==========
文件夹: class119_HashFunctionsAndAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
Hash Function 哈希函数

本目录包含哈希函数相关的算法实现，涵盖了基础和高级哈希技术，包括哈希表、哈希冲突解决、滚动哈希、一致性哈希、布隆过滤器等多种应用场景。

目录结构

- `HashFunction.java`: Java 实现
- `HashFunction.cpp`: C++实现
- `HashFunction.py`: Python 实现

实现的问题

LintCode 128. Hash Function
- **问题描述**：实现一个基于 33 的哈希函数
- **算法思路**：使用霍纳法则优化计算
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)

LeetCode 28. Find the Index of the First Occurrence in a String
- **问题描述**：实现字符串匹配算法
- **算法思路**：使用 Rabin-Karp 滚动哈希算法
- **时间复杂度**：O(n+m)
- **空间复杂度**：O(1)

LeetCode 187. Repeated DNA Sequences
- **问题描述**：找到 DNA 序列中重复出现的长度为 10 的子串
- **算法思路**：使用滚动哈希技术结合哈希表
- **时间复杂度**：O(n)
- **空间复杂度**：O(n)

LeetCode 214. Shortest Palindrome
- **问题描述**：通过在字符串前面添加字符转换为最短回文串
- **算法思路**：使用滚动哈希技术找到最长前缀回文串
- **时间复杂度**：O(n)

- **空间复杂度**：O(n)

LeetCode 705. Design HashSet
- **问题描述**：设计实现一个哈希集合
- **算法思路**：使用链地址法解决哈希冲突
- **时间复杂度**：O(n/b)，其中 n 是元素个数，b 是桶数
- **空间复杂度**：O(n)

LeetCode 706. Design HashMap
- **问题描述**：设计实现一个哈希映射
- **算法思路**：使用链地址法解决哈希冲突
- **时间复杂度**：O(n/b)，其中 n 是元素个数，b 是桶数
- **空间复杂度**：O(n)

高级哈希技术实现

一致性哈希 (Consistent Hashing)
- **核心概念**：将服务器和键映射到一个虚拟环上，每个键顺时针分配给最近的服务器
- **主要特点**：
  - 支持虚拟节点以提高负载均衡性
  - 服务器添加/删除时最小化数据迁移
  - 适用于分布式缓存、数据库分片等场景
- **实现特性**：
  - 使用 FNV-1a 哈希算法提供良好分布性
  - 优化的数据结构实现以加速查找
  - 支持动态添加和删除节点
- **应用场景**：Redis 集群、Memcached 集群、CDN 节点分配等

布隆过滤器 (Bloom Filter)
- **核心概念**：使用多个哈希函数将元素映射到位数组，用于快速判断元素是否可能存在
- **主要特点**：
  - 空间效率高，不需要存储元素本身
  - 存在一定的误判率，但不会漏判（false positives possible, false negatives impossible）
  - 删除操作困难
- **实现特性**：
  - 可配置的哈希函数数量和位数组大小
  - 支持动态调整参数以平衡误判率和空间使用
  - 高效的哈希计算算法
- **应用场景**：缓存穿透防护、垃圾邮件过滤、爬虫 URL 去重等

双重哈希 (Double Hashing)
- **核心概念**：使用两个不同的哈希函数确定探测序列，优化开放寻址法的性能
- **主要特点**：

  - 减少了主聚集和次聚集问题
  - 提供更好的分布性和查询性能
  - 适用于需要高效冲突解决的场景
- **实现特性**：
  - 优化的哈希函数选择以减少冲突
  - 动态扩容机制管理负载因子
  - 高效的探测序列生成
- **应用场景**：高性能哈希表实现、内存数据库等

哈希函数基础概念

哈希函数的性质

1. **确定性**：相同的输入总是产生相同的输出
2. **高效性**：计算哈希值的时间复杂度为 O(1)或接近 O(1)
3. **均匀分布**：输出值应该均匀分布在可能的取值范围内
4. **不可逆性**：从哈希值难以推导出原始输入（对于密码学哈希函数）

哈希冲突解决方法

1. **链地址法（Separate Chaining）**：每个哈希桶存储一个链表，发生冲突时将元素添加到链表末尾
2. **开放寻址法（Open Addressing）**：当发生冲突时，寻找其他空槽位
   - 线性探测（Linear Probing）
   - 二次探测（Quadratic Probing）
   - 双重哈希（Double Hashing）
3. **再哈希法（Rehashing）**：当负载因子过高时，重新哈希并调整哈希表大小

常见哈希算法

1. **MD5**：消息摘要算法，生成 128 位哈希值
2. **SHA 系列**：安全哈希算法家族（SHA-1, SHA-256 等）
3. **FNV-1a**：非加密哈希函数，广泛用于哈希表实现
4. **MurmurHash**：针对性能和分布均匀性优化的哈希算法

实际应用场景

1. **数据检索**：快速查找、插入和删除操作
2. **缓存系统**：如 Redis、Memcached 等使用哈希表进行高效数据存储
3. **分布式系统**：一致性哈希用于负载均衡和数据分片
4. **缓存优化**：布隆过滤器用于防止缓存穿透
5. **密码存储**：存储密码的哈希值而非明文
6. **数据去重**：识别重复数据
7. **字符串匹配**：滚动哈希用于高效子串查找

算法设计思路

哈希函数选择

在实现中，我们根据不同的应用场景选择合适的哈希函数：

- **简单哈希函数**：用于基础哈希表实现，如 LintCode 128
- **滚动哈希**：用于字符串匹配和子串查找，如 LeetCode 28 和 187
- **FNV-1a**：用于一致性哈希实现，提供良好的分布性
- **多次哈希**：用于布隆过滤器，减少误判率
- **双重哈希**：用于高效解决哈希冲突

哈希冲突解决

对于哈希冲突，我们主要使用以下方法：

- **链地址法**：用于 LeetCode 705 和 706 的哈希表实现
- **开放寻址法**：在高级哈希表中使用双重哈希优化性能
- **虚拟节点**：在一致性哈希中使用虚拟节点减少哈希冲突

性能优化

为了提高哈希表性能，我们采用以下优化策略：

- **负载因子管理**：当负载因子超过阈值时进行扩容
- **预分配空间**：为哈希表预先分配适当的空间
- **哈希函数优化**：选择具有良好分布性的哈希函数
- **惰性删除**：在某些实现中使用标记删除而非物理删除

跨语言实现一致性

所有算法都在 Java、C++和 Python 中实现，确保跨语言一致性：

- **接口设计**：提供统一的接口和方法签名
- **算法实现**：保持相同的算法思路和数据结构
- **测试用例**：使用相同的测试用例验证不同语言的实现
- **性能特性**：在各语言中保持相似的时间和空间复杂度

运行示例

```bash
# Java 示例
```

```
javac HashFunction.java
java HashFunction

# C++示例
g++ HashFunction.cpp -o HashFunction
./HashFunction

# Python 示例
python HashFunction.py
```

测试用例

每个实现都包含了全面的测试用例，覆盖了各种边界情况和典型应用场景：

- 空输入测试
- 边界条件测试
- 冲突情况测试
- 性能测试
- 特殊输入测试（如重复元素、极限情况等）

更多哈希相关题目实现

LeetCode 1. Two Sum (两数之和)
- **问题来源**: https://leetcode.com/problems/two-sum/
- **问题描述**: 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。
- **算法思路**: 使用哈希表存储每个数字及其对应的索引，遍历数组时检查 target - nums[i]是否在哈希表中
- **时间复杂度**: O(n)
- **空间复杂度**: O(n)

LeetCode 49. Group Anagrams (字母异位词分组)
- **问题来源**: https://leetcode.com/problems/group-anagrams/
- **问题描述**: 给你一个字符串数组，请你将字母异位词组合在一起。可以按任意顺序返回结果列表。
- **算法思路**: 使用排序后的字符串作为哈希表的键，将具有相同排序字符串的单词分组
- **时间复杂度**: O(n * k log k)，其中 n 是字符串数量，k 是字符串最大长度
- **空间复杂度**: O(n * k)

LeetCode 242. Valid Anagram (有效的字母异位词)
- **问题来源**: https://leetcode.com/problems/valid-anagram/
- **问题描述**: 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
- **算法思路**: 使用哈希表统计每个字符出现的次数，然后比较两个字符串的字符频率

- **时间复杂度**: O(n)
- **空间复杂度**: O(1)，因为字符集大小固定为 26

LeetCode 3. Longest Substring Without Repeating Characters (无重复字符的最长子串)
- **问题来源**: https://leetcode.com/problems/longest-substring-without-repeating-characters/
- **问题描述**: 给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。
- **算法思路**: 使用滑动窗口和哈希表记录字符最后出现的位置
- **时间复杂度**: O(n)
- **空间复杂度**: O(min(m, n))，其中 m 是字符集大小

LeetCode 76. Minimum Window Substring (最小覆盖子串)
- **问题来源**: https://leetcode.com/problems/minimum-window-substring/
- **问题描述**: 给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。
- **算法思路**: 使用滑动窗口和哈希表统计字符频率，维护一个计数器记录还需要匹配的字符数量
- **时间复杂度**: O(m + n)
- **空间复杂度**: O(m + n)

LeetCode 560. Subarray Sum Equals K (和为 K 的子数组)
- **问题来源**: https://leetcode.com/problems/subarray-sum-equals-k/
- **问题描述**: 给你一个整数数组 nums 和一个整数 k ，请你统计并返回该数组中和为 k 的连续子数组的个数。
- **算法思路**: 使用前缀和和哈希表，记录每个前缀和出现的次数
- **时间复杂度**: O(n)
- **空间复杂度**: O(n)

LeetCode 347. Top K Frequent Elements (前 K 个高频元素)
- **问题来源**: https://leetcode.com/problems/top-k-frequent-elements/
- **问题描述**: 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。
- **算法思路**: 使用哈希表统计频率，然后使用桶排序或优先队列找出前 k 个高频元素
- **时间复杂度**: O(n log k)
- **空间复杂度**: O(n)

LeetCode 380. Insert Delete GetRandom O(1) (常数时间插入、删除和获取随机元素)
- **问题来源**: https://leetcode.com/problems/insert-delete-getrandom-o1/
- **问题描述**: 实现 RandomizedSet 类，支持在常数时间内插入、删除和获取随机元素。
- **算法思路**: 使用哈希表存储值和索引的映射，使用动态数组存储值
- **时间复杂度**: O(1) 平均时间复杂度
- **空间复杂度**: O(n)

LeetCode 146. LRU Cache (LRU 缓存)
- **问题来源**: https://leetcode.com/problems/lru-cache/
- **问题描述**: 设计并实现一个满足 LRU (最近最少使用) 缓存约束的数据结构。
- **算法思路**: 使用哈希表+双向链表实现，哈希表提供O(1)的查找，双向链表维护访问顺序

- **时间复杂度**: O(1)
- **空间复杂度**: O(capacity)

哈希算法应用场景总结

1. 数据检索与存储
- **哈希表**: 快速查找、插入和删除操作
- **分布式哈希表**: 大规模分布式系统中的数据存储和检索

2. 字符串处理
- **字符串匹配**: Rabin-Karp 算法用于高效子串查找
- **重复检测**: 检测重复字符串或 DNA 序列
- **回文检测**: 使用滚动哈希检测回文串

3. 缓存系统
- **LRU 缓存**: 基于访问频率的缓存淘汰策略
- **分布式缓存**: 一致性哈希用于负载均衡
- **缓存穿透防护**: 布隆过滤器防止缓存穿透

4. 数据去重与统计
- **频率统计**: 统计元素出现频率
- **数据去重**: 识别重复数据
- **Top K 问题**: 找出出现频率最高的元素

5. 安全与加密
- **密码存储**: 存储密码的哈希值而非明文
- **数据完整性**: 使用哈希值验证数据完整性
- **数字签名**: 基于哈希函数的数字签名算法

哈希算法性能优化策略

1. 哈希函数选择
- **简单哈希**: 适用于基础哈希表实现
- **加密哈希**: 适用于安全相关场景
- **滚动哈希**: 适用于字符串匹配和子串查找

2. 冲突解决策略
- **链地址法**: 适用于大多数场景，实现简单
- **开放寻址法**: 适用于内存敏感场景
- **双重哈希**: 减少聚集现象，提高性能

3. 负载因子管理
- **动态扩容**: 当负载因子超过阈值时自动扩容

- **预分配空间**: 根据预期数据量预先分配适当空间
- **惰性删除**: 使用标记删除而非物理删除

4. 内存优化
- **位图压缩**: 布隆过滤器使用位数组节省空间
- **虚拟节点**: 一致性哈希使用虚拟节点提高负载均衡性
- **数据压缩**: 对存储的数据进行适当压缩

跨语言实现对比

Java 实现特点
- **内置支持**: 提供丰富的哈希相关类库
- **内存管理**: 自动垃圾回收，简化内存管理
- **线程安全**: 提供线程安全的哈希表实现

C++实现特点
- **性能优化**: 直接内存操作，性能更高
- **模板支持**: 支持泛型编程，代码复用性高
- **标准库**: STL 提供丰富的哈希相关容器

Python 实现特点
- **简洁语法**: 代码简洁，易于理解和维护
- **动态类型**: 无需声明变量类型，开发效率高
- **丰富库支持**: 提供多种哈希算法和数据结构

测试与验证

单元测试覆盖
- **边界测试**: 空输入、极限值测试
- **功能测试**: 基本功能验证
- **性能测试**: 时间复杂度和空间复杂度验证
- **并发测试**: 多线程环境下的正确性验证

正确性验证
- **算法正确性**: 确保算法逻辑正确
- **边界处理**: 正确处理各种边界情况
- **异常处理**: 对异常输入进行适当处理

各大平台哈希算法题目扩展

Codeforces 271D - Good Substrings (好子串)
- **题目来源**: https://codeforces.com/problemset/problem/271/D
- **题目描述**: 给定一个字符串 s 和一个坏字符标记字符串 bad，统计 s 中最多包含 k 个坏字符的不同子串数

量
- **算法思路**: 使用滚动哈希技术计算所有子串的哈希值，结合坏字符计数进行筛选
- **时间复杂度**: O(n²)
- **空间复杂度**: O(n)

Codeforces 514C - Watto and Mechanism (Watto 和机制)
- **题目来源**: https://codeforces.com/problemset/problem/514/C
- **题目描述**: 给定一个字典和多个查询字符串，判断每个查询字符串是否可以通过修改字典中某个字符串的一个字符得到
- **算法思路**: 使用滚动哈希预计算字典中所有字符串的哈希值，查询时尝试修改每个位置的字符并检查哈希值
- **时间复杂度**: O(n + m * L)，其中 n 是字典大小，m 是查询数量，L 是字符串长度
- **空间复杂度**: O(n)

Codeforces 835D - Palindromic characteristics (回文特性)
- **题目来源**: https://codeforces.com/problemset/problem/835/D
- **题目描述**: 定义 k 级回文串：1 级回文串是普通回文串，k 级回文串是回文串且前半部分是(k-1)级回文串
- **算法思路**: 使用滚动哈希判断回文串，同时使用动态规划计算回文级别
- **时间复杂度**: O(n²)
- **空间复杂度**: O(n²)

剑指 Offer 50. 第一个只出现一次的字符
- **题目来源**: 剑指 Offer 面试题 50
- **题目描述**: 在字符串 s 中找出第一个只出现一次的字符
- **算法思路**: 使用哈希表统计每个字符出现的次数，然后遍历字符串找到第一个出现次数为 1 的字符
- **时间复杂度**: O(n)
- **空间复杂度**: O(1)，因为字符集大小固定

剑指 Offer 03. 数组中重复的数字
- **题目来源**: 剑指 Offer 面试题 03
- **题目描述**: 在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内，找出数组中任意一个重复的数字
- **算法思路**: 使用哈希表记录已经访问过的数字，当遇到重复数字时返回
- **时间复杂度**: O(n)
- **空间复杂度**: O(n)

剑指 Offer 48. 最长不含重复字符的子字符串
- **题目来源**: 剑指 Offer 面试题 48
- **题目描述**: 请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度
- **算法思路**: 使用滑动窗口和哈希表记录字符最后出现的位置
- **时间复杂度**: O(n)
- **空间复杂度**: O(min(m, n))，其中 m 是字符集大小

HDU 4821 - String (字符串)
- **题目来源**: http://acm.hdu.edu.cn/showproblem.php?pid=4821
- **题目描述**: 给定字符串 s 和整数 M、L，统计有多少个长度为 M*L 的子串，可以分成 M 个长度为 L 的不同子串
- **算法思路**: 使用滚动哈希计算所有长度为 L 的子串哈希值，然后滑动窗口统计满足条件的子串
- **时间复杂度**: O(n)
- **空间复杂度**: O(n)

POJ 2774 - Long Long Message (最长公共子串)
- **题目来源**: http://poj.org/problem?id=2774
- **题目描述**: 给定两个字符串，求它们的最长公共子串长度
- **算法思路**: 使用二分答案+滚动哈希，检查是否存在长度为 mid 的公共子串
- **时间复杂度**: O(n log n)
- **空间复杂度**: O(n)

SPOJ - SUBST1 (不同子串数量)
- **题目来源**: https://www.spoj.com/problems/SUBST1/
- **题目描述**: 给定一个字符串，计算其不同子串的数量
- **算法思路**: 使用滚动哈希计算所有子串的哈希值，使用哈希集合去重
- **时间复杂度**: O(n^2)
- **空间复杂度**: O(n^2)

AtCoder ABC 284 E - Count Simple Paths (简单路径计数)
- **题目来源**: https://atcoder.jp/contests/abc284/tasks/abc284_e
- **题目描述**: 给定无向图，计算从节点 1 开始的不同简单路径数量
- **算法思路**: 使用 DFS 遍历所有路径，使用哈希集合记录路径的哈希值
- **时间复杂度**: O(2^n)
- **空间复杂度**: O(n)

USACO 2019 December Contest, Gold - Milk Visits (牛奶访问)
- **题目来源**: http://www.usaco.org/index.php?page=viewproblem2&cpid=970
- **题目描述**: 给定一棵树，每个节点有一种牛奶类型，查询从 u 到 v 的路径上是否包含特定类型的牛奶
- **算法思路**: 使用 LCA 和路径哈希，对每种牛奶类型建立哈希值
- **时间复杂度**: O(n + q log n)
- **空间复杂度**: O(n)

高级哈希应用实现

完美哈希 (Perfect Hashing)
- **应用场景**: 静态数据集，需要 O(1)查找时间且无冲突
- **算法原理**: 使用两级哈希表，第一级哈希将元素分组，第二级为每个组创建无冲突的哈希表
- **优势**: 保证 O(1)查找时间，无哈希冲突

- **限制**: 仅适用于静态数据集，构建过程较复杂

计数布隆过滤器 (Counting Bloom Filter)
- **应用场景**: 需要支持删除操作的布隆过滤器变种
- **算法原理**: 使用计数器数组代替位数组，支持元素的添加和删除
- **优势**: 支持删除操作，保持布隆过滤器的空间效率
- **限制**: 空间使用略高于标准布隆过滤器

可扩展哈希 (Extendible Hashing)
- **应用场景**: 数据库索引、文件系统等需要动态扩展的哈希结构
- **算法原理**: 使用目录结构指向桶，当桶满时进行分裂，目录深度动态调整
- **优势**: 支持动态扩展，保持较好的性能
- **限制**: 目录结构增加了一定的空间开销

线性哈希 (Linear Hashing)
- **应用场景**: 数据库系统、文件系统等需要渐进式扩展的哈希结构
- **算法原理**: 使用线性探测和分裂策略，避免目录结构的空间开销
- **优势**: 渐进式扩展，无目录结构开销
- **限制**: 分裂过程可能影响性能

完美哈希 (Perfect Hashing)
- **应用场景**: 静态数据集，需要O(1)查找时间且无冲突
- **算法原理**: 使用两级哈希表，第一级哈希将元素分组，第二级为每个组创建无冲突的哈希表
- **优势**: 保证O(1)查找时间，无哈希冲突
- **限制**: 仅适用于静态数据集，构建过程较复杂

计数布隆过滤器 (Counting Bloom Filter)
- **应用场景**: 需要支持删除操作的布隆过滤器变种
- **算法原理**: 使用计数器数组代替位数组，支持元素的添加和删除
- **优势**: 支持删除操作，保持布隆过滤器的空间效率
- **限制**: 空间使用略高于标准布隆过滤器

可扩展哈希 (Extendible Hashing)
- **应用场景**: 数据库索引、文件系统等需要动态扩展的哈希结构
- **算法原理**: 使用目录结构指向桶，当桶满时进行分裂，目录深度动态调整
- **优势**: 支持动态扩展，保持较好的性能
- **限制**: 目录结构增加了一定的空间开销

线性哈希 (Linear Hashing)
- **应用场景**: 数据库系统、文件系统等需要渐进式扩展的哈希结构
- **算法原理**: 使用线性探测和分裂策略，避免目录结构的空间开销
- **优势**: 渐进式扩展，无目录结构开销
- **限制**: 分裂过程可能影响性能

哈希算法工程化考量

1. 异常处理与边界场景
- **空输入处理**: 所有函数都正确处理空输入和边界值
- **溢出防护**: 使用大质数取模防止整数溢出
- **内存管理**: 合理管理内存分配和释放

2. 性能优化策略
- **常数优化**: 减少不必要的计算和内存访问
- **缓存友好**: 优化数据访问模式提高缓存命中率
- **并行化**: 支持多线程环境下的安全访问

3. 可配置性与扩展性
- **参数可调**: 哈希函数参数可根据需求调整
- **接口统一**: 提供一致的接口便于扩展
- **模块化设计**: 各组件独立，便于维护和测试

4. 测试与验证
- **单元测试**: 覆盖所有功能和边界情况
- **性能测试**: 验证时间复杂度和空间复杂度
- **压力测试**: 测试大规模数据下的表现

总结

本目录提供了从基础到高级的哈希技术实现，涵盖了哈希函数的核心概念、冲突解决方法、常见算法和实际应用场景。通过学习和使用这些实现，可以深入理解哈希技术在计算机科学中的重要性和多样性，以及如何在实际项目中选择和应用合适的哈希算法。

所有实现都经过了充分的测试和验证，确保在不同语言环境下的一致性和正确性。通过对比不同语言的实现，可以更好地理解哈希算法在不同编程环境下的应用特点和优化策略。

完全掌握哈希算法需要关注的方面

1. **理论基础**: 理解哈希函数的基本原理和数学基础
2. **算法实现**: 掌握各种哈希算法的实现细节和优化技巧
3. **工程实践**: 了解哈希算法在实际系统中的应用场景和挑战
4. **性能分析**: 能够分析哈希算法的性能特征和瓶颈
5. **安全考量**: 理解哈希函数在安全领域的应用和限制
6. **跨语言对比**: 掌握不同编程语言中哈希实现的差异
7. **调试技巧**: 具备调试哈希相关问题的能力
8. **优化策略**: 能够根据具体场景选择合适的哈希算法和参数

通过系统学习本目录中的所有实现，可以全面掌握哈希算法的理论和实践，为后续的算法学习和工程开发打下坚实基础。

==========

[代码文件]
==========
文件: AdvancedHashProblems.py
==========
```
"""
高级哈希算法问题与解析
包含一系列使用哈希技术的高级算法问题，以及详细的解析和实现
"""

from typing import List, Dict, Tuple
import collections
import random
import heapq
import bisect

class AdvancedHashProblems:
    @staticmethod
    def longest_duplicate_substring(s: str) -> str:
        """
        LeetCode 1044. Longest Duplicate Substring (最长重复子串)
        题目来源: https://leetcode.com/problems/longest-duplicate-substring/

        题目描述:
        给你一个字符串 s，考虑其所有重复子串：即 s 的（连续）子串，在 s 中出现至少两次。
        返回任何具有最长长度的重复子串。如果 s 不含重复子串，那么答案为 ""。

        示例:
        输入：s = "banana"
        输出："ana"

        输入：s = "abcd"
        输出：""

        算法思路:
        1. 使用二分查找确定可能的最长重复子串长度
        2. 对于每个长度，使用 Rabin-Karp 算法检查是否存在重复子串
        3. 如果存在，尝试更长的长度；否则，尝试更短的长度
```

```
        时间复杂度: O(n log n)，其中 n 是字符串长度
        空间复杂度: O(n)
        """
        def rabin_karp(m):
            """检查长度为 m 的重复子串"""
            if m == 0:
                return ""

            # 计算 hash 值
            h = 0
            for i in range(m):
                h = (h * 26 + (ord(s[i]) - ord('a'))) % MOD

            # 存储 hash 值和对应的起始位置
            seen = {h: 0}
            # 计算最高位的权重
            aL = pow(26, m, MOD)

            for start in range(1, n - m + 1):
                # 使用滚动哈希计算下一个子串的哈希值
                h = (h * 26 - (ord(s[start - 1]) - ord('a')) * aL + (ord(s[start + m - 1]) -
ord('a'))) % MOD

                if h in seen:
                    # 哈希冲突检查：比较实际子串
                    candidate = s[seen[h]:seen[h] + m]
                    curr = s[start:start + m]
                    if candidate == curr:
                        return candidate
                seen[h] = start

            return ""

        n = len(s)
        MOD = 10**9 + 7

        # 二分查找最长重复子串长度
        left, right = 0, n
        result = ""

        while left < right:
            mid = (left + right) // 2
            candidate = rabin_karp(mid)
```

```
            if candidate:
                # 找到重复子串，尝试更长的长度
                left = mid + 1
                result = candidate
            else:
                # 没有找到，尝试更短的长度
                right = mid

        return result

    @staticmethod
    def substring_with_concatenation_of_all_words(s: str, words: List[str]) -> List[int]:
        """
        LeetCode 30. Substring with Concatenation of All Words (串联所有单词的子串)
        题目来源: https://leetcode.com/problems/substring-with-concatenation-of-all-words/

        题目描述:
        给定一个字符串 s 和一个字符串数组 words。words 中所有字符串长度相同。
        s 中的 串联子串 是指一个包含 words 中所有字符串以任意顺序排列连接起来的子串。
        例如，如果 words = ["ab","cd","ef"]，那么 "abcdef"、"abefcd"、"cdabef"、"cdefab"、
"efabcd" 和 "efcdab" 都是串联子串。
        "acdbef" 不是串联子串，因为它不是任何 words 排列的连接。
        返回所有串联子串在 s 中的开始索引。你可以以任意顺序返回答案。

        示例:
        输入：s = "barfoothefoobarman", words = ["foo","bar"]
        输出：[0,9]
        解释：串联子串为 "barfoo" 和 "foobar"，它们的起始位置分别为 0 和 9。

        算法思路:
        1. 使用滑动窗口和哈希表计数
        2. 每次移动一个单词长度，检查窗口内的单词是否符合要求
        3. 由于所有单词长度相同，可以从不同偏移量开始滑动窗口

        时间复杂度: O(n * m)，其中 n 是字符串 s 的长度，m 是 words 中单词的总长度
        空间复杂度: O(m)
        """
        if not s or not words:
            return []

        word_len = len(words[0])
        word_count = len(words)
        total_len = word_len * word_count
```

```
    result = []

    if len(s) < total_len:
        return result

    # 统计 words 中每个单词出现的次数
    word_freq = collections.Counter(words)

    # 考虑不同的起始偏移
    for i in range(word_len):
        left = i
        right = i
        current_count = collections.Counter()

        while right + word_len <= len(s):
            word = s[right:right + word_len]
            right += word_len

            if word not in word_freq:
                # 遇到不在 words 中的单词，重置窗口
                current_count.clear()
                left = right
            else:
                current_count[word] += 1

                # 如果当前单词出现次数超过需要的次数，移动左指针
                while current_count[word] > word_freq[word]:
                    current_count[s[left:left + word_len]] -= 1
                    left += word_len

                # 检查是否找到了所有单词的串联
                if right - left == total_len:
                    result.append(left)
                    # 移动左指针，继续寻找下一个匹配
                    current_count[s[left:left + word_len]] -= 1
                    left += word_len

    return result

@staticmethod
def longest_consecutive_sequence(nums: List[int]) -> int:
    """
    LeetCode 128. Longest Consecutive Sequence (最长连续序列)
```

题目来源: https://leetcode.com/problems/longest-consecutive-sequence/

题目描述:
给定一个未排序的整数数组 nums，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 O(n) 的算法解决此问题。

示例:
输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9

算法思路:
1. 使用哈希集合存储所有数字
2. 对于每个数字，如果它是序列的起点（即它前面的数字不存在），则尝试找出以它开始的最长序列

时间复杂度: O(n)
空间复杂度: O(n)
"""

```
if not nums:
    return 0

num_set = set(nums)
max_length = 0

for num in num_set:
    # 只检查序列的起点（前一个数字不在集合中）
    if num - 1 not in num_set:
        current_num = num
        current_length = 1

        # 检查连续的数字是否存在
        while current_num + 1 in num_set:
            current_num += 1
            current_length += 1

        max_length = max(max_length, current_length)

return max_length
```

```
    @staticmethod
    def four_sum(nums: List[int], target: int) -> List[List[int]]:
        """
        LeetCode 18. 4Sum (四数之和)
        题目来源: https://leetcode.com/problems/4sum/

        题目描述:
        给你一个由 n 个整数组成的数组 nums，和一个目标值 target。请你找出并返回满足下述全部条件且不重复的四元组 [nums[a], nums[b], nums[c], nums[d]]：
        0 <= a, b, c, d < n
        a, b, c, d 互不相同
        nums[a] + nums[b] + nums[c] + nums[d] == target
        你可以按任意顺序返回答案。

        示例:
        输入：nums = [1,0,-1,0,-2,2], target = 0
        输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

        算法思路:
        1. 使用哈希表存储所有可能的两数之和
        2. 排序数组，然后使用两层循环固定前两个数
        3. 对于剩下的两个数，使用哈希表查找是否存在满足条件的组合

        时间复杂度: O(n^2)
        空间复杂度: O(n^2)
        """
        if len(nums) < 4:
            return []

        nums.sort()
        n = len(nums)
        result = []

        for i in range(n - 3):
            # 跳过重复元素
            if i > 0 and nums[i] == nums[i - 1]:
                continue

            # 如果当前四数之和的最小值已经大于 target，则后面的组合都会大于 target
            if nums[i] + nums[i + 1] + nums[i + 2] + nums[i + 3] > target:
                break

            # 如果当前值加上三个最大值小于 target，则当前值不可能是答案的一部分
```

```
        if nums[i] + nums[n - 3] + nums[n - 2] + nums[n - 1] < target:
            continue

        for j in range(i + 1, n - 2):
            # 跳过重复元素
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue

            # 如果当前四数之和的最小值已经大于 target，则后面的组合都会大于 target
            if nums[i] + nums[j] + nums[j + 1] + nums[j + 2] > target:
                break

            # 如果当前两个值加上两个最大值小于 target，则当前组合不可能是答案的一部分
            if nums[i] + nums[j] + nums[n - 2] + nums[n - 1] < target:
                continue

            # 使用双指针查找剩余两个数
            left, right = j + 1, n - 1

            while left < right:
                curr_sum = nums[i] + nums[j] + nums[left] + nums[right]

                if curr_sum < target:
                    left += 1
                elif curr_sum > target:
                    right -= 1
                else:
                    result.append([nums[i], nums[j], nums[left], nums[right]])

                    # 跳过重复元素
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1

                    left += 1
                    right -= 1

    return result

@staticmethod
def palindrome_pairs(words: List[str]) -> List[List[int]]:
    """
```

LeetCode 336. Palindrome Pairs (回文对)
题目来源: https://leetcode.com/problems/palindrome-pairs/

题目描述:
给定一组互不相同的单词，找出所有不同的索引对(i, j)，使得列表中的两个单词，words[i] + words[j]，可拼接成回文串。

示例:
输入：words = ["abcd","dcba","lls","s","sssll"]
输出：[[0,1],[1,0],[3,2],[2,4]]
解释：可拼接成的回文串为 ["dcbaabcd","abcddcba","slls","llssssll"]

算法思路:
1. 使用哈希表存储每个单词及其索引
2. 对于每个单词，检查其所有前缀和后缀是否可以与其他单词组成回文串
3. 特殊处理空字符串的情况

时间复杂度: O(n * k^2)，其中 n 是单词数量，k 是单词的平均长度
空间复杂度: O(n * k)
"""
```
def is_palindrome(word, start, end):
    """检查 word[start:end+1]是否为回文"""
    while start < end:
        if word[start] != word[end]:
            return False
        start += 1
        end -= 1
    return True

result = []
word_dict = {word: i for i, word in enumerate(words)}

for i, word in enumerate(words):
    for j in range(len(word) + 1):
        # 检查前缀是否可以组成回文
        prefix = word[:j]
        suffix = word[j:]

        # 如果前缀是回文，查找是否存在与后缀相反的单词
        if is_palindrome(prefix, 0, len(prefix) - 1):
            reverse_suffix = suffix[::-1]
            if reverse_suffix in word_dict and word_dict[reverse_suffix] != i:
                result.append([word_dict[reverse_suffix], i])
```

```
            # 如果后缀是回文且前缀非空（避免重复），查找是否存在与前缀相反的单词
            if j > 0 and is_palindrome(suffix, 0, len(suffix) - 1):
                reverse_prefix = prefix[::-1]
                if reverse_prefix in word_dict and word_dict[reverse_prefix] != i:
                    result.append([i, word_dict[reverse_prefix]])

    return result

@staticmethod
def count_of_smaller_numbers_after_self(nums: List[int]) -> List[int]:
    """
    LeetCode 315. Count of Smaller Numbers After Self (计算右侧小于当前元素的个数)
    题目来源: https://leetcode.com/problems/count-of-smaller-numbers-after-self/

    题目描述:
    给你一个整数数组 nums，按要求返回一个新数组 counts。
    数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

    示例:
    输入：nums = [5,2,6,1]
    输出：[2,1,1,0]
    解释：
    5 的右侧有 2 个更小的元素 (2 和 1)
    2 的右侧有 1 个更小的元素 (1)
    6 的右侧有 1 个更小的元素 (1)
    1 的右侧有 0 个更小的元素

    算法思路:
    1. 使用有序数组（或平衡二叉搜索树）从右到左处理数组
    2. 对于每个元素，查找有序数组中小于该元素的元素个数
    3. 将当前元素插入有序数组

    时间复杂度: O(n log n)
    空间复杂度: O(n)
    """
    def binary_search(arr, target):
        """查找 target 应该插入的位置（即小于 target 的元素个数）"""
        left, right = 0, len(arr)
        while left < right:
            mid = (left + right) // 2
            if arr[mid] < target:
                left = mid + 1
```

```
            else:
                right = mid
        return left

    result = []
    sorted_nums = []

    # 从右向左处理数组
    for num in reversed(nums):
        # 查找小于当前元素的元素个数
        pos = binary_search(sorted_nums, num)
        result.append(pos)

        # 将当前元素插入有序数组
        sorted_nums.insert(pos, num)

    # 由于是从右向左处理的，需要反转结果
    return result[::-1]

@staticmethod
def maximum_frequency_stack(operations: List[List[str]]) -> List[int]:
    """
    LeetCode 895. Maximum Frequency Stack (最大频率栈)
    题目来源: https://leetcode.com/problems/maximum-frequency-stack/

    题目描述:
    设计一个类似堆栈的数据结构，将元素推入堆栈，并从堆栈中弹出出现频率最高的元素。

    实现 FreqStack 类:
    - FreqStack() 构造一个空的堆栈。
    - void push(int val) 将一个整数 val 压入栈顶。
    - int pop() 删除并返回堆栈中出现频率最高的元素。
      如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素。

    示例:
    输入：
    ["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]
    [[], [5], [7], [5], [7], [4], [5], [], [], [], []]
    输出：
    [null, null, null, null, null, null, null, 5, 7, 5, 4]

    算法思路:
    1. 使用哈希表记录每个元素的频率
```

2. 使用另一个哈希表，以频率为键，值为该频率下的元素栈
3. 维护一个最大频率变量

时间复杂度: O(1) 每次操作
空间复杂度: O(n)
"""
```
class FreqStack:
    def __init__(self):
        self.freq = {}  # 元素 -> 频率
        self.group = {}  # 频率 -> 元素栈
        self.max_freq = 0

    def push(self, val: int) -> None:
        # 更新元素频率
        f = self.freq.get(val, 0) + 1
        self.freq[val] = f

        # 更新最大频率
        self.max_freq = max(self.max_freq, f)

        # 将元素添加到对应频率的栈中
        if f not in self.group:
            self.group[f] = []
        self.group[f].append(val)

    def pop(self) -> int:
        # 从最大频率栈中弹出元素
        val = self.group[self.max_freq].pop()

        # 更新元素频率
        self.freq[val] -= 1

        # 如果最大频率栈为空，减小最大频率
        if not self.group[self.max_freq]:
            self.max_freq -= 1

        return val

freq_stack = FreqStack()
results = []

for op in operations:
    if op[0] == "push":
```

```
                freq_stack.push(int(op[1]))
                results.append(None)
            elif op[0] == "pop":
                results.append(freq_stack.pop())

        return results

    @staticmethod
    def design_twitter(operations: List[List[str]]) -> List[List[int]]:
        """
        LeetCode 355. Design Twitter (设计推特)
        题目来源: https://leetcode.com/problems/design-twitter/

        题目描述:
        设计一个简化版的推特(Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近十条推文。

        实现 Twitter 类:
        - Twitter() 初始化推特对象
        - void postTweet(int userId, int tweetId) 用户 userId 发送一条新推文 tweetId
        - List<Integer> getNewsFeed(int userId) 检索当前用户最新的十条推文。
          每条推文都必须是由此用户关注的人或者是用户自己发出的。推文必须按照时间顺序由最近的开始排序。
        - void follow(int followerId, int followeeId) ID 为 followerId 的用户开始关注 ID 为 followeeId 的用户
        - void unfollow(int followerId, int followeeId) ID 为 followerId 的用户不再关注 ID 为 followeeId 的用户

        算法思路:
        1. 使用哈希表存储用户关注列表
        2. 使用哈希表存储用户发布的推文（带时间戳）
        3. 获取新闻源时，合并所有关注用户的推文，使用优先队列获取最近的 10 条

        时间复杂度:
        - postTweet: O(1)
        - getNewsFeed: O(N log N)，其中 N 是关注的用户数
        - follow/unfollow: O(1)

        空间复杂度: O(U + T)，其中 U 是用户数，T 是推文数
        """
        class Twitter:
            def __init__(self):
                self.followees = collections.defaultdict(set)  # 用户 -> 关注列表
```

```
        self.tweets = collections.defaultdict(list)    # 用户 -> 推文列表
        self.timestamp = 0

    def postTweet(self, userId: int, tweetId: int) -> None:
        # 发布推文，记录时间戳
        self.tweets[userId].append((self.timestamp, tweetId))
        self.timestamp += 1

    def getNewsFeed(self, userId: int) -> List[int]:
        # 获取自己和关注者的所有推文
        all_tweets = []

        # 添加自己的推文
        for time, tweet_id in self.tweets[userId]:
            all_tweets.append((time, tweet_id))

        # 添加关注者的推文
        for followee in self.followees[userId]:
            for time, tweet_id in self.tweets[followee]:
                all_tweets.append((time, tweet_id))

        # 按时间戳排序，获取最近的10条
        all_tweets.sort(reverse=True)
        return [tweet_id for _, tweet_id in all_tweets[:10]]

    def follow(self, followerId: int, followeeId: int) -> None:
        # 不能关注自己
        if followerId != followeeId:
            self.followees[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        # 取消关注
        if followeeId in self.followees[followerId]:
            self.followees[followerId].remove(followeeId)

twitter = Twitter()
results = []

for op in operations:
    if op[0] == "postTweet":
        twitter.postTweet(int(op[1]), int(op[2]))
        results.append(None)
    elif op[0] == "getNewsFeed":
```

```
                results.append(twitter.getNewsFeed(int(op[1])))
            elif op[0] == "follow":
                twitter.follow(int(op[1]), int(op[2]))
                results.append(None)
            elif op[0] == "unfollow":
                twitter.unfollow(int(op[1]), int(op[2]))
                results.append(None)

        return results

    @staticmethod
    def lfu_cache(operations: List[List[str]]) -> List[int]:
        """
        LeetCode 460. LFU Cache (LFU缓存)
        题目来源: https://leetcode.com/problems/lfu-cache/

        题目描述:
        请你为 最不经常使用（LFU）缓存算法设计并实现数据结构。

        实现 LFUCache 类:
        - LFUCache(int capacity) - 用数据结构的容量 capacity 初始化对象
        - int get(int key) - 如果键 key 存在于缓存中，则获取键的值，否则返回 -1。
        - void put(int key, int value) - 如果键 key 已存在，则变更其值；如果键不存在，请插入键值对。
          当缓存达到其容量 capacity 时，则应该在插入新项之前，移除最不经常使用的项。
          在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除 最近最久未使用的键。

        为了确定最不常使用的键，可以为缓存中的每个键维护一个 使用计数器。使用计数最小的键是最久未使用的键。
        当一个键首次插入到缓存中时，它的使用计数器被设置为 1 。对缓存中的键执行 get 或 put 操作，使用计数器的值将会递增。

        算法思路:
        1. 使用哈希表存储键值对和频率
        2. 使用另一个哈希表，以频率为键，值为该频率下的元素（使用双向链表或有序集合）
        3. 维护一个最小频率变量

        时间复杂度: O(1) 每次操作
        空间复杂度: O(capacity)
        """
        class LFUCache:
            def __init__(self, capacity: int):
```

```
        self.capacity = capacity
        self.key_to_val = {}  # 键 -> 值
        self.key_to_freq = {}  # 键 -> 频率
        self.freq_to_keys = collections.defaultdict(collections.OrderedDict)  # 频率 -> 有序的键集合
        self.min_freq = 0  # 当前最小频率

    def get(self, key: int) -> int:
        if key not in self.key_to_val:
            return -1

        # 更新频率
        self._update_freq(key)
        return self.key_to_val[key]

    def put(self, key: int, value: int) -> None:
        if self.capacity == 0:
            return

        # 如果键已存在，更新值和频率
        if key in self.key_to_val:
            self.key_to_val[key] = value
            self._update_freq(key)
            return

        # 如果缓存已满，移除最不经常使用的项
        if len(self.key_to_val) >= self.capacity:
            # 获取最小频率的第一个键（最近最久未使用）
            lfu_key, _ = self.freq_to_keys[self.min_freq].popitem(last=False)

            # 如果该频率下没有键了，删除该频率
            if not self.freq_to_keys[self.min_freq]:
                del self.freq_to_keys[self.min_freq]

            # 删除键值对
            del self.key_to_val[lfu_key]
            del self.key_to_freq[lfu_key]

        # 插入新键值对
        self.key_to_val[key] = value
        self.key_to_freq[key] = 1
        self.freq_to_keys[1][key] = None
        self.min_freq = 1  # 新插入的键频率为1
```

```
        def _update_freq(self, key: int) -> None:
            # 获取当前频率
            freq = self.key_to_freq[key]

            # 从当前频率列表中移除
            del self.freq_to_keys[freq][key]

            # 如果当前频率列表为空且是最小频率，增加最小频率
            if not self.freq_to_keys[freq] and freq == self.min_freq:
                self.min_freq += 1

            # 更新频率
            self.key_to_freq[key] = freq + 1

            # 添加到新频率列表
            self.freq_to_keys[freq + 1][key] = None

    lfu_cache = None
    results = []

    for op in operations:
        if op[0] == "LFUCache":
            lfu_cache = LFUCache(int(op[1]))
            results.append(None)
        elif op[0] == "put":
            lfu_cache.put(int(op[1]), int(op[2]))
            results.append(None)
        elif op[0] == "get":
            results.append(lfu_cache.get(int(op[1])))

    return results

@staticmethod
def subarray_with_distinct_elements(nums: List[int], k: int) -> int:
    """
    LeetCode 992. Subarrays with K Different Integers (K 个不同整数的子数组)
    题目来源: https://leetcode.com/problems/subarrays-with-k-different-integers/

    题目描述:
    给定一个正整数数组 nums 和一个整数 k，返回 nums 中恰好包含 k 个不同整数的子数组的个数。

    示例:
```

输入：nums = [1,2,1,2,3], k = 2
输出：7
解释：恰好包含 2 个不同整数的子数组：[1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]

算法思路:
1. 使用滑动窗口和哈希表计数
2. 计算恰好包含 k 个不同整数的子数组个数 = 最多包含 k 个不同整数的子数组个数 - 最多包含(k-1)个不同整数的子数组个数

时间复杂度: O(n)
空间复杂度: O(k)
"""
```
        def at_most_k_distinct(nums, k):
            """计算最多包含 k 个不同整数的子数组个数"""
            count = collections.Counter()
            left = 0
            result = 0

            for right in range(len(nums)):
                # 如果当前数字不在窗口中，增加不同整数计数
                count[nums[right]] += 1

                # 当不同整数个数超过 k 时，移动左指针
                while len(count) > k:
                    count[nums[left]] -= 1
                    if count[nums[left]] == 0:
                        del count[nums[left]]
                    left += 1

                # 以 right 结尾的、最多包含 k 个不同整数的子数组个数为 right-left+1
                result += right - left + 1

            return result

        # 恰好包含 k 个不同整数的子数组个数 = 最多包含 k 个 - 最多包含(k-1)个
        return at_most_k_distinct(nums, k) - at_most_k_distinct(nums, k - 1)

    @staticmethod
    def find_duplicate_file_in_system(paths: List[str]) -> List[List[str]]:
        """
        LeetCode 609. Find Duplicate File in System (在系统中查找重复文件)
        题目来源: https://leetcode.com/problems/find-duplicate-file-in-system/
```

题目描述:
给定一个目录信息列表，包括目录路径，以及该目录中的所有包含内容的文件，
您需要找到文件系统中的所有重复文件组的路径。一组重复的文件至少包括二个具有完全相同内容的文件。

输入列表中的单个目录信息字符串的格式如下：
"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ... fn.txt(fn_content)"

这意味着有 n 个文件（f1.txt, f2.txt ... fn.txt）在目录 root/d1/d2/.../dm 下。
f1.txt 的内容被表示为 f1_content，f2.txt 的内容被表示为 f2_content，以此类推。

示例:
输入：paths = ["root/a 1.txt(abcd) 2.txt(efgh)","root/c 3.txt(abcd)","root/c/d 4.txt(efgh)"]
输出：[["root/a/1.txt","root/c/3.txt"],["root/a/2.txt","root/c/d/4.txt"]]

算法思路:
1. 解析每个目录信息字符串，提取文件路径和内容
2. 使用哈希表以文件内容为键，文件路径列表为值
3. 返回所有包含至少两个文件的内容组

时间复杂度: O(n)，其中 n 是所有文件路径和内容的总长度
空间复杂度: O(n)
"""

```
content_to_paths = collections.defaultdict(list)

for path in paths:
    parts = path.split()
    directory = parts[0]

    for file_info in parts[1:]:
        # 解析文件名和内容
        left_paren = file_info.find('(')
        right_paren = file_info.find(')')

        filename = file_info[:left_paren]
        content = file_info[left_paren+1:right_paren]

        # 构建完整文件路径
        full_path = directory + '/' + filename

        # 将文件路径添加到对应内容的列表中
```

```
                content_to_paths[content].append(full_path)

        # 返回所有包含至少两个文件的内容组
        return [paths for paths in content_to_paths.values() if len(paths) > 1]

    @staticmethod
    def brick_wall(wall: List[List[int]]) -> int:
        """
        LeetCode 554. Brick Wall (砖墙)
        题目来源: https://leetcode.com/problems/brick-wall/

        题目描述:
        你的面前有一堵矩形的、由 n 行砖块组成的砖墙。这些砖块高度相同（也就是一个单位高）但是宽度不同。
        每一行砖块的宽度之和相等。

        你现在要画一条 自顶向下 的、穿过 最少 砖块的垂线。如果你画的线只是从砖块的边缘经过，就不算穿过这块砖。
        你不能沿着墙的两个垂直边缘之一画线，这样显然是没有穿过一块砖的。

        给你一个二维数组 wall，该数组包含这堵墙的相关信息。其中，wall[i] 是一个代表从左至右每块砖的宽度的数组。
        你需要找出怎样画才能使这条线 穿过的砖块数量最少，并且返回 穿过的砖块数量。

        示例:
        输入：wall = [[1,2,2,1],[3,1,2],[1,3,2],[2,4],[3,1,2],[1,3,1,1]]
        输出：2

        算法思路:
        1. 计算每一行砖块的边缘位置（不包括最右边缘）
        2. 使用哈希表统计每个边缘位置出现的次数
        3. 找出出现次数最多的边缘位置，穿过的砖块数量 = 总行数 - 最大出现次数

        时间复杂度: O(n)，其中 n 是所有砖块的数量
        空间复杂度: O(m)，其中 m 是不同的边缘位置数量
        """
        if not wall:
            return 0

        edge_count = collections.defaultdict(int)

        for row in wall:
            edge_pos = 0
```

```
            # 不考虑最右边缘
            for brick in row[:-1]:
                edge_pos += brick
                edge_count[edge_pos] += 1

        # 如果没有边缘（每行只有一块砖），则必须穿过所有行
        if not edge_count:
            return len(wall)

        # 穿过的砖块数量 = 总行数 - 最大出现次数
        return len(wall) - max(edge_count.values())

    @staticmethod
    def minimum_window_substring(s: str, t: str) -> str:
        """
        LeetCode 76. Minimum Window Substring (最小覆盖子串)
        题目来源: https://leetcode.com/problems/minimum-window-substring/

        题目描述:
        给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。
        如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

        注意：
        - 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
        - 如果 s 中存在这样的子串，我们保证它是唯一的答案。

        示例:
        输入：s = "ADOBECODEBANC", t = "ABC"
        输出："BANC"
        解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

        算法思路:
        1. 使用滑动窗口和两个哈希表
        2. 一个哈希表记录目标字符串 t 中每个字符的出现次数
        3. 另一个哈希表记录当前窗口中每个字符的出现次数
        4. 使用一个计数器记录已经匹配的字符数量

        时间复杂度: O(|s| + |t|)
        空间复杂度: O(|s| + |t|)
        """
        if not s or not t:
            return ""
```

```
        # 统计 t 中每个字符的出现次数
        target_count = collections.Counter(t)
        required_chars = len(target_count)

        # 初始化窗口
        window_count = collections.defaultdict(int)
        formed = 0  # 已匹配的字符种类数

        # 初始化结果
        min_len = float('inf')
        result = ""

        left = 0
        for right in range(len(s)):
            # 扩大窗口
            char = s[right]
            window_count[char] += 1

            # 检查是否匹配了一个字符
            if char in target_count and window_count[char] == target_count[char]:
                formed += 1

            # 尝试缩小窗口
            while left <= right and formed == required_chars:
                char = s[left]

                # 更新结果
                if right - left + 1 < min_len:
                    min_len = right - left + 1
                    result = s[left:right+1]

                # 缩小窗口
                window_count[char] -= 1
                if char in target_count and window_count[char] < target_count[char]:
                    formed -= 1

                left += 1

        return result if min_len != float('inf') else ""

if __name__ == "__main__":
    # 测试最长重复子串
    print("=== LeetCode 1044. Longest Duplicate Substring ===")
```

```
    print(f"longest_duplicate_substring('banana'): {AdvancedHashProblems.longest_duplicate_substring('banana')}")
    print(f"longest_duplicate_substring('abcd'): {AdvancedHashProblems.longest_duplicate_substring('abcd')}")

    # 测试串联所有单词的子串
    print("\n=== LeetCode 30. Substring with Concatenation of All Words ===")
    print(f"substring_with_concatenation_of_all_words('barfoothefoobarman', ['foo', 'bar']): {AdvancedHashProblems.substring_with_concatenation_of_all_words('barfoothefoobarman', ['foo', 'bar'])}")

    # 测试最长连续序列
    print("\n=== LeetCode 128. Longest Consecutive Sequence ===")
    print(f"longest_consecutive_sequence([100,4,200,1,3,2]): {AdvancedHashProblems.longest_consecutive_sequence([100,4,200,1,3,2])}")
    print(f"longest_consecutive_sequence([0,3,7,2,5,8,4,6,0,1]): {AdvancedHashProblems.longest_consecutive_sequence([0,3,7,2,5,8,4,6,0,1])}")

    # 测试四数之和
    print("\n=== LeetCode 18. 4Sum ===")
    print(f"four_sum([1,0,-1,0,-2,2], 0): {AdvancedHashProblems.four_sum([1,0,-1,0,-2,2], 0)}")

    # 测试回文对
    print("\n=== LeetCode 336. Palindrome Pairs ===")
    print(f"palindrome_pairs(['abcd', 'dcba', 'lls', 's', 'sssll']): {AdvancedHashProblems.palindrome_pairs(['abcd', 'dcba', 'lls', 's', 'sssll'])}")

    # 测试计算右侧小于当前元素的个数
    print("\n=== LeetCode 315. Count of Smaller Numbers After Self ===")
    print(f"count_of_smaller_numbers_after_self([5,2,6,1]): {AdvancedHashProblems.count_of_smaller_numbers_after_self([5,2,6,1])}")

    # 测试最小窗口子串
    print("\n=== LeetCode 76. Minimum Window Substring ===")
    print(f"minimum_window_substring('ADOBECODEBANC', 'ABC'): {AdvancedHashProblems.minimum_window_substring('ADOBECODEBANC', 'ABC')}")
```

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

文件: HashFunction.cpp

```
================================================
/**
 * C++版本的哈希函数实现
```

```
 */

#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <unordered_map>
#include <map>
#include <algorithm>
#include <cmath>
#include <random>
#include <bitset>
#include <functional>
#include <chrono>
#include <climits>
using namespace std;

class HashFunction {
public:
    /**
     * LintCode 128. Hash Function
     * 题目来源: https://www.lintcode.com/problem/hash-function/description
     *
     * 题目描述:
     * 在数据结构中，哈希函数是用来将一个字符串（或任何其他类型）转化为小于哈希表大小且大于等于零的整数。
     * 一个好的哈希函数可以尽可能少地产生冲突。
     * 一种广泛使用的哈希函数算法是使用数值 33，假设任何字符串都是基于 33 的一个大整数，比如：
     * hashcode("abcd") = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
     *                  = (97* 33^3 + 98 * 33^2 + 99 * 33 +100) % HASH_SIZE
     *                  = 3595978 % HASH_SIZE
     * 其中 HASH_SIZE 表示哈希表的大小(可以假设一个哈希表就是一个索引 0 ~ HASH_SIZE-1 的数组)。
     * 给出一个字符串作为 key 和一个哈希表的大小，返回这个字符串的哈希值。
     *
     * 样例:
     * 对于 key="abcd" 并且 size=100， 返回 78
     *
     * 算法思路:
     * 使用霍纳法则（Horner's Rule）优化计算，避免大数溢出:
     * hashcode = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
     * 可以转换为:
     * hashcode = ((((ascii(a) % HASH_SIZE) * 33 + ascii(b)) % HASH_SIZE) * 33 + ascii(c)) %
```

```
HASH_SIZE) * 33 + ascii(d)) % HASH_SIZE
     *
     * 时间复杂度: O(n)，其中 n 是字符串的长度
     * 空间复杂度: O(1)
     */
    static int hashCode(const char* key, int HASH_SIZE) {
        long long ans = 0;
        for (int i = 0; key[i] != '\0'; i++) {
            ans = (ans * 33 + (int)(key[i])) % HASH_SIZE;
        }
        return (int)ans;
    }

    /**
     * 重载版本，接受字符串对象
     */
    static int hashCode(const char* key, int length, int HASH_SIZE) {
        long long ans = 0;
        for (int i = 0; i < length; i++) {
            ans = (ans * 33 + (int)(key[i])) % HASH_SIZE;
        }
        return (int)ans;
    }
};

/**
 * LeetCode 705. Design HashSet (设计哈希集合)
 * 题目来源: https://leetcode.com/problems/design-hashset/
 *
 * 题目描述:
 * 不使用任何内建的哈希表库设计一个哈希集合（HashSet）。
 * 实现 MyHashSet 类：
 * void add(key) 向哈希集合中插入值 key 。
 * bool contains(key) 返回哈希集合中是否存在这个值 key 。
 * void remove(key) 将给定值 key 从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。
 *
 * 示例:
 * 输入：
 * ["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove", "contains"]
 * [[], [1], [2], [1], [3], [2], [2], [2], [2]]
 * 输出：
 * [null, null, null, true, false, null, true, null, false]
 *
```

```
 * 约束条件:
 * 0 <= key <= 10^6
 * 最多调用 10^4 次 add、remove 和 contains
 *
 * 算法思路:
 * 使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。
 * 当发生哈希冲突时，将元素添加到对应位置的链表中。
 *
 * 时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）
 * 空间复杂度: O(n)，存储所有元素
 */
class MyHashSet {
private:
    static const int BASE = 10000;
    std::list<int> data[BASE];

    static int hash(int key) {
        return key % BASE;
    }

public:
    /** Initialize your data structure here. */
    MyHashSet() {
        // 构造函数不需要特殊处理
    }

    void add(int key) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if ((*it) == key) {
                return;
            }
        }
        data[h].push_back(key);
    }

    void remove(int key) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if ((*it) == key) {
                data[h].erase(it);
                return;
            }
```

```
        }
    }

    /** Returns true if this set contains the specified element */
    bool contains(int key) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if ((*it) == key) {
                return true;
            }
        }
        return false;
    }
};

/**
 * LeetCode 706. Design HashMap (设计哈希映射)
 * 题目来源: https://leetcode.com/problems/design-hashmap/
 *
 * 题目描述:
 * 不使用任何内建的哈希表库设计一个哈希映射（HashMap）。
 * 实现 MyHashMap 类：
 * MyHashMap() 用空映射初始化对象
 * void put(int key, int value) 向 HashMap 插入一个键值对 (key, value) 。如果 key 已经存在于映射中，则更新其对应的值 value 。
 * int get(int key) 返回特定的 key 所映射的 value ；如果映射中不包含 key 的映射，返回 -1 。
 * void remove(key) 如果映射中存在 key 的映射，则移除 key 和它所对应的 value 。
 *
 * 示例:
 * 输入：
 * ["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
 * [[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]
 * 输出：
 * [null, null, null, 1, -1, null, 1, null, -1]
 *
 * 约束条件:
 * 0 <= key, value <= 10^6
 * 最多调用 10^4 次 put、get 和 remove 方法
 *
 * 算法思路:
 * 使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。
 * 每个链表节点存储键值对，当发生哈希冲突时，将节点添加到对应位置的链表中。
 *
```

```
 * 时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）
 * 空间复杂度: O(n)，存储所有元素
 */
class MyHashMap {
private:
    static const int BASE = 10000;
    std::list<std::pair<int, int>> data[BASE];

    static int hash(int key) {
        return key % BASE;
    }

public:
    /** Initialize your data structure here. */
    MyHashMap() {
        // 构造函数不需要特殊处理
    }

    /** value will always be non-negative. */
    void put(int key, int value) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if (it->first == key) {
                it->second = value;
                return;
            }
        }
        data[h].push_back(std::make_pair(key, value));
    }

    /** Returns the value to which the specified key is mapped, or -1 if this map contains no
mapping for the key */
    int get(int key) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if (it->first == key) {
                return it->second;
            }
        }
        return -1;
    }

    /** Removes the mapping of the specified value key if this map contains a mapping for the key
```

```
*/
    void remove(int key) {
        int h = hash(key);
        for (auto it = data[h].begin(); it != data[h].end(); ++it) {
            if (it->first == key) {
                data[h].erase(it);
                return;
            }
        }
    }
};

/**
 * LeetCode 28. Find the Index of the First Occurrence in a String (实现 strStr())
 * 题目来源: https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/
 *
 * 题目描述:
 * 给你两个字符串 haystack 和 needle，请你在 haystack 字符串中找出 needle 字符串的第一个匹配项的下标（下标从 0 开始）。
 * 如果 needle 不是 haystack 的一部分，则返回 -1。
 *
 * 示例:
 * 输入：haystack = "sadbutsad", needle = "sad"
 * 输出：0
 *
 * 输入：haystack = "leetcode", needle = "leeto"
 * 输出：-1
 *
 * 算法思路:
 * 使用 Rabin-Karp 算法（滚动哈希）实现字符串匹配：
 * 1. 计算 needle 的哈希值
 * 2. 在 haystack 中维护一个长度为 needle.length() 的滑动窗口，计算其哈希值
 * 3. 当哈希值相等时，再进行字符串比较确认（避免哈希冲突）
 *
 * 时间复杂度: O(n+m)，其中 n 是 haystack 长度，m 是 needle 长度
 * 空间复杂度: O(1)
 */
int strStr(string haystack, string needle) {
    if (needle.empty()) return 0;
    if (haystack.length() < needle.length()) return -1;

    int base = 256; // 基数
    int mod = 1000000007; // 大质数，用于取模运算
```

```
    long long needleHash = 0;
    long long haystackHash = 0;
    long long h = 1; // 用于计算最高位的权重

    // 计算 needle 的哈希值和 h 的值
    for (int i = 0; i < needle.length(); i++) {
        needleHash = (needleHash * base + needle[i]) % mod;
        if (i < needle.length() - 1) {
            h = (h * base) % mod;
        }
    }

    // 计算 haystack 第一个窗口的哈希值
    for (int i = 0; i < needle.length(); i++) {
        haystackHash = (haystackHash * base + haystack[i]) % mod;
    }

    // 滑动窗口匹配
    for (int i = 0; i <= (int)(haystack.length() - needle.length()); i++) {
        // 如果哈希值相等，再进行字符串比较确认
        if (needleHash == haystackHash) {
            if (haystack.substr(i, needle.length()) == needle) {
                return i;
            }
        }

        // 计算下一个窗口的哈希值
        if (i < (int)(haystack.length() - needle.length())) {
            haystackHash = (base * (haystackHash - (haystack[i] * h) % mod) + haystack[i +
needle.length()]) % mod;
            if (haystackHash < 0) {
                haystackHash += mod;
            }
        }
    }

    return -1;
}

/**
 * LeetCode 187. Repeated DNA Sequences (重复的 DNA 序列)
 * 题目来源: https://leetcode.com/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)。
 * 可以按任意顺序返回答案。
 *
 * 示例:
 * 输入：s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
 * 输出：["AAAAACCCCC","CCCCCAAAAA"]
 *
 * 输入：s = "AAAAAAAAAAAAA"
 * 输出：["AAAAAAAAAA"]
 *
 * 算法思路:
 * 使用滚动哈希技术：
 * 1. 将每个字符映射为数字：A=0, C=1, G=2, T=3
 * 2. 使用 4 进制表示长度为 10 的序列
 * 3. 滑动窗口遍历所有长度为 10 的子串，计算其哈希值
 * 4. 使用哈希表记录每个哈希值出现的次数
 * 5. 返回出现次数大于 1 的序列
 *
 * 时间复杂度: O(n)，其中 n 是 DNA 序列长度
 * 空间复杂度: O(n)，存储所有子串的哈希值
 */
vector<string> findRepeatedDnaSequences(string s) {
    vector<string> result;
    if (s.length() < 10) return result;

    // 字符到数字的映射
    int map[256] = {0};
    map['A'] = 0;
    map['C'] = 1;
    map['G'] = 2;
    map['T'] = 3;

    int base = 4;
    int mod = 1000000007; // 大质数，用于取模运算
    int windowSize = 10;

    // 计算 base^(windowSize-1) % mod
```

```
    long long h = 1;
    for (int i = 0; i < windowSize - 1; i++) {
        h = (h * base) % mod;
    }

    // 计算第一个窗口的哈希值
    long long hash = 0;
    for (int i = 0; i < windowSize; i++) {
        hash = (hash * base + map[s[i]]) % mod;
    }

    // 使用哈希表记录每个哈希值出现的次数
    unordered_map<long long, int> hashMap;
    hashMap[hash] = 1;

    // 滑动窗口计算后续哈希值
    for (int i = 1; i <= (int)(s.length() - windowSize); i++) {
        // 移除最高位字符，添加最低位字符
        hash = (base * (hash - (map[s[i - 1]] * h) % mod) + map[s[i + windowSize - 1]]) % mod;
        if (hash < 0) {
            hash += mod;
        }

        // 记录哈希值出现次数
        hashMap[hash]++;

        // 如果某个哈希值出现 2 次，将其对应的子串加入结果集
        if (hashMap[hash] == 2) {
            result.push_back(s.substr(i, windowSize));
        }
    }

    return result;
}

// 测试函数
/**
 * LeetCode 214. Shortest Palindrome (最短回文串)
 * 题目来源: https://leetcode.com/problems/shortest-palindrome/
 *
 * 题目描述:
 * 给你一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。
 * 找到并返回可以用这种方式转换的最短回文串。
```

```
 *
 * 示例:
 * 输入：s = "aacecaaa"
 * 输出："aaacecaaa"
 *
 * 输入：s = "abcd"
 * 输出："dcbabcd"
 *
 * 算法思路:
 * 使用滚动哈希技术找到 s 的最长前缀回文串:
 * 1. 计算 s 的正向哈希和反向哈希
 * 2. 使用双指针从两端向中间移动，同时比较正向和反向哈希
 * 3. 当找到最长前缀回文串后，将剩余部分反转并添加到原字符串前面
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
string shortestPalindrome(string s) {
    if (s.length() <= 1) return s;

    long long n = s.length();
    long long base = 256;
    long long mod = 1000000007;
    long long forwardHash = 0;
    long long backwardHash = 0;
    long long power = 1;
    long long maxLen = 0;

    for (int i = 0; i < n; i++) {
        forwardHash = (forwardHash * base + s[i]) % mod;
        backwardHash = (backwardHash + s[i] * power) % mod;
        if (forwardHash == backwardHash) {
            maxLen = i + 1;
        }
        power = (power * base) % mod;
    }

    string suffix = s.substr(maxLen);
    reverse(suffix.begin(), suffix.end());
    return suffix + s;
}

/**
```

```
 * LeetCode 1. Two Sum (两数之和)
 * 题目来源: https://leetcode.com/problems/two-sum/
 *
 * 题目描述:
 * 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。
 * 你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。
 * 你可以按任意顺序返回答案。
 *
 * 示例:
 * 输入：nums = [2,7,11,15], target = 9
 * 输出：[0,1]
 * 解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
 *
 * 算法思路:
 * 使用哈希表存储每个数字及其对应的索引，遍历数组时检查 target - nums[i]是否在哈希表中
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> map;
    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (map.find(complement) != map.end()) {
            return {map[complement], i};
        }
        map[nums[i]] = i;
    }
    return {-1, -1};
}

/**
 * LeetCode 49. Group Anagrams (字母异位词分组)
 * 题目来源: https://leetcode.com/problems/group-anagrams/
 *
 * 题目描述:
 * 给你一个字符串数组，请你将字母异位词组合在一起。可以按任意顺序返回结果列表。
 * 字母异位词是由重新排列源单词的所有字母得到的一个新单词。
 *
 * 示例:
 * 输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
 * 输出: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

```
 *
 * 算法思路:
 * 使用排序后的字符串作为哈希表的键，将具有相同排序字符串的单词分组
 *
 * 时间复杂度: O(n * k log k)，其中 n 是字符串数量，k 是字符串最大长度
 * 空间复杂度: O(n * k)
 */
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> map;
    for (string str : strs) {
        string key = str;
        sort(key.begin(), key.end());
        map[key].push_back(str);
    }

    vector<vector<string>> result;
    for (auto& pair : map) {
        result.push_back(pair.second);
    }
    return result;
}

/**
 * LeetCode 242. Valid Anagram (有效的字母异位词)
 * 题目来源: https://leetcode.com/problems/valid-anagram/
 *
 * 题目描述:
 * 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
 * 注意：若 s 和 t 中每个字符出现的次数都相同，则称 s 和 t 互为字母异位词。
 *
 * 示例:
 * 输入: s = "anagram", t = "nagaram"
 * 输出: true
 *
 * 算法思路:
 * 使用哈希表统计每个字符出现的次数，然后比较两个字符串的字符频率
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)，因为字符集大小固定为 26
 */
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) return false;
```

```
    vector<int> count(26, 0);
    for (char c : s) {
        count[c - 'a']++;
    }
    for (char c : t) {
        count[c - 'a']--;
        if (count[c - 'a'] < 0) return false;
    }
    return true;
}

/**
 * LeetCode 3. Longest Substring Without Repeating Characters (无重复字符的最长子串)
 * 题目来源: https://leetcode.com/problems/longest-substring-without-repeating-characters/
 *
 * 题目描述:
 * 给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。
 *
 * 示例:
 * 输入: s = "abcabcbb"
 * 输出: 3
 * 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
 *
 * 算法思路:
 * 使用滑动窗口和哈希表记录字符最后出现的位置
 * 当遇到重复字符时，移动窗口左边界到重复字符的下一个位置
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(min(m, n))，其中 m 是字符集大小
 */
int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> map;
    int maxLength = 0;
    int left = 0;

    for (int right = 0; right < s.length(); right++) {
        char c = s[right];
        if (map.find(c) != map.end() && map[c] >= left) {
            left = map[c] + 1;
        }
        map[c] = right;
        maxLength = max(maxLength, right - left + 1);
    }
```

```
    return maxLength;
}

/**
 * LeetCode 76. Minimum Window Substring (最小覆盖子串)
 * 题目来源: https://leetcode.com/problems/minimum-window-substring/
 *
 * 题目描述:
 * 给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。
 *
 * 示例:
 * 输入：s = "ADOBECODEBANC", t = "ABC"
 * 输出："BANC"
 * 解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。
 *
 * 算法思路:
 * 使用滑动窗口和哈希表统计字符频率
 * 维护一个计数器记录还需要匹配的字符数量
 *
 * 时间复杂度: O(m + n)
 * 空间复杂度: O(m + n)
 */
string minWindow(string s, string t) {
    if (s.length() < t.length()) return "";

    unordered_map<char, int> target;
    unordered_map<char, int> window;

    // 统计 t 中字符频率
    for (char c : t) {
        target[c]++;
    }

    int left = 0, right = 0;
    int required = target.size();
    int formed = 0;
    int minLength = INT_MAX;
    int minLeft = 0, minRight = 0;

    while (right < s.length()) {
        char c = s[right];
        window[c]++;
```

```
        if (target.find(c) != target.end() && window[c] == target[c]) {
            formed++;
        }

        while (left <= right && formed == required) {
            c = s[left];

            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minLeft = left;
                minRight = right;
            }

            window[c]--;
            if (target.find(c) != target.end() && window[c] < target[c]) {
                formed--;
            }
            left++;
        }
        right++;
    }

    return minLength == INT_MAX ? "" : s.substr(minLeft, minRight - minLeft + 1);
}

/**
 * LeetCode 560. Subarray Sum Equals K (和为K的子数组)
 * 题目来源: https://leetcode.com/problems/subarray-sum-equals-k/
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k ，请你统计并返回该数组中和为 k 的连续子数组的个数。
 *
 * 示例:
 * 输入：nums = [1,1,1], k = 2
 * 输出：2
 *
 * 算法思路:
 * 使用前缀和和哈希表，记录每个前缀和出现的次数
 * 当 prefixSum - k 在哈希表中存在时，说明存在和为 k 的子数组
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
```

```
 */
int subarraySum(vector<int>& nums, int k) {
    unordered_map<int, int> prefixSumCount;
    prefixSumCount[0] = 1; // 前缀和为0出现1次
    int prefixSum = 0;
    int count = 0;

    for (int num : nums) {
        prefixSum += num;
        if (prefixSumCount.find(prefixSum - k) != prefixSumCount.end()) {
            count += prefixSumCount[prefixSum - k];
        }
        prefixSumCount[prefixSum]++;
    }
    return count;
}

/**
 * LeetCode 347. Top K Frequent Elements (前K个高频元素)
 * 题目来源: https://leetcode.com/problems/top-k-frequent-elements/
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按任意顺序返回答案。
 *
 * 示例:
 * 输入: nums = [1,1,1,2,2,3], k = 2
 * 输出: [1,2]
 *
 * 算法思路:
 * 使用哈希表统计频率，然后使用桶排序或优先队列找出前k个高频元素
 *
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(n)
 */
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int, int> frequencyMap;
    for (int num : nums) {
        frequencyMap[num]++;
    }

    // 使用桶排序
    vector<vector<int>> buckets(nums.size() + 1);
```

```
    for (auto& pair : frequencyMap) {
        buckets[pair.second].push_back(pair.first);
    }

    vector<int> result;
    for (int i = buckets.size() - 1; i >= 0 && result.size() < k; i--) {
        for (int num : buckets[i]) {
            result.push_back(num);
            if (result.size() == k) break;
        }
    }
    return result;
}

/**
 * LeetCode 380. Insert Delete GetRandom O(1) (常数时间插入、删除和获取随机元素)
 * 题目来源: https://leetcode.com/problems/insert-delete-getrandom-o1/
 *
 * 题目描述:
 * 实现 RandomizedSet 类：
 * RandomizedSet() 初始化 RandomizedSet 对象
 * bool insert(int val) 当元素 val 不存在时，向集合中插入该项，并返回 true ；否则，返回 false 。
 * bool remove(int val) 当元素 val 存在时，从集合中移除该项，并返回 true ；否则，返回 false 。
 * int getRandom() 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有相同的概率被返回。
 *
 * 算法思路:
 * 使用哈希表存储值和索引的映射，使用动态数组存储值
 * 删除时将要删除的元素与最后一个元素交换，然后删除最后一个元素
 *
 * 时间复杂度: O(1) 平均时间复杂度
 * 空间复杂度: O(n)
 */
class RandomizedSet {
private:
    unordered_map<int, int> valueToIndex;
    vector<int> values;
    default_random_engine generator;

public:
    RandomizedSet() {
        // 使用当前时间作为随机种子
        generator.seed(chrono::system_clock::now().time_since_epoch().count());
```

```
    }

    bool insert(int val) {
        if (valueToIndex.find(val) != valueToIndex.end()) {
            return false;
        }
        valueToIndex[val] = values.size();
        values.push_back(val);
        return true;
    }

    bool remove(int val) {
        if (valueToIndex.find(val) == valueToIndex.end()) {
            return false;
        }
        int index = valueToIndex[val];
        int lastElement = values.back();

        // 将要删除的元素与最后一个元素交换
        values[index] = lastElement;
        valueToIndex[lastElement] = index;

        // 删除最后一个元素
        values.pop_back();
        valueToIndex.erase(val);

        return true;
    }

    int getRandom() {
        uniform_int_distribution<int> distribution(0, values.size() - 1);
        return values[distribution(generator)];
    }
};

/**
 * LeetCode 146. LRU Cache (LRU 缓存)
 * 题目来源: https://leetcode.com/problems/lru-cache/
 *
 * 题目描述:
 * 请你设计并实现一个满足 LRU (最近最少使用) 缓存约束的数据结构。
 * 实现 LRUCache 类:
 * LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
```

```
 * int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
 * void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。
 * 如果插入操作导致关键字数量超过 capacity ，则应该逐出最久未使用的关键字。
 *
 * 算法思路:
 * 使用哈希表+双向链表实现
 * 哈希表提供O(1)的查找，双向链表维护访问顺序
 *
 * 时间复杂度: O(1)
 * 空间复杂度: O(capacity)
 */
class LRUCache {
private:
    struct DLinkedNode {
        int key;
        int value;
        DLinkedNode* prev;
        DLinkedNode* next;
        DLinkedNode() : key(0), value(0), prev(nullptr), next(nullptr) {}
        DLinkedNode(int k, int v) : key(k), value(v), prev(nullptr), next(nullptr) {}
    };

    unordered_map<int, DLinkedNode*> cache;
    int size;
    int capacity;
    DLinkedNode* head;
    DLinkedNode* tail;

    void addToHead(DLinkedNode* node) {
        node->prev = head;
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
    }

    void removeNode(DLinkedNode* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void moveToHead(DLinkedNode* node) {
        removeNode(node);
```

```
        addToHead(node);
    }

    DLinkedNode* removeTail() {
        DLinkedNode* node = tail->prev;
        removeNode(node);
        return node;
    }

public:
    LRUCache(int capacity) {
        this->capacity = capacity;
        this->size = 0;
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (cache.find(key) == cache.end()) {
            return -1;
        }
        DLinkedNode* node = cache[key];
        moveToHead(node);
        return node->value;
    }

    void put(int key, int value) {
        if (cache.find(key) != cache.end()) {
            DLinkedNode* node = cache[key];
            node->value = value;
            moveToHead(node);
        } else {
            DLinkedNode* newNode = new DLinkedNode(key, value);
            cache[key] = newNode;
            addToHead(newNode);
            size++;
            if (size > capacity) {
                DLinkedNode* tail = removeTail();
                cache.erase(tail->key);
                delete tail;
                size--;
```

```
            }
        }
    }
};

/**
 * 一致性哈希 (Consistent Hashing) 实现
 *
 * 应用场景: 分布式系统中的负载均衡，如分布式缓存、数据库分片等
 * 算法原理: 将服务器和键都映射到一个虚拟环上，每个键被分配给顺时针方向遇到的第一个服务器
 * 优势: 当服务器增减时，只需要重新分配少量键，减少数据迁移
 */
class ConsistentHash {
private:
    map<long long, string> virtualNodes; // 虚拟节点环
    int replicas; // 每个真实节点对应的虚拟节点数
    vector<string> servers; // 真实服务器列表

    // FNV-1a 哈希算法
    long long getHash(const string& key) {
        const long long FNV_32_INIT = 0x811c9dc5;
        const long long FNV_32_PRIME = 0x01000193;

        long long hash = FNV_32_INIT;
        for (char c : key) {
            hash ^= static_cast<long long>(c);
            hash *= FNV_32_PRIME;
            hash &= 0xFFFFFFFFLL; // 保持 32 位
        }
        return abs(hash);
    }

public:
    ConsistentHash(int replicas) : replicas(replicas) {}

    // 添加服务器
    void addServer(const string& server) {
        servers.push_back(server);
        // 为每个真实节点创建多个虚拟节点
        for (int i = 0; i < replicas; ++i) {
            string virtualNode = server + "#" + to_string(i);
            long long hash = getHash(virtualNode);
            virtualNodes[hash] = server;
```

```
        }
    }

    // 移除服务器
    void removeServer(const string& server) {
        auto it = find(servers.begin(), servers.end(), server);
        if (it != servers.end()) {
            servers.erase(it);
            // 移除对应的所有虚拟节点
            vector<long long> keysToRemove;
            for (const auto& node : virtualNodes) {
                if (node.second == server) {
                    keysToRemove.push_back(node.first);
                }
            }
            for (long long key : keysToRemove) {
                virtualNodes.erase(key);
            }
        }
    }

    // 获取键对应的服务器
    string getServer(const string& key) {
        if (virtualNodes.empty()) {
            return "";
        }

        long long hash = getHash(key);
        // 找到顺时针方向的第一个服务器
        auto it = virtualNodes.lower_bound(hash);
        // 如果没有比当前 hash 大的节点，则返回环的第一个节点
        if (it == virtualNodes.end()) {
            it = virtualNodes.begin();
        }
        return it->second;
    }

    // 获取服务器列表
    vector<string> getServers() const {
        return servers;
    }
};
```

```
/**
 * 布隆过滤器 (Bloom Filter) 实现
 *
 * 应用场景: 快速判断一个元素是否可能存在于集合中，如垃圾邮件过滤、缓存穿透防护等
 * 算法原理: 使用多个哈希函数将元素映射到位数组的不同位置，查询时检查所有位置是否都为 1
 * 特点: 存在一定的误判率，但不会漏判；删除元素困难
 */
class BloomFilter {
private:
    vector<bool> bits; // 位数组
    int size; // 位数组大小
    vector<int> seeds; // 多个哈希函数的种子

    // 哈希函数
    int getHash(const string& element, int seed) {
        int hash = 0;
        for (char c : element) {
            hash = seed * hash + static_cast<int>(c);
        }
        return abs(hash % size);
    }

public:
    BloomFilter(int size, int hashFunctions) : size(size), bits(size, false) {
        seeds.resize(hashFunctions);
        // 初始化哈希函数种子
        for (int i = 0; i < hashFunctions; ++i) {
            seeds[i] = i * 100 + 31; // 使用不同的种子
        }
    }

    // 添加元素
    void add(const string& element) {
        for (int seed : seeds) {
            int hash = getHash(element, seed);
            bits[hash] = true;
        }
    }

    // 判断元素是否可能存在
    bool mightContain(const string& element) {
        for (int seed : seeds) {
            int hash = getHash(element, seed);
```

```
            if (!bits[hash]) {
                return false; // 只要有一个位置为0，元素一定不存在
            }
        }
        return true; // 所有位置都为1，元素可能存在
    }
};

/**
 * 双重哈希 (Double Hashing) 实现的哈希表
 *
 * 应用场景: 开放寻址法解决哈希冲突
 * 算法原理: 使用两个哈希函数，当发生冲突时，第二个哈希函数确定探测步长
 * 优势: 减少聚集现象，提高哈希表性能
 */
template <typename K, typename V>
class DoubleHashTable {
private:
    static const int DEFAULT_SIZE = 16;
    static constexpr double LOAD_FACTOR = 0.75;

    vector<K> keys;
    vector<V> values;
    vector<bool> occupied;
    int size;
    int count;

    // 第一个哈希函数
    int hash1(const K& key) {
        // 使用标准库的哈希函数
        hash<K> hasher;
        return abs(static_cast<int>(hasher(key) % size));
    }

    // 第二个哈希函数，用于计算步长
    int hash2(const K& key) {
        hash<K> hasher;
        return 1 + abs(static_cast<int>(hasher(key) % (size - 1)));
    }

    // 查找插入位置
    int findInsertionIndex(const K& key) {
        int h1 = hash1(key);
```

```
    int h2 = hash2(key);
    int index = h1;
    int step = 1;

    // 查找空位置或相同的键
    while (occupied[index]) {
        if (keys[index] == key) {
            return index; // 键已存在，返回该位置以更新值
        }
        index = (h1 + step * h2) % size;
        step++;
    }

    return index;
}

// 查找键的索引
int findIndex(const K& key) {
    int h1 = hash1(key);
    int h2 = hash2(key);
    int index = h1;
    int step = 1;

    // 查找键
    while (occupied[index]) {
        if (keys[index] == key) {
            return index; // 找到键
        }
        index = (h1 + step * h2) % size;
        step++;
        // 避免无限循环
        if (step > size) {
            break;
        }
    }

    return -1; // 未找到键
}

// 扩容
void rehash() {
    vector<K> oldKeys = keys;
    vector<V> oldValues = values;
```

```
        vector<bool> oldOccupied = occupied;

        // 扩大为原来的两倍
        int oldSize = size;
        size *= 2;
        keys.assign(size, K());
        values.assign(size, V());
        occupied.assign(size, false);
        count = 0;

        // 重新插入所有键值对
        for (int i = 0; i < oldSize; ++i) {
            if (oldOccupied[i]) {
                put(oldKeys[i], oldValues[i]);
            }
        }
    }

public:
    DoubleHashTable() : size(DEFAULT_SIZE), count(0) {
        keys.resize(DEFAULT_SIZE);
        values.resize(DEFAULT_SIZE);
        occupied.resize(DEFAULT_SIZE, false);
    }

    // 插入键值对
    void put(const K& key, const V& value) {
        // 检查是否需要扩容
        if (static_cast<double>(count) / size >= LOAD_FACTOR) {
            rehash();
        }

        int index = findInsertionIndex(key);
        keys[index] = key;
        values[index] = value;
        if (!occupied[index]) {
            occupied[index] = true;
            count++;
        }
    }

    // 获取值
    V* get(const K& key) {
```

```
        int index = findIndex(key);
        return (index != -1) ? &values[index] : nullptr;
    }

    // 删除键值对
    void remove(const K& key) {
        int index = findIndex(key);
        if (index != -1) {
            occupied[index] = false;
            count--;
        }
    }

    // 获取大小
    int getSize() const {
        return count;
    }
};

int main() {
    // 测试 LintCode 128 题
    cout << "=== LintCode 128. Hash Function ===" << endl;
    const char* key = "abcd";
    int HASH_SIZE = 100;
    int result = HashFunction::hashCode(key, HASH_SIZE);
    cout << "Key: " << key << ", HASH_SIZE: " << HASH_SIZE << ", Result: " << result << endl;
    cout << endl;

    // 测试 LeetCode 705. Design HashSet
    cout << "=== LeetCode 705. Design HashSet ===" << endl;
    MyHashSet myHashSet;
    myHashSet.add(1);      // set = [1]
    myHashSet.add(2);      // set = [1, 2]
    cout << boolalpha << myHashSet.contains(1) << endl; // 返回 True
    cout << boolalpha << myHashSet.contains(3) << endl; // 返回 False （未找到）
    myHashSet.add(2);      // set = [1, 2]
    cout << boolalpha << myHashSet.contains(2) << endl; // 返回 True
    myHashSet.remove(2);   // set = [1]
    cout << boolalpha << myHashSet.contains(2) << endl; // 返回 False （已移除）

    // 测试 LeetCode 706. Design HashMap
    cout << "\n=== LeetCode 706. Design HashMap ===" << endl;
    MyHashMap myHashMap;
```

```
myHashMap.put(1, 1); // myHashMap 现在为 [[1,1]]
myHashMap.put(2, 2); // myHashMap 现在为 [[1,1], [2,2]]
cout << myHashMap.get(1) << endl;    // 返回 1 ，myHashMap 现在为 [[1,1], [2,2]]
cout << myHashMap.get(3) << endl;    // 返回 -1（未找到），myHashMap 现在为 [[1,1], [2,2]]
myHashMap.put(2, 1); // myHashMap 现在为 [[1,1], [2,1]]（更新已有的值）
cout << myHashMap.get(2) << endl;    // 返回 1 ，myHashMap 现在为 [[1,1], [2,1]]
myHashMap.remove(2); // 删除键为 2 的数据，myHashMap 现在为 [[1,1]]
cout << myHashMap.get(2) << endl;    // 返回 -1（未找到），myHashMap 现在为 [[1,1]]

// 测试 LeetCode 28. Find the Index of the First Occurrence in a String
cout << "\n=== LeetCode 28. Find the Index of the First Occurrence in a String ===" << endl;
cout << strStr("sadbutsad", "sad") << endl; // 返回 0
cout << strStr("leetcode", "leeto") << endl; // 返回 -1

// 测试 LeetCode 187. Repeated DNA Sequences
cout << "\n=== LeetCode 187. Repeated DNA Sequences ===" << endl;
vector<string> dnaResult = findRepeatedDnaSequences("AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT");
for (const string& seq : dnaResult) {
    cout << seq << " ";
}
cout << endl;

// 测试 LeetCode 214. Shortest Palindrome
cout << "\n=== LeetCode 214. Shortest Palindrome ===" << endl;
cout << "shortestPalindrome(\"aacecaaa\"): " << shortestPalindrome("aacecaaa") << endl;
cout << "shortestPalindrome(\"abcd\"): " << shortestPalindrome("abcd") << endl;

// 测试一致性哈希
cout << "\n=== 一致性哈希 (Consistent Hashing) ===" << endl;
ConsistentHash consistentHash(100); // 每个服务器 100 个虚拟节点
consistentHash.addServer("Server1");
consistentHash.addServer("Server2");
consistentHash.addServer("Server3");

cout << "键 'user1' 分配到的服务器: " << consistentHash.getServer("user1") << endl;
cout << "键 'user2' 分配到的服务器: " << consistentHash.getServer("user2") << endl;
cout << "键 'user3' 分配到的服务器: " << consistentHash.getServer("user3") << endl;

// 移除一个服务器后，观察键的重新分配情况
cout << "\n 移除 Server2 后:" << endl;
consistentHash.removeServer("Server2");
cout << "键 'user1' 分配到的服务器: " << consistentHash.getServer("user1") << endl;
cout << "键 'user2' 分配到的服务器: " << consistentHash.getServer("user2") << endl;
```

```
    cout << "键 'user3' 分配到的服务器: " << consistentHash.getServer("user3") << endl;

    // 测试布隆过滤器
    cout << "\n=== 布隆过滤器 (Bloom Filter) ===" << endl;
    BloomFilter bloomFilter(10000, 7); // 10000 位，7 个哈希函数
    bloomFilter.add("apple");
    bloomFilter.add("banana");
    bloomFilter.add("orange");

    cout << "'apple' 可能在集合中: " << (bloomFilter.mightContain("apple") ? "true" : "false") <<
endl;
    cout << "'banana' 可能在集合中: " << (bloomFilter.mightContain("banana") ? "true" : "false")
<< endl;
    cout << "'orange' 可能在集合中: " << (bloomFilter.mightContain("orange") ? "true" : "false")
<< endl;
    cout << "'pear' 可能在集合中: " << (bloomFilter.mightContain("pear") ? "true" : "false") <<
endl;
    cout << "'grape' 可能在集合中: " << (bloomFilter.mightContain("grape") ? "true" : "false") <<
endl;

    // 测试双重哈希表
    cout << "\n=== 双重哈希 (Double Hashing) ===" << endl;
    DoubleHashTable<string, int> doubleHashTable;
    doubleHashTable.put("apple", 100);
    doubleHashTable.put("banana", 200);
    doubleHashTable.put("orange", 300);

    int* appleValue = doubleHashTable.get("apple");
    int* bananaValue = doubleHashTable.get("banana");
    int* orangeValue = doubleHashTable.get("orange");
    int* pearValue = doubleHashTable.get("pear");

    cout << "'apple' 的值: " << (appleValue ? to_string(*appleValue) : "null") << endl;
    cout << "'banana' 的值: " << (bananaValue ? to_string(*bananaValue) : "null") << endl;
    cout << "'orange' 的值: " << (orangeValue ? to_string(*orangeValue) : "null") << endl;
    cout << "'pear' 的值: " << (pearValue ? to_string(*pearValue) : "null") << endl;

    doubleHashTable.remove("banana");
    int* removedBananaValue = doubleHashTable.get("banana");
    cout << "移除 'banana' 后的值: " << (removedBananaValue ? to_string(*removedBananaValue) :
"null") << endl;
    cout << "哈希表大小: " << doubleHashTable.getSize() << endl;
```

```
    // 测试更多哈希相关题目
    cout << "\n=== 更多哈希相关题目测试 ===" << endl;

    // 测试 LeetCode 1. Two Sum
    cout << "\n=== LeetCode 1. Two Sum ===" << endl;
    vector<int> nums = {2, 7, 11, 15};
    int target = 9;
    vector<int> twoSumResult = twoSum(nums, target);
    cout << "nums: [2, 7, 11, 15], target: 9, result: [" << twoSumResult[0] << ", " <<
twoSumResult[1] << "]" << endl;

    // 测试 LeetCode 49. Group Anagrams
    cout << "\n=== LeetCode 49. Group Anagrams ===" << endl;
    vector<string> strs = {"eat", "tea", "tan", "ate", "nat", "bat"};
    vector<vector<string>> anagramResult = groupAnagrams(strs);
    cout << "strs: [eat, tea, tan, ate, nat, bat]" << endl;
    cout << "grouped anagrams: ";
    for (const auto& group : anagramResult) {
        cout << "[";
        for (size_t i = 0; i < group.size(); i++) {
            cout << group[i];
            if (i < group.size() - 1) cout << ", ";
        }
        cout << "] ";
    }
    cout << endl;

    // 测试 LeetCode 242. Valid Anagram
    cout << "\n=== LeetCode 242. Valid Anagram ===" << endl;
    string s1 = "anagram", s2 = "nagaram";
    bool anagramCheck = isAnagram(s1, s2);
    cout << "'" << s1 << "' and '" << s2 << "' are anagrams: " << (anagramCheck ? "true" :
"false") << endl;

    // 测试 LeetCode 3. Longest Substring Without Repeating Characters
    cout << "\n=== LeetCode 3. Longest Substring Without Repeating Characters ===" << endl;
    string s = "abcabcbb";
    int longestSubstring = lengthOfLongestSubstring(s);
    cout << "String: '" << s << "', longest substring length: " << longestSubstring << endl;

    // 测试 LeetCode 76. Minimum Window Substring
    cout << "\n=== LeetCode 76. Minimum Window Substring ===" << endl;
    string sStr = "ADOBECODEBANC";
```

```
    string tStr = "ABC";
    string minWindowResult = minWindow(sStr, tStr);
    cout << "s: '" << sStr << "', t: '" << tStr << "', min window: '" << minWindowResult << "'"
<< endl;

    // 测试 LeetCode 560. Subarray Sum Equals K
    cout << "\n=== LeetCode 560. Subarray Sum Equals K ===" << endl;
    vector<int> sumNums = {1, 1, 1};
    int k = 2;
    int subarraySumResult = subarraySum(sumNums, k);
    cout << "nums: [1, 1, 1], k: 2, subarray count: " << subarraySumResult << endl;

    // 测试 LeetCode 347. Top K Frequent Elements
    cout << "\n=== LeetCode 347. Top K Frequent Elements ===" << endl;
    vector<int> freqNums = {1, 1, 1, 2, 2, 3};
    int kFreq = 2;
    vector<int> topKFrequentResult = topKFrequent(freqNums, kFreq);
    cout << "nums: [1, 1, 1, 2, 2, 3], k: 2, top k frequent: [";
    for (size_t i = 0; i < topKFrequentResult.size(); i++) {
        cout << topKFrequentResult[i];
        if (i < topKFrequentResult.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    // 测试 LeetCode 380. Insert Delete GetRandom O(1)
    cout << "\n=== LeetCode 380. Insert Delete GetRandom O(1) ===" << endl;
    RandomizedSet randomizedSet;
    cout << "Insert 1: " << (randomizedSet.insert(1) ? "true" : "false") << endl;
    cout << "Insert 2: " << (randomizedSet.insert(2) ? "true" : "false") << endl;
    cout << "Insert 1 again: " << (randomizedSet.insert(1) ? "true" : "false") << endl;
    cout << "Get random: " << randomizedSet.getRandom() << endl;
    cout << "Remove 2: " << (randomizedSet.remove(2) ? "true" : "false") << endl;
    cout << "Remove 3: " << (randomizedSet.remove(3) ? "true" : "false") << endl;
    cout << "Get random: " << randomizedSet.getRandom() << endl;

    // 测试 LeetCode 146. LRU Cache
    cout << "\n=== LeetCode 146. LRU Cache ===" << endl;
    LRUCache lruCache(2);
    lruCache.put(1, 1);
    lruCache.put(2, 2);
    cout << "Get 1: " << lruCache.get(1) << endl;
    lruCache.put(3, 3);  // 这会使得键 2 被移除
    cout << "Get 2: " << lruCache.get(2) << endl;
```

```
    cout << "Get 3: " << lruCache.get(3) << endl;

    return 0;
}
```

==========

文件: HashFunction.java

==========

```
package class106;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Security;
import java.nio.charset.StandardCharsets;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import java.util.BitSet;
import java.util.Arrays;
import java.util.Collections;

public class HashFunction {

    // 哈希函数实例
    public static class Hash {

        private MessageDigest md;

        // 打印支持哪些哈希算法
        public static void showAlgorithms() {
            for (String algorithm : Security.getAlgorithms("MessageDigest")) {
                System.out.println(algorithm);
            }
        }

        // 用具体算法名字构造实例
```

```
        public Hash(String algorithm) {
            try {
                md = MessageDigest.getInstance(algorithm);
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }
        }

        // 输入字符串返回哈希值
        public String hashValue(String input) {
            byte[] hashInBytes = md.digest(input.getBytes(StandardCharsets.UTF_8));
            BigInteger bigInt = new BigInteger(1, hashInBytes);
            String hashText = bigInt.toString(16);
            return hashText;
        }

    }

    /**
     * LintCode 128. Hash Function
     * 题目来源: https://www.lintcode.com/problem/hash-function/description
     *
     * 题目描述:
     * 在数据结构中，哈希函数是用来将一个字符串（或任何其他类型）转化为小于哈希表大小且大于等于零的整数。
     * 一个好的哈希函数可以尽可能少地产生冲突。
     * 一种广泛使用的哈希函数算法是使用数值 33，假设任何字符串都是基于 33 的一个大整数，比如：
     * hashcode("abcd") = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
     *                  = (97* 33^3 + 98 * 33^2 + 99 * 33 +100) % HASH_SIZE
     *                  = 3595978 % HASH_SIZE
     * 其中 HASH_SIZE 表示哈希表的大小(可以假设一个哈希表就是一个索引 0 ~ HASH_SIZE-1 的数组)。
     * 给出一个字符串作为 key 和一个哈希表的大小，返回这个字符串的哈希值。
     *
     * 样例:
     * 对于 key="abcd" 并且 size=100， 返回 78
     *
     * 算法思路:
     * 使用霍纳法则（Horner's Rule）优化计算，避免大数溢出:
     * hashcode = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
     * 可以转换为:
     * hashcode = ((((ascii(a) % HASH_SIZE) * 33 + ascii(b)) % HASH_SIZE) * 33 + ascii(c)) % HASH_SIZE) * 33 + ascii(d)) % HASH_SIZE
```

```
     *
     * 时间复杂度: O(n)，其中 n 是字符串的长度
     * 空间复杂度: O(1)
     */
    public static int hashCode(char[] key, int HASH_SIZE) {
        long ans = 0;
        for (int i = 0; i < key.length; i++) {
            ans = (ans * 33 + (int) (key[i])) % HASH_SIZE;
        }
        return (int) ans;
    }

    /**
     * LintCode 128. Hash Function (字符串版本)
     *
     * 时间复杂度: O(n)，其中 n 是字符串的长度
     * 空间复杂度: O(1)
     */
    public static int hashCode(String key, int HASH_SIZE) {
        long ans = 0;
        for (int i = 0; i < key.length(); i++) {
            ans = (ans * 33 + (int) (key.charAt(i))) % HASH_SIZE;
        }
        return (int) ans;
    }

    /**
     * LeetCode 705. Design HashSet (设计哈希集合)
     * 题目来源: https://leetcode.com/problems/design-hashset/
     *
     * 题目描述:
     * 不使用任何内建的哈希表库设计一个哈希集合（HashSet）。
     * 实现 MyHashSet 类：
     * void add(key) 向哈希集合中插入值 key 。
     * bool contains(key) 返回哈希集合中是否存在这个值 key 。
     * void remove(key) 将给定值 key 从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。
     *
     * 示例:
     * 输入：
     * ["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove",
"contains"]
     * [[], [1], [2], [1], [3], [2], [2], [2], [2]]
     * 输出：
```

```
 * [null, null, null, true, false, null, true, null, false]
 *
 * 约束条件:
 * 0 <= key <= 10^6
 * 最多调用 10^4 次 add、remove 和 contains
 *
 * 算法思路:
 * 使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。
 * 当发生哈希冲突时，将元素添加到对应位置的链表中。
 *
 * 时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）
 * 空间复杂度: O(n)，存储所有元素
 */
static class MyHashSet {
    private static final int BASE = 10000;
    private LinkedList<Integer>[] data;

    /** Initialize your data structure here. */
    public MyHashSet() {
        data = new LinkedList[BASE];
        for (int i = 0; i < BASE; ++i) {
            data[i] = new LinkedList<Integer>();
        }
    }

    public void add(int key) {
        int h = hash(key);
        Iterator<Integer> iterator = data[h].iterator();
        while (iterator.hasNext()) {
            Integer element = iterator.next();
            if (element == key) {
                return;
            }
        }
        data[h].offerLast(key);
    }

    public void remove(int key) {
        int h = hash(key);
        Iterator<Integer> iterator = data[h].iterator();
        while (iterator.hasNext()) {
            Integer element = iterator.next();
            if (element == key) {
```

```
                iterator.remove();
                return;
            }
        }
    }

    /** Returns true if this set contains the specified element */
    public boolean contains(int key) {
        int h = hash(key);
        Iterator<Integer> iterator = data[h].iterator();
        while (iterator.hasNext()) {
            Integer element = iterator.next();
            if (element == key) {
                return true;
            }
        }
        return false;
    }

    private static int hash(int key) {
        return key % BASE;
    }
}

/**
 * LeetCode 706. Design HashMap (设计哈希映射)
 * 题目来源: https://leetcode.com/problems/design-hashmap/
 *
 * 题目描述:
 * 不使用任何内建的哈希表库设计一个哈希映射（HashMap）。
 * 实现 MyHashMap 类：
 * MyHashMap() 用空映射初始化对象
 * void put(int key, int value) 向 HashMap 插入一个键值对 (key, value) 。如果 key 已经存在于映射中，则更新其对应的值 value 。
 * int get(int key) 返回特定的 key 所映射的 value ；如果映射中不包含 key 的映射，返回 -1 。
 * void remove(key) 如果映射中存在 key 的映射，则移除 key 和它所对应的 value 。
 *
 * 示例:
 * 输入：
 * ["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
 * [[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]
 * 输出：
 * [null, null, null, 1, -1, null, 1, null, -1]
```

```
 *
 * 约束条件:
 * 0 <= key, value <= 10^6
 * 最多调用 10^4 次 put、get 和 remove 方法
 *
 * 算法思路:
 * 使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。
 * 每个链表节点存储键值对，当发生哈希冲突时，将节点添加到对应位置的链表中。
 *
 * 时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）
 * 空间复杂度: O(n)，存储所有元素
 */
static class MyHashMap {
    private static final int BASE = 10000;
    private LinkedList<Pair> data[];

    private static class Pair {
        private int key;
        private int value;

        public Pair(int key, int value) {
            this.key = key;
            this.value = value;
        }

        public int getKey() {
            return key;
        }

        public int getValue() {
            return value;
        }

        public void setValue(int value) {
            this.value = value;
        }
    }

    /** Initialize your data structure here. */
    public MyHashMap() {
        data = new LinkedList[BASE];
        for (int i = 0; i < BASE; ++i) {
            data[i] = new LinkedList<Pair>();
```

```
            }
        }

        /** value will always be non-negative. */
        public void put(int key, int value) {
            int h = hash(key);
            Iterator<Pair> iterator = data[h].iterator();
            while (iterator.hasNext()) {
                Pair pair = iterator.next();
                if (pair.getKey() == key) {
                    pair.setValue(value);
                    return;
                }
            }
            data[h].offerLast(new Pair(key, value));
        }

        /** Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key */
        public int get(int key) {
            int h = hash(key);
            Iterator<Pair> iterator = data[h].iterator();
            while (iterator.hasNext()) {
                Pair pair = iterator.next();
                if (pair.getKey() == key) {
                    return pair.getValue();
                }
            }
            return -1;
        }

        /** Removes the mapping of the specified value key if this map contains a mapping for the key */
        public void remove(int key) {
            int h = hash(key);
            Iterator<Pair> iterator = data[h].iterator();
            while (iterator.hasNext()) {
                Pair pair = iterator.next();
                if (pair.getKey() == key) {
                    iterator.remove();
                    return;
                }
            }
```

```
    }

    private static int hash(int key) {
        return key % BASE;
    }
}

/**
 * LeetCode 28. Find the Index of the First Occurrence in a String (实现 strStr())
 * 题目来源: https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/
 *
 * 题目描述:
 * 给你两个字符串 haystack 和 needle，请你在 haystack 字符串中找出 needle 字符串的第一个匹配项的下标（下标从 0 开始）。
 * 如果 needle 不是 haystack 的一部分，则返回 -1。
 *
 * 示例:
 * 输入：haystack = "sadbutsad", needle = "sad"
 * 输出：0
 *
 * 输入：haystack = "leetcode", needle = "leeto"
 * 输出：-1
 *
 * 算法思路:
 * 使用 Rabin-Karp 算法（滚动哈希）实现字符串匹配：
 * 1. 计算 needle 的哈希值
 * 2. 在 haystack 中维护一个长度为 needle.length() 的滑动窗口，计算其哈希值
 * 3. 当哈希值相等时，再进行字符串比较确认（避免哈希冲突）
 *
 * 时间复杂度: O(n+m)，其中 n 是 haystack 长度，m 是 needle 长度
 * 空间复杂度: O(1)
 */
public static int strStr(String haystack, String needle) {
    if (needle.isEmpty()) return 0;
    if (haystack.length() < needle.length()) return -1;

    int base = 256; // 基数
    int mod = 1000000007; // 大质数，用于取模运算

    int needleHash = 0;
    int haystackHash = 0;
    int h = 1; // 用于计算最高位的权重
```

```
        // 计算 needle 的哈希值和 h 的值
        for (int i = 0; i < needle.length(); i++) {
            needleHash = (needleHash * base + needle.charAt(i)) % mod;
            if (i < needle.length() - 1) {
                h = (h * base) % mod;
            }
        }

        // 计算 haystack 第一个窗口的哈希值
        for (int i = 0; i < needle.length(); i++) {
            haystackHash = (haystackHash * base + haystack.charAt(i)) % mod;
        }

        // 滑动窗口匹配
        for (int i = 0; i <= haystack.length() - needle.length(); i++) {
            // 如果哈希值相等，再进行字符串比较确认
            if (needleHash == haystackHash) {
                if (haystack.substring(i, i + needle.length()).equals(needle)) {
                    return i;
                }
            }

            // 计算下一个窗口的哈希值
            if (i < haystack.length() - needle.length()) {
                haystackHash = (base * (haystackHash - (haystack.charAt(i) * h) % mod) +
haystack.charAt(i + needle.length())) % mod;
                if (haystackHash < 0) {
                    haystackHash += mod;
                }
            }
        }

        return -1;
    }

    /**
     * LeetCode 187. Repeated DNA Sequences (重复的DNA序列)
     * 题目来源: https://leetcode.com/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)。
    * 可以按任意顺序返回答案。
    *
    * 示例:
    * 输入：s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
    * 输出：["AAAAACCCCC","CCCCCAAAAA"]
    *
    * 输入：s = "AAAAAAAAAAAAA"
    * 输出：["AAAAAAAAAA"]
    *
    * 算法思路:
    * 使用滚动哈希技术：
    * 1. 将每个字符映射为数字：A=0, C=1, G=2, T=3
    * 2. 使用 4 进制表示长度为 10 的序列
    * 3. 滑动窗口遍历所有长度为 10 的子串，计算其哈希值
    * 4. 使用哈希表记录每个哈希值出现的次数
    * 5. 返回出现次数大于 1 的序列
    *
    * 时间复杂度: O(n)，其中 n 是 DNA 序列长度
    * 空间复杂度: O(n)，存储所有子串的哈希值
    */
   public static List<String> findRepeatedDnaSequences(String s) {
       List<String> result = new ArrayList<>();
       if (s.length() < 10) return result;

       // 字符到数字的映射
       int[] map = new int[256];
       map['A'] = 0;
       map['C'] = 1;
       map['G'] = 2;
       map['T'] = 3;

       int base = 4;
       int mod = 1000000007; // 大质数，用于取模运算
       int windowSize = 10;

       // 计算 base^(windowSize-1) % mod
       long h = 1;
       for (int i = 0; i < windowSize - 1; i++) {
           h = (h * base) % mod;
       }
```

```
        // 计算第一个窗口的哈希值
        long hash = 0;
        for (int i = 0; i < windowSize; i++) {
            hash = (hash * base + map[s.charAt(i)]) % mod;
        }

        // 使用哈希表记录每个哈希值出现的次数
        Map<Long, Integer> hashMap = new HashMap<>();
        hashMap.put(hash, 1);

        // 滑动窗口计算后续哈希值
        for (int i = 1; i <= s.length() - windowSize; i++) {
            // 移除最高位字符，添加最低位字符
            hash = (base * (hash - (map[s.charAt(i - 1)] * h) % mod) + map[s.charAt(i + windowSize
- 1)]) % mod;
            if (hash < 0) {
                hash += mod;
            }

            // 记录哈希值出现次数
            hashMap.put(hash, hashMap.getOrDefault(hash, 0) + 1);

            // 如果某个哈希值出现 2 次，将其对应的子串加入结果集
            if (hashMap.get(hash) == 2) {
                result.add(s.substring(i, i + windowSize));
            }
        }

        return result;
    }

    public static List<String> generateStrings(char[] arr, int n) {
        char[] path = new char[n];
        List<String> ans = new ArrayList<>();
        f(arr, 0, n, path, ans);
        return ans;
    }

    public static void f(char[] arr, int i, int n, char[] path, List<String> ans) {
        if (i == n) {
            ans.add(String.valueOf(path));
        } else {
            for (char cha : arr) {
```

```
            path[i] = cha;
            f(arr, i + 1, n, path, ans);
        }
    }
}

/**
 * LeetCode 214. Shortest Palindrome (最短回文串)
 * 题目来源: https://leetcode.com/problems/shortest-palindrome/
 *
 * 题目描述:
 * 给你一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。
 * 找到并返回可以用这种方式转换的最短回文串。
 *
 * 示例:
 * 输入：s = "aacecaaa"
 * 输出："aaacecaaa"
 *
 * 输入：s = "abcd"
 * 输出："dcbabcd"
 *
 * 算法思路:
 * 使用滚动哈希技术找到 s 的最长前缀回文串:
 * 1. 计算 s 的正向哈希和反向哈希
 * 2. 使用双指针从两端向中间移动，同时比较正向和反向哈希
 * 3. 当找到最长前缀回文串后，将剩余部分反转并添加到原字符串前面
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static String shortestPalindrome(String s) {
    if (s.length() <= 1) return s;

    // 使用滚动哈希找到最长前缀回文串
    int n = s.length();
    long base = 256;
    long mod = 1000000007;
    long forwardHash = 0;
    long backwardHash = 0;
    long power = 1;
    int maxLen = 0;

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

```
            forwardHash = (forwardHash * base + s.charAt(i)) % mod;
            backwardHash = (backwardHash + s.charAt(i) * power) % mod;
            if (forwardHash == backwardHash) {
                maxLen = i + 1;
            }
            power = (power * base) % mod;
        }

        // 将剩余部分反转并添加到前面
        String suffix = s.substring(maxLen);
        StringBuilder reversedSuffix = new StringBuilder(suffix).reverse();
        return reversedSuffix.toString() + s;
    }

    /**
     * 一致性哈希 (Consistent Hashing) 实现
     *
     * 应用场景: 分布式系统中的负载均衡，如分布式缓存、数据库分片等
     * 算法原理: 将服务器和键都映射到一个虚拟环上，每个键被分配给顺时针方向遇到的第一个服务器
     * 优势: 当服务器增减时，只需要重新分配少量键，减少数据迁移
     */
    static class ConsistentHash {
        private final TreeMap<Integer, String> virtualNodes; // 虚拟节点环
        private final int replicas; // 每个真实节点对应的虚拟节点数
        private final List<String> servers; // 真实服务器列表

        public ConsistentHash(int replicas) {
            this.replicas = replicas;
            this.virtualNodes = new TreeMap<>();
            this.servers = new ArrayList<>();
        }

        // 添加服务器
        public void addServer(String server) {
            servers.add(server);
            // 为每个真实节点创建多个虚拟节点
            for (int i = 0; i < replicas; i++) {
                String virtualNode = server + "#" + i;
                int hash = getHash(virtualNode);
                virtualNodes.put(hash, server);
            }
        }
```

```
    // 移除服务器
    public void removeServer(String server) {
        servers.remove(server);
        // 移除对应的所有虚拟节点
        for (int i = 0; i < replicas; i++) {
            String virtualNode = server + "#" + i;
            int hash = getHash(virtualNode);
            virtualNodes.remove(hash);
        }
    }

    // 获取键对应的服务器
    public String getServer(String key) {
        if (virtualNodes.isEmpty()) {
            return null;
        }

        int hash = getHash(key);
        // 找到顺时针方向的第一个服务器
        Map.Entry<Integer, String> entry = virtualNodes.ceilingEntry(hash);
        // 如果没有比当前 hash 大的节点，则返回环的第一个节点
        if (entry == null) {
            entry = virtualNodes.firstEntry();
        }
        return entry.getValue();
    }

    // 哈希函数
    private int getHash(String key) {
        // 使用 FNV-1a 哈希算法
        final int FNV_32_INIT = 0x811c9dc5;
        final int FNV_32_PRIME = 0x01000193;

        int hash = FNV_32_INIT;
        for (int i = 0; i < key.length(); i++) {
            hash ^= key.charAt(i);
            hash *= FNV_32_PRIME;
        }
        return Math.abs(hash);
    }

    // 获取服务器列表
    public List<String> getServers() {
```

```
        return new ArrayList<>(servers);
    }
}

/**
 * 布隆过滤器 (Bloom Filter) 实现
 *
 * 应用场景: 快速判断一个元素是否可能存在于集合中，如垃圾邮件过滤、缓存穿透防护等
 * 算法原理: 使用多个哈希函数将元素映射到位数组的不同位置，查询时检查所有位置是否都为1
 * 特点: 存在一定的误判率，但不会漏判；删除元素困难
 */
static class BloomFilter {
    private final BitSet bits; // 位数组
    private final int size; // 位数组大小
    private final int[] seeds; // 多个哈希函数的种子

    public BloomFilter(int size, int hashFunctions) {
        this.size = size;
        this.bits = new BitSet(size);
        this.seeds = new int[hashFunctions];
        // 初始化哈希函数种子
        for (int i = 0; i < hashFunctions; i++) {
            seeds[i] = i * 100 + 31; // 使用不同的种子
        }
    }

    // 添加元素
    public void add(String element) {
        for (int seed : seeds) {
            int hash = getHash(element, seed);
            bits.set(hash);
        }
    }

    // 判断元素是否可能存在
    public boolean mightContain(String element) {
        for (int seed : seeds) {
            int hash = getHash(element, seed);
            if (!bits.get(hash)) {
                return false; // 只要有一个位置为0，元素一定不存在
            }
        }
        return true; // 所有位置都为1，元素可能存在
```

```
    }

    // 哈希函数
    private int getHash(String element, int seed) {
        int hash = 0;
        for (int i = 0; i < element.length(); i++) {
            hash = seed * hash + element.charAt(i);
        }
        return Math.abs(hash % size);
    }
}

/**
 * 双重哈希 (Double Hashing) 实现的哈希表
 *
 * 应用场景: 开放寻址法解决哈希冲突
 * 算法原理: 使用两个哈希函数，当发生冲突时，第二个哈希函数确定探测步长
 * 优势: 减少聚集现象，提高哈希表性能
 */
static class DoubleHashTable<K, V> {
    private static final int DEFAULT_SIZE = 16;
    private static final double LOAD_FACTOR = 0.75;

    private Object[] keys;
    private Object[] values;
    private boolean[] occupied;
    private int size;

    @SuppressWarnings("unchecked")
    public DoubleHashTable() {
        keys = new Object[DEFAULT_SIZE];
        values = new Object[DEFAULT_SIZE];
        occupied = new boolean[DEFAULT_SIZE];
        size = 0;
    }

    // 插入键值对
    @SuppressWarnings("unchecked")
    public void put(K key, V value) {
        if (key == null) throw new IllegalArgumentException("Key cannot be null");

        // 检查是否需要扩容
        if ((double) size / keys.length >= LOAD_FACTOR) {
```

```
            rehash();
        }

        int index = findInsertionIndex(key);
        keys[index] = key;
        values[index] = value;
        if (!occupied[index]) {
            occupied[index] = true;
            size++;
        }
    }

    // 获取值
    @SuppressWarnings("unchecked")
    public V get(K key) {
        if (key == null) return null;

        int index = findIndex(key);
        return index != -1 ? (V) values[index] : null;
    }

    // 删除键值对
    @SuppressWarnings("unchecked")
    public void remove(K key) {
        if (key == null) return;

        int index = findIndex(key);
        if (index != -1) {
            keys[index] = null;
            values[index] = null;
            occupied[index] = false;
            size--;
        }
    }

    // 查找插入位置
    @SuppressWarnings("unchecked")
    private int findInsertionIndex(K key) {
        int hash1 = hash1(key);
        int hash2 = hash2(key);
        int index = hash1;
        int step = 1;
```

```
        // 查找空位置或相同的键
        while (occupied[index]) {
            if (key.equals(keys[index])) {
                return index; // 键已存在，返回该位置以更新值
            }
            index = (hash1 + step * hash2) % keys.length;
            step++;
        }

        return index;
    }

    // 查找键的索引
    @SuppressWarnings("unchecked")
    private int findIndex(K key) {
        int hash1 = hash1(key);
        int hash2 = hash2(key);
        int index = hash1;
        int step = 1;

        // 查找键
        while (occupied[index]) {
            if (key.equals(keys[index])) {
                return index; // 找到键
            }
            index = (hash1 + step * hash2) % keys.length;
            step++;
            // 避免无限循环
            if (step > keys.length) {
                break;
            }
        }

        return -1; // 未找到键
    }

    // 扩容
    @SuppressWarnings("unchecked")
    private void rehash() {
        Object[] oldKeys = keys;
        Object[] oldValues = values;
        boolean[] oldOccupied = occupied;
```

```
            keys = new Object[keys.length * 2];
            values = new Object[keys.length * 2];
            occupied = new boolean[keys.length * 2];
            size = 0;

            // 重新插入所有键值对
            for (int i = 0; i < oldKeys.length; i++) {
                if (oldOccupied[i]) {
                    put((K) oldKeys[i], (V) oldValues[i]);
                }
            }
        }

        // 第一个哈希函数
        private int hash1(K key) {
            return Math.abs(key.hashCode() % keys.length);
        }

        // 第二个哈希函数，用于计算步长
        private int hash2(K key) {
            return 1 + Math.abs(key.hashCode() % (keys.length - 1));
        }

        // 获取大小
        public int size() {
            return size;
        }
    }

    /**
     * LeetCode 1. Two Sum (两数之和)
     * 题目来源: https://leetcode.com/problems/two-sum/
     *
     * 题目描述:
     * 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。
     * 你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。
     * 你可以按任意顺序返回答案。
     *
     * 示例:
     * 输入：nums = [2,7,11,15], target = 9
     * 输出：[0,1]
     * 解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
```

```
     *
     * 算法思路:
     * 使用哈希表存储每个数字及其对应的索引，遍历数组时检查 target - nums[i]是否在哈希表中
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    public static int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[]{-1, -1};
    }

    /**
     * LeetCode 49. Group Anagrams (字母异位词分组)
     * 题目来源: https://leetcode.com/problems/group-anagrams/
     *
     * 题目描述:
     * 给你一个字符串数组，请你将字母异位词组合在一起。可以按任意顺序返回结果列表。
     * 字母异位词是由重新排列源单词的所有字母得到的一个新单词。
     *
     * 示例:
     * 输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
     * 输出: [["bat"],["nat","tan"],["ate","eat","tea"]]
     *
     * 算法思路:
     * 使用排序后的字符串作为哈希表的键，将具有相同排序字符串的单词分组
     *
     * 时间复杂度: O(n * k log k)，其中 n 是字符串数量，k 是字符串最大长度
     * 空间复杂度: O(n * k)
     */
    public static List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();
        for (String str : strs) {
            char[] chars = str.toCharArray();
            Arrays.sort(chars);
            String key = new String(chars);
```

```
            map.putIfAbsent(key, new ArrayList<>());
            map.get(key).add(str);
        }
        return new ArrayList<>(map.values());
    }

    /**
     * LeetCode 242. Valid Anagram (有效的字母异位词)
     * 题目来源: https://leetcode.com/problems/valid-anagram/
     *
     * 题目描述:
     * 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
     * 注意：若 s 和 t 中每个字符出现的次数都相同，则称 s 和 t 互为字母异位词。
     *
     * 示例:
     * 输入: s = "anagram", t = "nagaram"
     * 输出: true
     *
     * 算法思路:
     * 使用哈希表统计每个字符出现的次数，然后比较两个字符串的字符频率
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)，因为字符集大小固定为 26
     */
    public static boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;

        int[] count = new int[26];
        for (char c : s.toCharArray()) {
            count[c - 'a']++;
        }
        for (char c : t.toCharArray()) {
            count[c - 'a']--;
            if (count[c - 'a'] < 0) return false;
        }
        return true;
    }

    /**
     * LeetCode 3. Longest Substring Without Repeating Characters (无重复字符的最长子串)
     * 题目来源: https://leetcode.com/problems/longest-substring-without-repeating-characters/
     *
     * 题目描述:
```

```
     * 给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。
     *
     * 示例:
     * 输入: s = "abcabcbb"
     * 输出: 3
     * 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
     *
     * 算法思路:
     * 使用滑动窗口和哈希表记录字符最后出现的位置
     * 当遇到重复字符时，移动窗口左边界到重复字符的下一个位置
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(min(m, n))，其中 m 是字符集大小
     */
    public static int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLength = 0;
        int left = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            if (map.containsKey(c) && map.get(c) >= left) {
                left = map.get(c) + 1;
            }
            map.put(c, right);
            maxLength = Math.max(maxLength, right - left + 1);
        }
        return maxLength;
    }

    /**
     * LeetCode 76. Minimum Window Substring (最小覆盖子串)
     * 题目来源: https://leetcode.com/problems/minimum-window-substring/
     *
     * 题目描述:
     * 给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。
     *
     * 示例:
     * 输入：s = "ADOBECODEBANC", t = "ABC"
     * 输出："BANC"
     * 解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。
     *
```

```
     * 算法思路:
     * 使用滑动窗口和哈希表统计字符频率
     * 维护一个计数器记录还需要匹配的字符数量
     *
     * 时间复杂度: O(m + n)
     * 空间复杂度: O(m + n)
     */
    public static String minWindow(String s, String t) {
        if (s.length() < t.length()) return "";

        Map<Character, Integer> target = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();

        // 统计 t 中字符频率
        for (char c : t.toCharArray()) {
            target.put(c, target.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int required = target.size();
        int formed = 0;
        int minLength = Integer.MAX_VALUE;
        int minLeft = 0, minRight = 0;

        while (right < s.length()) {
            char c = s.charAt(right);
            window.put(c, window.getOrDefault(c, 0) + 1);

            if (target.containsKey(c) && window.get(c).intValue() == target.get(c).intValue()) {
                formed++;
            }

            while (left <= right && formed == required) {
                c = s.charAt(left);

                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    minLeft = left;
                    minRight = right;
                }

                window.put(c, window.get(c) - 1);
                if (target.containsKey(c) && window.get(c) < target.get(c)) {
```

```
                formed--;
            }
            left++;
        }
        right++;
    }

    return minLength == Integer.MAX_VALUE ? "" : s.substring(minLeft, minRight + 1);
}

/**
 * LeetCode 560. Subarray Sum Equals K (和为K的子数组)
 * 题目来源: https://leetcode.com/problems/subarray-sum-equals-k/
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k ，请你统计并返回该数组中和为 k 的连续子数组的个数。
 *
 * 示例:
 * 输入：nums = [1,1,1], k = 2
 * 输出：2
 *
 * 算法思路:
 * 使用前缀和和哈希表，记录每个前缀和出现的次数
 * 当prefixSum - k在哈希表中存在时，说明存在和为k的子数组
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static int subarraySum(int[] nums, int k) {
    Map<Integer, Integer> prefixSumCount = new HashMap<>();
    prefixSumCount.put(0, 1); // 前缀和为0出现1次
    int prefixSum = 0;
    int count = 0;

    for (int num : nums) {
        prefixSum += num;
        if (prefixSumCount.containsKey(prefixSum - k)) {
            count += prefixSumCount.get(prefixSum - k);
        }
        prefixSumCount.put(prefixSum, prefixSumCount.getOrDefault(prefixSum, 0) + 1);
    }
    return count;
}
```

```
    /**
     * LeetCode 347. Top K Frequent Elements (前 K 个高频元素)
     * 题目来源: https://leetcode.com/problems/top-k-frequent-elements/
     *
     * 题目描述:
     * 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按任意顺序返回答案。
     *
     * 示例:
     * 输入: nums = [1,1,1,2,2,3], k = 2
     * 输出: [1,2]
     *
     * 算法思路:
     * 使用哈希表统计频率，然后使用桶排序或优先队列找出前 k 个高频元素
     *
     * 时间复杂度: O(n log k)
     * 空间复杂度: O(n)
     */
    public static int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // 使用桶排序
        List<Integer>[] buckets = new List[nums.length + 1];
        for (int num : frequencyMap.keySet()) {
            int frequency = frequencyMap.get(num);
            if (buckets[frequency] == null) {
                buckets[frequency] = new ArrayList<>();
            }
            buckets[frequency].add(num);
        }

        List<Integer> result = new ArrayList<>();
        for (int i = buckets.length - 1; i >= 0 && result.size() < k; i--) {
            if (buckets[i] != null) {
                result.addAll(buckets[i]);
            }
        }

        return result.stream().mapToInt(Integer::intValue).toArray();
```

```
    }

    /**
     * LeetCode 380. Insert Delete GetRandom O(1) (常数时间插入、删除和获取随机元素)
     * 题目来源: https://leetcode.com/problems/insert-delete-getrandom-o1/
     *
     * 题目描述:
     * 实现 RandomizedSet 类：
     * RandomizedSet() 初始化 RandomizedSet 对象
     * bool insert(int val) 当元素 val 不存在时，向集合中插入该项，并返回 true ；否则，返回 false 。
     * bool remove(int val) 当元素 val 存在时，从集合中移除该项，并返回 true ；否则，返回 false 。
     * int getRandom() 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有相同的概率被返回。
     *
     * 算法思路:
     * 使用哈希表存储值和索引的映射，使用动态数组存储值
     * 删除时将要删除的元素与最后一个元素交换，然后删除最后一个元素
     *
     * 时间复杂度: O(1) 平均时间复杂度
     * 空间复杂度: O(n)
     */
    static class RandomizedSet {
        private Map<Integer, Integer> valueToIndex;
        private List<Integer> values;
        private java.util.Random random;

        public RandomizedSet() {
            valueToIndex = new HashMap<>();
            values = new ArrayList<>();
            random = new java.util.Random();
        }

        public boolean insert(int val) {
            if (valueToIndex.containsKey(val)) {
                return false;
            }
            valueToIndex.put(val, values.size());
            values.add(val);
            return true;
        }

        public boolean remove(int val) {
```

```
        if (!valueToIndex.containsKey(val)) {
            return false;
        }
        int index = valueToIndex.get(val);
        int lastElement = values.get(values.size() - 1);

        // 将要删除的元素与最后一个元素交换
        values.set(index, lastElement);
        valueToIndex.put(lastElement, index);

        // 删除最后一个元素
        values.remove(values.size() - 1);
        valueToIndex.remove(val);

        return true;
    }

    public int getRandom() {
        return values.get(random.nextInt(values.size()));
    }
}

/**
 * LeetCode 146. LRU Cache (LRU缓存)
 * 题目来源: https://leetcode.com/problems/lru-cache/
 *
 * 题目描述:
 * 请你设计并实现一个满足 LRU (最近最少使用) 缓存约束的数据结构。
 * 实现 LRUCache 类：
 * LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
 * int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
 * void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。
 * 如果插入操作导致关键字数量超过 capacity ，则应该逐出最久未使用的关键字。
 *
 * 算法思路:
 * 使用哈希表+双向链表实现
 * 哈希表提供O(1)的查找，双向链表维护访问顺序
 *
 * 时间复杂度: O(1)
 * 空间复杂度: O(capacity)
 */
static class LRUCache {
```

```
class DLinkedNode {
    int key;
    int value;
    DLinkedNode prev;
    DLinkedNode next;

    DLinkedNode() {}
    DLinkedNode(int key, int value) {
        this.key = key;
        this.value = value;
    }
}

private Map<Integer, DLinkedNode> cache;
private int size;
private int capacity;
private DLinkedNode head, tail;

public LRUCache(int capacity) {
    this.capacity = capacity;
    this.size = 0;
    cache = new HashMap<>();
    head = new DLinkedNode();
    tail = new DLinkedNode();
    head.next = tail;
    tail.prev = head;
}

public int get(int key) {
    DLinkedNode node = cache.get(key);
    if (node == null) {
        return -1;
    }
    // 移动到头部
    moveToHead(node);
    return node.value;
}

public void put(int key, int value) {
    DLinkedNode node = cache.get(key);
    if (node == null) {
        DLinkedNode newNode = new DLinkedNode(key, value);
        cache.put(key, newNode);
```

```
            addToHead(newNode);
            size++;
            if (size > capacity) {
                DLinkedNode tail = removeTail();
                cache.remove(tail.key);
                size--;
            }
        } else {
            node.value = value;
            moveToHead(node);
        }
    }

    private void addToHead(DLinkedNode node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(DLinkedNode node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(DLinkedNode node) {
        removeNode(node);
        addToHead(node);
    }

    private DLinkedNode removeTail() {
        DLinkedNode res = tail.prev;
        removeNode(res);
        return res;
    }
}

/**
 * Codeforces 271D - Good Substrings (好子串)
 * 题目来源: https://codeforces.com/problemset/problem/271/D
 *
 * 题目描述:
 * 给定一个字符串 s 和一个长度为 26 的字符串 bad，bad[i]='1'表示第 i 个字母是好的，'0'表示坏的。
```

```
 * 一个子串被认为是好的，如果它包含的坏字母数量不超过 k。
 * 计算 s 中不同好子串的数量。
 *
 * 算法思路:
 * 使用滚动哈希技术计算所有子串的哈希值，同时统计坏字母数量
 * 使用哈希集合存储满足条件的子串哈希值
 *
 * 时间复杂度: O(n^2)，其中 n 是字符串长度
 * 空间复杂度: O(n^2)
 */
public static int countGoodSubstrings(String s, String bad, int k) {
    int n = s.length();
    Set<Long> set = new HashSet<>();
    long base = 131;
    long mod = (long)1e9 + 7;

    for (int i = 0; i < n; i++) {
        long hash = 0;
        int badCount = 0;

        for (int j = i; j < n; j++) {
            char c = s.charAt(j);
            if (bad.charAt(c - 'a') == '0') {
                badCount++;
            }

            if (badCount > k) {
                break;
            }

            hash = (hash * base + (c - 'a' + 1)) % mod;
            set.add(hash);
        }
    }

    return set.size();
}

/**
 * Codeforces 514C - Watto and Mechanism (瓦托和机制)
 * 题目来源: https://codeforces.com/problemset/problem/514/C
 *
 * 题目描述:
```

```
     * 给定 n 个字符串的字典和 m 个查询字符串。
     * 对于每个查询字符串，判断是否存在字典中的一个字符串，使得它们长度相同且最多有一个字符不同。
     *
     * 算法思路:
     * 使用滚动哈希预处理字典中所有字符串的哈希值
     * 对于每个查询字符串，尝试修改每个位置的字符，检查修改后的哈希值是否在字典中
     *
     * 时间复杂度: O(nL + mL)，其中 L 是字符串平均长度
     * 空间复杂度: O(n)
     */
    public static boolean[] wattoAndMechanism(String[] dictionary, String[] queries) {
        Set<Long> dictHashes = new HashSet<>();
        long base = 131;
        long mod = (long)1e9 + 7;

        // 预处理字典字符串的哈希值
        for (String word : dictionary) {
            long hash = 0;
            long power = 1;

            for (int i = 0; i < word.length(); i++) {
                hash = (hash * base + (word.charAt(i) - 'a' + 1)) % mod;
                if (i > 0) {
                    power = (power * base) % mod;
                }
            }
            dictHashes.add(hash);
        }

        boolean[] results = new boolean[queries.length];

        for (int idx = 0; idx < queries.length; idx++) {
            String query = queries[idx];
            int n = query.length();
            long[] prefixHash = new long[n + 1];
            long[] suffixHash = new long[n + 1];
            long[] powers = new long[n + 1];

            powers[0] = 1;
            for (int i = 1; i <= n; i++) {
                powers[i] = (powers[i - 1] * base) % mod;
            }
```

```
            // 计算前缀哈希
            for (int i = 0; i < n; i++) {
                prefixHash[i + 1] = (prefixHash[i] * base + (query.charAt(i) - 'a' + 1)) % mod;
            }

            // 计算后缀哈希
            for (int i = n - 1; i >= 0; i--) {
                suffixHash[i] = (suffixHash[i + 1] + (query.charAt(i) - 'a' + 1) * powers[n - i - 1]) % mod;
            }

            boolean found = false;

            // 尝试修改每个位置的字符
            for (int i = 0; i < n && !found; i++) {
                for (char c = 'a'; c <= 'c'; c++) {
                    if (c == query.charAt(i)) continue;

                    // 计算修改后的哈希值
                    long newHash = (prefixHash[i] * powers[n - i] +
                                    (c - 'a' + 1) * powers[n - i - 1] +
                                    suffixHash[i + 1]) % mod;

                    if (dictHashes.contains(newHash)) {
                        found = true;
                        break;
                    }
                }
            }

            results[idx] = found;
        }

        return results;
    }

    /**
     * Codeforces 835D - Palindromic characteristics (回文特性)
     * 题目来源: https://codeforces.com/problemset/problem/835/D
     *
     * 题目描述:
     * 定义k级回文串：1级回文串是普通回文串，k级回文串是回文串且前半部分(去掉中间字符)是(k-1)级回文串。
```

```
     * 给定字符串 s，对于 k=1 到 n，统计 s 中有多少个子串是 k 级回文串。
     *
     * 算法思路:
     * 使用滚动哈希判断回文串，同时使用动态规划计算回文级别
     *
     * 时间复杂度: O(n^2)
     * 空间复杂度: O(n^2)
     */
    public static int[] palindromicCharacteristics(String s) {
        int n = s.length();
        long base = 131;
        long mod = (long)1e9 + 7;

        long[] prefixHash = new long[n + 1];
        long[] suffixHash = new long[n + 1];
        long[] powers = new long[n + 1];

        powers[0] = 1;
        for (int i = 1; i <= n; i++) {
            powers[i] = (powers[i - 1] * base) % mod;
        }

        // 计算前缀哈希
        for (int i = 0; i < n; i++) {
            prefixHash[i + 1] = (prefixHash[i] * base + (s.charAt(i) - 'a' + 1)) % mod;
        }

        // 计算后缀哈希
        for (int i = n - 1; i >= 0; i--) {
            suffixHash[i] = (suffixHash[i + 1] + (s.charAt(i) - 'a' + 1) * powers[n - i - 1]) %
mod;
        }

        int[][] dp = new int[n][n];
        int[] result = new int[n + 1];

        for (int len = 1; len <= n; len++) {
            for (int i = 0; i + len <= n; i++) {
                int j = i + len - 1;

                // 计算子串 s[i..j]的哈希值
                long forwardHash = (prefixHash[j + 1] - prefixHash[i] * powers[len] % mod + mod) %
mod;
```

```
                long backwardHash = (suffixHash[i] - suffixHash[j + 1] * powers[len] % mod +
mod) % mod;

                if (forwardHash == backwardHash) {
                    dp[i][j] = 1;

                    // 检查前半部分
                    int halfLen = len / 2;
                    if (halfLen > 0) {
                        int mid = i + halfLen - 1;
                        if (dp[i][mid] > 0) {
                            dp[i][j] = dp[i][mid] + 1;
                        }
                    }

                    for (int k = 1; k <= dp[i][j]; k++) {
                        result[k]++;
                    }
                }
            }
        }

        return Arrays.copyOfRange(result, 1, n + 1);
    }

    /**
     * 完美哈希 (Perfect Hashing) 实现 - 二级哈希表
     *
     * 应用场景: 静态数据集，需要O(1)查找时间且无冲突
     * 算法原理: 使用两级哈希表，第一级哈希将元素分组，第二级为每个组创建无冲突的哈希表
     * 优势: 保证O(1)查找时间，无哈希冲突
     * 限制: 仅适用于静态数据集，构建过程较复杂
     */
    static class PerfectHashTable<K, V> {
        private static class SecondaryTable<K, V> {
            private Object[] keys;
            private Object[] values;
            private int size;

            public SecondaryTable(int size) {
                this.size = size;
                keys = new Object[size];
                values = new Object[size];
```

```
    }

    @SuppressWarnings("unchecked")
    public void put(K key, V value, int hash) {
        int index = Math.abs(hash % size);
        keys[index] = key;
        values[index] = value;
    }

    @SuppressWarnings("unchecked")
    public V get(K key, int hash) {
        int index = Math.abs(hash % size);
        if (keys[index] != null && keys[index].equals(key)) {
            return (V) values[index];
        }
        return null;
    }
}

private SecondaryTable<K, V>[] primaryTable;
private int primarySize;
private int[] hashSeeds;

@SuppressWarnings("unchecked")
public PerfectHashTable(List<Pair<K, V>> data) {
    // 使用平方和法确定主表大小
    primarySize = (int) Math.ceil(Math.sqrt(data.size()));
    primaryTable = new SecondaryTable[primarySize];
    hashSeeds = new int[primarySize];

    // 初始化主表
    List<List<Pair<K, V>>> buckets = new ArrayList<>();
    for (int i = 0; i < primarySize; i++) {
        buckets.add(new ArrayList<>());
    }

    // 第一级哈希分组
    for (Pair<K, V> pair : data) {
        int primaryHash = Math.abs(pair.getKey().hashCode() % primarySize);
        buckets.get(primaryHash).add(pair);
    }

    // 为每个桶创建二级哈希表
```

```
        for (int i = 0; i < primarySize; i++) {
            List<Pair<K, V>> bucket = buckets.get(i);
            if (!bucket.isEmpty()) {
                // 为二级表寻找无冲突的哈希种子
                int seed = findPerfectSeed(bucket);
                hashSeeds[i] = seed;

                // 创建二级哈希表
                int secondarySize = bucket.size() * bucket.size(); // 平方大小保证无冲突
                primaryTable[i] = new SecondaryTable<>(secondarySize);

                // 插入数据到二级表
                for (Pair<K, V> pair : bucket) {
                    int secondaryHash = Math.abs((pair.getKey().hashCode() ^ seed) %
secondarySize);
                    primaryTable[i].put(pair.getKey(), pair.getValue(), secondaryHash);
                }
            }
        }

        private int findPerfectSeed(List<Pair<K, V>> bucket) {
            int size = bucket.size() * bucket.size();
            int seed = 1;

            // 尝试不同的种子直到找到无冲突的
            while (true) {
                Set<Integer> hashes = new HashSet<>();
                boolean collision = false;

                for (Pair<K, V> pair : bucket) {
                    int hash = Math.abs((pair.getKey().hashCode() ^ seed) % size);
                    if (!hashes.add(hash)) {
                        collision = true;
                        break;
                    }
                }

                if (!collision) {
                    return seed;
                }

                seed++;
```

```
            if (seed > 1000) { // 防止无限循环
                throw new RuntimeException("Cannot find perfect hash seed");
            }
        }
    }

    @SuppressWarnings("unchecked")
    public V get(K key) {
        int primaryHash = Math.abs(key.hashCode() % primarySize);
        if (primaryTable[primaryHash] == null) {
            return null;
        }

        int secondaryHash = Math.abs((key.hashCode() ^ hashSeeds[primaryHash]) %
                (primaryTable[primaryHash].keys.length));
        return primaryTable[primaryHash].get(key, secondaryHash);
    }
}

/**
 * 计数布隆过滤器 (Counting Bloom Filter) 实现
 *
 * 应用场景: 需要支持删除操作的布隆过滤器变种
 * 算法原理: 使用计数器数组代替位数组，支持元素的添加和删除
 * 优势: 支持删除操作，保持布隆过滤器的空间效率
 * 限制: 空间使用略高于标准布隆过滤器
 */
static class CountingBloomFilter {
    private final int[] counters; // 计数器数组
    private final int size; // 数组大小
    private final int[] seeds; // 多个哈希函数的种子

    public CountingBloomFilter(int size, int hashFunctions) {
        this.size = size;
        this.counters = new int[size];
        this.seeds = new int[hashFunctions];
        // 初始化哈希函数种子
        for (int i = 0; i < hashFunctions; i++) {
            seeds[i] = i * 100 + 31; // 使用不同的种子
        }
    }

    // 添加元素
```

```
public void add(String element) {
    for (int seed : seeds) {
        int hash = getHash(element, seed);
        counters[hash]++;
    }
}

// 删除元素
public void remove(String element) {
    for (int seed : seeds) {
        int hash = getHash(element, seed);
        if (counters[hash] > 0) {
            counters[hash]--;
        }
    }
}

// 判断元素是否可能存在
public boolean mightContain(String element) {
    for (int seed : seeds) {
        int hash = getHash(element, seed);
        if (counters[hash] == 0) {
            return false; // 只要有一个位置为0，元素一定不存在
        }
    }
    return true; // 所有位置都大于0，元素可能存在
}

// 获取元素计数（近似值）
public int getCount(String element) {
    int minCount = Integer.MAX_VALUE;
    for (int seed : seeds) {
        int hash = getHash(element, seed);
        minCount = Math.min(minCount, counters[hash]);
    }
    return minCount;
}

// 哈希函数
private int getHash(String element, int seed) {
    int hash = 0;
    for (int i = 0; i < element.length(); i++) {
        hash = seed * hash + element.charAt(i);
```

```
        }
        return Math.abs(hash % size);
    }
}

/**
 * 可扩展哈希 (Extendible Hashing) 实现
 *
 * 应用场景: 数据库索引、文件系统等需要动态扩展的哈希结构
 * 算法原理: 使用目录结构指向桶，当桶满时进行分裂，目录深度动态调整
 * 优势: 支持动态扩展，保持较好的性能
 * 限制: 目录结构增加了一定的空间开销
 */
static class ExtendibleHashTable<K, V> {
    private static class Bucket<K, V> {
        private static final int BUCKET_SIZE = 4; // 桶容量
        private List<Pair<K, V>> entries;
        private int localDepth;

        public Bucket(int localDepth) {
            this.entries = new ArrayList<>();
            this.localDepth = localDepth;
        }

        public boolean isFull() {
            return entries.size() >= BUCKET_SIZE;
        }

        public void add(K key, V value) {
            entries.add(new Pair<>(key, value));
        }

        public V get(K key) {
            for (Pair<K, V> entry : entries) {
                if (entry.getKey().equals(key)) {
                    return entry.getValue();
                }
            }
            return null;
        }

        public void remove(K key) {
            entries.removeIf(entry -> entry.getKey().equals(key));
```

```
    }

    public List<Pair<K, V>> getEntries() {
        return new ArrayList<>(entries);
    }
}

private List<Bucket<K, V>> directory;
private int globalDepth;

@SuppressWarnings("unchecked")
public ExtendibleHashTable() {
    this.globalDepth = 1;
    this.directory = new ArrayList<>();
    // 初始化目录
    directory.add(new Bucket<>(1));
    directory.add(new Bucket<>(1));
}

private int getDirectoryIndex(K key) {
    int hash = key.hashCode();
    // 取哈希值的低 globalDepth 位
    return hash & ((1 << globalDepth) - 1);
}

public void put(K key, V value) {
    int index = getDirectoryIndex(key);
    Bucket<K, V> bucket = directory.get(index);

    if (!bucket.isFull()) {
        bucket.add(key, value);
    } else {
        // 桶满，需要分裂
        splitBucket(index);
        // 重新插入
        put(key, value);
    }
}

@SuppressWarnings("unchecked")
private void splitBucket(int index) {
    Bucket<K, V> oldBucket = directory.get(index);
```

```
        if (oldBucket.localDepth == globalDepth) {
            // 需要扩展目录
            extendDirectory();
        }

        // 创建新桶
        Bucket<K, V> newBucket = new Bucket<>(oldBucket.localDepth + 1);
        oldBucket.localDepth++;

        // 重新分配条目
        List<Pair<K, V>> entries = oldBucket.getEntries();
        oldBucket.entries.clear();

        for (Pair<K, V> entry : entries) {
            int newIndex = getDirectoryIndex(entry.getKey());
            if ((newIndex & (1 << (oldBucket.localDepth - 1))) != 0) {
                newBucket.add(entry.getKey(), entry.getValue());
            } else {
                oldBucket.add(entry.getKey(), entry.getValue());
            }
        }

        // 更新目录指针
        for (int i = 0; i < directory.size(); i++) {
            if ((i & ((1 << globalDepth) - 1)) == index) {
                if ((i & (1 << (oldBucket.localDepth - 1))) != 0) {
                    directory.set(i, newBucket);
                } else {
                    directory.set(i, oldBucket);
                }
            }
        }
    }

    @SuppressWarnings("unchecked")
    private void extendDirectory() {
        int oldSize = directory.size();
        globalDepth++;

        // 扩展目录
        for (int i = 0; i < oldSize; i++) {
            Bucket<K, V> bucket = directory.get(i);
            directory.add(bucket);
```

```
        }
    }

    public V get(K key) {
        int index = getDirectoryIndex(key);
        Bucket<K, V> bucket = directory.get(index);
        return bucket.get(key);
    }

    public void remove(K key) {
        int index = getDirectoryIndex(key);
        Bucket<K, V> bucket = directory.get(index);
        bucket.remove(key);

        // TODO: 实现桶合并逻辑
    }
}

/**
 * 线性哈希 (Linear Hashing) 实现
 *
 * 应用场景: 数据库系统、文件系统等需要渐进式扩展的哈希结构
 * 算法原理: 使用线性探测和分裂策略，避免目录结构的空间开销
 * 优势: 渐进式扩展，无目录结构开销
 * 限制: 分裂过程可能影响性能
 */
static class LinearHashTable<K, V> {
    private static final int INITIAL_SIZE = 4;
    private static final double LOAD_FACTOR = 0.75;

    private List<Bucket<K, V>> buckets;
    private int nextSplit;
    private int level;
    private int size;

    private static class Bucket<K, V> {
        private List<Pair<K, V>> entries;

        public Bucket() {
            this.entries = new ArrayList<>();
        }

        public void add(K key, V value) {
```

```
        entries.add(new Pair<>(key, value));
    }

    public V get(K key) {
        for (Pair<K, V> entry : entries) {
            if (entry.getKey().equals(key)) {
                return entry.getValue();
            }
        }
        return null;
    }

    public void remove(K key) {
        entries.removeIf(entry -> entry.getKey().equals(key));
    }

    public List<Pair<K, V>> getEntries() {
        return new ArrayList<>(entries);
    }
}

@SuppressWarnings("unchecked")
public LinearHashTable() {
    this.buckets = new ArrayList<>();
    this.nextSplit = 0;
    this.level = 0;
    this.size = 0;

    // 初始化桶
    for (int i = 0; i < INITIAL_SIZE; i++) {
        buckets.add(new Bucket<>());
    }
}

private int hash(K key, int level) {
    int hash = key.hashCode();
    int mask = (1 << level) - 1;
    return hash & mask;
}

public void put(K key, V value) {
    int index = getBucketIndex(key);
    Bucket<K, V> bucket = buckets.get(index);
```

```
        // 检查是否已存在
        V existing = bucket.get(key);
        if (existing != null) {
            // 更新值
            bucket.remove(key);
            bucket.add(key, value);
            return;
        }

        bucket.add(key, value);
        size++;

        // 检查是否需要分裂
        if ((double) size / buckets.size() > LOAD_FACTOR) {
            split();
        }
    }

    private int getBucketIndex(K key) {
        int hash1 = hash(key, level);
        int hash2 = hash(key, level + 1);

        if (hash1 < nextSplit) {
            return hash2;
        } else {
            return hash1;
        }
    }

    @SuppressWarnings("unchecked")
    private void split() {
        if (nextSplit >= (1 << level)) {
            // 进入下一层
            level++;
            nextSplit = 0;
        }

        // 分裂桶
        Bucket<K, V> oldBucket = buckets.get(nextSplit);
        Bucket<K, V> newBucket = new Bucket<>();

        // 重新分配条目
```

```
        List<Pair<K, V>> entries = oldBucket.getEntries();
        oldBucket.entries.clear();

        for (Pair<K, V> entry : entries) {
            int newIndex = hash(entry.getKey(), level + 1);
            if (newIndex == nextSplit + (1 << level)) {
                newBucket.add(entry.getKey(), entry.getValue());
            } else {
                oldBucket.add(entry.getKey(), entry.getValue());
            }
        }

        // 添加新桶
        buckets.add(newBucket);
        nextSplit++;
    }

    public V get(K key) {
        int index = getBucketIndex(key);
        Bucket<K, V> bucket = buckets.get(index);
        return bucket.get(key);
    }

    public void remove(K key) {
        int index = getBucketIndex(key);
        Bucket<K, V> bucket = buckets.get(index);
        V value = bucket.get(key);
        if (value != null) {
            bucket.remove(key);
            size--;
        }
    }
}

/**
 * 剑指 Offer 50. 第一个只出现一次的字符
 * 题目来源: https://leetcode.cn/problems/di-yi-ge-zhi-chu-xian-yi-ci-de-zi-fu-lcof/
 *
 * 题目描述:
 * 在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。
 *
 * 算法思路:
 * 使用哈希表统计每个字符出现的次数
```

```
 * 再次遍历字符串找到第一个出现次数为 1 的字符
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)，因为字符集大小固定
 */
public static char firstUniqChar(String s) {
    int[] count = new int[256]; // ASCII 字符集

    for (char c : s.toCharArray()) {
        count[c]++;
    }

    for (char c : s.toCharArray()) {
        if (count[c] == 1) {
            return c;
        }
    }

    return ' ';
}

/**
 * 剑指 Offer 03. 数组中重复的数字
 * 题目来源: https://leetcode.cn/problems/shu-zu-zhong-zhong-fu-de-shu-zi-lcof/
 *
 * 题目描述:
 * 在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。
 * 数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。
 * 请找出数组中任意一个重复的数字。
 *
 * 算法思路:
 * 使用哈希集合记录已经出现过的数字
 * 当遇到已经在集合中的数字时返回
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static int findRepeatNumber(int[] nums) {
    Set<Integer> set = new HashSet<>();

    for (int num : nums) {
        if (set.contains(num)) {
            return num;
```

```
            }
            set.add(num);
        }

        return -1;
    }

    /**
     * 剑指 Offer 48. 最长不含重复字符的子字符串
     * 题目来源: https://leetcode.cn/problems/zui-chang-bu-han-zhong-fu-zi-fu-de-zi-zi-fu-chuan-lcof/
     *
     * 题目描述:
     * 请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。
     *
     * 算法思路:
     * 使用滑动窗口和哈希表记录字符最后出现的位置
     * 当遇到重复字符时移动窗口左边界
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)，字符集大小固定
     */
    public static int lengthOfLongestSubstring2(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLength = 0;
        int left = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            if (map.containsKey(c) && map.get(c) >= left) {
                left = map.get(c) + 1;
            }
            map.put(c, right);
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    /**
     * HDU 4821 - String (字符串)
     * 题目来源: http://acm.hdu.edu.cn/showproblem.php?pid=4821
     *
```

```
     * 题目描述:
     * 给定字符串 s 和整数 M,L，统计有多少个长度为 M*L 的子串，使得该子串可以分成 M 个长度为 L 的不同子串。
     *
     * 算法思路:
     * 使用滚动哈希计算所有长度为 L 的子串哈希值
     * 使用滑动窗口统计长度为 M*L 的窗口中不同子串的数量
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    public static int countValidSubstrings(String s, int M, int L) {
        int n = s.length();
        if (n < M * L) return 0;

        long base = 131;
        long mod = (long)1e9 + 7;

        // 计算所有长度为 L 的子串哈希值
        long[] hashes = new long[n - L + 1];
        long power = 1;

        for (int i = 0; i < L; i++) {
            power = (power * base) % mod;
        }

        long hash = 0;
        for (int i = 0; i < n; i++) {
            hash = (hash * base + (s.charAt(i) - 'a' + 1)) % mod;
            if (i >= L) {
                hash = (hash - (s.charAt(i - L) - 'a' + 1) * power % mod + mod) % mod;
            }
            if (i >= L - 1) {
                hashes[i - L + 1] = hash;
            }
        }

        int count = 0;

        for (int start = 0; start < L; start++) {
            if (start + M * L > n) break;

            Map<Long, Integer> freq = new HashMap<>();
```

```
        // 初始化第一个窗口
        for (int i = 0; i < M; i++) {
            int pos = start + i * L;
            long h = hashes[pos];
            freq.put(h, freq.getOrDefault(h, 0) + 1);
        }

        if (freq.size() == M) {
            count++;
        }

        // 滑动窗口
        for (int i = start + L; i + M * L <= n; i += L) {
            // 移除最左边的子串
            long removeHash = hashes[i - L];
            freq.put(removeHash, freq.get(removeHash) - 1);
            if (freq.get(removeHash) == 0) {
                freq.remove(removeHash);
            }

            // 添加最右边的子串
            long addHash = hashes[i + (M - 1) * L];
            freq.put(addHash, freq.getOrDefault(addHash, 0) + 1);

            if (freq.size() == M) {
                count++;
            }
        }
    }

    return count;
}

/**
 * POJ 2774 - Long Long Message (最长公共子串)
 * 题目来源: http://poj.org/problem?id=2774
 *
 * 题目描述:
 * 求两个字符串的最长公共子串长度。
 *
 * 算法思路:
 * 使用二分答案+滚动哈希
```

```
 * 对可能的长度进行二分，检查是否存在长度为 mid 的公共子串
 *
 * 时间复杂度: O((n+m)log(min(n,m)))
 * 空间复杂度: O(n+m)
 */
public static int longestCommonSubstring(String s1, String s2) {
    int n = s1.length(), m = s2.length();
    int left = 0, right = Math.min(n, m);
    int result = 0;

    long base = 131;
    long mod = (long)1e9 + 7;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (mid == 0) {
            left = mid + 1;
            continue;
        }

        Set<Long> set = new HashSet<>();
        long hash = 0;
        long power = 1;

        // 计算 s1 中所有长度为 mid 的子串哈希值
        for (int i = 0; i < mid; i++) {
            power = (power * base) % mod;
        }

        for (int i = 0; i < n; i++) {
            hash = (hash * base + (s1.charAt(i) - 'a' + 1)) % mod;
            if (i >= mid) {
                hash = (hash - (s1.charAt(i - mid) - 'a' + 1) * power % mod + mod) % mod;
            }
            if (i >= mid - 1) {
                set.add(hash);
            }
        }

        // 检查 s2 中是否存在相同的哈希值
        hash = 0;
        boolean found = false;
```

```
        for (int i = 0; i < m; i++) {
            hash = (hash * base + (s2.charAt(i) - 'a' + 1)) % mod;
            if (i >= mid) {
                hash = (hash - (s2.charAt(i - mid) - 'a' + 1) * power % mod + mod) % mod;
            }
            if (i >= mid - 1 && set.contains(hash)) {
                found = true;
                break;
            }
        }

        if (found) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

/**
 * SPOJ - SUBST1 (不同子串数量)
 * 题目来源: https://www.spoj.com/problems/SUBST1/
 *
 * 题目描述:
 * 给定一个字符串，计算其不同子串的数量。
 *
 * 算法思路:
 * 使用滚动哈希计算所有子串的哈希值，存储在集合中
 * 集合的大小即为不同子串的数量
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n^2)
 */
public static int countDistinctSubstrings(String s) {
    int n = s.length();
    Set<Long> set = new HashSet<>();
    long base = 131;
    long mod = (long)1e9 + 7;
```

```
    for (int i = 0; i < n; i++) {
        long hash = 0;
        for (int j = i; j < n; j++) {
            hash = (hash * base + (s.charAt(j) - 'a' + 1)) % mod;
            set.add(hash);
        }
    }

    return set.size();
}

/**
 * AtCoder ABC 284 E - Count Simple Paths (简单路径计数)
 * 题目来源: https://atcoder.jp/contests/abc284/tasks/abc284_e
 *
 * 题目描述:
 * 给定无向图，计算从节点 1 开始的不同简单路径数量。
 * 简单路径指路径中不包含重复节点。
 *
 * 算法思路:
 * 使用 DFS 遍历所有路径，使用哈希集合记录访问过的节点
 * 使用滚动哈希记录路径的哈希值，避免重复计数
 *
 * 时间复杂度: O(2^n)
 * 空间复杂度: O(n)
 */
public static int countSimplePaths(int n, int[][] edges) {
    List<Integer>[] graph = new ArrayList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0], v = edge[1];
        graph[u].add(v);
        graph[v].add(u);
    }

    Set<Long> pathHashes = new HashSet<>();
    long base = 131;
    long mod = (long)1e9 + 7;

    dfs(1, graph, new boolean[n + 1], 0L, pathHashes, base, mod);
```

```
    return pathHashes.size();
}

private static void dfs(int node, List<Integer>[] graph, boolean[] visited,
                        long currentHash, Set<Long> pathHashes, long base, long mod) {
    visited[node] = true;
    currentHash = (currentHash * base + node) % mod;
    pathHashes.add(currentHash);

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, graph, visited, currentHash, pathHashes, base, mod);
        }
    }

    visited[node] = false;
}

/**
 * USACO 2019 December Contest, Gold - Milk Visits (牛奶访问)
 * 题目来源: http://www.usaco.org/index.php?page=viewproblem2&cpid=970
 *
 * 题目描述:
 * 给定一棵树，每个节点有一种牛奶类型。
 * 多个查询，每个查询问从 u 到 v 的路径上是否包含特定类型的牛奶。
 *
 * 算法思路:
 * 使用 LCA 和路径哈希，对每种牛奶类型建立哈希值
 * 查询时检查路径哈希值是否包含目标牛奶类型的哈希
 *
 * 时间复杂度: O(n + q log n)
 * 空间复杂度: O(n)
 */
public static boolean[] milkVisits(int n, int[] milkTypes, int[][] edges, int[][] queries) {
    // 构建树
    List<Integer>[] tree = new ArrayList[n + 1];
    for (int i = 1; i <= n; i++) {
        tree[i] = new ArrayList<>();
    }

    for (int[] edge : edges) {
        int u = edge[0], v = edge[1];
```

```
    tree[u].add(v);
    tree[v].add(u);
}

// LCA 预处理
int LOG = 20;
int[][] parent = new int[n + 1][LOG];
int[] depth = new int[n + 1];
long[][] pathHash = new long[n + 1][LOG];
long base = 131;
long mod = (long)1e9 + 7;

// BFS 预处理
Queue<Integer> queue = new LinkedList<>();
queue.offer(1);
depth[1] = 0;

while (!queue.isEmpty()) {
    int u = queue.poll();
    for (int v : tree[u]) {
        if (v == parent[u][0]) continue;

        depth[v] = depth[u] + 1;
        parent[v][0] = u;
        pathHash[v][0] = milkTypes[v - 1];

        for (int i = 1; i < LOG; i++) {
            parent[v][i] = parent[parent[v][i - 1]][i - 1];
            pathHash[v][i] = (pathHash[v][i - 1] * base +
                              pathHash[parent[v][i - 1]][i - 1]) % mod;
        }

        queue.offer(v);
    }
}

boolean[] results = new boolean[queries.length];

for (int i = 0; i < queries.length; i++) {
    int u = queries[i][0], v = queries[i][1], targetMilk = queries[i][2];

    // 计算 u 到 v 路径的哈希值
    long hash = 0;
```

```
        if (depth[u] < depth[v]) {
            int temp = u;
            u = v;
            v = temp;
        }

        // 提升 u 到 v 的深度
        for (int j = LOG - 1; j >= 0; j--) {
            if (depth[u] - (1 << j) >= depth[v]) {
                hash = (hash * base + pathHash[u][j]) % mod;
                u = parent[u][j];
            }
        }

        if (u == v) {
            hash = (hash * base + milkTypes[u - 1]) % mod;
        } else {
            for (int j = LOG - 1; j >= 0; j--) {
                if (parent[u][j] != parent[v][j]) {
                    hash = (hash * base + pathHash[u][j]) % mod;
                    hash = (hash * base + pathHash[v][j]) % mod;
                    u = parent[u][j];
                    v = parent[v][j];
                }
            }
            hash = (hash * base + milkTypes[u - 1]) % mod;
            hash = (hash * base + milkTypes[v - 1]) % mod;
            hash = (hash * base + milkTypes[parent[u][0] - 1]) % mod;
        }

        // 检查哈希值是否包含目标牛奶
        long targetHash = targetMilk;
        String pathHashStr = Long.toString(hash);
        String targetHashStr = Long.toString(targetHash);

        results[i] = pathHashStr.contains(targetHashStr);
    }

    return results;
}

/**
```

```
 * 完美哈希 (Perfect Hashing)
 * 应用场景: 静态数据集，需要O(1)查找时间且无冲突
 * 算法原理: 使用两级哈希表，第一级哈希将元素分组，第二级为每个组创建无冲突的哈希表
 * 优势: 保证O(1)查找时间，无哈希冲突
 * 限制: 仅适用于静态数据集，构建过程较复杂
 */
static class PerfectHash {
    private int[][] secondLevelTables;
    private int[] firstLevelTable;
    private String[] keys;

    public PerfectHash(String[] keys) {
        this.keys = keys;
        buildPerfectHash();
    }

    private void buildPerfectHash() {
        int n = keys.length;
        firstLevelTable = new int[n];
        secondLevelTables = new int[n][];

        // 第一级哈希：将元素分组
        List<List<String>> groups = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            groups.add(new ArrayList<>());
        }

        for (String key : keys) {
            int hash = Math.abs(key.hashCode()) % n;
            groups.get(hash).add(key);
        }

        // 第二级哈希：为每个组创建无冲突哈希表
        for (int i = 0; i < n; i++) {
            List<String> group = groups.get(i);
            if (group.isEmpty()) {
                secondLevelTables[i] = new int[0];
                continue;
            }

            int size = group.size() * group.size(); // 平方大小以减少冲突
            int[] table = new int[size];
            Arrays.fill(table, -1);
```

```
            boolean success = false;
            while (!success) {
                success = true;
                Arrays.fill(table, -1);

                for (String key : group) {
                    int hash = Math.abs(key.hashCode()) % size;
                    if (table[hash] != -1) {
                        success = false;
                        break;
                    }
                    table[hash] = Arrays.asList(keys).indexOf(key);
                }

                if (!success) {
                    size *= 2; // 如果冲突，增加表大小
                    table = new int[size];
                }
            }

            secondLevelTables[i] = table;
        }
    }

    public int get(String key) {
        int firstHash = Math.abs(key.hashCode()) % keys.length;
        int[] secondTable = secondLevelTables[firstHash];

        if (secondTable.length == 0) {
            return -1;
        }

        int secondHash = Math.abs(key.hashCode()) % secondTable.length;
        int index = secondTable[secondHash];

        if (index != -1 && keys[index].equals(key)) {
            return index;
        }

        return -1;
    }
}
```

```
/**
 * 计数布隆过滤器 (Counting Bloom Filter)
 * 应用场景: 需要支持删除操作的布隆过滤器变种
 * 算法原理: 使用计数器数组代替位数组，支持元素的添加和删除
 * 优势: 支持删除操作，保持布隆过滤器的空间效率
 * 限制: 空间使用略高于标准布隆过滤器
 */
static class CountingBloomFilter {
    private int[] counters;
    private int numHashFunctions;
    private int size;

    public CountingBloomFilter(int size, int numHashFunctions) {
        this.size = size;
        this.numHashFunctions = numHashFunctions;
        this.counters = new int[size];
    }

    private int[] getHashes(String element) {
        int[] hashes = new int[numHashFunctions];
        int hash1 = Math.abs(element.hashCode());
        int hash2 = hash1 * 31 + 17;

        for (int i = 0; i < numHashFunctions; i++) {
            hashes[i] = Math.abs(hash1 + i * hash2) % size;
        }

        return hashes;
    }

    public void add(String element) {
        int[] hashes = getHashes(element);
        for (int hash : hashes) {
            counters[hash]++;
        }
    }

    public void remove(String element) {
        int[] hashes = getHashes(element);
        for (int hash : hashes) {
            if (counters[hash] > 0) {
                counters[hash]--;
```

```
            }
        }
    }

    public int count(String element) {
        int[] hashes = getHashes(element);
        int minCount = Integer.MAX_VALUE;

        for (int hash : hashes) {
            minCount = Math.min(minCount, counters[hash]);
        }

        return minCount;
    }

    public boolean mightContain(String element) {
        return count(element) > 0;
    }
}

/**
 * 可扩展哈希 (Extendible Hashing)
 * 应用场景: 数据库索引、文件系统等需要动态扩展的哈希结构
 * 算法原理: 使用目录结构指向桶，当桶满时进行分裂，目录深度动态调整
 * 优势: 支持动态扩展，保持较好的性能
 * 限制: 目录结构增加了一定的空间开销
 */
static class ExtendibleHash {
    static class Bucket {
        List<Entry> entries;
        int localDepth;

        Bucket(int localDepth) {
            this.localDepth = localDepth;
            this.entries = new ArrayList<>();
        }
    }

    static class Entry {
        String key;
        String value;

        Entry(String key, String value) {
```

```
        this.key = key;
        this.value = value;
    }
}

private Bucket[] directory;
private int globalDepth;
private int bucketSize;

public ExtendibleHash(int bucketSize) {
    this.bucketSize = bucketSize;
    this.globalDepth = 1;
    this.directory = new Bucket[2];
    directory[0] = new Bucket(1);
    directory[1] = new Bucket(1);
}

private int getHash(String key) {
    return Math.abs(key.hashCode()) & ((1 << globalDepth) - 1);
}

public void put(String key, String value) {
    int hash = getHash(key);
    Bucket bucket = directory[hash];

    // 检查是否已存在该键
    for (Entry entry : bucket.entries) {
        if (entry.key.equals(key)) {
            entry.value = value;
            return;
        }
    }

    // 如果桶已满，需要分裂
    if (bucket.entries.size() >= bucketSize) {
        splitBucket(hash);
        put(key, value); // 重新尝试插入
    } else {
        bucket.entries.add(new Entry(key, value));
    }
}

private void splitBucket(int hash) {
```

```
        Bucket bucket = directory[hash];

        if (bucket.localDepth == globalDepth) {
            // 需要扩展目录
            extendDirectory();
        }

        // 创建新桶
        Bucket newBucket = new Bucket(bucket.localDepth + 1);

        // 重新分配条目
        List<Entry> oldEntries = new ArrayList<>(bucket.entries);
        bucket.entries.clear();

        int newLocalDepth = bucket.localDepth + 1;
        int mask = (1 << newLocalDepth) - 1;

        for (Entry entry : oldEntries) {
            int newHash = Math.abs(entry.key.hashCode()) & mask;
            if ((newHash & (1 << bucket.localDepth)) != 0) {
                newBucket.entries.add(entry);
            } else {
                bucket.entries.add(entry);
            }
        }

        // 更新目录指针
        int step = 1 << bucket.localDepth;
        for (int i = hash; i < directory.length; i += step) {
            if ((i & (1 << bucket.localDepth)) != 0) {
                directory[i] = newBucket;
            }
        }

        bucket.localDepth++;
    }

    private void extendDirectory() {
        int oldSize = directory.length;
        Bucket[] newDirectory = new Bucket[oldSize * 2];

        for (int i = 0; i < oldSize; i++) {
            newDirectory[i] = directory[i];
```

```
            newDirectory[i + oldSize] = directory[i];
        }

        directory = newDirectory;
        globalDepth++;
    }

    public String get(String key) {
        int hash = getHash(key);
        Bucket bucket = directory[hash];

        for (Entry entry : bucket.entries) {
            if (entry.key.equals(key)) {
                return entry.value;
            }
        }

        return null;
    }

    public int getDepth() {
        return globalDepth;
    }
}

/**
 * 线性哈希 (Linear Hashing)
 * 应用场景: 数据库系统、文件系统等需要渐进式扩展的哈希结构
 * 算法原理: 使用线性探测和分裂策略，避免目录结构的空间开销
 * 优势: 渐进式扩展，无目录结构开销
 * 限制: 分裂过程可能影响性能
 */
static class LinearHash {
    static class Entry {
        String key;
        String value;
        boolean deleted;

        Entry(String key, String value) {
            this.key = key;
            this.value = value;
            this.deleted = false;
        }
```

```
}

private List<Entry>[] table;
private int size;
private int splitPointer;
private int threshold;
private double loadFactor;

public LinearHash(int initialSize) {
    this.table = new ArrayList[initialSize];
    for (int i = 0; i < initialSize; i++) {
        table[i] = new ArrayList<>();
    }
    this.size = 0;
    this.splitPointer = 0;
    this.threshold = initialSize;
    this.loadFactor = 0.75;
}

private int hash1(String key) {
    return Math.abs(key.hashCode()) % table.length;
}

private int hash2(String key) {
    return Math.abs(key.hashCode()) % (table.length * 2);
}

public void put(String key, String value) {
    int hash = hash1(key);

    // 检查是否已存在
    for (Entry entry : table[hash]) {
        if (entry.key.equals(key) && !entry.deleted) {
            entry.value = value;
            return;
        }
    }

    // 插入新条目
    table[hash].add(new Entry(key, value));
    size++;

    // 检查是否需要分裂
```

```
        if (size > threshold * loadFactor) {
            split();
        }
    }

    private void split() {
        if (splitPointer >= table.length) {
            return;
        }

        // 创建新桶
        List<Entry>[] newTable = new ArrayList[table.length + 1];
        for (int i = 0; i < table.length; i++) {
            newTable[i] = table[i];
        }
        newTable[table.length] = new ArrayList<>();

        // 重新哈希 splitPointer 指向的桶
        List<Entry> oldBucket = table[splitPointer];
        List<Entry> newBucket = new ArrayList<>();

        for (Entry entry : oldBucket) {
            if (!entry.deleted) {
                int newHash = hash2(entry.key);
                if (newHash == table.length) {
                    newBucket.add(entry);
                }
            }
        }

        // 更新桶
        oldBucket.clear();
        for (Entry entry : newBucket) {
            oldBucket.add(entry);
        }

        table = newTable;
        splitPointer++;

        if (splitPointer >= table.length) {
            splitPointer = 0;
            threshold = table.length;
        }
```

```
    }

    public String get(String key) {
        int hash = hash1(key);

        for (Entry entry : table[hash]) {
            if (entry.key.equals(key) && !entry.deleted) {
                return entry.value;
            }
        }

        return null;
    }

    public int size() {
        return size;
    }
}

public static void main(String[] args) {
    System.out.println("支持的哈希算法 : ");
    Hash.showAlgorithms();
    System.out.println();

    String algorithm = "MD5";
    Hash hash = new Hash(algorithm);
    String str1 = "zuochengyunzuochengyunzuochengyun1";
    String str2 = "zuochengyunzuochengyunzuochengyun2";
    String str3 = "zuochengyunzuochengyunzuochengyun3";
    String str4 = "zuochengyunzuochengyunZuochengyun1";
    String str5 = "zuochengyunzuoChengyunzuochengyun2";
    String str6 = "zuochengyunzuochengyunzuochengyUn3";
    String str7 = "zuochengyunzuochengyunzuochengyun1";
    System.out.println("7 个字符串得到的哈希值 : ");
    System.out.println(hash.hashValue(str1));
    System.out.println(hash.hashValue(str2));
    System.out.println(hash.hashValue(str3));
    System.out.println(hash.hashValue(str4));
    System.out.println(hash.hashValue(str5));
    System.out.println(hash.hashValue(str6));
    System.out.println(hash.hashValue(str7));
    System.out.println();
```

```
        // 测试 LintCode 128 题
        System.out.println("=== LintCode 128. Hash Function ===");
        String key = "abcd";
        int HASH_SIZE = 100;
        int result = hashCode(key, HASH_SIZE);
        System.out.println("Key: " + key + ", HASH_SIZE: " + HASH_SIZE + ", Result: " + result);
        System.out.println();

        char[] arr = { 'a', 'b' };
        int n = 20;
        System.out.println("生成长度为 n，字符来自 arr，所有可能的字符串");
        List<String> strs = generateStrings(arr, n);
//      for (String str : strs) {
//          System.out.println(str);
//      }
        System.out.println("不同字符串的数量 : " + strs.size());
        HashSet<String> set = new HashSet<>();
        for (String str : strs) {
            set.add(hash.hashValue(str));
        }
//      for (String str : set) {
//          System.out.println(str);
//      }
        System.out.println("不同哈希值的数量 : " + set.size());
        System.out.println();

        int m = 13;
        int[] cnts = new int[m];
        System.out.println("现在看看这些哈希值，% " + m + " 之后的余数分布情况");
        BigInteger mod = new BigInteger(String.valueOf(m));
        for (String hashCode : set) {
            BigInteger bigInt = new BigInteger(hashCode, 16);
            int ans = bigInt.mod(mod).intValue();
            cnts[ans]++;
        }
        for (int i = 0; i < m; i++) {
            System.out.println("余数 " + i + " 出现了 " + cnts[i] + " 次");
        }

        // 测试 LeetCode 705. Design HashSet
        System.out.println("\n=== LeetCode 705. Design HashSet ===");
        MyHashSet myHashSet = new MyHashSet();
        myHashSet.add(1);      // set = [1]
```

```
        myHashSet.add(2);      // set = [1, 2]
        System.out.println(myHashSet.contains(1)); // 返回 True
        System.out.println(myHashSet.contains(3)); // 返回 False （未找到）
        myHashSet.add(2);      // set = [1, 2]
        System.out.println(myHashSet.contains(2)); // 返回 True
        myHashSet.remove(2);   // set = [1]
        System.out.println(myHashSet.contains(2)); // 返回 False （已移除）

        // 测试 LeetCode 706. Design HashMap
        System.out.println("\n=== LeetCode 706. Design HashMap ===");
        MyHashMap myHashMap = new MyHashMap();
        myHashMap.put(1, 1); // myHashMap 现在为 [[1,1]]
        myHashMap.put(2, 2); // myHashMap 现在为 [[1,1], [2,2]]
        System.out.println(myHashMap.get(1));    // 返回 1 ，myHashMap 现在为 [[1,1], [2,2]]
        System.out.println(myHashMap.get(3));    // 返回 -1（未找到），myHashMap 现在为 [[1,1],
[2,2]]
        myHashMap.put(2, 1); // myHashMap 现在为 [[1,1], [2,1]]（更新已有的值）
        System.out.println(myHashMap.get(2));    // 返回 1 ，myHashMap 现在为 [[1,1], [2,1]]
        myHashMap.remove(2); // 删除键为 2 的数据，myHashMap 现在为 [[1,1]]
        System.out.println(myHashMap.get(2));    // 返回 -1（未找到），myHashMap 现在为 [[1,1]]

        // 测试 LeetCode 28. Find the Index of the First Occurrence in a String
        System.out.println("\n=== LeetCode 28. Find the Index of the First Occurrence in a String
===");
        System.out.println(strStr("sadbutsad", "sad")); // 返回 0
        System.out.println(strStr("leetcode", "leeto")); // 返回 -1

        // 测试 LeetCode 187. Repeated DNA Sequences
        System.out.println("\n=== LeetCode 187. Repeated DNA Sequences ===");
        List<String> dnaResult = findRepeatedDnaSequences("AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT");
        System.out.println(dnaResult); // ["AAAAACCCCC","CCCCCAAAAA"]

        // 测试 LeetCode 214. Shortest Palindrome
        System.out.println("\n=== LeetCode 214. Shortest Palindrome ===");
        System.out.println(shortestPalindrome("aacecaaa")); // "aaacecaaa"
        System.out.println(shortestPalindrome("abcd")); // "dcbabcd"

        // 测试一致性哈希
        System.out.println("\n=== 一致性哈希 (Consistent Hashing) ===");
        ConsistentHash consistentHash = new ConsistentHash(100); // 每个服务器 100 个虚拟节点
        consistentHash.addServer("Server1");
        consistentHash.addServer("Server2");
        consistentHash.addServer("Server3");
```

```
        System.out.println("键 'user1' 分配到的服务器: " + consistentHash.getServer("user1"));
        System.out.println("键 'user2' 分配到的服务器: " + consistentHash.getServer("user2"));
        System.out.println("键 'user3' 分配到的服务器: " + consistentHash.getServer("user3"));

        // 移除一个服务器后，观察键的重新分配情况
        System.out.println("\n移除 Server2 后:");
        consistentHash.removeServer("Server2");
        System.out.println("键 'user1' 分配到的服务器: " + consistentHash.getServer("user1"));
        System.out.println("键 'user2' 分配到的服务器: " + consistentHash.getServer("user2"));
        System.out.println("键 'user3' 分配到的服务器: " + consistentHash.getServer("user3"));

        // 测试布隆过滤器
        System.out.println("\n=== 布隆过滤器 (Bloom Filter) ===");
        BloomFilter bloomFilter = new BloomFilter(10000, 7); // 10000 位，7 个哈希函数
        bloomFilter.add("apple");
        bloomFilter.add("banana");
        bloomFilter.add("orange");

        System.out.println("'apple' 可能在集合中: " + bloomFilter.mightContain("apple")); // true
        System.out.println("'banana' 可能在集合中: " + bloomFilter.mightContain("banana")); //
true
        System.out.println("'orange' 可能在集合中: " + bloomFilter.mightContain("orange")); //
true
        System.out.println("'pear' 可能在集合中: " + bloomFilter.mightContain("pear")); // false
        System.out.println("'grape' 可能在集合中: " + bloomFilter.mightContain("grape")); // false

        // 测试双重哈希表
        System.out.println("\n=== 双重哈希 (Double Hashing) ===");
        DoubleHashTable<String, Integer> doubleHashTable = new DoubleHashTable<>();
        doubleHashTable.put("apple", 100);
        doubleHashTable.put("banana", 200);
        doubleHashTable.put("orange", 300);

        System.out.println("'apple' 的值: " + doubleHashTable.get("apple")); // 100
        System.out.println("'banana' 的值: " + doubleHashTable.get("banana")); // 200
        System.out.println("'orange' 的值: " + doubleHashTable.get("orange")); // 300
        System.out.println("'pear' 的值: " + doubleHashTable.get("pear")); // null

        doubleHashTable.remove("banana");
        System.out.println("移除 'banana' 后的值: " + doubleHashTable.get("banana")); // null
        System.out.println("哈希表大小: " + doubleHashTable.size()); // 2
```

```
        // 测试更多哈希相关题目
        System.out.println("
=== 更多哈希相关题目测试 ===");

        // 测试 LeetCode 1. Two Sum
        System.out.println("
=== LeetCode 1. Two Sum ===");
        int[] nums = {2, 7, 11, 15};
        int target = 9;
        int[] twoSumResult = twoSum(nums, target);
        System.out.println("nums: " + Arrays.toString(nums) + ", target: " + target + ", result: "
+ Arrays.toString(twoSumResult));

        // 测试 LeetCode 49. Group Anagrams
        System.out.println("
=== LeetCode 49. Group Anagrams ===");
        String[] strs = {"eat", "tea", "tan", "ate", "nat", "bat"};
        List<List<String>> anagramResult = groupAnagrams(strs);
        System.out.println("strs: " + Arrays.toString(strs));
        System.out.println("grouped anagrams: " + anagramResult);

        // 测试 LeetCode 242. Valid Anagram
        System.out.println("
=== LeetCode 242. Valid Anagram ===");
        String s1 = "anagram", s2 = "nagaram";
        boolean anagramCheck = isAnagram(s1, s2);
        System.out.println("'" + s1 + "' and '" + s2 + "' are anagrams: " + anagramCheck);

        // 测试 LeetCode 3. Longest Substring Without Repeating Characters
        System.out.println("
=== LeetCode 3. Longest Substring Without Repeating Characters ===");
        String s = "abcabcbb";
        int longestSubstring = lengthOfLongestSubstring(s);
        System.out.println("String: '" + s + "', longest substring length: " + longestSubstring);

        // 测试 LeetCode 76. Minimum Window Substring
        System.out.println("
=== LeetCode 76. Minimum Window Substring ===");
        String sStr = "ADOBECODEBANC";
        String tStr = "ABC";
        String minWindowResult = minWindow(sStr, tStr);
        System.out.println("s: '" + sStr + "', t: '" + tStr + "', min window: '" + minWindowResult
+ "'");
```

```
        // 测试 LeetCode 560. Subarray Sum Equals K
        System.out.println("
=== LeetCode 560. Subarray Sum Equals K ===");
        int[] sumNums = {1, 1, 1};
        int k = 2;
        int subarraySumResult = subarraySum(sumNums, k);
        System.out.println("nums: " + Arrays.toString(sumNums) + ", k: " + k + ", subarray count:
" + subarraySumResult);

        // 测试 LeetCode 347. Top K Frequent Elements
        System.out.println("
=== LeetCode 347. Top K Frequent Elements ===");
        int[] freqNums = {1, 1, 1, 2, 2, 3};
        int kFreq = 2;
        int[] topKFrequentResult = topKFrequent(freqNums, kFreq);
        System.out.println("nums: " + Arrays.toString(freqNums) + ", k: " + kFreq + ", top k
frequent: " + Arrays.toString(topKFrequentResult));

        // 测试 LeetCode 380. Insert Delete GetRandom O(1)
        System.out.println("
=== LeetCode 380. Insert Delete GetRandom O(1) ===");
        RandomizedSet randomizedSet = new RandomizedSet();
        System.out.println("Insert 1: " + randomizedSet.insert(1));
        System.out.println("Insert 2: " + randomizedSet.insert(2));
        System.out.println("Insert 1 again: " + randomizedSet.insert(1));
        System.out.println("Get random: " + randomizedSet.getRandom());
        System.out.println("Remove 2: " + randomizedSet.remove(2));
        System.out.println("Remove 3: " + randomizedSet.remove(3));
        System.out.println("Get random: " + randomizedSet.getRandom());

        // 测试 LeetCode 146. LRU Cache
        System.out.println("
=== LeetCode 146. LRU Cache ===");
        LRUCache lruCache = new LRUCache(2);
        lruCache.put(1, 1);
        lruCache.put(2, 2);
        System.out.println("Get 1: " + lruCache.get(1));
        lruCache.put(3, 3);  // 这会使得键 2 被移除
        System.out.println("Get 2: " + lruCache.get(2));
        System.out.println("Get 3: " + lruCache.get(3));

        // 测试 Codeforces 271D - Good Substrings
```

```
        System.out.println("
=== Codeforces 271D - Good Substrings ===");
        String sGood = "abacaba";
        String bad = "00000000000000000000000000"; // 所有字母都是好的
        int kGood = 1;
        int goodSubstrings = countGoodSubstrings(sGood, bad, kGood);
        System.out.println("String: " + sGood + ", bad: " + bad + ", k: " + kGood + ", good
substrings: " + goodSubstrings);

        // 测试 Codeforces 514C - Watto and Mechanism
        System.out.println("
=== Codeforces 514C - Watto and Mechanism ===");
        String[] dictionary = {"abc", "def", "ghi"};
        String[] queries = {"abc", "dbc", "efg"};
        boolean[] mechanismResults = wattoAndMechanism(dictionary, queries);
        System.out.println("Dictionary: " + Arrays.toString(dictionary));
        System.out.println("Queries: " + Arrays.toString(queries));
        System.out.println("Results: " + Arrays.toString(mechanismResults));

        // 测试 Codeforces 835D - Palindromic characteristics
        System.out.println("
=== Codeforces 835D - Palindromic characteristics ===");
        String sPal = "ababa";
        int[] palResults = palindromicCharacteristics(sPal);
        System.out.println("String: " + sPal);
        System.out.println("Palindromic characteristics: " + Arrays.toString(palResults));

        // 测试剑指 Offer 题目
        System.out.println("
=== 剑指 Offer 50. 第一个只出现一次的字符 ===");
        String sOffer = "abaccdeff";
        char firstUniq = firstUniqChar(sOffer);
        System.out.println("String: " + sOffer + ", first unique char: " + firstUniq);

        System.out.println("
=== 剑指 Offer 03. 数组中重复的数字 ===");
        int[] numsOffer = {2, 3, 1, 0, 2, 5, 3};
        int repeatNum = findRepeatNumber(numsOffer);
        System.out.println("Nums: " + Arrays.toString(numsOffer) + ", repeat number: " +
repeatNum);

        System.out.println("
=== 剑指 Offer 48. 最长不含重复字符的子字符串 ===");
```

```
        String sSubstring = "abcabcbb";
        int longestSubstring2 = lengthOfLongestSubstring2(sSubstring);
        System.out.println("String: " + sSubstring + ", longest substring length: " +
longestSubstring2);

        // 测试 HDU 4821 - String
        System.out.println("
=== HDU 4821 - String ===");
        String sHDU = "abcabcabc";
        int M = 3, L = 3;
        int validSubstrings = countValidSubstrings(sHDU, M, L);
        System.out.println("String: " + sHDU + ", M: " + M + ", L: " + L + ", valid substrings: "
+ validSubstrings);

        // 测试 POJ 2774 - Long Long Message
        System.out.println("
=== POJ 2774 - Long Long Message ===");
        String s1POJ = "abcdefg";
        String s2POJ = "cdefghij";
        int lcsLength = longestCommonSubstring(s1POJ, s2POJ);
        System.out.println("String1: " + s1POJ + ", String2: " + s2POJ + ", LCS length: " +
lcsLength);

        // 测试 SPOJ - SUBST1
        System.out.println("
=== SPOJ - SUBST1 ===");
        String sSPOJ = "abc";
        int distinctSubstrings = countDistinctSubstrings(sSPOJ);
        System.out.println("String: " + sSPOJ + ", distinct substrings: " + distinctSubstrings);

        // 测试 AtCoder ABC 284 E - Count Simple Paths
        System.out.println("
=== AtCoder ABC 284 E - Count Simple Paths ===");
        int nAtCoder = 3;
        int[][] edgesAtCoder = {{1, 2}, {2, 3}};
        int simplePaths = countSimplePaths(nAtCoder, edgesAtCoder);
        System.out.println("Nodes: " + nAtCoder + ", edges: " + Arrays.deepToString(edgesAtCoder)
+ ", simple paths: " + simplePaths);

        // 测试 USACO 2019 December Contest, Gold - Milk Visits
        System.out.println("
=== USACO 2019 December Contest, Gold - Milk Visits ===");
        int nUSACO = 4;
```

```
        int[] milkTypes = {1, 2, 1, 2};
        int[][] edgesUSACO = {{1, 2}, {2, 3}, {2, 4}};
        int[][] queriesUSACO = {{1, 3, 1}, {1, 4, 2}};
        boolean[] milkResults = milkVisits(nUSACO, milkTypes, edgesUSACO, queriesUSACO);
        System.out.println("Nodes: " + nUSACO + ", milk types: " + Arrays.toString(milkTypes));
        System.out.println("Queries: " + Arrays.deepToString(queriesUSACO) + ", results: " +
Arrays.toString(milkResults));

        // 测试高级哈希应用
        System.out.println("
=== 高级哈希应用测试 ===");

        // 测试完美哈希
        System.out.println("
=== 完美哈希 (Perfect Hashing) ===");
        String[] keys = {"apple", "banana", "orange", "grape", "pear"};
        PerfectHash perfectHash = new PerfectHash(keys);
        for (String key : keys) {
            System.out.println("Key '" + key + "' 的哈希值: " + perfectHash.get(key));
        }
        System.out.println("Key 'mango' 的哈希值: " + perfectHash.get("mango")); // 应该返回-1

        // 测试计数布隆过滤器
        System.out.println("
=== 计数布隆过滤器 (Counting Bloom Filter) ===");
        CountingBloomFilter countingBloomFilter = new CountingBloomFilter(10000, 7);
        countingBloomFilter.add("apple");
        countingBloomFilter.add("banana");
        countingBloomFilter.add("apple"); // 重复添加
        System.out.println("'apple' 计数: " + countingBloomFilter.count("apple")); // 应该为 2
        System.out.println("'banana' 计数: " + countingBloomFilter.count("banana")); // 应该为 1
        System.out.println("'orange' 计数: " + countingBloomFilter.count("orange")); // 应该为 0

        countingBloomFilter.remove("apple");
        System.out.println("移除一个 'apple' 后计数: " + countingBloomFilter.count("apple")); //
应该为 1

        // 测试可扩展哈希
        System.out.println("
=== 可扩展哈希 (Extendible Hashing) ===");
        ExtendibleHash extendibleHash = new ExtendibleHash(2); // 每个桶最多 2 个元素
        extendibleHash.put("key1", "value1");
        extendibleHash.put("key2", "value2");
```

```
        extendibleHash.put("key3", "value3"); // 这会触发桶分裂
        System.out.println("key1: " + extendibleHash.get("key1"));
        System.out.println("key2: " + extendibleHash.get("key2"));
        System.out.println("key3: " + extendibleHash.get("key3"));
        System.out.println("目录深度: " + extendibleHash.getDepth());

        // 测试线性哈希
        System.out.println("
=== 线性哈希 (Linear Hashing) ===");
        LinearHash linearHash = new LinearHash(4); // 初始大小为 4
        linearHash.put("key1", "value1");
        linearHash.put("key2", "value2");
        linearHash.put("key3", "value3");
        linearHash.put("key4", "value4");
        linearHash.put("key5", "value5"); // 这会触发分裂
        System.out.println("key1: " + linearHash.get("key1"));
        System.out.println("key2: " + linearHash.get("key2"));
        System.out.println("key3: " + linearHash.get("key3"));
        System.out.println("key4: " + linearHash.get("key4"));
        System.out.println("key5: " + linearHash.get("key5"));
        System.out.println("哈希表大小: " + linearHash.size());
    }
}
}
```

==========

文件: HashFunction.py

==========

```
# -*- coding: utf-8 -*-
"""
Python 版本的哈希函数实现
"""

import hashlib
import random
import collections
import heapq
from typing import List, Optional


class HashFunction:
    """
```

```
    哈希函数工具类
    """

    @staticmethod
    def hash_code(key, hash_size):
        """
        LintCode 128. Hash Function
        题目来源: https://www.lintcode.com/problem/hash-function/description

        题目描述:
        在数据结构中，哈希函数是用来将一个字符串（或任何其他类型）转化为小于哈希表大小且大于等于零的整数。
        一个好的哈希函数可以尽可能少地产生冲突。
        一种广泛使用的哈希函数算法是使用数值 33，假设任何字符串都是基于 33 的一个大整数，比如：
        hashcode("abcd") = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
                         = (97* 33^3 + 98 * 33^2 + 99 * 33 +100) % HASH_SIZE
                         = 3595978 % HASH_SIZE
        其中 HASH_SIZE 表示哈希表的大小(可以假设一个哈希表就是一个索引 0 ~ HASH_SIZE-1 的数组)。
        给出一个字符串作为 key 和一个哈希表的大小，返回这个字符串的哈希值。

        样例:
        对于 key="abcd" 并且 size=100， 返回 78

        算法思路:
        使用霍纳法则（Horner's Rule）优化计算，避免大数溢出:
        hashcode = (ascii(a) * 33^3 + ascii(b) * 33^2 + ascii(c) *33 + ascii(d)) % HASH_SIZE
        可以转换为:
        hashcode = ((((ascii(a) % HASH_SIZE) * 33 + ascii(b)) % HASH_SIZE) * 33 + ascii(c)) % HASH_SIZE) * 33 + ascii(d)) % HASH_SIZE

        时间复杂度: O(n)，其中 n 是字符串的长度
        空间复杂度: O(1)

        :param key: 输入字符串
        :param hash_size: 哈希表大小
        :return: 哈希值
        """
        ans = 0
        for char in key:
            ans = (ans * 33 + ord(char)) % hash_size
        return ans
```

```
    @staticmethod
    def generate_strings(arr, n):
        """
        生成所有可能的字符串组合

        :param arr: 字符数组
        :param n: 字符串长度
        :return: 所有可能的字符串列表
        """
        result = []

        def helper(current_path, remaining):
            if remaining == 0:
                result.append(current_path)
                return
            for char in arr:
                helper(current_path + char, remaining - 1)

        helper("", n)
        return result

    @staticmethod
    def my_hash_set():
        """
        LeetCode 705. Design HashSet (设计哈希集合)
        题目来源: https://leetcode.com/problems/design-hashset/

        题目描述:
        不使用任何内建的哈希表库设计一个哈希集合（HashSet）。
        实现 MyHashSet 类：
        void add(key) 向哈希集合中插入值 key 。
        bool contains(key) 返回哈希集合中是否存在这个值 key 。
        void remove(key) 将给定值 key 从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。

        示例:
        输入：
        ["MyHashSet", "add", "add", "contains", "contains", "add", "contains", "remove",
"contains"]
        [[], [1], [2], [1], [3], [2], [2], [2], [2]]
        输出：
        [null, null, null, true, false, null, true, null, false]

        约束条件:
```

```
    0 <= key <= 10^6
    最多调用 10^4 次 add、remove 和 contains

    算法思路:
    使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。
    当发生哈希冲突时，将元素添加到对应位置的链表中。

    时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）
    空间复杂度: O(n)，存储所有元素
    """
    class MyHashSet:
        def __init__(self):
            self.BASE = 10000
            self.data = [[] for _ in range(self.BASE)]

        def _hash(self, key):
            return key % self.BASE

        def add(self, key):
            h = self._hash(key)
            if key not in self.data[h]:
                self.data[h].append(key)

        def remove(self, key):
            h = self._hash(key)
            if key in self.data[h]:
                self.data[h].remove(key)

        def contains(self, key):
            h = self._hash(key)
            return key in self.data[h]

    return MyHashSet()

@staticmethod
def my_hash_map():
    """
    LeetCode 706. Design HashMap (设计哈希映射)
    题目来源: https://leetcode.com/problems/design-hashmap/

    题目描述:
    不使用任何内建的哈希表库设计一个哈希映射（HashMap）。
    实现 MyHashMap 类：
```

MyHashMap() 用空映射初始化对象

void put(int key, int value) 向 HashMap 插入一个键值对 (key, value) 。如果 key 已经存在于映射中，则更新其对应的值 value 。

int get(int key) 返回特定的 key 所映射的 value ；如果映射中不包含 key 的映射，返回 -1 。

void remove(key) 如果映射中存在 key 的映射，则移除 key 和它所对应的 value 。

示例:

输入：

["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]

[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]

输出：

[null, null, null, 1, -1, null, 1, null, -1]

约束条件:

0 <= key, value <= 10^6

最多调用 10^4 次 put、get 和 remove 方法

算法思路:

使用链地址法实现哈希表，创建一个固定大小的数组，每个数组元素是一个链表。

每个链表节点存储键值对，当发生哈希冲突时，将节点添加到对应位置的链表中。

时间复杂度: O(n/b)，其中 n 是元素个数，b 是桶数（在实际实现中我们使用 10000 作为桶数）

空间复杂度: O(n)，存储所有元素

"""

```
class MyHashMap:
    def __init__(self):
        self.BASE = 10000
        self.data = [[] for _ in range(self.BASE)]

    def _hash(self, key):
        return key % self.BASE

    def put(self, key, value):
        h = self._hash(key)
        for i, (k, v) in enumerate(self.data[h]):
            if k == key:
                self.data[h][i] = (key, value)
                return
        self.data[h].append((key, value))

    def get(self, key):
        h = self._hash(key)
        for k, v in self.data[h]:
```

```
                if k == key:
                    return v
            return -1

        def remove(self, key):
            h = self._hash(key)
            for i, (k, v) in enumerate(self.data[h]):
                if k == key:
                    del self.data[h][i]
                    return

    return MyHashMap()

@staticmethod
def str_str(haystack, needle):
    """
    LeetCode 28. Find the Index of the First Occurrence in a String (实现 strStr())
    题目来源: https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/

    题目描述:
    给你两个字符串 haystack 和 needle，请你在 haystack 字符串中找出 needle 字符串的第一个匹配项的下标（下标从 0 开始）。
    如果 needle 不是 haystack 的一部分，则返回 -1。

    示例:
    输入：haystack = "sadbutsad", needle = "sad"
    输出：0

    输入：haystack = "leetcode", needle = "leeto"
    输出：-1

    算法思路:
    使用 Rabin-Karp 算法（滚动哈希）实现字符串匹配：
    1. 计算 needle 的哈希值
    2. 在 haystack 中维护一个长度为 needle.length() 的滑动窗口，计算其哈希值
    3. 当哈希值相等时，再进行字符串比较确认（避免哈希冲突）

    时间复杂度: O(n+m)，其中 n 是 haystack 长度，m 是 needle 长度
    空间复杂度: O(1)
    """
    if not needle:
        return 0
```

```
        if len(haystack) < len(needle):
            return -1

        base = 256  # 基数
        mod = 1000000007  # 大质数，用于取模运算

        # 计算 needle 的哈希值和 h 的值
        needle_hash = 0
        h = 1  # 用于计算最高位的权重

        for i in range(len(needle)):
            needle_hash = (needle_hash * base + ord(needle[i])) % mod
            if i < len(needle) - 1:
                h = (h * base) % mod

        # 计算 haystack 第一个窗口的哈希值
        haystack_hash = 0
        for i in range(len(needle)):
            haystack_hash = (haystack_hash * base + ord(haystack[i])) % mod

        # 滑动窗口匹配
        for i in range(len(haystack) - len(needle) + 1):
            # 如果哈希值相等，再进行字符串比较确认
            if needle_hash == haystack_hash:
                if haystack[i:i+len(needle)] == needle:
                    return i

            # 计算下一个窗口的哈希值
            if i < len(haystack) - len(needle):
                haystack_hash = (base * (haystack_hash - (ord(haystack[i]) * h) % mod) +
ord(haystack[i + len(needle)])) % mod
                if haystack_hash < 0:
                    haystack_hash += mod

        return -1

    @staticmethod
    def find_repeated_dna_sequences(s):
        """
        LeetCode 187. Repeated DNA Sequences (重复的DNA序列)
        题目来源: https://leetcode.com/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)。

可以按任意顺序返回答案。

示例:

输入：s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

输出：["AAAAACCCCC","CCCCCAAAAA"]

输入：s = "AAAAAAAAAAAAA"

输出：["AAAAAAAAAA"]

算法思路:

使用滚动哈希技术：

1. 将每个字符映射为数字：A=0, C=1, G=2, T=3
2. 使用 4 进制表示长度为 10 的序列
3. 滑动窗口遍历所有长度为 10 的子串，计算其哈希值
4. 使用哈希表记录每个哈希值出现的次数
5. 返回出现次数大于 1 的序列

时间复杂度: O(n)，其中 n 是 DNA 序列长度

空间复杂度: O(n)，存储所有子串的哈希值

"""

```
if len(s) < 10:
    return []

# 字符到数字的映射
char_map = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

base = 4
mod = 1000000007  # 大质数，用于取模运算
window_size = 10

# 计算 base^(window_size-1) % mod
h = 1
for i in range(window_size - 1):
    h = (h * base) % mod

# 计算第一个窗口的哈希值
hash_val = 0
for i in range(window_size):
```

```
            hash_val = (hash_val * base + char_map[s[i]]) % mod

        # 使用哈希表记录每个哈希值出现的次数
        hash_map = {hash_val: 1}
        result = []

        # 滑动窗口计算后续哈希值
        for i in range(1, len(s) - window_size + 1):
            # 移除最高位字符，添加最低位字符
            hash_val = (base * (hash_val - (char_map[s[i - 1]] * h) % mod) + char_map[s[i +
window_size - 1]]) % mod
            if hash_val < 0:
                hash_val += mod

            # 记录哈希值出现次数
            hash_map[hash_val] = hash_map.get(hash_val, 0) + 1

            # 如果某个哈希值出现 2 次，将其对应的子串加入结果集
            if hash_map[hash_val] == 2:
                result.append(s[i:i + window_size])

        return result

    @staticmethod
    def shortest_palindrome(s):
        """
        LeetCode 214. Shortest Palindrome (最短回文串)
        题目来源: https://leetcode.com/problems/shortest-palindrome/

        题目描述:
        给你一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。
        找到并返回可以用这种方式转换的最短回文串。

        示例:
        输入：s = "aacecaaa"
        输出："aaacecaaa"

        输入：s = "abcd"
        输出："dcbabcd"

        算法思路:
        使用滚动哈希技术找到 s 的最长前缀回文串:
        1. 计算 s 的正向哈希和反向哈希
```

```
        2. 使用双指针从两端向中间移动，同时比较正向和反向哈希
        3. 当找到最长前缀回文串后，将剩余部分反转并添加到原字符串前面

        时间复杂度: O(n)
        空间复杂度: O(n)
        """
        if len(s) <= 1:
            return s

        n = len(s)
        base = 256
        mod = 10**9 + 7
        forward_hash = 0
        backward_hash = 0
        power = 1
        max_len = 0

        for i in range(n):
            forward_hash = (forward_hash * base + ord(s[i])) % mod
            backward_hash = (backward_hash + ord(s[i]) * power) % mod
            if forward_hash == backward_hash:
                max_len = i + 1
            power = (power * base) % mod

        # 将剩余部分反转并添加到前面
        suffix = s[max_len:]
        reversed_suffix = suffix[::-1]
        return reversed_suffix + s


# 一致性哈希 (Consistent Hashing) 实现
class ConsistentHash:
    def __init__(self, replicas=100):
        self.replicas = replicas  # 每个真实节点对应的虚拟节点数
        self.ring = {}  # 虚拟节点环
        self.servers = set()  # 真实服务器集合

    def _get_hash(self, key):
        """使用 FNV-1a 哈希算法"""
        FNV_32_INIT = 0x811c9dc5
        FNV_32_PRIME = 0x01000193

        hash_val = FNV_32_INIT
```

```
        for char in key:
            hash_val ^= ord(char)
            hash_val *= FNV_32_PRIME
            hash_val &= 0xFFFFFFFF  # 保持32位
        return abs(hash_val)

    def add_server(self, server):
        """添加服务器"""
        self.servers.add(server)
        # 为每个真实节点创建多个虚拟节点
        for i in range(self.replicas):
            virtual_node = f"{server}#{i}"
            hash_val = self._get_hash(virtual_node)
            self.ring[hash_val] = server

    def remove_server(self, server):
        """移除服务器"""
        if server not in self.servers:
            return

        self.servers.remove(server)
        # 移除对应的所有虚拟节点
        keys_to_remove = [key for key, value in self.ring.items() if value == server]
        for key in keys_to_remove:
            del self.ring[key]

    def get_server(self, key):
        """获取键对应的服务器"""
        if not self.ring:
            return None

        hash_val = self._get_hash(key)
        # 找到顺时针方向的第一个服务器
        sorted_keys = sorted(self.ring.keys())
        # 使用二分查找找到第一个大于等于hash_val的键
        import bisect
        index = bisect.bisect_left(sorted_keys, hash_val)
        # 如果没有找到，则使用第一个键
        if index == len(sorted_keys):
            index = 0

        return self.ring[sorted_keys[index]]
```

```
    def get_servers(self):
        """获取服务器列表"""
        return list(self.servers)

# 布隆过滤器 (Bloom Filter) 实现
class BloomFilter:
    def __init__(self, size=10000, hash_functions=7):
        self.size = size
        self.bits = [False] * size
        self.hash_functions = hash_functions
        # 初始化哈希函数种子
        self.seeds = [i * 100 + 31 for i in range(hash_functions)]

    def _get_hash(self, element, seed):
        """哈希函数"""
        hash_val = 0
        for char in element:
            hash_val = seed * hash_val + ord(char)
        return abs(hash_val % self.size)

    def add(self, element):
        """添加元素"""
        for seed in self.seeds:
            hash_val = self._get_hash(element, seed)
            self.bits[hash_val] = True

    def might_contain(self, element):
        """判断元素是否可能存在"""
        for seed in self.seeds:
            hash_val = self._get_hash(element, seed)
            if not self.bits[hash_val]:
                return False  # 只要有一个位置为0，元素一定不存在
        return True  # 所有位置都为1，元素可能存在

# 双重哈希 (Double Hashing) 实现的哈希表
class DoubleHashTable:
    DEFAULT_SIZE = 16
    LOAD_FACTOR = 0.75

    def __init__(self):
        self.size = self.DEFAULT_SIZE
        self.keys = [None] * self.size
        self.values = [None] * self.size
```

```
        self.occupied = [False] * self.size
        self.count = 0

    def _hash1(self, key):
        """第一个哈希函数"""
        return abs(hash(key) % self.size)

    def _hash2(self, key):
        """第二个哈希函数，用于计算步长"""
        return 1 + abs(hash(key) % (self.size - 1))

    def _find_insertion_index(self, key):
        """查找插入位置"""
        h1 = self._hash1(key)
        h2 = self._hash2(key)
        index = h1
        step = 1

        # 查找空位置或相同的键
        while self.occupied[index]:
            if self.keys[index] == key:
                return index  # 键已存在，返回该位置以更新值
            index = (h1 + step * h2) % self.size
            step += 1

        return index

    def _find_index(self, key):
        """查找键的索引"""
        h1 = self._hash1(key)
        h2 = self._hash2(key)
        index = h1
        step = 1

        # 查找键
        while self.occupied[index]:
            if self.keys[index] == key:
                return index  # 找到键
            index = (h1 + step * h2) % self.size
            step += 1
            # 避免无限循环
            if step > self.size:
                break
```

```
        return -1  # 未找到键

    def _rehash(self):
        """扩容"""
        old_keys = self.keys.copy()
        old_values = self.values.copy()
        old_occupied = self.occupied.copy()

        # 扩大为原来的两倍
        self.size *= 2
        self.keys = [None] * self.size
        self.values = [None] * self.size
        self.occupied = [False] * self.size
        self.count = 0

        # 重新插入所有键值对
        for i in range(len(old_occupied)):
            if old_occupied[i]:
                self.put(old_keys[i], old_values[i])

    def put(self, key, value):
        """插入键值对"""
        # 检查是否需要扩容
        if self.count / self.size >= self.LOAD_FACTOR:
            self._rehash()

        index = self._find_insertion_index(key)
        self.keys[index] = key
        self.values[index] = value
        if not self.occupied[index]:
            self.occupied[index] = True
            self.count += 1

    def get(self, key):
        """获取值"""
        index = self._find_index(key)
        return self.values[index] if index != -1 else None

    def remove(self, key):
        """删除键值对"""
        index = self._find_index(key)
        if index != -1:
```

```
            self.occupied[index] = False
            self.count -= 1

    def get_size(self):
        """获取大小"""
        return self.count

# 更多LeetCode哈希相关题目实现

def two_sum(nums: List[int], target: int) -> List[int]:
    """
    LeetCode 1. Two Sum (两数之和)
    题目来源: https://leetcode.com/problems/two-sum/

    题目描述:
    给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。
    你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。
    你可以按任意顺序返回答案。

    示例:
    输入：nums = [2,7,11,15], target = 9
    输出：[0,1]
    解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。

    算法思路:
    使用哈希表存储每个数字及其对应的索引，遍历数组时检查target - nums[i]是否在哈希表中

    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    num_map = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map:
            return [num_map[complement], i]
        num_map[num] = i
    return [-1, -1]

def group_anagrams(strs: List[str]) -> List[List[str]]:
    """
    LeetCode 49. Group Anagrams (字母异位词分组)
    题目来源: https://leetcode.com/problems/group-anagrams/
```

```
    题目描述:
    给你一个字符串数组，请你将字母异位词组合在一起。可以按任意顺序返回结果列表。
    字母异位词是由重新排列源单词的所有字母得到的一个新单词。

    示例:
    输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
    输出: [["bat"],["nat","tan"],["ate","eat","tea"]]

    算法思路:
    使用排序后的字符串作为哈希表的键，将具有相同排序字符串的单词分组

    时间复杂度: O(n * k log k)，其中 n 是字符串数量，k 是字符串最大长度
    空间复杂度: O(n * k)
    """
    anagram_map = {}
    for s in strs:
        sorted_s = ''.join(sorted(s))
        if sorted_s not in anagram_map:
            anagram_map[sorted_s] = []
        anagram_map[sorted_s].append(s)
    return list(anagram_map.values())

def is_anagram(s: str, t: str) -> bool:
    """
    LeetCode 242. Valid Anagram (有效的字母异位词)
    题目来源: https://leetcode.com/problems/valid-anagram/

    题目描述:
    给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
    注意：若 s 和 t 中每个字符出现的次数都相同，则称 s 和 t 互为字母异位词。

    示例:
    输入: s = "anagram", t = "nagaram"
    输出: true

    算法思路:
    使用哈希表统计每个字符出现的次数，然后比较两个字符串的字符频率

    时间复杂度: O(n)
    空间复杂度: O(1)，因为字符集大小固定为 26
    """
    if len(s) != len(t):
```

```
        return False

    count = [0] * 26
    for char in s:
        count[ord(char) - ord('a')] += 1
    for char in t:
        count[ord(char) - ord('a')] -= 1
        if count[ord(char) - ord('a')] < 0:
            return False
    return True

def length_of_longest_substring(s: str) -> int:
    """
    LeetCode 3. Longest Substring Without Repeating Characters (无重复字符的最长子串)
    题目来源: https://leetcode.com/problems/longest-substring-without-repeating-characters/

    题目描述:
    给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。

    示例:
    输入: s = "abcabcbb"
    输出: 3
    解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

    算法思路:
    使用滑动窗口和哈希表记录字符最后出现的位置
    当遇到重复字符时，移动窗口左边界到重复字符的下一个位置

    时间复杂度: O(n)
    空间复杂度: O(min(m, n))，其中 m 是字符集大小
    """
    char_map = {}
    max_length = 0
    left = 0

    for right, char in enumerate(s):
        if char in char_map and char_map[char] >= left:
            left = char_map[char] + 1
        char_map[char] = right
        max_length = max(max_length, right - left + 1)

    return max_length
```

```
def min_window(s: str, t: str) -> str:
    """
    LeetCode 76. Minimum Window Substring (最小覆盖子串)
    题目来源: https://leetcode.com/problems/minimum-window-substring/

    题目描述:
    给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

    示例:
    输入：s = "ADOBECODEBANC", t = "ABC"
    输出："BANC"
    解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

    算法思路:
    使用滑动窗口和哈希表统计字符频率
    维护一个计数器记录还需要匹配的字符数量

    时间复杂度: O(m + n)
    空间复杂度: O(m + n)
    """
    if len(s) < len(t):
        return ""

    target = collections.Counter(t)
    window = {}
    left = 0
    right = 0
    required = len(target)
    formed = 0
    min_length = float('inf')
    min_left = 0
    min_right = 0

    while right < len(s):
        char = s[right]
        window[char] = window.get(char, 0) + 1

        if char in target and window[char] == target[char]:
            formed += 1

        while left <= right and formed == required:
            char = s[left]
```

```
            if right - left + 1 < min_length:
                min_length = right - left + 1
                min_left = left
                min_right = right

            window[char] -= 1
            if char in target and window[char] < target[char]:
                formed -= 1
            left += 1

        right += 1

    return "" if min_length == float('inf') else s[min_left:min_right+1]

def subarray_sum(nums: List[int], k: int) -> int:
    """
    LeetCode 560. Subarray Sum Equals K (和为K的子数组)
    题目来源: https://leetcode.com/problems/subarray-sum-equals-k/

    题目描述:
    给你一个整数数组 nums 和一个整数 k ，请你统计并返回该数组中和为 k 的连续子数组的个数。

    示例:
    输入：nums = [1,1,1], k = 2
    输出：2

    算法思路:
    使用前缀和和哈希表，记录每个前缀和出现的次数
    当prefixSum - k在哈希表中存在时，说明存在和为k的子数组

    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    prefix_sum_count = {0: 1}
    prefix_sum = 0
    count = 0

    for num in nums:
        prefix_sum += num
        if prefix_sum - k in prefix_sum_count:
            count += prefix_sum_count[prefix_sum - k]
        prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1
```

```
    return count

def top_k_frequent(nums: List[int], k: int) -> List[int]:
    """
    LeetCode 347. Top K Frequent Elements (前 K 个高频元素)
    题目来源: https://leetcode.com/problems/top-k-frequent-elements/

    题目描述:
    给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按任意顺序返回答案。

    示例:
    输入: nums = [1,1,1,2,2,3], k = 2
    输出: [1,2]

    算法思路:
    使用哈希表统计频率，然后使用桶排序或优先队列找出前 k 个高频元素

    时间复杂度: O(n log k)
    空间复杂度: O(n)
    """
    frequency_map = collections.Counter(nums)

    # 使用桶排序
    buckets = [[] for _ in range(len(nums) + 1)]
    for num, freq in frequency_map.items():
        buckets[freq].append(num)

    result = []
    for i in range(len(buckets) - 1, 0, -1):
        if buckets[i]:
            result.extend(buckets[i])
        if len(result) >= k:
            break

    return result[:k]

class RandomizedSet:
    """
    LeetCode 380. Insert Delete GetRandom O(1) (常数时间插入、删除和获取随机元素)
    题目来源: https://leetcode.com/problems/insert-delete-getrandom-o1/
```

题目描述:

实现 RandomizedSet 类：

RandomizedSet() 初始化 RandomizedSet 对象

bool insert(int val) 当元素 val 不存在时，向集合中插入该项，并返回 true ；否则，返回 false 。

bool remove(int val) 当元素 val 存在时，从集合中移除该项，并返回 true ；否则，返回 false 。

int getRandom() 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有相同的概率被返回。

算法思路:

使用哈希表存储值和索引的映射，使用动态数组存储值

删除时将要删除的元素与最后一个元素交换，然后删除最后一个元素

时间复杂度: O(1) 平均时间复杂度

空间复杂度: O(n)

"""

```
def __init__(self):
    self.value_to_index = {}
    self.values = []

def insert(self, val: int) -> bool:
    if val in self.value_to_index:
        return False
    self.value_to_index[val] = len(self.values)
    self.values.append(val)
    return True

def remove(self, val: int) -> bool:
    if val not in self.value_to_index:
        return False

    index = self.value_to_index[val]
    last_element = self.values[-1]

    # 将要删除的元素与最后一个元素交换
    self.values[index] = last_element
    self.value_to_index[last_element] = index

    # 删除最后一个元素
    self.values.pop()
    del self.value_to_index[val]

    return True
```

```
    def get_random(self) -> int:
        return random.choice(self.values)

class LRUCache:
    """
    LeetCode 146. LRU Cache (LRU缓存)
    题目来源: https://leetcode.com/problems/lru-cache/

    题目描述:
    请你设计并实现一个满足 LRU (最近最少使用) 缓存约束的数据结构。
    实现 LRUCache 类：
    LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
    int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
    void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。
    如果插入操作导致关键字数量超过 capacity ，则应该逐出最久未使用的关键字。

    算法思路:
    使用哈希表+双向链表实现
    哈希表提供O(1)的查找，双向链表维护访问顺序

    时间复杂度: O(1)
    空间复杂度: O(capacity)
    """
    class DLinkedNode:
        def __init__(self, key=0, value=0):
            self.key = key
            self.value = value
            self.prev = None
            self.next = None

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.size = 0
        self.cache = {}
        self.head = self.DLinkedNode()
        self.tail = self.DLinkedNode()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, node):
        """在头部添加节点"""
        node.prev = self.head
```

```
        node.next = self.head.next
        if self.head.next:
            self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        """移除节点"""
        if node.prev:
            node.prev.next = node.next
        if node.next:
            node.next.prev = node.prev

    def _move_to_head(self, node):
        """移动节点到头部"""
        self._remove_node(node)
        self._add_node(node)

    def _pop_tail(self):
        """弹出尾部节点"""
        if self.tail.prev != self.head:
            node = self.tail.prev
            self._remove_node(node)
            return node
        return None

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        node = self.cache[key]
        self._move_to_head(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            node = self.cache[key]
            node.value = value
            self._move_to_head(node)
        else:
            new_node = self.DLinkedNode(key, value)
            self.cache[key] = new_node
            self._add_node(new_node)
            self.size += 1
```

```
            if self.size > self.capacity:
                tail = self._pop_tail()
                if tail:
                    del self.cache[tail.key]
                    self.size -= 1

def main():
    print("=== LintCode 128. Hash Function ===")
    key = "abcd"
    hash_size = 100
    result = HashFunction.hash_code(key, hash_size)
    print(f"Key: {key}, HASH_SIZE: {hash_size}, Result: {result}")
    print()

    # 测试 MD5 哈希
    print("=== MD5 Hash Examples ===")
    str1 = "zuochengyunzuochengyunzuochengyun1"
    str2 = "zuochengyunzuochengyunzuochengyun2"
    str3 = "zuochengyunzuochengyunzuochengyun3"

    print(f"String 1 MD5: {hashlib.md5(str1.encode()).hexdigest()}")
    print(f"String 2 MD5: {hashlib.md5(str2.encode()).hexdigest()}")
    print(f"String 3 MD5: {hashlib.md5(str3.encode()).hexdigest()}")
    print()

    # 测试生成字符串
    arr = ['a', 'b']
    n = 5  # 使用较小的 n 值避免输出过多
    print(f"生成长度为{n}，字符来自 arr，所有可能的字符串:")
    strs = HashFunction.generate_strings(arr, n)
    print(f"不同字符串的数量 : {len(strs)}")

    hash_set = set()
    for s in strs:
        hash_set.add(hashlib.md5(s.encode()).hexdigest())
    print(f"不同哈希值的数量 : {len(hash_set)}")
    print()

    # 测试哈希分布
    m = 7
    cnts = [0] * m
    print(f"现在看看这些哈希值，% {m} 之后的余数分布情况")
    for hash_code in hash_set:
```

```
    # 简化处理，取前几位字符的 ASCII 值求模
    total = sum(ord(c) for c in hash_code[:4])
    remainder = total % m
    cnts[remainder] += 1
for i in range(m):
    print(f"余数 {i} 出现了 {cnts[i]} 次")

# 测试 LeetCode 705. Design HashSet
print("\n=== LeetCode 705. Design HashSet ===")
my_hash_set = HashFunction.my_hash_set()
my_hash_set.add(1)      # set = [1]
my_hash_set.add(2)      # set = [1, 2]
print(my_hash_set.contains(1))  # 返回 True
print(my_hash_set.contains(3))  # 返回 False （未找到）
my_hash_set.add(2)      # set = [1, 2]
print(my_hash_set.contains(2))  # 返回 True
my_hash_set.remove(2)   # set = [1]
print(my_hash_set.contains(2))  # 返回 False （已移除）

# 测试 LeetCode 706. Design HashMap
print("\n=== LeetCode 706. Design HashMap ===")
my_hash_map = HashFunction.my_hash_map()
my_hash_map.put(1, 1)  # myHashMap 现在为 [[1,1]]
my_hash_map.put(2, 2)  # myHashMap 现在为 [[1,1], [2,2]]
print(my_hash_map.get(1))     # 返回 1 ，myHashMap 现在为 [[1,1], [2,2]]
print(my_hash_map.get(3))     # 返回 -1（未找到），myHashMap 现在为 [[1,1], [2,2]]
my_hash_map.put(2, 1)  # myHashMap 现在为 [[1,1], [2,1]]（更新已有的值）
print(my_hash_map.get(2))     # 返回 1 ，myHashMap 现在为 [[1,1], [2,1]]
my_hash_map.remove(2)  # 删除键为 2 的数据，myHashMap 现在为 [[1,1]]
print(my_hash_map.get(2))     # 返回 -1（未找到），myHashMap 现在为 [[1,1]]

# 测试 LeetCode 28. Find the Index of the First Occurrence in a String
print("\n=== LeetCode 28. Find the Index of the First Occurrence in a String ===")
print(HashFunction.str_str("sadbutsad", "sad"))   # 返回 0
print(HashFunction.str_str("leetcode", "leeto"))  # 返回 -1

# 测试 LeetCode 187. Repeated DNA Sequences
print("\n=== LeetCode 187. Repeated DNA Sequences ===")
dna_result = HashFunction.find_repeated_dna_sequences("AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT")
print(dna_result)  # ["AAAAACCCCC","CCCCCAAAAA"]

# 测试 LeetCode 214 题
print("\n=== LeetCode 214. Shortest Palindrome ===")
```

```
print(f"shortest_palindrome('aacecaaa'): {HashFunction.shortest_palindrome('aacecaaa')}")
print(f"shortest_palindrome('abcd'): {HashFunction.shortest_palindrome('abcd')}")
print()

# 测试一致性哈希
print("=== 一致性哈希 (Consistent Hashing) ===")
consistent_hash = ConsistentHash(replicas=100)
consistent_hash.add_server("Server1")
consistent_hash.add_server("Server2")
consistent_hash.add_server("Server3")

print(f"键 'user1' 分配到的服务器: {consistent_hash.get_server('user1')}")
print(f"键 'user2' 分配到的服务器: {consistent_hash.get_server('user2')}")
print(f"键 'user3' 分配到的服务器: {consistent_hash.get_server('user3')}")

# 移除一个服务器后，观察键的重新分配情况
print("\n移除 Server2 后:")
consistent_hash.remove_server("Server2")
print(f"键 'user1' 分配到的服务器: {consistent_hash.get_server('user1')}")
print(f"键 'user2' 分配到的服务器: {consistent_hash.get_server('user2')}")
print(f"键 'user3' 分配到的服务器: {consistent_hash.get_server('user3')}")
print()

# 测试布隆过滤器
print("=== 布隆过滤器 (Bloom Filter) ===")
bloom_filter = BloomFilter(size=10000, hash_functions=7)
bloom_filter.add("apple")
bloom_filter.add("banana")
bloom_filter.add("orange")

print(f"'apple' 可能在集合中: {bloom_filter.might_contain('apple')}")
print(f"'banana' 可能在集合中: {bloom_filter.might_contain('banana')}")
print(f"'orange' 可能在集合中: {bloom_filter.might_contain('orange')}")
print(f"'pear' 可能在集合中: {bloom_filter.might_contain('pear')}")
print(f"'grape' 可能在集合中: {bloom_filter.might_contain('grape')}")
print()

# 测试双重哈希表
print("=== 双重哈希 (Double Hashing) ===")
double_hash_table = DoubleHashTable()
double_hash_table.put("apple", 100)
double_hash_table.put("banana", 200)
double_hash_table.put("orange", 300)
```

```
print(f"'apple' 的值: {double_hash_table.get('apple')}")
print(f"'banana' 的值: {double_hash_table.get('banana')}")
print(f"'orange' 的值: {double_hash_table.get('orange')}")
print(f"'pear' 的值: {double_hash_table.get('pear')}")

double_hash_table.remove("banana")
print(f"移除 'banana' 后的值: {double_hash_table.get('banana')}")
print(f"哈希表大小: {double_hash_table.get_size()}")

# 测试更多 LeetCode 哈希相关题目
print("\n=== 更多 LeetCode 哈希相关题目测试 ===")

# 测试 LeetCode 1. Two Sum
print("\n=== LeetCode 1. Two Sum ===")
nums = [2, 7, 11, 15]
target = 9
result = two_sum(nums, target)
print(f"nums: {nums}, target: {target}, result: {result}")

# 测试 LeetCode 49. Group Anagrams
print("\n=== LeetCode 49. Group Anagrams ===")
strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
result = group_anagrams(strs)
print(f"strs: {strs}")
print(f"grouped anagrams: {result}")

# 测试 LeetCode 242. Valid Anagram
print("\n=== LeetCode 242. Valid Anagram ===")
s1, s2 = "anagram", "nagaram"
result = is_anagram(s1, s2)
print(f"'{s1}' and '{s2}' are anagrams: {result}")

# 测试 LeetCode 3. Longest Substring Without Repeating Characters
print("\n=== LeetCode 3. Longest Substring Without Repeating Characters ===")
s = "abcabcbb"
result = length_of_longest_substring(s)
print(f"String: '{s}', longest substring length: {result}")

# 测试 LeetCode 76. Minimum Window Substring
print("\n=== LeetCode 76. Minimum Window Substring ===")
s = "ADOBECODEBANC"
t = "ABC"
```

```
    result = min_window(s, t)
    print(f"s: '{s}', t: '{t}', min window: '{result}'")

    # 测试 LeetCode 560. Subarray Sum Equals K
    print("\n=== LeetCode 560. Subarray Sum Equals K ===")
    nums = [1, 1, 1]
    k = 2
    result = subarray_sum(nums, k)
    print(f"nums: {nums}, k: {k}, subarray count: {result}")

    # 测试 LeetCode 347. Top K Frequent Elements
    print("\n=== LeetCode 347. Top K Frequent Elements ===")
    nums = [1, 1, 1, 2, 2, 3]
    k = 2
    result = top_k_frequent(nums, k)
    print(f"nums: {nums}, k: {k}, top k frequent: {result}")

    # 测试 LeetCode 380. Insert Delete GetRandom O(1)
    print("\n=== LeetCode 380. Insert Delete GetRandom O(1) ===")
    randomized_set = RandomizedSet()
    print(f"Insert 1: {randomized_set.insert(1)}")
    print(f"Insert 2: {randomized_set.insert(2)}")
    print(f"Insert 1 again: {randomized_set.insert(1)}")
    print(f"Get random: {randomized_set.get_random()}")
    print(f"Remove 2: {randomized_set.remove(2)}")
    print(f"Remove 3: {randomized_set.remove(3)}")
    print(f"Get random: {randomized_set.get_random()}")

    # 测试 LeetCode 146. LRU Cache
    print("\n=== LeetCode 146. LRU Cache ===")
    lru_cache = LRUCache(2)
    lru_cache.put(1, 1)
    lru_cache.put(2, 2)
    print(f"Get 1: {lru_cache.get(1)}")
    lru_cache.put(3, 3)  # 这会使得键 2 被移除
    print(f"Get 2: {lru_cache.get(2)}")
    print(f"Get 3: {lru_cache.get(3)}")


if __name__ == "__main__":
    main()

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

文件: HashFunctionExtension.cpp

```
==========================================
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <algorithm>

using namespace std;

/**
 * LeetCode 36. Valid Sudoku (有效的数独)
 * 题目来源: https://leetcode.com/problems/valid-sudoku/
 *
 * 题目描述:
 * 请你判断一个 9 x 9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
 * 1. 数字 1-9 在每一行只能出现一次。
 * 2. 数字 1-9 在每一列只能出现一次。
 * 3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
 *
 * 注意:
 * - 一个有效的数独（部分已填充）不一定是可解的。
 * - 只需要根据以上规则，验证已经填入的数字是否有效即可。
 * - 空白格用 '.' 表示。
 *
 * 算法思路:
 * 使用哈希表记录每行、每列、每个小方块中已经出现的数字
 *
 * 时间复杂度: O(1)，因为数独大小固定
 * 空间复杂度: O(1)
 */
bool isValidSudoku(vector<vector<char>>& board) {
    vector<unordered_set<char>> rows(9);
    vector<unordered_set<char>> cols(9);
    vector<unordered_set<char>> boxes(9);

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            char num = board[i][j];
            if (num == '.') {
                continue;
```

```
            }

            // 计算小方块的索引
            int boxIndex = (i / 3) * 3 + j / 3;

            // 检查当前数字是否已经在行、列或小方块中出现
            if (rows[i].count(num) || cols[j].count(num) || boxes[boxIndex].count(num)) {
                return false;
            }

            // 将当前数字添加到对应的哈希表中
            rows[i].insert(num);
            cols[j].insert(num);
            boxes[boxIndex].insert(num);
        }
    }

    return true;
}

/**
 * LeetCode 454. 4Sum II (四数相加 II)
 * 题目来源: https://leetcode.com/problems/4sum-ii/
 *
 * 题目描述:
 * 给你四个整数数组 nums1、nums2、nums3 和 nums4 ，数组长度都是 n ，请你计算有多少个元组 (i, j, k, l) 能满足：
 * 0 <= i, j, k, l < n
 * nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0
 *
 * 示例:
 * 输入: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
 * 输出: 2
 * 解释: 两个元组如下:
 * 1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0
 * 2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0
 *
 * 算法思路:
 * 将四个数组分成两部分，计算前两个数组所有可能的和及其出现次数，
 * 然后计算后两个数组的所有可能的和，检查其相反数在前两个数组的和中出现的次数
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n^2)
```

```
 */
int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4)
{
    unordered_map<int, int> sumCount;

    for (int num1 : nums1) {
        for (int num2 : nums2) {
            int sum = num1 + num2;
            sumCount[sum]++;
        }
    }

    int count = 0;

    for (int num3 : nums3) {
        for (int num4 : nums4) {
            int sum = num3 + num4;
            auto it = sumCount.find(-sum);
            if (it != sumCount.end()) {
                count += it->second;
            }
        }
    }

    return count;
}

/**
 * LeetCode 525. Contiguous Array (连续数组)
 * 题目来源: https://leetcode.com/problems/contiguous-array/
 *
 * 题目描述:
 * 给定一个二进制数组 nums , 找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。
 *
 * 示例:
 * 输入: nums = [0,1]
 * 输出: 2
 * 解释: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。
 *
 * 输入: nums = [0,1,0]
 * 输出: 2
 * 解释: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。
 *
```

```
 * 算法思路:
 * 将0视为-1，1视为1，问题转化为求和为0的最长子数组
 * 使用哈希表记录前缀和及其首次出现的位置
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
int findMaxLength(vector<int>& nums) {
    unordered_map<int, int> map;
    map[0] = -1; // 初始和为0，位置为-1
    int maxLength = 0;
    int count = 0;

    for (int i = 0; i < nums.size(); i++) {
        // 将0视为-1，1视为1
        count += nums[i] == 0 ? -1 : 1;

        if (map.find(count) != map.end()) {
            maxLength = max(maxLength, i - map[count]);
        } else {
            map[count] = i;
        }
    }

    return maxLength;
}

/**
 * LeetCode 692. Top K Frequent Words (前K个高频单词)
 * 题目来源: https://leetcode.com/problems/top-k-frequent-words/
 *
 * 题目描述:
 * 给你一个单词数组 words 和一个整数 k ，请你返回前 k 个出现次数最多的单词。
 * 返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。
 *
 * 示例:
 * 输入: words = ["i","love","leetcode","i","love","coding"], k = 2
 * 输出: ["i","love"]
 * 解释: "i" 和 "love" 是出现次数最多的两个单词，均为2次。
 * 注意，按字母顺序 "i" 在 "love" 之前。
 *
 * 算法思路:
 * 使用哈希表统计每个单词的频率，然后使用优先队列按频率和字典序排序
```

```
 *
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(n)
 */
vector<string> topKFrequent(vector<string>& words, int k) {
    unordered_map<string, int> wordCount;
    for (const string& word : words) {
        wordCount[word]++;
    }

    auto compare = [&](const string& a, const string& b) {
        if (wordCount[a] != wordCount[b]) {
            return wordCount[a] > wordCount[b];
        }
        return a < b;
    };

    vector<string> uniqueWords;
    for (const auto& pair : wordCount) {
        uniqueWords.push_back(pair.first);
    }

    sort(uniqueWords.begin(), uniqueWords.end(), compare);

    vector<string> result(uniqueWords.begin(), uniqueWords.begin() + k);
    return result;
}

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

文件: HashFunctionExtension.java
==========================================
package class106;

import java.util.*;

public class HashFunctionExtension {
    /**
     * LeetCode 36. Valid Sudoku (有效的数独)
     * 题目来源: https://leetcode.com/problems/valid-sudoku/
     *
     * 题目描述:
     * 请你判断一个 9 x 9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
```

```
     * 1. 数字 1-9 在每一行只能出现一次。
     * 2. 数字 1-9 在每一列只能出现一次。
     * 3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
     *
     * 注意:
     * - 一个有效的数独（部分已填充）不一定是可解的。
     * - 只需要根据以上规则，验证已经填入的数字是否有效即可。
     * - 空白格用 '.' 表示。
     *
     * 算法思路:
     * 使用哈希表记录每行、每列、每个小方块中已经出现的数字
     *
     * 时间复杂度: O(1)，因为数独大小固定
     * 空间复杂度: O(1)
     */
    public boolean isValidSudoku(char[][] board) {
        // 创建 3 个哈希表数组，分别存储每行、每列和每个小方块中出现的数字
        HashSet<Character>[] rows = new HashSet[9];
        HashSet<Character>[] cols = new HashSet[9];
        HashSet<Character>[] boxes = new HashSet[9];

        for (int i = 0; i < 9; i++) {
            rows[i] = new HashSet<>();
            cols[i] = new HashSet<>();
            boxes[i] = new HashSet<>();
        }

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                char num = board[i][j];
                if (num == '.') {
                    continue;
                }

                // 计算小方块的索引
                int boxIndex = (i / 3) * 3 + j / 3;

                // 检查当前数字是否已经在行、列或小方块中出现
                if (rows[i].contains(num) || cols[j].contains(num) ||
boxes[boxIndex].contains(num)) {
                    return false;
                }
```

```
                // 将当前数字添加到对应的哈希表中
                rows[i].add(num);
                cols[j].add(num);
                boxes[boxIndex].add(num);
            }
        }

        return true;
    }

    /**
     * LeetCode 454. 4Sum II (四数相加 II)
     * 题目来源: https://leetcode.com/problems/4sum-ii/
     *
     * 题目描述:
     * 给你四个整数数组 nums1、nums2、nums3 和 nums4 ，数组长度都是 n ，请你计算有多少个元组 (i, j, k, l) 能满足：
     * 0 <= i, j, k, l < n
     * nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0
     *
     * 示例:
     * 输入: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
     * 输出: 2
     * 解释: 两个元组如下:
     * 1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0
     * 2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0
     *
     * 算法思路:
     * 将四个数组分成两部分，计算前两个数组所有可能的和及其出现次数，
     * 然后计算后两个数组的所有可能的和，检查其相反数在前两个数组的和中出现的次数
     *
     * 时间复杂度: O(n^2)
     * 空间复杂度: O(n^2)
     */
    public int fourSumCount(int[] nums1, int[] nums2, int[] nums3, int[] nums4) {
        // 使用哈希表存储前两个数组的和及其出现次数
        Map<Integer, Integer> sumCount = new HashMap<>();

        for (int num1 : nums1) {
            for (int num2 : nums2) {
                int sum = num1 + num2;
                sumCount.put(sum, sumCount.getOrDefault(sum, 0) + 1);
            }
```

```
        }

        int count = 0;

        for (int num3 : nums3) {
            for (int num4 : nums4) {
                int sum = num3 + num4;
                count += sumCount.getOrDefault(-sum, 0);
            }
        }

        return count;
    }

    /**
     * LeetCode 525. Contiguous Array (连续数组)
     * 题目来源: https://leetcode.com/problems/contiguous-array/
     *
     * 题目描述:
     * 给定一个二进制数组 nums , 找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。
     *
     * 示例:
     * 输入: nums = [0,1]
     * 输出: 2
     * 解释: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。
     *
     * 输入: nums = [0,1,0]
     * 输出: 2
     * 解释: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。
     *
     * 算法思路:
     * 将 0 视为-1，1 视为 1，问题转化为求和为 0 的最长子数组
     * 使用哈希表记录前缀和及其首次出现的位置
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    public int findMaxLength(int[] nums) {
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, -1); // 初始和为 0，位置为-1
        int maxLength = 0;
        int count = 0;
```

```
    for (int i = 0; i < nums.length; i++) {
        // 将0视为-1，1视为1
        count += nums[i] == 0 ? -1 : 1;

        if (map.containsKey(count)) {
            maxLength = Math.max(maxLength, i - map.get(count));
        } else {
            map.put(count, i);
        }
    }

    return maxLength;
}

/**
 * LeetCode 692. Top K Frequent Words (前K个高频单词)
 * 题目来源: https://leetcode.com/problems/top-k-frequent-words/
 *
 * 题目描述:
 * 给你一个单词数组 words 和一个整数 k ，请你返回前 k 个出现次数最多的单词。
 * 返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。
 *
 * 示例:
 * 输入: words = ["i","love","leetcode","i","love","coding"], k = 2
 * 输出: ["i","love"]
 * 解释: "i" 和 "love" 是出现次数最多的两个单词，均为2次。
 * 注意，按字母顺序 "i" 在 "love" 之前。
 *
 * 算法思路:
 * 使用哈希表统计每个单词的频率，然后使用优先队列按频率和字典序排序
 *
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(n)
 */
public List<String> topKFrequent(String[] words, int k) {
    Map<String, Integer> wordCount = new HashMap<>();
    for (String word : words) {
        wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
    }

    PriorityQueue<String> pq = new PriorityQueue<>((a, b) -> {
        int freqCompare = wordCount.get(a) - wordCount.get(b);
```

```
            if (freqCompare != 0) {
                return freqCompare;
            }
            // 频率相同时，按字母顺序逆序排列（因为是小顶堆）
            return b.compareTo(a);
        });

        for (String word : wordCount.keySet()) {
            pq.offer(word);
            if (pq.size() > k) {
                pq.poll();
            }
        }

        List<String> result = new ArrayList<>();
        while (!pq.isEmpty()) {
            result.add(pq.poll());
        }
        Collections.reverse(result);

        return result;
    }
}
```

==========

文件: HashFunctionExtension.py

==========

"""

LeetCode 460. LFU Cache (LFU缓存)

题目来源: https://leetcode.com/problems/lfu-cache/

题目描述:

设计并实现最不经常使用（LFU）缓存的数据结构。

实现 LFUCache 类。

算法思路:

使用两个字典和双向链表实现:

1. key_to_node: 存储键到节点的映射
2. freq_to_list: 存储频率到双向链表的映射，每个频率对应一个双向链表
3. 同时维护一个最小频率变量，用于快速找到最不经常使用的节点

时间复杂度: O(1) - 所有操作都是常数时间

空间复杂度: O(capacity) - 存储容量大小的键值对

```
"""
class Node:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.frequency = 1
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def add_to_head(self, node):
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node
        self.size += 1

    def remove_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev
        self.size -= 1

    def remove_tail(self):
        if self.size == 0:
            return None
        node = self.tail.prev
        self.remove_node(node)
        return node

class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.size = 0
        self.min_freq = 0
        self.key_to_node = {}
        self.freq_to_list = {}
```

```
def get(self, key: int) -> int:
    if self.capacity == 0 or key not in self.key_to_node:
        return -1

    node = self.key_to_node[key]
    self._update_frequency(node)
    return node.value

def put(self, key: int, value: int) -> None:
    if self.capacity == 0:
        return

    if key in self.key_to_node:
        node = self.key_to_node[key]
        node.value = value
        self._update_frequency(node)
    else:
        if self.size == self.capacity:
            min_freq_list = self.freq_to_list[self.min_freq]
            removed = min_freq_list.remove_tail()
            del self.key_to_node[removed.key]
            self.size -= 1

        self.min_freq = 1  # 新节点的频率为1
        new_node = Node(key, value)

        if 1 not in self.freq_to_list:
            self.freq_to_list[1] = DoublyLinkedList()
        self.freq_to_list[1].add_to_head(new_node)
        self.key_to_node[key] = new_node
        self.size += 1

def _update_frequency(self, node):
    old_freq = node.frequency
    new_freq = old_freq + 1

    # 从旧频率列表中移除
    old_list = self.freq_to_list[old_freq]
    old_list.remove_node(node)

    # 如果旧频率列表为空，并且是最小频率，更新最小频率
    if old_freq == self.min_freq and old_list.size == 0:
```

```
            self.min_freq += 1

        # 更新节点频率
        node.frequency = new_freq

        # 添加到新频率列表
        if new_freq not in self.freq_to_list:
            self.freq_to_list[new_freq] = DoublyLinkedList()
        self.freq_to_list[new_freq].add_to_head(node)
"""
LeetCode 811. Subdomain Visit Count (子域名访问计数)
题目来源: https://leetcode.com/problems/subdomain-visit-count/

算法思路:
使用哈希表统计每个域名及其子域名的访问次数

时间复杂度: O(n), 其中 n 是域名字符串的总长度
空间复杂度: O(m), 其中 m 是不同域名的数量
"""
def subdomain_visits(cpdomains: List[str]) -> List[str]:
    counts = {}

    for cpdomain in cpdomains:
        count, domain = cpdomain.split()
        count = int(count)

        # 统计当前完整域名
        counts[domain] = counts.get(domain, 0) + count

        # 统计所有父域名
        parts = domain.split('.')
        for i in range(1, len(parts)):
            parent = '.'.join(parts[i:])
            counts[parent] = counts.get(parent, 0) + count

    # 构建结果
    return [f"{count} {domain}" for domain, count in counts.items()]

"""
LeetCode 554. Brick Wall (砖墙)
题目来源: https://leetcode.com/problems/brick-wall/

算法思路:
```

使用哈希表统计每个位置的砖块边缘数量，然后找出边缘数量最多的位置
穿过的砖块数量 = 总行数 - 该位置的边缘数量

时间复杂度: O(n)，其中 n 是墙中的砖块总数
空间复杂度: O(m)，其中 m 是不同的边缘位置数量
"""
```
def least_bricks(wall: List[List[int]]) -> int:
    edge_count = {}
    max_edges = 0

    for row in wall:
        position = 0
        # 不考虑最后一块砖的右边缘
        for i in range(len(row) - 1):
            position += row[i]
            edge_count[position] = edge_count.get(position, 0) + 1
            max_edges = max(max_edges, edge_count[position])

    # 穿过的砖块数量 = 总行数 - 最大边缘数量
    return len(wall) - max_edges
```

"""
LeetCode 957. Prison Cells After N Days (N 天后的牢房)
题目来源: https://leetcode.com/problems/prison-cells-after-n-days/

算法思路:
由于状态空间有限(2^8=256 种可能)，一定存在循环，使用哈希表检测循环
找到循环后，计算 n % 循环长度 来确定最终状态

时间复杂度: O(min(2^N, N))，其中 N 是牢房数量(这里是 8)
空间复杂度: O(2^N) = O(256)，存储所有可能的状态
"""
```
def prison_after_n_days(cells: List[int], n: int) -> List[int]:
    seen = {}

    # 将状态转换为字符串用于哈希表的键
    cells_key = ''.join(map(str, cells))
    day = 0

    while day < n:
        # 检测循环
        if cells_key in seen:
            cycle_length = day - seen[cells_key]
```

```
            # 跳过完整的循环
            day = day + ((n - day) // cycle_length) * cycle_length

            if day >= n:
                break

        seen[cells_key] = day
        day += 1

        # 计算下一天的状态
        next_cells = [0] * 8
        for i in range(1, 7):
            next_cells[i] = 1 if cells[i-1] == cells[i+1] else 0

        cells = next_cells
        cells_key = ''.join(map(str, cells))

    return cells

"""
LeetCode 1711. Count Good Meals (大餐计数)
题目来源: https://leetcode.com/problems/count-good-meals/

算法思路:
使用哈希表统计每个美味程度出现的次数
对于每个餐品，检查是否存在另一个餐品使得它们的和是 2 的幂

时间复杂度: O(n * log(max))，其中 n 是餐品数量，max 是最大美味程度
空间复杂度: O(n)
"""
def count_pairs(deliciousness: List[int]) -> int:
    freq_map = {}
    MOD = 10**9 + 7
    result = 0

    # 2 的幂可能值 (2^0 到 2^21，因为题目限制最大值为 2^20)
    powers = [1 << i for i in range(22)]

    for num in deliciousness:
        # 检查与当前 num 组成 2 的幂的可能
        for power in powers:
            complement = power - num
            if complement in freq_map:
```

```
                result = (result + freq_map[complement]) % MOD

        # 更新当前 num 的频率
        freq_map[num] = freq_map.get(num, 0) + 1

    return result

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

文件: HashFunctionSupplementary.py
==========================================
"""
哈希函数相关题目集合 - 补充题目
包含来自各大平台的哈希算法相关题目及其详细解析
"""

from typing import List, Dict, Tuple, Set
import collections
import random
import heapq
import bisect
import math
import functools

class HashFunctionProblems:
    @staticmethod
    def group_shifted_strings(strings: List[str]) -> List[List[str]]:
        """
        LeetCode 249. Group Shifted Strings (移位字符串分组)
        题目来源: https://leetcode.com/problems/group-shifted-strings/

        题目描述:
        给定一个字符串列表，将它们分组，使得同一组中的所有字符串都可以通过循环移位得到彼此。
        循环移位是指将字符串中的每个字符向前或向后移动相同的偏移量。
        例如，"abc" 可以通过向前移动 1 位得到 "bcd"，向前移动 2 位得到 "cde" 等。

        示例:
        输入: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]
        输出: [["abc","bcd","xyz"],["az","ba"],["acef"],["a","z"]]

        算法思路:
        1. 为每个字符串生成一个"哈希键"，表示其字符间的相对关系
        2. 哈希键可以是相邻字符之间的差值序列
```

```
    3. 使用哈希表将具有相同哈希键的字符串分组

    时间复杂度: O(n*k)，其中 n 是字符串数量，k 是字符串的平均长度
    空间复杂度: O(n*k)
    """
    def get_hash_key(s: str) -> str:
        # 如果字符串长度为 0 或 1，直接返回特殊标记
        if not s:
            return ""
        if len(s) == 1:
            return "single"

        # 计算相邻字符之间的差值，并考虑循环
        key = []
        for i in range(1, len(s)):
            # 计算差值，考虑循环 (例如 'z' -> 'a' 的差值是 1)
            diff = (ord(s[i]) - ord(s[i-1])) % 26
            key.append(str(diff))

        return ",".join(key)

    # 使用哈希表分组
    groups = collections.defaultdict(list)
    for s in strings:
        key = get_hash_key(s)
        groups[key].append(s)

    # 返回分组结果
    return list(groups.values())


@staticmethod
def isomorphic_strings(s: str, t: str) -> bool:
    """
    LeetCode 205. Isomorphic Strings (同构字符串)
    题目来源: https://leetcode.com/problems/isomorphic-strings/

    题目描述:
    给定两个字符串 s 和 t，判断它们是否是同构的。
    如果 s 中的字符可以按某种映射关系替换得到 t，那么这两个字符串是同构的。
    每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。
    不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上。

    示例:
```

```
    输入: s = "egg", t = "add"
    输出: true

    输入: s = "foo", t = "bar"
    输出: false

    算法思路:
    1. 使用两个哈希表分别记录 s->t 和 t->s 的映射关系
    2. 遍历字符串，检查映射关系是否一致

    时间复杂度: O(n)，其中 n 是字符串长度
    空间复杂度: O(k)，其中 k 是字符集大小，最坏情况为 O(n)
    """
    if len(s) != len(t):
        return False

    s_to_t = {}
    t_to_s = {}

    for i in range(len(s)):
        char_s = s[i]
        char_t = t[i]

        # 检查 s->t 的映射
        if char_s in s_to_t:
            if s_to_t[char_s] != char_t:
                return False
        else:
            s_to_t[char_s] = char_t

        # 检查 t->s 的映射
        if char_t in t_to_s:
            if t_to_s[char_t] != char_s:
                return False
        else:
            t_to_s[char_t] = char_s

    return True

@staticmethod
def longest_harmonious_subsequence(nums: List[int]) -> int:
    """
    LeetCode 594. Longest Harmonious Subsequence (最长和谐子序列)
```

题目来源: https://leetcode.com/problems/longest-harmonious-subsequence/

题目描述:
和谐数组是指一个数组里元素的最大值和最小值之间的差值正好是 1。
现在，给你一个整数数组 nums，请你在所有可能的子序列中找到最长的和谐子序列的长度。
数组的子序列是一个由数组派生出来的序列，它可以通过删除一些元素或不删除元素、且不改变其余元素的顺序而得到。

示例:
输入: nums = [1,3,2,2,5,2,3,7]
输出: 5
解释: 最长的和谐子序列是 [3,2,2,2,3]

算法思路:
1. 使用哈希表统计每个数字出现的次数
2. 遍历哈希表，对于每个数字 x，检查 x+1 是否也在哈希表中
3. 如果存在，则和谐子序列长度为 count(x) + count(x+1)

时间复杂度: O(n)，其中 n 是数组长度
空间复杂度: O(n)
"""
```
# 统计每个数字出现的次数
counter = collections.Counter(nums)

max_length = 0
# 遍历哈希表
for num in counter:
    # 检查 num+1 是否存在
    if num + 1 in counter:
        # 更新最长和谐子序列长度
        max_length = max(max_length, counter[num] + counter[num + 1])

return max_length
```

```
@staticmethod
def find_anagrams(s: str, p: str) -> List[int]:
```
"""
LeetCode 438. Find All Anagrams in a String (找到字符串中所有字母异位词)
题目来源: https://leetcode.com/problems/find-all-anagrams-in-a-string/

题目描述:
给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，且字符串 s 和 p 的长度都不超过 20100。

示例:
输入: s = "cbaebabacd", p = "abc"
输出: [0, 6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

算法思路:
1. 使用滑动窗口和哈希表计数
2. 维护一个长度为 p.length 的窗口，检查窗口中的字符频率是否与 p 相同

时间复杂度: O(n)，其中 n 是字符串 s 的长度
空间复杂度: O(k)，其中 k 是字符集大小，这里是 26
"""

```
if len(s) < len(p):
    return []

result = []
p_count = [0] * 26
s_count = [0] * 26

# 统计 p 中字符频率
for char in p:
    p_count[ord(char) - ord('a')] += 1

# 初始化窗口
for i in range(len(p)):
    s_count[ord(s[i]) - ord('a')] += 1

# 检查初始窗口
if p_count == s_count:
    result.append(0)

# 滑动窗口
for i in range(len(p), len(s)):
    # 添加新字符
    s_count[ord(s[i]) - ord('a')] += 1
    # 移除旧字符
    s_count[ord(s[i - len(p)]) - ord('a')] -= 1

    # 检查当前窗口
```

```
            if p_count == s_count:
                result.append(i - len(p) + 1)

        return result

    @staticmethod
    def longest_substring_with_at_most_k_distinct(s: str, k: int) -> int:
        """
        LeetCode 340. Longest Substring with At Most K Distinct Characters (至多包含K个不同字符的最长子串)
        题目来源: https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/

        题目描述:
        给定一个字符串 s 和一个整数 k，找出 s 中最多包含 k 个不同字符的最长子串的长度。

        示例:
        输入: s = "eceba", k = 2
        输出: 3
        解释: 子串 "ece" 包含 2 个不同的字符。

        算法思路:
        1. 使用滑动窗口和哈希表计数
        2. 维护窗口，使其最多包含k个不同字符
        3. 更新最长子串长度

        时间复杂度: O(n)，其中n是字符串长度
        空间复杂度: O(k)
        """
        if not s or k == 0:
            return 0

        # 滑动窗口左右指针
        left = 0
        max_length = 0
        char_count = {}

        for right in range(len(s)):
            # 更新字符计数
            char_count[s[right]] = char_count.get(s[right], 0) + 1

            # 当窗口中不同字符数量超过k时，移动左指针
            while len(char_count) > k:
```

```
                char_count[s[left]] -= 1
                if char_count[s[left]] == 0:
                    del char_count[s[left]]
                left += 1

            # 更新最长子串长度
            max_length = max(max_length, right - left + 1)

        return max_length
```

==========

文件: HashFunctionSupplementary2.py

==========

```
    @staticmethod
    def max_points_on_line(points: List[List[int]]) -> int:
        """
        LeetCode 149. Max Points on a Line (直线上最多的点数)
        题目来源: https://leetcode.com/problems/max-points-on-a-line/

        题目描述:
        给你一个数组 points，其中 points[i] = [xi, yi] 表示 X-Y 平面上的一个点。
        求最多有多少个点在同一条直线上。

        示例:
        输入: points = [[1,1],[2,2],[3,3]]
        输出: 3

        算法思路:
        1. 对于每个点，计算它与其他点的斜率
        2. 使用哈希表统计相同斜率的点数
        3. 注意处理垂直线（斜率无穷大）和重复点的情况

        时间复杂度: O(n^2)，其中 n 是点的数量
        空间复杂度: O(n)
        """
        if not points:
            return 0
        if len(points) <= 2:
            return len(points)

        max_points = 0
```

```
    for i in range(len(points)):
        # 当前点
        x1, y1 = points[i]

        # 统计重复点和各斜率出现的次数
        slope_count = {}
        duplicates = 0

        for j in range(len(points)):
            if i == j:
                continue

            x2, y2 = points[j]

            # 处理重复点
            if x1 == x2 and y1 == y2:
                duplicates += 1
                continue

            # 计算斜率
            if x1 == x2:  # 垂直线
                slope = float('inf')
            else:
                # 使用最大公约数简化分数，避免浮点数精度问题
                dx = x2 - x1
                dy = y2 - y1
                gcd = math.gcd(dx, dy)
                slope = (dy // gcd, dx // gcd)

            # 更新斜率计数
            slope_count[slope] = slope_count.get(slope, 0) + 1

        # 计算当前点能形成的最大直线点数
        current_max = duplicates + 1  # 包括当前点自身
        if slope_count:
            current_max = max(current_max, max(slope_count.values()) + 1)

        max_points = max(max_points, current_max)

    return max_points

@staticmethod
def contains_nearby_almost_duplicate(nums: List[int], k: int, t: int) -> bool:
```

```
    """
    LeetCode 220. Contains Duplicate III (存在重复元素 III)
    题目来源: https://leetcode.com/problems/contains-duplicate-iii/

    题目描述:
    给你一个整数数组 nums 和两个整数 k 与 t。
    请你判断是否存在两个不同下标 i 和 j，使得 abs(nums[i] - nums[j]) <= t，同时又满足 abs(i - j) <= k。

    示例:
    输入：nums = [1,2,3,1], k = 3, t = 0
    输出：true

    算法思路:
    1. 使用桶排序的思想，将数字分到不同的桶中
    2. 如果两个数字的差值不超过 t，则它们要么在同一个桶中，要么在相邻的桶中
    3. 维护一个大小为 k 的滑动窗口

    时间复杂度: O(n)，其中 n 是数组长度
    空间复杂度: O(min(n, k))
    """
    if t < 0:
        return False

    # 特殊情况处理
    if k <= 0:
        return False

    # 桶的大小为 t+1，确保差值为 t 的两个数在同一个桶或相邻桶中
    bucket_size = t + 1
    buckets = {}

    for i, num in enumerate(nums):
        # 计算桶编号
        bucket_id = num // bucket_size

        # 检查当前桶
        if bucket_id in buckets:
            return True

        # 检查相邻的桶
        if bucket_id - 1 in buckets and abs(num - buckets[bucket_id - 1]) <= t:
            return True
```

```
            if bucket_id + 1 in buckets and abs(num - buckets[bucket_id + 1]) <= t:
                return True

            # 将当前数字放入桶中
            buckets[bucket_id] = num

            # 移除窗口外的桶
            if i >= k:
                del buckets[nums[i - k] // bucket_size]

        return False

    @staticmethod
    def find_all_numbers_disappeared(nums: List[int]) -> List[int]:
        """
        LeetCode 448. Find All Numbers Disappeared in an Array (找到所有数组中消失的数字)
        题目来源: https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/

        题目描述:
        给你一个含 n 个整数的数组 nums，其中 nums[i] 在区间 [1, n] 内。
        请你找出所有在 [1, n] 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

        示例:
        输入：nums = [4,3,2,7,8,2,3,1]
        输出：[5,6]

        算法思路:
        1. 利用数组索引作为哈希表
        2. 遍历数组，将每个数对应位置的数标记为负数
        3. 再次遍历数组，如果某个位置的数是正数，说明该位置对应的数没有出现

        时间复杂度: O(n)
        空间复杂度: O(1)，不使用额外空间
        """
        # 第一次遍历，标记出现的数字
        for num in nums:
            # 取绝对值，因为可能已经被标记为负数
            index = abs(num) - 1
            # 将对应位置的数标记为负数
            if nums[index] > 0:
                nums[index] = -nums[index]

        # 第二次遍历，找出未标记的位置
```

```
    result = []
    for i in range(len(nums)):
        if nums[i] > 0:
            result.append(i + 1)

    return result

@staticmethod
def first_unique_character(s: str) -> int:
    """
    LeetCode 387. First Unique Character in a String (字符串中的第一个唯一字符)
    题目来源: https://leetcode.com/problems/first-unique-character-in-a-string/

    题目描述:
    给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

    示例:
    输入: s = "leetcode"
    输出: 0

    输入: s = "loveleetcode"
    输出: 2

    算法思路:
    1. 使用哈希表统计每个字符出现的次数
    2. 再次遍历字符串，找到第一个出现次数为 1 的字符

    时间复杂度: O(n)，其中 n 是字符串长度
    空间复杂度: O(k)，其中 k 是字符集大小，最多为 26
    """
    # 统计字符出现次数
    char_count = {}
    for char in s:
        char_count[char] = char_count.get(char, 0) + 1

    # 找到第一个出现次数为 1 的字符
    for i, char in enumerate(s):
        if char_count[char] == 1:
            return i

    return -1

@staticmethod
```

```
def longest_palindrome(s: str) -> int:
    """
    LeetCode 409. Longest Palindrome (最长回文串)
    题目来源: https://leetcode.com/problems/longest-palindrome/

    题目描述:
    给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。
    在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

    示例:
    输入: "abccccdd"
    输出: 7
    解释: 我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

    算法思路:
    1. 使用哈希表统计每个字符出现的次数
    2. 对于每个字符，可以使用其出现次数的最大偶数部分
    3. 如果存在出现奇数次的字符，可以额外使用一个字符作为回文中心

    时间复杂度: O(n)，其中 n 是字符串长度
    空间复杂度: O(k)，其中 k 是字符集大小
    """
    # 统计字符出现次数
    char_count = {}
    for char in s:
        char_count[char] = char_count.get(char, 0) + 1

    length = 0
    has_odd = False

    # 计算最长回文串长度
    for count in char_count.values():
        # 使用偶数部分
        length += count // 2 * 2
        # 检查是否有奇数次出现的字符
        if count % 2 == 1:
            has_odd = True

    # 如果存在奇数次出现的字符，可以额外使用一个字符作为回文中心
    if has_odd:
        length += 1

    return length
```

```
    @staticmethod
    def find_pairs_with_difference(nums: List[int], k: int) -> int:
        """
        LeetCode 532. K-diff Pairs in an Array (数组中的 k-diff 数对)
        题目来源: https://leetcode.com/problems/k-diff-pairs-in-an-array/

        题目描述:
        给定一个整数数组和一个整数 k，你需要在数组里找到不同的 k-diff 数对。
        这里将 k-diff 数对定义为一个整数对 (i, j)，其中 i 和 j 都是数组中的数字，且两数之差的绝对值是 k。

        示例:
        输入: [3, 1, 4, 1, 5], k = 2
        输出: 2
        解释: 数组中有两个 2-diff 数对, (1, 3) 和 (3, 5)。
        尽管数组中有两个 1，但我们只应返回不同的数对的数量。

        算法思路:
        1. 使用哈希表存储数组中的数字及其出现次数
        2. 对于每个数字，检查其加上 k 或减去 k 的数字是否在哈希表中
        3. 特殊处理 k=0 的情况，此时需要找出数组中出现次数大于 1 的数字

        时间复杂度: O(n)，其中 n 是数组长度
        空间复杂度: O(n)
        """
        if k < 0:
            return 0

        # 统计数字出现次数
        num_count = {}
        for num in nums:
            num_count[num] = num_count.get(num, 0) + 1

        count = 0

        # 特殊处理 k=0 的情况
        if k == 0:
            for num, freq in num_count.items():
                if freq > 1:
                    count += 1
            return count
```

```
        # 处理k>0的情况
        for num in num_count:
            if num + k in num_count:
                count += 1

        return count

    @staticmethod
    def subarray_sum_divisible_by_k(nums: List[int], k: int) -> int:
        """
        LeetCode 974. Subarray Sums Divisible by K (和可被 K 整除的子数组)
        题目来源: https://leetcode.com/problems/subarray-sums-divisible-by-k/

        题目描述:
        给定一个整数数组 nums 和一个整数 k，返回其中元素之和可被 k 整除的（连续、非空）子数组的数目。

        示例:
        输入: nums = [4,5,0,-2,-3,1], k = 5
        输出: 7
        解释: 有 7 个子数组满足其元素之和可被 k = 5 整除：
        [4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

        算法思路:
        1. 使用前缀和和哈希表
        2. 对于前缀和，我们关注的是前缀和除以k的余数
        3. 如果两个前缀和对k取余结果相同，则它们之间的子数组和能被k整除

        时间复杂度: O(n)，其中n是数组长度
        空间复杂度: O(k)，哈希表最多存储k个不同的余数
        """
        # 初始化哈希表，前缀和为0的情况出现1次
        prefix_sum_count = {0: 1}
        prefix_sum = 0
        count = 0

        for num in nums:
            # 计算前缀和
            prefix_sum = (prefix_sum + num) % k
            # 处理负数情况
            if prefix_sum < 0:
                prefix_sum += k
```

```
            # 如果当前前缀和的余数之前出现过，则可以形成能被 k 整除的子数组
            if prefix_sum in prefix_sum_count:
                count += prefix_sum_count[prefix_sum]

            # 更新前缀和余数的出现次数
            prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1

        return count

    @staticmethod
    def longest_word_in_dictionary(words: List[str]) -> str:
        """
        LeetCode 720. Longest Word in Dictionary (词典中最长的单词)
        题目来源: https://leetcode.com/problems/longest-word-in-dictionary/

        题目描述:
        给出一个字符串数组 words 组成的一本英语词典。从中找出最长的一个单词，该单词是由 words 词典中其他单词逐步添加一个字母组成。
        若其中有多个可行的答案，则返回答案中字典序最小的单词。若无答案，则返回空字符串。

        示例:
        输入: words = ["w","wo","wor","worl", "world"]
        输出: "world"
        解释: 单词"world"可由"w", "wo", "wor", 和 "worl"添加一个字母组成。

        算法思路:
        1. 将所有单词放入集合中，便于快速查找
        2. 对单词列表按长度降序、字典序升序排序
        3. 检查每个单词的所有前缀是否都在集合中

        时间复杂度: O(n * l^2)，其中 n 是单词数量，l 是单词的平均长度
        空间复杂度: O(n * l)
        """
        # 将所有单词放入集合
        word_set = set(words)

        # 按长度降序、字典序升序排序
        sorted_words = sorted(words, key=lambda x: (-len(x), x))

        for word in sorted_words:
            valid = True
            # 检查所有前缀是否都在集合中
            for i in range(1, len(word)):
```

```
            if word[:i] not in word_set:
                valid = False
                break

        if valid:
            return word

    return ""

@staticmethod
def valid_sudoku(board: List[List[str]]) -> bool:
    """
    LeetCode 36. Valid Sudoku (有效的数独)
    题目来源: https://leetcode.com/problems/valid-sudoku/

    题目描述:
    判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
    1. 数字 1-9 在每一行只能出现一次。
    2. 数字 1-9 在每一列只能出现一次。
    3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

    示例:
    输入:
    [
      ["5","3",".",".","7",".",".",".",".","."],
      ["6",".",".","1","9","5",".",".",".","."],
      [".","9","8",".",".",".",".",".","6","."],
      ["8",".",".",".",".","6",".",".",".","3"],
      ["4",".",".","8",".","3",".",".",".","1"],
      ["7",".",".",".",".","2",".",".",".","6"],
      [".","6",".",".",".",".",".","2","8","."],
      [".",".",".","4","1","9",".",".",".","5"],
      [".",".",".",".",".","8",".",".","7","9"]
    ]
    输出: true

    算法思路:
    1. 使用三个哈希表分别记录每行、每列、每个 3x3 宫格中数字的出现情况
    2. 遍历整个数独，检查是否有重复数字

    时间复杂度: O(1)，因为数独大小固定为 9x9
    空间复杂度: O(1)
    """
```

```
        # 初始化哈希表
        rows = [set() for _ in range(9)]
        cols = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # 遍历数独
        for i in range(9):
            for j in range(9):
                # 跳过空格
                if board[i][j] == '.':
                    continue

                num = board[i][j]
                # 计算 3x3 宫格的索引
                box_index = (i // 3) * 3 + j // 3

                # 检查是否有重复数字
                if num in rows[i] or num in cols[j] or num in boxes[box_index]:
                    return False

                # 记录数字出现情况
                rows[i].add(num)
                cols[j].add(num)
                boxes[box_index].add(num)

        return True
```

==========

文件: HashFunctionSupplementary3.py

==========

```
    @staticmethod
    def four_sum_ii(nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) ->
int:
        """
        LeetCode 454. 4Sum II (四数相加 II)
        题目来源: https://leetcode.com/problems/4sum-ii/

        题目描述:
        给定四个包含整数的数组列表 A, B, C, D, 计算有多少个元组 (i, j, k, l) , 使得 A[i] + B[j] +
C[k] + D[l] = 0。
        为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 0 ≤ N ≤ 500。所有整数的范围在
-2^28 到 2^28 - 1 之间，
```

```
    最终结果不会超过 2^31 - 1。

    示例:
    输入:
    A = [ 1, 2]
    B = [-2,-1]
    C = [-1, 2]
    D = [ 0, 2]
    输出: 2
    解释:
    两个元组如下:
    1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
    2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

    算法思路:
    1. 将四个数组分成两组，分别计算两数之和
    2. 使用哈希表存储第一组中所有可能的两数之和及其出现次数
    3. 遍历第二组中的所有两数之和，检查其相反数是否在哈希表中

    时间复杂度: O(n^2)，其中 n 是数组长度
    空间复杂度: O(n^2)
    """
    # 计算 nums1 和 nums2 中所有可能的两数之和及其出现次数
    sum_count = {}
    for a in nums1:
        for b in nums2:
            sum_ab = a + b
            sum_count[sum_ab] = sum_count.get(sum_ab, 0) + 1

    # 计算有多少对 nums3 和 nums4 的元素之和等于-(nums1+nums2)
    count = 0
    for c in nums3:
        for d in nums4:
            sum_cd = -(c + d)
            if sum_cd in sum_count:
                count += sum_count[sum_cd]

    return count

@staticmethod
def contiguous_array(nums: List[int]) -> int:
    """
    LeetCode 525. Contiguous Array (连续数组)
```

题目来源: https://leetcode.com/problems/contiguous-array/

题目描述:
给定一个二进制数组 nums, 找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。

示例:
输入: [0,1]
输出: 2
说明: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。

输入: [0,1,0]
输出: 2
说明: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。

算法思路:
1. 将 0 视为-1，1 保持不变，问题转化为求和为 0 的最长子数组
2. 使用前缀和和哈希表记录每个前缀和第一次出现的位置
3. 当遇到相同的前缀和时，计算子数组长度

时间复杂度: O(n)，其中 n 是数组长度
空间复杂度: O(n)
"""
```
# 初始化哈希表，前缀和为 0 的位置为-1
prefix_sum_index = {0: -1}
prefix_sum = 0
max_length = 0

for i, num in enumerate(nums):
    # 将 0 视为-1，1 保持不变
    prefix_sum += 1 if num == 1 else -1

    # 如果当前前缀和之前出现过，计算子数组长度
    if prefix_sum in prefix_sum_index:
        max_length = max(max_length, i - prefix_sum_index[prefix_sum])
    else:
        # 记录前缀和第一次出现的位置
        prefix_sum_index[prefix_sum] = i

return max_length

@staticmethod
def line_reflection(points: List[List[int]]) -> bool:
```

"""
LeetCode 356. Line Reflection (直线镜像)
题目来源: https://leetcode.com/problems/line-reflection/

题目描述:
在一个二维平面上，如果我们放置一些点，那么可能会有一些点在同一条垂直线上。
对于这样的每一条垂直线，我们可以找到其上最上面和最下面的点。
如果这些点可以构成一个镜像，则返回 true；否则，返回 false。

示例:
输入: [[1,1],[-1,1]]
输出: true

输入: [[1,1],[-1,-1]]
输出: false

算法思路:
1. 找到所有点的 x 坐标的最小值和最大值
2. 计算对称轴的 x 坐标：(min_x + max_x) / 2
3. 对于每个点，检查其关于对称轴的镜像点是否存在

时间复杂度: O(n)，其中 n 是点的数量
空间复杂度: O(n)
"""
```
if not points:
    return True

# 去除重复点
point_set = set()
for x, y in points:
    point_set.add((x, y))

# 找到所有点的 x 坐标的最小值和最大值
min_x = float('inf')
max_x = float('-inf')
for x, y in point_set:
    min_x = min(min_x, x)
    max_x = max(max_x, x)

# 计算对称轴的 x 坐标
mid_x = min_x + max_x

# 检查每个点的镜像是否存在
```

```
        for x, y in point_set:
            mirror_x = mid_x - x
            if (mirror_x, y) not in point_set:
                return False

        return True

    @staticmethod
    def find_duplicate_subtrees(root):
        """
        LeetCode 652. Find Duplicate Subtrees (寻找重复的子树)
        题目来源: https://leetcode.com/problems/find-duplicate-subtrees/

        题目描述:
        给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。
        两棵树重复是指它们具有相同的结构以及相同的结点值。

        示例:
        输入:
            1
          / \\
         2   3
        /   / \\
       4   2   4
          /
         4
        输出: [2,4]
        解释: 上面的 2 和 4 都是重复的子树。

        算法思路:
        1. 使用哈希表存储每个子树的序列化表示及其出现次数
        2. 使用后序遍历序列化每个子树
        3. 当某个子树的序列化表示出现第二次时，将其添加到结果中

        时间复杂度: O(n^2)，其中 n 是树中节点数量，序列化每个子树需要 O(n)时间
        空间复杂度: O(n^2)
        """
        # 定义 TreeNode 类
        class TreeNode:
            def __init__(self, val=0, left=None, right=None):
                self.val = val
                self.left = left
```

```
            self.right = right

        def serialize(node, subtree_map, result):
            if not node:
                return "#"

            # 后序遍历序列化子树
            left_serialize = serialize(node.left, subtree_map, result)
            right_serialize = serialize(node.right, subtree_map, result)

            # 当前子树的序列化表示
            serialized = left_serialize + "," + right_serialize + "," + str(node.val)

            # 更新子树出现次数
            subtree_map[serialized] = subtree_map.get(serialized, 0) + 1

            # 如果当前子树出现第二次，添加到结果中
            if subtree_map[serialized] == 2:
                result.append(node)

            return serialized

        result = []
        subtree_map = {}
        serialize(root, subtree_map, result)
        return result

    @staticmethod
    def unique_word_abbreviation(dictionary: List[str], word: str) -> bool:
        """
        LeetCode 288. Unique Word Abbreviation (单词的唯一缩写)
        题目来源: https://leetcode.com/problems/unique-word-abbreviation/

        题目描述:
        一个单词的缩写需要遵循 <首字母><中间字母数><尾字母> 的格式。
        例如，"abcde" 可以缩写为 "a3e"。
        给定一个字典和一个单词，判断该单词的缩写在字典中是否唯一。
        若单词的缩写在字典中没有任何 其他 单词与其缩写相同，则被认为是唯一的。

        示例:
        输入: dictionary = ["deer", "door", "cake", "card"], word = "dear"
        输出: false
        解释: 单词 "dear" 的缩写是 "d2r"，在字典中与 "deer" 的缩写相同。
```

```
    算法思路:
    1. 定义一个函数计算单词的缩写
    2. 使用哈希表存储字典中每个缩写对应的单词集合
    3. 检查 word 的缩写在字典中是否唯一

    时间复杂度: O(n)，其中 n 是字典中单词的数量
    空间复杂度: O(n)
    """
    def get_abbreviation(word):
        if len(word) <= 2:
            return word
        return word[0] + str(len(word) - 2) + word[-1]

    # 构建缩写到单词集合的映射
    abbr_to_words = {}
    for dict_word in dictionary:
        abbr = get_abbreviation(dict_word)
        if abbr not in abbr_to_words:
            abbr_to_words[abbr] = set()
        abbr_to_words[abbr].add(dict_word)

    # 计算 word 的缩写
    word_abbr = get_abbreviation(word)

    # 检查 word 的缩写是否唯一
    if word_abbr not in abbr_to_words:
        return True

    # 如果缩写对应的单词集合中只有 word 自己，也算唯一
    if len(abbr_to_words[word_abbr]) == 1 and word in abbr_to_words[word_abbr]:
        return True

    return False

@staticmethod
def logger_rate_limiter():
    """
    LeetCode 359. Logger Rate Limiter (日志速率限制器)
    题目来源: https://leetcode.com/problems/logger-rate-limiter/

    题目描述:
    设计一个日志系统，可以流式接收日志消息。每条消息都有一个唯一的 ID 和时间戳。
```

消息的时间戳是一个整数，表示消息发生的时间（以秒为单位）。
实现一个日志速率限制器，每条独特的消息最多每 10 秒打印一次。

示例:
logger.shouldPrintMessage(1, "foo"); // 返回 true，打印消息
logger.shouldPrintMessage(2, "bar"); // 返回 true，打印消息
logger.shouldPrintMessage(3, "foo"); // 返回 false，不打印消息，因为距离上次打印 "foo" 不足 10 秒
logger.shouldPrintMessage(8, "bar"); // 返回 false，不打印消息，因为距离上次打印 "bar" 不足 10 秒
logger.shouldPrintMessage(10, "foo"); // 返回 false，不打印消息，因为距离上次打印 "foo" 不足 10 秒
logger.shouldPrintMessage(11, "foo"); // 返回 true，打印消息，因为距离上次打印 "foo" 已经 10 秒了

算法思路:
1. 使用哈希表存储每个消息最后打印的时间戳
2. 对于每个新消息，检查是否距离上次打印已经过了 10 秒

时间复杂度: O(1) 每次操作
空间复杂度: O(m)，其中 m 是不同消息的数量
"""
```
class Logger:
    def __init__(self):
        # 存储消息最后打印的时间戳
        self.message_timestamps = {}

    def shouldPrintMessage(self, timestamp: int, message: str) -> bool:
        # 检查消息是否应该被打印
        if message not in self.message_timestamps or timestamp - self.message_timestamps[message] >= 10:
            self.message_timestamps[message] = timestamp
            return True
        return False

# 使用示例
logger = Logger()
print(logger.shouldPrintMessage(1, "foo"))  # 返回 true
print(logger.shouldPrintMessage(2, "bar"))  # 返回 true
print(logger.shouldPrintMessage(3, "foo"))  # 返回 false
print(logger.shouldPrintMessage(8, "bar"))  # 返回 false
print(logger.shouldPrintMessage(10, "foo"))  # 返回 false
print(logger.shouldPrintMessage(11, "foo"))  # 返回 true
```

```
        return logger

    @staticmethod
    def bulls_and_cows(secret: str, guess: str) -> str:
        """
        LeetCode 299. Bulls and Cows (猜数字游戏)
        题目来源: https://leetcode.com/problems/bulls-and-cows/

        题目描述:
        你在和朋友玩猜数字（Bulls and Cows）游戏，该游戏规则如下：
        1. 你写出一个秘密数字，并请朋友猜这个数字是多少。
        2. 朋友每猜测一次，你就会给他一个提示，告诉他的猜测数字中有多少位属于数字和确切位置都猜对了（称为"Bulls", 公牛），
           有多少位属于数字猜对了但是位置不对（称为"Cows", 奶牛）。
        3. 朋友根据提示继续猜，直到猜出秘密数字。

        请写出一个根据秘密数字和朋友的猜测数返回提示的函数，返回字符串的格式为 xAyB，x 表示公牛个数，y 表示奶牛个数。

        示例:
        输入: secret = "1807", guess = "7810"
        输出: "1A3B"
        解释: 数字和位置都对的是 8，数字对但位置不对的是 1, 7, 0

        算法思路:
        1. 遍历 secret 和 guess，统计 Bulls（位置和数字都正确的）
        2. 使用哈希表统计 secret 和 guess 中每个数字出现的次数
        3. 计算 Cows（数字正确但位置不对的）= min(secret 中数字出现次数, guess 中数字出现次数) - Bulls

        时间复杂度: O(n)，其中 n 是字符串长度
        空间复杂度: O(1)，因为数字只有 0-9，哈希表大小是常数
        """
        bulls = 0
        cows = 0

        # 统计 secret 和 guess 中每个数字出现的次数
        secret_count = [0] * 10
        guess_count = [0] * 10

        # 计算 Bulls 并统计数字出现次数
        for i in range(len(secret)):
```

```
            s = int(secret[i])
            g = int(guess[i])

            if s == g:
                bulls += 1
            else:
                secret_count[s] += 1
                guess_count[g] += 1

        # 计算 Cows
        for i in range(10):
            cows += min(secret_count[i], guess_count[i])

        return f"{bulls}A{cows}B"

    @staticmethod
    def sort_characters_by_frequency(s: str) -> str:
        """
        LeetCode 451. Sort Characters By Frequency (根据字符出现频率排序)
        题目来源: https://leetcode.com/problems/sort-characters-by-frequency/

        题目描述:
        给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

        示例:
        输入: "tree"
        输出: "eert"
        解释: 'e'出现两次，'r'和't'都只出现一次。因此'e'必须出现在'r'和't'之前。此外，"eetr"也是一个有效的答案。

        算法思路:
        1. 使用哈希表统计每个字符出现的频率
        2. 按照频率降序排列字符
        3. 根据排序后的结果重建字符串

        时间复杂度: O(n log k)，其中 n 是字符串长度，k 是不同字符的数量
        空间复杂度: O(n)
        """
        # 统计字符频率
        char_count = {}
        for char in s:
            char_count[char] = char_count.get(char, 0) + 1
```

```
        # 按频率降序排列字符
        sorted_chars = sorted(char_count.keys(), key=lambda x: char_count[x], reverse=True)

        # 重建字符串
        result = []
        for char in sorted_chars:
            result.append(char * char_count[char])

        return ''.join(result)

    @staticmethod
    def longest_substring_with_at_most_two_distinct(s: str) -> int:
        """
        LeetCode 159. Longest Substring with At Most Two Distinct Characters (至多包含两个不同字符的最长子串)
        题目来源: https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/

        题目描述:
        给定一个字符串 s，找出至多包含两个不同字符的最长子串的长度。

        示例:
        输入: "eceba"
        输出: 3
        解释: 子串 "ece" 包含 2 个不同的字符。

        输入: "ccaabbb"
        输出: 5
        解释: 子串 "aabbb" 包含 2 个不同的字符。

        算法思路:
        1. 使用滑动窗口和哈希表
        2. 维护一个窗口，使其最多包含两个不同字符
        3. 当窗口中不同字符超过两个时，移动左指针

        时间复杂度: O(n)，其中 n 是字符串长度
        空间复杂度: O(1)，因为最多只有两个不同字符
        """
        if not s:
            return 0

        # 滑动窗口左右指针
        left = 0
```

```
        max_length = 0
        char_count = {}

        for right in range(len(s)):
            # 更新字符计数
            char_count[s[right]] = char_count.get(s[right], 0) + 1

            # 当窗口中不同字符超过两个时，移动左指针
            while len(char_count) > 2:
                char_count[s[left]] -= 1
                if char_count[s[left]] == 0:
                    del char_count[s[left]]
                left += 1

            # 更新最长子串长度
            max_length = max(max_length, right - left + 1)

        return max_length

    @staticmethod
    def encode_and_decode_tiny_url():
        """
        LeetCode 535. Encode and Decode TinyURL (TinyURL 的加密与解密)
        题目来源: https://leetcode.com/problems/encode-and-decode-tinyurl/

        题目描述:
        TinyURL 是一种 URL 简化服务，比如：当你输入一个 URL https://leetcode.com/problems/design-tinyurl 时，
        它将返回一个简化的 URL http://tinyurl.com/4e9iAk.

        要求：设计一个 TinyURL 的加密 encode 和解密 decode 的方法。
        你的加密和解密算法如何设计和运作是没有限制的，你只需要保证一个 URL 可以被加密成一个 TinyURL，
        并且这个 TinyURL 可以用解密方法恢复成原本的 URL。

        算法思路:
        1. 使用哈希表存储长 URL 到短 URL 的映射，以及短 URL 到长 URL 的映射
        2. 生成短 URL 时，可以使用递增 ID、随机字符串或哈希函数
        3. 这里使用简单的递增 ID 方法

        时间复杂度: O(1) 每次操作
        空间复杂度: O(n)，其中 n 是不同 URL 的数量
        """
```

```
class Codec:
    def __init__(self):
        self.url_to_code = {}
        self.code_to_url = {}
        self.base = "http://tinyurl.com/"
        self.counter = 0

    def encode(self, longUrl: str) -> str:
        # 如果长URL已经被编码过，直接返回对应的短URL
        if longUrl in self.url_to_code:
            return self.base + self.url_to_code[longUrl]

        # 生成新的短URL编码
        self.counter += 1
        code = str(self.counter)

        # 存储映射关系
        self.url_to_code[longUrl] = code
        self.code_to_url[code] = longUrl

        return self.base + code

    def decode(self, shortUrl: str) -> str:
        # 提取短URL中的编码部分
        code = shortUrl.replace(self.base, "")

        # 返回对应的长URL
        return self.code_to_url.get(code, "")

# 使用示例
codec = Codec()
url = "https://leetcode.com/problems/design-tinyurl"
tiny_url = codec.encode(url)
original_url = codec.decode(tiny_url)
print(f"Original URL: {url}")
print(f"Tiny URL: {tiny_url}")
print(f"Decoded URL: {original_url}")

return codec
```

==========

文件: HashFunctionSupplementary4.py

```
==========================================
    @staticmethod
    def find_restaurant(list1: List[str], list2: List[str]) -> List[str]:
        """
        LeetCode 599. Minimum Index Sum of Two Lists (两个列表的最小索引总和)
        题目来源: https://leetcode.com/problems/minimum-index-sum-of-two-lists/

        题目描述:
        假设 Andy 和 Doris 想在晚餐时选择一家餐厅，并且他们都有一个表示最喜爱餐厅的列表，每个餐厅的名字用字符串表示。
        你需要帮助他们用最少的索引和找出他们共同喜爱的餐厅。 如果答案不止一个，则输出所有答案并且不考虑顺序。

        示例:
        输入: list1 = ["Shogun", "Tapioca Express", "Burger King", "KFC"]
                list2 = ["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]
        输出: ["Shogun"]
        解释: 他们唯一共同喜爱的餐厅是"Shogun"。

        算法思路:
        1. 使用哈希表存储 list1 中每个餐厅的名称和索引
        2. 遍历 list2，查找共同喜爱的餐厅，并计算索引和
        3. 返回索引和最小的餐厅

        时间复杂度: O(n+m)，其中 n 和 m 分别是两个列表的长度
        空间复杂度: O(n)
        """
        # 存储 list1 中餐厅名称和索引
        restaurant_index = {}
        for i, restaurant in enumerate(list1):
            restaurant_index[restaurant] = i

        min_index_sum = float('inf')
        result = []

        # 遍历 list2，查找共同喜爱的餐厅
        for j, restaurant in enumerate(list2):
            if restaurant in restaurant_index:
                index_sum = restaurant_index[restaurant] + j

                # 如果找到更小的索引和，更新结果
                if index_sum < min_index_sum:
```

```
                min_index_sum = index_sum
                result = [restaurant]
            # 如果索引和相等，添加到结果中
            elif index_sum == min_index_sum:
                result.append(restaurant)

    return result

@staticmethod
def find_common_characters(words: List[str]) -> List[str]:
    """
    LeetCode 1002. Find Common Characters (查找共用字符)
    题目来源: https://leetcode.com/problems/find-common-characters/

    题目描述:
    给定仅有小写字母组成的字符串数组 A，返回列表中的每个字符串中都显示的全部字符（包括重复字符）组成的列表。
    例如，如果一个字符在每个字符串中出现 3 次，但不是 4 次，则需要在最终答案中包含该字符 3 次。
    你可以按任意顺序返回答案。

    示例:
    输入：["bella","label","roller"]
    输出：["e","l","l"]

    输入：["cool","lock","cook"]
    输出：["c","o"]

    算法思路:
    1. 使用计数器统计每个单词中字符出现的次数
    2. 对于每个字符，取所有单词中出现次数的最小值
    3. 根据最小出现次数构建结果列表

    时间复杂度: O(n*k)，其中 n 是单词数量，k 是单词的平均长度
    空间复杂度: O(1)，因为字符集固定为 26 个小写字母
    """
    if not words:
        return []

    # 统计第一个单词中每个字符出现的次数
    char_count = {}
    for char in words[0]:
        char_count[char] = char_count.get(char, 0) + 1
```

```
        # 对于其他单词，更新每个字符出现的最小次数
        for i in range(1, len(words)):
            word_count = {}
            for char in words[i]:
                word_count[char] = word_count.get(char, 0) + 1

            # 更新最小出现次数
            for char in list(char_count.keys()):
                if char in word_count:
                    char_count[char] = min(char_count[char], word_count[char])
                else:
                    del char_count[char]

        # 构建结果列表
        result = []
        for char, count in char_count.items():
            result.extend([char] * count)

        return result

    @staticmethod
    def most_common_word(paragraph: str, banned: List[str]) -> str:
        """
        LeetCode 819. Most Common Word (最常见的单词)
        题目来源: https://leetcode.com/problems/most-common-word/

        题目描述:
        给定一个段落 (paragraph) 和一个禁用单词列表 (banned)。返回出现次数最多，同时不在禁用列表中的单词。
        题目保证至少有一个词不在禁用列表中，而且答案唯一。
        禁用列表中的单词用小写字母表示，不含标点符号。段落中的单词不区分大小写。答案都是小写字母。

        示例:
        输入:
        paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."
        banned = ["hit"]
        输出: "ball"
        解释:
        "hit" 出现了 3 次，但它是一个禁用的单词。
        "ball" 出现了 2 次 (同时没有其他单词出现 2 次)，所以它是段落里出现次数最多的，且不在禁用列表中的单词。
```

```
        算法思路:
        1. 将段落转换为小写，并替换所有标点符号为空格
        2. 分割段落为单词，并统计每个单词出现的次数
        3. 排除禁用单词，返回出现次数最多的单词

        时间复杂度: O(n)，其中 n 是段落的长度
        空间复杂度: O(n)
        """
        # 将段落转换为小写，并替换所有标点符号为空格
        for c in "!?',;.":
            paragraph = paragraph.replace(c, " ")

        # 分割段落为单词，并转换为小写
        words = paragraph.lower().split()

        # 创建禁用单词集合，便于快速查找
        banned_set = set(banned)

        # 统计单词出现次数
        word_count = {}
        for word in words:
            if word not in banned_set:
                word_count[word] = word_count.get(word, 0) + 1

        # 返回出现次数最多的单词
        return max(word_count.items(), key=lambda x: x[1])[0]

    @staticmethod
    def subdomain_visit_count(cpdomains: List[str]) -> List[str]:
        """
        LeetCode 811. Subdomain Visit Count (子域名访问计数)
        题目来源: https://leetcode.com/problems/subdomain-visit-count/

        题目描述:
        一个网站域名，如"discuss.leetcode.com"，包含了多个子域名。作为顶级域名，常用的有"com"，下一级则有"leetcode.com"，
        最低的一级为"discuss.leetcode.com"。当我们访问域名"discuss.leetcode.com"时，也同时访问了其父域名"leetcode.com"以及顶级域名 "com"。

        给定一个带访问次数和域名的组合，要求分别计算每个域名被访问的次数。其格式为访问次数+空格+地址，例如："9001 discuss.leetcode.com"。
```

接下来会给出一组访问次数和域名组合的列表 cpdomains。要求解析出所有域名的访问次数，输出格式和输入格式相同，不限定先后顺序。

示例:
输入: ["9001 discuss.leetcode.com"]
输出: ["9001 discuss.leetcode.com", "9001 leetcode.com", "9001 com"]
解释:
例子中仅包含一个网站域名："discuss.leetcode.com"。
按照前文假设，子域名"leetcode.com"和"com"都会被访问，所以它们都被访问了 9001 次。

算法思路:
1. 解析每个域名组合，提取访问次数和完整域名
2. 分割域名，统计每个子域名的访问次数
3. 格式化输出结果

时间复杂度: O(n)，其中 n 是域名组合的总长度
空间复杂度: O(m)，其中 m 是不同子域名的数量
"""
```
# 统计每个子域名的访问次数
domain_count = {}

for cpdomain in cpdomains:
    # 解析访问次数和完整域名
    count, domain = cpdomain.split()
    count = int(count)

    # 分割域名，统计每个子域名的访问次数
    parts = domain.split('.')
    for i in range(len(parts)):
        subdomain = '.'.join(parts[i:])
        domain_count[subdomain] = domain_count.get(subdomain, 0) + count

# 格式化输出结果
result = []
for domain, count in domain_count.items():
    result.append(f"{count} {domain}")

return result
```

```
@staticmethod
def uncommon_words_from_two_sentences(s1: str, s2: str) -> List[str]:
    """
    LeetCode 884. Uncommon Words from Two Sentences (两句话中的不常见单词)
```

```
    题目来源: https://leetcode.com/problems/uncommon-words-from-two-sentences/

    题目描述:
    给定两个句子 A 和 B，返回所有在句子 A 和 B 中都只出现一次的单词。
    你可以按任意顺序返回答案。

    示例:
    输入：A = "this apple is sweet", B = "this apple is sour"
    输出：["sweet","sour"]

    输入：A = "apple apple", B = "banana"
    输出：["banana"]

    算法思路:
    1. 将两个句子合并，分割为单词
    2. 统计每个单词出现的次数
    3. 返回只出现一次的单词

    时间复杂度: O(n+m)，其中 n 和 m 分别是两个句子的长度
    空间复杂度: O(n+m)
    """
    # 将两个句子合并，分割为单词
    words = s1.split() + s2.split()

    # 统计每个单词出现的次数
    word_count = {}
    for word in words:
        word_count[word] = word_count.get(word, 0) + 1

    # 返回只出现一次的单词
    return [word for word, count in word_count.items() if count == 1]

@staticmethod
def intersection_of_two_arrays(nums1: List[int], nums2: List[int]) -> List[int]:
    """
    LeetCode 349. Intersection of Two Arrays (两个数组的交集)
    题目来源: https://leetcode.com/problems/intersection-of-two-arrays/

    题目描述:
    给定两个数组，编写一个函数来计算它们的交集。

    示例:
    输入：nums1 = [1,2,2,1], nums2 = [2,2]
```

输出：[2]

输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]

说明：
输出结果中的每个元素一定是唯一的。
我们可以不考虑输出结果的顺序。

算法思路:
1. 使用集合存储 nums1 中的元素
2. 遍历 nums2，找出同时存在于 nums1 中的元素
3. 使用集合去重

时间复杂度: O(n+m)，其中 n 和 m 分别是两个数组的长度
空间复杂度: O(n+m)
"""
# 使用集合存储 nums1 中的元素
set1 = set(nums1)

# 找出同时存在于 nums1 和 nums2 中的元素
result = set()
for num in nums2:
    if num in set1:
        result.add(num)

return list(result)

```
@staticmethod
def intersection_of_two_arrays_ii(nums1: List[int], nums2: List[int]) -> List[int]:
    """
    LeetCode 350. Intersection of Two Arrays II (两个数组的交集 II)
    题目来源: https://leetcode.com/problems/intersection-of-two-arrays-ii/

    题目描述:
    给定两个数组，编写一个函数来计算它们的交集。

    示例:
    输入：nums1 = [1,2,2,1], nums2 = [2,2]
    输出：[2,2]

    输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
    输出：[4,9]
```

```
    说明：
    输出结果中每个元素出现的次数，应与元素在两个数组中出现次数的最小值一致。
    我们可以不考虑输出结果的顺序。

    算法思路:
    1. 使用哈希表统计 nums1 中每个元素出现的次数
    2. 遍历 nums2，如果元素在哈希表中且计数大于 0，则添加到结果中，并减少计数

    时间复杂度: O(n+m)，其中 n 和 m 分别是两个数组的长度
    空间复杂度: O(min(n,m))
    """
    # 使用哈希表统计 nums1 中每个元素出现的次数
    counter = {}
    for num in nums1:
        counter[num] = counter.get(num, 0) + 1

    # 遍历 nums2，找出交集元素
    result = []
    for num in nums2:
        if num in counter and counter[num] > 0:
            result.append(num)
            counter[num] -= 1

    return result

@staticmethod
def keyboard_row(words: List[str]) -> List[str]:
    """
    LeetCode 500. Keyboard Row (键盘行)
    题目来源: https://leetcode.com/problems/keyboard-row/

    题目描述:
    给你一个字符串数组 words，只返回可以使用在美式键盘同一行的字母打印出来的单词。

    美式键盘的三行分别是：
    第一行：'qwertyuiop'
    第二行：'asdfghjkl'
    第三行：'zxcvbnm'

    示例:
    输入：words = ["Hello","Alaska","Dad","Peace"]
    输出：["Alaska","Dad"]
```

```
        算法思路:
        1. 创建三个集合，分别存储键盘的三行字母
        2. 对于每个单词，检查其所有字母是否都在同一行

        时间复杂度: O(n)，其中 n 是所有单词的总长度
        空间复杂度: O(1)，因为键盘行是固定的
        """
        # 创建三个集合，存储键盘的三行字母
        row1 = set("qwertyuiop")
        row2 = set("asdfghjkl")
        row3 = set("zxcvbnm")

        result = []

        for word in words:
            # 转换为小写，便于比较
            w = word.lower()

            # 检查单词的所有字母是否都在同一行
            if all(c in row1 for c in w) or all(c in row2 for c in w) or all(c in row3 for c in
w):
                result.append(word)

        return result


    @staticmethod
    def jewels_and_stones(jewels: str, stones: str) -> int:
        """
        LeetCode 771. Jewels and Stones (宝石与石头)
        题目来源: https://leetcode.com/problems/jewels-and-stones/

        题目描述:
        给定字符串 J 代表石头中宝石的类型，和字符串 S 代表你拥有的石头。
        S 中每个字符代表了一种你拥有的石头的类型，你想知道你拥有的石头中有多少是宝石。
        J 中的字母不重复，J 和 S 中的所有字符都是字母。字母区分大小写，因此"a"和"A"是不同类型的石
头。

        示例:
        输入: J = "aA", S = "aAAbbbb"
        输出: 3

        算法思路:
```

```
        1. 使用集合存储宝石类型，便于快速查找
        2. 遍历石头，统计宝石数量

        时间复杂度: O(J+S)，其中 J 和 S 分别是两个字符串的长度
        空间复杂度: O(J)
        """
        # 使用集合存储宝石类型
        jewel_set = set(jewels)

        # 统计宝石数量
        count = 0
        for stone in stones:
            if stone in jewel_set:
                count += 1

        return count

    @staticmethod
    def longest_consecutive_sequence(nums: List[int]) -> int:
        """
        LeetCode 128. Longest Consecutive Sequence (最长连续序列)
        题目来源: https://leetcode.com/problems/longest-consecutive-sequence/

        题目描述:
        给定一个未排序的整数数组 nums，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。
        请你设计并实现时间复杂度为 O(n) 的算法解决此问题。

        示例:
        输入：nums = [100,4,200,1,3,2]
        输出：4
        解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

        算法思路:
        1. 使用哈希集合存储所有数字，便于 O(1)时间查找
        2. 对于每个数字 x，如果 x-1 不在集合中（说明 x 是序列的起点），则尝试找出以 x 开始的最长序列

        时间复杂度: O(n)，其中 n 是数组长度
        空间复杂度: O(n)
        """
        if not nums:
            return 0
```

```
        # 使用哈希集合存储所有数字
        num_set = set(nums)

        max_length = 0

        # 对于每个数字，检查它是否是序列的起点
        for num in num_set:
            # 如果 num-1 不在集合中，说明 num 是序列的起点
            if num - 1 not in num_set:
                current_num = num
                current_length = 1

                # 尝试找出以 num 开始的最长序列
                while current_num + 1 in num_set:
                    current_num += 1
                    current_length += 1

                # 更新最长序列长度
                max_length = max(max_length, current_length)

        return max_length
```

==========