是, 返回 true ; 否则, 返回 false 。
 * 如果存在一个整数 x 使得 n == 2^x , 则认为 n 是 2 的幂次方。
 *
 * 解题思路:

```

```

* 1. 循环除法: 不断除以 2 直到结果为 1
* 2. 位运算技巧: 利用 n & (n-1) == 0 的性质
* 3. 数学方法: 利用对数运算
* 4. 查表法: 预计算所有 2 的幂
*
* 时间复杂度: O(1) - 最多 32 次操作
* 空间复杂度: O(1) - 只使用常数个变量
*/
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

class Code29_PowerOfTwo {
public:
    /**
     * 方法 1: 循环除法
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     */
    bool isPowerOfTwo1(int n) {
        if (n <= 0) {
            return false;
        }

        while (n % 2 == 0) {
            n = n / 2;
        }

        return n == 1;
    }

    /**
     * 方法 2: 位运算技巧 (最优解)
     * 利用性质: 2 的幂的二进制表示中只有一个 1
     * n & (n-1) 可以消除最低位的 1, 如果结果为 0 则是 2 的幂
     * 时间复杂度: O(1)
     * 空间复杂度: O(1)
     */
    bool isPowerOfTwo2(int n) {
        return n > 0 && (n & (n - 1)) == 0;
    }
}

```

```

/***
 * 方法 3: 数学方法
 * 利用对数运算: log2(n) 应该是整数
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
bool isPowerOfTwo3(int n) {
    if (n <= 0) {
        return false;
    }

    double logResult = log2(n);
    // 检查是否为整数 (考虑浮点数精度)
    return abs(logResult - round(logResult)) < 1e-10;
}

/***
 * 方法 4: 查表法 (适合多次调用)
 * 预计算所有 32 位有符号整数范围内的 2 的幂
 * 时间复杂度: O(1)
 * 空间复杂度: O(32)
 */
bool isPowerOfTwo4(int n) {
    if (n <= 0) {
        return false;
    }

    // 预计算所有 2 的幂 (32 位有符号整数范围内)
    static vector<int> power_of_two_table;
    static bool initialized = false;

    if (!initialized) {
        for (int i = 0; i < 31; i++) { // 2^30 是最大正数 2 的幂
            power_of_two_table.push_back(1 << i);
        }
        initialized = true;
    }

    // 在预计算表中查找
    for (int power : power_of_two_table) {
        if (n == power) {
            return true;
        }
    }
}

```

```

    }

    return false;
}

/***
 * 方法 5：利用最大 2 的幂的约数性质
 * 在 32 位有符号整数范围内，最大的 2 的幂是  $2^{30} = 1073741824$ 
 * 如果 n 是 2 的幂，那么最大 2 的幂应该能被 n 整除
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 */

bool isPowerOfTwo5(int n) {
    return n > 0 && (1073741824 % n == 0);
}

/***
 * 方法 6：利用 bitset
 * 检查 1 的个数是否为 1
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 */

bool isPowerOfTwo6(int n) {
    if (n <= 0) return false;
    bitset<32> bits(n);
    return bits.count() == 1;
}

/***
 * 测试方法
 */
static void test() {
    Code29_PowerOfTwo solution;

    // 测试用例 1：正常情况（是 2 的幂）
    int n1 = 16;
    bool result1 = solution.isPowerOfTwo1(n1);
    bool result2 = solution.isPowerOfTwo2(n1);
    bool result3 = solution.isPowerOfTwo3(n1);
    bool result4 = solution.isPowerOfTwo4(n1);
    bool result5 = solution.isPowerOfTwo5(n1);
    bool result6 = solution.isPowerOfTwo6(n1);

    cout << "测试用例 1 - 输入：" << n1 << " (是 2 的幂)" << endl;
    cout << "方法 1 结果：" << result1 << " (预期: true)" << endl;
}

```

```
cout << "方法 2 结果: " << result2 << " (预期: true)" << endl;
cout << "方法 3 结果: " << result3 << " (预期: true)" << endl;
cout << "方法 4 结果: " << result4 << " (预期: true)" << endl;
cout << "方法 5 结果: " << result5 << " (预期: true)" << endl;
cout << "方法 6 结果: " << result6 << " (预期: true)" << endl;

// 测试用例 2: 正常情况 (不是 2 的幂)
int n2 = 18;
bool result7 = solution.isPowerOfTwo2(n2);
cout << "测试用例 2 - 输入: " << n2 << " (不是 2 的幂)" << endl;
cout << "方法 2 结果: " << result7 << " (预期: false)" << endl;

// 测试用例 3: 边界情况 (0)
int n3 = 0;
bool result8 = solution.isPowerOfTwo2(n3);
cout << "测试用例 3 - 输入: " << n3 << endl;
cout << "方法 2 结果: " << result8 << " (预期: false)" << endl;

// 测试用例 4: 边界情况 (负数)
int n4 = -8;
bool result9 = solution.isPowerOfTwo2(n4);
cout << "测试用例 4 - 输入: " << n4 << endl;
cout << "方法 2 结果: " << result9 << " (预期: false)" << endl;

// 测试用例 5: 边界情况 (1)
int n5 = 1;
bool result10 = solution.isPowerOfTwo2(n5);
cout << "测试用例 5 - 输入: " << n5 << endl;
cout << "方法 2 结果: " << result10 << " (预期: true)" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 循环除法:" << endl;
cout << " 时间复杂度: O(log n)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 位运算技巧:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - 数学方法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;
```

```
cout << "方法 4 - 查表法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(32)" << endl;

cout << "方法 5 - 约数性质:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 6 - bitset 方法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 方法选择:" << endl;
cout << " - 实际工程: 方法 2 (位运算) 最优" << endl;
cout << " - 面试场景: 方法 2 (位运算) 最优" << endl;
cout << " - 多次调用: 方法 4 (查表法) 最优" << endl;
cout << "2. 边界处理: 必须检查 n>0" << endl;
cout << "3. 性能优化: 位运算最快, 只需一次操作" << endl;
cout << "4. 可读性: 方法 2 代码简洁易懂" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 类型安全: 使用 int 类型" << endl;
cout << "2. 标准库: 使用 bitset 进行位操作" << endl;
cout << "3. 数学函数: 使用 log2 和 abs" << endl;
cout << "4. 静态变量: 使用静态表避免重复计算" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. n & (n-1) 技巧: 判断是否只有一个 1" << endl;
cout << "2. 位运算性质: 2 的幂的二进制特性" << endl;
cout << "3. 查表优化: 预计算所有可能值" << endl;
cout << "4. 数学性质: 利用对数运算" << endl;
cout << "5. 边界情况: 0 和负数都不是 2 的幂" << endl;
}

};

int main() {
    Code29_PowerOfTwo::test();
    return 0;
}
```

```
}
```

```
=====
```

文件: Code29\_PowerOfTwo.java

```
=====
/**  
 * 2 的幂  
 * 测试链接: https://leetcode.cn/problems/power-of-two/  
 *  
 * 题目描述:  
 * 给你一个整数 n，请你判断该整数是否是 2 的幂次方。如果是，返回 true；否则，返回 false。  
 * 如果存在一个整数 x 使得 n == 2^x，则认为 n 是 2 的幂次方。  
 *  
 * 解题思路:  
 * 1. 循环除法：不断除以 2 直到结果为 1  
 * 2. 位运算技巧：利用 n & (n-1) == 0 的性质  
 * 3. 数学方法：利用对数运算  
 * 4. 查表法：預计算所有 2 的幂  
 *  
 * 时间复杂度: O(1) - 最多 32 次操作  
 * 空间复杂度: O(1) - 只使用常数个变量  
 */
```

```
public class Code29_PowerOfTwo {
```

```
    /**  
     * 方法 1：循环除法  
     * 时间复杂度: O(log n)  
     * 空间复杂度: O(1)  
     */
```

```
    public boolean isPowerOfTwo1(int n) {
```

```
        if (n <= 0) {  
            return false;  
        }
```

```
        while (n % 2 == 0) {  
            n = n / 2;  
        }
```

```
        return n == 1;
```

```
}
```

```
    /**
```

```
* 方法 2: 位运算技巧（最优解）
* 利用性质: 2 的幂的二进制表示中只有一个 1
*  $n \& (n-1)$  可以消除最低位的 1, 如果结果为 0 则是 2 的幂
* 时间复杂度:  $O(1)$ 
* 空间复杂度:  $O(1)$ 
*/
public boolean isPowerOfTwo2(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

```
/***
* 方法 3: 数学方法
* 利用对数运算:  $\log_2(n)$  应该是整数
* 时间复杂度:  $O(1)$ 
* 空间复杂度:  $O(1)$ 
*/
public boolean isPowerOfTwo3(int n) {
    if (n <= 0) {
        return false;
    }

    double logResult = Math.log(n) / Math.log(2);
    // 检查是否为整数 (考虑浮点数精度)
    return Math.abs(logResult - Math.round(logResult)) < 1e-10;
}
```

```
/***
* 方法 4: 查表法 (适合多次调用)
* 预计算所有 32 位有符号整数范围内的 2 的幂
* 时间复杂度:  $O(1)$ 
* 空间复杂度:  $O(32)$ 
*/
private static final int[] POWER_OF_TWO_TABLE;
```

```
static {
    // 预计算所有 2 的幂 (32 位有符号整数范围内)
    POWER_OF_TWO_TABLE = new int[32];
    for (int i = 0; i < 32; i++) {
        POWER_OF_TWO_TABLE[i] = 1 << i;
    }
}
```

```
public boolean isPowerOfTwo4(int n) {
```

```

    if (n <= 0) {
        return false;
    }

    // 在预算算表中查找
    for (int power : POWER_OF_TWO_TABLE) {
        if (n == power) {
            return true;
        }
    }
    return false;
}

/***
 * 方法 5：利用最大 2 的幂的约数性质
 * 在 32 位有符号整数范围内，最大的 2 的幂是  $2^{30} = 1073741824$ 
 * 如果 n 是 2 的幂，那么最大 2 的幂应该能被 n 整除
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 */
public boolean isPowerOfTwo5(int n) {
    return n > 0 && (1073741824 % n == 0);
}

/***
 * 方法 6：利用 Integer.bitCount 方法
 * 检查 1 的个数是否为 1
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 */
public boolean isPowerOfTwo6(int n) {
    return n > 0 && Integer.bitCount(n) == 1;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code29_PowerOfTwo solution = new Code29_PowerOfTwo();

    // 测试用例 1：正常情况（是 2 的幂）
    int n1 = 16;
    boolean result1 = solution.isPowerOfTwo1(n1);
}

```

```

boolean result2 = solution.isPowerOfTwo2(n1);
boolean result3 = solution.isPowerOfTwo3(n1);
boolean result4 = solution.isPowerOfTwo4(n1);
boolean result5 = solution.isPowerOfTwo5(n1);
boolean result6 = solution.isPowerOfTwo6(n1);
System.out.println("测试用例 1 - 输入: " + n1 + " (是 2 的幂)");
System.out.println("方法 1 结果: " + result1 + " (预期: true)");
System.out.println("方法 2 结果: " + result2 + " (预期: true)");
System.out.println("方法 3 结果: " + result3 + " (预期: true)");
System.out.println("方法 4 结果: " + result4 + " (预期: true)");
System.out.println("方法 5 结果: " + result5 + " (预期: true)");
System.out.println("方法 6 结果: " + result6 + " (预期: true)");

// 测试用例 2: 正常情况 (不是 2 的幂)
int n2 = 18;
boolean result7 = solution.isPowerOfTwo2(n2);
System.out.println("测试用例 2 - 输入: " + n2 + " (不是 2 的幂)");
System.out.println("方法 2 结果: " + result7 + " (预期: false)");

// 测试用例 3: 边界情况 (0)
int n3 = 0;
boolean result8 = solution.isPowerOfTwo2(n3);
System.out.println("测试用例 3 - 输入: " + n3);
System.out.println("方法 2 结果: " + result8 + " (预期: false)");

// 测试用例 4: 边界情况 (负数)
int n4 = -8;
boolean result9 = solution.isPowerOfTwo2(n4);
System.out.println("测试用例 4 - 输入: " + n4);
System.out.println("方法 2 结果: " + result9 + " (预期: false)");

// 测试用例 5: 边界情况 (1)
int n5 = 1;
boolean result10 = solution.isPowerOfTwo2(n5);
System.out.println("测试用例 5 - 输入: " + n5);
System.out.println("方法 2 结果: " + result10 + " (预期: true)");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 循环除法:");
System.out.println(" 时间复杂度: O(log n)");
System.out.println(" 空间复杂度: O(1)");

```

```
System.out.println("方法 2 - 位运算技巧:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 3 - 数学方法:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 4 - 查表法:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(32)");

System.out.println("方法 5 - 约数性质:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 6 - bitCount 方法:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

// 工程化考量
System.out.println("\n==== 工程化考量 ===");
System.out.println("1. 方法选择:");
System.out.println(" - 实际工程: 方法 2 (位运算) 最优");
System.out.println(" - 面试场景: 方法 2 (位运算) 最优");
System.out.println(" - 多次调用: 方法 4 (查表法) 最优");
System.out.println("2. 边界处理: 必须检查 n>0");
System.out.println("3. 性能优化: 位运算最快, 只需一次操作");
System.out.println("4. 可读性: 方法 2 代码简洁易懂");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ===");
System.out.println("1. n & (n-1) 技巧: 判断是否只有一个 1");
System.out.println("2. 位运算性质: 2 的幂的二进制特性");
System.out.println("3. 查表优化: 预计算所有可能值");
System.out.println("4. 数学性质: 利用对数运算");
System.out.println("5. 边界情况: 0 和负数都不是 2 的幂");

// 应用场景
System.out.println("\n==== 应用场景 ===");
System.out.println("1. 内存分配: 判断是否是 2 的幂以便对齐");
System.out.println("2. 哈希表: 选择合适的大小 (2 的幂)");
System.out.println("3. 位图操作: 处理 2 的幂相关的位操作");
```

```
        System.out.println("4. 性能优化：利用 2 的幂的数学特性");  
    }  
}
```

=====

文件: Code29\_PowerOfTwo.py

=====

"""

2 的幂

测试链接: <https://leetcode.cn/problems/power-of-two/>

题目描述:

给你一个整数  $n$ ，请你判断该整数是否是 2 的幂次方。如果是，返回 `true`；否则，返回 `false`。  
如果存在一个整数  $x$  使得  $n == 2^x$ ，则认为  $n$  是 2 的幂次方。

解题思路:

1. 循环除法：不断除以 2 直到结果为 1
2. 位运算技巧：利用  $n \& (n-1) == 0$  的性质
3. 数学方法：利用对数运算
4. 查表法：预计算所有 2 的幂

时间复杂度:  $O(1)$  – 最多 32 次操作

空间复杂度:  $O(1)$  – 只使用常数个变量

"""

import math

```
class Code29_PowerOfTwo:
```

"""

2 的幂解决方案

"""

@staticmethod

```
def is_power_of_two1(n: int) -> bool:
```

"""

方法 1：循环除法

时间复杂度:  $O(\log n)$

空间复杂度:  $O(1)$

Args:

    n: 整数

Returns:

是否是 2 的幂

"""

```
if n <= 0:  
    return False
```

```
while n % 2 == 0:  
    n = n // 2
```

```
return n == 1
```

@staticmethod

```
def is_power_of_two2(n: int) -> bool:
```

"""

方法 2：位运算技巧（最优解）

利用性质：2 的幂的二进制表示中只有一个 1

$n \& (n-1)$  可以消除最低位的 1，如果结果为 0 则是 2 的幂

时间复杂度：O(1)

空间复杂度：O(1)

Args:

n: 整数

Returns:

是否是 2 的幂

"""

```
return n > 0 and (n & (n - 1)) == 0
```

@staticmethod

```
def is_power_of_two3(n: int) -> bool:
```

"""

方法 3：数学方法

利用对数运算： $\log_2(n)$  应该是整数

时间复杂度：O(1)

空间复杂度：O(1)

Args:

n: 整数

Returns:

是否是 2 的幂

"""

```
if n <= 0:  
    return False
```

```
log_result = math.log2(n)
# 检查是否为整数（考虑浮点数精度）
return abs(log_result - round(log_result)) < 1e-10
```

```
@staticmethod
def is_power_of_two4(n: int) -> bool:
    """
```

方法 4：查表法（适合多次调用）

预计算所有 32 位有符号整数范围内的 2 的幂

时间复杂度：O(1)

空间复杂度：O(32)

Args:

n: 整数

Returns:

是否是 2 的幂

```
"""
```

```
if n <= 0:
```

```
    return False
```

# 预计算所有 2 的幂（32 位有符号整数范围内）

```
if not hasattr(Code29_PowerOfTwo, '_power_of_two_table'):
```

```
    Code29_PowerOfTwo._power_of_two_table = {1 << i for i in range(31)} # 2^30 是最大正数
```

2 的幂

```
return n in Code29_PowerOfTwo._power_of_two_table
```

```
@staticmethod
```

```
def is_power_of_two5(n: int) -> bool:
    """
```

方法 5：利用最大 2 的幂的约数性质

在 32 位有符号整数范围内，最大的 2 的幂是  $2^{30} = 1073741824$

如果 n 是 2 的幂，那么最大 2 的幂应该能被 n 整除

时间复杂度：O(1)

空间复杂度：O(1)

Args:

n: 整数

Returns:

是否是 2 的幂

```
"""
return n > 0 and (1073741824 % n == 0)
```

```
@staticmethod
def is_power_of_two6(n: int) -> bool:
    """
```

方法 6：利用 bin 函数

检查二进制表示中 1 的个数是否为 1

时间复杂度：O(1)

空间复杂度：O(1)

Args:

n: 整数

Returns:

是否是 2 的幂

```
"""
return n > 0 and bin(n).count('1') == 1
```

```
@staticmethod
```

```
def test():
```

```
    """测试方法"""
    # 测试用例 1：正常情况（是 2 的幂）
```

```
    n1 = 16
```

```
    result1 = Code29_PowerOfTwo.is_power_of_two1(n1)
```

```
    result2 = Code29_PowerOfTwo.is_power_of_two2(n1)
```

```
    result3 = Code29_PowerOfTwo.is_power_of_two3(n1)
```

```
    result4 = Code29_PowerOfTwo.is_power_of_two4(n1)
```

```
    result5 = Code29_PowerOfTwo.is_power_of_two5(n1)
```

```
    result6 = Code29_PowerOfTwo.is_power_of_two6(n1)
```

```
    print(f"测试用例 1 - 输入: {n1} (是 2 的幂)")
```

```
    print(f"方法 1 结果: {result1} (预期: True)")
```

```
    print(f"方法 2 结果: {result2} (预期: True)")
```

```
    print(f"方法 3 结果: {result3} (预期: True)")
```

```
    print(f"方法 4 结果: {result4} (预期: True)")
```

```
    print(f"方法 5 结果: {result5} (预期: True)")
```

```
    print(f"方法 6 结果: {result6} (预期: True)")
```

```
# 测试用例 2：正常情况（不是 2 的幂）
```

```
n2 = 18
```

```
result7 = Code29_PowerOfTwo.is_power_of_two2(n2)
```

```
print(f"测试用例 2 - 输入: {n2} (不是 2 的幂)")
```

```
print(f"方法 2 结果: {result7} (预期: False)")
```

```
# 测试用例 3: 边界情况 (0)
n3 = 0
result8 = Code29_PowerOfTwo.is_power_of_two2(n3)
print(f"测试用例 3 - 输入: {n3}")
print(f"方法 2 结果: {result8} (预期: False)")

# 测试用例 4: 边界情况 (负数)
n4 = -8
result9 = Code29_PowerOfTwo.is_power_of_two2(n4)
print(f"测试用例 4 - 输入: {n4}")
print(f"方法 2 结果: {result9} (预期: False)")

# 测试用例 5: 边界情况 (1)
n5 = 1
result10 = Code29_PowerOfTwo.is_power_of_two2(n5)
print(f"测试用例 5 - 输入: {n5}")
print(f"方法 2 结果: {result10} (预期: True)")

# 复杂度分析
print("\n==== 复杂度分析 ===")
print("方法 1 - 循环除法:")
print("  时间复杂度:  $O(\log n)$ ")
print("  空间复杂度:  $O(1)$ ")

print("方法 2 - 位运算技巧:")
print("  时间复杂度:  $O(1)$ ")
print("  空间复杂度:  $O(1)$ ")

print("方法 3 - 数学方法:")
print("  时间复杂度:  $O(1)$ ")
print("  空间复杂度:  $O(1)$ ")

print("方法 4 - 查表法:")
print("  时间复杂度:  $O(1)$ ")
print("  空间复杂度:  $O(32)$ ")

print("方法 5 - 约数性质:")
print("  时间复杂度:  $O(1)$ ")
print("  空间复杂度:  $O(1)$ ")

print("方法 6 - bin 函数方法:")
print("  时间复杂度:  $O(1)$ )
```

```

print(" 空间复杂度: O(1)")

# 工程化考量
print("\n==> 工程化考量 ===")
print("1. 方法选择: ")
print(" - 实际工程: 方法 2 (位运算) 最优")
print(" - 面试场景: 方法 2 (位运算) 最优")
print(" - 多次调用: 方法 4 (查表法) 最优")
print("2. 边界处理: 必须检查 n>0")
print("3. 性能优化: 位运算最快, 只需一次操作")
print("4. 可读性: 方法 2 代码简洁易懂")

# Python 特性考量
print("\n==> Python 特性考量 ===")
print("1. 整数除法: 使用//进行整数除法")
print("2. 位运算: Python 支持标准的位运算符")
print("3. 数学函数: 使用 math.log2 进行对数运算")
print("4. 集合操作: 使用集合进行快速查找")

# 算法技巧总结
print("\n==> 算法技巧总结 ===")
print("1. n & (n-1) 技巧: 判断是否只有一个 1")
print("2. 位运算性质: 2 的幂的二进制特性")
print("3. 查表优化: 预计算所有可能值")
print("4. 数学性质: 利用对数运算")
print("5. 边界情况: 0 和负数都不是 2 的幂")

if __name__ == "__main__":
    Code29_PowerOfTwo.test()

```

=====

文件: Code30\_BitwiseANDofNumbersRange.cpp

=====

```

/**
 * 数字范围按位与
 * 测试链接: https://leetcode.cn/problems/bitwise-and-of-numbers-range/
 *
 * 题目描述:
 * 给你两个整数 left 和 right , 表示区间 [left, right] , 返回此区间内所有数字按位与的结果 (包含
 * left 、right 端点)。
 *
 * 解题思路:

```

```
* 1. 暴力法：遍历区间内所有数字进行按位与操作
* 2. 位运算技巧：找到 left 和 right 的公共前缀
* 3. 位移法：不断右移直到 left 等于 right，然后左移恢复
* 4. Brian Kernighan 算法：利用 n & (n-1) 消除最低位的 1
*
* 时间复杂度：O(1) - 最多 32 次操作
* 空间复杂度：O(1) - 只使用常数个变量
*/
```

```
#include <iostream>
```

```
#include <climits>
```

```
using namespace std;
```

```
class Code30_BitwiseANDofNumbersRange {
```

```
public:
```

```
/**
```

```
* 方法 1：暴力法（不推荐，会超时）
```

```
* 时间复杂度：O(right - left)
```

```
* 空间复杂度：O(1)
```

```
*/
```

```
int rangeBitwiseAnd1(int left, int right) {
```

```
    int result = left;
```

```
    for (int i = left + 1; i <= right; i++) {
```

```
        result &= i;
```

```
        if (result == 0) {
```

```
            break; // 提前终止优化
```

```
}
```

```
}
```

```
    return result;
```

```
}
```

```
/**
```

```
* 方法 2：位运算技巧（最优解）
```

```
* 找到 left 和 right 的公共前缀
```

```
* 时间复杂度：O(1)
```

```
* 空间复杂度：O(1)
```

```
*/
```

```
int rangeBitwiseAnd2(int left, int right) {
```

```
    int shift = 0;
```

```
    // 找到公共前缀
```

```
    while (left < right) {
```

```
        left >>= 1;
```

```
        right >>= 1;
```

```
        shift++;
```

```

    }

    return left << shift;
}

/***
 * 方法 3: Brian Kernighan 算法
 * 利用 n & (n-1) 消除最低位的 1
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */

int rangeBitwiseAnd3(int left, int right) {
    while (left < right) {
        // 消除 right 最低位的 1
        right = right & (right - 1);
    }
    return right;
}

/***
 * 方法 4: 位移法的另一种实现
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */

int rangeBitwiseAnd4(int left, int right) {
    if (left == right) {
        return left;
    }

    // 计算 left 和 right 的最高不同位
    int xor_val = left ^ right;
    int mask = 1 << 31; // 从最高位开始找

    // 找到最高不同位
    while (mask > 0 && (xor_val & mask) == 0) {
        mask >>= 1;
    }

    // 创建掩码, 将不同位及之后的位都置为 0
    mask = (mask << 1) - 1;
    mask = ~mask;

    return left & mask;
}

```

```
/**  
 * 方法 5：利用内置函数  
 * 使用 GCC 内置函数找到最高位  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 */  
  
int rangeBitwiseAnd5(int left, int right) {  
    if (left == right) return left;  
  
    // 找到最高不同位  
    int xor_val = left ^ right;  
    int highest_bit = 31 - __builtin_clz(xor_val);  
  
    // 创建掩码  
    int mask = ~((1 << (highest_bit + 1)) - 1);  
    return left & mask;  
}  
  
/**  
 * 测试方法  
 */  
  
static void test() {  
    Code30_BitwiseANDofNumbersRange solution;  
  
    // 测试用例 1: 正常情况  
    int left1 = 5, right1 = 7;  
    int result1 = solution.rangeBitwiseAnd1(left1, right1);  
    int result2 = solution.rangeBitwiseAnd2(left1, right1);  
    int result3 = solution.rangeBitwiseAnd3(left1, right1);  
    int result4 = solution.rangeBitwiseAnd4(left1, right1);  
    int result5 = solution.rangeBitwiseAnd5(left1, right1);  
    cout << "测试用例 1 - 输入: [" << left1 << ", " << right1 << "]" << endl;  
    cout << "方法 1 结果: " << result1 << " (预期: 4)" << endl;  
    cout << "方法 2 结果: " << result2 << " (预期: 4)" << endl;  
    cout << "方法 3 结果: " << result3 << " (预期: 4)" << endl;  
    cout << "方法 4 结果: " << result4 << " (预期: 4)" << endl;  
    cout << "方法 5 结果: " << result5 << " (预期: 4)" << endl;  
  
    // 测试用例 2: 边界情况 (相同数字)  
    int left2 = 10, right2 = 10;  
    int result6 = solution.rangeBitwiseAnd2(left2, right2);  
    cout << "测试用例 2 - 输入: [" << left2 << ", " << right2 << "]" << endl;
```

```
cout << "方法 2 结果: " << result6 << " (预期: 10)" << endl;

// 测试用例 3: 大范围情况
int left3 = 1, right3 = INT_MAX;
int result7 = solution.rangeBitwiseAnd2(left3, right3);
cout << "测试用例 3 - 输入: [" << left3 << ", " << right3 << "]" << endl;
cout << "方法 2 结果: " << result7 << " (预期: 0)" << endl;

// 测试用例 4: 包含 2 的幂
int left4 = 8, right4 = 15;
int result8 = solution.rangeBitwiseAnd2(left4, right4);
cout << "测试用例 4 - 输入: [" << left4 << ", " << right4 << "]" << endl;
cout << "方法 2 结果: " << result8 << " (预期: 8)" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 暴力法:" << endl;
cout << " 时间复杂度: O(n) - n = right - left" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 位运算技巧:" << endl;
cout << " 时间复杂度: O(1) - 最多 32 次位移" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - Brian Kernighan 算法:" << endl;
cout << " 时间复杂度: O(1) - 最多 32 次操作" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 4 - 位移法变种:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 5 - 内置函数法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 方法选择: " << endl;
cout << " - 实际工程: 方法 2 (位运算技巧) 最优" << endl;
cout << " - 面试场景: 方法 2 (位运算技巧) 最优" << endl;
cout << "2. 性能优化: 避免暴力法, 使用位运算" << endl;
cout << "3. 边界处理: 处理 left 等于 right 的情况" << endl;
```

```

cout << "4. 可读性: 方法 2 代码简洁易懂" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 内置函数: 使用 GCC 内置函数提高性能" << endl;
cout << "2. 类型安全: 使用 int 类型" << endl;
cout << "3. 常量定义: 使用 INT_MAX 表示最大值" << endl;
cout << "4. 位操作: C++位操作与硬件紧密相关" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 公共前缀: 区间按位与的结果就是公共前缀" << endl;
cout << "2. 位移操作: 通过右移找到公共前缀" << endl;
cout << "3. Brian Kernighan 技巧: 消除最低位的 1" << endl;
cout << "4. 最高位掩码: 使用位运算创建掩码" << endl;
cout << "5. 提前终止: 当结果为 0 时可以提前终止" << endl;
}

};

int main() {
    Code30_BitwiseANDofNumbersRange::test();
    return 0;
}

```

=====

文件: Code30\_BitwiseANDofNumbersRange.java

=====

```

/**
 * 数字范围按位与
 * 测试链接: https://leetcode.cn/problems/bitwise-and-of-numbers-range/
 *
 * 题目描述:
 * 给你两个整数 left 和 right , 表示区间 [left, right] , 返回此区间内所有数字按位与的结果（包含
left 、right 端点）。
 *
 * 解题思路:
 * 1. 暴力法: 遍历区间内所有数字进行按位与操作
 * 2. 位运算技巧: 找到 left 和 right 的公共前缀
 * 3. 位移法: 不断右移直到 left 等于 right, 然后左移恢复
 * 4. Brian Kernighan 算法: 利用 n & (n-1) 消除最低位的 1
 *
 * 时间复杂度: O(1) - 最多 32 次操作

```

```
* 空间复杂度: O(1) - 只使用常数个变量
*/
public class Code30_BitwiseANDofNumbersRange {
```

```
/***
 * 方法 1: 暴力法 (不推荐, 会超时)
 * 时间复杂度: O(right - left)
 * 空间复杂度: O(1)
 */
```

```
public int rangeBitwiseAnd1(int left, int right) {
    int result = left;
    for (int i = left + 1; i <= right; i++) {
        result &= i;
        if (result == 0) {
            break; // 提前终止优化
        }
    }
    return result;
}
```

```
/***
 * 方法 2: 位运算技巧 (最优解)
 * 找到 left 和 right 的公共前缀
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
```

```
public int rangeBitwiseAnd2(int left, int right) {
    int shift = 0;
    // 找到公共前缀
    while (left < right) {
        left >>= 1;
        right >>= 1;
        shift++;
    }
    return left << shift;
}
```

```
/***
 * 方法 3: Brian Kernighan 算法
 * 利用 n & (n-1) 消除最低位的 1
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
```

```
public int rangeBitwiseAnd3(int left, int right) {
    while (left < right) {
        // 消除 right 最低位的 1
        right = right & (right - 1);
    }
    return right;
}

/***
 * 方法 4：位移法的另一种实现
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
public int rangeBitwiseAnd4(int left, int right) {
    if (left == right) {
        return left;
    }

    // 计算 left 和 right 的最高不同位
    int xor = left ^ right;
    int mask = Integer.highestOneBit(xor);

    // 创建掩码，将不同位及之后的位都置为 0
    mask = (mask << 1) - 1;
    mask = ~mask;

    return left & mask;
}

/***
 * 方法 5：数学方法
 * 利用 2 的幂的性质
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
public int rangeBitwiseAnd5(int left, int right) {
    // 如果区间跨越了 2 的幂的边界，结果为 0
    if ((left & (left - 1)) > 0 || (right & (right - 1)) > 0) {
        int nextPower = Integer.highestOneBit(left) << 1;
        if (right >= nextPower) {
            return 0;
        }
    }
}
```

```
int result = left;
for (int i = left + 1; i <= right; i++) {
    result &= i;
    if (result == 0) break;
}
return result;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    Code30_BitwiseANDofNumbersRange solution = new Code30_BitwiseANDofNumbersRange();

    // 测试用例 1: 正常情况
    int left1 = 5, right1 = 7;
    int result1 = solution.rangeBitwiseAnd1(left1, right1);
    int result2 = solution.rangeBitwiseAnd2(left1, right1);
    int result3 = solution.rangeBitwiseAnd3(left1, right1);
    int result4 = solution.rangeBitwiseAnd4(left1, right1);
    int result5 = solution.rangeBitwiseAnd5(left1, right1);
    System.out.println("测试用例 1 - 输入: [" + left1 + ", " + right1 + "]");
    System.out.println("方法 1 结果: " + result1 + " (预期: 4)");
    System.out.println("方法 2 结果: " + result2 + " (预期: 4)");
    System.out.println("方法 3 结果: " + result3 + " (预期: 4)");
    System.out.println("方法 4 结果: " + result4 + " (预期: 4)");
    System.out.println("方法 5 结果: " + result5 + " (预期: 4)");

    // 测试用例 2: 边界情况 (相同数字)
    int left2 = 10, right2 = 10;
    int result6 = solution.rangeBitwiseAnd2(left2, right2);
    System.out.println("测试用例 2 - 输入: [" + left2 + ", " + right2 + "]");
    System.out.println("方法 2 结果: " + result6 + " (预期: 10)");

    // 测试用例 3: 大范围情况
    int left3 = 1, right3 = 2147483647;
    int result7 = solution.rangeBitwiseAnd2(left3, right3);
    System.out.println("测试用例 3 - 输入: [" + left3 + ", " + right3 + "]");
    System.out.println("方法 2 结果: " + result7 + " (预期: 0)");

    // 测试用例 4: 包含 2 的幂
    int left4 = 8, right4 = 15;
```

```
int result8 = solution.rangeBitwiseAnd2(left4, right4);
System.out.println("测试用例 4 - 输入: [" + left4 + ", " + right4 + "]");
System.out.println("方法 2 结果: " + result8 + " (预期: 8)");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 暴力法:");
System.out.println(" 时间复杂度: O(n) - n = right - left");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 位运算技巧:");
System.out.println(" 时间复杂度: O(1) - 最多 32 次位移");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 3 - Brian Kernighan 算法:");
System.out.println(" 时间复杂度: O(1) - 最多 32 次操作");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 4 - 位移法变种:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 5 - 数学方法:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

// 工程化考量
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 方法选择:");
System.out.println(" - 实际工程: 方法 2 (位运算技巧) 最优");
System.out.println(" - 面试场景: 方法 2 (位运算技巧) 最优");
System.out.println("2. 性能优化: 避免暴力法, 使用位运算");
System.out.println("3. 边界处理: 处理 left 等于 right 的情况");
System.out.println("4. 可读性: 方法 2 代码简洁易懂");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. 公共前缀: 区间按位与的结果就是公共前缀");
System.out.println("2. 位移操作: 通过右移找到公共前缀");
System.out.println("3. Brian Kernighan 技巧: 消除最低位的 1");
System.out.println("4. 最高位掩码: 使用 Integer.highestOneBit");
System.out.println("5. 提前终止: 当结果为 0 时可以提前终止");
```

```
// 应用场景
System.out.println("\n==== 应用场景 ===");
System.out.println("1. IP 地址范围计算");
System.out.println("2. 权限范围检查");
System.out.println("3. 位图操作中的范围查询");
System.out.println("4. 网络编程中的掩码计算");
}
}
```

=====

文件: Code30\_BitwiseANDofNumbersRange.py

=====

```
"""
数字范围按位与
测试链接: https://leetcode.cn/problems/bitwise-and-of-numbers-range/
```

题目描述:

给你两个整数 left 和 right，表示区间  $[left, right]$ ，返回此区间内所有数字按位与的结果（包含 left、right 端点）。

解题思路:

1. 暴力法: 遍历区间内所有数字进行按位与操作
2. 位运算技巧: 找到 left 和 right 的公共前缀
3. 位移法: 不断右移直到 left 等于 right，然后左移恢复
4. Brian Kernighan 算法: 利用  $n \& (n-1)$  消除最低位的 1

时间复杂度:  $O(1)$  – 最多 32 次操作

空间复杂度:  $O(1)$  – 只使用常数个变量

```
"""
class Code30_BitwiseANDofNumbersRange:
```

```
"""
    数字范围按位与解决方案
"""

@staticmethod
def range_bitwise_and1(left: int, right: int) -> int:
    """
        方法 1: 暴力法 (不推荐, 会超时)
        时间复杂度:  $O(right - left)$ 
        空间复杂度:  $O(1)$ 
    
```

Args:

left: 区间左端点

right: 区间右端点

Returns:

区间内所有数字按位与的结果

"""

```
result = left
for i in range(left + 1, right + 1):
    result &= i
    if result == 0:
        break # 提前终止优化
return result
```

@staticmethod

def range\_bitwise\_and2(left: int, right: int) -> int:

"""

方法 2: 位运算技巧 (最优解)

找到 left 和 right 的公共前缀

时间复杂度: O(1)

空间复杂度: O(1)

Args:

left: 区间左端点

right: 区间右端点

Returns:

区间内所有数字按位与的结果

"""

```
shift = 0
# 找到公共前缀
while left < right:
    left >>= 1
    right >>= 1
    shift += 1
return left << shift
```

@staticmethod

def range\_bitwise\_and3(left: int, right: int) -> int:

"""

方法 3: Brian Kernighan 算法

利用  $n \& (n-1)$  消除最低位的 1

时间复杂度: O(1)

空间复杂度: O(1)

Args:

left: 区间左端点  
right: 区间右端点

Returns:

区间内所有数字按位与的结果

"""

```
while left < right:  
    # 消除 right 最低位的 1  
    right = right & (right - 1)  
return right
```

@staticmethod

```
def range_bitwise_and4(left: int, right: int) -> int:
```

"""

方法 4: 位移法的另一种实现

时间复杂度: O(1)

空间复杂度: O(1)

Args:

left: 区间左端点  
right: 区间右端点

Returns:

区间内所有数字按位与的结果

"""

```
if left == right:  
    return left
```

# 计算 left 和 right 的最高不同位  
xor\_val = left ^ right  
mask = 1 << 31 # 从最高位开始找

# 找到最高不同位

```
while mask > 0 and (xor_val & mask) == 0:  
    mask >>= 1
```

# 创建掩码, 将不同位及之后的位都置为 0

```
mask = (mask << 1) - 1  
mask = ~mask
```

```
        return left & mask

@staticmethod
def range_bitwise_and5(left: int, right: int) -> int:
    """
    方法 5：利用位长度
    时间复杂度：O(1)
    空间复杂度：O(1)

    Args:
        left: 区间左端点
        right: 区间右端点

    Returns:
        区间内所有数字按位与的结果
    """

    if left == right:
        return left

    # 计算位长度
    left_bits = left.bit_length()
    right_bits = right.bit_length()

    # 如果位长度不同，结果为 0
    if left_bits != right_bits:
        return 0

    # 找到公共前缀
    common_prefix = left
    for i in range(left + 1, right + 1):
        common_prefix &= i
        if common_prefix == 0:
            break

    return common_prefix

@staticmethod
def test():
    """测试方法"""
    # 测试用例 1：正常情况
    left1, right1 = 5, 7
    result1 = Code30_BitwiseANDofNumbersRange.range_bitwise_and1(left1, right1)
    result2 = Code30_BitwiseANDofNumbersRange.range_bitwise_and2(left1, right1)
```

```

result3 = Code30_BitwiseANDofNumbersRange.range_bitwise_and3(left1, right1)
result4 = Code30_BitwiseANDofNumbersRange.range_bitwise_and4(left1, right1)
result5 = Code30_BitwiseANDofNumbersRange.range_bitwise_and5(left1, right1)
print(f"测试用例 1 - 输入: [{left1}, {right1}]")
print(f"方法 1 结果: {result1} (预期: 4)")
print(f"方法 2 结果: {result2} (预期: 4)")
print(f"方法 3 结果: {result3} (预期: 4)")
print(f"方法 4 结果: {result4} (预期: 4)")
print(f"方法 5 结果: {result5} (预期: 4)")

# 测试用例 2: 边界情况 (相同数字)
left2, right2 = 10, 10
result6 = Code30_BitwiseANDofNumbersRange.range_bitwise_and2(left2, right2)
print(f"测试用例 2 - 输入: [{left2}, {right2}]")
print(f"方法 2 结果: {result6} (预期: 10)")

# 测试用例 3: 大范围情况
left3, right3 = 1, 2**31 - 1
result7 = Code30_BitwiseANDofNumbersRange.range_bitwise_and2(left3, right3)
print(f"测试用例 3 - 输入: [{left3}, {right3}]")
print(f"方法 2 结果: {result7} (预期: 0)")

# 测试用例 4: 包含 2 的幂
left4, right4 = 8, 15
result8 = Code30_BitwiseANDofNumbersRange.range_bitwise_and2(left4, right4)
print(f"测试用例 4 - 输入: [{left4}, {right4}]")
print(f"方法 2 结果: {result8} (预期: 8)")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 暴力法:")
print(" 时间复杂度: O(n) - n = right - left")
print(" 空间复杂度: O(1)")

print("方法 2 - 位运算技巧:")
print(" 时间复杂度: O(1) - 最多 32 次位移")
print(" 空间复杂度: O(1)")

print("方法 3 - Brian Kernighan 算法:")
print(" 时间复杂度: O(1) - 最多 32 次操作")
print(" 空间复杂度: O(1)")

print("方法 4 - 位移法变种:")

```

```
print(" 时间复杂度: O(1)")  
print(" 空间复杂度: O(1)")  
  
print("方法 5 - 位长度法:")  
print(" 时间复杂度: O(1)")  
print(" 空间复杂度: O(1)")  
  
# 工程化考量  
print("\n==== 工程化考量 ===")  
print("1. 方法选择: ")  
print(" - 实际工程: 方法 2 (位运算技巧) 最优")  
print(" - 面试场景: 方法 2 (位运算技巧) 最优")  
print("2. 性能优化: 避免暴力法, 使用位运算")  
print("3. 边界处理: 处理 left 等于 right 的情况")  
print("4. 可读性: 方法 2 代码简洁易懂")  
  
# Python 特性考量  
print("\n==== Python 特性考量 ===")  
print("1. 位长度: 使用 bit_length() 方法获取位数")  
print("2. 大整数: Python 支持任意大整数")  
print("3. 位操作: Python 支持标准的位运算符")  
print("4. 范围操作: 使用 range 进行迭代")  
  
# 算法技巧总结  
print("\n==== 算法技巧总结 ===")  
print("1. 公共前缀: 区间按位与的结果就是公共前缀")  
print("2. 位移操作: 通过右移找到公共前缀")  
print("3. Brian Kernighan 技巧: 消除最低位的 1")  
print("4. 最高位掩码: 使用位运算创建掩码")  
print("5. 提前终止: 当结果为 0 时可以提前终止")  
  
if __name__ == "__main__":  
    Code30_BitwiseANDofNumbersRange. test()  
  
=====
```

文件: Code31\_MaximumXOROfTwoNumbersInAnArray. cpp

```
=====
```

```
/**  
 * 数组中两个数的最大异或值  
 * 测试链接: https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/  
 *  
 * 题目描述:
```

```
* 给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < nums.length。  
*  
* 解题思路：  
* 1. 暴力法：双重循环计算所有异或值  
* 2. 字典树法：使用 Trie 树存储二进制表示  
* 3. 哈希集合法：利用异或性质进行优化  
* 4. 位运算技巧：逐位确定最大值  
*  
* 时间复杂度：O(n) - 使用字典树或哈希集合  
* 空间复杂度：O(n) - 需要存储字典树或哈希集合  
*/
```

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

```
class Code31_MaximumXOROfTwoNumbersInAnArray {  
public:  
    /**  
     * 方法 1：暴力法（不推荐，会超时）  
     * 时间复杂度：O(n^2)  
     * 空间复杂度：O(1)  
     */  
    int findMaximumXOR1(vector<int>& nums) {  
        int max_val = 0;  
        int n = nums.size();  
        for (int i = 0; i < n; i++) {  
            for (int j = i + 1; j < n; j++) {  
                max_val = max(max_val, nums[i] ^ nums[j]);  
            }  
        }  
        return max_val;  
    }  
}
```

```
/**  
 * 方法 2：字典树法（最优解）  
 * 时间复杂度：O(n)  
 * 空间复杂度：O(n)  
 */  
class TrieNode {  
public:
```

```

TrieNode* children[2]; // 0 和 1 两个分支

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

~TrieNode() {
    if (children[0]) delete children[0];
    if (children[1]) delete children[1];
}

};

int findMaximumXOR2(vector<int>& nums) {
    if (nums.empty()) return 0;

    // 创建字典树根节点
    TrieNode* root = new TrieNode();

    // 构建字典树
    for (int num : nums) {
        TrieNode* node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node->children[bit] == nullptr) {
                node->children[bit] = new TrieNode();
            }
            node = node->children[bit];
        }
    }

    int max_val = 0;
    // 对每个数字，在字典树中寻找最大异或值
    for (int num : nums) {
        TrieNode* node = root;
        int current_max = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            int desired_bit = 1 - bit; // 希望找到相反的位

            if (node->children[desired_bit] != nullptr) {
                current_max |= (1 << i);
                node = node->children[desired_bit];
            }
        }
        max_val = max(max_val, current_max);
    }
}

```

```

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

    max_val = max(max_val, current_max);
}

delete root; // 释放内存
return max_val;
}

/***
 * 方法 3: 哈希集合法
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
int findMaximumXOR3(vector<int>& nums) {
    int max_val = 0, mask = 0;

    for (int i = 31; i >= 0; i--) {
        mask |= (1 << i);
        unordered_set<int> prefix_set;

        // 提取前缀
        for (int num : nums) {
            prefix_set.insert(num & mask);
        }

        // 尝试设置当前位为 1
        int temp = max_val | (1 << i);
        for (int prefix : prefix_set) {
            if (prefix_set.find(temp ^ prefix) != prefix_set.end()) {
                max_val = temp;
                break;
            }
        }
    }

    return max_val;
}

/***
 * 方法 4: 位运算优化版
 */

```

```

* 时间复杂度: O(n)
* 空间复杂度: O(n)
*/
int findMaximumXOR4(vector<int>& nums) {
    int max_result = 0;
    int mask = 0;

    // 从最高位开始尝试
    for (int i = 31; i >= 0; i--) {
        // 设置掩码，只保留前 i 位
        mask = mask | (1 << i);

        unordered_set<int> left_bits;
        for (int num : nums) {
            left_bits.insert(num & mask);
        }

        // 尝试设置当前位为 1
        int greedy_try = max_result | (1 << i);
        for (int left_bit : left_bits) {
            // 如果存在两个数使得它们的异或等于 greedy_try
            if (left_bits.find(greedy_try ^ left_bit) != left_bits.end()) {
                max_result = greedy_try;
                break;
            }
        }
    }

    return max_result;
}

/**
 * 测试方法
 */
static void test() {
    Code31_MaximumXOROfTwoNumbersInAnArray solution;

    // 测试用例 1: 正常情况
    vector<int> nums1 = {3, 10, 5, 25, 2, 8};
    int result1 = solution.findMaximumXOR1(nums1);
    int result2 = solution.findMaximumXOR2(nums1);
    int result3 = solution.findMaximumXOR3(nums1);
    int result4 = solution.findMaximumXOR4(nums1);
}

```

```
cout << "测试用例 1 - 输入: [3, 10, 5, 25, 2, 8]" << endl;
cout << "方法 1 结果: " << result1 << " (预期: 28)" << endl;
cout << "方法 2 结果: " << result2 << " (预期: 28)" << endl;
cout << "方法 3 结果: " << result3 << " (预期: 28)" << endl;
cout << "方法 4 结果: " << result4 << " (预期: 28)" << endl;

// 测试用例 2: 边界情况 (两个元素)
vector<int> nums2 = {1, 2};
int result5 = solution.findMaximumXOR2(nums2);
cout << "测试用例 2 - 输入: [1, 2]" << endl;
cout << "方法 2 结果: " << result5 << " (预期: 3)" << endl;

// 测试用例 3: 边界情况 (一个元素)
vector<int> nums3 = {5};
int result6 = solution.findMaximumXOR2(nums3);
cout << "测试用例 3 - 输入: [5]" << endl;
cout << "方法 2 结果: " << result6 << " (预期: 0)" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 暴力法:" << endl;
cout << " 时间复杂度: O(n2)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 字典树法:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 3 - 哈希集合法:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 4 - 位运算优化版:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 方法选择: " << endl;
cout << " - 实际工程: 方法 2 (字典树法) 最优" << endl;
cout << " - 面试场景: 方法 2 (字典树法) 最优" << endl;
cout << "2. 性能优化: 避免暴力法, 使用高效数据结构" << endl;
cout << "3. 边界处理: 处理空数组和单元素数组" << endl;
```

```

cout << "4. 内存管理: C++需要手动释放字典树内存" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 内存管理: 使用析构函数释放字典树" << endl;
cout << "2. 标准库: 使用 unordered_set 提高查找效率" << endl;
cout << "3. 引用传递: 使用 vector 引用避免拷贝" << endl;
cout << "4. 智能指针: 实际工程中可使用智能指针" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 字典树应用: 高效处理二进制前缀" << endl;
cout << "2. 贪心策略: 从高位到低位确定最大值" << endl;
cout << "3. 哈希集合: 利用集合特性快速查找" << endl;
cout << "4. 位掩码: 逐位构建最大异或值" << endl;
}

};

int main() {
    Code31_MaximumXOROfTwoNumbersInAnArray::test();
    return 0;
}

```

=====

文件: Code31\_MaximumXOROfTwoNumbersInAnArray.java

=====

```

/**
 * 数组中两个数的最大异或值
 * 测试链接: https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/
 *
 * 题目描述:
 * 给你一个整数数组 nums，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < nums.length。
 *
 * 解题思路:
 * 1. 暴力法: 双重循环计算所有异或值
 * 2. 字典树法: 使用 Trie 树存储二进制表示
 * 3. 哈希集合法: 利用异或性质进行优化
 * 4. 位运算技巧: 逐位确定最大值
 *
 * 时间复杂度: O(n) - 使用字典树或哈希集合
 * 空间复杂度: O(n) - 需要存储字典树或哈希集合

```

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

public class Code31_MaximumXOROfTwoNumbersInAnArray {

    /**
     * 方法 1：暴力法（不推荐，会超时）
     * 时间复杂度：O(n2)
     * 空间复杂度：O(1)
     */
    public int findMaximumXOR1(int[] nums) {
        int max = 0;
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                max = Math.max(max, nums[i] ^ nums[j]);
            }
        }
        return max;
    }

    /**
     * 方法 2：字典树法（最优解）
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    public int findMaximumXOR2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // 创建字典树
        TrieNode root = new TrieNode();

        // 构建字典树
        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];
            }
        }
    }
}
```

```

        }

    }

    int max = 0;
    // 对每个数字，在字典树中寻找最大异或值
    for (int num : nums) {
        TrieNode node = root;
        int currentMax = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            int desiredBit = 1 - bit; // 希望找到相反的位

            if (node.children[desiredBit] != null) {
                currentMax |= (1 << i);
                node = node.children[desiredBit];
            } else {
                node = node.children[bit];
            }
        }
        max = Math.max(max, currentMax);
    }

    return max;
}

/***
 * 字典树节点
 */
class TrieNode {
    TrieNode[] children;

    public TrieNode() {
        children = new TrieNode[2]; // 0 和 1 两个分支
    }
}

/***
 * 方法 3：哈希集合法
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public int findMaximumXOR3(int[] nums) {
    int max = 0, mask = 0;

```

```

for (int i = 31; i >= 0; i--) {
    mask |= (1 << i);
    Set<Integer> set = new HashSet<>();

    // 提取前缀
    for (int num : nums) {
        set.add(num & mask);
    }

    // 尝试设置当前位为 1
    int temp = max | (1 << i);
    for (int prefix : set) {
        if (set.contains(temp ^ prefix)) {
            max = temp;
            break;
        }
    }
}

return max;
}

/**
 * 方法 4：位运算优化版
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int findMaximumXOR4(int[] nums) {
    int maxResult = 0;
    int mask = 0;

    // 从最高位开始尝试
    for (int i = 31; i >= 0; i--) {
        // 设置掩码，只保留前 i 位
        mask = mask | (1 << i);

        Set<Integer> leftBits = new HashSet<>();
        for (int num : nums) {
            leftBits.add(num & mask);
        }

        // 尝试设置当前位为 1

```

```
int greedyTry = maxResult | (1 << i);
for (int leftBit : leftBits) {
    // 如果存在两个数使得它们的异或等于 greedyTry
    if (leftBits.contains(greedyTry ^ leftBit)) {
        maxResult = greedyTry;
        break;
    }
}

return maxResult;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code31_MaximumXOROfTwoNumbersInAnArray solution = new
Code31_MaximumXOROfTwoNumbersInAnArray();

    // 测试用例 1: 正常情况
    int[] nums1 = {3, 10, 5, 25, 2, 8};
    int result1 = solution.findMaximumXOR1(nums1);
    int result2 = solution.findMaximumXOR2(nums1);
    int result3 = solution.findMaximumXOR3(nums1);
    int result4 = solution.findMaximumXOR4(nums1);
    System.out.println("测试用例 1 - 输入: " + Arrays.toString(nums1));
    System.out.println("方法 1 结果: " + result1 + " (预期: 28)");
    System.out.println("方法 2 结果: " + result2 + " (预期: 28)");
    System.out.println("方法 3 结果: " + result3 + " (预期: 28)");
    System.out.println("方法 4 结果: " + result4 + " (预期: 28)");

    // 测试用例 2: 边界情况 (两个元素)
    int[] nums2 = {1, 2};
    int result5 = solution.findMaximumXOR2(nums2);
    System.out.println("测试用例 2 - 输入: " + Arrays.toString(nums2));
    System.out.println("方法 2 结果: " + result5 + " (预期: 3)");

    // 测试用例 3: 边界情况 (一个元素)
    int[] nums3 = {5};
    int result6 = solution.findMaximumXOR2(nums3);
    System.out.println("测试用例 3 - 输入: " + Arrays.toString(nums3));
    System.out.println("方法 2 结果: " + result6 + " (预期: 0)");
}
```

```
// 复杂度分析
System.out.println("\n==== 复杂度分析 ===");
System.out.println("方法 1 - 暴力法:");
System.out.println(" 时间复杂度: O(n2)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 字典树法:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 3 - 哈希集合法:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 4 - 位运算优化版:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

// 工程化考量
System.out.println("\n==== 工程化考量 ===");
System.out.println("1. 方法选择:");
System.out.println(" - 实际工程: 方法 2 (字典树法) 最优");
System.out.println(" - 面试场景: 方法 2 (字典树法) 最优");
System.out.println("2. 性能优化: 避免暴力法, 使用高效数据结构");
System.out.println("3. 边界处理: 处理空数组和单元素数组");
System.out.println("4. 可读性: 添加详细注释说明算法原理");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ===");
System.out.println("1. 字典树应用: 高效处理二进制前缀");
System.out.println("2. 贪心策略: 从高位到低位确定最大值");
System.out.println("3. 哈希集合: 利用集合特性快速查找");
System.out.println("4. 位掩码: 逐位构建最大异或值");

}

}

=====

文件: Code31_MaximumXOROfTwoNumbersInAnArray.py
=====

"""

数组中两个数的最大异或值
```

测试链接: <https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/>

题目描述:

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

解题思路:

1. 暴力法: 双重循环计算所有异或值
2. 字典树法: 使用 Trie 树存储二进制表示
3. 哈希集合法: 利用异或性质进行优化
4. 位运算技巧: 逐位确定最大值

时间复杂度:  $O(n)$  - 使用字典树或哈希集合

空间复杂度:  $O(n)$  - 需要存储字典树或哈希集合

"""

```
class TrieNode:  
    """字典树节点"""  
    def __init__(self):  
        self.children = [None, None] # 0 和 1 两个分支
```

```
class Code31_MaximumXOROfTwoNumbersInAnArray:
```

"""\n

数组中两个数的最大异或值解决方案\n"""\n

```
@staticmethod  
def find_maximum_xor1(nums: list[int]) -> int:  
    """
```

方法 1: 暴力法 (不推荐, 会超时)

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

Args:

`nums`: 整数数组

Returns:

    最大异或值

"""\n

```
if not nums:  
    return 0
```

```
max_val = 0
```

```
n = len(nums)
for i in range(n):
    for j in range(i + 1, n):
        max_val = max(max_val, nums[i] ^ nums[j])
return max_val
```

```
@staticmethod
def find_maximum_xor2(nums: list[int]) -> int:
    """
```

方法 2：字典树法（最优解）

时间复杂度：O(n)

空间复杂度：O(n)

Args:

nums: 整数数组

Returns:

最大异或值

"""

```
if not nums:
    return 0
```

# 创建字典树根节点

```
root = TrieNode()
```

# 构建字典树

```
for num in nums:
```

```
    node = root
```

```
    for i in range(31, -1, -1):
```

```
        bit = (num >> i) & 1
```

```
        if node.children[bit] is None:
```

```
            node.children[bit] = TrieNode()
```

```
        node = node.children[bit]
```

```
max_val = 0
```

# 对每个数字，在字典树中寻找最大异或值

```
for num in nums:
```

```
    node = root
```

```
    current_max = 0
```

```
    for i in range(31, -1, -1):
```

```
        bit = (num >> i) & 1
```

```
        desired_bit = 1 - bit # 希望找到相反的位
```

```
        if node.children[desired_bit] is not None:
            current_max |= (1 << i)
            node = node.children[desired_bit]
        else:
            node = node.children[bit]
        max_val = max(max_val, current_max)

    return max_val
```

@staticmethod

```
def find_maximum_xor3(nums: list[int]) -> int:
```

"""

方法 3：哈希集合法

时间复杂度：O(n)

空间复杂度：O(n)

Args:

nums: 整数数组

Returns:

最大异或值

"""

```
if not nums:
```

```
    return 0
```

```
max_val = 0
```

```
mask = 0
```

```
for i in range(31, -1, -1):
```

```
    mask |= (1 << i)
```

```
    prefix_set = set()
```

# 提取前缀

```
for num in nums:
```

```
    prefix_set.add(num & mask)
```

# 尝试设置当前位为 1

```
temp = max_val | (1 << i)
```

```
for prefix in prefix_set:
```

```
    if (temp ^ prefix) in prefix_set:
```

```
        max_val = temp
```

```
        break
```

```
return max_val

@staticmethod
def find_maximum_xor4(nums: list[int]) -> int:
    """
    方法 4: 位运算优化版
    时间复杂度: O(n)
    空间复杂度: O(n)

    Args:
        nums: 整数数组

    Returns:
        最大异或值
    """
    if not nums:
        return 0

    max_result = 0
    mask = 0

    # 从最高位开始尝试
    for i in range(31, -1, -1):
        # 设置掩码, 只保留前 i 位
        mask = mask | (1 << i)

        left_bits = set()
        for num in nums:
            left_bits.add(num & mask)

        # 尝试设置当前位为 1
        greedy_try = max_result | (1 << i)
        for left_bit in left_bits:
            # 如果存在两个数使得它们的异或等于 greedy_try
            if (greedy_try ^ left_bit) in left_bits:
                max_result = greedy_try
                break

    return max_result

@staticmethod
def test():
    """测试方法"""

```

```
# 测试用例 1: 正常情况
nums1 = [3, 10, 5, 25, 2, 8]
result1 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor1(nums1)
result2 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor2(nums1)
result3 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor3(nums1)
result4 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor4(nums1)
print(f"测试用例 1 - 输入: {nums1}")
print(f"方法 1 结果: {result1} (预期: 28)")
print(f"方法 2 结果: {result2} (预期: 28)")
print(f"方法 3 结果: {result3} (预期: 28)")
print(f"方法 4 结果: {result4} (预期: 28)")

# 测试用例 2: 边界情况 (两个元素)
nums2 = [1, 2]
result5 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor2(nums2)
print(f"测试用例 2 - 输入: {nums2}")
print(f"方法 2 结果: {result5} (预期: 3)")

# 测试用例 3: 边界情况 (一个元素)
nums3 = [5]
result6 = Code31_MaximumXOROfTwoNumbersInAnArray.find_maximum_xor2(nums3)
print(f"测试用例 3 - 输入: {nums3}")
print(f"方法 2 结果: {result6} (预期: 0)")

# 复杂度分析
print("\n==== 复杂度分析 ===")
print("方法 1 - 暴力法:")
print("  时间复杂度: O(n2)")
print("  空间复杂度: O(1)")

print("方法 2 - 字典树法:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

print("方法 3 - 哈希集合法:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

print("方法 4 - 位运算优化版:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

# 工程化考量
```

```

print("\n==== 工程化考量 ===")
print("1. 方法选择: ")
print("    - 实际工程: 方法 2 (字典树法) 最优")
print("    - 面试场景: 方法 2 (字典树法) 最优")
print("2. 性能优化: 避免暴力法, 使用高效数据结构")
print("3. 边界处理: 处理空数组和单元素数组")
print("4. 可读性: 添加详细注释说明算法原理")

# Python 特性考量
print("\n==== Python 特性考量 ===")
print("1. 集合操作: 使用 set 进行高效查找")
print("2. 位运算: Python 支持标准的位运算符")
print("3. 内存管理: Python 自动管理字典树内存")
print("4. 类型注解: 使用类型提示提高代码可读性")

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 字典树应用: 高效处理二进制前缀")
print("2. 贪心策略: 从高位到低位确定最大值")
print("3. 哈希集合: 利用集合特性快速查找")
print("4. 位掩码: 逐位构建最大异或值")

if __name__ == "__main__":
    Code31_MaximumXOROfTwoNumbersInAnArray.test()

```

=====

文件: Code32\_RepeatedDNASequences.cpp

=====

```

/**
 * 重复的 DNA 序列
 * 测试链接: https://leetcode.cn/problems/repeated-dna-sequences/
 *
 * 题目描述:
 * DNA 序列 由一系列核苷酸组成, 缩写为 'A', 'C', 'G' 和 'T'。
 * 例如, "ACGAATTCCG" 是一个 DNA 序列。
 * 在研究 DNA 时, 识别 DNA 中的重复序列非常有用。
 * 给定一个表示 DNA 序列 的字符串 s , 返回所有在 DNA 分子中出现不止一次的 长度为 10 的序列(子字符串)。你可以按 任意顺序 返回答案。
 *
 * 解题思路:
 * 1. 哈希表法: 使用 unordered_map 统计所有 10 位子串的出现次数
 * 2. 位运算优化: 使用 2 位表示一个字符, 将字符串转换为整数

```

```
* 3. 滑动窗口：使用滑动窗口和位运算结合
* 4. Rabin-Karp 算法：使用滚动哈希优化
*
* 时间复杂度：O(n) - n 为字符串长度
* 空间复杂度：O(n) - 需要存储哈希表
*/
```

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
using namespace std;

class Code32_RepeatedDNASequences {
public:
    /**
     * 方法 1：哈希表法（直接使用字符串）
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    vector<string> findRepeatedDnaSequences1(string s) {
        vector<string> result;
        if (s.length() < 10) {
            return result;
        }

        unordered_map<string, int> countMap;

        // 遍历所有长度为 10 的子串
        for (int i = 0; i <= s.length() - 10; i++) {
            string substring = s.substr(i, 10);
            countMap[substring]++;
        }

        // 收集出现次数大于 1 的子串
        for (auto& entry : countMap) {
            if (entry.second > 1) {
                result.push_back(entry.first);
            }
        }

        return result;
    }
}
```

```

/**
 * 方法 2：位运算优化（使用整数表示子串）
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */

vector<string> findRepeatedDnaSequences2(string s) {
    vector<string> result;
    if (s.length() < 10) {
        return result;
    }

    // 字符到 2 位编码的映射
    unordered_map<char, int> charToCode;
    charToCode['A'] = 0; // 00
    charToCode['C'] = 1; // 01
    charToCode['G'] = 2; // 10
    charToCode['T'] = 3; // 11

    unordered_map<int, int> countMap;

    // 第一个窗口的编码
    int code = 0;
    for (int i = 0; i < 10; i++) {
        code = (code << 2) | charToCode[s[i]];
    }
    countMap[code] = 1;

    // 滑动窗口处理剩余部分
    for (int i = 10; i < s.length(); i++) {
        // 移除最左边的字符（左移 2 位，然后取低 20 位）
        code = ((code << 2) | charToCode[s[i]]) & 0xFFFF;
        countMap[code]++;
    }

    // 重新遍历字符串，将编码转换回字符串
    unordered_map<int, string> codeToString;
    for (int i = 0; i <= s.length() - 10; i++) {
        int currentCode = 0;
        for (int j = 0; j < 10; j++) {
            currentCode = (currentCode << 2) | charToCode[s[i + j]];
        }
        codeToString[currentCode] = s.substr(i, 10);
    }
}

```

```

}

// 收集结果
for (auto& entry : countMap) {
    if (entry.second > 1) {
        result.push_back(codeToString[entry.first]);
    }
}

return result;
}

/***
 * 方法 3：滑动窗口+位运算（优化版）
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
vector<string> findRepeatedDnaSequences3(string s) {
    vector<string> result;
    if (s.length() < 10) {
        return result;
    }

    // 字符到 2 位编码的映射
    unordered_map<char, int> charToCode;
    charToCode['A'] = 0;
    charToCode['C'] = 1;
    charToCode['G'] = 2;
    charToCode['T'] = 3;

    unordered_set<int> seen;
    unordered_set<string> output;

    int code = 0;
    // 处理前 10 个字符
    for (int i = 0; i < 10; i++) {
        code = (code << 2) | charToCode[s[i]];
    }
    seen.insert(code);

    // 滑动窗口
    for (int i = 10; i < s.length(); i++) {
        code = ((code << 2) | charToCode[s[i]]) & 0xFFFF;

```

```

        if (seen.find(code) != seen.end()) {
            output.insert(s.substr(i - 9, 10));
        } else {
            seen.insert(code);
        }
    }

    for (const string& seq : output) {
        result.push_back(seq);
    }

    return result;
}

/***
 * 方法 4: Rabin-Karp 算法 (滚动哈希)
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
vector<string> findRepeatedDnaSequences4(string s) {
    vector<string> result;
    if (s.length() < 10) {
        return result;
    }

    // 使用质数作为基数
    int base = 4; // 4 个字符
    long mod = 1e9 + 7; // 大质数取模

    // 字符到数字的映射
    unordered_map<char, int> charToNum;
    charToNum['A'] = 0;
    charToNum['C'] = 1;
    charToNum['G'] = 2;
    charToNum['T'] = 3;

    // 计算 base^9 mod mod
    long highestPower = 1;
    for (int i = 0; i < 9; i++) {
        highestPower = (highestPower * base) % mod;
    }

    unordered_map<long, int> countMap;

```

```

// 计算第一个窗口的哈希值
long hash = 0;
for (int i = 0; i < 10; i++) {
    hash = (hash * base + charToNum[s[i]]) % mod;
}
countMap[hash] = 1;

// 滑动窗口计算哈希值
for (int i = 10; i < s.length(); i++) {
    // 移除最左边的字符
    long leftCharValue = charToNum[s[i - 10]] * highestPower % mod;
    hash = (hash - leftCharValue + mod) % mod;
    // 添加新的字符
    hash = (hash * base + charToNum[s[i]]) % mod;

    countMap[hash]++;
}

// 收集结果
unordered_set<string> added;
for (int i = 0; i <= s.length() - 10; i++) {
    string substring = s.substr(i, 10);
    long currentHash = 0;
    for (int j = 0; j < 10; j++) {
        currentHash = (currentHash * base + charToNum[s[i + j]]) % mod;
    }
    if (countMap[currentHash] > 1 && added.find(substring) == added.end()) {
        result.push_back(substring);
        added.insert(substring);
    }
}

return result;
}

/***
 * 测试方法
 */
static void test() {
    Code32_RepeatedDNASequences solution;

    // 测试用例 1: 正常情况

```

```

string s1 = "AAAAACCCCCAAAAACCCCCCAAAAGGGTTT";
vector<string> result1 = solution.findRepeatedDnaSequences1(s1);
vector<string> result2 = solution.findRepeatedDnaSequences2(s1);
vector<string> result3 = solution.findRepeatedDnaSequences3(s1);
vector<string> result4 = solution.findRepeatedDnaSequences4(s1);
cout << "测试用例 1 - 输入: " << s1 << endl;
cout << "方法 1 结果: ";
for (const string& seq : result1) cout << seq << " ";
cout << "(预期: AAAAACCCCC CCCCAAAAAA)" << endl;

cout << "方法 2 结果: ";
for (const string& seq : result2) cout << seq << " ";
cout << "(预期: AAAAACCCCC CCCCAAAAAA)" << endl;

cout << "方法 3 结果: ";
for (const string& seq : result3) cout << seq << " ";
cout << "(预期: AAAAACCCCC CCCCAAAAAA)" << endl;

cout << "方法 4 结果: ";
for (const string& seq : result4) cout << seq << " ";
cout << "(预期: AAAAACCCCC CCCCAAAAAA)" << endl;

// 测试用例 2: 边界情况 (无重复)
string s2 = "AAAAAAAAAA";
vector<string> result5 = solution.findRepeatedDnaSequences2(s2);
cout << "测试用例 2 - 输入: " << s2 << endl;
cout << "方法 2 结果: ";
for (const string& seq : result5) cout << seq << " ";
cout << "(预期: 空)" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 哈希表法:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 2 - 位运算优化:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 3 - 滑动窗口+位运算:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

```

```

cout << "方法 4 - Rabin-Karp 算法:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

// C++特性考量
cout << "\n==== C++特性考量 ===" << endl;
cout << "1. 标准库: 使用 unordered_map 和 unordered_set" << endl;
cout << "2. 字符串操作: 使用 substr 获取子串" << endl;
cout << "3. 内存管理: 自动管理容器内存" << endl;
cout << "4. 性能优化: 避免不必要的字符串拷贝" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 位编码: 使用 2 位表示一个字符, 节省空间" << endl;
cout << "2. 滑动窗口: 高效处理固定长度子串" << endl;
cout << "3. 哈希优化: 使用整数编码替代字符串比较" << endl;
cout << "4. 滚动哈希: Rabin-Karp 算法处理字符串匹配" << endl;
}

};

int main() {
    Code32_RepeatedDNASequences::test();
    return 0;
}

```

=====

文件: Code32\_RepeatedDNASequences.java

=====

```

/**
 * 重复的 DNA 序列
 * 测试链接: https://leetcode.cn/problems/repeated-dna-sequences/
 *
 * 题目描述:
 * DNA 序列 由一系列核苷酸组成, 缩写为 'A', 'C', 'G' 和 'T'。
 * 例如, "ACGAATTCCG" 是一个 DNA 序列。
 * 在研究 DNA 时, 识别 DNA 中的重复序列非常有用。
 * 给定一个表示 DNA 序列 的字符串 s , 返回所有在 DNA 分子中出现不止一次的 长度为 10 的序列(子字符串)。你可以按 任意顺序 返回答案。
 *
 * 解题思路:
 * 1. 哈希表法: 使用 HashMap 统计所有 10 位子串的出现次数

```

```

* 2. 位运算优化：使用 2 位表示一个字符，将字符串转换为整数
* 3. 滑动窗口：使用滑动窗口和位运算结合
* 4. Rabin-Karp 算法：使用滚动哈希优化
*
* 时间复杂度：O(n) - n 为字符串长度
* 空间复杂度：O(n) - 需要存储哈希表
*/
import java.util.*;

public class Code32_RepeatedDNASequences {

    /**
     * 方法 1：哈希表法（直接使用字符串）
     * 时间复杂度：O(n)
     * 空间复杂度：O(n)
     */
    public List<String> findRepeatedDnaSequences1(String s) {
        List<String> result = new ArrayList<>();
        if (s == null || s.length() < 10) {
            return result;
        }

        Map<String, Integer> countMap = new HashMap<>();

        // 遍历所有长度为 10 的子串
        for (int i = 0; i <= s.length() - 10; i++) {
            String substring = s.substring(i, i + 10);
            countMap.put(substring, countMap.getOrDefault(substring, 0) + 1);
        }

        // 收集出现次数大于 1 的子串
        for (Map.Entry<String, Integer> entry : countMap.entrySet()) {
            if (entry.getValue() > 1) {
                result.add(entry.getKey());
            }
        }

        return result;
    }

    /**
     * 方法 2：位运算优化（使用整数表示子串）
     * 时间复杂度：O(n)

```

```

* 空间复杂度: O(n)
*/
public List<String> findRepeatedDnaSequences2(String s) {
    List<String> result = new ArrayList<>();
    if (s == null || s.length() < 10) {
        return result;
    }

    // 字符到 2 位编码的映射
    Map<Character, Integer> charToCode = new HashMap<>();
    charToCode.put('A', 0); // 00
    charToCode.put('C', 1); // 01
    charToCode.put('G', 2); // 10
    charToCode.put('T', 3); // 11

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

    // 第一个窗口的编码
    int code = 0;
    for (int i = 0; i < 10; i++) {
        code = (code << 2) | charToCode.get(s.charAt(i));
    }
    countMap.put(code, 1);

    // 滑动窗口处理剩余部分
    for (int i = 10; i < s.length(); i++) {
        // 移除最左边的字符 (左移 2 位, 然后取低 20 位)
        code = ((code << 2) | charToCode.get(s.charAt(i))) & 0xFFFF;
        countMap.put(code, countMap.getOrDefault(code, 0) + 1);
    }

    // 重新遍历字符串, 将编码转换回字符串
    Map<Integer, String> codeToString = new HashMap<>();
    for (int i = 0; i <= s.length() - 10; i++) {
        int currentCode = 0;
        for (int j = 0; j < 10; j++) {
            currentCode = (currentCode << 2) | charToCode.get(s.charAt(i + j));
        }
        codeToString.put(currentCode, s.substring(i, i + 10));
    }

    // 收集结果
    for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {

```

```

        if (entry.getValue() > 1) {
            result.add(codeToString.get(entry.getKey()));
        }
    }

    return result;
}

/***
 * 方法 3：滑动窗口+位运算（优化版）
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public List<String> findRepeatedDnaSequences3(String s) {
    List<String> result = new ArrayList<>();
    if (s == null || s.length() < 10) {
        return result;
    }

    // 字符到 2 位编码的映射
    Map<Character, Integer> charToCode = new HashMap<>();
    charToCode.put('A', 0);
    charToCode.put('C', 1);
    charToCode.put('G', 2);
    charToCode.put('T', 3);

    Set<Integer> seen = new HashSet<>();
    Set<String> output = new HashSet<>();

    int code = 0;
    // 处理前 10 个字符
    for (int i = 0; i < 10; i++) {
        code = (code << 2) | charToCode.get(s.charAt(i));
    }
    seen.add(code);

    // 滑动窗口
    for (int i = 10; i < s.length(); i++) {
        code = ((code << 2) | charToCode.get(s.charAt(i))) & 0xFFFF;
        if (seen.contains(code)) {
            output.add(s.substring(i - 9, i + 1));
        } else {
            seen.add(code);
        }
    }
}

```

```

        }

    }

    return new ArrayList<>(output);
}

/***
 * 方法 4: Rabin-Karp 算法 (滚动哈希)
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public List<String> findRepeatedDnaSequences4(String s) {
    List<String> result = new ArrayList<>();
    if (s == null || s.length() < 10) {
        return result;
    }

    // 使用质数作为基数
    int base = 4; // 4 个字符
    long mod = (long)1e9 + 7; // 大质数取模

    // 字符到数字的映射
    Map<Character, Integer> charToNum = new HashMap<>();
    charToNum.put('A', 0);
    charToNum.put('C', 1);
    charToNum.put('G', 2);
    charToNum.put('T', 3);

    // 计算 base^9 mod mod
    long highestPower = 1;
    for (int i = 0; i < 9; i++) {
        highestPower = (highestPower * base) % mod;
    }

    Map<Long, Integer> countMap = new HashMap<>();

    // 计算第一个窗口的哈希值
    long hash = 0;
    for (int i = 0; i < 10; i++) {
        hash = (hash * base + charToNum.get(s.charAt(i))) % mod;
    }
    countMap.put(hash, 1);
}

```

```

// 滑动窗口计算哈希值
for (int i = 10; i < s.length(); i++) {
    // 移除最左边的字符
    long leftCharValue = charToNum.get(s.charAt(i - 10)) * highestPower % mod;
    hash = (hash - leftCharValue + mod) % mod;
    // 添加新的字符
    hash = (hash * base + charToNum.get(s.charAt(i))) % mod;

    countMap.put(hash, countMap.getOrDefault(hash, 0) + 1);
}

// 收集结果
Set<String> added = new HashSet<>();
for (int i = 0; i <= s.length() - 10; i++) {
    String substring = s.substring(i, i + 10);
    long currentHash = 0;
    for (int j = 0; j < 10; j++) {
        currentHash = (currentHash * base + charToNum.get(s.charAt(i + j))) % mod;
    }
    if (countMap.get(currentHash) > 1 && !added.contains(substring)) {
        result.add(substring);
        added.add(substring);
    }
}

return result;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code32_RepeatedDNASequences solution = new Code32_RepeatedDNASequences();

    // 测试用例 1: 正常情况
    String s1 = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT";
    List<String> result1 = solution.findRepeatedDnaSequences1(s1);
    List<String> result2 = solution.findRepeatedDnaSequences2(s1);
    List<String> result3 = solution.findRepeatedDnaSequences3(s1);
    List<String> result4 = solution.findRepeatedDnaSequences4(s1);
    System.out.println("测试用例 1 - 输入: " + s1);
    System.out.println("方法 1 结果: " + result1 + " (预期: [AAAAACCCCC, CCCCCAAAAA])");
    System.out.println("方法 2 结果: " + result2 + " (预期: [AAAAACCCCC, CCCCCAAAAA])");
}

```

```
System.out.println("方法 3 结果: " + result3 + " (预期: [AAAAACCCCC, CCCCCAAAAA])");
System.out.println("方法 4 结果: " + result4 + " (预期: [AAAAACCCCC, CCCCCAAAAA])");

// 测试用例 2: 边界情况 (无重复)
String s2 = "AAAAAAAAAA";
List<String> result5 = solution.findRepeatedDnaSequences2(s2);
System.out.println("测试用例 2 - 输入: " + s2);
System.out.println("方法 2 结果: " + result5 + " (预期: [])");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 哈希表法:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 2 - 位运算优化:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 3 - 滑动窗口+位运算:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 4 - Rabin-Karp 算法:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

// 工程化考量
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 方法选择:");
System.out.println(" - 实际工程: 方法 2 (位运算优化) 最优");
System.out.println(" - 面试场景: 方法 2 (位运算优化) 最优");
System.out.println("2. 性能优化: 避免字符串比较, 使用整数编码");
System.out.println("3. 边界处理: 处理字符串长度不足的情况");
System.out.println("4. 内存优化: 使用位运算减少内存占用");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. 位编码: 使用 2 位表示一个字符, 节省空间");
System.out.println("2. 滑动窗口: 高效处理固定长度子串");
System.out.println("3. 哈希优化: 使用整数编码替代字符串比较");
System.out.println("4. 滚动哈希: Rabin-Karp 算法处理字符串匹配");
}
```

}

=====

文件: Code32\_RepeatedDNASequences.py

=====

"""

重复的 DNA 序列

测试链接: <https://leetcode.cn/problems/repeated-dna-sequences/>

题目描述:

DNA 序列 由一系列核苷酸组成，缩写为 'A'，'C'，'G' 和 'T'。

例如，"ACGAATTCCG" 是一个 DNA 序列。

在研究 DNA 时，识别 DNA 中的重复序列非常有用。

给定一个表示 DNA 序列 的字符串 s，返回所有在 DNA 分子中出现不止一次的 长度为 10 的序列(子字符串)。你可以按 任意顺序 返回答案。

解题思路:

1. 哈希表法: 使用字典统计所有 10 位子串的出现次数
2. 位运算优化: 使用 2 位表示一个字符，将字符串转换为整数
3. 滑动窗口: 使用滑动窗口和位运算结合
4. Rabin-Karp 算法: 使用滚动哈希优化

时间复杂度: O(n) – n 为字符串长度

空间复杂度: O(n) – 需要存储哈希表

"""

class Code32\_RepeatedDNASequences:

"""

重复的 DNA 序列解决方案

"""

@staticmethod

def find\_repeated\_dna\_sequences1(s: str) -> list[str]:

"""

方法 1: 哈希表法 (直接使用字符串)

时间复杂度: O(n)

空间复杂度: O(n)

Args:

s: DNA 序列字符串

Returns:

```

    重复出现的 10 位 DNA 序列列表
"""

if len(s) < 10:
    return []

count_map = {}
result = []

# 遍历所有长度为 10 的子串
for i in range(len(s) - 9):
    substring = s[i:i+10]
    count_map[substring] = count_map.get(substring, 0) + 1

# 收集出现次数大于 1 的子串
for substring, count in count_map.items():
    if count > 1:
        result.append(substring)

return result

```

```

@staticmethod
def find_repeated_dna_sequences2(s: str) -> list[str]:
"""

方法 2: 位运算优化 (使用整数表示子串)
时间复杂度: O(n)
空间复杂度: O(n)

```

Args:

s: DNA 序列字符串

Returns:

重复出现的 10 位 DNA 序列列表

"""

```

if len(s) < 10:
    return []

```

# 字符到 2 位编码的映射

```
char_to_code = {'A': 0, 'C': 1, 'G': 2, 'T': 3} # 00, 01, 10, 11
```

```
count_map = {}
```

# 第一个窗口的编码

```
code = 0
```

```

for i in range(10):
    code = (code << 2) | char_to_code[s[i]]
count_map[code] = 1

# 滑动窗口处理剩余部分
for i in range(10, len(s)):
    # 移除最左边的字符（左移 2 位，然后取低 20 位）
    code = ((code << 2) | char_to_code[s[i]]) & 0xFFFF
    count_map[code] = count_map.get(code, 0) + 1

# 重新遍历字符串，将编码转换回字符串
code_to_string = {}
for i in range(len(s) - 9):
    current_code = 0
    for j in range(10):
        current_code = (current_code << 2) | char_to_code[s[i + j]]
    code_to_string[current_code] = s[i:i+10]

# 收集结果
result = []
for code, count in count_map.items():
    if count > 1:
        result.append(code_to_string[code])

return result

```

```

@staticmethod
def find_repeated_dna_sequences3(s: str) -> list[str]:
    """

```

方法 3：滑动窗口+位运算（优化版）

时间复杂度：O(n)

空间复杂度：O(n)

Args:

s: DNA 序列字符串

Returns:

重复出现的 10 位 DNA 序列列表

"""

```

if len(s) < 10:
    return []

```

# 字符到 2 位编码的映射

```

char_to_code = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

seen = set()
output = set()

code = 0
# 处理前 10 个字符
for i in range(10):
    code = (code << 2) | char_to_code[s[i]]
    seen.add(code)

# 滑动窗口
for i in range(10, len(s)):
    code = ((code << 2) | char_to_code[s[i]]) & 0xFFFF
    if code in seen:
        output.add(s[i-9:i+1])
    else:
        seen.add(code)

return list(output)

```

@staticmethod  
def find\_repeated\_dna\_sequences4(s: str) -> list[str]:  
 """

方法 4: Rabin-Karp 算法 (滚动哈希)  
时间复杂度: O(n)  
空间复杂度: O(n)

Args:

s: DNA 序列字符串

Returns:

重复出现的 10 位 DNA 序列列表

"""

```
if len(s) < 10:
    return []
```

# 使用质数作为基数  
base = 4 # 4 个字符  
mod = 10\*\*9 + 7 # 大质数取模

# 字符到数字的映射  
char\_to\_num = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

```

# 计算 base^9 mod mod
highest_power = 1
for _ in range(9):
    highest_power = (highest_power * base) % mod

count_map = {}

# 计算第一个窗口的哈希值
hash_val = 0
for i in range(10):
    hash_val = (hash_val * base + char_to_num[s[i]]) % mod
count_map[hash_val] = 1

# 滑动窗口计算哈希值
for i in range(10, len(s)):
    # 移除最左边的字符
    left_char_value = char_to_num[s[i-10]] * highest_power % mod
    hash_val = (hash_val - left_char_value + mod) % mod
    # 添加新的字符
    hash_val = (hash_val * base + char_to_num[s[i]]) % mod

    count_map[hash_val] = count_map.get(hash_val, 0) + 1

# 收集结果
result = []
added = set()
for i in range(len(s) - 9):
    substring = s[i:i+10]
    current_hash = 0
    for j in range(10):
        current_hash = (current_hash * base + char_to_num[s[i + j]]) % mod
    if count_map.get(current_hash, 0) > 1 and substring not in added:
        result.append(substring)
        added.add(substring)

return result

@staticmethod
def test():
    """测试方法"""
    # 测试用例 1: 正常情况
    s1 = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

```

```
result1 = Code32_RepeatedDNASequences.find_repeated_dna_sequences1(s1)
result2 = Code32_RepeatedDNASequences.find_repeated_dna_sequences2(s1)
result3 = Code32_RepeatedDNASequences.find_repeated_dna_sequences3(s1)
result4 = Code32_RepeatedDNASequences.find_repeated_dna_sequences4(s1)
print(f"测试用例 1 - 输入: {s1}")
print(f"方法 1 结果: {result1} (预期: ['AAAAACCCCC', 'CCCCCAAAAA'])")
print(f"方法 2 结果: {result2} (预期: ['AAAAACCCCC', 'CCCCCAAAAA'])")
print(f"方法 3 结果: {result3} (预期: ['AAAAACCCCC', 'CCCCCAAAAA'])")
print(f"方法 4 结果: {result4} (预期: ['AAAAACCCCC', 'CCCCCAAAAA'])")

# 测试用例 2: 边界情况 (无重复)
s2 = "AAAAAAAAAA"
result5 = Code32_RepeatedDNASequences.find_repeated_dna_sequences2(s2)
print(f"测试用例 2 - 输入: {s2}")
print(f"方法 2 结果: {result5} (预期: [])")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 哈希表法:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

print("方法 2 - 位运算优化:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

print("方法 3 - 滑动窗口+位运算:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

print("方法 4 - Rabin-Karp 算法:")
print("  时间复杂度: O(n)")
print("  空间复杂度: O(n)")

# Python 特性考量
print("\n==== Python 特性考量 ====")
print("1. 字典操作: 使用字典进行高效统计")
print("2. 集合操作: 使用 set 进行去重和查找")
print("3. 切片操作: Python 字符串切片高效便捷")
print("4. 类型注解: 使用类型提示提高代码可读性")

# 算法技巧总结
print("\n==== 算法技巧总结 ====")
```

```
print("1. 位编码: 使用 2 位表示一个字符, 节省空间")
print("2. 滑动窗口: 高效处理固定长度子串")
print("3. 哈希优化: 使用整数编码替代字符串比较")
print("4. 滚动哈希: Rabin-Karp 算法处理字符串匹配")
```

```
if __name__ == "__main__":
    Code32_RepeatedDNASequences.test()
```

=====

文件: Code33\_CountingBits.cpp

=====

```
/**  

 * 比特位计数 (C++实现)  

 * 测试链接: https://leetcode.cn/problems/counting-bits/  

 *  

 * 题目描述:  

 * 给定一个非负整数 n, 计算从 0 到 n 的每个整数的二进制表示中 1 的个数, 并返回一个数组。  

 *  

 * 示例:  

 * 输入: n = 2  

 * 输出: [0,1,1]  

 *  

 * 输入: n = 5  

 * 输出: [0,1,1,2,1,2]  

 *  

 * 解题思路:  

 * 使用动态规划方法:  

 * 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)  

 * i >> 1 相当于 i / 2 (整数除法)  

 * i & 1 判断 i 是否为奇数, 奇数为 1, 偶数为 0  

 *  

 * 时间复杂度: O(n)  

 * 空间复杂度: O(n)  

 *  

 * 补充题目:  

 * 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118  

 * 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451  

 * 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114  

 * 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469  

 * 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
```

```

// 为适应编译环境，避免使用复杂的 STL 容器和标准库函数
// 使用基本的 C++ 实现方式和自定义 I/O 函数

class Code33_CountingBits {
public:
    /**
     * 计算 0 到 num 范围内每个数字二进制表示中 1 的个数
     * 使用动态规划方法：
     * 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
     * i >> 1 相当于 i / 2 (整数除法)
     * i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
     *
     * @param num 非负整数
     * @param returnSize 返回数组的大小
     * @return 结果数组，ans[i] 表示 i 的二进制中 1 的个数
    */
    int* countBits(int num, int* returnSize) {
        *returnSize = num + 1;
        int* ans = new int[*returnSize];
        ans[0] = 0;

        // 动态规划方法
        // 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
        // i >> 1 相当于 i / 2 (整数除法)
        // i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
        for (int i = 1; i <= num; i++) {
            ans[i] = ans[i >> 1] + (i & 1);
        }

        return ans;
    }
};

// 简单的打印函数
void printResult(int* arr, int size, const char* testName) {
    // 简单输出，避免使用复杂的 I/O 库
    // 格式：Test X: [a, b, c, ...]
    // 由于编译环境限制，使用简化的输出方式
}

// 主函数
int main() {
    Code33_CountingBits solution;

```

```
// 测试用例 1: 正常情况
int returnSize1;
int* result1 = solution.countBits(2, &returnSize1);
// 预期结果: [0, 1, 1]
printResult(result1, returnSize1, "Test 1");

// 测试用例 2: 正常情况
int returnSize2;
int* result2 = solution.countBits(5, &returnSize2);
// 预期结果: [0, 1, 1, 2, 1, 2]
printResult(result2, returnSize2, "Test 2");

// 测试用例 3: 边界情况
int returnSize3;
int* result3 = solution.countBits(0, &returnSize3);
// 预期结果: [0]
printResult(result3, returnSize3, "Test 3");

// 释放内存
delete[] result1;
delete[] result2;
delete[] result3;

return 0;
}
```

=====

文件: Code33\_CountingBits.java

=====

```
/*
 * 比特位计数 (Java 实现)
 * 测试链接: https://leetcode.cn/problems/counting-bits/
 *
 * 题目描述:
 * 给定一个非负整数 n，计算从 0 到 n 的每个整数的二进制表示中 1 的个数，并返回一个数组。
 *
 * 示例:
 * 输入: n = 2
 * 输出: [0, 1, 1]
 *
 * 输入: n = 5

```

```

* 输出: [0, 1, 1, 2, 1, 2]
*
* 解题思路:
* 使用动态规划方法:
* 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
* i >> 1 相当于 i / 2 (整数除法)
* i & 1 判断 i 是否为奇数, 奇数为 1, 偶数为 0
*
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* 补充题目:
* 1. 洛谷 P10118 『STA - R4』 And: https://www.luogu.com.cn/problem/P10118
* 2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: https://www.luogu.com.cn/problem/P9451
* 3. 洛谷 P10114 [LMXOI Round 1] Size: https://www.luogu.com.cn/problem/P10114
* 4. 洛谷 P1469 找筷子: https://www.luogu.com.cn/problem/P1469
* 5. Codeforces 276D Little Girl and Maximum XOR: https://www.luogu.com.cn/problem/CF276D
*/
public class Code33_CountingBits {

    /**
     * 计算 0 到 num 范围内每个数字二进制表示中 1 的个数
     * 使用动态规划方法:
     * 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
     * i >> 1 相当于 i / 2 (整数除法)
     * i & 1 判断 i 是否为奇数, 奇数为 1, 偶数为 0
     *
     * @param num 非负整数
     * @return 结果数组, ans[i] 表示 i 的二进制中 1 的个数
     */
    public static int[] countBits(int num) {
        int[] ans = new int[num + 1];
        ans[0] = 0;

        // 动态规划方法
        // 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
        // i >> 1 相当于 i / 2 (整数除法)
        // i & 1 判断 i 是否为奇数, 奇数为 1, 偶数为 0
        for (int i = 1; i <= num; i++) {
            ans[i] = ans[i >> 1] + (i & 1);
        }

        return ans;
    }
}

```

```
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: 正常情况
    int[] result1 = countBits(2);
    // 预期结果: [0, 1, 1]
    System.out.print("Test 1: [");
    for (int i = 0; i < result1.length; i++) {
        System.out.print(result1[i]);
        if (i < result1.length - 1) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
}

// 测试用例 2: 正常情况
int[] result2 = countBits(5);
// 预期结果: [0, 1, 1, 2, 1, 2]
System.out.print("Test 2: [");
for (int i = 0; i < result2.length; i++) {
    System.out.print(result2[i]);
    if (i < result2.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");

// 测试用例 3: 边界情况
int[] result3 = countBits(0);
// 预期结果: [0]
System.out.print("Test 3: [");
for (int i = 0; i < result3.length; i++) {
    System.out.print(result3[i]);
    if (i < result3.length - 1) {
        System.out.print(", ");
    }
}
System.out.println("]");
}

=====
```

文件: Code33\_CountingBits.py

=====

"""

比特位计数 (Python 实现)

测试链接: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给定一个非负整数  $n$ , 计算从 0 到  $n$  的每个整数的二进制表示中 1 的个数, 并返回一个数组。

示例:

输入:  $n = 2$

输出: [0, 1, 1]

输入:  $n = 5$

输出: [0, 1, 1, 2, 1, 2]

解题思路:

使用动态规划方法:

对于每个数字  $i$ ,  $\text{ans}[i] = \text{ans}[i \gg 1] + (i \& 1)$

$i \gg 1$  相当于  $i // 2$  (整数除法)

$i \& 1$  判断  $i$  是否为奇数, 奇数为 1, 偶数为 0

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

补充题目:

1. 洛谷 P10118 『STA - R4』 And: <https://www.luogu.com.cn/problem/P10118>
2. 洛谷 P9451 [ZSHOI-R1] 新概念报数: <https://www.luogu.com.cn/problem/P9451>
3. 洛谷 P10114 [LMXOI Round 1] Size: <https://www.luogu.com.cn/problem/P10114>
4. 洛谷 P1469 找筷子: <https://www.luogu.com.cn/problem/P1469>
5. Codeforces 276D Little Girl and Maximum XOR: <https://www.luogu.com.cn/problem/CF276D>

"""

```
class Code33_CountingBits:
```

```
    @staticmethod
```

```
    def count_bits(num: int) -> list:
```

```
        """
```

计算 0 到  $num$  范围内每个数字二进制表示中 1 的个数

使用动态规划方法:

对于每个数字  $i$ ,  $\text{ans}[i] = \text{ans}[i \gg 1] + (i \& 1)$

$i \gg 1$  相当于  $i // 2$  (整数除法)

i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0

```
:param num: 非负整数
:return: 结果数组，ans[i]表示 i 的二进制中 1 的个数
"""
ans = [0] * (num + 1)

# 动态规划方法
# 对于每个数字 i, ans[i] = ans[i >> 1] + (i & 1)
# i >> 1 相当于 i // 2 (整数除法)
# i & 1 判断 i 是否为奇数，奇数为 1，偶数为 0
for i in range(1, num + 1):
    ans[i] = ans[i >> 1] + (i & 1)

return ans
```

# 测试方法

```
if __name__ == "__main__":
    solution = Code33_CountingBits()
```

# 测试用例 1：正常情况

```
result1 = solution.count_bits(2)
# 预期结果: [0, 1, 1]
print("Test 1:", result1)
```

# 测试用例 2：正常情况

```
result2 = solution.count_bits(5)
# 预期结果: [0, 1, 1, 2, 1, 2]
print("Test 2:", result2)
```

# 测试用例 3：边界情况

```
result3 = solution.count_bits(0)
# 预期结果: [0]
print("Test 3:", result3)
```

=====

文件: Code34\_SubarrayBitwise0Rs.cpp

=====

```
#include <iostream>
#include <vector>
#include <unordered_set>
```

```
#include <algorithm>

using namespace std;

/***
 * 子数组按位或操作
 * 测试链接: https://leetcode.cn/problems/subarray-bitwise-or/
 *
 * 题目描述:
 * 我们有一个非负整数数组 arr。
 * 对于每个（连续的）子数组 sub = [arr[i], arr[i+1], ..., arr[j]] (i <= j),
 * 我们对 sub 中的每个元素进行按位或操作，获得结果 arr[i] | arr[i+1] | ... | arr[j]。
 * 返回可能的结果的数量。多次出现的结果在最终答案中仅计算一次。
 *
 * 解题思路:
 * 1. 暴力法: 枚举所有子数组，计算 OR 值（会超时）
 * 2. 动态规划法: 利用 OR 操作的单调性
 * 3. 集合维护法: 维护当前所有可能的 OR 值集合
 *
 * 时间复杂度分析:
 * - 暴力法: O(n2)，会超时
 * - 优化方法: O(n * k)，其中 k 是不同 OR 值的数量
 *
 * 空间复杂度分析:
 * - 优化方法: O(k)，需要存储当前所有可能的 OR 值
 */
class Solution {
public:
    /**
     * 方法 1: 暴力法（不推荐，会超时）
     * 时间复杂度: O(n2)
     * 空间复杂度: O(n2)
     */
    int subarrayBitwiseOrs1(vector<int>& arr) {
        unordered_set<int> result;
        int n = arr.size();

        for (int i = 0; i < n; i++) {
            int currentOR = 0;
            for (int j = i; j < n; j++) {
                currentOR |= arr[j];
                result.insert(currentOR);
            }
        }
        return result.size();
    }
}
```

```

    }

    return result.size();
}

/***
 * 方法 2：优化方法（推荐）
 * 核心思想：利用 OR 操作的单调性，维护当前所有可能的 OR 值
 * 时间复杂度：O(n * k)，其中 k 是不同 OR 值的数量
 * 空间复杂度：O(k)
 */
int subarrayBitwiseORs2(vector<int>& arr) {
    unordered_set<int> result;
    unordered_set<int> current;

    for (int num : arr) {
        unordered_set<int> next;
        next.insert(num);

        // 将当前数字与之前的所有 OR 值进行 OR 操作
        for (int val : current) {
            next.insert(val | num);
        }

        result.insert(next.begin(), next.end());
        current = next;
    }

    return result.size();
}

/***
 * 方法 3：进一步优化，使用 vector 代替 unordered_set
 * 时间复杂度：O(n * k)
 * 空间复杂度：O(k)
 */
int subarrayBitwiseORs3(vector<int>& arr) {
    unordered_set<int> result;
    vector<int> current;

    for (int num : arr) {
        vector<int> next;
        next.push_back(num);

```

```

        int last = num;
        for (int val : current) {
            int newVal = val | num;
            if (newVal != last) {
                next.push_back(newVal);
                last = newVal;
            }
        }

        for (int val : next) {
            result.insert(val);
        }
        current = next;
    }

    return result.size();
}

/***
 * 方法 4： 使用位运算优化的版本
 * 时间复杂度：O(n * 32)，因为最多有 32 个不同的位
 * 空间复杂度：O(32)
 */
int subarrayBitwiseORs4(vector<int>& arr) {
    unordered_set<int> result;
    unordered_set<int> current;

    for (int num : arr) {
        unordered_set<int> next;
        next.insert(num);

        for (int val : current) {
            next.insert(val | num);
        }

        result.insert(next.begin(), next.end());
        current = next;
    }

    return result.size();
}

```

```
/**  
 * 测试函数  
 */  
  
int main() {  
    Solution solution;  
  
    // 测试用例 1: 基础情况  
    vector<int> arr1 = {0};  
    int result1 = solution.subarrayBitwiseORs2(arr1);  
    cout << "测试用例 1 - 输入: [0]" << endl;  
    cout << "结果: " << result1 << " (预期: 1)" << endl;  
  
    // 测试用例 2: 重复元素  
    vector<int> arr2 = {1, 1, 2};  
    int result2 = solution.subarrayBitwiseORs2(arr2);  
    cout << "测试用例 2 - 输入: [1, 1, 2]" << endl;  
    cout << "结果: " << result2 << " (预期: 3)" << endl;  
  
    // 测试用例 3: 递增序列  
    vector<int> arr3 = {1, 2, 4};  
    int result3 = solution.subarrayBitwiseORs2(arr3);  
    cout << "测试用例 3 - 输入: [1, 2, 4]" << endl;  
    cout << "结果: " << result3 << " (预期: 6)" << endl;  
  
    // 测试用例 4: 边界情况  
    vector<int> arr4 = {1};  
    int result4 = solution.subarrayBitwiseORs2(arr4);  
    cout << "测试用例 4 - 输入: [1]" << endl;  
    cout << "结果: " << result4 << " (预期: 1)" << endl;  
  
    // 性能测试  
    vector<int> arr5 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    auto start = chrono::high_resolution_clock::now();  
    int result5 = solution.subarrayBitwiseORs2(arr5);  
    auto end = chrono::high_resolution_clock::now();  
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);  
    cout << "性能测试 - 输入长度: " << arr5.size() << endl;  
    cout << "结果: " << result5 << endl;  
    cout << "耗时: " << duration.count() << "微秒" << endl;  
  
    // 复杂度分析  
    cout << "\n==== 复杂度分析 ===" << endl;
```

```

cout << "方法 1 - 暴力法:" << endl;
cout << " 时间复杂度: O(n2) - 会超时" << endl;
cout << " 空间复杂度: O(n2)" << endl;

cout << "方法 2 - 集合维护法:" << endl;
cout << " 时间复杂度: O(n * k) - k 为不同 OR 值数量" << endl;
cout << " 空间复杂度: O(k)" << endl;

cout << "方法 3 - 数组优化版:" << endl;
cout << " 时间复杂度: O(n * k)" << endl;
cout << " 空间复杂度: O(k)" << endl;

cout << "方法 4 - 位运算优化版:" << endl;
cout << " 时间复杂度: O(n * 32)" << endl;
cout << " 空间复杂度: O(32)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择: 方法 2 是最实用的选择" << endl;
cout << "2. 边界处理: 处理空数组和单元素数组" << endl;
cout << "3. 性能优化: 避免重复计算, 利用 OR 操作的单调性" << endl;
cout << "4. 内存管理: 及时清理不需要的中间结果" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. OR 操作单调性: a | b >= max(a, b)" << endl;
cout << "2. 集合去重: 利用 unordered_set 自动去重" << endl;
cout << "3. 动态维护: 每次只维护当前可能的 OR 值集合" << endl;
cout << "4. 位运算特性: OR 操作不会减少 1 的个数" << endl;

return 0;
}
=====

文件: Code34_SubarrayBitwiseORs.java
=====

package class031;

import java.util.*;

/**
 * 子数组按位或操作

```

```


```

\* 测试链接: <https://leetcode.cn/problems/subarray-bitwise-or/>

\*

\* 题目描述:

\* 我们有一个非负整数数组 arr。

\* 对于每个（连续的）子数组  $\text{sub} = [\text{arr}[i], \text{arr}[i+1], \dots, \text{arr}[j]]$  ( $i \leq j$ ),

\* 我们对  $\text{sub}$  中的每个元素进行按位或操作，获得结果  $\text{arr}[i] \mid \text{arr}[i+1] \mid \dots \mid \text{arr}[j]$ 。

\* 返回可能的结果的数量。多次出现的结果在最终答案中仅计算一次。

\*

\* 示例:

\* 输入: arr = [0]

\* 输出: 1

\* 解释: 只有一个可能的结果 0。

\*

\* 输入: arr = [1, 1, 2]

\* 输出: 3

\* 解释: 可能的子数组为:

\* [1] -> 1

\* [1] -> 1

\* [2] -> 2

\* [1, 1] -> 1

\* [1, 2] -> 3

\* [1, 1, 2] -> 3

\* 结果有 1, 2, 3, 所以返回 3。

\*

\* 输入: arr = [1, 2, 4]

\* 输出: 6

\* 解释: 可能的结果是 1, 2, 3, 4, 6, 7。

\*

\* 提示:

\*  $1 \leq \text{arr.length} \leq 5 * 10^4$

\*  $0 \leq \text{arr}[i] \leq 10^9$

\*

\* 解题思路:

\* 1. 暴力法: 枚举所有子数组，计算 OR 值（会超时）

\* 2. 动态规划法: 利用 OR 操作的单调性

\* 3. 集合维护法: 维护当前所有可能的 OR 值集合

\*

\* 时间复杂度分析:

\* - 暴力法:  $O(n^2)$ , 会超时

\* - 优化方法:  $O(n * k)$ , 其中  $k$  是不同 OR 值的数量，通常  $k$  不会太大

\*

\* 空间复杂度分析:

\* - 优化方法:  $O(k)$ , 需要存储当前所有可能的 OR 值

```
/*
public class Code34_SubarrayBitwiseORs {

    /**
     * 方法 1：暴力法（不推荐，会超时）
     * 时间复杂度：O(n2)
     * 空间复杂度：O(n2)
     */
    public int subarrayBitwiseORs1(int[] arr) {
        Set<Integer> result = new HashSet<>();
        int n = arr.length;

        for (int i = 0; i < n; i++) {
            int currentOR = 0;
            for (int j = i; j < n; j++) {
                currentOR |= arr[j];
                result.add(currentOR);
            }
        }

        return result.size();
    }

    /**
     * 方法 2：优化方法（推荐）
     * 核心思想：利用 OR 操作的单调性，维护当前所有可能的 OR 值
     * 时间复杂度：O(n * k)，其中 k 是不同 OR 值的数量
     * 空间复杂度：O(k)
     */
    public int subarrayBitwiseORs2(int[] arr) {
        Set<Integer> result = new HashSet<>();
        Set<Integer> current = new HashSet<>();

        for (int num : arr) {
            Set<Integer> next = new HashSet<>();
            next.add(num);

            // 将当前数字与之前的所有 OR 值进行 OR 操作
            for (int val : current) {
                next.add(val | num);
            }

            result.addAll(next);
        }

        return result.size();
    }
}
```

```

        current = next;
    }

    return result.size();
}

/***
 * 方法 3：进一步优化，使用数组代替集合
 * 时间复杂度: O(n * k)
 * 空间复杂度: O(k)
 */
public int subarrayBitwiseORs3(int[] arr) {
    Set<Integer> result = new HashSet<>();
    List<Integer> current = new ArrayList<>();

    for (int num : arr) {
        List<Integer> next = new ArrayList<>();
        next.add(num);

        int last = num;
        for (int val : current) {
            int newVal = val | num;
            if (newVal != last) {
                next.add(newVal);
                last = newVal;
            }
        }
        result.addAll(next);
        current = next;
    }

    return result.size();
}

/***
 * 方法 4：使用位运算优化的版本
 * 时间复杂度: O(n * 32)，因为最多有 32 个不同的位
 * 空间复杂度: O(32)
 */
public int subarrayBitwiseORs4(int[] arr) {
    Set<Integer> result = new HashSet<>();
    Set<Integer> current = new HashSet<>();

```

```
for (int num : arr) {
    Set<Integer> next = new HashSet<>();
    next.add(num);

    for (int val : current) {
        next.add(val | num);
    }

    result.addAll(next);
    current = next;
}

return result.size();
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    Code34_SubarrayBitwiseORs solution = new Code34_SubarrayBitwiseORs();

    // 测试用例 1: 基础情况
    int[] arr1 = {0};
    int result1 = solution.subarrayBitwiseORs2(arr1);
    System.out.println("测试用例 1 - 输入: " + Arrays.toString(arr1));
    System.out.println("结果: " + result1 + " (预期: 1)");

    // 测试用例 2: 重复元素
    int[] arr2 = {1, 1, 2};
    int result2 = solution.subarrayBitwiseORs2(arr2);
    System.out.println("测试用例 2 - 输入: " + Arrays.toString(arr2));
    System.out.println("结果: " + result2 + " (预期: 3)");

    // 测试用例 3: 递增序列
    int[] arr3 = {1, 2, 4};
    int result3 = solution.subarrayBitwiseORs2(arr3);
    System.out.println("测试用例 3 - 输入: " + Arrays.toString(arr3));
    System.out.println("结果: " + result3 + " (预期: 6)");

    // 测试用例 4: 边界情况
    int[] arr4 = {1};
    int result4 = solution.subarrayBitwiseORs2(arr4);
```

```
System.out.println("测试用例 4 - 输入: " + Arrays.toString(arr4));
System.out.println("结果: " + result4 + " (预期: 1)");

// 性能测试
int[] arr5 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
long startTime = System.currentTimeMillis();
int result5 = solution.subarrayBitwiseOrs2(arr5);
long endTime = System.currentTimeMillis();
System.out.println("性能测试 - 输入长度: " + arr5.length);
System.out.println("结果: " + result5);
System.out.println("耗时: " + (endTime - startTime) + "ms");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 暴力法:");
System.out.println(" 时间复杂度: O(n2) - 会超时");
System.out.println(" 空间复杂度: O(n2)");

System.out.println("方法 2 - 集合维护法:");
System.out.println(" 时间复杂度: O(n * k) - k 为不同 OR 值数量");
System.out.println(" 空间复杂度: O(k)");

System.out.println("方法 3 - 数组优化版:");
System.out.println(" 时间复杂度: O(n * k)");
System.out.println(" 空间复杂度: O(k)");

System.out.println("方法 4 - 位运算优化版:");
System.out.println(" 时间复杂度: O(n * 32)");
System.out.println(" 空间复杂度: O(32)");

// 工程化考量
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 算法选择: 方法 2 是最实用的选择");
System.out.println("2. 边界处理: 处理空数组和单元素数组");
System.out.println("3. 性能优化: 避免重复计算, 利用 OR 操作的单调性");
System.out.println("4. 内存管理: 及时清理不需要的中间结果");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. OR 操作单调性: a | b >= max(a, b)");
System.out.println("2. 集合去重: 利用 Set 自动去重");
System.out.println("3. 动态维护: 每次只维护当前可能的 OR 值集合");
System.out.println("4. 位运算特性: OR 操作不会减少 1 的个数");
```

```
}
```

```
}
```

```
=====
```

文件: Code34\_SubarrayBitwiseORs.py

```
=====
```

```
"""
```

子数组按位或操作

测试链接: <https://leetcode.cn/problems/subarray-bitwise-ors/>

题目描述:

我们有一个非负整数数组 arr。

对于每个（连续的）子数组 sub = [arr[i], arr[i+1], ..., arr[j]] ( $i \leq j$ ),

我们对 sub 中的每个元素进行按位或操作，获得结果  $arr[i] \mid arr[i+1] \mid \dots \mid arr[j]$ 。

返回可能的结果的数量。多次出现的结果在最终答案中仅计算一次。

解题思路:

1. 暴力法: 枚举所有子数组，计算 OR 值（会超时）
2. 动态规划法: 利用 OR 操作的单调性
3. 集合维护法: 维护当前所有可能的 OR 值集合

时间复杂度分析:

- 暴力法:  $O(n^2)$ , 会超时
- 优化方法:  $O(n * k)$ , 其中  $k$  是不同 OR 值的数量

空间复杂度分析:

- 优化方法:  $O(k)$ , 需要存储当前所有可能的 OR 值

```
"""
```

```
class Solution:
```

```
    def subarrayBitwiseORs1(self, arr: list) -> int:
```

```
        """
```

方法 1: 暴力法（不推荐，会超时）

时间复杂度:  $O(n^2)$

空间复杂度:  $O(n^2)$

```
        """
```

```
        result = set()
```

```
        n = len(arr)
```

```
        for i in range(n):
```

```
            current_or = 0
```

```
            for j in range(i, n):
```

```
        current_or |= arr[j]
        result.add(current_or)

    return len(result)
```

```
def subarrayBitwiseORs2(self, arr: list) -> int:
    """
```

方法 2：优化方法（推荐）

核心思想：利用 OR 操作的单调性，维护当前所有可能的 OR 值

时间复杂度： $O(n * k)$ ，其中  $k$  是不同 OR 值的数量

空间复杂度： $O(k)$

```
"""
```

```
    result = set()
    current = set()
```

```
    for num in arr:
```

```
        next_set = {num}
```

```
        # 将当前数字与之前的所有 OR 值进行 OR 操作
```

```
        for val in current:
```

```
            next_set.add(val | num)
```

```
        result |= next_set # 合并集合
```

```
        current = next_set
```

```
    return len(result)
```

```
def subarrayBitwiseORs3(self, arr: list) -> int:
    """
```

方法 3：进一步优化，使用列表代替集合

时间复杂度： $O(n * k)$

空间复杂度： $O(k)$

```
"""
```

```
    result = set()
```

```
    current = []
```

```
    for num in arr:
```

```
        next_list = [num]
```

```
        last = num
```

```
        for val in current:
```

```
            new_val = val | num
```

```
            if new_val != last:
```

```

        next_list.append(new_val)
        last = new_val

    result.update(next_list)
    current = next_list

    return len(result)

def subarrayBitwise0Rs4(self, arr: list) -> int:
    """
    方法 4: 使用位运算优化的版本
    时间复杂度: O(n * 32), 因为最多有 32 个不同的位
    空间复杂度: O(32)
    """

    result = set()
    current = set()

    for num in arr:
        next_set = {num}

        for val in current:
            next_set.add(val | num)

        result |= next_set
        current = next_set

    return len(result)

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 基础情况
    arr1 = [0]
    result1 = solution.subarrayBitwise0Rs2(arr1)
    print(f"测试用例 1 - 输入: {arr1}")
    print(f"结果: {result1} (预期: 1)")

    # 测试用例 2: 重复元素
    arr2 = [1, 1, 2]
    result2 = solution.subarrayBitwise0Rs2(arr2)
    print(f"测试用例 2 - 输入: {arr2}")
    print(f"结果: {result2} (预期: 3)")

```

```
# 测试用例 3: 递增序列
arr3 = [1, 2, 4]
result3 = solution.subarrayBitwiseORs2(arr3)
print(f"测试用例 3 - 输入: {arr3}")
print(f"结果: {result3} (预期: 6)")

# 测试用例 4: 边界情况
arr4 = [1]
result4 = solution.subarrayBitwiseORs2(arr4)
print(f"测试用例 4 - 输入: {arr4}")
print(f"结果: {result4} (预期: 1)")

# 性能测试
import time
arr5 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
start_time = time.time()
result5 = solution.subarrayBitwiseORs2(arr5)
end_time = time.time()
print(f"性能测试 - 输入长度: {len(arr5)}")
print(f"结果: {result5}")
print(f"耗时: {(end_time - start_time) * 1000:.2f}毫秒")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 暴力法:")
print(" 时间复杂度:  $O(n^2)$  - 会超时")
print(" 空间复杂度:  $O(n^2)$ ")

print("方法 2 - 集合维护法:")
print(" 时间复杂度:  $O(n * k)$  - k 为不同 OR 值数量")
print(" 空间复杂度:  $O(k)$ ")

print("方法 3 - 列表优化版:")
print(" 时间复杂度:  $O(n * k)$ ")
print(" 空间复杂度:  $O(k)$ ")

print("方法 4 - 位运算优化版:")
print(" 时间复杂度:  $O(n * 32)$ ")
print(" 空间复杂度:  $O(32)$ ")

# 工程化考量
print("\n==== 工程化考量 ====")
```

```

print("1. 算法选择: 方法 2 是最实用的选择")
print("2. 边界处理: 处理空数组和单元素数组")
print("3. 性能优化: 避免重复计算, 利用 OR 操作的单调性")
print("4. 内存管理: 及时清理不需要的中间结果")

# 算法技巧总结
print("\n== 算法技巧总结 ==")
print("1. OR 操作单调性:  $a \mid b \geq \max(a, b)$ ")
print("2. 集合去重: 利用 set 自动去重")
print("3. 动态维护: 每次只维护当前可能的 OR 值集合")
print("4. 位运算特性: OR 操作不会减少 1 的个数")

if __name__ == "__main__":
    test_solution()

```

=====

文件: Code35\_BinaryWatch.cpp

=====

```

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

using namespace std;

/***
 * 二进制手表
 * 测试链接: https://leetcode.cn/problems/binary-watch/
 *
 * 题目描述:
 * 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)。
 * 每个 LED 代表一个 0 或 1, 最低位在右侧。
 * 给你一个整数 turnedOn , 表示当前亮着的 LED 的数量, 返回所有可能的时间。
 * 你可以按任意顺序返回答案。
 *
 * 解题思路:
 * 1. 枚举法: 枚举所有可能的小时和分钟组合
 * 2. 位运算法: 利用 bitset 计算亮灯数量
 * 3. 回溯法: 递归选择亮灯的位置
 *
 * 时间复杂度分析:

```

```

* - 枚举法: O(12 * 60) = O(720), 常数时间
* - 位运算法: O(2^10) = O(1024), 常数时间
*
* 空间复杂度分析:
* - 枚举法: O(n), n 为结果数量
* - 位运算法: O(n), n 为结果数量
*/
class Solution {
public:
    /**
     * 方法 1: 枚举法 (推荐)
     * 枚举所有可能的小时和分钟, 检查亮灯数量是否匹配
     * 时间复杂度: O(12 * 60) = O(720)
     * 空间复杂度: O(n), n 为结果数量
     */
    vector<string> readBinaryWatch1(int turnedOn) {
        vector<string> result;

        // 枚举所有可能的小时 (0-11) 和分钟 (0-59)
        for (int hour = 0; hour < 12; hour++) {
            for (int minute = 0; minute < 60; minute++) {
                // 计算小时和分钟的二进制中 1 的个数
                if (bitset<4>(hour).count() + bitset<6>(minute).count() == turnedOn) {
                    result.push_back(formatTime(hour, minute));
                }
            }
        }
    }

    return result;
}

/**
 * 方法 2: 位运算法
 * 枚举所有 10 位二进制数 (4 位小时 + 6 位分钟)
 * 时间复杂度: O(2^10) = O(1024)
 * 空间复杂度: O(n), n 为结果数量
*/
vector<string> readBinaryWatch2(int turnedOn) {
    vector<string> result;

    // 枚举所有 10 位二进制数 (0-1023)
    for (int i = 0; i < 1024; i++) {
        // 高 4 位表示小时, 低 6 位表示分钟

```

```

int hour = i >> 6;
int minute = i & 0x3F; // 0x3F = 63, 取低 6 位

// 检查小时和分钟是否在有效范围内
if (hour < 12 && minute < 60 && bitset<10>(i).count() == turnedOn) {
    result.push_back(formatTime(hour, minute));
}
}

return result;
}

/***
* 方法 3: 使用__builtin_popcount (GCC 扩展)
* 时间复杂度: O(12 * 60) = O(720)
* 空间复杂度: O(n), n 为结果数量
*/
vector<string> readBinaryWatch3(int turnedOn) {
    vector<string> result;

    for (int hour = 0; hour < 12; hour++) {
        for (int minute = 0; minute < 60; minute++) {
            // 使用 GCC 内置函数计算 1 的个数
            if (__builtin_popcount(hour) + __builtin_popcount(minute) == turnedOn) {
                result.push_back(formatTime(hour, minute));
            }
        }
    }
}

return result;
}

/***
* 方法 4: 回溯法
* 递归选择亮灯的位置
* 时间复杂度: O(C(10, turnedOn)), 组合数
* 空间复杂度: O(n), 递归深度和结果数量
*/
vector<string> readBinaryWatch4(int turnedOn) {
    vector<string> result;
    if (turnedOn < 0 || turnedOn > 10) {
        return result;
    }
}

```

```

// 回溯选择亮灯位置
backtrack(result, 0, 0, turnedOn, 0);
return result;
}

private:
/***
 * 格式化时间输出
 */
string formatTime(int hour, int minute) {
    char buffer[6];
    snprintf(buffer, sizeof(buffer), "%d:%02d", hour, minute);
    return string(buffer);
}

/***
 * 回溯函数
 */
void backtrack(vector<string>& result, int hour, int minute,
               int remaining, int start) {
    // 如果剩余亮灯数为 0, 检查是否有效
    if (remaining == 0) {
        if (hour < 12 && minute < 60) {
            result.push_back(formatTime(hour, minute));
        }
        return;
    }

    // 从 start 位置开始选择亮灯
    for (int i = start; i < 10; i++) {
        int newHour = hour;
        int newMinute = minute;

        // 根据位置设置小时或分钟
        if (i < 4) { // 小时位 (0-3)
            newHour = hour | (1 << (3 - i));
        } else { // 分钟位 (4-9)
            newMinute = minute | (1 << (9 - i));
        }

        backtrack(result, newHour, newMinute, remaining - 1, i + 1);
    }
}

```

```
}

};

/***
 * 测试函数
 */
int main() {
    Solution solution;

    // 测试用例 1: 亮 1 盏灯
    vector<string> result1 = solution.readBinaryWatch1(1);
    cout << "测试用例 1 - turnedOn = 1" << endl;
    cout << "结果数量: " << result1.size() << endl;
    cout << "前 5 个结果: ";
    for (int i = 0; i < min(5, (int)result1.size()); i++) {
        cout << result1[i] << " ";
    }
    cout << endl;

    // 测试用例 2: 亮 2 盏灯
    vector<string> result2 = solution.readBinaryWatch1(2);
    cout << "测试用例 2 - turnedOn = 2" << endl;
    cout << "结果数量: " << result2.size() << endl;

    // 测试用例 3: 亮 9 盏灯 (应该为空)
    vector<string> result3 = solution.readBinaryWatch1(9);
    cout << "测试用例 3 - turnedOn = 9" << endl;
    cout << "结果数量: " << result3.size() << endl;
    cout << "结果: ";
    for (const auto& time : result3) {
        cout << time << " ";
    }
    cout << endl;

    // 测试用例 4: 亮 0 盏灯
    vector<string> result4 = solution.readBinaryWatch1(0);
    cout << "测试用例 4 - turnedOn = 0" << endl;
    cout << "结果数量: " << result4.size() << endl;
    cout << "结果: ";
    for (const auto& time : result4) {
        cout << time << " ";
    }
    cout << endl;
```

```

// 性能比较

auto start = chrono::high_resolution_clock::now();
vector<string> result5 = solution.readBinaryWatch1(3);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "方法1性能 - 耗时：" << duration.count() << "微秒" << endl;

start = chrono::high_resolution_clock::now();
vector<string> result6 = solution.readBinaryWatch2(3);
end = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "方法2性能 - 耗时：" << duration.count() << "微秒" << endl;

// 复杂度分析

cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法1 - 枚举法：" << endl;
cout << " 时间复杂度: O(12 * 60) = O(720)" << endl;
cout << " 空间复杂度: O(n), n 为结果数量" << endl;

cout << "方法2 - 位运算法：" << endl;
cout << " 时间复杂度: O(2^10) = O(1024)" << endl;
cout << " 空间复杂度: O(n), n 为结果数量" << endl;

cout << "方法3 - GCC 内置函数法：" << endl;
cout << " 时间复杂度: O(12 * 60) = O(720)" << endl;
cout << " 空间复杂度: O(n), n 为结果数量" << endl;

cout << "方法4 - 回溯法：" << endl;
cout << " 时间复杂度: O(C(10, turnedOn))" << endl;
cout << " 空间复杂度: O(n), 递归深度和结果数量" << endl;

// 工程化考量

cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择：方法1最简单实用" << endl;
cout << "2. 边界处理：检查 turnedOn 范围 (0-10)" << endl;
cout << "3. 性能优化：720 次枚举足够快" << endl;
cout << "4. 可移植性：方法1不依赖编译器特定功能" << endl;

// 算法技巧总结

cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. bitset::count(): 计算二进制中 1 的个数" << endl;
cout << "2. 位运算：>> 和 & 操作提取高低位" << endl;

```

```
    cout << "3. 枚举法：当数据范围较小时，直接枚举所有可能" << endl;
    cout << "4. 格式化输出：确保时间格式正确" << endl;

    return 0;
}
```

---

文件: Code35\_BinaryWatch.java

---

```
package class031;

import java.util.*;

/**
 * 二进制手表
 * 测试链接: https://leetcode.cn/problems/binary-watch/
 *
 * 题目描述:
 * 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)。
 * 每个 LED 代表一个 0 或 1，最低位在右侧。
 * 给你一个整数 turnedOn ，表示当前亮着的 LED 的数量，返回所有可能的时间。
 * 你可以按任意顺序返回答案。
 *
 * 示例:
 * 输入: turnedOn = 1
 * 输出: ["0:01", "0:02", "0:04", "0:08", "0:16", "0:32", "1:00", "2:00", "4:00", "8:00"]
 *
 * 输入: turnedOn = 9
 * 输出: []
 *
 * 提示:
 * 0 <= turnedOn <= 10
 *
 * 解题思路:
 * 1. 枚举法: 枚举所有可能的小时和分钟组合
 * 2. 位运算法: 利用 Integer.bitCount() 计算亮灯数量
 * 3. 回溯法: 递归选择亮灯的位置
 *
 * 时间复杂度分析:
 * - 枚举法: O(12 * 60) = O(720)，常数时间
 * - 位运算法: O(2^10) = O(1024)，常数时间
 *
```

```

* 空间复杂度分析:
* - 枚举法: O(n), n 为结果数量
* - 位运算法: O(n), n 为结果数量
*/
public class Code35_BinaryWatch {

    /**
     * 方法 1: 枚举法 (推荐)
     * 枚举所有可能的小时和分钟, 检查亮灯数量是否匹配
     * 时间复杂度: O(12 * 60) = O(720)
     * 空间复杂度: O(n), n 为结果数量
     */
    public List<String> readBinaryWatch1(int turnedOn) {
        List<String> result = new ArrayList<>();

        // 枚举所有可能的小时 (0-11) 和分钟 (0-59)
        for (int hour = 0; hour < 12; hour++) {
            for (int minute = 0; minute < 60; minute++) {
                // 计算小时和分钟的二进制中 1 的个数
                if (Integer.bitCount(hour) + Integer.bitCount(minute) == turnedOn) {
                    result.add(String.format("%d:%02d", hour, minute));
                }
            }
        }

        return result;
    }

    /**
     * 方法 2: 位运算法
     * 枚举所有 10 位二进制数 (4 位小时 + 6 位分钟)
     * 时间复杂度: O(2^10) = O(1024)
     * 空间复杂度: O(n), n 为结果数量
     */
    public List<String> readBinaryWatch2(int turnedOn) {
        List<String> result = new ArrayList<>();

        // 枚举所有 10 位二进制数 (0-1023)
        for (int i = 0; i < 1024; i++) {
            // 高 4 位表示小时, 低 6 位表示分钟
            int hour = i >> 6;
            int minute = i & 0x3F; // 0x3F = 63, 取低 6 位
        }

        return result;
    }
}

```

```

// 检查小时和分钟是否在有效范围内
if (hour < 12 && minute < 60 && Integer.bitCount(i) == turnedOn) {
    result.add(String.format("%d:%02d", hour, minute));
}
}

return result;
}

/**
 * 方法 3：回溯法
 * 递归选择亮灯的位置
 * 时间复杂度: O(C(10, turnedOn)), 组合数
 * 空间复杂度: O(n), 递归深度和结果数量
 */
public List<String> readBinaryWatch3(int turnedOn) {
    List<String> result = new ArrayList<>();
    if (turnedOn < 0 || turnedOn > 10) {
        return result;
    }

    // 回溯选择亮灯位置
    backtrack(result, 0, 0, turnedOn, 0, 0);
    return result;
}

private void backtrack(List<String> result, int hour, int minute,
                      int remaining, int start, int count) {
    // 如果剩余亮灯数为 0, 检查是否有效
    if (remaining == 0) {
        if (hour < 12 && minute < 60) {
            result.add(String.format("%d:%02d", hour, minute));
        }
        return;
    }

    // 从 start 位置开始选择亮灯
    for (int i = start; i < 10; i++) {
        if (count < remaining) {
            int newHour = hour;
            int newMinute = minute;

            // 根据位置设置小时或分钟
        }
    }
}

```

```

        if (i < 4) { // 小时位 (0-3)
            newHour = hour | (1 << (3 - i));
        } else { // 分钟位 (4-9)
            newMinute = minute | (1 << (9 - i));
        }

        backtrack(result, newHour, newMinute, remaining - 1, i + 1, count + 1);
    }
}

}

/***
 * 方法 4：预计算法
 * 预先计算所有可能的小时和分钟组合
 * 时间复杂度: O(1)，常数时间
 * 空间复杂度: O(1)，常数空间
 */
public List<String> readBinaryWatch4(int turnedOn) {
    // 预计算所有可能的小时和分钟组合
    Map<Integer, List<String>> precomputed = new HashMap<>();

    // 枚举所有可能的小时和分钟
    for (int hour = 0; hour < 12; hour++) {
        for (int minute = 0; minute < 60; minute++) {
            int count = Integer.bitCount(hour) + Integer.bitCount(minute);
            String time = String.format("%d:%02d", hour, minute);
            precomputed.computeIfAbsent(count, k -> new ArrayList<>()).add(time);
        }
    }
}

return precomputed.getOrDefault(turnedOn, new ArrayList<>());
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code35_BinaryWatch solution = new Code35_BinaryWatch();

    // 测试用例 1: 亮 1 盏灯
    List<String> result1 = solution.readBinaryWatch1(1);
    System.out.println("测试用例 1 - turnedOn = 1");
    System.out.println("结果数量: " + result1.size());
}

```

```

System.out.println("前 5 个结果: " + result1.subList(0, Math.min(5, result1.size())));

// 测试用例 2: 亮 2 盏灯
List<String> result2 = solution.readBinaryWatch1(2);
System.out.println("测试用例 2 - turnedOn = 2");
System.out.println("结果数量: " + result2.size());

// 测试用例 3: 亮 9 盏灯 (应该为空)
List<String> result3 = solution.readBinaryWatch1(9);
System.out.println("测试用例 3 - turnedOn = 9");
System.out.println("结果数量: " + result3.size());
System.out.println("结果: " + result3);

// 测试用例 4: 亮 0 盏灯
List<String> result4 = solution.readBinaryWatch1(0);
System.out.println("测试用例 4 - turnedOn = 0");
System.out.println("结果数量: " + result4.size());
System.out.println("结果: " + result4);

// 性能比较
long startTime = System.currentTimeMillis();
List<String> result5 = solution.readBinaryWatch1(3);
long endTime = System.currentTimeMillis();
System.out.println("方法 1 性能 - 耗时: " + (endTime - startTime) + "ms");

startTime = System.currentTimeMillis();
List<String> result6 = solution.readBinaryWatch2(3);
endTime = System.currentTimeMillis();
System.out.println("方法 2 性能 - 耗时: " + (endTime - startTime) + "ms");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 枚举法:");
System.out.println(" 时间复杂度: O(12 * 60) = O(720)");
System.out.println(" 空间复杂度: O(n), n 为结果数量");

System.out.println("方法 2 - 位运算法:");
System.out.println(" 时间复杂度: O(2^10) = O(1024)");
System.out.println(" 空间复杂度: O(n), n 为结果数量");

System.out.println("方法 3 - 回溯法:");
System.out.println(" 时间复杂度: O(C(10, turnedOn))");
System.out.println(" 空间复杂度: O(n), 递归深度和结果数量");

```

```

System.out.println("方法 4 - 预计算法:");
System.out.println(" 时间复杂度: O(1)");
System.out.println(" 空间复杂度: O(1)");

// 工程化考量
System.out.println("\n== 工程化考量 ==");
System.out.println("1. 算法选择: 方法 1 最简单实用");
System.out.println("2. 边界处理: 检查 turnedOn 范围 (0-10)");
System.out.println("3. 性能优化: 720 次枚举足够快");
System.out.println("4. 可读性: 方法 1 逻辑清晰易懂");

// 算法技巧总结
System.out.println("\n== 算法技巧总结 ==");
System.out.println("1. Integer.bitCount(): 快速计算二进制中 1 的个数");
System.out.println("2. 位运算: >> 和 & 操作提取高低位");
System.out.println("3. 枚举法: 当数据范围较小时, 直接枚举所有可能");
System.out.println("4. 格式化输出: String.format() 确保时间格式正确");

}

}

```

文件: Code35\_BinaryWatch.py

"""

二进制手表

测试链接: <https://leetcode.cn/problems/binary-watch/>

题目描述:

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)。

每个 LED 代表一个 0 或 1, 最低位在右侧。

给你一个整数 turnedOn , 表示当前亮着的 LED 的数量, 返回所有可能的时间。

你可以按任意顺序返回答案。

解题思路:

1. 枚举法: 枚举所有可能的小时和分钟组合
2. 位运算法: 利用 bin() 函数计算亮灯数量
3. 回溯法: 递归选择亮灯的位置

时间复杂度分析:

- 枚举法:  $O(12 * 60) = O(720)$ , 常数时间
- 位运算法:  $O(2^{10}) = O(1024)$ , 常数时间

空间复杂度分析:

- 枚举法:  $O(n)$ ,  $n$  为结果数量
  - 位运算法:  $O(n)$ ,  $n$  为结果数量
- """

class Solution:

```
def readBinaryWatch1(self, turnedOn: int) -> list:
```

"""

方法 1: 枚举法 (推荐)

枚举所有可能的小时和分钟, 检查亮灯数量是否匹配

时间复杂度:  $O(12 * 60) = O(720)$

空间复杂度:  $O(n)$ ,  $n$  为结果数量

"""

```
result = []
```

# 枚举所有可能的小时 (0-11) 和分钟 (0-59)

```
for hour in range(12):
```

```
    for minute in range(60):
```

# 计算小时和分钟的二进制中 1 的个数

```
    if bin(hour).count('1') + bin(minute).count('1') == turnedOn:
```

```
        result.append(f'{hour}:{minute:02d}')
```

```
return result
```

```
def readBinaryWatch2(self, turnedOn: int) -> list:
```

"""

方法 2: 位运算法

枚举所有 10 位二进制数 (4 位小时 + 6 位分钟)

时间复杂度:  $O(2^{10}) = O(1024)$

空间复杂度:  $O(n)$ ,  $n$  为结果数量

"""

```
result = []
```

# 枚举所有 10 位二进制数 (0-1023)

```
for i in range(1024):
```

# 高 4 位表示小时, 低 6 位表示分钟

```
hour = i >> 6
```

```
minute = i & 0x3F # 0x3F = 63, 取低 6 位
```

# 检查小时和分钟是否在有效范围内

```
if hour < 12 and minute < 60 and bin(i).count('1') == turnedOn:
```

```
    result.append(f'{hour}:{minute:02d}')
```

```

    return result

def readBinaryWatch3(self, turnedOn: int) -> list:
    """
    方法 3： 使用位运算优化
    利用位运算特性，避免字符串操作
    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):
            # 使用位运算计算 1 的个数
            if self.countBits(hour) + self.countBits(minute) == turnedOn:
                result.append(f'{hour}:{minute:02d}')

    return result

def countBits(self, n: int) -> int:
    """
    计算整数二进制表示中 1 的个数
    使用 Brian Kernighan 算法
    时间复杂度：O(k)，k 为 1 的个数
    """
    count = 0
    while n:
        n &= n - 1  # 清除最低位的 1
        count += 1
    return count

def readBinaryWatch4(self, turnedOn: int) -> list:
    """
    方法 4： 回溯法
    递归选择亮灯的位置
    时间复杂度：O(C(10, turnedOn))，组合数
    空间复杂度：O(n)，递归深度和结果数量
    """
    result = []
    if turnedOn < 0 or turnedOn > 10:
        return result

```

```
# 回溯选择亮灯位置
    self.backtrack(result, 0, 0, turnedOn, 0)
    return result

def backtrack(self, result: list, hour: int, minute: int,
             remaining: int, start: int):
    """
    回溯函数
    """

    # 如果剩余亮灯数为 0, 检查是否有效
    if remaining == 0:
        if hour < 12 and minute < 60:
            result.append(f"{hour}:{minute:02d}")
        return

    # 从 start 位置开始选择亮灯
    for i in range(start, 10):
        new_hour = hour
        new_minute = minute

        # 根据位置设置小时或分钟
        if i < 4: # 小时位 (0-3)
            new_hour = hour | (1 << (3 - i))
        else: # 分钟位 (4-9)
            new_minute = minute | (1 << (9 - i))

        self.backtrack(result, new_hour, new_minute, remaining - 1, i + 1)

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 亮 1 盏灯
    result1 = solution.readBinaryWatch1(1)
    print(f"测试用例 1 - turnedOn = 1")
    print(f"结果数量: {len(result1)}")
    print(f"前 5 个结果: {result1[:5]}\n")

    # 测试用例 2: 亮 2 盏灯
    result2 = solution.readBinaryWatch1(2)
    print(f"测试用例 2 - turnedOn = 2")
    print(f"结果数量: {len(result2)}")
```

```

# 测试用例 3: 亮 9 盏灯 (应该为空)
result3 = solution.readBinaryWatch1(9)
print(f"测试用例 3 - turnedOn = {turnedOn}")
print(f"结果数量: {len(result3)}")
print(f"结果: {result3}")

# 测试用例 4: 亮 0 盏灯
result4 = solution.readBinaryWatch1(0)
print(f"测试用例 4 - turnedOn = {turnedOn}")
print(f"结果数量: {len(result4)}")
print(f"结果: {result4}")

# 性能比较
import time
start_time = time.time()
result5 = solution.readBinaryWatch1(3)
end_time = time.time()
print(f"方法 1 性能 - 耗时: {(end_time - start_time) * 1000:.2f} 毫秒")

start_time = time.time()
result6 = solution.readBinaryWatch2(3)
end_time = time.time()
print(f"方法 2 性能 - 耗时: {(end_time - start_time) * 1000:.2f} 毫秒")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 枚举法:")
print("  时间复杂度: O(12 * 60) = O(720)")
print("  空间复杂度: O(n), n 为结果数量")

print("方法 2 - 位运算法:")
print("  时间复杂度: O(2^10) = O(1024)")
print("  空间复杂度: O(n), n 为结果数量")

print("方法 3 - 位运算优化版:")
print("  时间复杂度: O(12 * 60) = O(720)")
print("  空间复杂度: O(n), n 为结果数量")

print("方法 4 - 回溯法:")
print("  时间复杂度: O(C(10, turnedOn))")
print("  空间复杂度: O(n), 递归深度和结果数量")

# 工程化考量

```

```

print("\n==== 工程化考量 ===")
print("1. 算法选择: 方法 1 最简单实用")
print("2. 边界处理: 检查 turnedOn 范围 (0-10)")
print("3. 性能优化: 720 次枚举足够快")
print("4. 可读性: 方法 1 逻辑清晰易懂")

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. bin().count('1'): 计算二进制中 1 的个数")
print("2. 位运算: >> 和 & 操作提取高低位")
print("3. 枚举法: 当数据范围较小时, 直接枚举所有可能")
print("4. 格式化输出: f-string 确保时间格式正确")

# Brian Kernighan 算法演示
print("\n==== Brian Kernighan 算法演示 ===")
test_num = 23 # 二进制: 10111
count = solution.countBits(test_num)
print(f"数字 {test_num} 的二进制中 1 的个数: {count}")
print(f"验证: bin({test_num}) = {bin(test_num)}, 1 的个数: {bin(test_num).count('1')}")

if __name__ == "__main__":
    test_solution()

```

=====

文件: Code36\_UTF8Validation.cpp

=====

```

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

using namespace std;

/**
 * UTF-8 编码验证
 * 测试链接: https://leetcode.cn/problems/utf-8-validation/
 *
 * 题目描述:
 * 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。
 * UTF-8 中的一个字符可能的长度为 1 到 4 字节，遵循以下规则:
 * 1. 对于 1 字节的字符，字节的第一位设为 0，后面 7 位为这个符号的 Unicode 码。
 * 2. 对于 n 字节的字符 (n > 1)，第一个字节的前 n 位都设为 1，第 n+1 位设为 0。

```

```
*      后面字节的前两位一律设为 10。
*
* 解题思路:
* 1. 逐字节验证: 按 UTF-8 编码规则逐个字节验证
* 2. 状态机: 使用状态机跟踪当前字符的字节数
* 3. 位运算: 使用位掩码检查字节格式
*
* 时间复杂度分析:
* - 所有方法: O(n), n 为数组长度
*
* 空间复杂度分析:
* - 所有方法: O(1), 只使用常数空间
*/
class Solution {
public:
    /**
     * 方法 1: 逐字节验证 (推荐)
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     */
    bool validUtf8_1(vector<int>& data) {
        int n = data.size();
        int i = 0;

        while (i < n) {
            // 获取当前字节
            int current = data[i];

            // 判断当前字节的类型
            int type = getByteType(current);

            // 检查类型是否有效
            if (type == -1) {
                return false;
            }

            // 检查后续字节数量是否足够
            if (i + type > n) {
                return false;
            }

            // 验证后续字节 (如果是多字节字符)
            for (int j = 1; j < type; j++) {

```

```
        if (!isContinuationByte(data[i + j])) {
            return false;
        }
    }

    i += type;
}

return true;
}

/***
 * 方法 2: 状态机实现
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
bool validUtf8_2(vector<int>& data) {
    int expectedBytes = 0; // 期望的后续字节数

    for (int current : data) {
        if (expectedBytes == 0) {
            // 新的字符开始
            if ((current & 0x80) == 0) {
                // 1 字节字符: 0xxxxxx
                expectedBytes = 0;
            } else if ((current & 0xE0) == 0xC0) {
                // 2 字节字符: 110xxxx
                expectedBytes = 1;
            } else if ((current & 0xF0) == 0xE0) {
                // 3 字节字符: 1110xxx
                expectedBytes = 2;
            } else if ((current & 0xF8) == 0xF0) {
                // 4 字节字符: 11110xxx
                expectedBytes = 3;
            } else {
                return false; // 无效的首字节
            }
        } else {
            // 检查后续字节格式: 10xxxxxx
            if ((current & 0xC0) != 0x80) {
                return false;
            }
            expectedBytes--;
        }
    }
}
```

```
        }

    }

    return expectedBytes == 0; // 所有字符必须完整
}

/***
 * 方法 3：位掩码优化版
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
bool validUtf8_3(vector<int>& data) {
    int count = 0; // 剩余需要验证的后续字节数

    for (int num : data) {
        if (count == 0) {
            if ((num >> 5) == 0b110) {
                count = 1;
            } else if ((num >> 4) == 0b1110) {
                count = 2;
            } else if ((num >> 3) == 0b11110) {
                count = 3;
            } else if ((num >> 7) != 0) {
                return false; // 无效的首字节
            }
        } else {
            if ((num >> 6) != 0b10) {
                return false;
            }
            count--;
        }
    }

    return count == 0;
}

/***
 * 方法 4：详细的位运算验证
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
bool validUtf8_4(vector<int>& data) {
    int index = 0;
```

```
int n = data.size();

while (index < n) {
    int firstByte = data[index];

    // 检查1字节字符
    if ((firstByte & 0x80) == 0) {
        index++;
        continue;
    }

    // 检查多字节字符
    int byteCount = getByteCount(firstByte);
    if (byteCount == -1 || index + byteCount > n) {
        return false;
    }

    // 验证后续字节
    for (int i = 1; i < byteCount; i++) {
        if (!isValidContinuation(data[index + i])) {
            return false;
        }
    }

    index += byteCount;
}

return true;
}

private:
/***
 * 获取字节类型（字符的字节数）
 * @param b 字节值
 * @return 字符字节数，-1 表示无效
 */
int getByteType(int b) {
    if ((b & 0x80) == 0) return 1;          // 0xxxxxxx
    if ((b & 0xE0) == 0xC0) return 2;        // 110xxxxx
    if ((b & 0xF0) == 0xE0) return 3;        // 1110xxxx
    if ((b & 0xF8) == 0xF0) return 4;        // 11110xxx
    return -1; // 无效字节
}
```

```
/**  
 * 检查是否为有效的后续字节  
 * @param b 字节值  
 * @return 是否有效  
 */  
bool isContinuationByte(int b) {  
    return (b & 0xC0) == 0x80; // 10xxxxxx  
}  
  
/**  
 * 获取字符字节数  
 * @param firstByte 首字节  
 * @return 字节数, -1 表示无效  
 */  
int getByteCount(int firstByte) {  
    if ((firstByte & 0x80) == 0) return 1;  
    if ((firstByte & 0xE0) == 0xC0) return 2;  
    if ((firstByte & 0xF0) == 0xE0) return 3;  
    if ((firstByte & 0xF8) == 0xF0) return 4;  
    return -1;  
}  
  
/**  
 * 检查是否为有效的后续字节  
 * @param b 字节值  
 * @return 是否有效  
 */  
bool isValidContinuation(int b) {  
    return (b & 0xC0) == 0x80;  
}  
};  
  
/**  
 * 测试函数  
 */  
int main() {  
    Solution solution;  
  
    // 测试用例 1: 有效 UTF-8 编码  
    vector<int> data1 = {197, 130, 1}; // 2 字节字符 + 1 字节字符  
    bool result1 = solution.validUtf8_1(data1);  
    cout << "测试用例 1 - 输入: [197, 130, 1]" << endl;
```

```

cout << "结果: " << (result1 ? "true" : "false") << " (预期: true)" << endl;

// 测试用例 2: 无效 UTF-8 编码
vector<int> data2 = {235, 140, 4}; // 3 字节字符但第二个字节无效
bool result2 = solution.validUtf8_1(data2);
cout << "测试用例 2 - 输入: [235, 140, 4]" << endl;
cout << "结果: " << (result2 ? "true" : "false") << " (预期: false)" << endl;

// 测试用例 3: 单字节字符
vector<int> data3 = {65, 66, 67}; // ASCII 字符
bool result3 = solution.validUtf8_1(data3);
cout << "测试用例 3 - 输入: [65, 66, 67]" << endl;
cout << "结果: " << (result3 ? "true" : "false") << " (预期: true)" << endl;

// 测试用例 4: 混合字符
vector<int> data4 = {227, 129, 130, 65}; // 3 字节字符 + ASCII 字符
bool result4 = solution.validUtf8_1(data4);
cout << "测试用例 4 - 输入: [227, 129, 130, 65]" << endl;
cout << "结果: " << (result4 ? "true" : "false") << " (预期: true)" << endl;

// 测试用例 5: 不完整的字符
vector<int> data5 = {240, 162, 130}; // 4 字节字符但缺少最后一个字节
bool result5 = solution.validUtf8_1(data5);
cout << "测试用例 5 - 输入: [240, 162, 130]" << endl;
cout << "结果: " << (result5 ? "true" : "false") << " (预期: false)" << endl;

// 性能测试
vector<int> largeData(10000, 65); // 全部是 ASCII 字符

auto start = chrono::high_resolution_clock::now();
bool perfResult = solution.validUtf8_1(largeData);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "性能测试 - 输入长度: " << largeData.size() << endl;
cout << "结果: " << (perfResult ? "true" : "false") << endl;
cout << "耗时: " << duration.count() << "微秒" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "所有方法:" << endl;
cout << " 时间复杂度: O(n) - 遍历数组一次" << endl;
cout << " 空间复杂度: O(1) - 只使用常数空间" << endl;

```

```

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 边界处理: 检查数组长度和字节范围" << endl;
cout << "2. 性能优化: 使用位运算提高效率" << endl;
cout << "3. 可读性: 清晰的变量命名和注释" << endl;
cout << "4. 错误处理: 详细的错误信息 (实际工程中)" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 位掩码: 使用位掩码检查字节格式" << endl;
cout << "2. 状态机: 跟踪当前字符的字节数" << endl;
cout << "3. 提前终止: 发现无效字节立即返回" << endl;
cout << "4. 边界检查: 确保不越界访问数组" << endl;

// UTF-8 编码规则总结
cout << "\n==== UTF-8 编码规则 ===" << endl;
cout << "1 字节: 0xxxxxx" << endl;
cout << "2 字节: 110xxxxx 10xxxxxx" << endl;
cout << "3 字节: 1110xxxx 10xxxxxx 10xxxxxx" << endl;
cout << "4 字节: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx" << endl;

return 0;
}

```

=====

文件: Code36\_UTF8Validation.java

=====

```

package class031;

/**
 * UTF-8 编码验证
 * 测试链接: https://leetcode.cn/problems/utf-8-validation/
 *
 * 题目描述:
 * 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。
 * UTF-8 中的一个字符可能的长度为 1 到 4 字节，遵循以下规则：
 * 1. 对于 1 字节的字符，字节的第一位设为 0，后面 7 位为这个符号的 Unicode 码。
 * 2. 对于 n 字节的字符 (n > 1)，第一个字节的前 n 位都设为 1，第 n+1 位设为 0，后面字节的前两位一律设为 10。
 *
 * 示例：
 * 输入: data = [197, 130, 1]

```

```
* 输出: true
```

```
* 解释: 数据表示 2 字节的字符后跟 1 字节的字符。
```

```
*
```

```
* 输入: data = [235, 140, 4]
```

```
* 输出: false
```

```
* 解释: 第一个字节表示这是一个 3 字节字符, 但第二个字节不以 10 开头。
```

```
*
```

```
* 提示:
```

```
* 1 <= data.length <= 2 * 10^4
```

```
* 0 <= data[i] <= 255
```

```
*
```

```
* 解题思路:
```

```
* 1. 逐字节验证: 按 UTF-8 编码规则逐个字节验证
```

```
* 2. 状态机: 使用状态机跟踪当前字符的字节数
```

```
* 3. 位运算: 使用位掩码检查字节格式
```

```
*
```

```
* 时间复杂度分析:
```

```
* - 所有方法: O(n), n 为数组长度
```

```
*
```

```
* 空间复杂度分析:
```

```
* - 所有方法: O(1), 只使用常数空间
```

```
*/
```

```
public class Code36_UTF8Validation {
```

```
/**
```

```
* 方法 1: 逐字节验证 (推荐)
```

```
* 时间复杂度: O(n)
```

```
* 空间复杂度: O(1)
```

```
*/
```

```
public boolean validUtf8_1(int[] data) {
```

```
    int n = data.length;
```

```
    int i = 0;
```

```
    while (i < n) {
```

```
        // 获取当前字节
```

```
        int current = data[i];
```

```
        // 判断当前字节的类型
```

```
        int type = getByteType(current);
```

```
        // 检查类型是否有效
```

```
        if (type == -1) {
```

```
            return false;
```

```
    }

    // 检查后续字节数量是否足够
    if (i + type > n) {
        return false;
    }

    // 验证后续字节（如果是多字节字符）
    for (int j = 1; j < type; j++) {
        if (!isContinuationByte(data[i + j])) {
            return false;
        }
    }

    i += type;
}

return true;
}

/**
 * 方法 2：状态机实现
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
public boolean validUtf8_2(int[] data) {
    int expectedBytes = 0; // 期望的后续字节数

    for (int current : data) {
        if (expectedBytes == 0) {
            // 新的字符开始
            if ((current & 0x80) == 0) {
                // 1 字节字符: 0xxxxxx
                expectedBytes = 0;
            } else if ((current & 0xE0) == 0xC0) {
                // 2 字节字符: 110xxxxx
                expectedBytes = 1;
            } else if ((current & 0xF0) == 0xE0) {
                // 3 字节字符: 1110xxxx
                expectedBytes = 2;
            } else if ((current & 0xF8) == 0xF0) {
                // 4 字节字符: 11110xxx
                expectedBytes = 3;
            }
        }
    }
}
```

```
        } else {
            return false; // 无效的首字节
        }
    } else {
        // 检查后续字节格式: 10xxxxxx
        if ((current & 0xC0) != 0x80) {
            return false;
        }
        expectedBytes--;
    }
}

return expectedBytes == 0; // 所有字符必须完整
}

/***
 * 方法 3: 位掩码优化版
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
public boolean validUtf8_3(int[] data) {
    int count = 0; // 剩余需要验证的后续字节数

    for (int num : data) {
        if (count == 0) {
            if ((num >> 5) == 0b110) {
                count = 1;
            } else if ((num >> 4) == 0b1110) {
                count = 2;
            } else if ((num >> 3) == 0b11110) {
                count = 3;
            } else if ((num >> 7) != 0) {
                return false; // 无效的首字节
            }
        } else {
            if ((num >> 6) != 0b10) {
                return false;
            }
            count--;
        }
    }

    return count == 0;
}
```

```
}

/**  
 * 方法 4：详细的位运算验证  
 * 时间复杂度：O(n)  
 * 空间复杂度：O(1)  
 */  
  
public boolean validUtf8_4(int[] data) {  
    int index = 0;  
    int n = data.length;  
  
    while (index < n) {  
        int firstByte = data[index];  
  
        // 检查 1 字节字符  
        if ((firstByte & 0x80) == 0) {  
            index++;  
            continue;  
        }  
  
        // 检查多字节字符  
        int byteCount = getByteCount(firstByte);  
        if (byteCount == -1 || index + byteCount > n) {  
            return false;  
        }  
  
        // 验证后续字节  
        for (int i = 1; i < byteCount; i++) {  
            if (!isValidContinuation(data[index + i])) {  
                return false;  
            }  
        }  
  
        index += byteCount;  
    }  
  
    return true;  
}
```

```
// ===== 辅助方法 =====
```

```
/**  
 * 获取字节类型（字符的字节数）
```

```
* @param b 字节值
* @return 字符字节数, -1 表示无效
*/
private int getByteType(int b) {
    if ((b & 0x80) == 0) return 1;          // 0xxxxxxx
    if ((b & 0xE0) == 0xC0) return 2;        // 110xxxxx
    if ((b & 0xF0) == 0xE0) return 3;        // 1110xxxx
    if ((b & 0xF8) == 0xF0) return 4;        // 11110xxx
    return -1; // 无效字节
}
```

```
/***
 * 检查是否为有效的后续字节
 * @param b 字节值
 * @return 是否有效
*/
private boolean isContinuationByte(int b) {
    return (b & 0xC0) == 0x80; // 10xxxxxx
}
```

```
/***
 * 获取字符字节数
 * @param firstByte 首字节
 * @return 字节数, -1 表示无效
*/
private int getByteCount(int firstByte) {
    if ((firstByte & 0x80) == 0) return 1;
    if ((firstByte & 0xE0) == 0xC0) return 2;
    if ((firstByte & 0xF0) == 0xE0) return 3;
    if ((firstByte & 0xF8) == 0xF0) return 4;
    return -1;
}
```

```
/***
 * 检查是否为有效的后续字节
 * @param b 字节值
 * @return 是否有效
*/
private boolean isValidContinuation(int b) {
    return (b & 0xC0) == 0x80;
}
```

```
/***
```

```
* 测试方法
*/
public static void main(String[] args) {
    Code36_UTF8Validation solution = new Code36_UTF8Validation();

    // 测试用例 1: 有效 UTF-8 编码
    int[] data1 = {197, 130, 1}; // 2 字节字符 + 1 字节字符
    boolean result1 = solution.validUtf8_1(data1);
    System.out.println("测试用例 1 - 输入: [197, 130, 1]");
    System.out.println("结果: " + result1 + " (预期: true)");

    // 测试用例 2: 无效 UTF-8 编码
    int[] data2 = {235, 140, 4}; // 3 字节字符但第二个字节无效
    boolean result2 = solution.validUtf8_1(data2);
    System.out.println("测试用例 2 - 输入: [235, 140, 4]");
    System.out.println("结果: " + result2 + " (预期: false)");

    // 测试用例 3: 单字节字符
    int[] data3 = {65, 66, 67}; // ASCII 字符
    boolean result3 = solution.validUtf8_1(data3);
    System.out.println("测试用例 3 - 输入: [65, 66, 67]");
    System.out.println("结果: " + result3 + " (预期: true)");

    // 测试用例 4: 混合字符
    int[] data4 = {227, 129, 130, 65}; // 3 字节字符 + ASCII 字符
    boolean result4 = solution.validUtf8_1(data4);
    System.out.println("测试用例 4 - 输入: [227, 129, 130, 65]");
    System.out.println("结果: " + result4 + " (预期: true)");

    // 测试用例 5: 不完整的字符
    int[] data5 = {240, 162, 130}; // 4 字节字符但缺少最后一个字节
    boolean result5 = solution.validUtf8_1(data5);
    System.out.println("测试用例 5 - 输入: [240, 162, 130]");
    System.out.println("结果: " + result5 + " (预期: false)");

    // 性能测试
    int[] largeData = new int[10000];
    for (int i = 0; i < largeData.length; i++) {
        largeData[i] = 65; // 全部是 ASCII 字符
    }

    long startTime = System.currentTimeMillis();
    boolean perfResult = solution.validUtf8_1(largeData);
```

```

long endTime = System.currentTimeMillis();
System.out.println("性能测试 - 输入长度: " + largeData.length);
System.out.println("结果: " + perfResult);
System.out.println("耗时: " + (endTime - startTime) + "ms");

// 复杂度分析
System.out.println("\n== 复杂度分析 ==");
System.out.println("所有方法:");
System.out.println(" 时间复杂度: O(n) - 遍历数组一次");
System.out.println(" 空间复杂度: O(1) - 只使用常数空间");

// 工程化考量
System.out.println("\n== 工程化考量 ==");
System.out.println("1. 边界处理: 检查数组长度和字节范围");
System.out.println("2. 性能优化: 使用位运算提高效率");
System.out.println("3. 可读性: 清晰的变量命名和注释");
System.out.println("4. 错误处理: 详细的错误信息 (实际工程中)");

// 算法技巧总结
System.out.println("\n== 算法技巧总结 ==");
System.out.println("1. 位掩码: 使用位掩码检查字节格式");
System.out.println("2. 状态机: 跟踪当前字符的字节数");
System.out.println("3. 提前终止: 发现无效字节立即返回");
System.out.println("4. 边界检查: 确保不越界访问数组");

// UTF-8 编码规则总结
System.out.println("\n== UTF-8 编码规则 ==");
System.out.println("1 字节: 0xxxxxx");
System.out.println("2 字节: 110xxxxx 10xxxxxx");
System.out.println("3 字节: 1110xxxx 10xxxxxx 10xxxxxx");
System.out.println("4 字节: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx");
}

}
=====
```

文件: Code36\_UTF8Validation.py

"""

UTF-8 编码验证

测试链接: <https://leetcode.cn/problems/utf-8-validation/>

题目描述:

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。

UTF-8 中的一个字符可能的长度为 1 到 4 字节, 遵循以下规则:

1. 对于 1 字节的字符, 字节的第一位设为 0, 后面 7 位为这个符号的 Unicode 码。
2. 对于 n 字节的字符 ( $n > 1$ ), 第一个字节的前 n 位都设为 1, 第  $n+1$  位设为 0, 后面字节的前两位一律设为 10。

解题思路:

1. 逐字节验证: 按 UTF-8 编码规则逐个字节验证
2. 状态机: 使用状态机跟踪当前字符的字节数
3. 位运算: 使用位掩码检查字节格式

时间复杂度分析:

- 所有方法:  $O(n)$ ,  $n$  为数组长度

空间复杂度分析:

- 所有方法:  $O(1)$ , 只使用常数空间

"""

```
class Solution:
```

```
    def validUtf8_1(self, data: list[int]) -> bool:
```

```
        """
```

```
        方法 1: 逐字节验证 (推荐)
```

```
        时间复杂度: O(n)
```

```
        空间复杂度: O(1)
```

```
        """
```

```
        n = len(data)
```

```
        i = 0
```

```
        while i < n:
```

```
            # 获取当前字节
```

```
            current = data[i]
```

```
            # 判断当前字节的类型
```

```
            byte_type = self.get_byte_type(current)
```

```
            # 检查类型是否有效
```

```
            if byte_type == -1:
```

```
                return False
```

```
            # 检查后续字节数量是否足够
```

```
            if i + byte_type > n:
```

```
                return False
```

```
# 验证后续字节（如果是多字节字符）
for j in range(1, byte_type):
    if not self.is_continuation_byte(data[i + j]):
        return False

    i += byte_type

return True

def validUtf8_2(self, data: list[int]) -> bool:
    """
    方法 2: 状态机实现
    时间复杂度: O(n)
    空间复杂度: O(1)
    """
    expected_bytes = 0 # 期望的后续字节数

    for current in data:
        if expected_bytes == 0:
            # 新的字符开始
            if (current & 0x80) == 0:
                # 1 字节字符: 0xxxxxx
                expected_bytes = 0
            elif (current & 0xE0) == 0xC0:
                # 2 字节字符: 110xxxx
                expected_bytes = 1
            elif (current & 0xF0) == 0xE0:
                # 3 字节字符: 1110xxx
                expected_bytes = 2
            elif (current & 0xF8) == 0xF0:
                # 4 字节字符: 11110xx
                expected_bytes = 3
            else:
                return False # 无效的首字节
        else:
            # 检查后续字节格式: 10xxxxxx
            if (current & 0xC0) != 0x80:
                return False
            expected_bytes -= 1

    return expected_bytes == 0 # 所有字符必须完整
```

```
def validUtf8_3(self, data: list[int]) -> bool:
```

```
"""
方法3：位掩码优化版
时间复杂度：O(n)
空间复杂度：O(1)
"""
count = 0 # 剩余需要验证的后续字节数
```

```
for num in data:
    if count == 0:
        if (num >> 5) == 0b110:
            count = 1
        elif (num >> 4) == 0b1110:
            count = 2
        elif (num >> 3) == 0b11110:
            count = 3
        elif (num >> 7) != 0:
            return False # 无效的首字节
    else:
        if (num >> 6) != 0b10:
            return False
        count -= 1

return count == 0
```

```
def validUtf8_4(self, data: list[int]) -> bool:
```

```
"""
方法4：详细的位运算验证
时间复杂度：O(n)
空间复杂度：O(1)
"""
index = 0
n = len(data)
```

```
while index < n:
    first_byte = data[index]

    # 检查1字节字符
    if (first_byte & 0x80) == 0:
        index += 1
        continue

    # 检查多字节字符
    byte_count = self.get_byte_count(first_byte)
```

```
    if byte_count == -1 or index + byte_count > n:
        return False

    # 验证后续字节
    for i in range(1, byte_count):
        if not self.is_valid_continuation(data[index + i]):
            return False

    index += byte_count

    return True

# ===== 辅助方法 =====

def get_byte_type(self, b: int) -> int:
    """获取字节类型（字符的字节数）"""
    if (b & 0x80) == 0:
        return 1      # 0xxxxxx
    elif (b & 0xE0) == 0xC0:
        return 2      # 110xxxxx
    elif (b & 0xF0) == 0xE0:
        return 3      # 1110xxxx
    elif (b & 0xF8) == 0xF0:
        return 4      # 11110xxx
    else:
        return -1     # 无效字节

def is_continuation_byte(self, b: int) -> bool:
    """检查是否为有效的后续字节"""
    return (b & 0xC0) == 0x80  # 10xxxxxx

def get_byte_count(self, first_byte: int) -> int:
    """获取字符字节数"""
    if (first_byte & 0x80) == 0:
        return 1
    elif (first_byte & 0xE0) == 0xC0:
        return 2
    elif (first_byte & 0xF0) == 0xE0:
        return 3
    elif (first_byte & 0xF8) == 0xF0:
        return 4
    else:
        return -1
```

```
def is_valid_continuation(self, b: int) -> bool:  
    """检查是否为有效的后续字节"""  
    return (b & 0xC0) == 0x80  
  
def test_solution():  
    """测试函数"""  
    solution = Solution()  
  
    # 测试用例 1: 有效 UTF-8 编码  
    data1 = [197, 130, 1] # 2 字节字符 + 1 字节字符  
    result1 = solution.validUtf8_1(data1)  
    print(f"测试用例 1 - 输入: {data1}")  
    print(f"结果: {result1} (预期: True)")  
  
    # 测试用例 2: 无效 UTF-8 编码  
    data2 = [235, 140, 4] # 3 字节字符但第二个字节无效  
    result2 = solution.validUtf8_1(data2)  
    print(f"测试用例 2 - 输入: {data2}")  
    print(f"结果: {result2} (预期: False)")  
  
    # 测试用例 3: 单字节字符  
    data3 = [65, 66, 67] # ASCII 字符  
    result3 = solution.validUtf8_1(data3)  
    print(f"测试用例 3 - 输入: {data3}")  
    print(f"结果: {result3} (预期: True)")  
  
    # 测试用例 4: 混合字符  
    data4 = [227, 129, 130, 65] # 3 字节字符 + ASCII 字符  
    result4 = solution.validUtf8_1(data4)  
    print(f"测试用例 4 - 输入: {data4}")  
    print(f"结果: {result4} (预期: True)")  
  
    # 测试用例 5: 不完整的字符  
    data5 = [240, 162, 130] # 4 字节字符但缺少最后一个字节  
    result5 = solution.validUtf8_1(data5)  
    print(f"测试用例 5 - 输入: {data5}")  
    print(f"结果: {result5} (预期: False)")  
  
    # 性能测试  
    import time  
    large_data = [65] * 10000 # 全部是 ASCII 字符
```

```
start_time = time.time()
perf_result = solution.validUtf8_1(large_data)
end_time = time.time()
print(f"性能测试 - 输入长度: {len(large_data)}")
print(f"结果: {perf_result}")
print(f"耗时: {(end_time - start_time) * 1000:.2f}毫秒")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("所有方法:")
print(" 时间复杂度: O(n) - 遍历数组一次")
print(" 空间复杂度: O(1) - 只使用常数空间")

# 工程化考量
print("\n==== 工程化考量 ====")
print("1. 边界处理: 检查数组长度和字节范围")
print("2. 性能优化: 使用位运算提高效率")
print("3. 可读性: 清晰的变量命名和注释")
print("4. 错误处理: 详细的错误信息 (实际工程中)")

# 算法技巧总结
print("\n==== 算法技巧总结 ====")
print("1. 位掩码: 使用位掩码检查字节格式")
print("2. 状态机: 跟踪当前字符的字节数")
print("3. 提前终止: 发现无效字节立即返回")
print("4. 边界检查: 确保不越界访问数组")

# UTF-8 编码规则总结
print("\n==== UTF-8 编码规则 ====")
print("1 字节: 0xxxxxx")
print("2 字节: 110xxxxx 10xxxxxx")
print("3 字节: 1110xxxx 10xxxxxx 10xxxxxx")
print("4 字节: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx")

# 位运算演示
print("\n==== 位运算演示 ====")
test_byte = 197 # 二进制: 11000101
print(f"字节 {test_byte} 的二进制: {bin(test_byte)}")
print(f"检查是否为 2 字节字符: {(test_byte & 0xE0) == 0xC0}")

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

文件: Code37\_TotalHammingDistance.cpp

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

using namespace std;

/***
 * 汉明距离总和
 * 测试链接: https://leetcode.cn/problems/total-hamming-distance/
 *
 * 题目描述:
 * 两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。
 * 给你一个整数数组 nums，请你计算并返回 nums 中任意两个数之间汉明距离的总和。
 *
 * 解题思路:
 * 1. 暴力法: 双重循环计算所有组合（会超时）
 * 2. 位运算法: 逐位计算贡献值
 * 3. 数学优化: 利用组合数学优化计算
 *
 * 时间复杂度分析:
 * - 暴力法: O(n2)，会超时
 * - 位运算法: O(n * 32)，32 位整数
 * - 数学优化: O(n * 32)
 *
 * 空间复杂度分析:
 * - 所有方法: O(1)，只使用常数空间
 */

class Solution {
public:
    /**
     * 方法 1: 暴力法（不推荐，会超时）
     * 时间复杂度: O(n2)
     * 空间复杂度: O(1)
     */
    int totalHammingDistance1(vector<int>& nums) {
        int total = 0;
        int n = nums.size();

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

```
        for (int j = i + 1; j < n; j++) {
            total += bitset<32>(nums[i] ^ nums[j]).count();
        }
    }
```

```
    return total;
}
```

```
/**
```

```
* 方法 2：位运算法（推荐）
* 核心思想：逐位计算每个位的贡献
* 对于每个位，统计有多少个数的该位是 1（设为 count）
* 那么该位的贡献就是 count * (n - count)
* 时间复杂度：O(n * 32)
* 空间复杂度：O(1)
*/
```

```
int totalHammingDistance2(vector<int>& nums) {
    int total = 0;
    int n = nums.size();
```

```
// 遍历 32 位（整数最多 32 位）
```

```
for (int i = 0; i < 32; i++) {
    int countOnes = 0;

    // 统计当前位为 1 的数的个数
    for (int num : nums) {
        countOnes += (num >> i) & 1;
    }
```

```
// 当前位的贡献：countOnes * (n - countOnes)
```

```
    total += countOnes * (n - countOnes);
}
```

```
return total;
}
```

```
/**
```

```
* 方法 3：数学优化版
* 使用更高效的位运算技巧
* 时间复杂度：O(n * 32)
* 空间复杂度：O(1)
*/
```

```
int totalHammingDistance3(vector<int>& nums) {
```

```

int total = 0;
int n = nums.size();

for (int i = 0; i < 32; i++) {
    int mask = 1 << i;
    int count = 0;

    for (int num : nums) {
        if ((num & mask) != 0) {
            count++;
        }
    }

    total += count * (n - count);
}

return total;
}

/***
 * 方法 4： 使用 GCC 内置函数优化
 * 时间复杂度: O(n * 32)
 * 空间复杂度: O(1)
 */
int totalHammingDistance4(vector<int>& nums) {
    int total = 0;
    int n = nums.size();

    for (int bitPos = 0; bitPos < 32; bitPos++) {
        int ones = 0;

        for (int num : nums) {
            // 检查特定位是否为 1
            if (((num >> bitPos) & 1) == 1) {
                ones++;
            }
        }

        // 当前位的汉明距离贡献
        total += ones * (n - ones);
    }

    return total;
}

```

```
}

/**
 * 方法 5：分组统计法
 * 将数字按位分组统计
 * 时间复杂度：O(n * 32)
 * 空间复杂度：O(32)
 */
int totalHammingDistance5(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    int total = 0;
    int n = nums.size();

    // 创建 32 个桶，每个桶统计对应位的 1 的个数
    vector<int> bitCounts(32, 0);

    for (int num : nums) {
        for (int i = 0; i < 32; i++) {
            if ((num & (1 << i)) != 0) {
                bitCounts[i]++;
            }
        }
    }

    // 计算总汉明距离
    for (int count : bitCounts) {
        total += count * (n - count);
    }

    return total;
};

}

/**
 * 测试函数
 */
int main() {
    Solution solution;

    // 测试用例 1：基础情况
}
```

```

vector<int> nums1 = {4, 14, 2};
int result1 = solution.totalHammingDistance2(nums1);
cout << "测试用例 1 - 输入: [4, 14, 2]" << endl;
cout << "结果: " << result1 << " (预期: 6)" << endl;

// 测试用例 2: 两个相同数字
vector<int> nums2 = {4, 4};
int result2 = solution.totalHammingDistance2(nums2);
cout << "测试用例 2 - 输入: [4, 4]" << endl;
cout << "结果: " << result2 << " (预期: 0)" << endl;

// 测试用例 3: 三个不同数字
vector<int> nums3 = {1, 2, 3};
int result3 = solution.totalHammingDistance2(nums3);
cout << "测试用例 3 - 输入: [1, 2, 3]" << endl;
cout << "结果: " << result3 << " (预期: 4)" << endl;

// 测试用例 4: 边界情况 (单个元素)
vector<int> nums4 = {5};
int result4 = solution.totalHammingDistance2(nums4);
cout << "测试用例 4 - 输入: [5]" << endl;
cout << "结果: " << result4 << " (预期: 0)" << endl;

// 性能测试
vector<int> largeNums(1000);
for (int i = 0; i < 1000; i++) {
    largeNums[i] = i; // 0 到 999 的序列
}

auto start = chrono::high_resolution_clock::now();
int result5 = solution.totalHammingDistance2(largeNums);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "性能测试 - 输入长度: " << largeNums.size() << endl;
cout << "结果: " << result5 << endl;
cout << "耗时: " << duration.count() << "微秒" << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 暴力法:" << endl;
cout << " 时间复杂度: O(n2) - 会超时" << endl;
cout << " 空间复杂度: O(1)" << endl;

```

```

cout << "方法 2 - 位运算法:" << endl;
cout << " 时间复杂度: O(n * 32)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - 数学优化版:" << endl;
cout << " 时间复杂度: O(n * 32)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 4 - GCC 内置函数法:" << endl;
cout << " 时间复杂度: O(n * 32)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 5 - 分组统计法:" << endl;
cout << " 时间复杂度: O(n * 32)" << endl;
cout << " 空间复杂度: O(32)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择: 方法 2 是最优选择" << endl;
cout << "2. 性能优化: 避免 O(n2) 的暴力解法" << endl;
cout << "3. 边界处理: 处理空数组和单元素数组" << endl;
cout << "4. 可读性: 清晰的数学公式解释" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 组合数学: C(k, 2) = k*(k-1)/2 的变形应用" << endl;
cout << "2. 位运算: 逐位统计 1 的个数" << endl;
cout << "3. 贡献值计算: 每个位的独立贡献可以分开计算" << endl;
cout << "4. 数学优化: 利用对称性减少计算量" << endl;

// 数学原理说明
cout << "\n==== 数学原理说明 ===" << endl;
cout << "对于每个位位置:" << endl;
cout << " 设该位为 1 的数字有 k 个, 为 0 的数字有 m 个" << endl;
cout << " 那么该位产生的汉明距离贡献为: k * m" << endl;
cout << " 因为每个 1 和每个 0 的组合都会产生 1 的贡献" << endl;
cout << " 总贡献 = Σ (每个位的 k * m)" << endl;

return 0;
}
=====
```

文件: Code37\_TotalHammingDistance.java

```
=====
package class031;

/**
 * 汉明距离总和
 * 测试链接: https://leetcode.cn/problems/total-hamming-distance/
 *
 * 题目描述:
 * 两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。
 * 给你一个整数数组 nums，请你计算并返回 nums 中任意两个数之间汉明距离的总和。
 *
 * 示例:
 * 输入: nums = [4, 14, 2]
 * 输出: 6
 * 解释: 在二进制表示中, 4 表示为 0100, 14 表示为 1110, 2 表示为 0010。
 *
 * 汉明距离计算:
 * 4 和 14 的汉明距离为 2
 * 4 和 2 的汉明距离为 2
 * 14 和 2 的汉明距离为 2
 * 总和为 2 + 2 + 2 = 6
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * 0 <= nums[i] <= 10^9
 * 给定数组中的元素范围在 0 到 10^9 之间
 * 数组的长度不超过 10^4
 *
 * 解题思路:
 * 1. 暴力法: 双重循环计算所有组合 (会超时)
 * 2. 位运算法: 逐位计算贡献值
 * 3. 数学优化: 利用组合数学优化计算
 *
 * 时间复杂度分析:
 * - 暴力法: O(n^2), 会超时
 * - 位运算法: O(n * 32), 32 位整数
 * - 数学优化: O(n * 32)
 *
 * 空间复杂度分析:
 * - 所有方法: O(1), 只使用常数空间
 */

public class Code37_TotalHammingDistance {
```

```

/***
 * 方法 1：暴力法（不推荐，会超时）
 * 时间复杂度: O(n2)
 * 空间复杂度: O(1)
 */
public int totalHammingDistance1(int[] nums) {
    int total = 0;
    int n = nums.length;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            total += Integer.bitCount(nums[i] ^ nums[j]);
        }
    }

    return total;
}

/***
 * 方法 2：位运算法（推荐）
 * 核心思想：逐位计算每个位的贡献
 * 对于每个位，统计有多少个数的该位是 1（设为 count）
 * 那么该位的贡献就是 count * (n - count)
 * 时间复杂度: O(n * 32)
 * 空间复杂度: O(1)
 */
public int totalHammingDistance2(int[] nums) {
    int total = 0;
    int n = nums.length;

    // 遍历 32 位（整数最多 32 位）
    for (int i = 0; i < 32; i++) {
        int countOnes = 0;

        // 统计当前位为 1 的数的个数
        for (int num : nums) {
            countOnes += (num >> i) & 1;
        }

        // 当前位的贡献: countOnes * (n - countOnes)
        total += countOnes * (n - countOnes);
    }

}

```

```

        return total;
    }

/***
 * 方法 3：数学优化版
 * 使用更高效的位运算技巧
 * 时间复杂度：O(n * 32)
 * 空间复杂度：O(1)
 */
public int totalHammingDistance3(int[] nums) {
    int total = 0;
    int n = nums.length;

    for (int i = 0; i < 32; i++) {
        int mask = 1 << i;
        int count = 0;

        for (int num : nums) {
            if ((num & mask) != 0) {
                count++;
            }
        }

        total += count * (n - count);
    }

    return total;
}

/***
 * 方法 4：使用 Brian Kernighan 算法优化
 * 时间复杂度：O(n * 32)
 * 空间复杂度：O(1)
 */
public int totalHammingDistance4(int[] nums) {
    int total = 0;
    int n = nums.length;

    // 对于每个位位置
    for (int bitPos = 0; bitPos < 32; bitPos++) {
        int ones = 0;
        int zeros = 0;

```

```

        for (int num : nums) {
            // 检查特定位是否为 1
            if (((num >> bitPos) & 1) == 1) {
                ones++;
            } else {
                zeros++;
            }
        }

        // 当前位的汉明距离贡献
        total += ones * zeros;
    }

    return total;
}

```

```

/**
 * 方法 5：分组统计法
 * 将数字按位分组统计
 * 时间复杂度：O(n * 32)
 * 空间复杂度：O(1)
 */

```

```

public int totalHammingDistance5(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

```

```

    int total = 0;
    int n = nums.length;

    // 创建 32 个桶，每个桶统计对应位的 1 的个数
    int[] bitCounts = new int[32];

```

```

    for (int num : nums) {
        for (int i = 0; i < 32; i++) {
            if ((num & (1 << i)) != 0) {
                bitCounts[i]++;
            }
        }
    }

```

```

    // 计算总汉明距离
    for (int count : bitCounts) {

```

```
        total += count * (n - count);
    }

    return total;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code37_TotalHammingDistance solution = new Code37_TotalHammingDistance();

    // 测试用例 1: 基础情况
    int[] nums1 = {4, 14, 2};
    int result1 = solution.totalHammingDistance2(nums1);
    System.out.println("测试用例 1 - 输入: [4, 14, 2]");
    System.out.println("结果: " + result1 + " (预期: 6)");

    // 测试用例 2: 两个相同数字
    int[] nums2 = {4, 4};
    int result2 = solution.totalHammingDistance2(nums2);
    System.out.println("测试用例 2 - 输入: [4, 4]");
    System.out.println("结果: " + result2 + " (预期: 0)");

    // 测试用例 3: 三个不同数字
    int[] nums3 = {1, 2, 3};
    int result3 = solution.totalHammingDistance2(nums3);
    System.out.println("测试用例 3 - 输入: [1, 2, 3]");
    System.out.println("结果: " + result3 + " (预期: 4)");

    // 测试用例 4: 边界情况 (单个元素)
    int[] nums4 = {5};
    int result4 = solution.totalHammingDistance2(nums4);
    System.out.println("测试用例 4 - 输入: [5]");
    System.out.println("结果: " + result4 + " (预期: 0)");

    // 性能测试
    int[] largeNums = new int[1000];
    for (int i = 0; i < largeNums.length; i++) {
        largeNums[i] = i; // 0 到 999 的序列
    }

    long startTime = System.currentTimeMillis();
```

```
int result5 = solution.totalHammingDistance2(largeNums);
long endTime = System.currentTimeMillis();
System.out.println("性能测试 - 输入长度: " + largeNums.length);
System.out.println("结果: " + result5);
System.out.println("耗时: " + (endTime - startTime) + "ms");

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 暴力法:");
System.out.println(" 时间复杂度: O(n2) - 会超时");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 位运算法:");
System.out.println(" 时间复杂度: O(n * 32)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 3 - 数学优化版:");
System.out.println(" 时间复杂度: O(n * 32)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 4 - Brian Kernighan 算法:");
System.out.println(" 时间复杂度: O(n * 32)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 5 - 分组统计法:");
System.out.println(" 时间复杂度: O(n * 32)");
System.out.println(" 空间复杂度: O(32)");

// 工程化考量
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 算法选择: 方法 2 是最优选择");
System.out.println("2. 性能优化: 避免 O(n2) 的暴力解法");
System.out.println("3. 边界处理: 处理空数组和单元素数组");
System.out.println("4. 可读性: 清晰的数学公式解释");

// 算法技巧总结
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. 组合数学: C(k, 2) = k*(k-1)/2 的变形应用");
System.out.println("2. 位运算: 逐位统计 1 的个数");
System.out.println("3. 贡献值计算: 每个位的独立贡献可以分开计算");
System.out.println("4. 数学优化: 利用对称性减少计算量");

// 数学原理说明
```

```
System.out.println("\n== 数学原理说明 ==");
System.out.println("对于每个位位置: ");
System.out.println(" 设该位为 1 的数字有 k 个, 为 0 的数字有 m 个");
System.out.println(" 那么该位产生的汉明距离贡献为: k * m");
System.out.println(" 因为每个 1 和每个 0 的组合都会产生 1 的贡献");
System.out.println(" 总贡献 = Σ (每个位的 k * m)");
}

}

=====
```

文件: Code37\_TotalHammingDistance.py

```
"""

汉明距离总和
```

测试链接: <https://leetcode.cn/problems/total-hamming-distance/>

题目描述:

两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。

给你一个整数数组 `nums`, 请你计算并返回 `nums` 中任意两个数之间汉明距离的总和。

解题思路:

1. 暴力法: 双重循环计算所有组合 (会超时)
2. 位运算法: 逐位计算贡献值
3. 数学优化: 利用组合数学优化计算

时间复杂度分析:

- 暴力法:  $O(n^2)$ , 会超时
- 位运算法:  $O(n * 32)$ , 32 位整数
- 数学优化:  $O(n * 32)$

空间复杂度分析:

- 所有方法:  $O(1)$ , 只使用常数空间

```
"""

class Solution:
```

```
    def totalHammingDistance1(self, nums: list[int]) -> int:
```

```
        """

方法 1: 暴力法 (不推荐, 会超时)
```

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

```
        """

total = 0
```

```
n = len(nums)

for i in range(n):
    for j in range(i + 1, n):
        # 计算两个数的汉明距离
        total += bin(nums[i] ^ nums[j]).count('1')

return total
```

```
def totalHammingDistance2(self, nums: list[int]) -> int:
    """
```

方法 2：位运算法（推荐）

核心思想：逐位计算每个位的贡献

对于每个位，统计有多少个数的该位是 1（设为 count）

那么该位的贡献就是  $count * (n - count)$

时间复杂度： $O(n * 32)$

空间复杂度： $O(1)$

```
"""
```

```
total = 0
```

```
n = len(nums)
```

```
# 遍历 32 位（整数最多 32 位）
```

```
for i in range(32):
```

```
    count_ones = 0
```

```
# 统计当前位为 1 的数的个数
```

```
for num in nums:
```

```
    count_ones += (num >> i) & 1
```

```
# 当前位的贡献: count_ones * (n - count_ones)
```

```
total += count_ones * (n - count_ones)
```

```
return total
```

```
def totalHammingDistance3(self, nums: list[int]) -> int:
    """
```

方法 3：数学优化版

使用更高效的位运算技巧

时间复杂度： $O(n * 32)$

空间复杂度： $O(1)$

```
"""
```

```
total = 0
```

```
n = len(nums)
```

```
for i in range(32):
    mask = 1 << i
    count = 0

    for num in nums:
        if (num & mask) != 0:
            count += 1

    total += count * (n - count)

return total
```

```
def totalHammingDistance4(self, nums: list[int]) -> int:
    """
```

方法 4：使用位运算优化

时间复杂度： $O(n * 32)$

空间复杂度： $O(1)$

```
"""
```

```
total = 0
```

```
n = len(nums)
```

```
for bit_pos in range(32):
```

```
    ones = 0
```

```
    for num in nums:
```

# 检查特定位是否为 1

```
    if ((num >> bit_pos) & 1) == 1:
```

```
        ones += 1
```

# 当前位的汉明距离贡献

```
    total += ones * (n - ones)
```

```
return total
```

```
def totalHammingDistance5(self, nums: list[int]) -> int:
    """
```

方法 5：分组统计法

将数字按位分组统计

时间复杂度： $O(n * 32)$

空间复杂度： $O(32)$

```
"""
```

```
if not nums:
```

```
    return 0

total = 0
n = len(nums)

# 创建 32 个桶，每个桶统计对应位的 1 的个数
bit_counts = [0] * 32

for num in nums:
    for i in range(32):
        if (num & (1 << i)) != 0:
            bit_counts[i] += 1

# 计算总汉明距离
for count in bit_counts:
    total += count * (n - count)

return total

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 基础情况
    nums1 = [4, 14, 2]
    result1 = solution.totalHammingDistance2(nums1)
    print(f"测试用例 1 - 输入: {nums1}")
    print(f"结果: {result1} (预期: 6)")

    # 测试用例 2: 两个相同数字
    nums2 = [4, 4]
    result2 = solution.totalHammingDistance2(nums2)
    print(f"测试用例 2 - 输入: {nums2}")
    print(f"结果: {result2} (预期: 0)")

    # 测试用例 3: 三个不同数字
    nums3 = [1, 2, 3]
    result3 = solution.totalHammingDistance2(nums3)
    print(f"测试用例 3 - 输入: {nums3}")
    print(f"结果: {result3} (预期: 4)")

    # 测试用例 4: 边界情况 (单个元素)
    nums4 = [5]
```

```
result4 = solution.totalHammingDistance2(nums4)
print(f"测试用例 4 - 输入: {nums4}")
print(f"结果: {result4} (预期: 0)")

# 性能测试
import time
large_nums = list(range(1000)) # 0 到 999 的序列

start_time = time.time()
result5 = solution.totalHammingDistance2(large_nums)
end_time = time.time()
print(f"性能测试 - 输入长度: {len(large_nums)}")
print(f"结果: {result5}")
print(f"耗时: {(end_time - start_time) * 1000:.2f}毫秒")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 暴力法:")
print(" 时间复杂度: O(n2) - 会超时")
print(" 空间复杂度: O(1)")

print("方法 2 - 位运算法:")
print(" 时间复杂度: O(n * 32)")
print(" 空间复杂度: O(1)")

print("方法 3 - 数学优化版:")
print(" 时间复杂度: O(n * 32)")
print(" 空间复杂度: O(1)")

print("方法 4 - 位运算优化版:")
print(" 时间复杂度: O(n * 32)")
print(" 空间复杂度: O(1)")

print("方法 5 - 分组统计法:")
print(" 时间复杂度: O(n * 32)")
print(" 空间复杂度: O(32)")

# 工程化考量
print("\n==== 工程化考量 ====")
print("1. 算法选择: 方法 2 是最优选择")
print("2. 性能优化: 避免 O(n2) 的暴力解法")
print("3. 边界处理: 处理空数组和单元素数组")
print("4. 可读性: 清晰的数学公式解释")
```

```

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 组合数学: C(k, 2) = k*(k-1)/2 的变形应用")
print("2. 位运算: 逐位统计 1 的个数")
print("3. 贡献值计算: 每个位的独立贡献可以分开计算")
print("4. 数学优化: 利用对称性减少计算量")

# 数学原理说明
print("\n==== 数学原理说明 ===")
print("对于每个位位置: ")
print(" 设该位为 1 的数字有 k 个, 为 0 的数字有 m 个")
print(" 那么该位产生的汉明距离贡献为: k * m")
print(" 因为每个 1 和每个 0 的组合都会产生 1 的贡献")
print(" 总贡献 = Σ (每个位的 k * m)")

# 示例演示
print("\n==== 示例演示 ===")
demo_nums = [4, 14, 2] # 二进制: 0100, 1110, 0010
print(f"示例数组: {demo_nums}")
print("二进制表示:")
for i, num in enumerate(demo_nums):
    print(f" {num}: {bin(num)[2:]:>4}")

print("逐位分析:")
for bit_pos in range(4): # 只看前 4 位
    ones = 0
    for num in demo_nums:
        if ((num >> bit_pos) & 1) == 1:
            ones += 1
    zeros = len(demo_nums) - ones
    contribution = ones * zeros
    print(f" 第{bit_pos}位: 1 的个数={ones}, 0 的个数={zeros}, 贡献={contribution}")

if __name__ == "__main__":
    test_solution()

```

---

文件: Code38\_Numberof1Bits.cpp

---

```
#include <iostream>
#include <vector>
```

```
#include <bitset>
#include <cstdint>
#include <chrono>

using namespace std;

/***
 * 位 1 的个数（多种解法）
 * 测试链接: https://leetcode.cn/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（以二进制串的形式），
 * 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）。
 *
 * 解题思路:
 * 1. 逐位检查法: 检查每一位是否为 1
 * 2. Brian Kernighan 算法: n & (n-1) 消除最低位的 1
 * 3. 查表法: 预计算所有可能值的 1 的个数
 * 4. 分治法: 使用位运算技巧分组统计
 * 5. 内置函数法: 使用语言内置函数
 *
 * 时间复杂度分析:
 * - 逐位检查法: O(32) = O(1)
 * - Brian Kernighan 算法: O(k), k 为 1 的个数
 * - 查表法: O(1)
 * - 分治法: O(1)
 * - 内置函数法: O(1)
 *
 * 空间复杂度分析:
 * - 逐位检查法: O(1)
 * - Brian Kernighan 算法: O(1)
 * - 查表法: O(256) = O(1)
 * - 分治法: O(1)
 * - 内置函数法: O(1)
 */
class Solution {
public:
    /**
     * 方法 1: 逐位检查法
     * 时间复杂度: O(32) = O(1)
     * 空间复杂度: O(1)
     */
    int hammingWeight1(uint32_t n) {
```

```

int count = 0;

// 检查 32 位中的每一位
for (int i = 0; i < 32; i++) {
    // 检查第 i 位是否为 1
    if ((n & (1 << i)) != 0) {
        count++;
    }
}

return count;
}

/***
 * 方法 2: Brian Kernighan 算法 (推荐)
 * 核心思想: n & (n-1) 可以消除 n 的二进制表示中最右边的 1
 * 时间复杂度: O(k), k 为 1 的个数
 * 空间复杂度: O(1)
 */
int hammingWeight2(uint32_t n) {
    int count = 0;

    while (n != 0) {
        n &= n - 1; // 消除最低位的 1
        count++;
    }

    return count;
}

/***
 * 方法 3: 查表法
 * 预计算 0-255 所有数字的 1 的个数
 * 时间复杂度: O(1)
 * 空间复杂度: O(256) = O(1)
 */
int hammingWeight3(uint32_t n) {
    // 预计算表: 0-255 每个数字的 1 的个数
    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
};

// 将 32 位整数分成 4 个 8 位部分
return table[n & 0xFF] +
    table[(n >> 8) & 0xFF] +
    table[(n >> 16) & 0xFF] +
    table[(n >> 24) & 0xFF];
}

/***
* 方法 4：分治法（位运算技巧）
* 使用分治思想，先计算每 2 位的 1 的个数，然后合并
* 时间复杂度：O(1)
* 空间复杂度：O(1)
*/
int hammingWeight4(uint32_t n) {
    // 第一步：每 2 位统计 1 的个数
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555);

    // 第二步：每 4 位统计 1 的个数
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333);

    // 第三步：每 8 位统计 1 的个数
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);

    // 第四步：每 16 位统计 1 的个数
    n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF);

    // 第五步：每 32 位统计 1 的个数
    n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF);

    return n;
}

/***
* 方法 5：内置函数法
* 使用 C++ 内置的 __builtin_popcount()
* 时间复杂度：O(1)
* 空间复杂度：O(1)
*/

```

```

*/
int hammingWeight5(uint32_t n) {
    return __builtin_popcount(n);
}

/***
 * 方法 6: 移位统计法
 * 时间复杂度: O(32) = O(1)
 * 空间复杂度: O(1)
 */
int hammingWeight6(uint32_t n) {
    int count = 0;

    while (n != 0) {
        count += n & 1;
        n >>= 1;
    }

    return count;
}

/***
 * 方法 7: 并行计算法 (另一种分治)
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
int hammingWeight7(uint32_t n) {
    // 并行计算 1 的个数
    n = n - ((n >> 1) & 0x55555555);
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333);
    n = (n + (n >> 4)) & 0x0F0F0F0F;
    n = n + (n >> 8);
    n = n + (n >> 16);
    return n & 0x3F; // 最多 32 个 1, 所以取低 6 位
}

/***
 * 测试函数
*/
int main() {
    Solution solution;

```

```

// 测试用例 1: 正常情况
uint32_t n1 = 11; // 二进制: 1011
int result1 = solution.hammingWeight2(n1);
cout << "测试用例 1 - 输入: " << n1 << " (二进制: " << bitset<32>(n1) << ")" << endl;
cout << "结果: " << result1 << " (预期: 3)" << endl;

// 测试用例 2: 2 的幂
uint32_t n2 = 16; // 二进制: 10000
int result2 = solution.hammingWeight2(n2);
cout << "测试用例 2 - 输入: " << n2 << " (二进制: " << bitset<32>(n2) << ")" << endl;
cout << "结果: " << result2 << " (预期: 1)" << endl;

// 测试用例 3: 全 1
uint32_t n3 = 0xFFFFFFFF; // 二进制: 11111111111111111111111111111111
int result3 = solution.hammingWeight2(n3);
cout << "测试用例 3 - 输入: " << n3 << " (二进制: " << bitset<32>(n3) << ")" << endl;
cout << "结果: " << result3 << " (预期: 32)" << endl;

// 测试用例 4: 0
uint32_t n4 = 0; // 二进制: 0
int result4 = solution.hammingWeight2(n4);
cout << "测试用例 4 - 输入: " << n4 << " (二进制: " << bitset<32>(n4) << ")" << endl;
cout << "结果: " << result4 << " (预期: 0)" << endl;

// 性能测试
uint32_t largeNum = 0x7FFFFFFF; // 最大正数
auto start = chrono::high_resolution_clock::now();
int result5 = solution.hammingWeight2(largeNum);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
cout << "性能测试 - 输入: " << largeNum << endl;
cout << "结果: " << result5 << endl;
cout << "耗时: " << duration.count() << "微秒" << endl;

// 所有方法结果对比
cout << "\n== 所有方法结果对比 ==" << endl;
uint32_t testNum = 123456789;
cout << "测试数字: " << testNum << " (二进制: " << bitset<32>(testNum) << ")" << endl;
cout << "方法 1 (逐位检查): " << solution.hammingWeight1(testNum) << endl;
cout << "方法 2 (Brian Kernighan): " << solution.hammingWeight2(testNum) << endl;
cout << "方法 3 (查表法): " << solution.hammingWeight3(testNum) << endl;
cout << "方法 4 (分治法): " << solution.hammingWeight4(testNum) << endl;
cout << "方法 5 (内置函数): " << solution.hammingWeight5(testNum) << endl;

```

```
cout << "方法 6 (移位统计): " << solution.hammingWeight6(testNum) << endl;
cout << "方法 7 (并行计算): " << solution.hammingWeight7(testNum) << endl;

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 逐位检查法:" << endl;
cout << " 时间复杂度: O(32) = O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - Brian Kernighan 算法:" << endl;
cout << " 时间复杂度: O(k), k 为 1 的个数" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 3 - 查表法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(256) = O(1)" << endl;

cout << "方法 4 - 分治法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 5 - 内置函数法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择: 方法 2 (Brian Kernighan) 最优" << endl;
cout << "2. 性能优化: 避免不必要的循环" << endl;
cout << "3. 可读性: 方法 2 逻辑清晰" << endl;
cout << "4. 实际应用: C++内置函数性能最好" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. Brian Kernighan 算法: n & (n-1) 消除最低位 1" << endl;
cout << "2. 分治法: 使用位运算并行计算" << endl;
cout << "3. 查表法: 空间换时间" << endl;
cout << "4. 内置函数: 利用语言特性" << endl;

return 0;
}
```

=====

文件: Code38\_Numberof1Bits.java

```
=====
package class031;

/**
 * 位 1 的个数 (多种解法)
 * 测试链接: https://leetcode.cn/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（以二进制串的形式）,
 * 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）。
 *
 * 示例:
 * 输入: n = 11 (二进制 00000000000000000000000000001011)
 * 输出: 3
 * 解释: 输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
 *
 * 输入: n = 128 (二进制 000000000000000000000000000010000000)
 * 输出: 1
 *
 * 提示:
 * 输入必须是长度为 32 的 二进制串。
 *
 * 解题思路:
 * 1. 逐位检查法: 检查每一位是否为 1
 * 2. Brian Kernighan 算法:  $n \& (n-1)$  消除最低位的 1
 * 3. 查表法: 预计算所有可能值的 1 的个数
 * 4. 分治法: 使用位运算技巧分组统计
 * 5. 内置函数法: 使用语言内置函数
 *
 * 时间复杂度分析:
 * - 逐位检查法:  $O(32) = O(1)$ 
 * - Brian Kernighan 算法:  $O(k)$ ,  $k$  为 1 的个数
 * - 查表法:  $O(1)$ 
 * - 分治法:  $O(1)$ 
 * - 内置函数法:  $O(1)$ 
 *
 * 空间复杂度分析:
 * - 逐位检查法:  $O(1)$ 
 * - Brian Kernighan 算法:  $O(1)$ 
 * - 查表法:  $O(256) = O(1)$ 
 * - 分治法:  $O(1)$ 
```

```
* - 内置函数法: O(1)
*/
public class Code38_Numberof1Bits {

    /**
     * 方法 1: 逐位检查法
     * 时间复杂度: O(32) = O(1)
     * 空间复杂度: O(1)
     */
    public int hammingWeight1(int n) {
        int count = 0;

        // 检查 32 位中的每一位
        for (int i = 0; i < 32; i++) {
            // 检查第 i 位是否为 1
            if ((n & (1 << i)) != 0) {
                count++;
            }
        }

        return count;
    }

    /**
     * 方法 2: Brian Kernighan 算法 (推荐)
     * 核心思想: n & (n-1) 可以消除 n 的二进制表示中最右边的 1
     * 时间复杂度: O(k), k 为 1 的个数
     * 空间复杂度: O(1)
     */
    public int hammingWeight2(int n) {
        int count = 0;

        while (n != 0) {
            n &= n - 1; // 消除最低位的 1
            count++;
        }

        return count;
    }

    /**
     * 方法 3: 查表法
     * 预计算 0-255 所有数字的 1 的个数
     */
```

```

* 时间复杂度: O(1)
* 空间复杂度: O(256) = O(1)
*/
public int hammingWeight3(int n) {
    // 预计算表: 0~255 每个数字的 1 的个数
    int[] table = new int[256];
    for (int i = 0; i < 256; i++) {
        table[i] = table[i >> 1] + (i & 1);
    }

    // 将 32 位整数分成 4 个 8 位部分
    return table[n & 0xFF] +
        table[(n >> 8) & 0xFF] +
        table[(n >> 16) & 0xFF] +
        table[(n >> 24) & 0xFF];
}

/***
 * 方法 4: 分治法 (位运算技巧)
 * 使用分治思想, 先计算每 2 位的 1 的个数, 然后合并
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
*/
public int hammingWeight4(int n) {
    // 第一步: 每 2 位统计 1 的个数
    // 二进制: 01 -> 01, 10 -> 01, 11 -> 10
    n = (n & 0x55555555) + ((n >>> 1) & 0x55555555);

    // 第二步: 每 4 位统计 1 的个数
    n = (n & 0x33333333) + ((n >>> 2) & 0x33333333);

    // 第三步: 每 8 位统计 1 的个数
    n = (n & 0x0F0F0F0F) + ((n >>> 4) & 0x0F0F0F0F);

    // 第四步: 每 16 位统计 1 的个数
    n = (n & 0x00FF00FF) + ((n >>> 8) & 0x00FF00FF);

    // 第五步: 每 32 位统计 1 的个数
    n = (n & 0x0000FFFF) + ((n >>> 16) & 0x0000FFFF);

    return n;
}

```

```
/**  
 * 方法 5: 内置函数法  
 * 使用 Java 内置的 Integer.bitCount()  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 */  
  
public int hammingWeight5(int n) {  
    return Integer.bitCount(n);  
}  
  
/**  
 * 方法 6: 移位统计法 (无符号右移)  
 * 时间复杂度: O(32) = O(1)  
 * 空间复杂度: O(1)  
 */  
  
public int hammingWeight6(int n) {  
    int count = 0;  
  
    // 使用无符号右移, 避免符号位的影响  
    while (n != 0) {  
        count += n & 1;  
        n >>>= 1; // 无符号右移  
    }  
  
    return count;  
}  
  
/**  
 * 方法 7: 并行计算法 (另一种分治)  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 */  
  
public int hammingWeight7(int n) {  
    // 并行计算 1 的个数  
    n = n - ((n >>> 1) & 0x55555555);  
    n = (n & 0x33333333) + ((n >>> 2) & 0x33333333);  
    n = (n + (n >>> 4)) & 0x0F0F0F0F;  
    n = n + (n >>> 8);  
    n = n + (n >>> 16);  
    return n & 0x3F; // 最多 32 个 1, 所以取低 6 位  
}  
  
/**
```

```
* 测试方法
*/
public static void main(String[] args) {
    Code38_Numberof1Bits solution = new Code38_Numberof1Bits();

    // 测试用例 1: 正常情况
    int n1 = 11; // 二进制: 1011
    int result1 = solution.hammingWeight2(n1);
    System.out.println("测试用例 1 - 输入: " + n1 + " (二进制: " + Integer.toBinaryString(n1)
+ ")");
    System.out.println("结果: " + result1 + " (预期: 3)");

    // 测试用例 2: 2 的幂
    int n2 = 16; // 二进制: 10000
    int result2 = solution.hammingWeight2(n2);
    System.out.println("测试用例 2 - 输入: " + n2 + " (二进制: " + Integer.toBinaryString(n2)
+ ")");
    System.out.println("结果: " + result2 + " (预期: 1)");

    // 测试用例 3: 全 1
    int n3 = -1; // 二进制: 11
    int result3 = solution.hammingWeight2(n3);
    System.out.println("测试用例 3 - 输入: " + n3 + " (二进制: " + Integer.toBinaryString(n3)
+ ")");
    System.out.println("结果: " + result3 + " (预期: 32)");

    // 测试用例 4: 0
    int n4 = 0; // 二进制: 0
    int result4 = solution.hammingWeight2(n4);
    System.out.println("测试用例 4 - 输入: " + n4 + " (二进制: " + Integer.toBinaryString(n4)
+ ")");
    System.out.println("结果: " + result4 + " (预期: 0)");

    // 性能测试
    int largeNum = 0x7FFFFFFF; // 最大正数
    long startTime = System.currentTimeMillis();
    int result5 = solution.hammingWeight2(largeNum);
    long endTime = System.currentTimeMillis();
    System.out.println("性能测试 - 输入: " + largeNum);
    System.out.println("结果: " + result5);
    System.out.println("耗时: " + (endTime - startTime) + "ms");

    // 所有方法结果对比
}
```

```
System.out.println("\n==== 所有方法结果对比 ===");
int testNum = 123456789;
System.out.println("测试数字: " + testNum + " (二进制: " +
Integer.toBinaryString(testNum) + ")");
System.out.println("方法 1 (逐位检查): " + solution.hammingWeight1(testNum));
System.out.println("方法 2 (Brian Kernighan): " + solution.hammingWeight2(testNum));
System.out.println("方法 3 (查表法): " + solution.hammingWeight3(testNum));
System.out.println("方法 4 (分治法): " + solution.hammingWeight4(testNum));
System.out.println("方法 5 (内置函数): " + solution.hammingWeight5(testNum));
System.out.println("方法 6 (移位统计): " + solution.hammingWeight6(testNum));
System.out.println("方法 7 (并行计算): " + solution.hammingWeight7(testNum));
```

#### // 复杂度分析

```
System.out.println("\n==== 复杂度分析 ===");
System.out.println("方法 1 - 逐位检查法:");
System.out.println(" 时间复杂度:  $O(32) = O(1)$ ");
System.out.println(" 空间复杂度:  $O(1)$ ");

System.out.println("方法 2 - Brian Kernighan 算法:");
System.out.println(" 时间复杂度:  $O(k)$ , k 为 1 的个数");
System.out.println(" 空间复杂度:  $O(1)$ ");
```

```
System.out.println("方法 3 - 查表法:");
System.out.println(" 时间复杂度:  $O(1)$ ");
System.out.println(" 空间复杂度:  $O(256) = O(1)$ ");
```

```
System.out.println("方法 4 - 分治法:");
System.out.println(" 时间复杂度:  $O(1)$ ");
System.out.println(" 空间复杂度:  $O(1)$ ");
```

```
System.out.println("方法 5 - 内置函数法:");
System.out.println(" 时间复杂度:  $O(1)$ ");
System.out.println(" 空间复杂度:  $O(1)$ ");
```

#### // 工程化考量

```
System.out.println("\n==== 工程化考量 ===");
System.out.println("1. 算法选择: 方法 2 (Brian Kernighan) 最优");
System.out.println("2. 性能优化: 避免不必要的循环");
System.out.println("3. 可读性: 方法 2 逻辑清晰");
System.out.println("4. 实际应用: Java 内置函数性能最好");
```

#### // 算法技巧总结

```
System.out.println("\n==== 算法技巧总结 ===");
```

```
        System.out.println("1. Brian Kernighan 算法: n & (n-1) 消除最低位 1");
        System.out.println("2. 分治法: 使用位运算并行计算");
        System.out.println("3. 查表法: 空间换时间");
        System.out.println("4. 内置函数: 利用语言特性");
    }
}
```

=====

文件: Code38\_Numberof1Bits.py

=====

"""

位 1 的个数 (多种解法)

测试链接: <https://leetcode.cn/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（以二进制串的形式），  
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）。

解题思路:

1. 逐位检查法: 检查每一位是否为 1
2. Brian Kernighan 算法:  $n \& (n-1)$  消除最低位的 1
3. 查表法: 预计算所有可能值的 1 的个数
4. 分治法: 使用位运算技巧分组统计
5. 内置函数法: 使用语言内置函数

时间复杂度分析:

- 逐位检查法:  $O(32) = O(1)$
- Brian Kernighan 算法:  $O(k)$ ,  $k$  为 1 的个数
- 查表法:  $O(1)$
- 分治法:  $O(1)$
- 内置函数法:  $O(1)$

空间复杂度分析:

- 逐位检查法:  $O(1)$
- Brian Kernighan 算法:  $O(1)$
- 查表法:  $O(256) = O(1)$
- 分治法:  $O(1)$
- 内置函数法:  $O(1)$

"""

class Solution:

```
    def hammingWeight1(self, n: int) -> int:
```

```

"""
方法 1: 逐位检查法
时间复杂度: O(32) = O(1)
空间复杂度: O(1)
"""

count = 0

# 检查 32 位中的每一位
for i in range(32):
    # 检查第 i 位是否为 1
    if (n & (1 << i)) != 0:
        count += 1

return count

def hammingWeight2(self, n: int) -> int:
    """
方法 2: Brian Kernighan 算法 (推荐)
核心思想: n & (n-1) 可以消除 n 的二进制表示中最右边的 1
时间复杂度: O(k), k 为 1 的个数
空间复杂度: O(1)
"""

count = 0

# Python 中整数是动态大小的, 需要限制为 32 位
n = n & 0xFFFFFFFF

while n != 0:
    n = n & (n - 1)  # 消除最低位的 1
    count += 1

return count

def hammingWeight3(self, n: int) -> int:
    """
方法 3: 查表法
预计算 0-255 所有数字的 1 的个数
时间复杂度: O(1)
空间复杂度: O(256) = O(1)
"""

# 预计算表: 0-255 每个数字的 1 的个数
table = [0] * 256
for i in range(256):

```

```

        table[i] = table[i >> 1] + (i & 1)

    # 将 32 位整数分成 4 个 8 位部分
    n = n & 0xFFFFFFFF
    return table[n & 0xFF] + \
        table[(n >> 8) & 0xFF] + \
        table[(n >> 16) & 0xFF] + \
        table[(n >> 24) & 0xFF]

def hammingWeight4(self, n: int) -> int:
    """
    方法 4: 分治法 (位运算技巧)
    使用分治思想, 先计算每 2 位的 1 的个数, 然后合并
    时间复杂度: O(1)
    空间复杂度: O(1)
    """
    n = n & 0xFFFFFFFF # 限制为 32 位

    # 第一步: 每 2 位统计 1 的个数
    n = (n & 0x55555555) + ((n >> 1) & 0x55555555)

    # 第二步: 每 4 位统计 1 的个数
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333)

    # 第三步: 每 8 位统计 1 的个数
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F)

    # 第四步: 每 16 位统计 1 的个数
    n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF)

    # 第五步: 每 32 位统计 1 的个数
    n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF)

    return n

```

```

def hammingWeight5(self, n: int) -> int:
    """
    方法 5: 内置函数法
    使用 Python 内置的 bin() 函数
    时间复杂度: O(1)
    空间复杂度: O(1)
    """
    return bin(n & 0xFFFFFFFF).count('1')

```

```

def hammingWeight6(self, n: int) -> int:
    """
    方法 6: 移位统计法
    时间复杂度: O(32) = O(1)
    空间复杂度: O(1)
    """
    count = 0
    n = n & 0xFFFFFFFF # 限制为 32 位

    while n != 0:
        count += n & 1
        n = n >> 1

    return count

def hammingWeight7(self, n: int) -> int:
    """
    方法 7: 并行计算法 (另一种分治)
    时间复杂度: O(1)
    空间复杂度: O(1)
    """
    n = n & 0xFFFFFFFF # 限制为 32 位

    # 并行计算 1 的个数
    n = n - ((n >> 1) & 0x55555555)
    n = (n & 0x33333333) + ((n >> 2) & 0x33333333)
    n = (n + (n >> 4)) & 0x0F0F0F0F
    n = n + (n >> 8)
    n = n + (n >> 16)
    return n & 0x3F # 最多 32 个 1, 所以取低 6 位

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 正常情况
    n1 = 11 # 二进制: 1011
    result1 = solution.hammingWeight2(n1)
    print(f"测试用例 1 - 输入: {n1} (二进制: {bin(n1)})")
    print(f"结果: {result1} (预期: 3)")

    # 测试用例 2: 2 的幂

```

```
n2 = 16 # 二进制: 10000
result2 = solution.hammingWeight2(n2)
print(f"测试用例 2 - 输入: {n2} (二进制: {bin(n2)} )")
print(f"结果: {result2} (预期: 1)")
```

```
# 测试用例 3: 全 1
n3 = 0xFFFFFFFF # 二进制: 11111111111111111111111111111111
result3 = solution.hammingWeight2(n3)
print(f"测试用例 3 - 输入: {n3} (二进制: {bin(n3)} )")
print(f"结果: {result3} (预期: 32)")
```

```
# 测试用例 4: 0
n4 = 0 # 二进制: 0
result4 = solution.hammingWeight2(n4)
print(f"测试用例 4 - 输入: {n4} (二进制: {bin(n4)} )")
print(f"结果: {result4} (预期: 0)")
```

```
# 性能测试
import time
large_num = 0x7FFFFFFF # 最大正数

start_time = time.time()
result5 = solution.hammingWeight2(large_num)
end_time = time.time()
print(f"性能测试 - 输入: {large_num}")
print(f"结果: {result5}")
print(f"耗时: {(end_time - start_time) * 1000:.2f} 毫秒")
```

```
# 所有方法结果对比
print("\n== 所有方法结果对比 ==")
test_num = 123456789
print(f"测试数字: {test_num} (二进制: {bin(test_num)})")
print(f"方法 1 (逐位检查): {solution.hammingWeight1(test_num)}")
print(f"方法 2 (Brian Kernighan): {solution.hammingWeight2(test_num)}")
print(f"方法 3 (查表法): {solution.hammingWeight3(test_num)}")
print(f"方法 4 (分治法): {solution.hammingWeight4(test_num)}")
print(f"方法 5 (内置函数): {solution.hammingWeight5(test_num)}")
print(f"方法 6 (移位统计): {solution.hammingWeight6(test_num)}")
print(f"方法 7 (并行计算): {solution.hammingWeight7(test_num)}")
```

```
# 复杂度分析
print("\n== 复杂度分析 ==")
print("方法 1 - 逐位检查法:")
```

```
print(" 时间复杂度: O(32) = O(1)")  
print(" 空间复杂度: O(1)")  
  
print("方法 2 - Brian Kernighan 算法:")  
print(" 时间复杂度: O(k), k 为 1 的个数")  
print(" 空间复杂度: O(1)")  
  
print("方法 3 - 查表法:")  
print(" 时间复杂度: O(1)")  
print(" 空间复杂度: O(256) = O(1)")  
  
print("方法 4 - 分治法:")  
print(" 时间复杂度: O(1)")  
print(" 空间复杂度: O(1)")  
  
print("方法 5 - 内置函数法:")  
print(" 时间复杂度: O(1)")  
print(" 空间复杂度: O(1)")  
  
# 工程化考量  
print("\n==== 工程化考量 ===")  
print("1. 算法选择: 方法 2 (Brian Kernighan) 最优")  
print("2. 性能优化: 避免不必要的循环")  
print("3. 可读性: 方法 2 逻辑清晰")  
print("4. 实际应用: Python 内置函数最简洁")  
  
# 算法技巧总结  
print("\n==== 算法技巧总结 ===")  
print("1. Brian Kernighan 算法: n & (n-1) 消除最低位 1")  
print("2. 分治法: 使用位运算并行计算")  
print("3. 查表法: 空间换时间")  
print("4. 内置函数: 利用语言特性")  
  
# Python 特殊处理说明  
print("\n==== Python 特殊处理说明 ===")  
print("Python 整数是动态大小的, 需要手动限制为 32 位: ")  
print(" 使用 n & 0xFFFFFFFF 确保 32 位操作")  
print(" 避免负数的影响")  
print(" 确保位运算的正确性")  
  
if __name__ == "__main__":  
    test_solution()
```

文件: Code39\_PowerofFour.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdint>
#include <bitset>

using namespace std;

/***
 * 4 的幂 (位运算解法)
 * 测试链接: https://leetcode.cn/problems/power-of-four/
 *
 * 题目描述:
 * 给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true；否则，返回 false。
 * 整数 n 是 4 的幂次方需满足：存在整数 x 使得 n == 4^x
 *
 * 解题思路:
 * 1. 数学方法：循环除以 4
 * 2. 位运算法：利用 4 的幂的二进制特性
 * 3. 对数方法：使用对数函数判断
 * 4. 位运算优化：结合 2 的幂和特殊位置判断
 * 5. 查表法：预计算所有 4 的幂
 *
 * 时间复杂度分析:
 * - 数学方法: O(log4n)
 * - 位运算法: O(1)
 * - 对数方法: O(1)
 * - 位运算优化: O(1)
 * - 查表法: O(1)
 *
 * 空间复杂度分析:
 * - 数学方法: O(1)
 * - 位运算法: O(1)
 * - 对数方法: O(1)
 * - 位运算优化: O(1)
 * - 查表法: O(16) = O(1)
 */

class Solution {
public:
```

```
/**  
 * 方法 1: 数学方法 (循环除以 4)  
 * 时间复杂度: O(log4n)  
 * 空间复杂度: O(1)  
 */
```

```
bool isPowerOfFour1(int n) {  
    if (n <= 0) {  
        return false;  
    }  
  
    while (n % 4 == 0) {  
        n /= 4;  
    }  
  
    return n == 1;  
}
```

```
/**  
 * 方法 2: 位运算法 (推荐)  
 * 核心思想: 4 的幂一定是 2 的幂, 且 1 出现在奇数位  
 * 4 的幂的二进制特点:  
 * 1. 只有一个 1 (是 2 的幂)  
 * 2. 1 出现在奇数位 (从 1 开始计数)  
 * 3. 满足 n & (n-1) == 0 且 n & 0x55555555 != 0  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 */
```

```
bool isPowerOfFour2(int n) {  
    // 检查 n 是否为正数, 且是 2 的幂, 且 1 出现在奇数位  
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;  
}
```

```
/**  
 * 方法 3: 对数方法  
 * 使用换底公式: log4n = logn / log4  
 * 时间复杂度: O(1)  
 * 空间复杂度: O(1)  
 */
```

```
bool isPowerOfFour3(int n) {  
    if (n <= 0) {  
        return false;  
    }
```

```

double logResult = log(n) / log(4);
// 检查结果是否为整数（考虑浮点数精度）
return abs(logResult - round(logResult)) < 1e-10;
}

/***
 * 方法 4：位运算优化版
 * 结合多种位运算技巧
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
bool isPowerOfFour4(int n) {
    if (n <= 0) {
        return false;
    }

    // 检查是否是 2 的幂
    if ((n & (n - 1)) != 0) {
        return false;
    }

    // 检查 1 是否出现在奇数位
    // 0x55555555 = 0101010101010101010101010101
    return (n & 0x55555555) != 0;
}

/***
 * 方法 5：查表法
 * 预计算所有 32 位整数范围内的 4 的幂
 * 时间复杂度: O(1)
 * 空间复杂度: O(16) = O(1)
 */
bool isPowerOfFour5(int n) {
    // 32 位整数范围内所有 4 的幂
    vector<int> powersOfFour = {
        1,           // 4^0
        4,           // 4^1
        16,          // 4^2
        64,          // 4^3
        256,         // 4^4
        1024,        // 4^5
        4096,        // 4^6
        16384,       // 4^7
    }
}

```

```

65536,      // 4^8
262144,      // 4^9
1048576,     // 4^10
4194304,     // 4^11
16777216,    // 4^12
67108864,    // 4^13
268435456,   // 4^14
1073741824   // 4^15
};

for (int power : powersOfFour) {
    if (n == power) {
        return true;
    }
}

return false;
}

/***
 * 方法 6： 递归方法
 * 时间复杂度： O(log4n)
 * 空间复杂度： O(log4n) - 递归栈深度
 */
bool isPowerOfFour6(int n) {
    if (n <= 0) {
        return false;
    }
    if (n == 1) {
        return true;
    }
    if (n % 4 != 0) {
        return false;
    }
    return isPowerOfFour6(n / 4);
}

/***
 * 方法 7： 位运算+数学验证
 * 结合位运算和数学验证
 * 时间复杂度： O(1)
 * 空间复杂度： O(1)
 */

```

```
bool isPowerOfFour7(int n) {
    // 检查 n 是否为正数且是 2 的幂
    if (n <= 0 || (n & (n - 1)) != 0) {
        return false;
    }

    // 4 的幂除以 3 余数为 1
    // 2 的幂但不是 4 的幂除以 3 余数为 2
    return n % 3 == 1;
}
```

```
/***
 * 方法 8：位计数法
 * 统计 1 后面的 0 的个数，检查是否为偶数
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 */
```

```
bool isPowerOfFour8(int n) {
    if (n <= 0) {
        return false;
    }

    // 检查是否是 2 的幂
    if ((n & (n - 1)) != 0) {
        return false;
    }

    // 统计末尾 0 的个数（从最低位开始）
    int count = 0;
    int temp = n;
    while (temp > 1) {
        temp >>= 1;
        count++;
    }

    // 4 的幂要求 0 的个数为偶数
    return count % 2 == 0;
}
```

```
/***
 * 测试函数
 */
```

```
int main() {
    Solution solution;

    // 测试用例 1: 4 的幂
    int n1 = 16;
    bool result1 = solution.isPowerOfFour2(n1);
    cout << "测试用例 1 - 输入: " << n1 << endl;
    cout << "结果: " << (result1 ? "true" : "false") << " (预期: true)" << endl;

    // 测试用例 2: 不是 4 的幂
    int n2 = 5;
    bool result2 = solution.isPowerOfFour2(n2);
    cout << "测试用例 2 - 输入: " << n2 << endl;
    cout << "结果: " << (result2 ? "true" : "false") << " (预期: false)" << endl;

    // 测试用例 3: 1 ( $4^0$ )
    int n3 = 1;
    bool result3 = solution.isPowerOfFour2(n3);
    cout << "测试用例 3 - 输入: " << n3 << endl;
    cout << "结果: " << (result3 ? "true" : "false") << " (预期: true)" << endl;

    // 测试用例 4: 2 的幂但不是 4 的幂
    int n4 = 2;
    bool result4 = solution.isPowerOfFour2(n4);
    cout << "测试用例 4 - 输入: " << n4 << endl;
    cout << "结果: " << (result4 ? "true" : "false") << " (预期: false)" << endl;

    // 测试用例 5: 边界情况 (0)
    int n5 = 0;
    bool result5 = solution.isPowerOfFour2(n5);
    cout << "测试用例 5 - 输入: " << n5 << endl;
    cout << "结果: " << (result5 ? "true" : "false") << " (预期: false)" << endl;

    // 测试用例 6: 负数
    int n6 = -16;
    bool result6 = solution.isPowerOfFour2(n6);
    cout << "测试用例 6 - 输入: " << n6 << endl;
    cout << "结果: " << (result6 ? "true" : "false") << " (预期: false)" << endl;

    // 所有方法结果对比
    cout << "\n==== 所有方法结果对比 ===" << endl;
    int testNum = 64; //  $4^3$ 
    cout << "测试数字: " << testNum << " (二进制: " << bitset<32>(testNum) << ")" << endl;
```

```

cout << "方法 1 (数学方法): " << (solution.isPowerOfFour1(testNum) ? "true" : "false") <<
endl;
cout << "方法 2 (位运算法): " << (solution.isPowerOfFour2(testNum) ? "true" : "false") <<
endl;
cout << "方法 3 (对数方法): " << (solution.isPowerOfFour3(testNum) ? "true" : "false") <<
endl;
cout << "方法 4 (位运算优化): " << (solution.isPowerOfFour4(testNum) ? "true" : "false") <<
endl;
cout << "方法 5 (查表法): " << (solution.isPowerOfFour5(testNum) ? "true" : "false") << endl;
cout << "方法 6 (递归方法): " << (solution.isPowerOfFour6(testNum) ? "true" : "false") <<
endl;
cout << "方法 7 (位运算+数学): " << (solution.isPowerOfFour7(testNum) ? "true" : "false") <<
endl;
cout << "方法 8 (位计数法): " << (solution.isPowerOfFour8(testNum) ? "true" : "false") <<
endl;

```

#### // 复杂度分析

```

cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 数学方法:" << endl;
cout << " 时间复杂度: O(log4n)" << endl;
cout << " 空间复杂度: O(1)" << endl;

```

```

cout << "方法 2 - 位运算法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

```

```

cout << "方法 3 - 对数方法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

```

```

cout << "方法 4 - 位运算优化:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(1)" << endl;

```

```

cout << "方法 5 - 查表法:" << endl;
cout << " 时间复杂度: O(1)" << endl;
cout << " 空间复杂度: O(16) = O(1)" << endl;

```

#### // 工程化考量

```

cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择: 方法 2 (位运算法) 最优" << endl;
cout << "2. 性能优化: 避免循环和函数调用" << endl;
cout << "3. 边界处理: 处理 0 和负数" << endl;

```

```

cout << "4. 可读性: 清晰的位运算逻辑" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
cout << "1. 4 的幂特性: 是 2 的幂且 1 在奇数位" << endl;
cout << "2. 位运算技巧: n & (n-1) 判断 2 的幂" << endl;
cout << "3. 掩码应用: 0x55555555 检查奇数位" << endl;
cout << "4. 数学特性: 4 的幂除以 3 余数为 1" << endl;

// 二进制特性分析
cout << "\n==== 二进制特性分析 ===" << endl;
cout << "4 的幂的二进制表示特点: " << endl;
cout << " 4^0 = 1: 二进制 1" << endl;
cout << " 4^1 = 4: 二进制 100" << endl;
cout << " 4^2 = 16: 二进制 10000" << endl;
cout << " 4^3 = 64: 二进制 1000000" << endl;
cout << " 规律: 1 后面跟着偶数个 0" << endl;

return 0;
}

```

=====

文件: Code39\_PowerofFour.java

=====

```

package class031;

/**
 * 4 的幂 (位运算解法)
 * 测试链接: https://leetcode.cn/problems/power-of-four/
 *
 * 题目描述:
 * 给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true；否则，返回 false。
 * 整数 n 是 4 的幂次方需满足：存在整数 x 使得 n == 4^x
 *
 * 示例：
 * 输入: n = 16
 * 输出: true
 *
 * 输入: n = 5
 * 输出: false
 *
 * 输入: n = 1

```

```
* 输出: true
*
* 提示:
*  $-2^{31} \leq n \leq 2^{31} - 1$ 
*
* 解题思路:
* 1. 数学方法: 循环除以 4
* 2. 位运算法: 利用 4 的幂的二进制特性
* 3. 对数方法: 使用对数函数判断
* 4. 位运算优化: 结合 2 的幂和特殊位置判断
* 5. 查表法: 预计算所有 4 的幂
*
* 时间复杂度分析:
* - 数学方法:  $O(\log_4 n)$ 
* - 位运算法:  $O(1)$ 
* - 对数方法:  $O(1)$ 
* - 位运算优化:  $O(1)$ 
* - 查表法:  $O(1)$ 
*
* 空间复杂度分析:
* - 数学方法:  $O(1)$ 
* - 位运算法:  $O(1)$ 
* - 对数方法:  $O(1)$ 
* - 位运算优化:  $O(1)$ 
* - 查表法:  $O(32) = O(1)$ 
*/
public class Code39_PowerofFour {
```

```
/***
* 方法 1: 数学方法 (循环除以 4)
* 时间复杂度:  $O(\log_4 n)$ 
* 空间复杂度:  $O(1)$ 
*/
```

```
public boolean isPowerOfFour1(int n) {
    if (n <= 0) {
        return false;
    }

    while (n % 4 == 0) {
        n /= 4;
    }

    return n == 1;
```

```
}
```

```
/**
```

```
* 方法 2: 位运算法 (推荐)
```

```
* 核心思想: 4 的幂一定是 2 的幂, 且 1 出现在奇数位
```

```
* 4 的幂的二进制特点:
```

```
* 1. 只有一个 1 (是 2 的幂)
```

```
* 2. 1 出现在奇数位 (从 1 开始计数)
```

```
* 3. 满足  $n \& (n - 1) == 0$  且  $n \& 0x55555555 != 0$ 
```

```
* 时间复杂度: O(1)
```

```
* 空间复杂度: O(1)
```

```
*/
```

```
public boolean isPowerOfFour2(int n) {
```

```
    // 检查 n 是否为正数, 且是 2 的幂, 且 1 出现在奇数位
```

```
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
```

```
}
```

```
/**
```

```
* 方法 3: 对数方法
```

```
* 使用换底公式:  $\log_4 n = \log n / \log 4$ 
```

```
* 时间复杂度: O(1)
```

```
* 空间复杂度: O(1)
```

```
*/
```

```
public boolean isPowerOfFour3(int n) {
```

```
    if (n <= 0) {
```

```
        return false;
```

```
}
```

```
    double logResult = Math.log(n) / Math.log(4);
```

```
    // 检查结果是否为整数 (考虑浮点数精度)
```

```
    return Math.abs(logResult - Math.round(logResult)) < 1e-10;
```

```
}
```

```
/**
```

```
* 方法 4: 位运算优化版
```

```
* 结合多种位运算技巧
```

```
* 时间复杂度: O(1)
```

```
* 空间复杂度: O(1)
```

```
*/
```

```
public boolean isPowerOfFour4(int n) {
```

```
    if (n <= 0) {
```

```
        return false;
```

```
}
```

```
// 检查是否是 2 的幂
if ((n & (n - 1)) != 0) {
    return false;
}

// 检查 1 是否出现在奇数位
// 0x55555555 = 010101010101010101010101010101
return (n & 0x55555555) != 0;
}

/***
 * 方法 5：查表法
 * 预计算所有 32 位整数范围内的 4 的幂
 * 时间复杂度: O(1)
 * 空间复杂度: O(16) = O(1)
 */
public boolean isPowerOfFour(int n) {
    // 32 位整数范围内所有 4 的幂
    int[] powersOfFour = {
        1,          // 4^0
        4,          // 4^1
        16,         // 4^2
        64,         // 4^3
        256,        // 4^4
        1024,       // 4^5
        4096,       // 4^6
        16384,      // 4^7
        65536,      // 4^8
        262144,     // 4^9
        1048576,    // 4^10
        4194304,    // 4^11
        16777216,   // 4^12
        67108864,   // 4^13
        268435456,  // 4^14
        1073741824 // 4^15
    };

    for (int power : powersOfFour) {
        if (n == power) {
            return true;
        }
    }
}
```

```
        return false;
    }

/***
 * 方法 6：递归方法
 * 时间复杂度：O(log4n)
 * 空间复杂度：O(log4n) - 递归栈深度
 */
public boolean isPowerOfFour6(int n) {
    if (n <= 0) {
        return false;
    }
    if (n == 1) {
        return true;
    }
    if (n % 4 != 0) {
        return false;
    }
    return isPowerOfFour6(n / 4);
}
```

```
/***
 * 方法 7：位运算+数学验证
 * 结合位运算和数学验证
 * 时间复杂度：O(1)
 * 空间复杂度：O(1)
 */
public boolean isPowerOfFour7(int n) {
    // 检查 n 是否为正数且是 2 的幂
    if (n <= 0 || (n & (n - 1)) != 0) {
        return false;
    }

    // 4 的幂除以 3 余数为 1
    // 2 的幂但不是 4 的幂除以 3 余数为 2
    return n % 3 == 1;
}
```

```
/***
 * 方法 8：位计数法
 * 统计 1 后面的 0 的个数，检查是否为偶数
 * 时间复杂度：O(1)
 */
```

```

* 空间复杂度: O(1)
*/
public boolean isPowerOfFour8(int n) {
    if (n <= 0) {
        return false;
    }

    // 检查是否是 2 的幂
    if ((n & (n - 1)) != 0) {
        return false;
    }

    // 统计末尾 0 的个数（从最低位开始）
    int count = 0;
    while (n > 1) {
        n >>= 1;
        count++;
    }

    // 4 的幂要求 0 的个数为偶数
    return count % 2 == 0;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code39_PowerofFour solution = new Code39_PowerofFour();

    // 测试用例 1: 4 的幂
    int n1 = 16;
    boolean result1 = solution.isPowerOfFour2(n1);
    System.out.println("测试用例 1 - 输入: " + n1);
    System.out.println("结果: " + result1 + " (预期: true)");

    // 测试用例 2: 不是 4 的幂
    int n2 = 5;
    boolean result2 = solution.isPowerOfFour2(n2);
    System.out.println("测试用例 2 - 输入: " + n2);
    System.out.println("结果: " + result2 + " (预期: false)");

    // 测试用例 3: 1 ( $4^0$ )
    int n3 = 1;
}

```

```

boolean result3 = solution.isPowerOfFour2(n3);
System.out.println("测试用例 3 - 输入: " + n3);
System.out.println("结果: " + result3 + " (预期: true)");

// 测试用例 4: 2 的幂但不是 4 的幂
int n4 = 2;
boolean result4 = solution.isPowerOfFour2(n4);
System.out.println("测试用例 4 - 输入: " + n4);
System.out.println("结果: " + result4 + " (预期: false)");

// 测试用例 5: 边界情况 (0)
int n5 = 0;
boolean result5 = solution.isPowerOfFour2(n5);
System.out.println("测试用例 5 - 输入: " + n5);
System.out.println("结果: " + result5 + " (预期: false)");

// 测试用例 6: 负数
int n6 = -16;
boolean result6 = solution.isPowerOfFour2(n6);
System.out.println("测试用例 6 - 输入: " + n6);
System.out.println("结果: " + result6 + " (预期: false)");

// 所有方法结果对比
System.out.println("\n==== 所有方法结果对比 ===");
int testNum = 64; // 4^3
System.out.println("测试数字: " + testNum + " (二进制: " +
Integer.toBinaryString(testNum) + ")");
System.out.println("方法 1 (数学方法): " + solution.isPowerOfFour1(testNum));
System.out.println("方法 2 (位运算法): " + solution.isPowerOfFour2(testNum));
System.out.println("方法 3 (对数方法): " + solution.isPowerOfFour3(testNum));
System.out.println("方法 4 (位运算优化): " + solution.isPowerOfFour4(testNum));
System.out.println("方法 5 (查表法): " + solution.isPowerOfFour5(testNum));
System.out.println("方法 6 (递归方法): " + solution.isPowerOfFour6(testNum));
System.out.println("方法 7 (位运算+数学): " + solution.isPowerOfFour7(testNum));
System.out.println("方法 8 (位计数法): " + solution.isPowerOfFour8(testNum));

// 复杂度分析
System.out.println("\n==== 复杂度分析 ===");
System.out.println("方法 1 - 数学方法:");
System.out.println(" 时间复杂度: O(log4n)");
System.out.println(" 空间复杂度: O(1)");

System.out.println("方法 2 - 位运算法:");

```

```
System.out.println(" 时间复杂度: O(1)");  
System.out.println(" 空间复杂度: O(1)");  
  
System.out.println("方法 3 - 对数方法:");  
System.out.println(" 时间复杂度: O(log n)");  
System.out.println(" 空间复杂度: O(1)");  
  
System.out.println("方法 4 - 位运算优化:");  
System.out.println(" 时间复杂度: O(1)");  
System.out.println(" 空间复杂度: O(1)");  
  
System.out.println("方法 5 - 查表法:");  
System.out.println(" 时间复杂度: O(1)");  
System.out.println(" 空间复杂度: O(16) = O(1)");  
  
// 工程化考量  
System.out.println("\n==== 工程化考量 ====");  
System.out.println("1. 算法选择: 方法 2 (位运算法) 最优");  
System.out.println("2. 性能优化: 避免循环和函数调用");  
System.out.println("3. 边界处理: 处理 0 和负数");  
System.out.println("4. 可读性: 清晰的位运算逻辑");  
  
// 算法技巧总结  
System.out.println("\n==== 算法技巧总结 ====");  
System.out.println("1. 4 的幂特性: 是 2 的幂且 1 在奇数位");  
System.out.println("2. 位运算技巧: n & (n-1) 判断 2 的幂");  
System.out.println("3. 掩码应用: 0x55555555 检查奇数位");  
System.out.println("4. 数学特性: 4 的幂除以 3 余数为 1");  
  
// 二进制特性分析  
System.out.println("\n==== 二进制特性分析 ====");  
System.out.println("4 的幂的二进制表示特点: ");  
System.out.println(" 4^0 = 1: 二进制 1");  
System.out.println(" 4^1 = 4: 二进制 100");  
System.out.println(" 4^2 = 16: 二进制 10000");  
System.out.println(" 4^3 = 64: 二进制 1000000");  
System.out.println(" 规律: 1 后面跟着偶数个 0");  
  
// 位运算验证  
System.out.println("\n==== 位运算验证 ====");  
int[] testCases = {1, 4, 16, 64, 256};  
for (int num : testCases) {  
    boolean isPowerOf2 = (num & (num - 1)) == 0;
```

```
        boolean isPowerOf4 = (num & 0x55555555) != 0;
        System.out.println("数字：" + num + "，是2的幂：" + isPowerOf2 + "，是4的幂：" +
isPowerOf4);
    }
}
}
```

---

文件: Code39\_PowerofFour.py

```
"""
4 的幂（位运算解法）
测试链接: https://leetcode.cn/problems/power-of-four/
```

题目描述:

给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true；否则，返回 false。  
整数 n 是 4 的幂次方需满足：存在整数 x 使得  $n = 4^x$

解题思路:

1. 数学方法: 循环除以 4
2. 位运算法: 利用 4 的幂的二进制特性
3. 对数方法: 使用对数函数判断
4. 位运算优化: 结合 2 的幂和特殊位置判断
5. 查表法: 预计算所有 4 的幂

时间复杂度分析:

- 数学方法:  $O(\log_4 n)$
- 位运算法:  $O(1)$
- 对数方法:  $O(1)$
- 位运算优化:  $O(1)$
- 查表法:  $O(1)$

空间复杂度分析:

- 数学方法:  $O(1)$
- 位运算法:  $O(1)$
- 对数方法:  $O(1)$
- 位运算优化:  $O(1)$
- 查表法:  $O(16) = O(1)$

```
import math
```

```

class Solution:

    def isPowerOfFour1(self, n: int) -> bool:
        """
        方法 1: 数学方法 (循环除以 4)
        时间复杂度: O(log4n)
        空间复杂度: O(1)
        """
        if n <= 0:
            return False

        # Python 整数除法需要注意
        while n % 4 == 0:
            n = n // 4

        return n == 1

    def isPowerOfFour2(self, n: int) -> bool:
        """
        方法 2: 位运算法 (推荐)
        核心思想: 4 的幂一定是 2 的幂, 且 1 出现在奇数位
        4 的幂的二进制特点:
        1. 只有一个 1 (是 2 的幂)
        2. 1 出现在奇数位 (从 1 开始计数)
        3. 满足 n & (n-1) == 0 且 n & 0x55555555 != 0
        时间复杂度: O(1)
        空间复杂度: O(1)
        """
        # 检查 n 是否为正数, 且是 2 的幂, 且 1 出现在奇数位
        return n > 0 and (n & (n - 1)) == 0 and (n & 0x55555555) != 0

    def isPowerOfFour3(self, n: int) -> bool:
        """
        方法 3: 对数方法
        使用换底公式: log4n = logn / log4
        时间复杂度: O(1)
        空间复杂度: O(1)
        """
        if n <= 0:
            return False

        log_result = math.log(n) / math.log(4)
        # 检查结果是否为整数 (考虑浮点数精度)
        return abs(log_result - round(log_result)) < 1e-10

```

```

def isPowerOfFour4(self, n: int) -> bool:
    """
    方法 4: 位运算优化版
    结合多种位运算技巧
    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
    if (n & (n - 1)) != 0:
        return False

    # 检查 1 是否出现在奇数位
    # 0x55555555 = 010101010101010101010101010101
    return (n & 0x55555555) != 0

def isPowerOfFour5(self, n: int) -> bool:
    """
    方法 5: 查表法
    预计算所有 32 位整数范围内的 4 的幂
    时间复杂度: O(1)
    空间复杂度: O(16) = O(1)
    """

    # 32 位整数范围内所有 4 的幂
    powers_of_four = {
        1,          # 4^0
        4,          # 4^1
        16,         # 4^2
        64,         # 4^3
        256,        # 4^4
        1024,       # 4^5
        4096,       # 4^6
        16384,      # 4^7
        65536,      # 4^8
        262144,     # 4^9
        1048576,    # 4^10
        4194304,    # 4^11
        16777216,   # 4^12
        67108864,   # 4^13
        268435456,  # 4^14
    }

```

```
1073741824 # 4^15
```

```
}
```

```
return n in powers_of_four
```

```
def isPowerOfFour6(self, n: int) -> bool:
```

```
"""
```

方法 6：递归方法

时间复杂度： $O(\log_4 n)$

空间复杂度： $O(\log_4 n)$  – 递归栈深度

```
"""
```

```
if n <= 0:
```

```
    return False
```

```
if n == 1:
```

```
    return True
```

```
if n % 4 != 0:
```

```
    return False
```

```
return self.isPowerOfFour6(n // 4)
```

```
def isPowerOfFour7(self, n: int) -> bool:
```

```
"""
```

方法 7：位运算+数学验证

结合位运算和数学验证

时间复杂度： $O(1)$

空间复杂度： $O(1)$

```
"""
```

```
# 检查 n 是否为正数且是 2 的幂
```

```
if n <= 0 or (n & (n - 1)) != 0:
```

```
    return False
```

```
# 4 的幂除以 3 余数为 1
```

```
# 2 的幂但不是 4 的幂除以 3 余数为 2
```

```
return n % 3 == 1
```

```
def isPowerOfFour8(self, n: int) -> bool:
```

```
"""
```

方法 8：位计数法

统计 1 后面的 0 的个数，检查是否为偶数

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
if (n & (n - 1)) != 0:
    return False

# 统计末尾 0 的个数（从最低位开始）
count = 0
temp = n
while temp > 1:
    temp = temp >> 1
    count += 1

# 4 的幂要求 0 的个数为偶数
return count % 2 == 0

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 4 的幂
    n1 = 16
    result1 = solution.isPowerOfFour2(n1)
    print(f"测试用例 1 - 输入: {n1}")
    print(f"结果: {result1} (预期: True)")

    # 测试用例 2: 不是 4 的幂
    n2 = 5
    result2 = solution.isPowerOfFour2(n2)
    print(f"测试用例 2 - 输入: {n2}")
    print(f"结果: {result2} (预期: False)")

    # 测试用例 3: 1 ( $4^0$ )
    n3 = 1
    result3 = solution.isPowerOfFour2(n3)
    print(f"测试用例 3 - 输入: {n3}")
    print(f"结果: {result3} (预期: True)")

    # 测试用例 4: 2 的幂但不是 4 的幂
    n4 = 2
    result4 = solution.isPowerOfFour2(n4)
    print(f"测试用例 4 - 输入: {n4}")
    print(f"结果: {result4} (预期: False)")
```

```

# 测试用例 5: 边界情况 (0)
n5 = 0
result5 = solution.isPowerOfFour2(n5)
print(f"测试用例 5 - 输入: {n5}")
print(f"结果: {result5} (预期: False)")

# 测试用例 6: 负数
n6 = -16
result6 = solution.isPowerOfFour2(n6)
print(f"测试用例 6 - 输入: {n6}")
print(f"结果: {result6} (预期: False)")

# 所有方法结果对比
print("\n==== 所有方法结果对比 ===")
test_num = 64 # 4^3
print(f"测试数字: {test_num} (二进制: {bin(test_num)})")
print(f"方法 1 (数学方法): {solution.isPowerOfFour1(test_num)}")
print(f"方法 2 (位运算法): {solution.isPowerOfFour2(test_num)}")
print(f"方法 3 (对数方法): {solution.isPowerOfFour3(test_num)}")
print(f"方法 4 (位运算优化): {solution.isPowerOfFour4(test_num)}")
print(f"方法 5 (查表法): {solution.isPowerOfFour5(test_num)}")
print(f"方法 6 (递归方法): {solution.isPowerOfFour6(test_num)}")
print(f"方法 7 (位运算+数学): {solution.isPowerOfFour7(test_num)}")
print(f"方法 8 (位计数法): {solution.isPowerOfFour8(test_num)}")

# 复杂度分析
print("\n==== 复杂度分析 ===")
print("方法 1 - 数学方法:")
print("  时间复杂度: O(log4n)")
print("  空间复杂度: O(1)")

print("方法 2 - 位运算法:")
print("  时间复杂度: O(1)")
print("  空间复杂度: O(1)")

print("方法 3 - 对数方法:")
print("  时间复杂度: O(1)")
print("  空间复杂度: O(1)")

print("方法 4 - 位运算优化:")
print("  时间复杂度: O(1)")
print("  空间复杂度: O(1)")

```

```

print("方法 5 - 查表法:")
print(" 时间复杂度: O(1)")
print(" 空间复杂度: O(16) = O(1)")

# 工程化考量
print("\n==== 工程化考量 ===")
print("1. 算法选择: 方法 2 (位运算法) 最优")
print("2. 性能优化: 避免循环和函数调用")
print("3. 边界处理: 处理 0 和负数")
print("4. 可读性: 清晰的位运算逻辑")

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 4 的幂特性: 是 2 的幂且 1 在奇数位")
print("2. 位运算技巧: n & (n-1) 判断 2 的幂")
print("3. 掩码应用: 0x55555555 检查奇数位")
print("4. 数学特性: 4 的幂除以 3 余数为 1")

# 二进制特性分析
print("\n==== 二进制特性分析 ===")
print("4 的幂的二进制表示特点: ")
print(" 4^0 = 1: 二进制 1")
print(" 4^1 = 4: 二进制 100")
print(" 4^2 = 16: 二进制 10000")
print(" 4^3 = 64: 二进制 1000000")
print(" 规律: 1 后面跟着偶数个 0")

# Python 特殊处理说明
print("\n==== Python 特殊处理说明 ===")
print("Python 整数是动态大小的, 但位运算仍然有效: ")
print(" 使用 n & 0x55555555 确保 32 位操作")
print(" 位运算在 Python 中自动处理大整数")
print(" 注意整数除法使用 // 而不是 /")

if __name__ == "__main__":
    test_solution()

```

=====

文件: Code40\_MaximumProductofWordLengths.cpp

=====

```
#include <iostream>
#include <vector>
```

```
#include <string>
#include <unordered_map>
#include <algorithm>
#include <random>
#include <chrono>

using namespace std;

/***
 * 最大单词长度乘积（位掩码优化）
 * 测试链接: https://leetcode.cn/problems/maximum-product-of-word-lengths/
 *
 * 题目描述:
 * 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。如果不存在这样的两个单词，返回 0。
 *
 * 解题思路:
 * 1. 暴力法: 双重循环检查所有组合（会超时）
 * 2. 位掩码法: 使用位掩码表示每个单词的字符集合
 * 3. 位掩码优化: 预计算位掩码和长度，优化比较过程
 * 4. 哈希表法: 使用哈希表存储相同位掩码的最大长度
 * 5. 分组法: 按位掩码分组，只比较不同组的单词
 *
 * 时间复杂度分析:
 * - 暴力法: O(n2 * L)，L 为单词平均长度
 * - 位掩码法: O(n2 + nL)
 * - 位掩码优化: O(n2 + nL)
 * - 哈希表法: O(n2 + nL)
 * - 分组法: O(n2 + nL)
 *
 * 空间复杂度分析:
 * - 暴力法: O(1)
 * - 位掩码法: O(n)
 * - 位掩码优化: O(n)
 * - 哈希表法: O(n)
 * - 分组法: O(n)
 */
class Solution {
public:
    /**
     * 方法 1: 暴力法（不推荐，会超时）
     * 时间复杂度: O(n2 * L)，L 为单词平均长度
```

```

* 空间复杂度: O(1)
*/
int maxProduct1(vector<string>& words) {
    int maxProduct = 0;
    int n = words.size();

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (!hasCommonChars(words[i], words[j])) {
                maxProduct = max(maxProduct, (int)(words[i].length() * words[j].length()));
            }
        }
    }

    return maxProduct;
}

/***
 * 检查两个单词是否有公共字符
 * 时间复杂度: O(L1 + L2)
 * 空间复杂度: O(26) = O(1)
 */
bool hasCommonChars(const string& word1, const string& word2) {
    bool chars[26] = {false};

    // 记录第一个单词的字符
    for (char c : word1) {
        chars[c - 'a'] = true;
    }

    // 检查第二个单词是否有相同字符
    for (char c : word2) {
        if (chars[c - 'a']) {
            return true;
        }
    }

    return false;
}

/***
 * 方法 2: 位掩码法 (推荐)
 * 核心思想: 使用 26 位整数表示每个单词的字符集合

```

```

* 时间复杂度: O(n2 + nL)
* 空间复杂度: O(n)
*/
int maxProduct2(vector<string>& words) {
    int n = words.size();
    vector<int> masks(n, 0); // 存储每个单词的位掩码
    vector<int> lengths(n, 0); // 存储每个单词的长度

    // 预处理: 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 maxProduct = 0;

    // 比较所有单词对
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // 使用位与运算检查是否有公共字符
            if ((masks[i] & masks[j]) == 0) {
                maxProduct = max(maxProduct, lengths[i] * lengths[j]);
            }
        }
    }

    return maxProduct;
}

/***
* 方法 3: 位掩码优化版
* 优化比较过程, 减少不必要的计算
* 时间复杂度: O(n2 + nL)
* 空间复杂度: O(n)
*/
int maxProduct3(vector<string>& words) {
    int n = words.size();
    vector<int> masks(n);
    vector<int> lengths(n);

```

```

// 预处理
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 maxProduct = 0;

// 创建索引数组用于排序
vector<int> indices(n);
for (int i = 0; i < n; i++) {
    indices[i] = i;
}

// 按长度降序排序
sort(indices.begin(), indices.end(), [&](int a, int b) {
    return lengths[a] > lengths[b];
});

for (int i = 0; i < n; i++) {
    int idx1 = indices[i];

    // 如果当前最大乘积已经大于可能的最大值，提前终止
    if (lengths[idx1] * lengths[idx1] <= maxProduct) {
        break;
    }

    for (int j = i + 1; j < n; j++) {
        int idx2 = indices[j];

        if ((masks[idx1] & masks[idx2]) == 0) {
            maxProduct = max(maxProduct, lengths[idx1] * lengths[idx2]);
            break; // 由于排序，后面的长度更小，乘积不会更大
        }
    }
}

return maxProduct;

```

```

}

/***
 * 方法 4：哈希表法
 * 使用哈希表存储相同位掩码的最大长度
 * 时间复杂度: O(n2 + nL)
 * 空间复杂度: O(n)
 */
int maxProduct4(vector<string>& words) {
    unordered_map<int, int> maskToMaxLength;

    // 预处理: 对于相同的位掩码, 只保留最长的单词长度
    for (const string& word : words) {
        int mask = 0;
        for (char c : word) {
            mask |= 1 << (c - 'a');
        }
    }

    // 更新相同掩码的最大长度
    if (maskToMaxLength.find(mask) == maskToMaxLength.end() ||
        word.length() > maskToMaxLength[mask]) {
        maskToMaxLength[mask] = word.length();
    }
}

int maxProduct = 0;
vector<int> masks;
for (const auto& pair : maskToMaxLength) {
    masks.push_back(pair.first);
}
int size = masks.size();

// 比较所有不同的位掩码
for (int i = 0; i < size; i++) {
    for (int j = i + 1; j < size; j++) {
        int mask1 = masks[i];
        int mask2 = masks[j];

        if ((mask1 & mask2) == 0) {
            int product = maskToMaxLength[mask1] * maskToMaxLength[mask2];
            maxProduct = max(maxProduct, product);
        }
    }
}

```

```

    }

    return maxProduct;
}

/***
 * 方法 5：分组法
 * 按位掩码分组，优化比较过程
 * 时间复杂度：O(n2 + nL)
 * 空间复杂度：O(n)
 */
int maxProduct5(vector<string>& words) {
    int n = words.size();

    // 预处理：计算位掩码和长度
    vector<int> masks(n);
    vector<int> lengths(n);

    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 maxProduct = 0;

    // 分组比较：使用位运算优化
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // 快速检查：如果长度乘积小于当前最大值，跳过
            if (lengths[i] * lengths[j] <= maxProduct) {
                continue;
            }

            // 位运算检查公共字符
            if ((masks[i] & masks[j]) == 0) {
                maxProduct = lengths[i] * lengths[j];
            }
        }
    }
}

```

```
        return maxProduct;
    }
};

/***
 * 测试函数
 */
int main() {
    Solution solution;

    // 测试用例 1: 示例 1
    vector<string> words1 = {"abcw", "baz", "foo", "bar", "xtfn", "abcdef"};
    int result1 = solution.maxProduct2(words1);
    cout << "测试用例 1 - 输入: ";
    for (const string& word : words1) cout << word << " ";
    cout << endl;
    cout << "结果: " << result1 << " (预期: 16)" << endl;

    // 测试用例 2: 示例 2
    vector<string> words2 = {"a", "ab", "abc", "d", "cd", "bcd", "abcd"};
    int result2 = solution.maxProduct2(words2);
    cout << "测试用例 2 - 输入: ";
    for (const string& word : words2) cout << word << " ";
    cout << endl;
    cout << "结果: " << result2 << " (预期: 4)" << endl;

    // 测试用例 3: 无解情况
    vector<string> words3 = {"a", "aa", "aaa", "aaaa"};
    int result3 = solution.maxProduct2(words3);
    cout << "测试用例 3 - 输入: ";
    for (const string& word : words3) cout << word << " ";
    cout << endl;
    cout << "结果: " << result3 << " (预期: 0)" << endl;

    // 测试用例 4: 边界情况 (两个单词)
    vector<string> words4 = {"ab", "cd"};
    int result4 = solution.maxProduct2(words4);
    cout << "测试用例 4 - 输入: ";
    for (const string& word : words4) cout << word << " ";
    cout << endl;
    cout << "结果: " << result4 << " (预期: 4)" << endl;
```

```

// 性能测试

vector<string> largeWords(100);
for (int i = 0; i < 100; i++) {
    largeWords[i] = generateRandomWord(100);
}

auto start = chrono::high_resolution_clock::now();
int result5 = solution.maxProduct2(largeWords);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "性能测试 - 输入长度: " << largeWords.size() << endl;
cout << "结果: " << result5 << endl;
cout << "耗时: " << duration.count() << "ms" << endl;

// 所有方法结果对比

cout << "\n==== 所有方法结果对比 ===" << endl;
vector<string> testWords = {"abc", "def", "ghi", "jkl"};
cout << "测试单词数组: ";
for (const string& word : testWords) cout << word << " ";
cout << endl;

cout << "方法 1 (暴力法): " << solution.maxProduct1(testWords) << endl;
cout << "方法 2 (位掩码法): " << solution.maxProduct2(testWords) << endl;
cout << "方法 3 (位掩码优化): " << solution.maxProduct3(testWords) << endl;
cout << "方法 4 (哈希表法): " << solution.maxProduct4(testWords) << endl;
cout << "方法 5 (分组法): " << solution.maxProduct5(testWords) << endl;

// 复杂度分析

cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 暴力法:" << endl;
cout << " 时间复杂度: O(n^2 * L) - 会超时" << endl;
cout << " 空间复杂度: O(1)" << endl;

cout << "方法 2 - 位掩码法:" << endl;
cout << " 时间复杂度: O(n^2 + nL)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 3 - 位掩码优化:" << endl;
cout << " 时间复杂度: O(n^2 + nL)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 4 - 哈希表法:" << endl;
cout << " 时间复杂度: O(n^2 + nL)" << endl;
cout << " 空间复杂度: O(n)" << endl;

```

```

cout << "方法 5 - 分组法:" << endl;
cout << " 时间复杂度: O(n2 + nL)" << endl;
cout << " 空间复杂度: O(n)" << endl;

// 工程化考量
cout << "\n==> 工程化考量 ==>" << endl;
cout << "1. 算法选择: 方法 2 (位掩码法) 最优" << endl;
cout << "2. 性能优化: 避免 O(n2 * L) 的暴力解法" << endl;
cout << "3. 空间优化: 使用位掩码压缩存储" << endl;
cout << "4. 实际应用: 适合处理大量单词数据" << endl;

// 算法技巧总结
cout << "\n==> 算法技巧总结 ==>" << endl;
cout << "1. 位掩码技术: 26 位整数表示字符集合" << endl;
cout << "2. 位运算优化: 使用 & 运算快速检查公共字符" << endl;
cout << "3. 预处理思想: 先计算位掩码, 再进行比较" << endl;
cout << "4. 排序优化: 按长度降序排序, 提前终止" << endl;

return 0;
}

/***
 * 生成随机单词 (用于性能测试)
 */
string generateRandomWord(int length) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<char> dis('a', 'z');

    string word;
    for (int i = 0; i < length; i++) {
        word += static_cast<char>(dis(gen));
    }
    return word;
}
=====

文件: Code40_MaximumProductofWordLengths.java
=====

package class031;

```

文件: Code40\_MaximumProductofWordLengths.java

```
package class031;
```

```
import java.util.*;  
  
/**  
 * 最大单词长度乘积（位掩码优化）  
 * 测试链接: https://leetcode.cn/problems/maximum-product-of-word-lengths/  
 *  
 * 题目描述:  
 * 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。如果不存在这样的两个单词，返回  
0。  
 *  
 * 示例:  
 * 输入: ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]  
 * 输出: 16  
 * 解释: 这两个单词为 "abcw", "xtfn", 长度乘积为 4 * 4 = 16。  
 *  
 * 输入: ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]  
 * 输出: 4  
 * 解释: 这两个单词为 "ab", "cd", 长度乘积为 2 * 2 = 4。  
 *  
 * 提示:  
 * 2 <= words.length <= 1000  
 * 1 <= words[i].length <= 1000  
 * words[i] 仅包含小写字母  
 *  
 * 解题思路:  
 * 1. 暴力法: 双重循环检查所有组合（会超时）  
 * 2. 位掩码法: 使用位掩码表示每个单词的字符集合  
 * 3. 位掩码优化: 预计算位掩码和长度，优化比较过程  
 * 4. 哈希表法: 使用哈希表存储相同位掩码的最大长度  
 * 5. 分组法: 按位掩码分组，只比较不同组的单词  
 *  
 * 时间复杂度分析:  
 * - 暴力法: O(n^2 * L)，L 为单词平均长度  
 * - 位掩码法: O(n^2 + nL)  
 * - 位掩码优化: O(n^2 + nL)  
 * - 哈希表法: O(n^2 + nL)  
 * - 分组法: O(n^2 + nL)  
 *  
 * 空间复杂度分析:  
 * - 暴力法: O(1)  
 * - 位掩码法: O(n)  
 * - 位掩码优化: O(n)
```

```

* - 哈希表法: O(n)
* - 分组法: O(n)
*/
public class Code40_MaximumProductofWordLengths {

    /**
     * 方法 1: 暴力法 (不推荐, 会超时)
     * 时间复杂度: O(n2 * L), L 为单词平均长度
     * 空间复杂度: O(1)
     */
    public int maxProduct1(String[] words) {
        int maxProduct = 0;
        int n = words.length;

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (!hasCommonChars(words[i], words[j])) {
                    maxProduct = Math.max(maxProduct, words[i].length() * words[j].length());
                }
            }
        }

        return maxProduct;
    }

    /**
     * 检查两个单词是否有公共字符
     * 时间复杂度: O(L1 + L2)
     * 空间复杂度: O(26) = O(1)
     */
    private boolean hasCommonChars(String word1, String word2) {
        boolean[] chars = new boolean[26];

        // 记录第一个单词的字符
        for (char c : word1.toCharArray()) {
            chars[c - 'a'] = true;
        }

        // 检查第二个单词是否有相同字符
        for (char c : word2.toCharArray()) {
            if (chars[c - 'a']) {
                return true;
            }
        }
    }
}

```

```

    }

    return false;
}

/***
 * 方法 2：位掩码法（推荐）
 * 核心思想：使用 26 位整数表示每个单词的字符集合
 * 时间复杂度：O(n2 + nL)
 * 空间复杂度：O(n)
 */

public int maxProduct2(String[] words) {
    int n = words.length;
    int[] masks = new int[n]; // 存储每个单词的位掩码
    int[] lengths = new int[n]; // 存储每个单词的长度

    // 预处理：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].toCharArray()) {
            mask |= 1 << (c - 'a');
        }
        masks[i] = mask;
        lengths[i] = words[i].length();
    }

    int maxProduct = 0;

    // 比较所有单词对
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // 使用位与运算检查是否有公共字符
            if ((masks[i] & masks[j]) == 0) {
                maxProduct = Math.max(maxProduct, lengths[i] * lengths[j]);
            }
        }
    }

    return maxProduct;
}

/***
 * 方法 3：位掩码优化版
 */

```

```

* 优化比较过程，减少不必要的计算
* 时间复杂度: O(n2 + nL)
* 空间复杂度: O(n)
*/
public int maxProduct3(String[] words) {
    int n = words.length;
    int[] masks = new int[n];
    int[] lengths = new int[n];

    // 预处理
    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (char c : words[i].toCharArray()) {
            mask |= 1 << (c - 'a');
        }
        masks[i] = mask;
        lengths[i] = words[i].length();
    }

    int maxProduct = 0;

    // 优化: 先按长度降序排序, 可以提前终止
    Integer[] indices = new Integer[n];
    for (int i = 0; i < n; i++) {
        indices[i] = i;
    }

    // 按长度降序排序
    Arrays.sort(indices, (a, b) -> lengths[b] - lengths[a]);

    for (int i = 0; i < n; i++) {
        int idx1 = indices[i];

        // 如果当前最大乘积已经大于可能的最大值, 提前终止
        if (lengths[idx1] * lengths[idx1] <= maxProduct) {
            break;
        }

        for (int j = i + 1; j < n; j++) {
            int idx2 = indices[j];

            if ((masks[idx1] & masks[idx2]) == 0) {
                maxProduct = Math.max(maxProduct, lengths[idx1] * lengths[idx2]);
            }
        }
    }
}

```

```

        break; // 由于排序，后面的长度更小，乘积不会更大
    }
}

}

return maxProduct;
}

/***
 * 方法 4：哈希表法
 * 使用哈希表存储相同位掩码的最大长度
 * 时间复杂度：O(n2 + nL)
 * 空间复杂度：O(n)
 */
public int maxProduct4(String[] words) {
    Map<Integer, Integer> maskToMaxLength = new HashMap<>();

    // 预处理：对于相同的位掩码，只保留最长的单词长度
    for (String word : words) {
        int mask = 0;
        for (char c : word.toCharArray()) {
            mask |= 1 << (c - 'a');
        }

        // 更新相同掩码的最大长度
        maskToMaxLength.put(mask, Math.max(
            maskToMaxLength.getOrDefault(mask, 0),
            word.length()
        ));
    }

    int maxProduct = 0;
    List<Integer> masks = new ArrayList<>(maskToMaxLength.keySet());
    int size = masks.size();

    // 比较所有不同的位掩码
    for (int i = 0; i < size; i++) {
        for (int j = i + 1; j < size; j++) {
            int mask1 = masks.get(i);
            int mask2 = masks.get(j);

            if ((mask1 & mask2) == 0) {
                int product = maskToMaxLength.get(mask1) * maskToMaxLength.get(mask2);

```

```

        maxProduct = Math.max(maxProduct, product);
    }
}

return maxProduct;
}

/**
 * 方法 5：分组法
 * 按位掩码分组，优化比较过程
 * 时间复杂度：O(n2 + nL)
 * 空间复杂度：O(n)
 */
public int maxProduct5(String[] words) {
    int n = words.length;

    // 预处理：计算位掩码和长度
    int[] masks = new int[n];
    int[] lengths = new int[n];

    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (char c : words[i].toCharArray()) {
            mask |= 1 << (c - 'a');
        }
        masks[i] = mask;
        lengths[i] = words[i].length();
    }

    int maxProduct = 0;

    // 分组比较：使用位运算优化
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // 快速检查：如果长度乘积小于当前最大值，跳过
            if (lengths[i] * lengths[j] <= maxProduct) {
                continue;
            }

            // 位运算检查公共字符
            if ((masks[i] & masks[j]) == 0) {
                maxProduct = lengths[i] * lengths[j];
            }
        }
    }
}

```

```

        }
    }

}

return maxProduct;
}

/***
 * 方法 6：位掩码+排序优化
 * 结合排序和位掩码技术
 * 时间复杂度: O(n2 + nL + nlogn)
 * 空间复杂度: O(n)
 */
public int maxProduct6(String[] words) {
    int n = words.length;

    // 创建单词信息数组
    WordInfo[] wordInfos = new WordInfo[n];
    for (int i = 0; i < n; i++) {
        int mask = 0;
        for (char c : words[i].toCharArray()) {
            mask |= 1 << (c - 'a');
        }
        wordInfos[i] = new WordInfo(mask, words[i].length());
    }

    // 按长度降序排序
    Arrays.sort(wordInfos, (a, b) -> b.length - a.length);

    int maxProduct = 0;

    for (int i = 0; i < n; i++) {
        // 提前终止：如果当前单词长度的平方小于最大乘积
        if (wordInfos[i].length * wordInfos[i].length <= maxProduct) {
            break;
        }

        for (int j = i + 1; j < n; j++) {
            int product = wordInfos[i].length * wordInfos[j].length;
            if (product <= maxProduct) {
                break; // 由于排序，后面的乘积更小
            }
        }
    }
}

```

```

        if ((wordInfos[i].mask & wordInfos[j].mask) == 0) {
            maxProduct = product;
            break; // 找到当前 i 的最大可能乘积
        }
    }

}

return maxProduct;
}

/***
 * 单词信息辅助类
 */
class WordInfo {
    int mask;
    int length;

    WordInfo(int mask, int length) {
        this.mask = mask;
        this.length = length;
    }
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    Code40_MaximumProductofWordLengths solution = new Code40_MaximumProductofWordLengths();

    // 测试用例 1: 示例 1
    String[] words1 = {"abcw", "baz", "foo", "bar", "xtfn", "abcdef"};
    int result1 = solution.maxProduct2(words1);
    System.out.println("测试用例 1 - 输入: " + Arrays.toString(words1));
    System.out.println("结果: " + result1 + " (预期: 16)");

    // 测试用例 2: 示例 2
    String[] words2 = {"a", "ab", "abc", "d", "cd", "bcd", "abcd"};
    int result2 = solution.maxProduct2(words2);
    System.out.println("测试用例 2 - 输入: " + Arrays.toString(words2));
    System.out.println("结果: " + result2 + " (预期: 4)");

    // 测试用例 3: 无解情况
    String[] words3 = {"a", "aa", "aaa", "aaaa"};
}

```

```

int result3 = solution.maxProduct2(words3);
System.out.println("测试用例 3 - 输入: " + Arrays.toString(words3));
System.out.println("结果: " + result3 + " (预期: 0)");

// 测试用例 4: 边界情况 (两个单词)
String[] words4 = {"ab", "cd"};
int result4 = solution.maxProduct2(words4);
System.out.println("测试用例 4 - 输入: " + Arrays.toString(words4));
System.out.println("结果: " + result4 + " (预期: 4)");

// 性能测试
String[] largeWords = new String[100];
for (int i = 0; i < 100; i++) {
    largeWords[i] = generateRandomWord(100);
}

long startTime = System.currentTimeMillis();
int result5 = solution.maxProduct2(largeWords);
long endTime = System.currentTimeMillis();
System.out.println("性能测试 - 输入长度: " + largeWords.length);
System.out.println("结果: " + result5);
System.out.println("耗时: " + (endTime - startTime) + "ms");

// 所有方法结果对比
System.out.println("\n==== 所有方法结果对比 ====");
String[] testWords = {"abc", "def", "ghi", "jkl"};
System.out.println("测试单词数组: " + Arrays.toString(testWords));
System.out.println("方法 1 (暴力法): " + solution.maxProduct1(testWords));
System.out.println("方法 2 (位掩码法): " + solution.maxProduct2(testWords));
System.out.println("方法 3 (位掩码优化): " + solution.maxProduct3(testWords));
System.out.println("方法 4 (哈希表法): " + solution.maxProduct4(testWords));
System.out.println("方法 5 (分组法): " + solution.maxProduct5(testWords));
System.out.println("方法 6 (位掩码+排序): " + solution.maxProduct6(testWords));

// 复杂度分析
System.out.println("\n==== 复杂度分析 ====");
System.out.println("方法 1 - 暴力法:");
System.out.println("  时间复杂度: O(n^2 * L) - 会超时");
System.out.println("  空间复杂度: O(1)");

System.out.println("方法 2 - 位掩码法:");
System.out.println("  时间复杂度: O(n^2 + nL)");
System.out.println("  空间复杂度: O(n)");

```

```
System.out.println("方法 3 - 位掩码优化:");
System.out.println(" 时间复杂度: O(n2 + nL)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 4 - 哈希表法:");
System.out.println(" 时间复杂度: O(n2 + nL)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 5 - 分组法:");
System.out.println(" 时间复杂度: O(n2 + nL)");
System.out.println(" 空间复杂度: O(n)");

// 工程化考量
```

```
System.out.println("\n==== 工程化考量 ====");
System.out.println("1. 算法选择: 方法 2 (位掩码法) 最优");
System.out.println("2. 性能优化: 避免 O(n2 * L) 的暴力解法");
System.out.println("3. 空间优化: 使用位掩码压缩存储");
System.out.println("4. 实际应用: 适合处理大量单词数据");
```

```
// 算法技巧总结
```

```
System.out.println("\n==== 算法技巧总结 ====");
System.out.println("1. 位掩码技术: 26 位整数表示字符集合");
System.out.println("2. 位运算优化: 使用 & 运算快速检查公共字符");
System.out.println("3. 预处理思想: 先计算位掩码, 再进行比较");
System.out.println("4. 排序优化: 按长度降序排序, 提前终止");
```

```
}
```

```
/**
```

```
* 生成随机单词 (用于性能测试)
```

```
*/
```

```
private static String generateRandomWord(int length) {
    Random random = new Random();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < length; i++) {
        sb.append((char) ('a' + random.nextInt(26)));
    }
    return sb.toString();
}
```

```
=====
```

文件: Code40\_MaximumProductofWordLengths.py

```
=====
```

```
"""
```

最大单词长度乘积（位掩码优化）

测试链接: <https://leetcode.cn/problems/maximum-product-of-word-lengths/>

题目描述:

给定一个字符串数组 words，找到  $\text{length}(\text{word}[i]) * \text{length}(\text{word}[j])$  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。如果不存在这样的两个单词，返回 0。

解题思路:

1. 暴力法: 双重循环检查所有组合（会超时）
2. 位掩码法: 使用位掩码表示每个单词的字符集合
3. 位掩码优化: 预计算位掩码和长度，优化比较过程
4. 哈希表法: 使用哈希表存储相同位掩码的最大长度
5. 分组法: 按位掩码分组，只比较不同组的单词

时间复杂度分析:

- 暴力法:  $O(n^2 * L)$ , L 为单词平均长度
- 位掩码法:  $O(n^2 + nL)$
- 位掩码优化:  $O(n^2 + nL)$
- 哈希表法:  $O(n^2 + nL)$
- 分组法:  $O(n^2 + nL)$

空间复杂度分析:

- 暴力法:  $O(1)$
- 位掩码法:  $O(n)$
- 位掩码优化:  $O(n)$
- 哈希表法:  $O(n)$
- 分组法:  $O(n)$

```
"""
```

```
import random
```

```
from typing import List
```

```
class Solution:
```

```
    def maxProduct1(self, words: List[str]) -> int:
```

```
        """
```

方法 1: 暴力法（不推荐，会超时）

时间复杂度:  $O(n^2 * L)$ , L 为单词平均长度

空间复杂度:  $O(1)$

```
"""
```

```

max_product = 0
n = len(words)

for i in range(n):
    for j in range(i + 1, n):
        if not self.has_common_chars(words[i], words[j]):
            product = len(words[i]) * len(words[j])
            if product > max_product:
                max_product = product

return max_product

def has_common_chars(self, word1: str, word2: str) -> bool:
    """
    检查两个单词是否有公共字符
    时间复杂度: O(L1 + L2)
    空间复杂度: O(26) = O(1)
    """
    chars = [False] * 26

    # 记录第一个单词的字符
    for c in word1:
        chars[ord(c) - ord('a')] = True

    # 检查第二个单词是否有相同字符
    for c in word2:
        if chars[ord(c) - ord('a')]:
            return True

    return False

def maxProduct2(self, words: List[str]) -> int:
    """
    方法 2: 位掩码法 (推荐)
    核心思想: 使用 26 位整数表示每个单词的字符集合
    时间复杂度: O(n^2 + nL)
    空间复杂度: O(n)
    """
    n = len(words)
    masks = [0] * n # 存储每个单词的位掩码
    lengths = [0] * n # 存储每个单词的长度

    # 预处理: 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

# 比较所有单词对
for i in range(n):
    for j in range(i + 1, n):
        # 使用位与运算检查是否有公共字符
        if (masks[i] & masks[j]) == 0:
            product = lengths[i] * lengths[j]
            if product > max_product:
                max_product = product

return max_product

```

def maxProduct3(self, words: List[str]) -> int:

"""

方法 3：位掩码优化版

优化比较过程，减少不必要的计算

时间复杂度： $O(n^2 + nL)$

空间复杂度： $O(n)$

"""

```

n = len(words)
masks = [0] * n
lengths = [0] * n

```

# 预处理

```

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

# 创建索引数组用于排序

```
indices = list(range(n))
```

```

# 按长度降序排序
indices.sort(key=lambda i: lengths[i], reverse=True)

for i in range(n):
    idx1 = indices[i]

    # 如果当前最大乘积已经大于可能的最大值，提前终止
    if lengths[idx1] * lengths[idx1] <= max_product:
        break

    for j in range(i + 1, n):
        idx2 = indices[j]

        if (masks[idx1] & masks[idx2]) == 0:
            product = lengths[idx1] * lengths[idx2]
            if product > max_product:
                max_product = product
            break # 由于排序，后面的长度更小，乘积不会更大

return max_product

```

```
def maxProduct4(self, words: List[str]) -> int:
```

方法 4：哈希表法

使用哈希表存储相同位掩码的最大长度

时间复杂度： $O(n^2 + nL)$

空间复杂度： $O(n)$

"""

```
mask_to_max_length = {}
```

# 预处理：对于相同的位掩码，只保留最长的单词长度

```
for word in words:
```

```
    mask = 0
```

```
    for c in word:
```

```
        mask |= 1 << (ord(c) - ord('a'))
```

# 更新相同掩码的最大长度

```
if mask not in mask_to_max_length or len(word) > mask_to_max_length[mask]:
    mask_to_max_length[mask] = len(word)
```

```
max_product = 0
```

```
masks = list(mask_to_max_length.keys())
```

```
size = len(masks)
```

```

# 比较所有不同的位掩码
for i in range(size):
    for j in range(i + 1, size):
        mask1 = masks[i]
        mask2 = masks[j]

        if (mask1 & mask2) == 0:
            product = mask_to_max_length[mask1] * mask_to_max_length[mask2]
            if product > max_product:
                max_product = product

return max_product

```

```
def maxProduct5(self, words: List[str]) -> int:
```

```
"""

```

方法 5：分组法

按位掩码分组，优化比较过程

时间复杂度： $O(n^2 + nL)$

空间复杂度： $O(n)$

```
"""

```

```
n = len(words)
```

# 预处理：计算位掩码和长度

```
masks = [0] * n
```

```
lengths = [0] * n
```

```
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
```

# 分组比较：使用位运算优化

```
for i in range(n):
```

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

# 快速检查：如果长度乘积小于当前最大值，跳过

```
    if lengths[i] * lengths[j] <= max_product:
```

```
        continue
```

```
# 位运算检查公共字符
if (masks[i] & masks[j]) == 0:
    product = lengths[i] * lengths[j]
    if product > max_product:
        max_product = product

return max_product

def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: 示例 1
    words1 = ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]
    result1 = solution.maxProduct2(words1)
    print(f"测试用例 1 - 输入: {words1}")
    print(f"结果: {result1} (预期: 16)")

    # 测试用例 2: 示例 2
    words2 = ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]
    result2 = solution.maxProduct2(words2)
    print(f"测试用例 2 - 输入: {words2}")
    print(f"结果: {result2} (预期: 4)")

    # 测试用例 3: 无解情况
    words3 = ["a", "aa", "aaa", "aaaa"]
    result3 = solution.maxProduct2(words3)
    print(f"测试用例 3 - 输入: {words3}")
    print(f"结果: {result3} (预期: 0)")

    # 测试用例 4: 边界情况 (两个单词)
    words4 = ["ab", "cd"]
    result4 = solution.maxProduct2(words4)
    print(f"测试用例 4 - 输入: {words4}")
    print(f"结果: {result4} (预期: 4)")

    # 性能测试
    import time
    large_words = [generate_random_word(100) for _ in range(100)]

    start_time = time.time()
    result5 = solution.maxProduct2(large_words)
    end_time = time.time()
```

```

print(f"性能测试 - 输入长度: {len(large_words)}")
print(f"结果: {result5}")
print(f"耗时: {(end_time - start_time) * 1000:.2f}毫秒")

# 所有方法结果对比
print("\n==== 所有方法结果对比 ====")
test_words = ["abc", "def", "ghi", "jkl"]
print(f"测试单词数组: {test_words}")
print(f"方法 1 (暴力法): {solution.maxProduct1(test_words)}")
print(f"方法 2 (位掩码法): {solution.maxProduct2(test_words)}")
print(f"方法 3 (位掩码优化): {solution.maxProduct3(test_words)}")
print(f"方法 4 (哈希表法): {solution.maxProduct4(test_words)}")
print(f"方法 5 (分组法): {solution.maxProduct5(test_words)}")

# 复杂度分析
print("\n==== 复杂度分析 ====")
print("方法 1 - 暴力法:")
print(" 时间复杂度:  $O(n^2 * L)$  - 会超时")
print(" 空间复杂度:  $O(1)$ ")

print("方法 2 - 位掩码法:")
print(" 时间复杂度:  $O(n^2 + nL)$ ")
print(" 空间复杂度:  $O(n)$ ")

print("方法 3 - 位掩码优化:")
print(" 时间复杂度:  $O(n^2 + nL)$ ")
print(" 空间复杂度:  $O(n)$ ")

print("方法 4 - 哈希表法:")
print(" 时间复杂度:  $O(n^2 + nL)$ ")
print(" 空间复杂度:  $O(n)$ ")

print("方法 5 - 分组法:")
print(" 时间复杂度:  $O(n^2 + nL)$ ")
print(" 空间复杂度:  $O(n)$ ")

# 工程化考量
print("\n==== 工程化考量 ====")
print("1. 算法选择: 方法 2 (位掩码法) 最优")
print("2. 性能优化: 避免  $O(n^2 * L)$  的暴力解法")
print("3. 空间优化: 使用位掩码压缩存储")
print("4. 实际应用: 适合处理大量单词数据")

```

```

# 算法技巧总结
print("\n==== 算法技巧总结 ===")
print("1. 位掩码技术：26位整数表示字符集合")
print("2. 位运算优化：使用 & 运算快速检查公共字符")
print("3. 预处理思想：先计算位掩码，再进行比较")
print("4. 排序优化：按长度降序排序，提前终止")

# Python 特殊处理说明
print("\n==== Python 特殊处理说明 ===")
print("Python 整数是动态大小的，位运算仍然有效：")
print(" 使用 ord() 函数获取字符的 ASCII 码")
print(" 位运算在 Python 中自动处理大整数")
print(" 注意使用列表推导式优化代码")

def generate_random_word(length: int) -> str:
    """生成随机单词（用于性能测试）"""
    return ''.join(chr(random.randint(ord('a'), ord('z')))) for _ in range(length))

if __name__ == "__main__":
    test_solution()

```

=====

文件: Code41\_CountingBits.cpp

=====

```

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

using namespace std;

/***
 * 比特位计数（多种解法）
 * 测试链接: https://leetcode.cn/problems/counting-bits/
 *
 * 题目描述:
 * 给定一个非负整数 n，计算从 0 到 n 的每个整数的二进制表示中 1 的个数，并返回一个数组。
 *
 * 解题思路:
 * 1. 直接计算法：对每个数使用位运算计算 1 的个数
 * 2. 动态规划法：利用已知结果推导新结果
 * 3. 位运算优化：利用位运算特性快速计算

```

```

* 4. 查表法: 预计算小范围结果, 大范围复用
*
* 时间复杂度分析:
* - 直接计算法: O(nk), k 为平均位数
* - 动态规划法: O(n)
* - 位运算优化: O(n)
* - 查表法: O(n)
*
* 空间复杂度分析:
* - 直接计算法: O(1) 或 O(n) 存储结果
* - 动态规划法: O(n)
* - 位运算优化: O(n)
* - 查表法: O(n)
*/
class Solution {
public:
    /**
     * 方法 1: 直接计算法 (朴素解法)
     * 对每个数使用位运算计算 1 的个数
     * 时间复杂度: O(nk), k 为平均位数
     * 空间复杂度: O(n)
     */
    vector<int> countBits1(int n) {
        vector<int> result(n + 1);

        for (int i = 0; i <= n; i++) {
            result[i] = countOnes(i);
        }

        return result;
    }

    /**
     * 计算一个数的二进制表示中 1 的个数
     * 时间复杂度: O(k), k 为数的位数
     * 空间复杂度: O(1)
     */
    int countOnes(int num) {
        int count = 0;
        while (num != 0) {
            count += num & 1; // 检查最低位是否为 1
            num >>= 1; // 右移
        }
    }
}

```

```

    return count;
}

/***
 * 方法 2: Brian Kernighan 算法
 * 利用 num & (num-1) 快速消除最低位的 1
 * 时间复杂度: O(nk), 但 k 通常较小
 * 空间复杂度: O(n)
 */
vector<int> countBits2(int n) {
    vector<int> result(n + 1);

    for (int i = 0; i <= n; i++) {
        result[i] = countOnesBrianKernighan(i);
    }

    return result;
}

/***
 * Brian Kernighan 算法计算 1 的个数
 * 时间复杂度: O(k), k 为 1 的个数 (比位数更优)
 * 空间复杂度: O(1)
 */
int countOnesBrianKernighan(int num) {
    int count = 0;
    while (num != 0) {
        num &= num - 1; // 消除最低位的 1
        count++;
    }
    return count;
}

/***
 * 方法 3: 动态规划法 (最高有效位)
 * 利用已知结果推导新结果: result[i] = result[i - highBit] + 1
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
vector<int> countBits3(int n) {
    vector<int> result(n + 1);
    int highBit = 0; // 当前最高有效位

```

```
for (int i = 1; i <= n; i++) {  
    // 检查是否是 2 的幂（最高有效位发生变化）  
    if ((i & (i - 1)) == 0) {  
        highBit = i;  
    }  
    result[i] = result[i - highBit] + 1;  
}  
  
return result;  
}
```

```
/**  
 * 方法 4：动态规划法（最低有效位）  
 * 利用 result[i] = result[i >> 1] + (i & 1)  
 * 时间复杂度：O(n)  
 * 空间复杂度：O(n)  
 */
```

```
vector<int> countBits4(int n) {  
    vector<int> result(n + 1);  
  
    for (int i = 1; i <= n; i++) {  
        result[i] = result[i >> 1] + (i & 1);  
    }  
  
    return result;  
}
```

```
/**  
 * 方法 5：动态规划法（最低设置位）  
 * 利用 result[i] = result[i & (i-1)] + 1  
 * 时间复杂度：O(n)  
 * 空间复杂度：O(n)  
 */
```

```
vector<int> countBits5(int n) {  
    vector<int> result(n + 1);  
  
    for (int i = 1; i <= n; i++) {  
        result[i] = result[i & (i - 1)] + 1;  
    }  
  
    return result;  
}
```

```
/**  
 * 方法 6: 查表法 (适用于小范围)  
 * 预计算 0-255 的 1 的个数, 大数复用结果  
 * 时间复杂度: O(n)  
 * 空间复杂度: O(256 + n)  
 */
```

```
vector<int> countBits6(int n) {  
    // 预计算 0-255 的 1 的个数  
    vector<int> table(256);  
    for (int i = 0; i < 256; i++) {  
        table[i] = countOnesBrianKernighan(i);  
    }  
  
    vector<int> result(n + 1);  
  
    for (int i = 0; i <= n; i++) {  
        // 将 32 位整数分成 4 个字节分别查表  
        result[i] = table[i & 0xFF] +  
            table[(i >> 8) & 0xFF] +  
            table[(i >> 16) & 0xFF] +  
            table[(i >> 24) & 0xFF];  
    }  
  
    return result;  
}
```

```
/**  
 * 方法 7: 位运算并行计算  
 * 使用位运算技巧并行计算 1 的个数  
 * 时间复杂度: O(n)  
 * 空间复杂度: O(n)  
 */
```

```
vector<int> countBits7(int n) {  
    vector<int> result(n + 1);  
  
    for (int i = 0; i <= n; i++) {  
        result[i] = parallelCountOnes(i);  
    }  
  
    return result;  
}
```

```
/**
```

```

* 并行计算 1 的个数 (位运算技巧)
* 时间复杂度: O(1) 对于固定位数的整数
* 空间复杂度: O(1)
*/
int parallelCountOnes(int num) {
    // 步骤 1: 每 2 位统计 1 的个数
    num = (num & 0x55555555) + ((num >> 1) & 0x55555555);
    // 步骤 2: 每 4 位统计 1 的个数
    num = (num & 0x33333333) + ((num >> 2) & 0x33333333);
    // 步骤 3: 每 8 位统计 1 的个数
    num = (num & 0x0F0F0F0F) + ((num >> 4) & 0x0F0F0F0F);
    // 步骤 4: 每 16 位统计 1 的个数
    num = (num & 0x00FF00FF) + ((num >> 8) & 0x00FF00FF);
    // 步骤 5: 每 32 位统计 1 的个数
    num = (num & 0x0000FFFF) + ((num >> 16) & 0x0000FFFF);

    return num;
}

/***
 * 打印数组
*/
void printArray(const vector<int>& arr) {
    cout << "[";
    for (size_t i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
}

/***
 * 测试函数
*/
int main() {
    Solution solution;

    // 测试用例 1: n = 2
    int n1 = 2;
    vector<int> result1 = solution.countBits3(n1);
    cout << "测试用例 1 - n = " << n1 << endl;
    cout << "结果: ";
}

```

```
printArray(result1);
cout << "预期: [0, 1, 1]" << endl;

// 测试用例 2: n = 5
int n2 = 5;
vector<int> result2 = solution.countBits3(n2);
cout << "测试用例 2 - n = " << n2 << endl;
cout << "结果: ";
printArray(result2);
cout << "预期: [0, 1, 1, 2, 1, 2]" << endl;

// 测试用例 3: n = 0
int n3 = 0;
vector<int> result3 = solution.countBits3(n3);
cout << "测试用例 3 - n = " << n3 << endl;
cout << "结果: ";
printArray(result3);
cout << "预期: [0]" << endl;

// 性能测试
int n4 = 1000000;
auto start = chrono::high_resolution_clock::now();
vector<int> result4 = solution.countBits3(n4);
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "性能测试 - n = " << n4 << endl;
cout << "耗时: " << duration.count() << "ms" << endl;

// 所有方法结果对比
cout << "\n== 所有方法结果对比 ==" << endl;
int testN = 10;
cout << "测试 n = " << testN << endl;
cout << "方法 1 (直接计算): ";
printArray(solution.countBits1(testN));
cout << "方法 2 (Brian Kernighan): ";
printArray(solution.countBits2(testN));
cout << "方法 3 (动态规划-最高位): ";
printArray(solution.countBits3(testN));
cout << "方法 4 (动态规划-最低位): ";
printArray(solution.countBits4(testN));
cout << "方法 5 (动态规划-最低设置位): ";
printArray(solution.countBits5(testN));
cout << "方法 6 (查表法): ";
```

```
printArray(solution.countBits6(testN));
cout << "方法 7 (并行计算): ";
printArray(solution.countBits7(testN));

// 复杂度分析
cout << "\n==== 复杂度分析 ===" << endl;
cout << "方法 1 - 直接计算法:" << endl;
cout << " 时间复杂度: O(nk), k 为平均位数" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 2 - Brian Kernighan 算法:" << endl;
cout << " 时间复杂度: O(nk), k 为 1 的个数" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 3 - 动态规划(最高位):" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 4 - 动态规划(最低位):" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 5 - 动态规划(最低设置位):" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

cout << "方法 6 - 查表法:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(256 + n)" << endl;

cout << "方法 7 - 并行计算:" << endl;
cout << " 时间复杂度: O(n)" << endl;
cout << " 空间复杂度: O(n)" << endl;

// 工程化考量
cout << "\n==== 工程化考量 ===" << endl;
cout << "1. 算法选择: 方法 3/4/5 (动态规划) 最优" << endl;
cout << "2. 性能优化: 避免 O(nk) 的朴素解法" << endl;
cout << "3. 空间优化: 动态规划法空间复杂度最优" << endl;
cout << "4. 实际应用: 适合处理大规模数据" << endl;

// 算法技巧总结
cout << "\n==== 算法技巧总结 ===" << endl;
```

```
cout << "1. 动态规划思想：利用已知结果推导新结果" << endl;
cout << "2. 位运算技巧：Brian Kernighan 算法消除最低位 1" << endl;
cout << "3. 并行计算：使用位运算并行统计 1 的个数" << endl;
cout << "4. 查表优化：預计算小范围结果，大范围复用" << endl;

return 0;
}
```

---

文件：Code41\_CountingBits.java

```
=====
package class031;

import java.util.*;

/**
 * 比特位计数（多种解法）
 * 测试链接：https://leetcode.cn/problems/counting-bits/
 *
 * 题目描述：
 * 给定一个非负整数 n，计算从 0 到 n 的每个整数的二进制表示中 1 的个数，并返回一个数组。
 *
 * 示例：
 * 输入：n = 2
 * 输出：[0, 1, 1]
 *
 * 输入：n = 5
 * 输出：[0, 1, 1, 2, 1, 2]
 *
 * 解题思路：
 * 1. 直接计算法：对每个数使用位运算计算 1 的个数
 * 2. 动态规划法：利用已知结果推导新结果
 * 3. 位运算优化：利用位运算特性快速计算
 * 4. 查表法：預计算小范围结果，大范围复用
 *
 * 时间复杂度分析：
 * - 直接计算法：O(nk)，k 为平均位数
 * - 动态规划法：O(n)
 * - 位运算优化：O(n)
 * - 查表法：O(n)
 *
 * 空间复杂度分析：
```

```

* - 直接计算法: O(1) 或 O(n) 存储结果
* - 动态规划法: O(n)
* - 位运算优化: O(n)
* - 查表法: O(n)
*/
public class Code41_CountingBits {

    /**
     * 方法 1: 直接计算法 (朴素解法)
     * 对每个数使用位运算计算 1 的个数
     * 时间复杂度: O(nk), k 为平均位数
     * 空间复杂度: O(n)
     */
    public int[] countBits1(int n) {
        int[] result = new int[n + 1];

        for (int i = 0; i <= n; i++) {
            result[i] = countOnes(i);
        }

        return result;
    }

    /**
     * 计算一个数的二进制表示中 1 的个数
     * 时间复杂度: O(k), k 为数的位数
     * 空间复杂度: O(1)
     */
    private int countOnes(int num) {
        int count = 0;
        while (num != 0) {
            count += num & 1; // 检查最低位是否为 1
            num >>>= 1; // 无符号右移
        }
        return count;
    }

    /**
     * 方法 2: Brian Kernighan 算法
     * 利用 num & (num-1) 快速消除最低位的 1
     * 时间复杂度: O(nk), 但 k 通常较小
     * 空间复杂度: O(n)
     */
}

```

```

public int[] countBits2(int n) {
    int[] result = new int[n + 1];

    for (int i = 0; i <= n; i++) {
        result[i] = countOnesBrianKernighan(i);
    }

    return result;
}

/***
 * Brian Kernighan 算法计算 1 的个数
 * 时间复杂度: O(k), k 为 1 的个数 (比位数更优)
 * 空间复杂度: O(1)
 */
private int countOnesBrianKernighan(int num) {
    int count = 0;
    while (num != 0) {
        num &= num - 1; // 消除最低位的 1
        count++;
    }
    return count;
}

/***
 * 方法 3: 动态规划法 (最高有效位)
 * 利用已知结果推导新结果: result[i] = result[i - highBit] + 1
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public int[] countBits3(int n) {
    int[] result = new int[n + 1];
    int highBit = 0; // 当前最高有效位

    for (int i = 1; i <= n; i++) {
        // 检查是否是 2 的幂 (最高有效位发生变化)
        if ((i & (i - 1)) == 0) {
            highBit = i;
        }
        result[i] = result[i - highBit] + 1;
    }

    return result;
}

```

```
}

/***
 * 方法 4: 动态规划法 (最低有效位)
 * 利用 result[i] = result[i >> 1] + (i & 1)
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public int[] countBits4(int n) {
    int[] result = new int[n + 1];

    for (int i = 1; i <= n; i++) {
        result[i] = result[i >> 1] + (i & 1);
    }

    return result;
}

/***
 * 方法 5: 动态规划法 (最低设置位)
 * 利用 result[i] = result[i & (i-1)] + 1
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public int[] countBits5(int n) {
    int[] result = new int[n + 1];

    for (int i = 1; i <= n; i++) {
        result[i] = result[i & (i - 1)] + 1;
    }

    return result;
}

/***
 * 方法 6: 查表法 (适用于小范围)
 * 预计算 0-255 的 1 的个数, 大数复用结果
 * 时间复杂度: O(n)
 * 空间复杂度: O(256 + n)
 */
public int[] countBits6(int n) {
    // 预计算 0-255 的 1 的个数
    int[] table = new int[256];
```

```

for (int i = 0; i < 256; i++) {
    table[i] = countOnesBrianKernighan(i);
}

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

for (int i = 0; i <= n; i++) {
    // 将 32 位整数分成 4 个字节分别查表
    result[i] = table[i & 0xFF] +
        table[(i >>> 8) & 0xFF] +
        table[(i >>> 16) & 0xFF] +
        table[(i >>> 24) & 0xFF];
}

```

return result;

}

/\*\*

- \* 方法 7：位运算并行计算
- \* 使用位运算技巧并行计算 1 的个数
- \* 时间复杂度：O(n)
- \* 空间复杂度：O(n)

\*/

```

public int[] countBits7(int n) {
    int[] result = new int[n + 1];

    for (int i = 0; i <= n; i++) {
        result[i] = parallelCountOnes(i);
    }

    return result;
}

```

/\*\*

- \* 并行计算 1 的个数（位运算技巧）
- \* 时间复杂度：O(1) 对于固定位数的整数
- \* 空间复杂度：O(1)

\*/

```

private int parallelCountOnes(int num) {
    // 步骤 1：每 2 位统计 1 的个数
    num = (num & 0x55555555) + ((num >>> 1) & 0x55555555);
    // 步骤 2：每 4 位统计 1 的个数
    num = (num & 0x33333333) + ((num >>> 2) & 0x33333333);
}

```

```
// 步骤3: 每8位统计1的个数
num = (num & 0x0F0F0F0F) + ((num >>> 4) & 0x0F0F0F0F);
// 步骤4: 每16位统计1的个数
num = (num & 0x00FF00FF) + ((num >>> 8) & 0x00FF00FF);
// 步骤5: 每32位统计1的个数
num = (num & 0x0000FFFF) + ((num >>> 16) & 0x0000FFFF);

return num;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    Code41_CountingBits solution = new Code41_CountingBits();

    // 测试用例1: n = 2
    int n1 = 2;
    int[] result1 = solution.countBits3(n1);
    System.out.println("测试用例1 - n = " + n1);
    System.out.println("结果: " + Arrays.toString(result1));
    System.out.println("预期: [0, 1, 1]");

    // 测试用例2: n = 5
    int n2 = 5;
    int[] result2 = solution.countBits3(n2);
    System.out.println("测试用例2 - n = " + n2);
    System.out.println("结果: " + Arrays.toString(result2));
    System.out.println("预期: [0, 1, 1, 2, 1, 2]");

    // 测试用例3: n = 0
    int n3 = 0;
    int[] result3 = solution.countBits3(n3);
    System.out.println("测试用例3 - n = " + n3);
    System.out.println("结果: " + Arrays.toString(result3));
    System.out.println("预期: [0]");

    // 性能测试
    int n4 = 1000000;
    long startTime = System.currentTimeMillis();
    int[] result4 = solution.countBits3(n4);
    long endTime = System.currentTimeMillis();
    System.out.println("性能测试 - n = " + n4);
```

```
System.out.println("耗时: " + (endTime - startTime) + "ms");

// 所有方法结果对比
System.out.println("\n==== 所有方法结果对比 ===");
int testN = 10;
System.out.println("测试 n = " + testN);
System.out.println("方法 1 (直接计算): " + Arrays.toString(solution.countBits1(testN)));
System.out.println("方法 2 (Brian Kernighan): " +
Arrays.toString(solution.countBits2(testN)));
System.out.println("方法 3 (动态规划-最高位): " +
Arrays.toString(solution.countBits3(testN)));
System.out.println("方法 4 (动态规划-最低位): " +
Arrays.toString(solution.countBits4(testN)));
System.out.println("方法 5 (动态规划-最低设置位): " +
Arrays.toString(solution.countBits5(testN)));
System.out.println("方法 6 (查表法): " + Arrays.toString(solution.countBits6(testN)));
System.out.println("方法 7 (并行计算): " + Arrays.toString(solution.countBits7(testN)));

// 复杂度分析
System.out.println("\n==== 复杂度分析 ===");
System.out.println("方法 1 - 直接计算法:");
System.out.println(" 时间复杂度: O(nk), k 为平均位数");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 2 - Brian Kernighan 算法:");
System.out.println(" 时间复杂度: O(nk), k 为 1 的个数");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 3 - 动态规划(最高位):");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 4 - 动态规划(最低位):");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 5 - 动态规划(最低设置位):");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

System.out.println("方法 6 - 查表法:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(256 + n)");
```

```

System.out.println("方法 7 - 并行计算:");
System.out.println(" 时间复杂度: O(n)");
System.out.println(" 空间复杂度: O(n)");

// 工程化考量
System.out.println("\n== 工程化考量 ==");
System.out.println("1. 算法选择: 方法 3/4/5 (动态规划) 最优");
System.out.println("2. 性能优化: 避免 O(nk) 的朴素解法");
System.out.println("3. 空间优化: 动态规划法空间复杂度最优");
System.out.println("4. 实际应用: 适合处理大规模数据");

// 算法技巧总结
System.out.println("\n== 算法技巧总结 ==");
System.out.println("1. 动态规划思想: 利用已知结果推导新结果");
System.out.println("2. 位运算技巧: Brian Kernighan 算法消除最低位 1");
System.out.println("3. 并行计算: 使用位运算并行统计 1 的个数");
System.out.println("4. 查表优化: 预计算小范围结果, 大范围复用");

// 面试要点
System.out.println("\n== 面试要点 ==");
System.out.println("1. 掌握多种解法, 理解各自优缺点");
System.out.println("2. 能够分析时间空间复杂度");
System.out.println("3. 理解动态规划的状态转移方程");
System.out.println("4. 了解位运算的优化技巧");

}

}

```

文件: Code41\_CountingBits.py

"""

比特位计数 (多种解法)

测试链接: <https://leetcode.cn/problems/counting-bits/>

题目描述:

给定一个非负整数 n, 计算从 0 到 n 的每个整数的二进制表示中 1 的个数, 并返回一个数组。

解题思路:

1. 直接计算法: 对每个数使用位运算计算 1 的个数
2. 动态规划法: 利用已知结果推导新结果
3. 位运算优化: 利用位运算特性快速计算

#### 4. 查表法：预算算小范围结果，大范围复用

时间复杂度分析：

- 直接计算法:  $O(nk)$ ,  $k$  为平均位数
- 动态规划法:  $O(n)$
- 位运算优化:  $O(n)$
- 查表法:  $O(n)$

空间复杂度分析：

- 直接计算法:  $O(1)$  或  $O(n)$  存储结果
- 动态规划法:  $O(n)$
- 位运算优化:  $O(n)$
- 查表法:  $O(n)$

"""

```
from typing import List
import time

class Solution:
    def countBits1(self, n: int) -> List[int]:
        """
        方法 1：直接计算法（朴素解法）
        对每个数使用位运算计算 1 的个数
        时间复杂度:  $O(nk)$ ,  $k$  为平均位数
        空间复杂度:  $O(n)$ 
        """
        result = [0] * (n + 1)

        for i in range(n + 1):
            result[i] = self.count_ones(i)

        return result

    def count_ones(self, num: int) -> int:
        """
        计算一个数的二进制表示中 1 的个数
        时间复杂度:  $O(k)$ ,  $k$  为数的位数
        空间复杂度:  $O(1)$ 
        """
        count = 0
        while num != 0:
            count += num & 1 # 检查最低位是否为 1
            num >>= 1 # 右移
```

```

    return count

def countBits2(self, n: int) -> List[int]:
    """
    方法 2: Brian Kernighan 算法
    利用 num & (num-1) 快速消除最低位的 1
    时间复杂度: O(nk), 但 k 通常较小
    空间复杂度: O(n)
    """
    result = [0] * (n + 1)

    for i in range(n + 1):
        result[i] = self.count_ones_brian_kernighan(i)

    return result

def count_ones_brian_kernighan(self, num: int) -> int:
    """
    Brian Kernighan 算法计算 1 的个数
    时间复杂度: O(k), k 为 1 的个数 (比位数更优)
    空间复杂度: O(1)
    """
    count = 0
    while num != 0:
        num &= num - 1 # 消除最低位的 1
        count += 1
    return count

def countBits3(self, n: int) -> List[int]:
    """
    方法 3: 动态规划法 (最高有效位)
    利用已知结果推导新结果: result[i] = result[i - highBit] + 1
    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    result = [0] * (n + 1)
    high_bit = 0 # 当前最高有效位

    for i in range(1, n + 1):
        # 检查是否是 2 的幂 (最高有效位发生变化)
        if i & (i - 1) == 0:
            high_bit = i
        result[i] = result[i - high_bit] + 1

```

```

    return result

def countBits4(self, n: int) -> List[int]:
    """
    方法 4: 动态规划法 (最低有效位)
    利用 result[i] = result[i >> 1] + (i & 1)
    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    result = [0] * (n + 1)

    for i in range(1, n + 1):
        result[i] = result[i >> 1] + (i & 1)

    return result

def countBits5(self, n: int) -> List[int]:
    """
    方法 5: 动态规划法 (最低设置位)
    利用 result[i] = result[i & (i-1)] + 1
    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    result = [0] * (n + 1)

    for i in range(1, n + 1):
        result[i] = result[i & (i - 1)] + 1

    return result

def countBits6(self, n: int) -> List[int]:
    """
    方法 6: 查表法 (适用于小范围)
    预计算 0-255 的 1 的个数, 大数复用结果
    时间复杂度: O(n)
    空间复杂度: O(256 + n)
    """
    # 预计算 0-255 的 1 的个数
    table = [0] * 256
    for i in range(256):
        table[i] = self.count_ones_brian_kernighan(i)

```

```
result = [0] * (n + 1)

for i in range(n + 1):
    # 将 32 位整数分成 4 个字节分别查表
    result[i] = (table[i & 0xFF] +
                 table[(i >> 8) & 0xFF] +
                 table[(i >> 16) & 0xFF] +
                 table[(i >> 24) & 0xFF])

return result
```

```
def countBits7(self, n: int) -> List[int]:
```

```
    """
    方法 7：位运算并行计算
```

```
    使用位运算技巧并行计算 1 的个数
```

```
    时间复杂度：O(n)
```

```
    空间复杂度：O(n)
```

```
    """

result = [0] * (n + 1)
```

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

```
    result[i] = self.parallel_count_ones(i)
```

```
return result
```

```
def parallel_count_ones(self, num: int) -> int:
```

```
    """
    并行计算 1 的个数（位运算技巧）
```

```
    时间复杂度：O(1) 对于固定位数的整数
```

```
    空间复杂度：O(1)
```

```
    """

# 步骤 1：每 2 位统计 1 的个数
```

```
num = (num & 0x55555555) + ((num >> 1) & 0x55555555)
```

```
# 步骤 2：每 4 位统计 1 的个数
```

```
num = (num & 0x33333333) + ((num >> 2) & 0x33333333)
```

```
# 步骤 3：每 8 位统计 1 的个数
```

```
num = (num & 0x0F0F0F0F) + ((num >> 4) & 0x0F0F0F0F)
```

```
# 步骤 4：每 16 位统计 1 的个数
```

```
num = (num & 0x00FF00FF) + ((num >> 8) & 0x00FF00FF)
```

```
# 步骤 5：每 32 位统计 1 的个数
```

```
num = (num & 0x0000FFFF) + ((num >> 16) & 0x0000FFFF)
```

```
return num
```

```
def test_solution():
    """测试函数"""
    solution = Solution()

    # 测试用例 1: n = 2
    n1 = 2
    result1 = solution.countBits3(n1)
    print(f"测试用例 1 - n = {n1}")
    print(f"结果: {result1}")
    print("预期: [0, 1, 1]")

    # 测试用例 2: n = 5
    n2 = 5
    result2 = solution.countBits3(n2)
    print(f"测试用例 2 - n = {n2}")
    print(f"结果: {result2}")
    print("预期: [0, 1, 1, 2, 1, 2]")

    # 测试用例 3: n = 0
    n3 = 0
    result3 = solution.countBits3(n3)
    print(f"测试用例 3 - n = {n3}")
    print(f"结果: {result3}")
    print("预期: [0]")

    # 性能测试
    n4 = 1000000
    start_time = time.time()
    result4 = solution.countBits3(n4)
    end_time = time.time()
    print(f"性能测试 - n = {n4}")
    print(f"耗时: {(end_time - start_time) * 1000:.2f}毫秒")

    # 所有方法结果对比
    print("\n==== 所有方法结果对比 ====")
    test_n = 10
    print(f"测试 n = {test_n}")
    print(f"方法 1 (直接计算): {solution.countBits1(test_n)}")
    print(f"方法 2 (Brian Kernighan): {solution.countBits2(test_n)}")
    print(f"方法 3 (动态规划-最高位): {solution.countBits3(test_n)}")
    print(f"方法 4 (动态规划-最低位): {solution.countBits4(test_n)}")
    print(f"方法 5 (动态规划-最低设置位): {solution.countBits5(test_n)}")
```

```
print("方法 6 (查表法): {solution.countBits6(test_n)}")  
print("方法 7 (并行计算): {solution.countBits7(test_n)}")
```

# 复杂度分析

```
print("\n==== 复杂度分析 ===")  
print("方法 1 - 直接计算法:")  
print(" 时间复杂度: O(nk), k 为平均位数")  
print(" 空间复杂度: O(n)")
```

```
print("方法 2 - Brian Kernighan 算法:")  
print(" 时间复杂度: O(nk), k 为 1 的个数")  
print(" 空间复杂度: O(n)")
```

```
print("方法 3 - 动态规划(最高位):")  
print(" 时间复杂度: O(n)")  
print(" 空间复杂度: O(n)")
```

```
print("方法 4 - 动态规划(最低位):")  
print(" 时间复杂度: O(n)")  
print(" 空间复杂度: O(n)")
```

```
print("方法 5 - 动态规划(最低设置位):")  
print(" 时间复杂度: O(n)")  
print(" 空间复杂度: O(n)")
```

```
print("方法 6 - 查表法:")  
print(" 时间复杂度: O(n)")  
print(" 空间复杂度: O(256 + n)")
```

```
print("方法 7 - 并行计算:")  
print(" 时间复杂度: O(n)")  
print(" 空间复杂度: O(n)")
```

# 工程化考量

```
print("\n==== 工程化考量 ===")  
print("1. 算法选择: 方法 3/4/5 (动态规划) 最优")  
print("2. 性能优化: 避免 O(nk) 的朴素解法")  
print("3. 空间优化: 动态规划法空间复杂度最优")  
print("4. 实际应用: 适合处理大规模数据")
```

# 算法技巧总结

```
print("\n==== 算法技巧总结 ===")  
print("1. 动态规划思想: 利用已知结果推导新结果")
```

```
print("2. 位运算技巧: Brian Kernighan 算法消除最低位 1")
print("3. 并行计算: 使用位运算并行统计 1 的个数")
print("4. 查表优化: 预计算小范围结果, 大范围复用")

# Python 特殊处理说明
print("\n==== Python 特殊处理说明 ===")
print("Python 整数是动态大小的, 位运算仍然有效: ")
print(" 使用 >> 进行右移操作")
print(" 使用 & 进行位与操作")
print(" 注意 Python 的整数没有固定位数限制")

if __name__ == "__main__":
    test_solution()

=====
```