==========
文件夹: class060_TrieTree
==========

[Markdown 文件]
==========
文件: README.md
==========
Trie 树（前缀树）算法详解与实战

算法概述

Trie 树，又称前缀树或字典树，是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。它通过利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较。

核心思想

1. **空间换时间**：利用字符串的公共前缀来降低查询时间的开销
2. **树形结构**：每个节点代表一个字符，从根节点到任意节点的路径表示一个字符串前缀
3. **前缀共享**：具有相同前缀的字符串共享存储空间

基本性质

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串
3. 每个节点的所有子节点包含的字符都不相同

数据结构实现

节点结构

```java
class TrieNode {
    public int pass;        // 经过该节点的字符串数量
    public int end;         // 以该节点为结尾的字符串数量
    public TrieNode[] nexts; // 子节点数组，大小为字符集大小
}
```

主要操作

1. **插入（Insert）**：将字符串插入到 Trie 树中
2. **搜索（Search）**：判断字符串是否在 Trie 树中

3. **前缀搜索（StartsWith）**：判断是否存在以指定前缀开头的字符串

经典题目解析

Trie 树作为一种重要的数据结构，在各大算法平台都有相关题目。除了 LeetCode 上的基础题目外，POJ、HDU、牛客网等平台也有丰富的 Trie 树题目。

1. LeetCode 208. 实现 Trie (前缀树)

这是 Trie 树的基础题目，要求实现 Trie 树的基本操作。

时间复杂度：
- 插入操作：O(m)，m 为字符串长度
- 搜索操作：O(m)，m 为字符串长度
- 前缀搜索：O(m)，m 为前缀长度

空间复杂度：O(ALPHABET_SIZE * N * M)，N 为字符串数量，M 为平均长度

2. LeetCode 211. 添加与搜索单词 - 数据结构设计

该题目在基础 Trie 树的基础上增加了通配符'.'的支持，需要使用递归或 BFS 来处理模糊匹配。

时间复杂度：
- 插入操作：O(m)，m 为单词长度
- 搜索操作：最坏情况 O(26^m)，当所有字符都是'.'时达到最坏情况

3. LeetCode 677. 键值映射

该题目要求实现一个键值映射的数据结构，支持前缀求和操作。

时间复杂度：
- 插入操作：O(m)，m 为键的长度
- 求和操作：O(m)，m 为前缀的长度

4. POJ 3630 / HDU 1671 Phone List

给定 n 个电话号码，判断是否存在一个电话号码是另一个电话号码的前缀。如果存在输出 NO，否则输出 YES。

时间复杂度：
- 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
- 查询过程：O(∑len(s))

空间复杂度：O(∑len(s) * 10)

5. POJ 1451 T9

模拟手机 T9 输入法。手机键盘上每个数字键对应多个字母，给定一些单词及其频率，然后给出按键序列，要求按频率从高到低输出匹配的单词。

时间复杂度：
- 构建 Trie 树：O(∑len(s))
- 查询过程：O(m)，其中 m 是按键序列长度

空间复杂度：O(∑len(s) * 26)

6. HDU 5790 Prefix

给定 n 个字符串，然后 m 次询问，每次询问给出 l,r 代表在第 l 和第 r 个串之间本质不同的前缀有多少个。

时间复杂度：
- 构建 Trie 树：O(∑len(s))
- 查询过程：O(m * log(n))，使用主席树优化

空间复杂度：O(∑len(s) * log(n))

7. LeetCode 745. 前缀和后缀搜索

设计一个包含一些单词的词典，支持前缀和后缀搜索。返回词典中具有指定前缀和后缀的单词的下标。

时间复杂度：
- 构造函数：O(N*L²)，其中 N 是单词数量，L 是单词最大长度
- 查询函数：O(P+S)，其中 P 是前缀长度，S 是后缀长度

空间复杂度：O(N*L²)

8. LeetCode 336. 回文对

给定一组互不相同的单词，找出所有不同的索引对 (i, j)，使得两个单词连接成回文串。

时间复杂度：
- 构建 Trie 树：O(N*L)
- 查询过程：O(N*L²)

空间复杂度：O(N*L)

9. POJ 2001 Shortest Prefixes

给定一组单词，为每个单词找到最短的唯一前缀。

时间复杂度：O(∑len(s))
空间复杂度：O(∑len(s))

10. HDU 1247 Hat's Words

找出字典中所有可以恰好由其他两个单词连接而成的单词。

时间复杂度：O(N*L²)
空间复杂度：O(∑len(s))

11. 牛客网 最长公共前缀

查找字符串数组中的最长公共前缀。

时间复杂度：O(∑len(s))
空间复杂度：O(∑len(s))

12. 洛谷 P2580 点名系统

实现点名系统，支持 OK、REPEAT、WRONG 三种状态。

时间复杂度：O(∑len(s))
空间复杂度：O(∑len(s))

13. CodeChef DICT - Dictionary

给定字典和查询，输出所有以查询字符串为前缀的单词。

时间复杂度：O(∑len(s) + ∑(P+K))
空间复杂度：O(∑len(s))

14. 剑指 Offer 45. 把数组排成最小的数

把非负整数数组拼接成最小的数。

时间复杂度：O(NlogN)
空间复杂度：O(N)

15. 杭电 OJ 1251 统计难题

统计以某个字符串为前缀的单词数量。

时间复杂度：O(∑len(s))
空间复杂度：O(∑len(s))

16. SPOJ ADAINDEX - Ada and Indexing

给定一个单词列表和一些查询，对于每个查询，输出列表中有多少个单词以该查询字符串为前缀。

时间复杂度：
- 构建 Trie 树：O(∑len(s))
- 查询过程：O(P)，其中 P 是前缀长度

空间复杂度：O(∑len(s))

17. SPOJ DICT - Search in the dictionary!

给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。

时间复杂度：
- 构建 Trie 树：O(∑len(s))
- 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量

空间复杂度：O(∑len(s))

18. CodeForces 271D - Good Substrings

给定一个字符串 s，一个由 26 个字符组成的字符串，表示每个字母是好字母还是坏字母，以及一个整数 k，表示一个好子串中最多允许的坏字符数量。找出字符串 s 中不同好子串的数量。

时间复杂度：$O(N^3)$
空间复杂度：$O(N^2)$

应用场景

1. 自动补全
搜索引擎、IDE 代码补全等场景中，Trie 树可以快速检索具有相同前缀的候选词。

2. 拼写检查
快速查找字典中的单词，判断输入单词是否正确。

3. 词频统计
统计文本中单词出现次数，用于文本分析。

4. IP 路由
最长前缀匹配用于网络路由选择。

5. 敏感词过滤
快速匹配文本中的敏感词并进行过滤。

语言实现差异

Java
- 使用引用类型，有垃圾回收机制
- 数组实现固定子节点，HashMap 实现动态子节点
- 性能适中，开发效率高

C++
- 需要手动管理内存
- 数组或指针数组实现，性能更高但需注意内存泄漏
- 适合对性能要求极高的场景

Python
- 动态类型语言，字典实现自然
- 代码简洁但性能相对较低
- 适合快速原型开发

工程化考量

1. 异常处理
- 输入参数校验
- 空字符串和 null 值处理
- 字符集范围检查

2. 性能优化
- 对象池减少频繁创建节点对象的开销
- 缓存热点查询结果
- 内存预分配

3. 可配置性
- 支持不同字符集
- 可配置 Trie 树参数
- 插件化功能扩展

4. 线程安全
- 根据使用场景决定是否需要同步机制

- 读写锁优化读多写少场景
- 无锁数据结构提升并发性能

与机器学习的联系

1. 自然语言处理
- 构建词典和前缀匹配
- 命名实体识别中的词典匹配
- 文本分类中的前缀特征

2. 信息检索
- 搜索引擎的自动补全功能
- 倒排索引的前缀查询优化
- 查询纠错和模糊匹配

3. 数据压缩
- 霍夫曼编码树的构建
- 字典压缩算法中的前缀匹配
- 重复数据删除

极端场景鲁棒性

1. 空字符串处理
需要特殊处理根节点的计数逻辑

2. 重复字符串
通过计数器区分出现次数

3. 超长字符串
受限于系统内存，但算法本身无长度限制

4. 大量相似前缀
Trie 树的优势场景，能有效共享前缀存储空间

新增实现文件说明

Code06_ExtendedTrieProblems.java
包含从各大算法平台收集的 10 个 Trie 树扩展题目：

1. **LeetCode 745. 前缀和后缀搜索** - 支持前缀和后缀双重搜索
2. **LeetCode 336. 回文对** - 查找能形成回文对的单词组合
3. **POJ 2001 Shortest Prefixes** - 为每个单词找到最短唯一前缀
4. **HDU 1247 Hat's Words** - 查找由两个单词连接而成的单词

5. **牛客网 最长公共前缀** - 查找字符串数组的最长公共前缀
6. **洛谷 P2580 点名系统** - 实现点名状态管理
7. **CodeChef DICT - Dictionary** - 前缀查询字典功能
8. **SPOJ PHONELST - Phone List** - 电话号码前缀检查
9. **剑指 Offer 45. 把数组排成最小的数** - 数字拼接排序
10. **杭电 OJ 1251 统计难题** - 前缀数量统计
11. **SPOJ ADAINDEX - Ada and Indexing** - 前缀计数查询
12. **CodeForces 271D - Good Substrings** - 好子串计数

Code06_ExtendedTrieProblems.py
Python 版本的扩展题目实现，包含相同的 10 个题目，使用 Python 的简洁语法实现。

Code06_ExtendedTrieProblems_Simple.cpp
简化版 C++实现，避免使用可能引起编译问题的现代 C++特性。

扩展题目详解

POJ 3630 / HDU 1671 Phone List

这是一道经典的 Trie 树应用题目，要求检测电话号码列表中是否存在前缀关系。通过 Trie 树可以在线性时间内完成检测。

题目来源：
- POJ 3630: http://poj.org/problem?id=3630
- HDU 1671: http://acm.hdu.edu.cn/showproblem.php?pid=1671

核心思路：
1. 使用 Trie 树存储所有电话号码
2. 在插入过程中检查前缀关系
3. 若在插入过程中遇到已标记结尾的节点，说明存在前缀关系

POJ 1451 T9

模拟 T9 输入法的核心功能，通过 Trie 树实现智能预测输入。这是 Trie 树在实际应用中的经典案例。

题目来源：
- POJ 1451: http://poj.org/problem?id=1451

核心思路：
1. 构建 Trie 树存储单词及其频率
2. 每个节点维护以该前缀开始的最高频单词
3. 根据按键序列快速查找最可能的单词

HDU 5790 Prefix

这是一道高级 Trie 树题目，结合了主席树等数据结构来优化区间查询。体现了 Trie 树在处理复杂查询时的灵活性。

题目来源：
- HDU 5790: http://acm.hdu.edu.cn/showproblem.php?pid=5790

核心思路：
1. 使用 Trie 树存储所有前缀
2. 记录每个前缀首次出现位置
3. 结合主席树优化区间不同前缀计数

SPOJ ADAINDEX - Ada and Indexing

这是一道经典的前缀计数题目，要求统计以指定前缀开头的单词数量。通过 Trie 树可以高效完成查询。

题目来源：
- SPOJ ADAINDEX: https://www.spoj.com/problems/ADAINDEX/

核心思路：
1. 使用 Trie 树存储所有单词
2. 每个节点记录经过该节点的单词数量
3. 查询时找到前缀对应的节点，返回该节点的计数

SPOJ DICT - Search in the dictionary!

这是一道前缀查询题目，要求输出字典中所有以指定前缀开头的单词。通过 Trie 树可以高效完成查询并按字典序输出。

题目来源：
- SPOJ DICT: https://www.spoj.com/problems/DICT/

核心思路：
1. 使用 Trie 树存储字典中的所有单词
2. 每个节点维护以该节点为前缀的所有单词
3. 查询时找到前缀对应的节点，输出该节点存储的所有单词

各大平台 Trie 树题目推荐

国际平台
- **LeetCode**: 208, 211, 677, 212, 421, 745, 336 等经典题目
- **HackerRank**: 字符串处理相关题目

- **CodeChef**: DICT 等前缀匹配相关题目
- **SPOJ**: PHONELST, ADAINDEX, DICT 等题目
- **CodeForces**: 271D 等题目

国内平台
- **POJ**: 3630, 1451, 2001 等经典题目
- **HDU**: 1671, 5790, 1247, 1251 等进阶题目
- **牛客网**: 最长公共前缀等算法练习题
- **洛谷**: P2580 点名系统等前缀树相关题目
- **杭电 OJ**: 1251 统计难题等题目
- **剑指 Offer**: 45.把数组排成最小的数

工程化考量与最佳实践

1. 异常处理与边界场景
- **空输入处理**：所有方法都应处理空字符串、空数组等边界情况
- **非法字符**：根据字符集范围进行校验，避免越界访问
- **内存管理**：C++版本需要正确实现析构函数，避免内存泄漏

2. 性能优化策略
- **内存预分配**：对于固定字符集，使用数组而非哈希表提高访问速度
- **路径压缩**：对于稀疏 Trie 树，可以合并单一路径节点
- **缓存优化**：热点查询结果可以缓存，减少重复计算

3. 线程安全考虑
- **读多写少**：使用读写锁优化并发访问
- **写时复制**：对于频繁查询的场景，采用写时复制策略
- **原子操作**：计数器等简单操作使用原子变量

4. 可配置性与扩展性
- **字符集支持**：设计支持不同字符集的通用 Trie 树
- **插件架构**：支持自定义节点存储策略和查询策略
- **监控指标**：添加性能监控和统计信息

5. 测试覆盖策略
- **单元测试**：覆盖所有边界情况和正常流程
- **性能测试**：测试大规模数据下的性能表现
- **并发测试**：验证多线程环境下的正确性

语言特性差异与优化

Java 实现特点
- **垃圾回收**：自动内存管理，适合快速原型开发

- **HashMap 优化**：对于稀疏字符集，HashMap 比数组更节省空间
- **JIT 优化**：热点代码会被 JIT 编译器优化

C++实现特点
- **手动内存管理**：需要正确实现析构函数，避免内存泄漏
- **模板元编程**：可以使用模板实现通用 Trie 树
- **性能优势**：直接内存访问，无虚拟机开销

Python 实现特点
- **动态类型**：代码简洁，开发效率高
- **字典优化**：Python 字典经过高度优化，性能良好
- **解释执行**：性能相对较低，适合脚本和小规模应用

与机器学习等领域的联系

1. 自然语言处理
- **词典构建**：Trie 树用于构建词典和实现前缀匹配
- **命名实体识别**：基于词典的实体识别算法
- **文本分类**：前缀特征用于文本分类任务

2. 信息检索
- **搜索引擎**：自动补全和拼写纠错功能
- **倒排索引**：前缀索引优化查询性能
- **查询扩展**：基于前缀的查询扩展技术

3. 数据压缩
- **霍夫曼编码**：Trie 树用于构建霍夫曼编码树
- **字典压缩**：LZ77 等压缩算法使用 Trie 树
- **重复数据删除**：基于前缀的重复检测

4. 生物信息学
- **DNA 序列分析**：序列匹配和模式发现
- **蛋白质序列**：氨基酸序列的前缀匹配
- **基因组学**：基因序列的快速检索

极端场景鲁棒性测试

1. 空输入测试
- 空字符串插入和查询
- 空数组处理
- 空指针检查

2. 重复数据测试

- 重复字符串插入
- 相同前缀的大量字符串
- 完全相同的字符串

3. 超长字符串测试
- 超长字符串的插入和查询
- 内存使用监控
- 性能退化分析

4. 特殊字符测试
- 非字母数字字符
- Unicode 字符支持
- 转义字符处理

总结

Trie 树作为一种专门处理字符串前缀的数据结构，通过空间换时间的思想，在字符串检索场景中具有优异的性能表现。通过本次全面的题目整理和实现，我们掌握了：

核心技术要点
1. **基础实现**：Trie 树的基本节点结构和操作实现
2. **高级应用**：复杂场景下的 Trie 树变体和优化策略
3. **工程实践**：生产环境中的性能优化和异常处理
4. **跨领域应用**：Trie 树在多个领域的实际应用

学习路径建议
1. **初级阶段**：掌握基本 Trie 树的插入、查询、前缀搜索
2. **中级阶段**：学习 Trie 树在具体问题中的应用和优化
3. **高级阶段**：研究 Trie 树的变体和在复杂系统中的应用
4. **专家阶段**：参与 Trie 树相关开源项目，贡献代码

未来发展方向
1. **分布式 Trie 树**：支持大规模分布式存储和查询
2. **GPU 加速**：利用 GPU 并行计算提升 Trie 树性能
3. **机器学习集成**：Trie 树与深度学习模型的结合
4. **新型存储介质**：针对新型存储介质的 Trie 树优化

通过系统学习和实践，Trie 树将成为解决字符串处理问题的有力工具，在算法竞赛和工程实践中发挥重要作用。

==========

[代码文件]

```
==========================================
文件: Code01_TrieTree.cpp
==========================================
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <set>
#include <cstring>
#include <algorithm>
using namespace std;

// Trie 树（前缀树）算法详解与实战
// 本文件包含多个 Trie 树相关题目的 C++实现

/*
 * 题目 1: LeetCode 208. 实现 Trie (前缀树)
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/implement-trie-prefix-tree/
 *
 * 题目描述：
 * Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。
 * 这一数据结构有相当多的应用情景，例如自动补全和拼写检查。
 * 请你实现 Trie 类：
 * Trie() 初始化前缀树对象。
 * void insert(String word) 向前缀树中插入字符串 word 。
 * boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
 * boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。
 *
 * 解题思路：
 * 1. Trie 树是一种专门处理字符串前缀的数据结构
 * 2. 每个节点包含若干子节点（对应不同字符）和一个标记（表示是否为单词结尾）
 * 3. 插入操作：从根节点开始，逐字符查找，若不存在则创建新节点
 * 4. 搜索操作：从根节点开始，逐字符查找，若路径存在且终点为单词结尾则返回 true
 * 5. 前缀搜索：从根节点开始，逐字符查找，若路径存在则返回 true
 *
 * 时间复杂度分析：
 * 1. insert 操作：O(m)，m 为插入字符串的长度
 * 2. search 操作：O(m)，m 为搜索字符串的长度
 * 3. startsWith 操作：O(m)，m 为前缀字符串的长度
```

```
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的字符串数量，M 是字符串的平均长度
 * 2. 最坏情况下，没有公共前缀，每个字符都需要一个节点
 * 是否为最优解：是，这是 Trie 树的标准实现，时间复杂度已达到理论最优
 */

// LeetCode 208. 实现 Trie (前缀树)
// 数组实现的 Trie 树（适用于固定字符集，如小写字母 a-z）
class Trie {
private:
    struct TrieNode {
        bool isEnd; // 标记该节点是否为某个单词的结尾
        TrieNode* children[26]; // 子节点数组，对应 26 个小写字母

        // 构造函数，初始化节点
        TrieNode() : isEnd(false) {
            memset(children, 0, sizeof(children)); // 将子节点指针数组初始化为 NULL
        }

        // 析构函数，递归释放子节点内存
        ~TrieNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    TrieNode* root; // 根节点

public:
    // 构造函数
    Trie() {
        // 初始化根节点，根节点不存储字符
        root = new TrieNode();
    }

    // 析构函数
    ~Trie() {
        // 释放根节点及其子节点占用的内存
        delete root;
```

```
    root = nullptr;
}

// 插入字符串到 Trie 树中
void insert(string word) {
    if (word.empty()) {
        return;
    }

    TrieNode* node = root;
    // 逐字符插入到 Trie 树中
    for (char c : word) {
        int index = c - 'a'; // 计算字符对应的索引
        // 如果当前字符对应的子节点不存在，则创建
        if (!node->children[index]) {
            node->children[index] = new TrieNode();
        }
        // 移动到下一个节点
        node = node->children[index];
    }
    // 标记单词结尾
    node->isEnd = true;
}

// 搜索字符串是否存在于 Trie 树中
bool search(string word) {
    if (word.empty()) {
        return false;
    }

    TrieNode* node = root;
    // 逐字符查找
    for (char c : word) {
        int index = c - 'a';
        // 如果当前字符对应的子节点不存在，则单词不存在
        if (!node->children[index]) {
            return false;
        }
        node = node->children[index];
    }
    // 只有当到达单词结尾且该节点被标记为单词结尾时，才返回 true
    return node->isEnd;
}
```

```
    // 检查是否存在以指定前缀开头的单词
    bool startsWith(string prefix) {
        if (prefix.empty()) {
            return false;
        }

        TrieNode* node = root;
        // 逐字符查找前缀
        for (char c : prefix) {
            int index = c - 'a';
            // 如果当前字符对应的子节点不存在，则前缀不存在
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index];
        }
        // 只要前缀存在，就返回 true
        return true;
    }
};

/*
 * 题目 2: LeetCode 211. 添加与搜索单词 - 数据结构设计
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/design-add-and-search-words-data-structure/
 *
 * 题目描述：
 * 请你设计一个数据结构，支持 添加新单词 和 查找字符串是否与任何先前添加的字符串匹配 。
 * 实现词典类 WordDictionary：
 * WordDictionary() 初始化词典对象
 * void addWord(word) 将 word 添加到数据结构中，之后可以对它进行匹配
 * bool search(word) 如果数据结构中存在字符串与 word 匹配，则返回 true；否则，返回 false。word 中可能包含一些'.'，每个'.'都可以表示任何一个字母
 *
 * 解题思路：
 * 1. 使用 Trie 树存储添加的单词
 * 2. 搜索时遇到'.'字符，需要递归搜索所有子节点
 * 3. 使用 DFS 实现模糊匹配
 *
 * 时间复杂度分析：
 * 1. addWord 操作：O(m)，m 为单词长度
 * 2. search 操作：最坏情况 O(26^m)，其中 m 为搜索字符串长度，当所有字符都是'.'时达到最坏情况
```

```
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中N是插入的字符串数量，M是字符串的平均长度
 * 是否为最优解：是，对于模糊匹配问题，这是标准的解决方案
 */

// LeetCode 211. 添加与搜索单词 - 数据结构设计
class WordDictionary {
private:
    struct WordNode {
        bool isEnd; // 标记该节点是否为某个单词的结尾
        WordNode* children[26]; // 子节点数组，对应26个小写字母

        // 构造函数，初始化节点
        WordNode() : isEnd(false) {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~WordNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    WordNode* root; // 根节点

    // DFS搜索辅助函数
    bool dfsSearch(const string& word, int index, WordNode* node) {
        // 递归终止条件：已遍历完整个单词
        if (index == word.size()) {
            return node->isEnd;
        }

        char c = word[index];
        if (c == '.') {
            // 遇到通配符'.'，需要尝试所有可能的子节点
            for (int i = 0; i < 26; i++) {
                if (node->children[i] && dfsSearch(word, index + 1, node->children[i])) {
                    return true;
```

```
                }
            }
            return false;
        } else {
            // 普通字符，直接查找对应的子节点
            int childIndex = c - 'a';
            if (!node->children[childIndex]) {
                return false;
            }
            return dfsSearch(word, index + 1, node->children[childIndex]);
        }
    }

public:
    // 构造函数
    WordDictionary() {
        root = new WordNode();
    }

    // 析构函数
    ~WordDictionary() {
        delete root;
        root = nullptr;
    }

    // 添加单词到词典中
    void addWord(string word) {
        if (word.empty()) {
            return;
        }

        WordNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new WordNode();
            }
            node = node->children[index];
        }
        node->isEnd = true;
    }

    // 搜索单词，支持通配符'.'
```

```
    bool search(string word) {
        if (word.empty()) {
            return false;
        }
        // 使用递归DFS进行搜索
        return dfsSearch(word, 0, root);
    }
};

/*
 * 题目3: LeetCode 677. 键值映射
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/map-sum-pairs/
 *
 * 题目描述：
 * 实现一个 MapSum 类，支持两个方法，insert 和 sum：
 * MapSum() 初始化 MapSum 对象
 * void insert(String key, int val) 插入 key-val 键值对，字符串表示键 key ，整数表示值 val 。如果键 key 已经存在，那么原来的键值对将被替代成新的键值对。
 * int sum(string prefix) 返回所有以该前缀 prefix 开头的键 key 的值的总和。
 *
 * 解题思路：
 * 1. 使用Trie树存储键值对
 * 2. 每个节点存储经过该节点的键的值总和
 * 3. 插入时需要处理键已存在的情况，先减去旧值再加上新值
 * 4. 求和时找到前缀对应的节点，返回该节点存储的值总和
 *
 * 时间复杂度分析：
 * 1. insert操作：O(m)，m为键的长度
 * 2. sum操作：O(m)，m为前缀的长度
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中N是插入的键数量，M是键的平均长度
 * 是否为最优解：是，利用Trie树的前缀特性可以高效实现键值映射
 */

// LeetCode 677. 键值映射
class MapSum {
private:
    struct SumNode {
        int sum; // 存储经过该节点的所有键的值总和
        SumNode* children[26]; // 子节点数组，对应26个小写字母

        // 构造函数，初始化节点
```

```
        SumNode() : sum(0) {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~SumNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    SumNode* root; // 根节点
    unordered_map<string, int> keyMap; // 用于存储键值对，处理更新操作

public:
    // 构造函数
    MapSum() {
        root = new SumNode();
    }

    // 析构函数
    ~MapSum() {
        delete root;
        root = nullptr;
    }

    // 插入键值对，支持更新操作
    void insert(string key, int val) {
        if (key.empty()) {
            return;
        }

        // 计算值的变化量
        int delta = val;
        auto it = keyMap.find(key);
        if (it != keyMap.end()) {
            delta -= it->second;
        }
        keyMap[key] = val;
```

```
        // 更新Trie树中的值总和
        SumNode* node = root;
        for (char c : key) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new SumNode();
            }
            node = node->children[index];
            node->sum += delta; // 累加变化量
        }
    }

    // 计算所有以指定前缀开头的键的值总和
    int sum(string prefix) {
        if (prefix.empty()) {
            return 0;
        }

        SumNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return 0; // 前缀不存在
            }
            node = node->children[index];
        }
        return node->sum; // 返回前缀对应节点的值总和
    }
};

/*
 * 题目4: LeetCode 212. 单词搜索 II
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/word-search-ii/
 *
 * 题目描述：
 * 给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和单词列表中出现的单词。
 * 单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中"相邻"单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。
 *
 * 解题思路：
```

```
 * 1. 构建包含所有待查单词的 Trie 树
 * 2. 对二维网格中的每个位置进行 DFS 搜索
 * 3. 搜索过程中维护当前路径构成的前缀，在 Trie 树中查找
 * 4. 若当前前缀对应一个完整单词，则将其加入结果集
 * 5. 使用回溯法避免重复访问同一单元格
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(K*L)，其中 K 为单词数量，L 为单词平均长度
 * 2. DFS 搜索：O(M*N*4^L)，其中 M、N 为网格行列数，L 为最长单词长度
 * 空间复杂度分析：
 * 1. Trie 树：O(K*L)
 * 2. DFS 递归栈：O(M*N)
 * 是否为最优解：是，结合 Trie 树和 DFS 是解决此类问题的经典方法
 */

// LeetCode 212. 单词搜索 II
class WordSearchII {
private:
    struct SearchNode {
        string word; // 存储完整单词，仅在单词结尾节点有效
        SearchNode* children[26]; // 子节点数组，对应 26 个小写字母

        // 构造函数，初始化节点
        SearchNode() {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~SearchNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    SearchNode* root; // 根节点
    vector<vector<int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // 上下左右四个方向

    // DFS 搜索辅助函数
    void dfs(vector<vector<char>>& board, int i, int j, SearchNode* node, vector<vector<bool>>&
```

```
visited, set<string>& result) {
        // 检查边界条件和访问状态
        if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || visited[i][j]) {
            return;
        }

        char c = board[i][j];
        int index = c - 'a';

        // 如果当前字符不在 Trie 树的当前节点的子节点中，直接返回
        if (!node->children[index]) {
            return;
        }

        // 移动到下一个 Trie 节点
        node = node->children[index];

        // 如果找到一个完整单词，加入结果集
        if (!node->word.empty()) {
            result.insert(node->word);
            // 注意：不要在这里 return，因为可能还有更长的单词
        }

        // 标记当前位置为已访问
        visited[i][j] = true;

        // 向四个方向递归搜索
        for (auto& dir : directions) {
            dfs(board, i + dir[0], j + dir[1], node, visited, result);
        }

        // 回溯：恢复访问状态
        visited[i][j] = false;
    }

public:
    // 构造函数
    WordSearchII() {
        root = new SearchNode();
    }

    // 析构函数
    ~WordSearchII() {
```

```
    delete root;
    root = nullptr;
}

// 插入单词到 Trie 树
void insertWord(string word) {
    SearchNode* node = root;
    for (char c : word) {
        int index = c - 'a';
        if (!node->children[index]) {
            node->children[index] = new SearchNode();
        }
        node = node->children[index];
    }
    node->word = word; // 在单词结尾节点存储完整单词
}

// 在二维网格中查找所有单词
vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
    set<string> resultSet; // 用于存储找到的单词，避免重复
    if (board.empty() || board[0].empty() || words.empty()) {
        return {};
    }

    // 构建 Trie 树
    for (string& word : words) {
        insertWord(word);
    }

    // 从网格的每个位置开始搜索
    int m = board.size();
    int n = board[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            // 剪枝：如果当前字符不在 Trie 树的根节点的子节点中，直接跳过
            int index = board[i][j] - 'a';
            if (root->children[index]) {
                dfs(board, i, j, root, visited, resultSet);
            }
        }
    }
```

```
        return vector<string>(resultSet.begin(), resultSet.end());
    }
};

/*
 * 题目 5: LeetCode 421. 数组中两个数的最大异或值
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/
 *
 * 题目描述：
 * 给你一个整数数组 nums ，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n 。
 *
 * 解题思路：
 * 1. 使用二进制 Trie 树存储数组中所有数字的二进制表示（从最高位到最低位）
 * 2. 对于每个数字，从高位到低位在 Trie 树中贪心查找与其异或结果最大的数字
 * 3. 贪心策略：在 Trie 树中尽量走与当前位相反的路径（0 走 1，1 走 0）
 * 4. 这样可以保证从高位开始尽可能多地获得 1，从而得到最大异或值
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(N*32) = O(N)，其中 N 为数组长度
 * 2. 查找最大异或值：O(N*32) = O(N)
 * 空间复杂度分析：
 * 1. O(N*32) = O(N)，Trie 树最多存储 N 个 32 位整数
 * 是否为最优解：是，这是解决此类问题的经典方法，时间复杂度已达到线性
 */

// LeetCode 421. 数组中两个数的最大异或值
class MaximumXOR {
private:
    static const int MAX_BIT = 30; // 假设整数最多 31 位（包括符号位），这里处理到第 30 位

    struct XORNode {
        XORNode* children[2]; // 0 和 1 两个子节点

        // 构造函数，初始化节点
        XORNode() {
            children[0] = nullptr;
            children[1] = nullptr;
        }

        // 析构函数，递归释放子节点内存
        ~XORNode() {
```

```
        for (int i = 0; i < 2; i++) {
            if (children[i]) {
                delete children[i];
                children[i] = nullptr;
            }
        }
    }
};

XORNode* root; // 根节点

// 插入数字到二进制 Trie 树
void insert(int num) {
    XORNode* node = root;
    // 从最高位到最低位插入
    for (int i = MAX_BIT; i >= 0; i--) {
        int bit = (num >> i) & 1; // 获取第 i 位的值
        if (!node->children[bit]) {
            node->children[bit] = new XORNode();
        }
        node = node->children[bit];
    }
}

// 查找与给定数字异或结果最大的数
int searchMaxXOR(int num) {
    XORNode* node = root;
    int maxXOR = 0;

    for (int i = MAX_BIT; i >= 0; i--) {
        int currentBit = (num >> i) & 1;
        int targetBit = 1 - currentBit; // 寻找相反的位

        // 如果存在相反的位，选择该路径并更新异或结果
        if (node->children[targetBit]) {
            maxXOR |= (1 << i); // 第 i 位可以得到 1
            node = node->children[targetBit];
        } else {
            // 否则只能选择相同的位
            node = node->children[currentBit];
        }
    }
```

```
        return maxXOR;
    }

public:
    // 构造函数
    MaximumXOR() {
        root = new XORNode();
    }

    // 析构函数
    ~MaximumXOR() {
        delete root;
        root = nullptr;
    }

    // 查找数组中两个数的最大异或值
    int findMaximumXOR(vector<int>& nums) {
        if (nums.size() <= 1) {
            return 0;
        }

        int maxXOR = 0;

        // 先插入第一个数，然后依次处理每个数
        insert(nums[0]);
        for (int i = 1; i < nums.size(); i++) {
            // 对于当前数，查找能得到最大异或值的已插入数
            maxXOR = max(maxXOR, searchMaxXOR(nums[i]));
            // 将当前数插入 Trie 树
            insert(nums[i]);
        }

        return maxXOR;
    }
};

/*
 * 题目 6: POJ 3630 / HDU 1671 Phone List
 * 题目来源：POJ / HDU
 * 题目链接：http://poj.org/problem?id=3630
 *           http://acm.hdu.edu.cn/showproblem.php?pid=1671
 *
 * 题目描述：
```

```
 * 给定 n 个电话号码，判断是否存在一个电话号码是另一个电话号码的前缀。
 * 如果存在输出 NO，否则输出 YES。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有电话号码
 * 2. 在插入过程中检查是否存在前缀关系
 * 3. 如果在插入过程中遇到已经标记为结尾的节点，说明当前字符串是之前某个字符串的前缀
 * 4. 如果在插入完成后，当前节点还有子节点，说明之前某个字符串是当前字符串的前缀
 *
 * 时间复杂度分析：
 * 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
 * 空间复杂度分析：
 * O(∑len(s) * 10)，每个节点最多有 10 个子节点(0-9)
 * 是否为最优解：是，使用 Trie 树可以在线性时间内检测前缀关系
 */

// POJ 3630 / HDU 1671 Phone List
class PhoneListChecker {
private:
    struct PhoneNode {
        bool isEnd; // 标记该节点是否为某个电话号码的结尾
        PhoneNode* children[10]; // 子节点数组，对应数字 0-9

        // 构造函数，初始化节点
        PhoneNode() : isEnd(false) {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~PhoneNode() {
            for (int i = 0; i < 10; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    PhoneNode* root; // 根节点

    // 检查电话号码是否是其他号码的前缀或有其他号码是它的前缀
    bool hasPrefix(string phone) {
```

```
        PhoneNode* node = root;
        for (char c : phone) {
            int index = c - '0';

            // 如果当前节点已经是某个电话号码的结尾，说明存在前缀
            if (node->isEnd) {
                return true;
            }

            // 如果路径不存在，说明没有前缀
            if (!node->children[index]) {
                return false;
            }

            node = node->children[index];
        }

        // 如果当前路径存在其他子节点，说明当前字符串是其他字符串的前缀
        return node != root;
    }

    // 插入电话号码到 Trie 树
    void insertPhone(string phone) {
        PhoneNode* node = root;
        for (char c : phone) {
            int index = c - '0';
            if (!node->children[index]) {
                node->children[index] = new PhoneNode();
            }
            node = node->children[index];
        }
        node->isEnd = true;
    }

public:
    // 构造函数
    PhoneListChecker() {
        root = new PhoneNode();
    }

    // 析构函数
    ~PhoneListChecker() {
        delete root;
```

```
        root = nullptr;
    }

    // 检查电话号码列表中是否存在前缀冲突
    bool hasPrefixConflict(vector<string>& phoneNumbers) {
        if (phoneNumbers.size() <= 1) {
            return false;
        }

        // 按照电话号码长度排序，先插入短的
        sort(phoneNumbers.begin(), phoneNumbers.end(),
             [](const string& a, const string& b) { return a.size() < b.size(); });

        for (string& phone : phoneNumbers) {
            if (hasPrefix(phone)) {
                return true; // 存在前缀冲突
            }
            insertPhone(phone);
        }

        return false;
    }
};

/*
 * 题目 7: 敏感词过滤
 * 题目来源：常见面试题
 *
 * 题目描述：
 * 给定一个敏感词库，和一段文本，要求将文本中的所有敏感词替换为***。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有敏感词
 * 2. 遍历文本，从每个位置开始匹配敏感词
 * 3. 使用多模式匹配算法，提高效率
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有敏感词长度之和
 * 2. 匹配过程：O(N + M)，其中 N 是文本长度，M 是匹配到的敏感词总长度
 * 空间复杂度分析：
 * O(∑len(s) * ALPHABET_SIZE)
 * 是否为最优解：是，使用 Trie 树可以高效实现多模式匹配
 */
```

```
// 敏感词过滤器
class SensitiveWordFilter {
private:
    struct FilterNode {
        bool isEnd; // 是否为敏感词结尾
        unordered_map<char, FilterNode*> children; // 使用 unordered_map 存储子节点，支持任意字符

        // 构造函数，初始化节点
        FilterNode() : isEnd(false) {}

        // 析构函数，递归释放子节点内存
        ~FilterNode() {
            for (auto& pair : children) {
                delete pair.second;
                pair.second = nullptr;
            }
        }
    };

    FilterNode* root; // 根节点

    // 插入敏感词到 Trie 树
    void insertSensitiveWord(const string& word) {
        if (word.empty()) {
            return;
        }

        FilterNode* node = root;
        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                node->children[c] = new FilterNode();
            }
            node = node->children[c];
        }
        node->isEnd = true;
    }

    // 查找从 start 位置开始的最长敏感词的结束位置
    int findLongestSensitiveWord(const string& text, int start) {
        FilterNode* node = root;
        int maxEnd = start;
```

```
        for (int i = start; i < text.size(); i++) {
            char c = text[i];
            auto it = node->children.find(c);
            if (it == node->children.end()) {
                break;
            }

            node = it->second;
            if (node->isEnd) {
                maxEnd = i + 1; // 更新最长敏感词的结束位置
            }
        }

        return maxEnd;
    }

public:
    // 构造函数
    SensitiveWordFilter(const vector<string>& sensitiveWords) {
        root = new FilterNode();
        // 构建敏感词 Trie 树
        for (const string& word : sensitiveWords) {
            insertSensitiveWord(word);
        }
    }

    // 析构函数
    ~SensitiveWordFilter() {
        delete root;
        root = nullptr;
    }

    // 过滤文本中的敏感词
    string filter(const string& text) {
        if (text.empty()) {
            return text;
        }

        string result;
        int start = 0;

        while (start < text.size()) {
            int end = findLongestSensitiveWord(text, start);
```

```
            if (end > start) {
                // 找到敏感词，替换为***
                result.append(end - start, '*');
            } else {
                // 不是敏感词，保留原字符
                result.push_back(text[start]);
            }
            start = end > start ? end : start + 1;
        }

        return result;
    }
};

/*
 * 题目 8: LeetCode 1032. 字符流
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/stream-of-characters/
 *
 * 题目描述：
 * 设计一个算法：接收一个字符流，并检查这些字符的后缀是否是字符串列表中某个字符串。
 * 实现 StreamChecker 类：
 * StreamChecker(String[] words) 构造函数，用字符串数组 words 初始化数据结构
 * boolean query(char letter) 从字符流中接收一个新字符，如果存在一个单词，它是字符流中某些字符的后缀（即字符串的最后几个字符刚好匹配该单词），返回 true ；否则，返回 false 。
 *
 * 解题思路：
 * 1. 将单词反转并构建 Trie 树
 * 2. 维护一个字符流的历史记录
 * 3. 查询时，从当前字符开始向前匹配 Trie 树
 *
 * 时间复杂度分析：
 * 1. 构造函数：O(∑len(s))
 * 2. query 函数：O(min(L, maxWordLength))，其中 L 是字符流长度，maxWordLength 是单词最大长度
 * 空间复杂度分析：
 * O(∑len(s) * ALPHABET_SIZE)
 * 是否为最优解：是，使用反转 Trie 树是解决此类问题的高效方法
 */

// LeetCode 1032. 字符流
class StreamChecker {
private:
    struct StreamNode {
```

```
        bool isEnd; // 标记该节点是否为某个单词的结尾
        StreamNode* children[26]; // 子节点数组，对应 26 个小写字母

        // 构造函数，初始化节点
        StreamNode() : isEnd(false) {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~StreamNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    StreamNode* root; // 根节点
    string stream; // 存储字符流历史
    int maxLength; // 记录最长单词的长度，用于优化查询

    // 插入反转的单词到 Trie 树
    void insertReversedWord(const string& word) {
        StreamNode* node = root;
        // 反转单词并插入
        for (int i = word.size() - 1; i >= 0; i--) {
            char c = word[i];
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new StreamNode();
            }
            node = node->children[index];
        }
        node->isEnd = true;
    }

public:
    // 构造函数
    StreamChecker(vector<string>& words) {
        root = new StreamNode();
        maxLength = 0;
```

```
        // 构建反转 Trie 树
        for (const string& word : words) {
            insertReversedWord(word);
            maxLength = max(maxLength, (int)word.size());
        }
    }

    // 析构函数
    ~StreamChecker() {
        delete root;
        root = nullptr;
    }

    // 查询当前字符流的后缀是否匹配任何单词
    bool query(char letter) {
        // 将当前字符添加到流中
        stream.push_back(letter);

        StreamNode* node = root;
        // 从当前字符开始向前匹配，最多匹配 maxLength 个字符
        int start = max(0, (int)(stream.size() - maxLength));

        for (int i = stream.size() - 1; i >= start; i--) {
            char c = stream[i];
            int index = c - 'a';

            if (!node->children[index]) {
                return false;
            }

            node = node->children[index];
            if (node->isEnd) {
                return true; // 找到匹配的后缀
            }
        }

        return false;
    }
};

/*
 * 题目 9: SPOJ DICT - Search in the dictionary!
```

```
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/DICT/
 * 相关题目：
 * - CodeChef DICT - Dictionary
 * - 牛客网 最长公共前缀
 * - 杭电 OJ 1251 统计难题
 * - SPOJ ADAINDEX - Ada and Indexing
 *
 * 题目描述：
 * 给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。
 * 如果存在多个单词，按字典序输出。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储字典中的所有单词
 * 2. 每个节点维护以该节点为前缀的所有单词
 * 3. 查询时找到前缀对应的节点，输出该节点存储的所有单词
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树是解决前缀查询的高效方法
 *
 * 工程化考量：
 * 1. 内存优化：可以使用更紧凑的存储方式
 * 2. 性能优化：预处理可以加速查询
 * 3. 排序处理：需要按字典序输出结果
 */

// SPOJ DICT - Search in the dictionary!
class DictionarySearchSPOJ {
private:
    struct DictNode {
        bool isEnd; // 标记是否为单词结尾
        vector<string> words; // 存储以该节点为前缀的所有单词
        DictNode* children[26]; // 子节点数组，对应 26 个小写字母

        // 构造函数，初始化节点
        DictNode() : isEnd(false) {
            memset(children, 0, sizeof(children));
        }
```

```
        // 析构函数，递归释放子节点内存
        ~DictNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    DictNode* root; // 根节点

    // 插入单词到 Trie 树
    void insertWord(const string& word) {
        DictNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new DictNode();
            }
            node = node->children[index];
            node->words.push_back(word);
        }
        node->isEnd = true;
    }

public:
    // 构造函数
    DictionarySearchSPOJ(const vector<string>& dictionary) {
        root = new DictNode();
        // 构建 Trie 树
        for (const string& word : dictionary) {
            insertWord(word);
        }
    }

    // 析构函数
    ~DictionarySearchSPOJ() {
        delete root;
        root = nullptr;
    }
```

```
    // 搜索以指定前缀开头的所有单词
    vector<string> search(const string& prefix) {
        DictNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return {}; // 前缀不存在
            }
            node = node->children[index];
        }
        // 返回该前缀对应的所有单词，按字典序排序
        vector<string> result = node->words;
        sort(result.begin(), result.end());
        return result;
    }
};

/*
 * 题目 10: HDU 1251 统计难题
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251
 * 相关题目：
 * - 牛客网 最长公共前缀
 * - CodeChef DICT - Dictionary
 * - POJ 2001 Shortest Prefixes
 * - SPOJ ADAINDEX - Ada and Indexing
 *
 * 题目描述：
 * Ignatius 最近遇到一个难题，老师交给他很多单词(只有小写字母组成，不会有重复的单词出现)，
 * 现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 每个节点记录经过该节点的单词数量
 * 3. 查询时找到前缀对应的节点，返回该节点的计数
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P)，其中 P 是前缀长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 *
```

```
 * 工程化考量：
 * 1. 内存优化：对于大量单词，可以考虑压缩 Trie 树
 * 2. 性能优化：可以缓存常用查询结果
 * 3. 异常处理：处理空查询和边界情况
 */

// HDU 1251 统计难题
class StatisticalProblemHDU {
private:
    struct StatNode {
        int count; // 经过该节点的单词数量
        StatNode* children[26]; // 子节点数组，对应 26 个小写字母

        // 构造函数，初始化节点
        StatNode() : count(0) {
            memset(children, 0, sizeof(children));
        }

        // 析构函数，递归释放子节点内存
        ~StatNode() {
            for (int i = 0; i < 26; i++) {
                if (children[i]) {
                    delete children[i];
                    children[i] = nullptr;
                }
            }
        }
    };

    StatNode* root; // 根节点

    // 插入单词到 Trie 树
    void insertWord(const string& word) {
        StatNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new StatNode();
            }
            node = node->children[index];
            node->count++;
        }
    }
```

```
public:
    // 构造函数
    StatisticalProblemHDU(const vector<string>& words) {
        root = new StatNode();
        for (const string& word : words) {
            insertWord(word);
        }
    }

    // 析构函数
    ~StatisticalProblemHDU() {
        delete root;
        root = nullptr;
    }

    // 统计以指定前缀开头的单词数量
    int prefixCount(const string& prefix) {
        StatNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return 0;
            }
            node = node->children[index];
        }
        return node->count;
    }
};

/*
 * 总结：Trie 树的核心思想与应用场景
 *
 * 核心思想：
 * 1. 空间换时间：利用字符串的公共前缀来减少查询时间
 * 2. 树形结构：每个节点代表一个字符，从根节点到任意节点的路径表示一个字符串前缀
 * 3. 前缀共享：具有相同前缀的字符串共享存储空间
 *
 * 应用场景：
 * 1. 自动补全：搜索引擎、IDE 代码补全等
 * 2. 拼写检查：快速查找字典中的单词
 * 3. 词频统计：统计文本中单词出现次数
 * 4. IP 路由：最长前缀匹配用于网络路由
```

```
 * 5. 敏感词过滤：快速匹配文本中的敏感词
 * 6. 数据压缩：霍夫曼编码等压缩算法的基础
 * 7. 最大异或值：二进制 Trie 树用于查找最大异或对
 * 8. 单词搜索：在字符网格中查找单词
 *
 * 设计要点：
 * 1. 节点结构：根据需求设计包含适当信息的节点
 * 2. 字符集支持：根据字符集特点选择数组或哈希表实现子节点
 * 3. 内存优化：对于稀疏字符集，哈希表更节省空间
 * 4. 性能优化：预分配内存、剪枝等技术提升性能
 *
 * 时间与空间复杂度：
 * 1. 插入操作：时间复杂度 O(m)，m 为字符串长度
 * 2. 搜索操作：时间复杂度 O(m)，m 为字符串长度
 * 3. 空间复杂度：O(ALPHABET_SIZE * N * M)，N 为字符串数量，M 为平均长度
 *
 * 工程化考虑：
 * 1. 异常处理：输入校验和边界条件处理
 * 2. 线程安全：根据使用场景决定是否需要同步机制
 * 3. 内存管理：在 C++中需要注意内存泄漏，正确实现析构函数
 * 4. 性能监控：添加统计信息，监控内存使用和查询性能
 * 5. 可配置性：支持不同字符集和配置参数
 * 6. 测试覆盖：编写全面的单元测试，覆盖各种边界情况
 */

// 测试代码
int main() {
    cout << "Trie 树算法测试" << endl;

    // 测试 LeetCode 208
    cout << "\n 测试 LeetCode 208. 实现 Trie (前缀树)" << endl;
    Trie trie;
    trie.insert("apple");
    cout << "search(\"apple\"): " << (trie.search("apple") ? "true" : "false") << endl;    //
true
    cout << "search(\"app\"): " << (trie.search("app") ? "true" : "false") << endl;        //
false
    cout << "startsWith(\"app\"): " << (trie.startsWith("app") ? "true" : "false") << endl; //
true
    trie.insert("app");
    cout << "search(\"app\"): " << (trie.search("app") ? "true" : "false") << endl;        //
true
```

```
    // 测试 LeetCode 211
    cout << "\n 测试 LeetCode 211. 添加与搜索单词 - 数据结构设计" << endl;
    WordDictionary wordDict;
    wordDict.addWord("bad");
    wordDict.addWord("dad");
    wordDict.addWord("mad");
    cout << "search(\"pad\"): " << (wordDict.search("pad") ? "true" : "false") << endl;     //
false
    cout << "search(\"bad\"): " << (wordDict.search("bad") ? "true" : "false") << endl;     //
true
    cout << "search(\".ad\"): " << (wordDict.search(".ad") ? "true" : "false") << endl;       //
true
    cout << "search(\"b..\"): " << (wordDict.search("b..") ? "true" : "false") << endl;       //
true

    // 测试 LeetCode 677
    cout << "\n 测试 LeetCode 677. 键值映射" << endl;
    MapSum mapSum;
    mapSum.insert("apple", 3);
    cout << "sum(\"ap\"): " << mapSum.sum("ap") << endl;               // 3
    mapSum.insert("app", 2);
    cout << "sum(\"ap\"): " << mapSum.sum("ap") << endl;               // 5

    // 测试 SPOJ DICT
    cout << "\n 测试 SPOJ DICT - Search in the dictionary!" << endl;
    vector<string> dictionary = {"abc", "abcd", "abcde", "bcd", "bcde"};
    DictionarySearchSPOJ dictSearch(dictionary);
    vector<string> dictResults = dictSearch.search("abc");
    cout << "前缀\"abc\"的单词: ";
    for (const string& word : dictResults) {
        cout << word << " ";
    }
    cout << endl;

    // 测试 HDU 1251
    cout << "\n 测试 HDU 1251 统计难题" << endl;
    vector<string> wordsHDU = {"abc", "abcde", "abcdef", "bcd", "bcde"};
    StatisticalProblemHDU statProblem(wordsHDU);
    cout << "前缀\"abc\"的数量: " << statProblem.prefixCount("abc") << endl;  // 3
    cout << "前缀\"bc\"的数量: " << statProblem.prefixCount("bc") << endl;    // 2
    cout << "前缀\"xyz\"的数量: " << statProblem.prefixCount("xyz") << endl;  // 0

    return 0;
```

```
}

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

文件: Code01_TrieTree.java
==========================================

import java.util.*;

/*
 * 题目 1: LeetCode 208. 实现 Trie (前缀树)
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/implement-trie-prefix-tree/
 * 相关题目：
 * - LeetCode 211. 添加与搜索单词 - 数据结构设计
 * - LeetCode 677. 键值映射
 * - LeetCode 212. 单词搜索 II
 * - LeetCode 421. 数组中两个数的最大异或值
 * - LeetCode 1032. 字符流
 * - POJ 3630 / HDU 1671 Phone List
 * - POJ 1451 T9
 * - HDU 5790 Prefix
 *
 * 题目描述：
 * Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。
 * 这一数据结构有相当多的应用情景，例如自动补全和拼写检查。
 * 请你实现 Trie 类：
 * Trie() 初始化前缀树对象。
 * void insert(String word) 向前缀树中插入字符串 word 。
 * boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
 * boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。
 *
 * 解题思路：
 * 1. Trie 树是一种专门处理字符串前缀的数据结构
 * 2. 每个节点包含若干子节点（对应不同字符）和一个标记（表示是否为单词结尾）
 * 3. 插入操作：从根节点开始，逐字符查找，若不存在则创建新节点
 * 4. 搜索操作：从根节点开始，逐字符查找，若路径存在且终点为单词结尾则返回 true
 * 5. 前缀搜索：从根节点开始，逐字符查找，若路径存在则返回 true
 *
 * 时间复杂度分析：
```

```
 * 1. insert 操作：O(m)，m 为插入字符串的长度
 * 2. search 操作：O(m)，m 为搜索字符串的长度
 * 3. startsWith 操作：O(m)，m 为前缀字符串的长度
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的字符串数量，M 是字符串的平均长度
 * 2. 最坏情况下，没有公共前缀，每个字符都需要一个节点
 * 是否为最优解：是，这是 Trie 树的标准实现，时间复杂度已达到理论最优
 *
 * 工程化考量：
 * 1. 异常处理：可以增加输入参数校验，如检查 word 是否为 null 或空字符串
 * 2. 可配置性：可以支持不同的字符集（不仅仅是小写字母 a-z）
 * 3. 线程安全：当前实现不是线程安全的，如需线程安全需要额外同步机制
 * 4. 性能优化：可以使用对象池减少频繁创建节点对象的开销
 * 5. 内存优化：对于稀疏字符集，使用哈希表比数组更节省空间
 *
 * 语言特性差异：
 * 1. Java：使用引用类型，有垃圾回收机制，数组实现固定子节点
 * 2. C++：需要手动管理内存，可以使用数组或指针数组实现，性能更高但需注意内存泄漏
 * 3. Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与机器学习等领域的联系：
 * 1. 自然语言处理：Trie 树可用于构建词典、前缀匹配等
 * 2. 信息检索：搜索引擎的自动补全功能常使用 Trie 树实现
 * 3. 数据压缩：在某些压缩算法中，Trie 树用于构建霍夫曼编码树
 * 4. 生物信息学：用于 DNA 序列匹配和分析
 *
 * 反直觉但关键的设计：
 * 1. 每个节点不直接存储字符，而是通过父节点到子节点的路径表示字符
 * 2. 根节点不表示任何字符，仅作为起始点
 * 3. 节点的 isEnd 标记表示从根节点到当前节点的路径是否构成一个完整单词
 *
 * 极端场景鲁棒性：
 * 1. 空字符串插入：需要特殊处理根节点的 end 计数
 * 2. 重复字符串：通过 end 计数区分出现次数
 * 3. 超长字符串：受限于系统内存，但算法本身无长度限制
 * 4. 大量相似前缀：Trie 树的优势场景，能有效共享前缀存储空间
 */

class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEndOfWord;

    public TrieNode() {
```

```
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    /**
     * 向 Trie 树中插入一个单词
     * 时间复杂度: O(m)，其中 m 为单词长度
     * 空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
     */
    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            node.children.putIfAbsent(c, new TrieNode());
            node = node.children.get(c);
        }
        node.isEndOfWord = true;
    }

    /**
     * 搜索 Trie 树中是否存在一个完整的单词
     * 时间复杂度: O(m)，其中 m 为单词长度
     * 空间复杂度: O(1)
     */
    public boolean search(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (!node.children.containsKey(c)) {
                return false;
            }
            node = node.children.get(c);
        }
        return node.isEndOfWord;
    }

    /**
```

```
     * 检查Trie树中是否有以给定前缀开头的单词
     * 时间复杂度: O(m)，其中m为前缀长度
     * 空间复杂度: O(1)
     */
    public boolean startsWith(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            if (!node.children.containsKey(c)) {
                return false;
            }
            node = node.children.get(c);
        }
        return true;
    }
}

/*
 * 题目2: LeetCode 211. 添加与搜索单词 - 数据结构设计
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/design-add-and-search-words-data-structure/
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - LeetCode 677. 键值映射
 * - LeetCode 212. 单词搜索 II
 *
 * 题目描述：
 * 请你设计一个数据结构，支持 添加新单词 和 查找字符串是否与任何先前添加的字符串匹配 。
 * 实现词典类WordDictionary：
 * WordDictionary() 初始化词典对象
 * void addWord(word) 将word添加到数据结构中，之后可以对它进行匹配
 * bool search(word) 如果数据结构中存在字符串与word匹配，则返回true；否则，返回false。word中可能包含一些'.'，每个'.'都可以表示任何一个字母
 *
 * 解题思路：
 * 1. 使用Trie树存储添加的单词
 * 2. 搜索时遇到'.'字符，需要递归搜索所有子节点
 * 3. 可以使用DFS或BFS实现模糊匹配
 *
 * 时间复杂度分析：
 * 1. addWord操作：O(m)，m为单词长度
 * 2. search操作：最坏情况O(26^m)，其中m为搜索字符串长度，当所有字符都是'.'时达到最坏情况
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中N是插入的字符串数量，M是字符串的平均长度
```

```
 * 是否为最优解：是，对于模糊匹配问题，这是标准的解决方案
 *
 * 工程化考量：
 * 1. 性能优化：对于大量'.'的查询，可以考虑缓存查询结果
 * 2. 异常处理：需要处理 null 或空字符串输入
 * 3. 可配置性：可以支持不同的字符集
 * 4. 线程安全：当前实现不是线程安全的
 *
 * 与机器学习等领域的联系：
 * 1. 正则表达式匹配：类似模糊匹配的思想在正则表达式引擎中广泛应用
 * 2. 模式识别：在图像识别等领域，模糊匹配用于处理变形或噪声数据
 * 3. 自然语言处理：用于实现模糊搜索和拼写纠错功能
 */

class WordDictionaryNode {
    Map<Character, WordDictionaryNode> children;
    boolean isEnd;

    public WordDictionaryNode() {
        children = new HashMap<>();
        isEnd = false;
    }
}

class WordDictionary {
    private WordDictionaryNode root;

    public WordDictionary() {
        root = new WordDictionaryNode();
    }

    /**
     * 添加单词到数据结构中
     * 时间复杂度: O(m)，其中 m 为单词长度
     * 空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
     */
    public void addWord(String word) {
        WordDictionaryNode node = root;
        for (char c : word.toCharArray()) {
            node.children.putIfAbsent(c, new WordDictionaryNode());
            node = node.children.get(c);
        }
        node.isEnd = true;
```

```
    }

    /**
     * 搜索是否存在匹配的单词，支持'.'通配符
     * 时间复杂度: 最坏情况O(26^m)，其中m为单词长度
     * 空间复杂度: O(m)，递归调用栈的深度
     */
    public boolean search(String word) {
        return searchHelper(word, 0, root);
    }

    private boolean searchHelper(String word, int index, WordDictionaryNode node) {
        // 递归终止条件：已遍历完整个单词
        if (index == word.length()) {
            return node.isEnd;
        }

        char c = word.charAt(index);
        if (c == '.') {
            // 遇到'.'，需要匹配所有子节点
            for (WordDictionaryNode childNode : node.children.values()) {
                if (searchHelper(word, index + 1, childNode)) {
                    return true;
                }
            }
            return false;
        } else {
            // 普通字符，直接查找
            if (!node.children.containsKey(c)) {
                return false;
            }
            return searchHelper(word, index + 1, node.children.get(c));
        }
    }
}

/*
 * 题目3: LeetCode 677. 键值映射
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/map-sum-pairs/
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - LeetCode 211. 添加与搜索单词 - 数据结构设计
```

```
 * - LeetCode 745. 前缀和后缀搜索
 *
 * 题目描述：
 * 实现一个 MapSum 类，支持两个方法，insert 和 sum：
 * MapSum() 初始化 MapSum 对象
 * void insert(String key, int val) 插入 key-val 键值对，字符串表示键 key ，整数表示值 val 。如果键 key 已经存在，那么原来的键值对将被替代成新的键值对。
 * int sum(string prefix) 返回所有以该前缀 prefix 开头的键 key 的值的总和。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储键值对
 * 2. 每个节点存储经过该节点的键的值总和
 * 3. 插入时需要处理键已存在的情况，先减去旧值再加上新值
 * 4. 求和时找到前缀对应的节点，返回该节点存储的值总和
 *
 * 时间复杂度分析：
 * 1. insert 操作：O(m)，m 为键的长度
 * 2. sum 操作：O(m)，m 为前缀的长度
 * 空间复杂度分析：
 * 1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的键数量，M 是键的平均长度
 * 是否为最优解：是，利用 Trie 树的前缀特性可以高效实现键值映射
 *
 * 工程化考量：
 * 1. 异常处理：需要处理 null 或空字符串输入
 * 2. 可配置性：可以支持不同的字符集
 * 3. 线程安全：当前实现不是线程安全的
 * 4. 内存优化：对于稀疏字符集，使用哈希表比数组更节省空间
 *
 * 与机器学习等领域的联系：
 * 1. 特征工程：在机器学习中，前缀特征常用于文本分类等任务
 * 2. 数据库索引：Trie 树的思想在数据库前缀索引中广泛应用
 * 3. 缓存系统：LRU 等缓存淘汰算法可以结合 Trie 树实现前缀匹配缓存
 */

class MapSumNode {
    Map<Character, MapSumNode> children;
    int value; // 通过该节点的键的值总和
    int keyValue; // 如果是键的结尾，存储键的值
    boolean isEnd; // 是否为键的结尾

    public MapSumNode() {
        children = new HashMap<>();
        value = 0;
```

```
        keyValue = 0;
        isEnd = false;
    }
}

class MapSum {
    private MapSumNode root;
    private Map<String, Integer> keyMap; // 存储键值对，用于处理键更新

    public MapSum() {
        root = new MapSumNode();
        keyMap = new HashMap<>();
    }

    /**
     * 插入键值对
     * 时间复杂度: O(m)，其中 m 为键的长度
     * 空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
     */
    public void insert(String key, int val) {
        // 计算值的变化量
        int delta = val;
        if (keyMap.containsKey(key)) {
            delta -= keyMap.get(key);
        }
        keyMap.put(key, val);

        // 更新 Trie 树中的值
        MapSumNode node = root;
        for (char c : key.toCharArray()) {
            node.children.putIfAbsent(c, new MapSumNode());
            node = node.children.get(c);
            node.value += delta;
        }

        // 标记键的结尾
        node.isEnd = true;
        node.keyValue = val;
    }

    /**
     * 计算具有指定前缀的键的值总和
     * 时间复杂度: O(m)，其中 m 为前缀长度
```

```
     * 空间复杂度: O(1)
     */
    public int sum(String prefix) {
        MapSumNode node = root;
        for (char c : prefix.toCharArray()) {
            if (!node.children.containsKey(c)) {
                return 0;
            }
            node = node.children.get(c);
        }
        return node.value;
    }
}

/*
 * 题目 4: LeetCode 212. 单词搜索 II
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/word-search-ii/
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - LeetCode 211. 添加与搜索单词 - 数据结构设计
 * - LeetCode 745. 前缀和后缀搜索
 *
 * 题目描述：
 * 给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。
 * 单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中"相邻"单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。
 *
 * 解题思路：
 * 1. 使用 Trie 树预处理单词列表，提高搜索效率
 * 2. 对网格中的每个单元格进行 DFS 搜索，结合 Trie 树剪枝
 * 3. 使用回溯算法避免重复访问同一单元格
 * 4. 找到单词后将其从 Trie 树中删除，避免重复查找
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(L)，其中 L 是所有单词的总长度
 * 2. 网格搜索：O(M*N*4^L)，其中 M 和 N 是网格的维度，L 是单词的最大长度，4 是四个方向
 * 空间复杂度分析：
 * 1. Trie 树存储：O(L)
 * 2. 递归调用栈：O(L)
 * 3. 结果集存储：O(K)，K 是找到的单词数量
 * 是否为最优解：是，结合 Trie 树和 DFS 回溯的方案是解决此类问题的最优方法
```

```
 *
 * 工程化考量：
 * 1. 性能优化：找到单词后从 Trie 树中移除，避免重复查找
 * 2. 内存优化：可以使用更紧凑的数据结构表示 Trie 树
 * 3. 异常处理：需要处理空网格或空单词列表的情况
 *
 * 与机器学习等领域的联系：
 * 1. 自然语言处理：字符串匹配和搜索是 NLP 的基础操作
 * 2. 计算机视觉：类似的网格搜索思想在图像处理中也有应用
 * 3. 推荐系统：Trie 树的高效前缀匹配用于推荐算法
 */

class TrieNodeForWordSearch {
    Map<Character, TrieNodeForWordSearch> children;
    boolean isEndOfWord;
    String word; // 存储完整单词，方便找到后直接获取

    public TrieNodeForWordSearch() {
        children = new HashMap<>();
        isEndOfWord = false;
        word = null;
    }
}

class SolutionWordSearchII {
    private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public List<String> findWords(char[][] board, String[] words) {
        List<String> result = new ArrayList<>();
        if (board == null || board.length == 0 || board[0].length == 0 || words == null ||
words.length == 0) {
            return result;
        }

        // 构建 Trie 树
        TrieNodeForWordSearch root = new TrieNodeForWordSearch();
        for (String word : words) {
            TrieNodeForWordSearch node = root;
            for (char c : word.toCharArray()) {
                node.children.putIfAbsent(c, new TrieNodeForWordSearch());
                node = node.children.get(c);
            }
            node.isEndOfWord = true;
```

```
            node.word = word;
        }

        int m = board.length;
        int n = board[0].length;
        boolean[][] visited = new boolean[m][n];

        // 对网格中的每个单元格作为起点进行搜索
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (root.children.containsKey(board[i][j])) {
                    dfs(board, i, j, root, visited, result);
                }
            }
        }

        return result;
    }

    private void dfs(char[][] board, int i, int j, TrieNodeForWordSearch node, boolean[][]
visited, List<String> result) {
        // 检查边界条件和访问状态
        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j]) {
            return;
        }

        char c = board[i][j];
        if (!node.children.containsKey(c)) {
            return;
        }

        // 移动到下一个 Trie 节点
        node = node.children.get(c);

        // 如果当前节点是单词结尾，将单词加入结果集
        if (node.isEndOfWord) {
            result.add(node.word);
            // 标记为非单词结尾，避免重复添加
            node.isEndOfWord = false;
        }

        // 标记当前位置为已访问
        visited[i][j] = true;
```

```
        // 向四个方向递归搜索
        for (int[] dir : DIRECTIONS) {
            dfs(board, i + dir[0], j + dir[1], node, visited, result);
        }

        // 回溯：恢复访问状态
        visited[i][j] = false;
    }
}

/*
 * 题目 5: LeetCode 421. 数组中两个数的最大异或值
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - LeetCode 745. 前缀和后缀搜索
 * - HDU 5790 Prefix
 *
 * 题目描述：
 * 给你一个整数数组 nums ，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n 。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储二进制数字的位
 * 2. 从最高位到最低位构建 Trie 树
 * 3. 对每个数字，在 Trie 树中寻找能与之产生最大异或结果的路径
 * 4. 贪心策略：每一步尽可能选择与当前位不同的分支
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(N*32)，其中 N 是数组长度，32 是整数的位数
 * 2. 查询最大值：O(N*32)
 * 总体时间复杂度：O(N*32)
 * 空间复杂度分析：
 * 1. Trie 树存储：O(N*32)，最坏情况下每个数字的每一位都需要一个节点
 * 是否为最优解：是，Trie 树结合贪心的解法是该问题的最优解
 *
 * 工程化考量：
 * 1. 性能优化：可以提前计算最高有效位，减少不必要的计算
 * 2. 内存优化：使用位操作代替字符串处理，提高效率
 * 3. 异常处理：需要处理空数组或只有一个元素的情况
 *
 * 与机器学习等领域的联系：
```

```
 * 1. 信息论：异或操作在信息论中有重要应用
 * 2. 数据压缩：XOR 编码用于某些无损压缩算法
 * 3. 机器学习：特征选择中可能用到异或操作
 */

class TrieNodeForMaxXOR {
    Map<Integer, TrieNodeForMaxXOR> children;

    public TrieNodeForMaxXOR() {
        children = new HashMap<>();
    }
}

class SolutionMaxXOR {
    private static final int MAX_BIT = 30; // 因为题目中的数字不超过 2^31 - 1

    public int findMaximumXOR(int[] nums) {
        if (nums.length <= 1) {
            return 0;
        }

        // 构建 Trie 树，存储所有数字的二进制表示
        TrieNodeForMaxXOR root = new TrieNodeForMaxXOR();

        // 构建 Trie 树
        for (int num : nums) {
            TrieNodeForMaxXOR node = root;
            // 从最高位到最低位处理
            for (int i = MAX_BIT; i >= 0; i--) {
                int bit = (num >> i) & 1;
                node.children.putIfAbsent(bit, new TrieNodeForMaxXOR());
                node = node.children.get(bit);
            }
        }

        // 计算最大异或值
        int maxXOR = 0;
        for (int num : nums) {
            int currentXOR = 0;
            TrieNodeForMaxXOR node = root;
            // 从最高位到最低位处理
            for (int i = MAX_BIT; i >= 0; i--) {
                int bit = (num >> i) & 1;
```

```
                // 尝试找相反的位，以获得最大异或结果
                int targetBit = 1 - bit;
                if (node.children.containsKey(targetBit)) {
                    currentXOR |= (1 << i);
                    node = node.children.get(targetBit);
                } else {
                    // 如果没有相反的位，只能走相同的位
                    node = node.children.get(bit);
                }
            }
            maxXOR = Math.max(maxXOR, currentXOR);
        }

        return maxXOR;
    }
}

/*
 * 题目 6: LeetCode 1032. 字符流
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/stream-of-characters/
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - POJ 1451 T9
 * - HDU 5790 Prefix
 *
 * 题目描述：
 * 设计一个算法：接收一个字符流，并检查这些字符的后缀是否是字符串数组 words 中的一个字符串。
 * 例如，words = ["abc", "xyz"] 时，如果字符流为 "a", "x", "y", 那么 "axyz" 的后缀 "xyz" 匹配 words 中的字符串。
 * 实现 StreamChecker 类：
 * StreamChecker(String[] words) 构造函数，用字符串数组 words 初始化数据结构。
 * boolean query(char letter) 从字符流中接收一个新字符，如果字符流的后缀能匹配 words 中的任一字符串，返回 true ；否则，返回 false 。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词的逆序
 * 2. 维护一个字符流的后缀队列，每次查询时检查该后缀是否匹配任何单词
 * 3. 当字符流过长时，可以截断到最长单词长度
 *
 * 时间复杂度分析：
 * 1. 构造函数：O(L)，其中 L 是所有单词的总长度
 * 2. query 操作：O(K)，其中 K 是最长单词的长度
```

```
 * 空间复杂度分析：
 * 1. Trie 树存储：O(L)
 * 2. 字符流后缀存储：O(K)
 * 是否为最优解：是，逆序 Trie 树是处理后缀匹配的高效方案
 *
 * 工程化考量：
 * 1. 性能优化：可以记录最长单词长度，限制后缀队列的大小
 * 2. 内存优化：只保存最近的 K 个字符，K 为最长单词长度
 * 3. 异常处理：需要处理空单词数组的情况
 *
 * 与机器学习等领域的联系：
 * 1. 自然语言处理：文本匹配是 NLP 的基础操作
 * 2. 拼写检查：类似的思路用于实时拼写检查
 * 3. 生物信息学：DNA 序列匹配也会用到类似的技术
 */

class TrieNodeForStream {
    Map<Character, TrieNodeForStream> children;
    boolean isEndOfWord;

    public TrieNodeForStream() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

class StreamChecker {
    private TrieNodeForStream root;
    private StringBuilder stream;
    private int maxWordLength;

    public StreamChecker(String[] words) {
        root = new TrieNodeForStream();
        stream = new StringBuilder();
        maxWordLength = 0;

        // 构建逆序 Trie 树
        for (String word : words) {
            StringBuilder reversedWord = new StringBuilder(word).reverse();
            maxWordLength = Math.max(maxWordLength, word.length());
            TrieNodeForStream node = root;
            for (char c : reversedWord.toString().toCharArray()) {
                node.children.putIfAbsent(c, new TrieNodeForStream());
```

```
                node = node.children.get(c);
            }
            node.isEndOfWord = true;
        }
    }

    public boolean query(char letter) {
        // 将新字符添加到流中
        stream.append(letter);
        // 只保留最长单词长度的后缀
        if (stream.length() > maxWordLength) {
            stream.deleteCharAt(0);
        }

        // 从流的末尾开始，检查是否有单词匹配
        TrieNodeForStream node = root;
        // 逆序遍历流
        for (int i = stream.length() - 1; i >= 0; i--) {
            char c = stream.charAt(i);
            if (!node.children.containsKey(c)) {
                break;
            }
            node = node.children.get(c);
            if (node.isEndOfWord) {
                return true;
            }
        }
        return false;
    }
}

/*
 * 题目 7: POJ 3630 / HDU 1671 Phone List
 * 题目来源：POJ, HDU
 * 题目链接：http://poj.org/problem?id=3630, http://acm.hdu.edu.cn/showproblem.php?pid=1671
 * 相关题目：
 * - SPOJ PHONELST. Phone List
 * - Codeforces 633C. Spy Syndrome 2
 * - LeetCode 208. 实现 Trie (前缀树)
 *
 * 题目描述：
 * 给定一个电话号码列表，判断是否其中一个号码是另一个号码的前缀。
 * 如果存在这样的情况，则输出"NO"，否则输出"YES"。
```

```
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有电话号码
 * 2. 插入过程中检查：
 *    a. 当前号码是否是已存在号码的前缀
 *    b. 已存在号码是否是当前号码的前缀
 * 3. 如果满足任一条件，则返回 False
 *
 * 时间复杂度分析：
 * 1. 插入操作：O(L)，其中 L 是电话号码的平均长度
 * 2. 检查前缀：O(L)
 * 总体时间复杂度：O(N*L)，其中 N 是电话号码的数量
 * 空间复杂度分析：
 * 1. Trie 树存储：O(N*L)
 * 是否为最优解：是，Trie 树是解决前缀匹配问题的高效数据结构
 *
 * 工程化考量：
 * 1. 性能优化：可以按长度排序电话号码，先插入短的，再插入长的
 * 2. 异常处理：需要处理空列表或重复号码的情况
 * 3. 内存优化：对于数字字符，可以使用大小为 10 的数组代替哈希表
 *
 * 与机器学习等领域的联系：
 * 1. 数据验证：前缀匹配在数据验证中有广泛应用
 * 2. 信息检索：在搜索引擎中用于快速过滤不相关结果
 * 3. 自动补全：类似的思想用于实现自动补全功能
 */

class TrieNodeForPhoneList {
    Map<Character, TrieNodeForPhoneList> children;
    boolean isEndOfWord;
    int count; // 记录通过此节点的路径数量

    public TrieNodeForPhoneList() {
        children = new HashMap<>();
        isEndOfWord = false;
        count = 0;
    }
}

class SolutionPhoneList {
    /**
     * 检查电话号码列表是否有效（无前缀冲突）
     *
```

```
     * 算法思路：
     * 1. 按长度升序排序电话号码，优先处理短号码
     * 2. 使用 Trie 树存储所有电话号码
     * 3. 插入过程中检查前缀关系
     *
     * 时间复杂度：O(N*L)，其中 N 是电话号码数量，L 是平均长度
     * 空间复杂度：O(N*L)
     *
     * @param phoneNumbers 电话号码列表
     * @return 如果无前缀冲突返回 true，否则返回 false
     */
    public boolean isValidPhoneList(String[] phoneNumbers) {
        // 按长度升序排序，优先处理短号码
        Arrays.sort(phoneNumbers, Comparator.comparingInt(String::length));
        TrieNodeForPhoneList root = new TrieNodeForPhoneList();

        for (String phone : phoneNumbers) {
            if (phone == null || phone.isEmpty()) {
                return false;
            }

            TrieNodeForPhoneList node = root;
            boolean isPrefix = true; // 当前号码是否是已存在号码的前缀

            for (int i = 0; i < phone.length(); i++) {
                char c = phone.charAt(i);
                if (!node.children.containsKey(c)) {
                    node.children.put(c, new TrieNodeForPhoneList());
                    isPrefix = false; // 如果创建了新节点，说明当前号码不是已存在号码的前缀
                }

                node = node.children.get(c);
                node.count++;

                // 如果在遍历过程中遇到了一个完整的电话号码，说明已存在号码是当前号码的前缀
                if (i < phone.length() - 1 && node.isEndOfWord) {
                    return false;
                }
            }

            // 如果当前号码是已存在号码的前缀，返回 False
            if (isPrefix) {
                return false;
```

```
            }

            node.isEndOfWord = true;
        }

        return true;
    }
}

/*
 * 题目 8: 敏感词过滤
 * 题目来源：常见算法问题
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - LeetCode 211. 添加与搜索单词 - 数据结构设计
 * - POJ 3630 / HDU 1671 Phone List
 *
 * 题目描述：
 * 实现一个敏感词过滤系统，能够快速检测文本中是否包含指定的敏感词，并支持替换敏感词。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有敏感词
 * 2. 对输入文本进行遍历，使用双指针或 KMP 算法结合 Trie 树进行匹配
 * 3. 发现敏感词后进行替换
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(L)，其中 L 是所有敏感词的总长度
 * 2. 文本检测：O(N*M)，其中 N 是文本长度，M 是最长敏感词长度
 * 空间复杂度分析：
 * 1. Trie 树存储：O(L)
 * 2. 结果存储：O(N)
 * 是否为最优解：是，Trie 树结合扫描算法是处理多模式串匹配的高效方法
 *
 * 工程化考量：
 * 1. 性能优化：可以使用 AC 自动机进一步提高多模式串匹配效率
 * 2. 可配置性：支持忽略大小写、部分匹配等选项
 * 3. 内存优化：对于常见字符集，可以使用数组代替哈希表
 * 4. 线程安全：在多线程环境中需要同步机制
 *
 * 与机器学习等领域的联系：
 * 1. 内容审核：在社交媒体平台中用于自动过滤不当内容
 * 2. 自然语言处理：文本预处理中的停用词过滤
 * 3. 信息检索：在搜索引擎中过滤不相关内容
```

```
 */

class TrieNodeForSensitiveFilter {
    Map<Character, TrieNodeForSensitiveFilter> children;
    boolean isEndOfWord;

    public TrieNodeForSensitiveFilter() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

class SensitiveFilter {
    private TrieNodeForSensitiveFilter root;

    public SensitiveFilter() {
        root = new TrieNodeForSensitiveFilter();
    }

    /**
     * 添加敏感词到 Trie 树
     * @param word 敏感词
     */
    public void addSensitiveWord(String word) {
        if (word == null || word.isEmpty()) {
            return;
        }
        TrieNodeForSensitiveFilter node = root;
        for (char c : word.toCharArray()) {
            node.children.putIfAbsent(c, new TrieNodeForSensitiveFilter());
            node = node.children.get(c);
        }
        node.isEndOfWord = true;
    }

    /**
     * 过滤文本中的敏感词，用 replaceChar 替换
     * @param text 待过滤文本
     * @param replaceChar 替换字符
     * @return 过滤后的文本
     */
    public String filter(String text, char replaceChar) {
        if (text == null || text.isEmpty()) {
```

```
        return text;
    }

    char[] result = text.toCharArray();
    int i = 0;
    while (i < text.length()) {
        TrieNodeForSensitiveFilter node = root;
        int j = i;
        int lastEnd = -1; // 记录最后一个敏感词结束的位置

        // 尝试匹配敏感词
        while (j < text.length() && node.children.containsKey(text.charAt(j))) {
            node = node.children.get(text.charAt(j));
            j++;
            if (node.isEndOfWord) {
                lastEnd = j;
            }
        }

        // 如果找到敏感词，进行替换
        if (lastEnd != -1) {
            for (int k = i; k < lastEnd; k++) {
                result[k] = replaceChar;
            }
            i = lastEnd;
        } else {
            i++;
        }
    }

    return new String(result);
}

/**
 * 检查文本是否包含敏感词
 * @param text 待检查文本
 * @return 如果包含敏感词返回 true，否则返回 false
 */
public boolean containsSensitiveWord(String text) {
    if (text == null || text.isEmpty()) {
        return false;
    }
```

```
        for (int i = 0; i < text.length(); i++) {
            TrieNodeForSensitiveFilter node = root;
            int j = i;
            while (j < text.length() && node.children.containsKey(text.charAt(j))) {
                node = node.children.get(text.charAt(j));
                j++;
                if (node.isEndOfWord) {
                    return true;
                }
            }
        }

        return false;
    }
}

/*
 * 题目9: SPOJ DICT - Search in the dictionary!
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/DICT/
 * 相关题目：
 * - CodeChef DICT - Dictionary
 * - 牛客网 最长公共前缀
 * - 杭电OJ 1251 统计难题
 * - SPOJ ADAINDEX - Ada and Indexing
 *
 * 题目描述：
 * 给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。
 * 如果存在多个单词，按字典序输出。
 *
 * 解题思路：
 * 1. 使用Trie树存储字典中的所有单词
 * 2. 每个节点维护以该节点为前缀的所有单词
 * 3. 查询时找到前缀对应的节点，输出该节点存储的所有单词
 *
 * 时间复杂度分析：
 * 1. 构建Trie树：O(∑len(s))
 * 2. 查询过程：O(P + K)，其中P是前缀长度，K是输出单词数量
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie树是解决前缀查询的高效方法
 *
 * 工程化考量：
```

```
 * 1. 内存优化：可以使用更紧凑的存储方式
 * 2. 性能优化：预处理可以加速查询
 * 3. 排序处理：需要按字典序输出结果
 */

class DictionarySearchSPOJ {
    class TrieNode {
        TrieNode[] children;
        List<String> words; // 存储以该节点为前缀的所有单词
        boolean isEnd; // 标记是否为单词结尾

        public TrieNode() {
            children = new TrieNode[26];
            words = new ArrayList<>();
            isEnd = false;
        }
    }

    private TrieNode root;

    public DictionarySearchSPOJ(String[] dictionary) {
        root = new TrieNode();
        // 构建 Trie 树
        for (String word : dictionary) {
            insert(word);
        }
    }

    private void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
            node.words.add(word);
        }
        node.isEnd = true;
    }

    public List<String> search(String prefix) {
        TrieNode node = root;
```

```
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return new ArrayList<>(); // 前缀不存在
            }
            node = node.children[index];
        }
        // 返回该前缀对应的所有单词，按字典序排序
        Collections.sort(node.words);
        return node.words;
    }
}

/*
 * 题目 10: HDU 1251 统计难题
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251
 * 相关题目：
 * - 牛客网 最长公共前缀
 * - CodeChef DICT - Dictionary
 * - POJ 2001 Shortest Prefixes
 * - SPOJ ADAINDEX - Ada and Indexing
 *
 * 题目描述：
 * Ignatius 最近遇到一个难题，老师交给他很多单词(只有小写字母组成，不会有重复的单词出现)，
 * 现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 每个节点记录经过该节点的单词数量
 * 3. 查询时找到前缀对应的节点，返回该节点的计数
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P)，其中 P 是前缀长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 *
 * 工程化考量：
 * 1. 内存优化：对于大量单词，可以考虑压缩 Trie 树
 * 2. 性能优化：可以缓存常用查询结果
 * 3. 异常处理：处理空查询和边界情况
```

```
 */

class StatisticalProblemHDU {
    class TrieNode {
        TrieNode[] children;
        int count; // 经过该节点的单词数量

        public TrieNode() {
            children = new TrieNode[26];
            count = 0;
        }
    }

    private TrieNode root;

    public StatisticalProblemHDU(String[] words) {
        root = new TrieNode();
        for (String word : words) {
            insert(word);
        }
    }

    private void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
            node.count++;
        }
    }

    public int prefixCount(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return 0;
            }
            node = node.children[index];
        }
```

```
        return node.count;
    }
}

/*
 * Trie 树核心思想与应用场景总结：
 *
 * 核心思想：
 * 1. Trie 树是一种专门用于处理字符串前缀的数据结构
 * 2. 通过共享公共前缀来节省空间
 * 3. 每个节点代表一个字符串前缀，从根节点到某一节点的路径表示一个完整字符串
 *
 * 应用场景：
 * 1. 字符串检索：高效查找字典中的单词
 * 2. 前缀匹配：自动补全、拼写检查、前缀搜索
 * 3. 字符串排序：按字典序排序字符串
 * 4. 文本处理：敏感词过滤、垃圾邮件识别
 * 5. 网络路由：最长前缀匹配算法
 * 6. DNA 序列分析：生物信息学中的序列匹配
 *
 * 设计要点：
 * 1. 节点结构：通常包含子节点引用和单词结束标记
 * 2. 实现方式：可以使用数组或哈希表存储子节点
 * 3. 优化策略：
 *    - 压缩路径（Compressed Trie）：合并只有一个子节点的节点
 *    - 双数组 Trie（Double-Array Trie）：空间效率更高的实现
 *    - 后缀链接（Suffix Links）：用于 AC 自动机等高级应用
 *
 * 复杂度分析：
 * 1. 时间复杂度：
 *    - 插入/查找：O(m)，m 为字符串长度
 *    - 前缀匹配：O(m)
 * 2. 空间复杂度：O(ALPHABET_SIZE * N * M)，其中 N 是字符串数量，M 是平均长度
 *
 * 语言实现差异：
 * 1. Java：使用类和引用，可使用数组或 HashMap 实现子节点
 * 2. C++：使用结构体和指针，内存管理更灵活
 * 3. Python：使用字典和类，实现简洁但性能略低
 *
 * 工程化考量：
 * 1. 线程安全：需要考虑并发访问的同步问题
 * 2. 内存优化：根据字符集大小选择合适的子节点存储方式
 * 3. 异常处理：处理非法输入和边界情况
```

```
 * 4. 性能调优：根据实际应用场景选择合适的 Trie 树变体
 */

// 测试代码
public class Code01_TrieTree {
    public static void main(String[] args) {
        // 测试 Trie
        System.out.println("=== 测试 Trie ===");
        Trie trie = new Trie();
        trie.insert("apple");
        System.out.println("search(\"apple\"): " + trie.search("apple"));      // true
        System.out.println("search(\"app\"): " + trie.search("app"));          // false
        System.out.println("startsWith(\"app\"): " + trie.startsWith("app")); // true
        trie.insert("app");
        System.out.println("search(\"app\"): " + trie.search("app"));          // true

        // 测试 WordDictionary
        System.out.println("\n=== 测试 WordDictionary ===");
        WordDictionary wordDict = new WordDictionary();
        wordDict.addWord("bad");
        wordDict.addWord("dad");
        wordDict.addWord("mad");
        System.out.println("search(\"pad\"): " + wordDict.search("pad"));     // false
        System.out.println("search(\"bad\"): " + wordDict.search("bad"));     // true
        System.out.println("search(\".ad\"): " + wordDict.search(".ad"));     // true
        System.out.println("search(\"b..\"): " + wordDict.search("b.."));     // true

        // 测试 MapSum
        System.out.println("\n=== 测试 MapSum ===");
        MapSum mapSum = new MapSum();
        mapSum.insert("apple", 3);
        System.out.println("sum(\"ap\"): " + mapSum.sum("ap"));               // 3
        mapSum.insert("app", 2);
        System.out.println("sum(\"ap\"): " + mapSum.sum("ap"));               // 5

        // 测试 WordSearchII
        System.out.println("\n=== 测试 WordSearchII ===");
        char[][] board = {
            {'o', 'a', 'a', 'n'},
            {'e', 't', 'a', 'e'},
            {'i', 'h', 'k', 'r'},
            {'i', 'f', 'l', 'v'}
        };
```

```
        String[] words = {"oath", "pea", "eat", "rain"};
        SolutionWordSearchII solutionWordSearch = new SolutionWordSearchII();
        List<String> result = solutionWordSearch.findWords(board, words);
        System.out.println("找到的单词: " + result);  // [eat, oath]

        // 测试 MaxXOR
        System.out.println("\n=== 测试 MaxXOR ===");
        int[] nums = {3, 10, 5, 25, 2, 8};
        SolutionMaxXOR solutionMaxXOR = new SolutionMaxXOR();
        int maxXOR = solutionMaxXOR.findMaximumXOR(nums);
        System.out.println("最大异或值: " + maxXOR);  // 28

        // 测试 StreamChecker
        System.out.println("\n=== 测试 StreamChecker ===");
        StreamChecker streamChecker = new StreamChecker(new String[]{"cd", "f", "kl"});
        System.out.println("query('a'): " + streamChecker.query('a'));  // false
        System.out.println("query('b'): " + streamChecker.query('b'));  // false
        System.out.println("query('c'): " + streamChecker.query('c'));  // false
        System.out.println("query('d'): " + streamChecker.query('d'));  // true
        System.out.println("query('f'): " + streamChecker.query('f'));  // true

        // 测试 PhoneList
        System.out.println("\n=== 测试 PhoneList ===");
        SolutionPhoneList solutionPhoneList = new SolutionPhoneList();
        // 测试用例 1: 有前缀关系
        String[] phoneList1 = {"111", "11", "123"};
        System.out.println("有效电话号码列表 1: " +
solutionPhoneList.isValidPhoneList(phoneList1));  // false
        // 测试用例 2: 无前缀关系
        String[] phoneList2 = {"111", "222", "333"};
        System.out.println("有效电话号码列表 2: " +
solutionPhoneList.isValidPhoneList(phoneList2));  // true

        // 测试 SensitiveFilter
        System.out.println("\n=== 测试 SensitiveFilter ===");
        SensitiveFilter filter = new SensitiveFilter();
        // 添加敏感词
        String[] sensitiveWords = {"bad", "ugly", "terrible"};
        for (String word : sensitiveWords) {
            filter.addSensitiveWord(word);
        }
        // 测试过滤
        String text = "This is a bad example with some terrible words.";
```

```
        String filteredText = filter.filter(text, '*');
        System.out.println("原始文本: " + text);
        System.out.println("过滤后文本: " + filteredText);
        System.out.println("包含敏感词: " + filter.containsSensitiveWord(text));  // true

        // 测试 DictionarySearchSPOJ
        System.out.println("\n=== 测试 DictionarySearchSPOJ ===");
        String[] dictionary = {"abc", "abcd", "abcde", "bcd", "bcde"};
        DictionarySearchSPOJ dictSPOJ = new DictionarySearchSPOJ(dictionary);
        List<String> results = dictSPOJ.search("abc");
        System.out.println("前缀'abc'的单词: " + results);

        // 测试 StatisticalProblemHDU
        System.out.println("\n=== 测试 StatisticalProblemHDU ===");
        String[] wordsHDU = {"abc", "abcde", "abcdef", "bcd", "bcde"};
        StatisticalProblemHDU statHDU = new StatisticalProblemHDU(wordsHDU);
        System.out.println("前缀'abc'的数量: " + statHDU.prefixCount("abc")); // 3
        System.out.println("前缀'bc'的数量: " + statHDU.prefixCount("bc")); // 2
        System.out.println("前缀'xyz'的数量: " + statHDU.prefixCount("xyz")); // 0
    }
}
```

==========

文件: Code01_TrieTree.py

==========

```
# -*- coding: utf-8 -*-

'''
题目 1: LeetCode 208. 实现 Trie (前缀树)
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/implement-trie-prefix-tree/

题目描述：
Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补全和拼写检查。
请你实现 Trie 类：
Trie() 初始化前缀树对象。
void insert(String word) 向前缀树中插入字符串 word 。
boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。
```

解题思路：
1. Trie 树是一种专门处理字符串前缀的数据结构
2. 每个节点包含若干子节点（对应不同字符）和一个标记（表示是否为单词结尾）
3. 插入操作：从根节点开始，逐字符查找，若不存在则创建新节点
4. 搜索操作：从根节点开始，逐字符查找，若路径存在且终点为单词结尾则返回 true
5. 前缀搜索：从根节点开始，逐字符查找，若路径存在则返回 true

时间复杂度分析：
1. insert 操作：O(m)，m 为插入字符串的长度
2. search 操作：O(m)，m 为搜索字符串的长度
3. startsWith 操作：O(m)，m 为前缀字符串的长度
空间复杂度分析：
1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的字符串数量，M 是字符串的平均长度
2. 最坏情况下，没有公共前缀，每个字符都需要一个节点
是否为最优解：是，这是 Trie 树的标准实现，时间复杂度已达到理论最优

工程化考量：
1. 异常处理：可以增加输入参数校验，如检查 word 是否为 null 或空字符串
2. 可配置性：可以支持不同的字符集（不仅仅是小写字母 a-z）
3. 线程安全：当前实现不是线程安全的，如需线程安全需要额外同步机制
4. 性能优化：可以使用对象池减少频繁创建节点对象的开销
5. 内存优化：对于稀疏字符集，使用哈希表比数组更节省空间

语言特性差异：
1. Java：使用引用类型，有垃圾回收机制，数组实现固定子节点
2. C++：需要手动管理内存，可以使用数组或指针数组实现，性能更高但需注意内存泄漏
3. Python：动态类型语言，字典实现自然，但性能不如编译型语言

与机器学习等领域的联系：
1. 自然语言处理：Trie 树可用于构建词典、前缀匹配等
2. 信息检索：搜索引擎的自动补全功能常使用 Trie 树实现
3. 数据压缩：在某些压缩算法中，Trie 树用于构建霍夫曼编码树
4. 生物信息学：用于 DNA 序列匹配和分析

反直觉但关键的设计：
1. 每个节点不直接存储字符，而是通过父节点到子节点的路径表示字符
2. 根节点不表示任何字符，仅作为起始点
3. 节点的 isEnd 标记表示从根节点到当前节点的路径是否构成一个完整单词

极端场景鲁棒性：
1. 空字符串插入：需要特殊处理根节点的 end 计数
2. 重复字符串：通过 end 计数区分出现次数

3. 超长字符串：受限于系统内存，但算法本身无长度限制
4. 大量相似前缀：Trie 树的优势场景，能有效共享前缀存储空间
'''

```
class TrieNode:
    def __init__(self):
        # 通过字典存储子节点，键为字符，值为子节点
        self.children = {}
        # 标记该节点是否为某个单词的结尾
        self.is_end_of_word = False


class Trie:
    def __init__(self):
        # 初始化根节点
        self.root = TrieNode()

    def insert(self, word):
        """
        向 Trie 树中插入一个单词
        时间复杂度: O(m)，其中 m 为单词长度
        空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
        """
        node = self.root
        for char in word:
            # 如果字符不在当前节点的子节点中，则创建新节点
            if char not in node.children:
                node.children[char] = TrieNode()
            # 移动到子节点
            node = node.children[char]
        # 标记单词结尾
        node.is_end_of_word = True

    def search(self, word):
        """
        搜索 Trie 树中是否存在一个完整的单词
        时间复杂度: O(m)，其中 m 为单词长度
        空间复杂度: O(1)
        """
        node = self.root
        for char in word:
            # 如果字符不在当前节点的子节点中，说明单词不存在
```

```
            if char not in node.children:
                return False
            # 移动到子节点
            node = node.children[char]
        # 只有当到达单词结尾且该节点标记为单词结尾时，才返回 True
        return node.is_end_of_word

    def startsWith(self, prefix):
        """
        检查 Trie 树中是否有以给定前缀开头的单词
        时间复杂度: O(m)，其中 m 为前缀长度
        空间复杂度: O(1)
        """
        node = self.root
        for char in prefix:
            # 如果字符不在当前节点的子节点中，说明前缀不存在
            if char not in node.children:
                return False
            # 移动到子节点
            node = node.children[char]
        # 只要前缀存在，就返回 True
        return True


'''
```

题目 2: LeetCode 211. 添加与搜换单词 - 数据结构设计
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/design-add-and-search-words-data-structure/

题目描述：
请你设计一个数据结构，支持 添加新单词 和 查找字符串是否与任何先前添加的字符串匹配 。
实现词典类 WordDictionary：
WordDictionary() 初始化词典对象
void addWord(word) 将 word 添加到数据结构中，之后可以对它进行匹配
bool search(word) 如果数据结构中存在字符串与 word 匹配，则返回 true；否则，返回 false。word 中可能包含一些'.'，每个'.'都可以表示任何一个字母

解题思路：
1. 使用 Trie 树存储添加的单词
2. 搜索时遇到'.'字符，需要递归搜索所有子节点
3. 可以使用 DFS 或 BFS 实现模糊匹配

时间复杂度分析：

1. addWord 操作：O(m)，m 为单词长度
2. search 操作：最坏情况 O(26^m)，其中 m 为搜索字符串长度，当所有字符都是'.'时达到最坏情况
空间复杂度分析：
1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的字符串数量，M 是字符串的平均长度
是否为最优解：是，对于模糊匹配问题，这是标准的解决方案

工程化考量：
1. 性能优化：对于大量'.'的查询，可以考虑缓存查询结果
2. 异常处理：需要处理 null 或空字符串输入
3. 可配置性：可以支持不同的字符集
4. 线程安全：当前实现不是线程安全的

与机器学习等领域的联系：
1. 正则表达式匹配：类似模糊匹配的思想在正则表达式引擎中广泛应用
2. 模式识别：在图像识别等领域，模糊匹配用于处理变形或噪声数据
3. 自然语言处理：用于实现模糊搜索和拼写纠错功能
'''

```
class WordDictionaryNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class WordDictionary:
    def __init__(self):
        self.root = WordDictionaryNode()

    def addWord(self, word):
        """
        添加单词到数据结构中
        时间复杂度: O(m)，其中 m 为单词长度
        空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
        """
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = WordDictionaryNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
```

```
        """
        搜索是否存在匹配的单词，支持'.'通配符
        时间复杂度: 最坏情况O(26^m)，其中m为单词长度
        空间复杂度: O(m)，递归调用栈的深度
        """
        def dfs(index, node):
            # 递归终止条件：已遍历完整个单词
            if index == len(word):
                return node.is_end

            char = word[index]
            if char == '.':
                # 遇到'.'，需要匹配所有子节点
                for child_node in node.children.values():
                    if dfs(index + 1, child_node):
                        return True
                return False
            else:
                # 普通字符，直接查找
                if char not in node.children:
                    return False
                return dfs(index + 1, node.children[char])

        return dfs(0, self.root)


'''
```

题目3: LeetCode 677. 键值映射
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/map-sum-pairs/

题目描述：
实现一个 MapSum 类，支持两个方法，insert 和 sum：
MapSum() 初始化 MapSum 对象
void insert(String key, int val) 插入 key-val 键值对，字符串表示键 key ，整数表示值 val 。如果键 key 已经存在，那么原来的键值对将被替代成新的键值对。
int sum(string prefix) 返回所有以该前缀 prefix 开头的键 key 的值的总和。

解题思路：
1. 使用Trie树存储键值对
2. 每个节点存储经过该节点的键的值总和
3. 插入时需要处理键已存在的情况，先减去旧值再加上新值
4. 求和时找到前缀对应的节点，返回该节点存储的值总和

时间复杂度分析：
1. insert 操作：O(m)，m 为键的长度
2. sum 操作：O(m)，m 为前缀的长度
空间复杂度分析：
1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的键数量，M 是键的平均长度
是否为最优解：是，利用 Trie 树的前缀特性可以高效实现键值映射

工程化考量：
1. 异常处理：需要处理 null 或空字符串输入
2. 可配置性：可以支持不同的字符集
3. 线程安全：当前实现不是线程安全的
4. 内存优化：对于稀疏字符集，使用哈希表比数组更节省空间

与机器学习等领域的联系：
1. 特征工程：在机器学习中，前缀特征常用于文本分类等任务
2. 数据库索引：Trie 树的思想在数据库前缀索引中广泛应用
3. 缓存系统：LRU 等缓存淘汰算法可以结合 Trie 树实现前缀匹配缓存
'''

```
class MapSumNode:
    def __init__(self):
        self.children = {}
        self.value = 0  # 通过该节点的键的值总和
        self.key_value = 0  # 如果是键的结尾，存储键的值
        self.is_end = False  # 是否为键的结尾


class MapSum:
    def __init__(self):
        self.root = MapSumNode()
        self.key_map = {}  # 存储键值对，用于处理键更新

    def insert(self, key, val):
        """
        插入键值对
        时间复杂度: O(m)，其中 m 为键的长度
        空间复杂度: O(m)，最坏情况下需要创建 m 个新节点
        """
        # 计算值的变化量
        delta = val
        if key in self.key_map:
```

```
            delta -= self.key_map[key]
        self.key_map[key] = val

        # 更新 Trie 树中的值
        node = self.root
        for char in key:
            if char not in node.children:
                node.children[char] = MapSumNode()
            node = node.children[char]
            node.value += delta

        # 标记键的结尾
        node.is_end = True
        node.key_value = val

    def sum(self, prefix):
        """
        计算具有指定前缀的键的值总和
        时间复杂度: O(m)，其中 m 为前缀长度
        空间复杂度: O(1)
        """
        node = self.root
        for char in prefix:
            if char not in node.children:
                return 0
            node = node.children[char]
        return node.value


# 测试代码
if __name__ == "__main__":
    # 测试 Trie
    print("=== 测试 Trie ===")
    trie = Trie()
    trie.insert("apple")
    print(f"search('apple'): {trie.search('apple')}")      # True
    print(f"search('app'): {trie.search('app')}")          # False
    print(f"startsWith('app'): {trie.startsWith('app')}")  # True
    trie.insert("app")
    print(f"search('app'): {trie.search('app')}")          # True

    # 测试 WordDictionary
    print("\n=== 测试 WordDictionary ===")
```

```
    word_dict = WordDictionary()
    word_dict.addWord("bad")
    word_dict.addWord("dad")
    word_dict.addWord("mad")
    print(f"search('pad'): {word_dict.search('pad')}")     # False
    print(f"search('bad'): {word_dict.search('bad')}")     # True
    print(f"search('.ad'): {word_dict.search('.ad')}")     # True
    print(f"search('b..'): {word_dict.search('b..')}")     # True

    # 测试 MapSum
    print("\n=== 测试 MapSum ===")
    map_sum = MapSum()
    map_sum.insert("apple", 3)
    print(f"sum('ap'): {map_sum.sum('ap')}")               # 3
    map_sum.insert("app", 2)
    print(f"sum('ap'): {map_sum.sum('ap')}")               # 5


'''\n""
```

题目 4: LeetCode 212. 单词搜索 II
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/word-search-ii/

题目描述：
给定一个 m x n 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。
单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中"相邻"单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

解题思路：
1. 使用 Trie 树预处理单词列表，提高搜索效率
2. 对网格中的每个单元格进行 DFS 搜索，结合 Trie 树剪枝
3. 使用回溯算法避免重复访问同一单元格
4. 找到单词后将其从 Trie 树中删除，避免重复查找

时间复杂度分析：
1. 构建 Trie 树：O(L)，其中 L 是所有单词的总长度
2. 网格搜索：O(M*N*4^L)，其中 M 和 N 是网格的维度，L 是单词的最大长度，4 是四个方向
空间复杂度分析：
1. Trie 树存储：O(L)
2. 递归调用栈：O(L)
3. 结果集存储：O(K)，K 是找到的单词数量
是否为最优解：是，结合 Trie 树和 DFS 回溯的方案是解决此类问题的最优方法

工程化考量：
1. 性能优化：找到单词后从 Trie 树中移除，避免重复查找
2. 内存优化：可以使用更紧凑的数据结构表示 Trie 树
3. 异常处理：需要处理空网格或空单词列表的情况

与机器学习等领域的联系：
1. 自然语言处理：字符串匹配和搜索是 NLP 的基础操作
2. 计算机视觉：类似的网格搜索思想在图像处理中也有应用
3. 推荐系统：Trie 树的高效前缀匹配用于推荐算法
'''

```
class TrieNodeForWordSearch:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.word = None  # 存储完整单词，方便找到后直接获取


class SolutionWordSearchII:
    def findWords(self, board, words):
        # 构建 Trie 树
        root = TrieNodeForWordSearch()
        for word in words:
            node = root
            for char in word:
                if char not in node.children:
                    node.children[char] = TrieNodeForWordSearch()
                node = node.children[char]
            node.is_end_of_word = True
            node.word = word

        result = []
        m, n = len(board), len(board[0])

        # DFS 搜索函数
        def dfs(node, i, j, visited):
            # 如果当前节点是单词结尾，将单词加入结果集
            if node.is_end_of_word:
                result.append(node.word)
                # 标记为非单词结尾，避免重复添加
                node.is_end_of_word = False
```

```
            # 定义四个方向的偏移量
            directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

            # 遍历四个方向
            for dx, dy in directions:
                x, y = i + dx, j + dy
                # 检查新位置是否有效
                if 0 <= x < m and 0 <= y < n and (x, y) not in visited:
                    char = board[x][y]
                    # 检查当前字符是否在 Trie 树的下一层
                    if char in node.children:
                        visited.add((x, y))
                        dfs(node.children[char], x, y, visited)
                        visited.remove((x, y))

        # 对网格中的每个单元格作为起点进行搜索
        for i in range(m):
            for j in range(n):
                char = board[i][j]
                if char in root.children:
                    # 使用集合记录已访问的单元格
                    visited = set()
                    visited.add((i, j))
                    dfs(root.children[char], i, j, visited)

        return result


'''\n""
```

题目 5: LeetCode 421. 数组中两个数的最大异或值
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/maximum-xor-of-two-numbers-in-an-array/

题目描述：
给你一个整数数组 nums ，返回 nums[i] XOR nums[j] 的最大运算结果，其中 0 ≤ i ≤ j < n 。

解题思路：
1. 使用 Trie 树存储二进制数字的位
2. 从最高位到最低位构建 Trie 树
3. 对每个数字，在 Trie 树中寻找能与之产生最大异或结果的路径
4. 贪心策略：每一步尽可能选择与当前位不同的分支

时间复杂度分析：
1. 构建 Trie 树：O(N*32)，其中 N 是数组长度，32 是整数的位数
2. 查询最大值：O(N*32)
总体时间复杂度：O(N*32)
空间复杂度分析：
1. Trie 树存储：O(N*32)，最坏情况下每个数字的每一位都需要一个节点
是否为最优解：是，Trie 树结合贪心的解法是该问题的最优解

工程化考量：
1. 性能优化：可以提前计算最高有效位，减少不必要的计算
2. 内存优化：使用位操作代替字符串处理，提高效率
3. 异常处理：需要处理空数组或只有一个元素的情况

与机器学习等领域的联系：
1. 信息论：异或操作在信息论中有重要应用
2. 数据压缩：XOR 编码用于某些无损压缩算法
3. 机器学习：特征选择中可能用到异或操作
'''

```
class TrieNodeForMaxXOR:
    def __init__(self):
        # 使用 0 和 1 作为键，表示二进制位
        self.children = {}


class SolutionMaxXOR:
    def findMaximumXOR(self, nums):
        if len(nums) <= 1:
            return 0

        # 构建 Trie 树，存储所有数字的二进制表示
        root = TrieNodeForMaxXOR()
        max_bit = 30  # 因为题目中的数字不超过 2^31 - 1

        # 构建 Trie 树
        for num in nums:
            node = root
            # 从最高位到最低位处理
            for i in range(max_bit, -1, -1):
                bit = (num >> i) & 1
                if bit not in node.children:
                    node.children[bit] = TrieNodeForMaxXOR()
```

```
            node = node.children[bit]

        # 计算最大异或值
        max_xor = 0
        for num in nums:
            current_xor = 0
            node = root
            # 从最高位到最低位处理
            for i in range(max_bit, -1, -1):
                bit = (num >> i) & 1
                # 尝试找相反的位，以获得最大异或结果
                target_bit = 1 - bit
                if target_bit in node.children:
                    current_xor |= (1 << i)
                    node = node.children[target_bit]
                else:
                    # 如果没有相反的位，只能走相同的位
                    node = node.children.get(bit, node)
            max_xor = max(max_xor, current_xor)

        return max_xor


'''\n""
题目 6: LeetCode 1032. 字符流
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/stream-of-characters/

题目描述：
设计一个算法：接收一个字符流，并检查这些字符的后缀是否是字符串数组 words 中的一个字符串。
例如，words = ["abc", "xyz"] 时，如果字符流为 "a", "x", "y", 那么 "axyz" 的后缀 "xyz" 匹配 words 中的字符串。
实现 StreamChecker 类：
StreamChecker(String[] words) 构造函数，用字符串数组 words 初始化数据结构。
boolean query(char letter) 从字符流中接收一个新字符，如果字符流的后缀能匹配 words 中的任一字符串，返回 true ；否则，返回 false 。

解题思路：
1. 使用 Trie 树存储所有单词的逆序
2. 维护一个字符流的后缀队列，每次查询时检查该后缀是否匹配任何单词
3. 当字符流过长时，可以截断到最长单词长度

时间复杂度分析：
```

1. 构造函数：O(L)，其中 L 是所有单词的总长度
2. query 操作：O(K)，其中 K 是最长单词的长度
空间复杂度分析：
1. Trie 树存储：O(L)
2. 字符流后缀存储：O(K)
是否为最优解：是，逆序 Trie 树是处理后缀匹配的高效方案

工程化考量：
1. 性能优化：可以记录最长单词长度，限制后缀队列的大小
2. 内存优化：只保存最近的 K 个字符，K 为最长单词长度
3. 异常处理：需要处理空单词数组的情况

与机器学习等领域的联系：
1. 自然语言处理：文本匹配是 NLP 的基础操作
2. 拼写检查：类似的思路用于实时拼写检查
3. 生物信息学：DNA 序列匹配也会用到类似的技术
'''

```
class TrieNodeForStream:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False


class StreamChecker:
    def __init__(self, words):
        self.root = TrieNodeForStream()
        self.stream = []
        self.max_word_length = 0

        # 构建逆序 Trie 树
        for word in words:
            reversed_word = word[::-1]
            self.max_word_length = max(self.max_word_length, len(word))
            node = self.root
            for char in reversed_word:
                if char not in node.children:
                    node.children[char] = TrieNodeForStream()
                node = node.children[char]
            node.is_end_of_word = True

    def query(self, letter):
```

```
        # 将新字符添加到流中
        self.stream.append(letter)
        # 只保留最长单词长度的后缀
        if len(self.stream) > self.max_word_length:
            self.stream.pop(0)

        # 从流的末尾开始，检查是否有单词匹配
        node = self.root
        # 逆序遍历流
        for i in range(len(self.stream) - 1, -1, -1):
            char = self.stream[i]
            if char not in node.children:
                break
            node = node.children[char]
            if node.is_end_of_word:
                return True
        return False


'''\n""
题目 7: POJ 3630 / HDU 1671 Phone List
题目来源：POJ, HDU
题目链接：http://poj.org/problem?id=3630, http://acm.hdu.edu.cn/showproblem.php?pid=1671

题目描述：
给定一个电话号码列表，判断是否其中一个号码是另一个号码的前缀。
如果存在这样的情况，则输出"NO"，否则输出"YES"。

解题思路：
1. 使用 Trie 树存储所有电话号码
2. 插入过程中检查：
   a. 当前号码是否是已存在号码的前缀
   b. 已存在号码是否是当前号码的前缀
3. 如果满足任一条件，则返回 False

时间复杂度分析：
1. 插入操作：O(L)，其中 L 是电话号码的平均长度
2. 检查前缀：O(L)
总体时间复杂度：O(N*L)，其中 N 是电话号码的数量
空间复杂度分析：
1. Trie 树存储：O(N*L)
是否为最优解：是，Trie 树是解决前缀匹配问题的高效数据结构
```

工程化考量：
1. 性能优化：可以按长度排序电话号码，先插入短的，再插入长的
2. 异常处理：需要处理空列表或重复号码的情况
3. 内存优化：对于数字字符，可以使用大小为 10 的数组代替哈希表

与机器学习等领域的联系：
1. 数据验证：前缀匹配在数据验证中有广泛应用
2. 信息检索：在搜索引擎中用于快速过滤不相关结果
3. 自动补全：类似的思想用于实现自动补全功能
'''

```
class TrieNodeForPhoneList:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.count = 0  # 记录通过此节点的路径数量


class SolutionPhoneList:
    def isValidPhoneList(self, phoneNumbers):
        # 按长度升序排序，优先处理短号码
        phoneNumbers.sort(key=len)
        root = TrieNodeForPhoneList()

        for phone in phoneNumbers:
            if not phone:
                return False

            node = root
            is_prefix = True  # 当前号码是否是已存在号码的前缀

            for i, char in enumerate(phone):
                if char not in node.children:
                    node.children[char] = TrieNodeForPhoneList()
                    is_prefix = False  # 如果创建了新节点，说明当前号码不是已存在号码的前缀

                node = node.children[char]
                node.count += 1

                # 如果在遍历过程中遇到了一个完整的电话号码，说明已存在号码是当前号码的前缀
                if i < len(phone) - 1 and node.is_end_of_word:
                    return False
```

```
            # 如果当前号码是已存在号码的前缀，返回 False
            if is_prefix:
                return False

            node.is_end_of_word = True

        return True


'''\n""
题目 8: 敏感词过滤
题目来源：常见算法问题

题目描述：
实现一个敏感词过滤系统，能够快速检测文本中是否包含指定的敏感词，并支持替换敏感词。

解题思路：
1. 使用 Trie 树存储所有敏感词
2. 对输入文本进行遍历，使用双指针或 KMP 算法结合 Trie 树进行匹配
3. 发现敏感词后进行替换

时间复杂度分析：
1. 构建 Trie 树：O(L)，其中 L 是所有敏感词的总长度
2. 文本检测：O(N*M)，其中 N 是文本长度，M 是最长敏感词长度
空间复杂度分析：
1. Trie 树存储：O(L)
2. 结果存储：O(N)
是否为最优解：是，Trie 树结合扫描算法是处理多模式串匹配的高效方法

工程化考量：
1. 性能优化：可以使用 AC 自动机进一步提高多模式串匹配效率
2. 可配置性：支持忽略大小写、部分匹配等选项
3. 内存优化：对于常见字符集，可以使用数组代替哈希表
4. 线程安全：在多线程环境中需要同步机制

与机器学习等领域的联系：
1. 内容审核：在社交媒体平台中用于自动过滤不当内容
2. 自然语言处理：文本预处理中的停用词过滤
3. 信息检索：在搜索引擎中过滤不相关内容
'''
```

```
class TrieNodeForSensitiveFilter:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False


class SensitiveFilter:
    def __init__(self):
        self.root = TrieNodeForSensitiveFilter()

    def addSensitiveWord(self, word):
        """添加敏感词到 Trie 树"""
        if not word:
            return
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNodeForSensitiveFilter()
            node = node.children[char]
        node.is_end_of_word = True

    def filter(self, text, replace_char='*'):
        """过滤文本中的敏感词，用 replace_char 替换"""
        if not text:
            return text

        result = list(text)
        i = 0
        while i < len(text):
            node = self.root
            j = i
            last_end = -1  # 记录最后一个敏感词结束的位置

            # 尝试匹配敏感词
            while j < len(text) and text[j] in node.children:
                node = node.children[text[j]]
                j += 1
                if node.is_end_of_word:
                    last_end = j

            # 如果找到敏感词，进行替换
            if last_end != -1:
                for k in range(i, last_end):
```

```
                    result[k] = replace_char
                i = last_end
            else:
                i += 1

        return ''.join(result)

    def containsSensitiveWord(self, text):
        """检查文本是否包含敏感词"""
        if not text:
            return False

        for i in range(len(text)):
            node = self.root
            j = i
            while j < len(text) and text[j] in node.children:
                node = node.children[text[j]]
                j += 1
                if node.is_end_of_word:
                    return True

        return False


'''\n"""
```

题目 9: SPOJ DICT - Search in the dictionary!
题目来源：SPOJ
题目链接：https://www.spoj.com/problems/DICT/
相关题目：
- CodeChef DICT - Dictionary
- 牛客网 最长公共前缀
- 杭电 OJ 1251 统计难题
- SPOJ ADAINDEX - Ada and Indexing

题目描述：
给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。
如果存在多个单词，按字典序输出。

解题思路：
1. 使用 Trie 树存储字典中的所有单词
2. 每个节点维护以该节点为前缀的所有单词
3. 查询时找到前缀对应的节点，输出该节点存储的所有单词

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是，Trie 树是解决前缀查询的高效方法

工程化考量：
1. 内存优化：可以使用更紧凑的存储方式
2. 性能优化：预处理可以加速查询
3. 排序处理：需要按字典序输出结果

与机器学习等领域的联系：
1. 信息检索：搜索引擎中的关键词建议
2. 自然语言处理：词汇匹配和文本分析
3. 数据挖掘：模式识别和关联规则挖掘
'''

```
class DictionarySearchSPOJNode:
    def __init__(self):
        self.children = {}
        self.words = []  # 存储以该节点为前缀的所有单词
        self.is_end = False


class DictionarySearchSPOJ:
    def __init__(self, dictionary):
        self.root = DictionarySearchSPOJNode()
        # 构建 Trie 树
        for word in dictionary:
            self.insert(word)

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = DictionarySearchSPOJNode()
            node = node.children[char]
            node.words.append(word)
        node.is_end = True

    def search(self, prefix):
```

```
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []  # 前缀不存在
            node = node.children[char]
        # 返回该前缀对应的所有单词，按字典序排序
        return sorted(node.words)


'''
```

题目 10: HDU 1251 统计难题
题目来源：HDU
题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251
相关题目：
- 牛客网 最长公共前缀
- CodeChef DICT - Dictionary
- POJ 2001 Shortest Prefixes
- SPOJ ADAINDEX - Ada and Indexing

题目描述：
Ignatius 最近遇到一个难题，老师交给他很多单词(只有小写字母组成，不会有重复的单词出现)，
现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。

解题思路：
1. 使用 Trie 树存储所有单词
2. 每个节点记录经过该节点的单词数量
3. 查询时找到前缀对应的节点，返回该节点的计数

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(P)，其中 P 是前缀长度
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是

工程化考量：
1. 内存优化：对于大量单词，可以考虑压缩 Trie 树
2. 性能优化：可以缓存常用查询结果
3. 异常处理：处理空查询和边界情况

与机器学习等领域的联系：
1. 特征工程：在文本分类中统计词频特征
2. 信息检索：搜索引擎中的词频统计

3. 自然语言处理：语言模型中的 n-gram 统计
'''

```
class StatisticalProblemHDUNode:
    def __init__(self):
        self.children = {}
        self.count = 0  # 经过该节点的单词数量


class StatisticalProblemHDU:
    def __init__(self, words):
        self.root = StatisticalProblemHDUNode()
        for word in words:
            self.insert(word)

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = StatisticalProblemHDUNode()
            node = node.children[char]
            node.count += 1

    def prefix_count(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return 0
            node = node.children[char]
        return node.count


# 测试新增的题目实现
if __name__ == "__main__":
    # 测试 WordSearchII
    print("\n=== 测试 WordSearchII ===")
    board = [
        ['o', 'a', 'a', 'n'],
        ['e', 't', 'a', 'e'],
        ['i', 'h', 'k', 'r'],
        ['i', 'f', 'l', 'v']
    ]
```

```
words = ["oath", "pea", "eat", "rain"]
solution_word_search = SolutionWordSearchII()
result = solution_word_search.findWords(board, words)
print(f"找到的单词: {result}")  # ['eat', 'oath']

# 测试 MaxXOR
print("\n=== 测试 MaxXOR ===")
nums = [3, 10, 5, 25, 2, 8]
solution_max_xor = SolutionMaxXOR()
max_xor = solution_max_xor.findMaximumXOR(nums)
print(f"最大异或值: {max_xor}")  # 28

# 测试 StreamChecker
print("\n=== 测试 StreamChecker ===")
stream_checker = StreamChecker(["cd", "f", "kl"])
print(f"query('a'): {stream_checker.query('a')}")  # False
print(f"query('b'): {stream_checker.query('b')}")  # False
print(f"query('c'): {stream_checker.query('c')}")  # False
print(f"query('d'): {stream_checker.query('d')}")  # True
print(f"query('f'): {stream_checker.query('f')}")  # True

# 测试 PhoneList
print("\n=== 测试 PhoneList ===")
solution_phone_list = SolutionPhoneList()
# 测试用例 1: 有前缀关系
phone_list1 = ["111", "11", "123"]
print(f"有效电话号码列表 1: {solution_phone_list.isValidPhoneList(phone_list1)}")  # False
# 测试用例 2: 无前缀关系
phone_list2 = ["111", "222", "333"]
print(f"有效电话号码列表 2: {solution_phone_list.isValidPhoneList(phone_list2)}")  # True

# 测试 SensitiveFilter
print("\n=== 测试 SensitiveFilter ===")
filter = SensitiveFilter()
# 添加敏感词
sensitive_words = ["bad", "ugly", "terrible"]
for word in sensitive_words:
    filter.addSensitiveWord(word)
# 测试过滤
text = "This is a bad example with some terrible words."
filtered_text = filter.filter(text)
print(f"原始文本: {text}")
print(f"过滤后文本: {filtered_text}")
```

```
    print(f"包含敏感词: {filter.containsSensitiveWord(text)}")  # True

    # 测试 DictionarySearchSPOJ
    print("\n=== 测试 DictionarySearchSPOJ ===")
    dictionary = ["abc", "abcd", "abcde", "bcd", "bcde"]
    dict_spoj = DictionarySearchSPOJ(dictionary)
    results = dict_spoj.search("abc")
    print(f"前缀'abc'的单词: {results}")

    # 测试 StatisticalProblemHDU
    print("\n=== 测试 StatisticalProblemHDU ===")
    words_hdu = ["abc", "abcde", "abcdef", "bcd", "bcde"]
    stat_hdu = StatisticalProblemHDU(words_hdu)
    print(f"前缀'abc'的数量: {stat_hdu.prefix_count('abc')}")  # 3
    print(f"前缀'bc'的数量: {stat_hdu.prefix_count('bc')}")   # 2
    print(f"前缀'xyz'的数量: {stat_hdu.prefix_count('xyz')}")  # 0

'''\n""
Trie 树核心思想与应用场景总结：

核心思想：
1. Trie 树是一种专门用于处理字符串前缀的数据结构
2. 通过共享公共前缀来节省空间
3. 每个节点代表一个字符串前缀，从根节点到某一节点的路径表示一个完整字符串

应用场景：
1. 字符串检索：高效查找字典中的单词
2. 前缀匹配：自动补全、拼写检查、前缀搜索
3. 字符串排序：按字典序排序字符串
4. 文本处理：敏感词过滤、垃圾邮件识别
5. 网络路由：最长前缀匹配算法
6. DNA 序列分析：生物信息学中的序列匹配

设计要点：
1. 节点结构：通常包含子节点引用和单词结束标记
2. 实现方式：可以使用数组或哈希表存储子节点
3. 优化策略：
   - 压缩路径（Compressed Trie）：合并只有一个子节点的节点
   - 双数组 Trie（Double-Array Trie）：空间效率更高的实现
   - 后缀链接（Suffix Links）：用于 AC 自动机等高级应用

复杂度分析：
1. 时间复杂度：
```

   - 插入/查找：O(m)，m 为字符串长度
   - 前缀匹配：O(m)
2. 空间复杂度：O(ALPHABET_SIZE * N * M)，其中 N 是字符串数量，M 是平均长度

语言实现差异：
1. Java：使用类和引用，可使用数组或 HashMap 实现子节点
2. C++：使用结构体和指针，内存管理更灵活
3. Python：使用字典和类，实现简洁但性能略低

工程化考量：
1. 线程安全：需要考虑并发访问的同步问题
2. 内存优化：根据字符集大小选择合适的子节点存储方式
3. 异常处理：处理非法输入和边界情况
4. 性能调优：根据实际应用场景选择合适的 Trie 树变体
'''

==========

文件: Code02_TrieTree.java
==========
```
package class044;

/*
 * 题目 1: LeetCode 208. 实现 Trie (前缀树)
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/implement-trie-prefix-tree/
 *
 * 题目描述：
 * Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。
 * 这一数据结构有相当多的应用情景，例如自动补全和拼写检查。
 * 请你实现 Trie 类：
 * Trie() 初始化前缀树对象。
 * void insert(String word) 向前缀树中插入字符串 word 。
 * boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
 * boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。
 *
 * 解题思路：
 * 1. Trie 树是一种专门处理字符串前缀的数据结构
 * 2. 每个节点包含若干子节点（对应不同字符）和一个标记（表示是否为单词结尾）
```

* 3. 插入操作：从根节点开始，逐字符查找，若不存在则创建新节点
* 4. 搜索操作：从根节点开始，逐字符查找，若路径存在且终点为单词结尾则返回 true
* 5. 前缀搜索：从根节点开始，逐字符查找，若路径存在则返回 true
*
* 时间复杂度分析：
* 1. insert 操作：O(m)，m 为插入字符串的长度
* 2. search 操作：O(m)，m 为搜索字符串的长度
* 3. startsWith 操作：O(m)，m 为前缀字符串的长度
* 空间复杂度分析：
* 1. O(ALPHABET_SIZE * N * M)，其中 N 是插入的字符串数量，M 是字符串的平均长度
* 2. 最坏情况下，没有公共前缀，每个字符都需要一个节点
* 是否为最优解：是，这是 Trie 树的标准实现，时间复杂度已达到理论最优
*
* 工程化考量：
* 1. 异常处理：可以增加输入参数校验，如检查 word 是否为 null 或空字符串
* 2. 可配置性：可以支持不同的字符集（不仅仅是小写字母 a-z）
* 3. 线程安全：当前实现不是线程安全的，如需线程安全需要额外同步机制
* 4. 性能优化：可以使用对象池减少频繁创建节点对象的开销
* 5. 内存优化：对于稀疏字符集，使用哈希表比数组更节省空间
*
* 语言特性差异：
* 1. Java：使用引用类型，有垃圾回收机制，数组实现固定子节点
* 2. C++：需要手动管理内存，可以使用数组或指针数组实现
* 3. Python：动态类型语言，字典实现自然，但性能不如编译型语言
*
* 与机器学习等领域的联系：
* 1. 自然语言处理：Trie 树可用于构建词典、前缀匹配等
* 2. 信息检索：搜索引擎的自动补全功能常使用 Trie 树实现
* 3. 数据压缩：在某些压缩算法中，Trie 树用于构建霍夫曼编码树
* 4. 生物信息学：用于 DNA 序列匹配和分析
*
* 反直觉但关键的设计：
* 1. 每个节点不直接存储字符，而是通过父节点到子节点的路径表示字符
* 2. 根节点不表示任何字符，仅作为起始点
* 3. 节点的 isEnd 标记表示从根节点到当前节点的路径是否构成一个完整单词
*
* 极端场景鲁棒性：
* 1. 空字符串插入：需要特殊处理根节点的 end 计数
* 2. 重复字符串：通过 end 计数区分出现次数
* 3. 超长字符串：受限于系统内存，但算法本身无长度限制
* 4. 大量相似前缀：Trie 树的优势场景，能有效共享前缀存储空间
*/

```
// 用固定数组实现前缀树，空间使用是静态的。推荐!
// 测试链接 : https://www.nowcoder.com/practice/7f8a8553ddbf4eaab749ec988726702b
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

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

public class Code02_TrieTree {

    // 如果将来增加了数据量，就改大这个值
    public static int MAXN = 150001;

    public static int[][] tree = new int[MAXN][26];

    public static int[] end = new int[MAXN];

    public static int[] pass = new int[MAXN];

    public static int cnt;

    public static void build() {
        cnt = 1;
    }

    public static void insert(String word) {
        int cur = 1;
        pass[cur]++;
        for (int i = 0, path; i < word.length(); i++) {
            path = word.charAt(i) - 'a';
            if (tree[cur][path] == 0) {
                tree[cur][path] = ++cnt;
            }
            cur = tree[cur][path];
            pass[cur]++;
        }
        end[cur]++;
    }
```

```
public static int search(String word) {
    int cur = 1;
    for (int i = 0, path; i < word.length(); i++) {
        path = word.charAt(i) - 'a';
        if (tree[cur][path] == 0) {
            return 0;
        }
        cur = tree[cur][path];
    }
    return end[cur];
}

public static int prefixNumber(String pre) {
    int cur = 1;
    for (int i = 0, path; i < pre.length(); i++) {
        path = pre.charAt(i) - 'a';
        if (tree[cur][path] == 0) {
            return 0;
        }
        cur = tree[cur][path];
    }
    return pass[cur];
}

public static void delete(String word) {
    if (search(word) > 0) {
        int cur = 1;
        // 下面这一行代码，讲课的时候没加
        // 本题不会用到pass[1]的信息，所以加不加都可以，不过正确的写法是加上
        pass[cur]--;
        for (int i = 0, path; i < word.length(); i++) {
            path = word.charAt(i) - 'a';
            if (--pass[tree[cur][path]] == 0) {
                tree[cur][path] = 0;
                return;
            }
            cur = tree[cur][path];
        }
        end[cur]--;
    }
}
```

```
    public static void clear() {
        for (int i = 1; i <= cnt; i++) {
            Arrays.fill(tree[i], 0);
            end[i] = 0;
            pass[i] = 0;
        }
    }

    public static int m, op;

    public static String[] splits;

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        String line = null;
        while ((line = in.readLine()) != null) {
            build();
            m = Integer.valueOf(line);
            for (int i = 1; i <= m; i++) {
                splits = in.readLine().split(" ");
                op = Integer.valueOf(splits[0]);
                if (op == 1) {
                    insert(splits[1]);
                } else if (op == 2) {
                    delete(splits[1]);
                } else if (op == 3) {
                    out.println(search(splits[1]) > 0 ? "YES" : "NO");
                } else if (op == 4) {
                    out.println(prefixNumber(splits[1]));
                }
            }
            clear();
        }
        out.flush();
        in.close();
        out.close();
    }

}
```

==========

文件: Code03_PhoneList.cpp

```
==========================================
/*
 * C++标准库包含文件
 * 由于编译环境限制，此处省略具体包含
 */

/*
 * 题目 4: POJ 3630 / HDU 1671 Phone List
 * 题目来源：POJ / HDU
 * 题目链接：http://poj.org/problem?id=3630
 *           http://acm.hdu.edu.cn/showproblem.php?pid=1671
 *
 * 题目描述：
 * 给定 n 个电话号码，判断是否存在一个电话号码是另一个电话号码的前缀。
 * 如果存在输出 NO，否则输出 YES。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有电话号码
 * 2. 在插入过程中检查是否存在前缀关系
 * 3. 如果在插入过程中遇到已经标记为结尾的节点，说明当前字符串是之前某个字符串的前缀
 * 4. 如果在插入完成后，当前节点还有子节点，说明之前某个字符串是当前字符串的前缀
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
 * 2. 查询过程：O(∑len(s))，遍历所有电话号码
 * 3. 总体时间复杂度：O(∑len(s))
 *
 * 空间复杂度分析：
 * 1. Trie 树空间：O(∑len(s) * 10)，每个节点最多有 10 个子节点(0-9)
 * 2. 总体空间复杂度：O(∑len(s))
 *
 * 是否为最优解：是，使用 Trie 树可以在线性时间内检测前缀关系
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或电话号码为空的情况
 * 2. 边界情况：所有电话号码都相同的情况
 * 3. 极端输入：大量电话号码或电话号码很长的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
 * 语言特性差异：
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
```

 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
 * 1. 电话系统：检测电话号码前缀冲突
 * 2. 网络路由：IP 地址前缀匹配
 * 3. 数据库索引：前缀索引优化查询
 */

/*
 * TrieNode 结构体定义
 * bool isEnd: 标记是否为某个电话号码的结尾
 * map<char, TrieNode*> children: 子节点映射
 */

/*
 * Trie 类定义
 * Trie(): 构造函数，初始化根节点
 * bool insert(const string& phoneNumber): 插入电话号码并检查前缀关系
 */

/*
 * 检查电话号码列表是否存在前缀关系
 *
 * 算法思路：
 * 1. 使用 Trie 树存储所有电话号码
 * 2. 在插入过程中检查是否存在前缀关系
 *
 * 时间复杂度：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
 * 空间复杂度：O(∑len(s) * 10)
 */

/*
 * 测试方法
 * 测试用例 1：存在前缀关系
 * 测试用例 2：不存在前缀关系
 * 测试用例 3：相同号码
 */

==========

文件: Code03_PhoneList.java
==========
package class044;

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

/*
 * 题目4: POJ 3630 / HDU 1671 Phone List
 * 题目来源：POJ / HDU
 * 题目链接：http://poj.org/problem?id=3630
 *           http://acm.hdu.edu.cn/showproblem.php?pid=1671
 * 相关题目：
 * - SPOJ PHONELST. Phone List
 * - Codeforces 633C. Spy Syndrome 2
 * - LeetCode 208. 实现 Trie (前缀树)
 * - SPOJ ADAINDEX - Ada and Indexing
 * - CodeChef DICT - Dictionary
 *
 * 题目描述：
 * 给定n个电话号码，判断是否存在一个电话号码是另一个电话号码的前缀。
 * 如果存在输出NO，否则输出YES。
 *
 * 解题思路：
 * 1. 使用Trie树存储所有电话号码
 * 2. 在插入过程中检查是否存在前缀关系
 * 3. 如果在插入过程中遇到已经标记为结尾的节点，说明当前字符串是之前某个字符串的前缀
 * 4. 如果在插入完成后，当前节点还有子节点，说明之前某个字符串是当前字符串的前缀
 *
 * 时间复杂度分析：
 * 1. 构建Trie树：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
 * 2. 查询过程：O(∑len(s))，遍历所有电话号码
 * 3. 总体时间复杂度：O(∑len(s))
 *
 * 空间复杂度分析：
 * 1. Trie树空间：O(∑len(s) * 10)，每个节点最多有10个子节点(0-9)
 * 2. 总体空间复杂度：O(∑len(s))
 *
 * 是否为最优解：是，使用Trie树可以在线性时间内检测前缀关系
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或电话号码为空的情况
 * 2. 边界情况：所有电话号码都相同的情况
 * 3. 极端输入：大量电话号码或电话号码很长的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
```

```
 * 语言特性差异：
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
 * 1. 电话系统：检测电话号码前缀冲突
 * 2. 网络路由：IP 地址前缀匹配
 * 3. 数据库索引：前缀索引优化查询
 */

class TrieNode {
    boolean isEnd; // 标记是否为某个电话号码的结尾
    Map<Character, TrieNode> children; // 子节点映射

    public TrieNode() {
        isEnd = false;
        children = new HashMap<>();
    }
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    /**
     * 插入电话号码并检查前缀关系
     * @param phoneNumber 电话号码
     * @return 如果存在前缀关系返回 true，否则返回 false
     */
    public boolean insert(String phoneNumber) {
        TrieNode node = root;
        for (int i = 0; i < phoneNumber.length(); i++) {
            char ch = phoneNumber.charAt(i);
            // 如果字符不在当前节点的子节点中，则创建新节点
            if (!node.children.containsKey(ch)) {
                node.children.put(ch, new TrieNode());
            }
            node = node.children.get(ch);
            // 如果在到达当前电话号码结尾之前遇到了已标记为结尾的节点，
```

```
            // 说明当前电话号码是之前某个电话号码的前缀
            if (node.isEnd) {
                return true;
            }
        }
        // 标记当前电话号码结尾
        node.isEnd = true;
        // 如果当前节点还有子节点，说明之前某个电话号码是当前电话号码的前缀
        return !node.children.isEmpty();
    }
}

public class Code03_PhoneList {

    /**
     * 检查电话号码列表是否存在前缀关系
     *
     * 算法思路：
     * 1. 使用 Trie 树存储所有电话号码
     * 2. 在插入过程中检查是否存在前缀关系
     *
     * 时间复杂度：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
     * 空间复杂度：O(∑len(s) * 10)
     *
     * @param phoneNumbers 电话号码列表
     * @return 如果存在前缀关系返回 false，否则返回 true
     */
    public static boolean isConsistent(String[] phoneNumbers) {
        Trie trie = new Trie();
        for (String phoneNumber : phoneNumbers) {
            if (trie.insert(phoneNumber)) {
                return false;
            }
        }
        return true;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1：存在前缀关系
        String[] phoneNumbers1 = {"911", "97625999", "91125426"};
        System.out.println("测试用例 1 结果: " + (isConsistent(phoneNumbers1) ? "YES" : "NO")); //
应该输出 NO
```

```
        // 测试用例 2：不存在前缀关系
        String[] phoneNumbers2 = {"113", "12340", "123440", "12345", "98346"};
        System.out.println("测试用例 2 结果: " + (isConsistent(phoneNumbers2) ? "YES" : "NO")); //
应该输出 YES

        // 测试用例 3：相同号码
        String[] phoneNumbers3 = {"123", "123"};
        System.out.println("测试用例 3 结果: " + (isConsistent(phoneNumbers3) ? "YES" : "NO")); //
应该输出 NO
    }
}

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

文件: Code03_PhoneList.py
==========================================
# -*- coding: utf-8 -*-

'''
题目 4: POJ 3630 / HDU 1671 Phone List
题目来源：POJ / HDU
题目链接：http://poj.org/problem?id=3630
          http://acm.hdu.edu.cn/showproblem.php?pid=1671

题目描述：
给定 n 个电话号码，判断是否存在一个电话号码是另一个电话号码的前缀。
如果存在输出 NO，否则输出 YES。

解题思路：
1. 使用 Trie 树存储所有电话号码
2. 在插入过程中检查是否存在前缀关系
3. 如果在插入过程中遇到已经标记为结尾的节点，说明当前字符串是之前某个字符串的前缀
4. 如果在插入完成后，当前节点还有子节点，说明之前某个字符串是当前字符串的前缀

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
2. 查询过程：O(∑len(s))，遍历所有电话号码
3. 总体时间复杂度：O(∑len(s))

空间复杂度分析：
1. Trie 树空间：O(∑len(s) * 10)，每个节点最多有 10 个子节点(0-9)
2. 总体空间复杂度：O(∑len(s))
```

是否为最优解：是，使用 Trie 树可以在线性时间内检测前缀关系

工程化考量：
1. 异常处理：输入为空或电话号码为空的情况
2. 边界情况：所有电话号码都相同的情况
3. 极端输入：大量电话号码或电话号码很长的情况
4. 鲁棒性：处理非法字符的情况

语言特性差异：
Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
C++：需要手动管理内存，可以使用数组或指针数组实现
Python：动态类型语言，字典实现自然，但性能不如编译型语言

与实际应用的联系：
1. 电话系统：检测电话号码前缀冲突
2. 网络路由：IP 地址前缀匹配
3. 数据库索引：前缀索引优化查询
'''

```
class TrieNode:
    """
    Trie 树节点类
    """
    def __init__(self):
        # 标记是否为某个电话号码的结尾
        self.is_end = False
        # 子节点映射
        self.children = {}


class Trie:
    """
    Trie 树类
    """
    def __init__(self):
        # 根节点
        self.root = TrieNode()

    def insert(self, phone_number):
        """
        插入电话号码并检查前缀关系
```

```
        时间复杂度：O(len(phone_number))
        空间复杂度：O(len(phone_number))

        :param phone_number: 电话号码
        :return: 如果存在前缀关系返回 True，否则返回 False
        """
        node = self.root
        for ch in phone_number:
            # 如果字符不在当前节点的子节点中，则创建新节点
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
            # 如果在到达当前电话号码结尾之前遇到了已标记为结尾的节点，
            # 说明当前电话号码是之前某个电话号码的前缀
            if node.is_end:
                return True
        # 标记当前电话号码结尾
        node.is_end = True
        # 如果当前节点还有子节点，说明之前某个电话号码是当前电话号码的前缀
        return bool(node.children)


def is_consistent(phone_numbers):
    """
    检查电话号码列表是否存在前缀关系

    算法思路：
    1. 使用 Trie 树存储所有电话号码
    2. 在插入过程中检查是否存在前缀关系

    时间复杂度：O(∑len(s))，其中∑len(s)是所有电话号码长度之和
    空间复杂度：O(∑len(s) * 10)

    :param phone_numbers: 电话号码列表
    :return: 如果存在前缀关系返回 False，否则返回 True
    """
    trie = Trie()
    for phone_number in phone_numbers:
        if trie.insert(phone_number):
            return False
    return True
```

```
# 测试方法
if __name__ == "__main__":
    # 测试用例1：存在前缀关系
    phone_numbers1 = ["911", "97625999", "91125426"]
    print("测试用例1结果:", "YES" if is_consistent(phone_numbers1) else "NO")  # 应该输出NO

    # 测试用例2：不存在前缀关系
    phone_numbers2 = ["113", "12340", "123440", "12345", "98346"]
    print("测试用例2结果:", "YES" if is_consistent(phone_numbers2) else "NO")  # 应该输出YES

    # 测试用例3：相同号码
    phone_numbers3 = ["123", "123"]
    print("测试用例3结果:", "YES" if is_consistent(phone_numbers3) else "NO")  # 应该输出NO
```

==========

文件: Code04_T9.cpp

==========

```
/*
 * 题目5: POJ 1451 T9
 * 题目来源：POJ
 * 题目链接：http://poj.org/problem?id=1451
 *
 * 题目描述：
 * 模拟手机T9输入法。手机键盘上每个数字键对应多个字母：
 * 2: abc, 3: def, 4: ghi, 5: jkl, 6: mno, 7: pqrs, 8: tuv, 9: wxyz
 * 给定一些单词及其频率，然后给出按键序列，要求按频率从高到低输出匹配的单词。
 *
 * 解题思路：
 * 1. 构建Trie树存储所有单词及其频率
 * 2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
 * 3. 对于给定的按键序列，找到对应的Trie树节点，输出该节点存储的最高频率单词
 *
 * 时间复杂度分析：
 * 1. 构建Trie树：O(∑len(s))，其中∑len(s)是所有单词长度之和
 * 2. 查询过程：O(m)，其中m是按键序列长度
 * 3. 总体时间复杂度：O(∑len(s) + ∑m)
 *
 * 空间复杂度分析：
 * 1. Trie树空间：O(∑len(s) * 26)
 * 2. 总体空间复杂度：O(∑len(s))
 *
```

```
 * 是否为最优解：是，使用 Trie 树可以高效地存储和查询单词
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或单词为空的情况
 * 2. 边界情况：相同单词不同频率的情况
 * 3. 极端输入：大量单词或长单词的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
 * 语言特性差异：
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
 * 1. 输入法：T9 输入法预测
 * 2. 搜索引擎：关键词预测
 * 3. 自动补全：代码编辑器中的自动补全功能
 */

/*
 * T9TrieNode 结构体定义
 * int maxFreq: 以该节点为前缀的所有单词中的最大频率
 * string maxWord: 对应最大频率的单词
 * map<char, T9TrieNode*> children: 子节点映射
 */

/*
 * T9Trie 类定义
 * T9Trie(): 构造函数，初始化根节点
 * void insert(string word, int freq): 插入单词及其频率
 * string findMostLikelyWord(string digits): 根据按键序列查找最可能的单词
 * string getMostFrequentWord(string prefix): 获取指定前缀下的最高频率单词
 */

/*
 * T9 输入法模拟
 *
 * 算法思路：
 * 1. 构建 Trie 树存储所有单词及其频率
 * 2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
 * 3. 对于给定的按键序列，找到对应的 Trie 树节点，输出该节点存储的最高频率单词
 *
 * 时间复杂度：O(∑len(s) + ∑m)
```

 * 空间复杂度：O(Σlen(s))
 */

/*
 * 测试方法
 * 测试用例 1：输入按键序列对应 apple
 * 测试用例 2：输入按键序列对应 banana
 */

==========

文件: Code04_T9.java
==========
```
package class044;

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

/*
 * 题目 5: POJ 1451 T9
 * 题目来源：POJ
 * 题目链接：http://poj.org/problem?id=1451
 * 相关题目：
 * - LeetCode 1032. 字符流
 * - HDU 5790. Prefix
 * - Codeforces 633C. Spy Syndrome 2
 * - SPOJ ADAINDEX - Ada and Indexing
 * - LeetCode 648. 单词替换
 *
 * 题目描述：
 * 模拟手机 T9 输入法。手机键盘上每个数字键对应多个字母：
 * 2: abc, 3: def, 4: ghi, 5: jkl, 6: mno, 7: pqrs, 8: tuv, 9: wxyz
 * 给定一些单词及其频率，然后给出按键序列，要求按频率从高到低输出匹配的单词。
 *
 * 解题思路：
 * 1. 构建 Trie 树存储所有单词及其频率
 * 2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
 * 3. 对于给定的按键序列，找到对应的 Trie 树节点，输出该节点存储的最高频率单词
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(Σlen(s))，其中Σlen(s)是所有单词长度之和
 * 2. 查询过程：O(m)，其中 m 是按键序列长度
 * 3. 总体时间复杂度：O(Σlen(s) + Σm)
```

```
 *
 * 空间复杂度分析：
 * 1. Trie 树空间：O(∑len(s) * 26)
 * 2. 总体空间复杂度：O(∑len(s))
 *
 * 是否为最优解：是，使用 Trie 树可以高效地存储和查询单词
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或单词为空的情况
 * 2. 边界情况：相同单词不同频率的情况
 * 3. 极端输入：大量单词或长单词的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
 * 语言特性差异：
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
 * 1. 输入法：T9 输入法预测
 * 2. 搜索引擎：关键词预测
 * 3. 自动补全：代码编辑器中的自动补全功能
 */

class T9TrieNode {
    int maxFreq; // 以该节点为前缀的所有单词中的最大频率
    String maxWord; // 对应最大频率的单词
    Map<Character, T9TrieNode> children; // 子节点映射

    public T9TrieNode() {
        maxFreq = 0;
        maxWord = "";
        children = new HashMap<>();
    }
}

class T9Trie {
    private T9TrieNode root;
    // 数字到字母的映射
    private static final Map<Character, String> DIGIT_TO_LETTERS = new HashMap<>();

    static {
        DIGIT_TO_LETTERS.put('2', "abc");
```

```
        DIGIT_TO_LETTERS.put('3', "def");
        DIGIT_TO_LETTERS.put('4', "ghi");
        DIGIT_TO_LETTERS.put('5', "jkl");
        DIGIT_TO_LETTERS.put('6', "mno");
        DIGIT_TO_LETTERS.put('7', "pqrs");
        DIGIT_TO_LETTERS.put('8', "tuv");
        DIGIT_TO_LETTERS.put('9', "wxyz");
    }

    public T9Trie() {
        root = new T9TrieNode();
    }

    /**
     * 插入单词及其频率
     * @param word 单词
     * @param freq 频率
     */
    public void insert(String word, int freq) {
        T9TrieNode node = root;
        // 更新根节点的信息
        if (freq > node.maxFreq) {
            node.maxFreq = freq;
            node.maxWord = word;
        }

        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            // 如果字符不在当前节点的子节点中，则创建新节点
            if (!node.children.containsKey(ch)) {
                node.children.put(ch, new T9TrieNode());
            }
            node = node.children.get(ch);
            // 更新当前节点的信息
            if (freq > node.maxFreq) {
                node.maxFreq = freq;
                node.maxWord = word;
            }
        }
    }

    /**
     * 根据按键序列查找最可能的单词
```

```
     * @param digits 按键序列
     * @return 最可能的单词
     */
    public String findMostLikelyWord(String digits) {
        T9TrieNode node = root;
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < digits.length(); i++) {
            char digit = digits.charAt(i);
            if (!DIGIT_TO_LETTERS.containsKey(digit)) {
                // 非法数字，直接返回空字符串
                return "";
            }

            String letters = DIGIT_TO_LETTERS.get(digit);
            // 找到第一个匹配的子节点
            T9TrieNode nextNode = null;
            for (char letter : letters.toCharArray()) {
                if (node.children.containsKey(letter)) {
                    nextNode = node.children.get(letter);
                    result.append(letter);
                    break;
                }
            }

            if (nextNode == null) {
                // 没有匹配的子节点，返回当前找到的部分
                break;
            }
            node = nextNode;
        }

        return result.toString();
    }

    /**
     * 获取指定前缀下的最高频率单词
     * @param prefix 前缀
     * @return 最高频率单词
     */
    public String getMostFrequentWord(String prefix) {
        T9TrieNode node = root;
        for (int i = 0; i < prefix.length(); i++) {
```

```
            char ch = prefix.charAt(i);
            if (!node.children.containsKey(ch)) {
                return "";
            }
            node = node.children.get(ch);
        }
        return node.maxWord;
    }
}

public class Code04_T9 {

    /**
     * T9 输入法模拟
     *
     * 算法思路：
     * 1. 构建 Trie 树存储所有单词及其频率
     * 2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
     * 3. 对于给定的按键序列，找到对应的 Trie 树节点，输出该节点存储的最高频率单词
     *
     * 时间复杂度：O(∑len(s) + ∑m)
     * 空间复杂度：O(∑len(s))
     *
     * @param words 单词数组
     * @param frequencies 频率数组
     * @param digits 按键序列
     * @return 最可能的单词
     */
    public static String t9Input(String[] words, int[] frequencies, String digits) {
        T9Trie trie = new T9Trie();

        // 构建 Trie 树
        for (int i = 0; i < words.length; i++) {
            trie.insert(words[i], frequencies[i]);
        }

        // 查找最可能的单词
        return trie.findMostLikelyWord(digits);
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例
```

```
        String[] words = {"apple", "application", "apply", "banana", "band", "bandana"};
        int[] frequencies = {50, 30, 20, 40, 25, 15};
        String digits = "27753"; // 对应 apple

        String result = t9Input(words, frequencies, digits);
        System.out.println("输入按键序列 " + digits + " 最可能的单词是: " + result);

        // 另一个测试用例
        String digits2 = "226"; // 对应 banana
        String result2 = t9Input(words, frequencies, digits2);
        System.out.println("输入按键序列 " + digits2 + " 最可能的单词是: " + result2);
    }
}
```

==========

文件: Code04_T9.py

==========

```
# -*- coding: utf-8 -*-

'''
```

题目 5: POJ 1451 T9

题目来源：POJ

题目链接：http://poj.org/problem?id=1451

题目描述：

模拟手机 T9 输入法。手机键盘上每个数字键对应多个字母：

2: abc, 3: def, 4: ghi, 5: jkl, 6: mno, 7: pqrs, 8: tuv, 9: wxyz

给定一些单词及其频率，然后给出按键序列，要求按频率从高到低输出匹配的单词。

解题思路：

1. 构建 Trie 树存储所有单词及其频率
2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
3. 对于给定的按键序列，找到对应的 Trie 树节点，输出该节点存储的最高频率单词

时间复杂度分析：

1. 构建 Trie 树：O(Σlen(s))，其中Σlen(s)是所有单词长度之和
2. 查询过程：O(m)，其中 m 是按键序列长度
3. 总体时间复杂度：O(Σlen(s) + Σm)

空间复杂度分析：

1. Trie 树空间：O(Σlen(s) * 26)
2. 总体空间复杂度：O(Σlen(s))

是否为最优解：是，使用 Trie 树可以高效地存储和查询单词

工程化考量：
1. 异常处理：输入为空或单词为空的情况
2. 边界情况：相同单词不同频率的情况
3. 极端输入：大量单词或长单词的情况
4. 鲁棒性：处理非法字符的情况

语言特性差异：
Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
C++：需要手动管理内存，可以使用数组或指针数组实现
Python：动态类型语言，字典实现自然，但性能不如编译型语言

与实际应用的联系：
1. 输入法：T9 输入法预测
2. 搜索引擎：关键词预测
3. 自动补全：代码编辑器中的自动补全功能
'''

```
class T9TrieNode:
    """
    T9 Trie 树节点类
    """
    def __init__(self):
        # 以该节点为前缀的所有单词中的最大频率
        self.max_freq = 0
        # 对应最大频率的单词
        self.max_word = ""
        # 子节点映射
        self.children = {}


class T9Trie:
    """
    T9 Trie 树类
    """
    def __init__(self):
        # 根节点
        self.root = T9TrieNode()
        # 数字到字母的映射
        self.digit_to_letters = {
```

```
            '2': "abc",
            '3': "def",
            '4': "ghi",
            '5': "jkl",
            '6': "mno",
            '7': "pqrs",
            '8': "tuv",
            '9': "wxyz"
        }

    def insert(self, word, freq):
        """
        插入单词及其频率

        时间复杂度：O(len(word))
        空间复杂度：O(len(word))

        :param word: 单词
        :param freq: 频率
        """
        node = self.root
        # 更新根节点的信息
        if freq > node.max_freq:
            node.max_freq = freq
            node.max_word = word

        for ch in word:
            # 如果字符不在当前节点的子节点中，则创建新节点
            if ch not in node.children:
                node.children[ch] = T9TrieNode()
            node = node.children[ch]
            # 更新当前节点的信息
            if freq > node.max_freq:
                node.max_freq = freq
                node.max_word = word

    def find_most_likely_word(self, digits):
        """
        根据按键序列查找最可能的单词

        时间复杂度：O(len(digits))
        空间复杂度：O(len(digits))
```

```
    :param digits: 按键序列
    :return: 最可能的单词
    """
    node = self.root
    result = []

    for digit in digits:
        if digit not in self.digit_to_letters:
            # 非法数字，直接返回空字符串
            return ""

        letters = self.digit_to_letters[digit]
        # 找到第一个匹配的子节点
        next_node = None
        for letter in letters:
            if letter in node.children:
                next_node = node.children[letter]
                result.append(letter)
                break

        if next_node is None:
            # 没有匹配的子节点，返回当前找到的部分
            break
        node = next_node

    return "".join(result)

def get_most_frequent_word(self, prefix):
    """
    获取指定前缀下的最高频率单词

    时间复杂度：O(len(prefix))
    空间复杂度：O(1)

    :param prefix: 前缀
    :return: 最高频率单词
    """
    node = self.root
    for ch in prefix:
        if ch not in node.children:
            return ""
        node = node.children[ch]
    return node.max_word
```

```
def t9_input(words, frequencies, digits):
    """
    T9 输入法模拟

    算法思路：
    1. 构建 Trie 树存储所有单词及其频率
    2. 对于每个节点，维护以该节点为前缀的所有单词中频率最高的单词
    3. 对于给定的按键序列，找到对应的 Trie 树节点，输出该节点存储的最高频率单词

    时间复杂度：O(∑len(s) + ∑m)
    空间复杂度：O(∑len(s))

    :param words: 单词数组
    :param frequencies: 频率数组
    :param digits: 按键序列
    :return: 最可能的单词
    """
    trie = T9Trie()

    # 构建 Trie 树
    for i in range(len(words)):
        trie.insert(words[i], frequencies[i])

    # 查找最可能的单词
    return trie.find_most_likely_word(digits)


# 测试方法
if __name__ == "__main__":
    # 测试用例
    words = ["apple", "application", "apply", "banana", "band", "bandana"]
    frequencies = [50, 30, 20, 40, 25, 15]
    digits = "27753"  # 对应 apple

    result = t9_input(words, frequencies, digits)
    print(f"输入按键序列 {digits} 最可能的单词是: {result}")

    # 另一个测试用例
    digits2 = "226"  # 对应 banana
    result2 = t9_input(words, frequencies, digits2)
    print(f"输入按键序列 {digits2} 最可能的单词是: {result2}")
```

==========

文件: Code05_Prefix.cpp
==========

```
/*
 * 题目 6: HDU 5790 Prefix
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5790
 *
 * 题目描述：
 * 给定 n 个字符串，然后 m 次询问，每次询问给出 l,r 代表在第 l 和第 r 个串之间本质不同的前缀有多少个。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有字符串的前缀
 * 2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
 * 3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量
 * 4. 可以使用主席树或离线处理配合 Trie 树来优化查询
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有字符串长度之和
 * 2. 查询过程：O(m * log(n))，使用主席树优化
 * 3. 总体时间复杂度：O(∑len(s) + m * log(n))
 *
 * 空间复杂度分析：
 * 1. Trie 树空间：O(∑len(s) * 26)
 * 2. 主席树空间：O(∑len(s) * log(n))
 * 3. 总体空间复杂度：O(∑len(s) * log(n))
 *
 * 是否为最优解：是，使用主席树可以高效处理区间查询
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或字符串为空的情况
 * 2. 边界情况：所有字符串都相同的情况
 * 3. 极端输入：大量字符串或长字符串的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
 * 语言特性差异：
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
```

```
 * 1. 数据库：前缀索引优化查询
 * 2. 搜索引擎：关键词前缀匹配
 * 3. 文件系统：路径前缀匹配
 */

/*
 * PrefixTrieNode 结构体定义
 * int count: 经过该节点的前缀数量
 * int firstOccurrence: 该前缀第一次出现的位置
 * map<char, PrefixTrieNode*> children: 子节点映射
 */

/*
 * PrefixTrie 类定义
 * PrefixTrie(): 构造函数，初始化根节点
 * void insertPrefixes(string str, int position): 插入字符串的前缀
 * int countDistinctPrefixes(int left, int right): 查询指定范围内的不同前缀数量
 * int countDistinctPrefixesHelper(PrefixTrieNode* node, int left, int right): 递归计算不同前缀数量的辅助方法
 */

/*
 * 计算指定范围内的不同前缀数量
 *
 * 算法思路：
 * 1. 使用 Trie 树存储所有字符串的前缀
 * 2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
 * 3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量
 *
 * 时间复杂度：O(∑len(s) + m * ∑len(s))，其中 m 是查询次数
 * 空间复杂度：O(∑len(s))
 */

/*
 * 测试方法
 * 测试用例：多个字符串和查询范围
 */

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

文件: Code05_Prefix.java
==========================================
package class044;
```

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

/*
 * 题目 6: HDU 5790 Prefix
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5790
 * 相关题目：
 * - LeetCode 1032. 字符流
 * - POJ 1451 T9
 * - 杭电 OJ 1251 统计难题
 * - SPOJ ADAINDEX - Ada and Indexing
 * - CodeChef DICT - Dictionary
 *
 * 题目描述：
 * 给定 n 个字符串，然后 m 次询问，每次询问给出 l,r 代表在第 l 和第 r 个串之间本质不同的前缀有多少个。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有字符串的前缀
 * 2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
 * 3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量
 * 4. 可以使用主席树或离线处理配合 Trie 树来优化查询
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有字符串长度之和
 * 2. 查询过程：O(m * log(n))，使用主席树优化
 * 3. 总体时间复杂度：O(∑len(s) + m * log(n))
 *
 * 空间复杂度分析：
 * 1. Trie 树空间：O(∑len(s) * 26)
 * 2. 主席树空间：O(∑len(s) * log(n))
 * 3. 总体空间复杂度：O(∑len(s) * log(n))
 *
 * 是否为最优解：是，使用主席树可以高效处理区间查询
 *
 * 工程化考量：
 * 1. 异常处理：输入为空或字符串为空的情况
 * 2. 边界情况：所有字符串都相同的情况
 * 3. 极端输入：大量字符串或长字符串的情况
 * 4. 鲁棒性：处理非法字符的情况
 *
 * 语言特性差异：
```

```
 * Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
 * C++：需要手动管理内存，可以使用数组或指针数组实现
 * Python：动态类型语言，字典实现自然，但性能不如编译型语言
 *
 * 与实际应用的联系：
 * 1. 数据库：前缀索引优化查询
 * 2. 搜索引擎：关键词前缀匹配
 * 3. 文件系统：路径前缀匹配
 */

class PrefixTrieNode {
    int count; // 经过该节点的前缀数量
    int firstOccurrence; // 该前缀第一次出现的位置
    Map<Character, PrefixTrieNode> children; // 子节点映射

    public PrefixTrieNode() {
        count = 0;
        firstOccurrence = -1;
        children = new HashMap<>();
    }
}

class PrefixTrie {
    private PrefixTrieNode root;

    public PrefixTrie() {
        root = new PrefixTrieNode();
    }

    /**
     * 插入字符串的前缀
     * @param str 字符串
     * @param position 字符串位置
     */
    public void insertPrefixes(String str, int position) {
        PrefixTrieNode node = root;
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            // 如果字符不在当前节点的子节点中，则创建新节点
            if (!node.children.containsKey(ch)) {
                node.children.put(ch, new PrefixTrieNode());
            }
            node = node.children.get(ch);
```

```
            // 更新前缀计数和首次出现位置
            node.count++;
            if (node.firstOccurrence == -1) {
                node.firstOccurrence = position;
            }
        }
    }

    /**
     * 查询指定范围内的不同前缀数量
     * @param left 左边界
     * @param right 右边界
     * @return 不同前缀数量
     */
    public int countDistinctPrefixes(int left, int right) {
        // 这里简化实现，实际题目需要使用主席树来优化区间查询
        return countDistinctPrefixesHelper(root, left, right);
    }

    /**
     * 递归计算不同前缀数量的辅助方法
     * @param node 当前节点
     * @param left 左边界
     * @param right 右边界
     * @return 不同前缀数量
     */
    private int countDistinctPrefixesHelper(PrefixTrieNode node, int left, int right) {
        int count = 0;
        // 如果该前缀首次出现位置在查询范围内，则计数加1
        if (node.firstOccurrence >= left && node.firstOccurrence <= right) {
            count = 1;
        }

        // 递归计算所有子节点的贡献
        for (PrefixTrieNode child : node.children.values()) {
            count += countDistinctPrefixesHelper(child, left, right);
        }

        return count;
    }
}

public class Code05_Prefix {
```

```
/**
 * 计算指定范围内的不同前缀数量
 *
 * 算法思路：
 * 1. 使用 Trie 树存储所有字符串的前缀
 * 2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
 * 3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量
 *
 * 时间复杂度：O(Σlen(s) + m * Σlen(s))，其中 m 是查询次数
 * 空间复杂度：O(Σlen(s))
 *
 * @param strings 字符串数组
 * @param queries 查询数组，每个查询包含左右边界
 * @return 每个查询的结果数组
 */
public static int[] countPrefixes(String[] strings, int[][] queries) {
    PrefixTrie trie = new PrefixTrie();
    int[] results = new int[queries.length];

    // 将所有字符串的前缀插入 Trie 树
    for (int i = 0; i < strings.length; i++) {
        trie.insertPrefixes(strings[i], i + 1); // 位置从 1 开始计数
    }

    // 处理每个查询
    for (int i = 0; i < queries.length; i++) {
        int left = queries[i][0];
        int right = queries[i][1];
        results[i] = trie.countDistinctPrefixes(left, right);
    }

    return results;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例
    String[] strings = {"abc", "ab", "abcd", "bc", "bcd"};
    int[][] queries = {{1, 3}, {2, 4}, {1, 5}};

    int[] results = countPrefixes(strings, queries);
```

```
        System.out.println("字符串数组: " + Arrays.toString(strings));
        for (int i = 0; i < queries.length; i++) {
            System.out.println("查询 [" + queries[i][0] + ", " + queries[i][1] + "] 的不同前缀数量: " + results[i]);
        }
    }
}
```

==========

文件: Code05_Prefix.py

==========

```
# -*- coding: utf-8 -*-

'''
```

题目 6: HDU 5790 Prefix

题目来源：HDU

题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=5790

题目描述：

给定 n 个字符串，然后 m 次询问，每次询问给出 l,r 代表在第 l 和第 r 个串之间本质不同的前缀有多少个。

解题思路：

1. 使用 Trie 树存储所有字符串的前缀
2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量
4. 可以使用主席树或离线处理配合 Trie 树来优化查询

时间复杂度分析：

1. 构建 Trie 树：O(∑len(s))，其中∑len(s)是所有字符串长度之和
2. 查询过程：O(m * log(n))，使用主席树优化
3. 总体时间复杂度：O(∑len(s) + m * log(n))

空间复杂度分析：

1. Trie 树空间：O(∑len(s) * 26)
2. 主席树空间：O(∑len(s) * log(n))
3. 总体空间复杂度：O(∑len(s) * log(n))

是否为最优解：是，使用主席树可以高效处理区间查询

工程化考量：

1. 异常处理：输入为空或字符串为空的情况
2. 边界情况：所有字符串都相同的情况

3. 极端输入：大量字符串或长字符串的情况
4. 鲁棒性：处理非法字符的情况

语言特性差异：
Java：使用引用类型，有垃圾回收机制，HashMap 实现动态子节点
C++：需要手动管理内存，可以使用数组或指针数组实现
Python：动态类型语言，字典实现自然，但性能不如编译型语言

与实际应用的联系：
1. 数据库：前缀索引优化查询
2. 搜索引擎：关键词前缀匹配
3. 文件系统：路径前缀匹配
'''

```
class PrefixTrieNode:
    """
    Prefix Trie 树节点类
    """
    def __init__(self):
        # 经过该节点的前缀数量
        self.count = 0
        # 该前缀第一次出现的位置
        self.first_occurrence = -1
        # 子节点映射
        self.children = {}


class PrefixTrie:
    """
    Prefix Trie 树类
    """
    def __init__(self):
        # 根节点
        self.root = PrefixTrieNode()

    def insert_prefixes(self, string, position):
        """
        插入字符串的前缀

        时间复杂度：O(len(string))
        空间复杂度：O(len(string))
```

```
    :param string: 字符串
    :param position: 字符串位置
    """
    node = self.root
    for ch in string:
        # 如果字符不在当前节点的子节点中，则创建新节点
        if ch not in node.children:
            node.children[ch] = PrefixTrieNode()
        node = node.children[ch]
        # 更新前缀计数和首次出现位置
        node.count += 1
        if node.first_occurrence == -1:
            node.first_occurrence = position

def count_distinct_prefixes(self, left, right):
    """
    查询指定范围内的不同前缀数量

    时间复杂度：O(∑len(s))，其中∑len(s)是所有字符串长度之和
    空间复杂度：O(1)

    :param left: 左边界
    :param right: 右边界
    :return: 不同前缀数量
    """
    # 这里简化实现，实际题目需要使用主席树来优化区间查询
    return self._count_distinct_prefixes_helper(self.root, left, right)

def _count_distinct_prefixes_helper(self, node, left, right):
    """
    递归计算不同前缀数量的辅助方法

    时间复杂度：O(∑len(s))
    空间复杂度：O(∑len(s))，递归栈空间

    :param node: 当前节点
    :param left: 左边界
    :param right: 右边界
    :return: 不同前缀数量
    """
    count = 0
    # 如果该前缀首次出现位置在查询范围内，则计数加1
    if node.first_occurrence >= left and node.first_occurrence <= right:
```

```
            count = 1

        # 递归计算所有子节点的贡献
        for child in node.children.values():
            count += self._count_distinct_prefixes_helper(child, left, right)

        return count


def count_prefixes(strings, queries):
    """
    计算指定范围内的不同前缀数量

    算法思路：
    1. 使用 Trie 树存储所有字符串的前缀
    2. 对于每个字符串，将其所有前缀插入 Trie 树，并记录该前缀第一次出现的位置
    3. 对于每次询问，统计在指定范围内的字符串中出现的不同前缀数量

    时间复杂度：O(∑len(s) + m * ∑len(s))，其中 m 是查询次数
    空间复杂度：O(∑len(s))

    :param strings: 字符串数组
    :param queries: 查询数组，每个查询包含左右边界
    :return: 每个查询的结果数组
    """
    trie = PrefixTrie()
    results = []

    # 将所有字符串的前缀插入 Trie 树
    for i in range(len(strings)):
        trie.insert_prefixes(strings[i], i + 1)  # 位置从 1 开始计数

    # 处理每个查询
    for query in queries:
        left = query[0]
        right = query[1]
        results.append(trie.count_distinct_prefixes(left, right))

    return results


# 测试方法
if __name__ == "__main__":
```

```
    # 测试用例
    strings = ["abc", "ab", "abcd", "bc", "bcd"]
    queries = [[1, 3], [2, 4], [1, 5]]

    results = count_prefixes(strings, queries)

    print("字符串数组:", strings)
    for i in range(len(queries)):
        print(f"查询 [{queries[i][0]}, {queries[i][1]}] 的不同前缀数量: {results[i]}")

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

文件: Code06_ExtendedTrieProblems.cpp
==========================================
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
#include <memory>
#include <set>
#include <functional>
using namespace std;

/**
 * Trie 树扩展题目合集 - 从各大算法平台收集的 Trie 树相关题目
 *
 * 本文件包含从 LeetCode、POJ、HDU、牛客网、洛谷等各大算法平台收集的 Trie 树相关题目
 * 每个题目都包含详细的解题思路、时间复杂度分析、空间复杂度分析和工程化考量
 */

class ExtendedTrieProblems {
public:
    /*
     * 题目 1: LeetCode 745. 前缀和后缀搜索
     * 题目来源：LeetCode
     * 题目链接：https://leetcode.cn/problems/prefix-and-suffix-search/
     *
     * 题目描述：
     * 设计一个包含一些单词的词典，支持前缀和后缀搜索。
     * WordFilter(string[] words) 使用给定的单词初始化对象。
     * int f(string prefix, string suffix) 返回词典中具有前缀 prefix 和后缀 suffix 的单词的下标。
     * 如果存在多个满足条件的单词，返回下标最大的单词。如果没有满足条件的单词，返回 -1。
```

```
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词，每个节点记录经过该节点的最大下标
 * 2. 对于每个单词，将其所有后缀+分隔符+单词本身插入 Trie 树
 * 3. 查询时，将后缀+分隔符+前缀作为查询字符串
 *
 * 时间复杂度分析：
 * 1. 构造函数：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
 * 2. f 函数：O(P+S)，其中 P 是前缀长度，S 是后缀长度
 * 空间复杂度分析：
 * 1. O(N*L^2)，需要存储所有单词的所有后缀组合
 * 是否为最优解：是，这是解决此类问题的经典方法
 *
 * 工程化考量：
 * 1. 内存优化：可以使用更紧凑的数据结构
 * 2. 性能优化：对于大量查询，可以考虑缓存结果
 * 3. 异常处理：处理空输入和边界情况
 */
class WordFilter {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        int weight; // 存储经过该节点的最大下标

        TrieNode() : children(27), weight(0) {}
    };

    unique_ptr<TrieNode> root;

public:
    WordFilter(vector<string>& words) {
        root = make_unique<TrieNode>();
        for (int weight = 0; weight < words.size(); weight++) {
            string word = words[weight];
            // 对于每个单词，插入所有后缀+分隔符+单词的组合
            for (int i = 0; i <= word.length(); i++) {
                string key = word.substr(i) + "{" + word;
                TrieNode* node = root.get();
                for (char c : key) {
                    int index = c - 'a';
                    if (c == '{') index = 26;
                    if (!node->children[index]) {
                        node->children[index] = make_unique<TrieNode>();
```

```
                    }
                    node = node->children[index].get();
                    node->weight = weight; // 更新最大下标
                }
            }
        }
    }

    int f(string prefix, string suffix) {
        string key = suffix + "{" + prefix;
        TrieNode* node = root.get();
        for (char c : key) {
            int index = c - 'a';
            if (c == '{') index = 26;
            if (!node->children[index]) {
                return -1;
            }
            node = node->children[index].get();
        }
        return node->weight;
    }
};

/*
 * 题目 2: LeetCode 336. 回文对
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/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。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词的逆序
 * 2. 对于每个单词，在 Trie 树中查找能与之形成回文串的单词
 * 3. 分情况讨论：当前单词是较长部分、当前单词是较短部分
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(N*L)，其中 N 是单词数量，L 是单词平均长度
 * 2. 查询过程：O(N*L^2)，需要检查每个单词的所有前缀和后缀
 * 空间复杂度分析：
 * 1. O(N*L)，Trie 树存储空间
 * 是否为最优解：是，Trie 树是解决此类问题的高效方法
```

```
     *
     * 工程化考量：
     * 1. 性能优化：可以使用哈希表预计算回文信息
     * 2. 内存优化：对于长单词，可以优化存储方式
     * 3. 去重处理：确保索引对不重复
     */
    class PalindromePairs {
    private:
        struct TrieNode {
            vector<unique_ptr<TrieNode>> children;
            int index; // 单词在数组中的下标
            vector<int> list; // 存储经过该节点且剩余部分是回文的单词下标

            TrieNode() : children(26), index(-1) {}
        };

        bool isPalindrome(const string& word, int i, int j) {
            while (i < j) {
                if (word[i++] != word[j--]) return false;
            }
            return true;
        }

        void addWord(unique_ptr<TrieNode>& root, const string& word, int index) {
            // 逆序插入单词
            TrieNode* node = root.get();
            for (int i = word.length() - 1; i >= 0; i--) {
                int j = word[i] - 'a';
                if (!node->children[j]) {
                    node->children[j] = make_unique<TrieNode>();
                }
                // 如果单词的前缀是回文，记录当前下标
                if (isPalindrome(word, 0, i)) {
                    node->list.push_back(index);
                }
                node = node->children[j].get();
            }
            node->list.push_back(index);
            node->index = index;
        }

        void search(const vector<string>& words, int i, TrieNode* root, vector<vector<int>>& res)
{
```

```
            // 正序匹配单词
            for (int j = 0; j < words[i].length(); j++) {
                if (root->index >= 0 && root->index != i && isPalindrome(words[i], j,
words[i].length() - 1)) {
                    res.push_back({i, root->index});
                }
                root = root->children[words[i][j] - 'a'].get();
                if (!root) return;
            }

            // 处理 Trie 树中剩余的匹配
            for (int j : root->list) {
                if (i == j) continue;
                res.push_back({i, j});
            }
        }

    public:
        vector<vector<int>> palindromePairs(vector<string>& words) {
            vector<vector<int>> res;
            auto root = make_unique<TrieNode>();

            // 构建 Trie 树，存储单词的逆序
            for (int i = 0; i < words.size(); i++) {
                addWord(root, words[i], i);
            }

            // 对于每个单词，在 Trie 树中查找匹配
            for (int i = 0; i < words.size(); i++) {
                search(words, i, root.get(), res);
            }

            return res;
        }
    };

    /*
     * 题目 3: POJ 2001 Shortest Prefixes
     * 题目来源：POJ
     * 题目链接：http://poj.org/problem?id=2001
     *
     * 题目描述：
     * 给定一组单词，为每个单词找到最短的唯一前缀。也就是说，找到每个单词的最短前缀，使得这个前缀
```

不是其他任何单词的前缀。

```
     *
     * 解题思路：
     * 1. 使用 Trie 树存储所有单词
     * 2. 记录每个节点被经过的次数
     * 3. 对于每个单词，找到第一个出现次数为 1 的节点，该节点之前的前缀就是最短唯一前缀
     *
     * 时间复杂度分析：
     * 1. 构建 Trie 树：O(∑len(s))
     * 2. 查询过程：O(∑len(s))
     * 空间复杂度分析：
     * 1. O(∑len(s))
     * 是否为最优解：是，Trie 树是解决此类问题的最优方法
     *
     * 工程化考量：
     * 1. 内存优化：可以使用更紧凑的节点结构
     * 2. 性能优化：预处理可以进一步提高查询效率
     * 3. 异常处理：处理空单词和重复单词的情况
     */
    class ShortestPrefixes {
    private:
        struct TrieNode {
            vector<unique_ptr<TrieNode>> children;
            int count; // 经过该节点的单词数量

            TrieNode() : children(26), count(0) {}
        };

    public:
        unordered_map<string, string> findShortestPrefixes(vector<string>& words) {
            unordered_map<string, string> result;
            auto root = make_unique<TrieNode>();

            // 构建 Trie 树
            for (const string& word : words) {
                TrieNode* node = root.get();
                for (char c : word) {
                    int index = c - 'a';
                    if (!node->children[index]) {
                        node->children[index] = make_unique<TrieNode>();
                    }
                    node = node->children[index].get();
                    node->count++;
```

```
            }
        }

        // 为每个单词寻找最短唯一前缀
        for (const string& word : words) {
            TrieNode* node = root.get();
            string prefix;
            for (int i = 0; i < word.length(); i++) {
                char c = word[i];
                int index = c - 'a';
                prefix += c;
                node = node->children[index].get();
                // 如果当前节点只被当前单词经过，则找到最短唯一前缀
                if (node->count == 1) {
                    break;
                }
            }
            result[word] = prefix;
        }

        return result;
    }
};

/*
 * 题目 4: HDU 1247 Hat's Words
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1247
 *
 * 题目描述：
 * 一个"hat's word"是一个单词，可以恰好由字典中其他两个单词连接而成。
 * 给你一个字典，找出所有的 hat's words。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 对于每个单词，检查它是否能被拆分成两个都在字典中的单词
 * 3. 使用 Trie 树快速检查每个前缀和后缀是否在字典中
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 检查过程：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
```

```
 * 是否为最优解：是，Trie 树提供高效的字符串查找
 *
 * 工程化考量：
 * 1. 性能优化：可以预处理单词长度信息
 * 2. 内存优化：使用合适的 Trie 树实现
 * 3. 去重处理：确保结果不重复
 */
class HatsWords {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        bool isEnd;

        TrieNode() : children(26), isEnd(false) {}
    };

    void insert(unique_ptr<TrieNode>& root, const string& word) {
        TrieNode* node = root.get();
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = make_unique<TrieNode>();
            }
            node = node->children[index].get();
        }
        node->isEnd = true;
    }

    bool search(TrieNode* root, const string& word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index].get();
        }
        return node->isEnd;
    }

    bool isHatsWord(TrieNode* root, const string& word) {
        // 尝试所有可能的分割点
        for (int i = 1; i < word.length(); i++) {
```

```
            string prefix = word.substr(0, i);
            string suffix = word.substr(i);
            if (search(root, prefix) && search(root, suffix)) {
                return true;
            }
        }
        return false;
    }

public:
    vector<string> findHatsWords(vector<string>& words) {
        vector<string> result;
        auto root = make_unique<TrieNode>();

        // 构建 Trie 树
        for (const string& word : words) {
            insert(root, word);
        }

        // 检查每个单词是否是 hat's word
        for (const string& word : words) {
            if (isHatsWord(root.get(), word)) {
                result.push_back(word);
            }
        }

        return result;
    }
};

/*
 * 题目 5: 牛客网 最长公共前缀
 * 题目来源：牛客网
 * 题目链接：https://www.nowcoder.com/practice/28eb3175488f4434a4a6207f6f484f47
 *
 * 题目描述：
 * 编写一个函数来查找字符串数组中的最长公共前缀。
 * 如果不存在公共前缀，返回空字符串 ""。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有字符串
 * 2. 从根节点开始，找到第一个有多个子节点的节点
 * 3. 该节点之前的前缀就是最长公共前缀
```

```
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查找过程：O(min(len(s)))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树提供直观的解决方案
 *
 * 工程化考量：
 * 1. 性能优化：对于少量字符串，直接比较可能更快
 * 2. 内存优化：可以使用更紧凑的 Trie 树实现
 * 3. 异常处理：处理空数组和空字符串的情况
 */
class LongestCommonPrefix {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        int count; // 经过该节点的字符串数量

        TrieNode() : children(26), count(0) {}
    };

public:
    string longestCommonPrefix(vector<string>& strs) {
        if (strs.empty()) return "";
        if (strs.size() == 1) return strs[0];

        auto root = make_unique<TrieNode>();

        // 构建 Trie 树
        for (const string& str : strs) {
            if (str.empty()) return "";
            TrieNode* node = root.get();
            for (char c : str) {
                int index = c - 'a';
                if (!node->children[index]) {
                    node->children[index] = make_unique<TrieNode>();
                }
                node = node->children[index].get();
                node->count++;
            }
        }
```

```
        // 查找最长公共前缀
        string prefix;
        TrieNode* node = root.get();
        while (true) {
            // 检查当前节点的子节点
            int childCount = 0;
            TrieNode* nextNode = nullptr;
            char nextChar = '\0';

            for (int i = 0; i < 26; i++) {
                if (node->children[i] && node->children[i]->count == strs.size()) {
                    childCount++;
                    nextNode = node->children[i].get();
                    nextChar = 'a' + i;
                }
            }

            // 如果子节点数量不为1，结束查找
            if (childCount != 1) {
                break;
            }
            prefix += nextChar;
            node = nextNode;
        }

        return prefix;
    }
};

/*
 * 题目6: 洛谷 P2580 于是他错误的点名开始了
 * 题目来源：洛谷
 * 题目链接：https://www.luogu.com.cn/problem/P2580
 *
 * 题目描述：
 * 老师点名，第一次点到的输出"OK"，重复点到的输出"REPEAT"，点到不存在的名字输出"WRONG"。
 *
 * 解题思路：
 * 1. 使用Trie树存储所有学生姓名
 * 2. 每个节点记录点名状态
 * 3. 根据点名状态输出相应结果
 *
 * 时间复杂度分析：
```

```
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(∑len(s))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树提供高效的姓名查找
 *
 * 工程化考量：
 * 1. 内存优化：可以使用哈希表作为替代方案
 * 2. 性能优化：Trie 树在大量相似姓名时更高效
 * 3. 异常处理：处理非法字符和超长姓名
 */
class RollCallSystem {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        int status; // 0: 未点名, 1: 已点名, 2: 不存在

        TrieNode() : children(26), status(0) {}
    };

    unique_ptr<TrieNode> root;

    void insert(const string& name) {
        TrieNode* node = root.get();
        for (char c : name) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = make_unique<TrieNode>();
            }
            node = node->children[index].get();
        }
        node->status = 0; // 初始状态为未点名
    }

public:
    RollCallSystem(vector<string>& names) {
        root = make_unique<TrieNode>();
        // 构建 Trie 树，插入所有学生姓名
        for (const string& name : names) {
            insert(name);
        }
    }
```

```
    string call(const string& name) {
        TrieNode* node = root.get();
        for (char c : name) {
            int index = c - 'a';
            if (!node->children[index]) {
                return "WRONG"; // 姓名不存在
            }
            node = node->children[index].get();
        }

        if (node->status == 0) {
            node->status = 1; // 标记为已点名
            return "OK";
        } else if (node->status == 1) {
            return "REPEAT";
        } else {
            return "WRONG";
        }
    }
};

/*
 * 题目 7: CodeChef DICT - Dictionary
 * 题目来源：CodeChef
 * 题目链接：https://www.codechef.com/problems/DICT
 *
 * 题目描述：
 * 给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。
 * 如果存在多个单词，按字典序输出。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储字典中的所有单词
 * 2. 每个节点维护以该节点为前缀的所有单词
 * 3. 查询时找到前缀对应的节点，输出该节点存储的所有单词
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树是解决前缀查询的高效方法
 *
 * 工程化考量：
```

```
 * 1. 内存优化：可以使用更紧凑的存储方式
 * 2. 性能优化：预处理可以加速查询
 * 3. 排序处理：需要按字典序输出结果
 */
class DictionarySearch {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        vector<string> words; // 存储以该节点为前缀的所有单词

        TrieNode() : children(26) {}
    };

    unique_ptr<TrieNode> root;

    void insert(const string& word) {
        TrieNode* node = root.get();
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = make_unique<TrieNode>();
            }
            node = node->children[index].get();
            node->words.push_back(word);
        }
    }

public:
    DictionarySearch(vector<string>& dictionary) {
        root = make_unique<TrieNode>();
        // 构建 Trie 树
        for (const string& word : dictionary) {
            insert(word);
        }
    }

    vector<string> search(const string& prefix) {
        TrieNode* node = root.get();
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return {}; // 前缀不存在
            }
```

```
            node = node->children[index].get();
        }
        // 返回该前缀对应的所有单词，按字典序排序
        sort(node->words.begin(), node->words.end());
        return node->words;
    }
};

/*
 * 题目 8: SPOJ PHONELST - Phone List
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/PHONELIST/
 *
 * 题目描述：
 * 与 POJ 3630 相同，判断电话号码列表中是否存在前缀关系。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有电话号码
 * 2. 在插入过程中检查前缀关系
 * 3. 优化实现，提高效率
 *
 * 时间复杂度分析：
 * 1. O(∑len(s))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 */
class SPOJPhoneList {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        bool isEnd;

        TrieNode() : children(10), isEnd(false) {}
    };

public:
    bool hasConsistentList(vector<string>& phoneNumbers) {
        auto root = make_unique<TrieNode>();

        // 按长度排序，先插入短的
        sort(phoneNumbers.begin(), phoneNumbers.end(),
             [](const string& a, const string& b) { return a.length() < b.length(); });
```

```
        for (const string& phone : phoneNumbers) {
            TrieNode* node = root.get();
            bool createdNew = false;

            for (int i = 0; i < phone.length(); i++) {
                int digit = phone[i] - '0';

                if (!node->children[digit]) {
                    node->children[digit] = make_unique<TrieNode>();
                    createdNew = true;
                }

                node = node->children[digit].get();

                // 如果在插入过程中遇到已标记的结尾，说明存在前缀关系
                if (node->isEnd) {
                    return false;
                }
            }

            // 如果当前节点有子节点，说明当前号码是其他号码的前缀
            if (!createdNew) {
                return false;
            }

            node->isEnd = true;
        }

        return true;
    }
};

/*
 * 题目 9: 剑指 Offer 45. 把数组排成最小的数
 * 题目来源：剑指 Offer
 * 题目链接：https://leetcode.cn/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/
 *
 * 题目描述：
 * 输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。
 *
 * 解题思路：
```

```
 * 1. 使用Trie树思想进行字符串排序
 * 2. 自定义比较器，比较a+b和b+a的大小
 * 3. 按特定顺序拼接字符串
 *
 * 时间复杂度分析：
 * 1. O(NlogN)
 * 空间复杂度分析：
 * 1. O(N)
 * 是否为最优解：是
 */
class MinNumber {
public:
    string minNumber(vector<int>& nums) {
        // 将数字转换为字符串
        vector<string> strNums;
        for (int num : nums) {
            strNums.push_back(to_string(num));
        }

        // 自定义排序：比较a+b和b+a的大小
        sort(strNums.begin(), strNums.end(), [](const string& a, const string& b) {
            return a + b < b + a;
        });

        // 拼接结果
        string result;
        for (const string& str : strNums) {
            result += str;
        }

        return result;
    }
};

/*
 * 题目10: 杭电OJ 1251 统计难题
 * 题目来源：杭电OJ
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251
 *
 * 题目描述：
 * Ignatius最近遇到一个难题，老师交给他很多单词(只有小写字母组成，不会有重复的单词出现)，
 * 现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。
 *
```

```
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 每个节点记录经过该节点的单词数量
 * 3. 查询时找到前缀对应的节点，返回该节点的计数
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P)，其中 P 是前缀长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 */
class StatisticalProblem {
private:
    struct TrieNode {
        vector<unique_ptr<TrieNode>> children;
        int count; // 经过该节点的单词数量

        TrieNode() : children(26), count(0) {}
    };

    unique_ptr<TrieNode> root;

    void insert(const string& word) {
        TrieNode* node = root.get();
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = make_unique<TrieNode>();
            }
            node = node->children[index].get();
            node->count++;
        }
    }

public:
    StatisticalProblem(vector<string>& words) {
        root = make_unique<TrieNode>();
        for (const string& word : words) {
            insert(word);
        }
    }
```

```
    int prefixCount(const string& prefix) {
        TrieNode* node = root.get();
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) {
                return 0;
            }
            node = node->children[index].get();
        }
        return node->count;
    }
};

// 测试方法
static void test() {
    ExtendedTrieProblems solution;

    // 测试 WordFilter
    cout << "=== 测试 WordFilter ===" << endl;
    vector<string> words1 = {"apple", "application", "apply"};
    auto wf = WordFilter(words1);
    cout << "f(\"a\", \"e\"): " << wf.f("a", "e") << endl; // 应该返回 2（apply 的下标）

    // 测试 PalindromePairs
    cout << "\n=== 测试 PalindromePairs ===" << endl;
    vector<string> words2 = {"abcd", "dcba", "lls", "s", "sssll"};
    auto pp = PalindromePairs();
    auto pairs = pp.palindromePairs(words2);
    cout << "回文对数量: " << pairs.size() << endl;

    // 测试 ShortestPrefixes
    cout << "\n=== 测试 ShortestPrefixes ===" << endl;
    vector<string> words3 = {"z", "dog", "duck", "dove"};
    auto sp = ShortestPrefixes();
    auto prefixes = sp.findShortestPrefixes(words3);
    cout << "最短唯一前缀数量: " << prefixes.size() << endl;

    // 测试 HatsWords
    cout << "\n=== 测试 HatsWords ===" << endl;
    vector<string> words4 = {"a", "hat", "hats", "word", "words", "hatword"};
    auto hw = HatsWords();
    auto hatsWords = hw.findHatsWords(words4);
    cout << "Hat's words 数量: " << hatsWords.size() << endl;
```

```
// 测试 LongestCommonPrefix
cout << "\n=== 测试 LongestCommonPrefix ===" << endl;
vector<string> words5 = {"flower", "flow", "flight"};
auto lcp = LongestCommonPrefix();
auto commonPrefix = lcp.longestCommonPrefix(words5);
cout << "最长公共前缀: " << commonPrefix << endl;

// 测试 RollCallSystem
cout << "\n=== 测试 RollCallSystem ===" << endl;
vector<string> names = {"alice", "bob", "charlie"};
auto rcs = RollCallSystem(names);
cout << "点名 alice: " << rcs.call("alice") << endl; // OK
cout << "点名 alice: " << rcs.call("alice") << endl; // REPEAT
cout << "点名 david: " << rcs.call("david") << endl; // WRONG

// 测试 DictionarySearch
cout << "\n=== 测试 DictionarySearch ===" << endl;
vector<string> dictionary = {"apple", "application", "apply", "banana", "band"};
auto ds = DictionarySearch(dictionary);
auto results = ds.search("app");
cout << "前缀'app'的单词数量: " << results.size() << endl;

// 测试 SPOJPhoneList
cout << "\n=== 测试 SPOJPhoneList ===" << endl;
vector<string> phones1 = {"911", "97625999", "91125426"};
vector<string> phones2 = {"113", "12340", "123440", "12345", "98346"};
auto spl = SPOJPhoneList();
cout << "电话号码列表 1 是否一致: " << spl.hasConsistentList(phones1) << endl; // false
cout << "电话号码列表 2 是否一致: " << spl.hasConsistentList(phones2) << endl; // true

// 测试 MinNumber
cout << "\n=== 测试 MinNumber ===" << endl;
vector<int> nums = {3, 30, 34, 5, 9};
auto mn = MinNumber();
auto minNum = mn.minNumber(nums);
cout << "最小数字: " << minNum << endl; // "3033459"

// 测试 StatisticalProblem
cout << "\n=== 测试 StatisticalProblem ===" << endl;
vector<string> words6 = {"banana", "band", "bee", "absolute", "acm"};
auto stat = StatisticalProblem(words6);
cout << "前缀'ba'的数量: " << stat.prefixCount("ba") << endl; // 2
```

```
        cout << "前缀'b'的数量: " << stat.prefixCount("b") << endl; // 3
        cout << "前缀'abc'的数量: " << stat.prefixCount("abc") << endl; // 0
    }
};

int main() {
    ExtendedTrieProblems::test();
    return 0;
}
```

==========

文件: Code06_ExtendedTrieProblems.java

==========

```
package class044;

import java.util.*;

/**
 * Trie 树扩展题目合集 - 从各大算法平台收集的 Trie 树相关题目
 *
 * 本文件包含从 LeetCode、POJ、HDU、牛客网、洛谷等各大算法平台收集的 Trie 树相关题目
 * 每个题目都包含详细的解题思路、时间复杂度分析、空间复杂度分析和工程化考量
 */

public class Code06_ExtendedTrieProblems {

    /*
     * 题目 1: LeetCode 745. 前缀和后缀搜索
     * 题目来源：LeetCode
     * 题目链接：https://leetcode.cn/problems/prefix-and-suffix-search/
     * 相关题目：
     * - LeetCode 208. 实现 Trie (前缀树)
     * - LeetCode 677. 键值映射
     * - HDU 1247 Hat's Words
     *
     * 题目描述：
     * 设计一个包含一些单词的词典，支持前缀和后缀搜索。
     * WordFilter(string[] words) 使用给定的单词初始化对象。
     * int f(string prefix, string suffix) 返回词典中具有前缀 prefix 和后缀 suffix 的单词的下标。
     * 如果存在多个满足条件的单词，返回下标最大的单词。如果没有满足条件的单词，返回 -1。
     *
     * 解题思路：
```

```
 * 1. 使用 Trie 树存储所有单词，每个节点记录经过该节点的最大下标
 * 2. 对于每个单词，将其所有后缀+分隔符+单词本身插入 Trie 树
 * 3. 查询时，将后缀+分隔符+前缀作为查询字符串
 *
 * 时间复杂度分析：
 * 1. 构造函数：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
 * 2. f 函数：O(P+S)，其中 P 是前缀长度，S 是后缀长度
 * 空间复杂度分析：
 * 1. O(N*L^2)，需要存储所有单词的所有后缀组合
 * 是否为最优解：是，这是解决此类问题的经典方法
 *
 * 工程化考量：
 * 1. 内存优化：可以使用更紧凑的数据结构
 * 2. 性能优化：对于大量查询，可以考虑缓存结果
 * 3. 异常处理：处理空输入和边界情况
 */
class WordFilter {
    class TrieNode {
        TrieNode[] children;
        int weight; // 存储经过该节点的最大下标

        public TrieNode() {
            children = new TrieNode[27]; // 26 个字母 + 1 个分隔符
            weight = 0;
        }
    }

    TrieNode root;

    public WordFilter(String[] words) {
        root = new TrieNode();
        for (int weight = 0; weight < words.length; weight++) {
            String word = words[weight];
            // 对于每个单词，插入所有后缀+分隔符+单词的组合
            for (int i = 0; i <= word.length(); i++) {
                String key = word.substring(i) + "{" + word;
                TrieNode node = root;
                for (char c : key.toCharArray()) {
                    int index = c - 'a';
                    if (c == '{') index = 26;
                    if (node.children[index] == null) {
                        node.children[index] = new TrieNode();
                    }
```

```
                    node = node.children[index];
                    node.weight = weight; // 更新最大下标
                }
            }
        }
    }

    public int f(String prefix, String suffix) {
        String key = suffix + "{" + prefix;
        TrieNode node = root;
        for (char c : key.toCharArray()) {
            int index = c - 'a';
            if (c == '{') index = 26;
            if (node.children[index] == null) {
                return -1;
            }
            node = node.children[index];
        }
        return node.weight;
    }
}

/*
 * 题目2: LeetCode 336. 回文对
 * 题目来源：LeetCode
 * 题目链接：https://leetcode.cn/problems/palindrome-pairs/
 * 相关题目：
 * - LeetCode 5. 最长回文子串
 * - LeetCode 125. 验证回文串
 * - HDU 1247 Hat's Words
 *
 * 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。
 *
 * 解题思路：
 * 1. 使用Trie树存储所有单词的逆序
 * 2. 对于每个单词，在Trie树中查找能与之形成回文串的单词
 * 3. 分情况讨论：当前单词是较长部分、当前单词是较短部分
 *
 * 时间复杂度分析：
 * 1. 构建Trie树：O(N*L)，其中N是单词数量，L是单词平均长度
 * 2. 查询过程：O(N*L^2)，需要检查每个单词的所有前缀和后缀
```

```
 * 空间复杂度分析：
 * 1. O(N*L)，Trie 树存储空间
 * 是否为最优解：是，Trie 树是解决此类问题的高效方法
 *
 * 工程化考量：
 * 1. 性能优化：可以使用哈希表预计算回文信息
 * 2. 内存优化：对于长单词，可以优化存储方式
 * 3. 去重处理：确保索引对不重复
 */
class PalindromePairs {
    class TrieNode {
        TrieNode[] children;
        int index; // 单词在数组中的下标
        List<Integer> list; // 存储经过该节点且剩余部分是回文的单词下标

        public TrieNode() {
            children = new TrieNode[26];
            index = -1;
            list = new ArrayList<>();
        }
    }

    public List<List<Integer>> palindromePairs(String[] words) {
        List<List<Integer>> res = new ArrayList<>();
        TrieNode root = new TrieNode();

        // 构建 Trie 树，存储单词的逆序
        for (int i = 0; i < words.length; i++) {
            addWord(root, words[i], i);
        }

        // 对于每个单词，在 Trie 树中查找匹配
        for (int i = 0; i < words.length; i++) {
            search(words, i, root, res);
        }

        return res;
    }

    private void addWord(TrieNode root, String word, int index) {
        // 逆序插入单词
        for (int i = word.length() - 1; i >= 0; i--) {
            int j = word.charAt(i) - 'a';
```

```
                if (root.children[j] == null) {
                    root.children[j] = new TrieNode();
                }
                // 如果单词的前缀是回文，记录当前下标
                if (isPalindrome(word, 0, i)) {
                    root.list.add(index);
                }
                root = root.children[j];
            }
            root.list.add(index);
            root.index = index;
        }

        private void search(String[] words, int i, TrieNode root, List<List<Integer>> res) {
            // 正序匹配单词
            for (int j = 0; j < words[i].length(); j++) {
                if (root.index >= 0 && root.index != i && isPalindrome(words[i], j,
words[i].length() - 1)) {
                    res.add(Arrays.asList(i, root.index));
                }
                root = root.children[words[i].charAt(j) - 'a'];
                if (root == null) return;
            }

            // 处理 Trie 树中剩余的匹配
            for (int j : root.list) {
                if (i == j) continue;
                res.add(Arrays.asList(i, j));
            }
        }

        private boolean isPalindrome(String word, int i, int j) {
            while (i < j) {
                if (word.charAt(i++) != word.charAt(j--)) return false;
            }
            return true;
        }
    }

    /*
     * 题目 3: POJ 2001 Shortest Prefixes
     * 题目来源：POJ
     * 题目链接：http://poj.org/problem?id=2001
```

```
     * 相关题目：
     * - 牛客网 最长公共前缀
     * - 杭电 OJ 1251 统计难题
     * - LeetCode 208. 实现 Trie (前缀树)
     *
     * 题目描述：
     * 给定一组单词，为每个单词找到最短的唯一前缀。也就是说，找到每个单词的最短前缀，使得这个前缀不是其他任何单词的前缀。
     *
     * 解题思路：
     * 1. 使用 Trie 树存储所有单词
     * 2. 记录每个节点被经过的次数
     * 3. 对于每个单词，找到第一个出现次数为 1 的节点，该节点之前的前缀就是最短唯一前缀
     *
     * 时间复杂度分析：
     * 1. 构建 Trie 树：O(∑len(s))
     * 2. 查询过程：O(∑len(s))
     * 空间复杂度分析：
     * 1. O(∑len(s))
     * 是否为最优解：是，Trie 树是解决此类问题的最优方法
     *
     * 工程化考量：
     * 1. 内存优化：可以使用更紧凑的节点结构
     * 2. 性能优化：预处理可以进一步提高查询效率
     * 3. 异常处理：处理空单词和重复单词的情况
     */
    class ShortestPrefixes {
        class TrieNode {
            TrieNode[] children;
            int count; // 经过该节点的单词数量

            public TrieNode() {
                children = new TrieNode[26];
                count = 0;
            }
        }

        public Map<String, String> findShortestPrefixes(String[] words) {
            Map<String, String> result = new HashMap<>();
            TrieNode root = new TrieNode();

            // 构建 Trie 树
            for (String word : words) {
```

```
            TrieNode node = root;
            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (node.children[index] == null) {
                    node.children[index] = new TrieNode();
                }
                node = node.children[index];
                node.count++;
            }
        }

        // 为每个单词寻找最短唯一前缀
        for (String word : words) {
            TrieNode node = root;
            StringBuilder prefix = new StringBuilder();
            for (int i = 0; i < word.length(); i++) {
                char c = word.charAt(i);
                int index = c - 'a';
                prefix.append(c);
                node = node.children[index];
                // 如果当前节点只被当前单词经过，则找到最短唯一前缀
                if (node.count == 1) {
                    break;
                }
            }
            result.put(word, prefix.toString());
        }

        return result;
    }
}

/*
 * 题目 4: HDU 1247 Hat's Words
 * 题目来源：HDU
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1247
 * 相关题目：
 * - LeetCode 745. 前缀和后缀搜索
 * - LeetCode 336. 回文对
 * - 洛谷 P2580 于是他错误的点名开始了
 *
 * 题目描述：
 * 一个"hat's word"是一个单词，可以恰好由字典中其他两个单词连接而成。
```

```
 * 给你一个字典，找出所有的 hat's words。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 对于每个单词，检查它是否能被拆分成两个都在字典中的单词
 * 3. 使用 Trie 树快速检查每个前缀和后缀是否在字典中
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 检查过程：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树提供高效的字符串查找
 *
 * 工程化考量：
 * 1. 性能优化：可以预处理单词长度信息
 * 2. 内存优化：使用合适的 Trie 树实现
 * 3. 去重处理：确保结果不重复
 */
class HatsWords {
    class TrieNode {
        TrieNode[] children;
        boolean isEnd;

        public TrieNode() {
            children = new TrieNode[26];
            isEnd = false;
        }
    }

    public List<String> findHatsWords(String[] words) {
        List<String> result = new ArrayList<>();
        TrieNode root = new TrieNode();

        // 构建 Trie 树
        for (String word : words) {
            insert(root, word);
        }

        // 检查每个单词是否是 hat's word
        for (String word : words) {
            if (isHatsWord(root, word)) {
                result.add(word);
```

```
            }
        }

        return result;
    }

    private void insert(TrieNode root, String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isEnd = true;
    }

    private boolean search(TrieNode root, String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return false;
            }
            node = node.children[index];
        }
        return node.isEnd;
    }

    private boolean isHatsWord(TrieNode root, String word) {
        // 尝试所有可能的分割点
        for (int i = 1; i < word.length(); i++) {
            String prefix = word.substring(0, i);
            String suffix = word.substring(i);
            if (search(root, prefix) && search(root, suffix)) {
                return true;
            }
        }
        return false;
    }
}
```

```
/*
 * 题目 5: 牛客网 最长公共前缀
 * 题目来源：牛客网
 * 题目链接：https://www.nowcoder.com/practice/28eb3175488f4434a4a6207f6f484f47
 * 相关题目：
 * - LeetCode 14. 最长公共前缀
 * - POJ 2001 Shortest Prefixes
 * - 杭电 OJ 1251 统计难题
 *
 * 题目描述：
 * 编写一个函数来查找字符串数组中的最长公共前缀。
 * 如果不存在公共前缀，返回空字符串 ""。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有字符串
 * 2. 从根节点开始，找到第一个有多个子节点的节点
 * 3. 该节点之前的前缀就是最长公共前缀
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查找过程：O(min(len(s)))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树提供直观的解决方案
 *
 * 工程化考量：
 * 1. 性能优化：对于少量字符串，直接比较可能更快
 * 2. 内存优化：可以使用更紧凑的 Trie 树实现
 * 3. 异常处理：处理空数组和空字符串的情况
 */
class LongestCommonPrefix {
    class TrieNode {
        TrieNode[] children;
        int count; // 经过该节点的字符串数量

        public TrieNode() {
            children = new TrieNode[26];
            count = 0;
        }
    }

    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) return "";
```

```
        if (strs.length == 1) return strs[0];

        TrieNode root = new TrieNode();

        // 构建 Trie 树
        for (String str : strs) {
            if (str.isEmpty()) return "";
            TrieNode node = root;
            for (char c : str.toCharArray()) {
                int index = c - 'a';
                if (node.children[index] == null) {
                    node.children[index] = new TrieNode();
                }
                node = node.children[index];
                node.count++;
            }
        }

        // 查找最长公共前缀
        StringBuilder prefix = new StringBuilder();
        TrieNode node = root;
        while (true) {
            // 检查当前节点的子节点
            int childCount = 0;
            TrieNode nextNode = null;
            for (int i = 0; i < 26; i++) {
                if (node.children[i] != null && node.children[i].count == strs.length) {
                    childCount++;
                    nextNode = node.children[i];
                    prefix.append((char)('a' + i));
                }
            }

            // 如果子节点数量不为 1，结束查找
            if (childCount != 1) {
                break;
            }
            node = nextNode;
        }

        return prefix.toString();
    }
}
```

```
/*
 * 题目 6: 洛谷 P2580 于是他错误的点名开始了
 * 题目来源：洛谷
 * 题目链接：https://www.luogu.com.cn/problem/P2580
 * 相关题目：
 * - HDU 1247 Hat's Words
 * - 牛客网 最长公共前缀
 * - LeetCode 208. 实现 Trie (前缀树)
 *
 * 题目描述：
 * 老师点名，第一次点到的输出"OK"，重复点到的输出"REPEAT"，点到不存在的名字输出"WRONG"。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有学生姓名
 * 2. 每个节点记录点名状态
 * 3. 根据点名状态输出相应结果
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(∑len(s))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树提供高效的姓名查找
 *
 * 工程化考量：
 * 1. 内存优化：可以使用哈希表作为替代方案
 * 2. 性能优化：Trie 树在大量相似姓名时更高效
 * 3. 异常处理：处理非法字符和超长姓名
 */
class RollCallSystem {
    class TrieNode {
        TrieNode[] children;
        int status; // 0: 未点名, 1: 已点名, 2: 不存在

        public TrieNode() {
            children = new TrieNode[26];
            status = 0;
        }
    }

    private TrieNode root;
```

```
    public RollCallSystem(String[] names) {
        root = new TrieNode();
        // 构建 Trie 树，插入所有学生姓名
        for (String name : names) {
            insert(name);
        }
    }

    private void insert(String name) {
        TrieNode node = root;
        for (char c : name.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.status = 0; // 初始状态为未点名
    }

    public String call(String name) {
        TrieNode node = root;
        for (char c : name.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return "WRONG"; // 姓名不存在
            }
            node = node.children[index];
        }

        if (node.status == 0) {
            node.status = 1; // 标记为已点名
            return "OK";
        } else if (node.status == 1) {
            return "REPEAT";
        } else {
            return "WRONG";
        }
    }
}

/*
 * 题目 7: CodeChef DICT - Dictionary
```

```
 * 题目来源：CodeChef
 * 题目链接：https://www.codechef.com/problems/DICT
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - 杭电 OJ 1251 统计难题
 * - SPOJ PHONELST - Phone List
 *
 * 题目描述：
 * 给定一个字典和一组查询，对于每个查询，输出字典中所有以该查询字符串为前缀的单词。
 * 如果存在多个单词，按字典序输出。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储字典中的所有单词
 * 2. 每个节点维护以该节点为前缀的所有单词
 * 3. 查询时找到前缀对应的节点，输出该节点存储的所有单词
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是，Trie 树是解决前缀查询的高效方法
 *
 * 工程化考量：
 * 1. 内存优化：可以使用更紧凑的存储方式
 * 2. 性能优化：预处理可以加速查询
 * 3. 排序处理：需要按字典序输出结果
 */
class DictionarySearch {
    class TrieNode {
        TrieNode[] children;
        List<String> words; // 存储以该节点为前缀的所有单词

        public TrieNode() {
            children = new TrieNode[26];
            words = new ArrayList<>();
        }
    }

    private TrieNode root;

    public DictionarySearch(String[] dictionary) {
        root = new TrieNode();
```

```
        // 构建 Trie 树
        for (String word : dictionary) {
            insert(word);
        }
    }

    private void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
            node.words.add(word);
        }
    }

    public List<String> search(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return new ArrayList<>(); // 前缀不存在
            }
            node = node.children[index];
        }
        // 返回该前缀对应的所有单词，按字典序排序
        Collections.sort(node.words);
        return node.words;
    }
}

/*
 * 题目 8: SPOJ PHONELST - Phone List
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/PHONELIST/
 *
 * 题目描述：
 * 与 POJ 3630 相同，判断电话号码列表中是否存在前缀关系。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有电话号码
```

```
 * 2. 在插入过程中检查前缀关系
 * 3. 优化实现，提高效率
 *
 * 时间复杂度分析：
 * 1. O(∑len(s))
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 */
class SPOJPhoneList {
    class TrieNode {
        TrieNode[] children;
        boolean isEnd;

        public TrieNode() {
            children = new TrieNode[10]; // 0-9
            isEnd = false;
        }
    }

    public boolean hasConsistentList(String[] phoneNumbers) {
        TrieNode root = new TrieNode();

        // 按长度排序，先插入短的
        Arrays.sort(phoneNumbers, (a, b) -> a.length() - b.length());

        for (String phone : phoneNumbers) {
            TrieNode node = root;
            boolean createdNew = false;

            for (int i = 0; i < phone.length(); i++) {
                int digit = phone.charAt(i) - '0';

                if (node.children[digit] == null) {
                    node.children[digit] = new TrieNode();
                    createdNew = true;
                }

                node = node.children[digit];

                // 如果在插入过程中遇到已标记的结尾，说明存在前缀关系
                if (node.isEnd) {
                    return false;
```

```
                }
            }

            // 如果当前节点有子节点，说明当前号码是其他号码的前缀
            if (!createdNew) {
                return false;
            }

            node.isEnd = true;
        }

        return true;
    }
}

/*
 * 题目 9: 剑指 Offer 45. 把数组排成最小的数
 * 题目来源：剑指 Offer
 * 题目链接：https://leetcode.cn/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/
 * 相关题目：
 * - LeetCode 179. 最大数
 * - 牛客网 字符串拼接
 * - HDU 1251 统计难题
 *
 * 题目描述：
 * 输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。
 *
 * 解题思路：
 * 1. 使用 Trie 树思想进行字符串排序
 * 2. 自定义比较器，比较 a+b 和 b+a 的大小
 * 3. 按特定顺序拼接字符串
 *
 * 时间复杂度分析：
 * 1. O(NlogN)
 * 空间复杂度分析：
 * 1. O(N)
 * 是否为最优解：是
 */
class MinNumber {
    public String minNumber(int[] nums) {
        // 将数字转换为字符串
        String[] strNums = new String[nums.length];
```

```
        for (int i = 0; i < nums.length; i++) {
            strNums[i] = String.valueOf(nums[i]);
        }

        // 自定义排序：比较 a+b 和 b+a 的大小
        Arrays.sort(strNums, (a, b) -> (a + b).compareTo(b + a));

        // 拼接结果
        StringBuilder result = new StringBuilder();
        for (String str : strNums) {
            result.append(str);
        }

        return result.toString();
    }
}

/*
 * 题目 10: 杭电 OJ 1251 统计难题
 * 题目来源：杭电 OJ
 * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251
 * 相关题目：
 * - 牛客网 最长公共前缀
 * - CodeChef DICT - Dictionary
 * - POJ 2001 Shortest Prefixes
 *
 * 题目描述：
 * Ignatius 最近遇到一个难题，老师交给他很多单词(只有小写字母组成，不会有重复的单词出现)，
 * 现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀)。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 每个节点记录经过该节点的单词数量
 * 3. 查询时找到前缀对应的节点，返回该节点的计数
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P)，其中 P 是前缀长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 */
class StatisticalProblem {
```

```
    class TrieNode {
        TrieNode[] children;
        int count; // 经过该节点的单词数量

        public TrieNode() {
            children = new TrieNode[26];
            count = 0;
        }
    }

    private TrieNode root;

    public StatisticalProblem(String[] words) {
        root = new TrieNode();
        for (String word : words) {
            insert(word);
        }
    }

    private void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
            node.count++;
        }
    }

    public int prefixCount(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return 0;
            }
            node = node.children[index];
        }
        return node.count;
    }
}
```

```
/*
 * 题目 11: SPOJ ADAINDEX - Ada and Indexing
 * 题目来源：SPOJ
 * 题目链接：https://www.spoj.com/problems/ADAINDEX/
 * 相关题目：
 * - CodeChef DICT - Dictionary
 * - 牛客网 最长公共前缀
 * - 杭电 OJ 1251 统计难题
 *
 * 题目描述：
 * Ada the Ladybug 有很多事情要做，几乎没有时间。她想在搜索某些东西时节省时间。
 * 给定一个单词列表和一些查询，对于每个查询，输出列表中有多少个单词以该查询字符串为前缀。
 *
 * 解题思路：
 * 1. 使用 Trie 树存储所有单词
 * 2. 每个节点记录经过该节点的单词数量
 * 3. 查询时找到前缀对应的节点，返回该节点的计数
 *
 * 时间复杂度分析：
 * 1. 构建 Trie 树：O(∑len(s))
 * 2. 查询过程：O(P)，其中 P 是前缀长度
 * 空间复杂度分析：
 * 1. O(∑len(s))
 * 是否为最优解：是
 */
class AdaAndIndexing {
    class TrieNode {
        TrieNode[] children;
        int count; // 经过该节点的单词数量

        public TrieNode() {
            children = new TrieNode[26];
            count = 0;
        }
    }

    private TrieNode root;

    public AdaAndIndexing(String[] words) {
        root = new TrieNode();
        for (String word : words) {
            insert(word);
```

```
        }
    }

    private void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
            node.count++;
        }
    }

    public int prefixCount(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return 0;
            }
            node = node.children[index];
        }
        return node.count;
    }
}

/*
 * 题目 12: CodeForces 271D - Good Substrings
 * 题目来源：CodeForces
 * 题目链接：https://codeforces.com/problemset/problem/271/D
 * 相关题目：
 * - LeetCode 208. 实现 Trie (前缀树)
 * - HDU 1251 统计难题
 * - SPOJ DICT - Dictionary
 *
 * 题目描述：
 * 给定一个字符串 s，一个由 26 个字符组成的字符串，表示每个字母是好字母还是坏字母，
 * 以及一个整数 k，表示一个好子串中最多允许的坏字符数量。
 * 找出字符串 s 中不同好子串的数量。
 *
 * 解题思路：
```

```
 * 1. 使用 Trie 树存储所有好子串
 * 2. 枚举所有可能的子串，检查是否为好子串
 * 3. 将好子串插入 Trie 树，避免重复计数
 *
 * 时间复杂度分析：
 * 1. O(N^3)，其中 N 是字符串长度
 * 空间复杂度分析：
 * 1. O(N^2)
 * 是否为最优解：可以使用更优化的方法
 */
class GoodSubstrings {
    class TrieNode {
        TrieNode[] children;
        boolean isEnd;

        public TrieNode() {
            children = new TrieNode[26];
            isEnd = false;
        }
    }

    private TrieNode root;

    public GoodSubstrings() {
        root = new TrieNode();
    }

    public int countGoodSubstrings(String s, String goodChars, int k) {
        int count = 0;
        int n = s.length();
        char[] good = goodChars.toCharArray();
        Set<String> uniqueSubstrings = new HashSet<>();

        // 枚举所有子串
        for (int i = 0; i < n; i++) {
            int badCount = 0;
            StringBuilder substring = new StringBuilder();

            for (int j = i; j < n; j++) {
                char c = s.charAt(j);
                int index = c - 'a'; // 修正索引计算

                // 检查字符是否为坏字符
```

```
                if (index >= 0 && index < 26 && good[index] == '0') {
                    badCount++;
                }

                // 如果坏字符数量超过限制，停止扩展
                if (badCount > k) {
                    break;
                }

                substring.append(c);
                String currentSubstring = substring.toString();

                // 如果该子串尚未被计数，计数并添加到集合中
                if (!uniqueSubstrings.contains(currentSubstring)) {
                    uniqueSubstrings.add(currentSubstring);
                    count++;
                }
            }
        }

        return count;
    }
}

// 测试方法
public static void main(String[] args) {
    Code06_ExtendedTrieProblems solution = new Code06_ExtendedTrieProblems();

    // 测试 WordFilter
    System.out.println("=== 测试 WordFilter ===");
    String[] words1 = {"apple", "application", "apply"};
    WordFilter wf = solution.new WordFilter(words1);
    System.out.println("f(\"a\", \"e\"): " + wf.f("a", "e")); // 应该返回 2（apply 的下标）

    // 测试 PalindromePairs
    System.out.println("\n=== 测试 PalindromePairs ===");
    String[] words2 = {"abcd", "dcba", "lls", "s", "sssll"};
    PalindromePairs pp = solution.new PalindromePairs();
    List<List<Integer>> pairs = pp.palindromePairs(words2);
    System.out.println("回文对: " + pairs);

    // 测试 ShortestPrefixes
    System.out.println("\n=== 测试 ShortestPrefixes ===");
```

```
String[] words3 = {"z", "dog", "duck", "dove"};
ShortestPrefixes sp = solution.new ShortestPrefixes();
Map<String, String> prefixes = sp.findShortestPrefixes(words3);
System.out.println("最短唯一前缀: " + prefixes);

// 测试 HatsWords
System.out.println("\n=== 测试 HatsWords ===");
String[] words4 = {"a", "hat", "hats", "word", "words", "hatword"};
HatsWords hw = solution.new HatsWords();
List<String> hatsWords = hw.findHatsWords(words4);
System.out.println("Hat's words: " + hatsWords);

// 测试 LongestCommonPrefix
System.out.println("\n=== 测试 LongestCommonPrefix ===");
String[] words5 = {"flower", "flow", "flight"};
LongestCommonPrefix lcp = solution.new LongestCommonPrefix();
String commonPrefix = lcp.longestCommonPrefix(words5);
System.out.println("最长公共前缀: " + commonPrefix);

// 测试 RollCallSystem
System.out.println("\n=== 测试 RollCallSystem ===");
String[] names = {"alice", "bob", "charlie"};
RollCallSystem rcs = solution.new RollCallSystem(names);
System.out.println("点名 alice: " + rcs.call("alice")); // OK
System.out.println("点名 alice: " + rcs.call("alice")); // REPEAT
System.out.println("点名 david: " + rcs.call("david")); // WRONG

// 测试 DictionarySearch
System.out.println("\n=== 测试 DictionarySearch ===");
String[] dictionary = {"apple", "application", "apply", "banana", "band"};
DictionarySearch ds = solution.new DictionarySearch(dictionary);
List<String> results = ds.search("app");
System.out.println("前缀'app'的单词: " + results);

// 测试 SPOJPhoneList
System.out.println("\n=== 测试 SPOJPhoneList ===");
String[] phones1 = {"911", "97625999", "91125426"};
String[] phones2 = {"113", "12340", "123440", "12345", "98346"};
SPOJPhoneList spl = solution.new SPOJPhoneList();
System.out.println("电话号码列表 1 是否一致: " + spl.hasConsistentList(phones1)); // false
System.out.println("电话号码列表 2 是否一致: " + spl.hasConsistentList(phones2)); // true

// 测试 MinNumber
```

```
        System.out.println("\n=== 测试 MinNumber ===");
        int[] nums = {3, 30, 34, 5, 9};
        MinNumber mn = solution.new MinNumber();
        String minNum = mn.minNumber(nums);
        System.out.println("最小数字: " + minNum); // "3033459"

        // 测试 StatisticalProblem
        System.out.println("\n=== 测试 StatisticalProblem ===");
        String[] words6 = {"banana", "band", "bee", "absolute", "acm"};
        StatisticalProblem stat = solution.new StatisticalProblem(words6);
        System.out.println("前缀'ba'的数量: " + stat.prefixCount("ba")); // 2
        System.out.println("前缀'b'的数量: " + stat.prefixCount("b")); // 3
        System.out.println("前缀'abc'的数量: " + stat.prefixCount("abc")); // 0

        // 测试 AdaAndIndexing
        System.out.println("\n=== 测试 AdaAndIndexing ===");
        String[] words7 = {"abc", "abcde", "abcdef", "bcd", "bcde"};
        AdaAndIndexing ada = solution.new AdaAndIndexing(words7);
        System.out.println("前缀'abc'的数量: " + ada.prefixCount("abc")); // 3
        System.out.println("前缀'bc'的数量: " + ada.prefixCount("bc")); // 2
        System.out.println("前缀'xyz'的数量: " + ada.prefixCount("xyz")); // 0

        // 测试 GoodSubstrings
        System.out.println("\n=== 测试 GoodSubstrings ===");
        GoodSubstrings gs = solution.new GoodSubstrings();
        String s = "ababac";
        String goodChars = "10000000000000000000000000"; // 只有'a'是好字符
        int k = 1; // 最多允许 1 个坏字符
        int result = gs.countGoodSubstrings(s, goodChars, k);
        System.out.println("好子串数量: " + result);
    }
}

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

文件: Code06_ExtendedTrieProblems.py
==========================================
# -*- coding: utf-8 -*-

"""
Trie 树扩展题目合集 - Python 版本
从各大算法平台收集的 Trie 树相关题目
"""
```

```
class ExtendedTrieProblems:
    """
    题目 1: LeetCode 745. 前缀和后缀搜索
    题目来源：LeetCode
    题目链接：https://leetcode.cn/problems/prefix-and-suffix-search/

    解题思路：
    1. 使用 Trie 树存储所有单词，每个节点记录经过该节点的最大下标
    2. 对于每个单词，将其所有后缀+分隔符+单词本身插入 Trie 树
    3. 查询时，将后缀+分隔符+前缀作为查询字符串

    时间复杂度分析：
    1. 构造函数：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
    2. f 函数：O(P+S)，其中 P 是前缀长度，S 是后缀长度
    空间复杂度分析：
    1. O(N*L^2)，需要存储所有单词的所有后缀组合
    是否为最优解：是，这是解决此类问题的经典方法
    """
    class WordFilter:
        class TrieNode:
            def __init__(self):
                self.children = {}
                self.weight = 0  # 存储经过该节点的最大下标

        def __init__(self, words):
            self.root = self.TrieNode()
            for weight, word in enumerate(words):
                # 对于每个单词，插入所有后缀+分隔符+单词的组合
                for i in range(len(word) + 1):
                    key = word[i:] + "{" + word
                    node = self.root
                    for c in key:
                        if c not in node.children:
                            node.children[c] = self.TrieNode()
                        node = node.children[c]
                        node.weight = weight  # 更新最大下标

        def f(self, prefix, suffix):
            key = suffix + "{" + prefix
            node = self.root
            for c in key:
                if c not in node.children:
```

```
                return -1
            node = node.children[c]
        return node.weight

"""
题目 2: LeetCode 336. 回文对
题目来源：LeetCode
题目链接：https://leetcode.cn/problems/palindrome-pairs/

解题思路：
1. 使用 Trie 树存储所有单词的逆序
2. 对于每个单词，在 Trie 树中查找能与之形成回文串的单词
3. 分情况讨论：当前单词是较长部分、当前单词是较短部分

时间复杂度分析：
1. 构建 Trie 树：O(N*L)，其中 N 是单词数量，L 是单词平均长度
2. 查询过程：O(N*L^2)，需要检查每个单词的所有前缀和后缀
空间复杂度分析：
1. O(N*L)，Trie 树存储空间
是否为最优解：是，Trie 树是解决此类问题的高效方法
"""
class PalindromePairs:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.index = -1  # 单词在数组中的下标
            self.list = []   # 存储经过该节点且剩余部分是回文的单词下标

    def is_palindrome(self, word, i, j):
        """检查子串 word[i:j+1]是否是回文"""
        while i < j:
            if word[i] != word[j]:
                return False
            i += 1
            j -= 1
        return True

    def add_word(self, root, word, index):
        """逆序插入单词到 Trie 树"""
        node = root
        # 逆序插入单词
        for i in range(len(word)-1, -1, -1):
            c = word[i]
```

```
                if c not in node.children:
                    node.children[c] = self.TrieNode()
                node = node.children[c]
                # 如果单词的前缀是回文，记录当前下标
                if self.is_palindrome(word, 0, i):
                    node.list.append(index)
            node.list.append(index)
            node.index = index

        def search(self, words, i, node, result):
            """在 Trie 树中搜索能与 words[i]形成回文对的单词"""
            # 正序匹配单词
            for j in range(len(words[i])):
                if node.index >= 0 and node.index != i and self.is_palindrome(words[i], j,
len(words[i])-1):
                    result.append([i, node.index])

                c = words[i][j]
                if c not in node.children:
                    return
                node = node.children[c]

            # 处理 Trie 树中剩余的匹配
            for j in node.list:
                if i == j:
                    continue
                result.append([i, j])

        def palindrome_pairs(self, words):
            result = []
            root = self.TrieNode()

            # 构建 Trie 树，存储单词的逆序
            for i, word in enumerate(words):
                self.add_word(root, word, i)

            # 对于每个单词，在 Trie 树中查找匹配
            for i in range(len(words)):
                self.search(words, i, root, result)

            return result

    """
```

题目 3: POJ 2001 Shortest Prefixes
题目来源：POJ
题目链接：http://poj.org/problem?id=2001

解题思路：
1. 使用 Trie 树存储所有单词
2. 记录每个节点被经过的次数
3. 对于每个单词，找到第一个出现次数为 1 的节点，该节点之前的前缀就是最短唯一前缀

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(∑len(s))
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是，Trie 树是解决此类问题的最优方法
"""

```
class ShortestPrefixes:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.count = 0  # 经过该节点的单词数量

    def find_shortest_prefixes(self, words):
        result = {}
        root = self.TrieNode()

        # 构建 Trie 树
        for word in words:
            node = root
            for c in word:
                if c not in node.children:
                    node.children[c] = self.TrieNode()
                node = node.children[c]
                node.count += 1

        # 为每个单词寻找最短唯一前缀
        for word in words:
            node = root
            prefix = ""
            for c in word:
                prefix += c
                node = node.children[c]
                # 如果当前节点只被当前单词经过，则找到最短唯一前缀
```

```
                if node.count == 1:
                    break
            result[word] = prefix

        return result

"""
题目 4: HDU 1247 Hat's Words
题目来源：HDU
题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1247

解题思路：
1. 使用 Trie 树存储所有单词
2. 对于每个单词，检查它是否能被拆分成两个都在字典中的单词
3. 使用 Trie 树快速检查每个前缀和后缀是否在字典中

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 检查过程：O(N*L^2)，其中 N 是单词数量，L 是单词最大长度
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是，Trie 树提供高效的字符串查找
"""
class HatsWords:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end = False

    def insert(self, root, word):
        """插入单词到 Trie 树"""
        node = root
        for c in word:
            if c not in node.children:
                node.children[c] = self.TrieNode()
            node = node.children[c]
        node.is_end = True

    def search(self, root, word):
        """在 Trie 树中搜索单词"""
        node = root
        for c in word:
            if c not in node.children:
```

```
                return False
            node = node.children[c]
        return node.is_end

    def is_hats_word(self, root, word):
        """检查单词是否是 hat's word"""
        # 尝试所有可能的分割点
        for i in range(1, len(word)):
            prefix = word[:i]
            suffix = word[i:]
            if self.search(root, prefix) and self.search(root, suffix):
                return True
        return False

    def find_hats_words(self, words):
        result = []
        root = self.TrieNode()

        # 构建 Trie 树
        for word in words:
            self.insert(root, word)

        # 检查每个单词是否是 hat's word
        for word in words:
            if self.is_hats_word(root, word):
                result.append(word)

        return result

"""
```

题目 5: 牛客网 最长公共前缀
题目来源：牛客网
题目链接：https://www.nowcoder.com/practice/28eb3175488f4434a4a6207f6f484f47

解题思路：
1. 使用 Trie 树存储所有字符串
2. 从根节点开始，找到第一个有多个子节点的节点
3. 该节点之前的前缀就是最长公共前缀

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查找过程：O(min(len(s)))
空间复杂度分析：

1. O(∑len(s))
是否为最优解：是，Trie 树提供直观的解决方案
"""

```
class LongestCommonPrefix:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.count = 0  # 经过该节点的字符串数量

    def longest_common_prefix(self, strs):
        if not strs:
            return ""
        if len(strs) == 1:
            return strs[0]

        # 直接比较法，避免 Trie 树的内存问题
        prefix = ""
        min_len = min(len(s) for s in strs)

        for i in range(min_len):
            char = strs[0][i]
            for j in range(1, len(strs)):
                if strs[j][i] != char:
                    return prefix
            prefix += char

        return prefix

"""
```

题目 6: 洛谷 P2580 点名系统
题目来源：洛谷
题目链接：https://www.luogu.com.cn/problem/P2580

解题思路：
1. 使用 Trie 树存储所有学生姓名
2. 每个节点记录点名状态
3. 根据点名状态输出相应结果

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(∑len(s))
空间复杂度分析：
1. O(∑len(s))

是否为最优解：是，Trie 树提供高效的姓名查找
"""
```
class RollCallSystem:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.status = 0  # 0: 未点名, 1: 已点名

    def __init__(self, names):
        self.root = self.TrieNode()
        # 构建 Trie 树，插入所有学生姓名
        for name in names:
            self.insert(name)

    def insert(self, name):
        """插入学生姓名"""
        node = self.root
        for c in name:
            if c not in node.children:
                node.children[c] = self.TrieNode()
            node = node.children[c]
        node.status = 0  # 初始状态为未点名

    def call(self, name):
        """点名"""
        node = self.root
        for c in name:
            if c not in node.children:
                return "WRONG"  # 姓名不存在
            node = node.children[c]

        if node.status == 0:
            node.status = 1  # 标记为已点名
            return "OK"
        elif node.status == 1:
            return "REPEAT"
        else:
            return "WRONG"

"""
```
题目 7: CodeChef DICT - Dictionary
题目来源：CodeChef
题目链接：https://www.codechef.com/problems/DICT

解题思路：
1. 使用 Trie 树存储字典中的所有单词
2. 每个节点维护以该节点为前缀的所有单词
3. 查询时找到前缀对应的节点，输出该节点存储的所有单词

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(P + K)，其中 P 是前缀长度，K 是输出单词数量
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是，Trie 树是解决前缀查询的高效方法
"""

```
class DictionarySearch:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.words = []  # 存储以该节点为前缀的所有单词

    def __init__(self, dictionary):
        self.root = self.TrieNode()
        # 构建 Trie 树
        for word in dictionary:
            self.insert(word)

    def insert(self, word):
        """插入单词"""
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = self.TrieNode()
            node = node.children[c]
            node.words.append(word)

    def search(self, prefix):
        """搜索前缀"""
        node = self.root
        for c in prefix:
            if c not in node.children:
                return []  # 前缀不存在
            node = node.children[c]
        # 返回该前缀对应的所有单词，按字典序排序
        return sorted(node.words)
```

"""
题目 8: SPOJ PHONELST - Phone List
题目来源：SPOJ
题目链接：https://www.spoj.com/problems/PHONELIST/

解题思路：
1. 使用 Trie 树存储所有电话号码
2. 在插入过程中检查前缀关系
3. 优化实现，提高效率

时间复杂度分析：
1. O(∑len(s))
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是
"""

```
class SPOJPhoneList:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end = False

    def has_consistent_list(self, phone_numbers):
        # 按长度排序，先插入短的
        phone_numbers.sort(key=len)
        root = self.TrieNode()

        for phone in phone_numbers:
            node = root
            created_new = False

            for i in range(len(phone)):
                digit = phone[i]
                if digit not in node.children:
                    node.children[digit] = self.TrieNode()
                    created_new = True

                node = node.children[digit]

                # 如果在插入过程中遇到已标记的结尾，说明存在前缀关系
                if node.is_end:
                    return False
```

```
            # 如果当前节点有子节点，说明当前号码是其他号码的前缀
            if not created_new:
                return False

            node.is_end = True

        return True

"""
题目 9: 剑指 Offer 45. 把数组排成最小的数
题目来源：剑指 Offer
题目链接：https://leetcode.cn/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/

解题思路：
1. 使用自定义比较器进行字符串排序
2. 比较 a+b 和 b+a 的大小
3. 按特定顺序拼接字符串

时间复杂度分析：
1. O(NlogN)
空间复杂度分析：
1. O(N)
是否为最优解：是
"""
class MinNumber:
    def min_number(self, nums):
        # 将数字转换为字符串
        str_nums = [str(num) for num in nums]

        # 自定义排序：比较 a+b 和 b+a 的大小
        from functools import cmp_to_key
        def compare(x, y):
            if x + y < y + x:
                return -1
            elif x + y > y + x:
                return 1
            else:
                return 0

        str_nums.sort(key=cmp_to_key(compare))

        # 拼接结果
```

```
        return ''.join(str_nums)

"""
题目 10: 杭电 OJ 1251 统计难题
题目来源：杭电 OJ
题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=1251

解题思路：
1. 使用 Trie 树存储所有单词
2. 每个节点记录经过该节点的单词数量
3. 查询时找到前缀对应的节点，返回该节点的计数

时间复杂度分析：
1. 构建 Trie 树：O(∑len(s))
2. 查询过程：O(P)，其中 P 是前缀长度
空间复杂度分析：
1. O(∑len(s))
是否为最优解：是
"""
class StatisticalProblem:
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.count = 0  # 经过该节点的单词数量

    def __init__(self, words):
        self.root = self.TrieNode()
        for word in words:
            self.insert(word)

    def insert(self, word):
        """插入单词"""
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = self.TrieNode()
            node = node.children[c]
            node.count += 1

    def prefix_count(self, prefix):
        """统计前缀数量"""
        node = self.root
        for c in prefix:
```

```
                if c not in node.children:
                    return 0
                node = node.children[c]
            return node.count


def test():
    """测试所有题目"""
    solution = ExtendedTrieProblems()

    print("=== 测试 WordFilter ===")
    words1 = ["apple", "application", "apply"]
    wf = solution.WordFilter(words1)
    print(f"f('a', 'e'): {wf.f('a', 'e')}")  # 应该返回 2（apply 的下标）

    print("\n=== 测试 PalindromePairs ===")
    words2 = ["abcd", "dcba", "lls", "s", "sssll"]
    pp = solution.PalindromePairs()
    pairs = pp.palindrome_pairs(words2)
    print(f"回文对数量: {len(pairs)}")

    print("\n=== 测试 ShortestPrefixes ===")
    words3 = ["z", "dog", "duck", "dove"]
    sp = solution.ShortestPrefixes()
    prefixes = sp.find_shortest_prefixes(words3)
    print(f"最短唯一前缀: {prefixes}")

    print("\n=== 测试 HatsWords ===")
    words4 = ["a", "hat", "hats", "word", "words", "hatword"]
    hw = solution.HatsWords()
    hats_words = hw.find_hats_words(words4)
    print(f"Hat's words: {hats_words}")

    print("\n=== 测试 LongestCommonPrefix ===")
    words5 = ["flower", "flow", "flight"]
    lcp = solution.LongestCommonPrefix()
    common_prefix = lcp.longest_common_prefix(words5)
    print(f"最长公共前缀: '{common_prefix}'")

    print("\n=== 测试 RollCallSystem ===")
    names = ["alice", "bob", "charlie"]
    rcs = solution.RollCallSystem(names)
    print(f"点名 alice: {rcs.call('alice')}")  # OK
```

```
    print(f"点名 alice: {rcs.call('alice')}")  # REPEAT
    print(f"点名 david: {rcs.call('david')}")  # WRONG

    print("\n=== 测试 DictionarySearch ===")
    dictionary = ["apple", "application", "apply", "banana", "band"]
    ds = solution.DictionarySearch(dictionary)
    results = ds.search("app")
    print(f"前缀'app'的单词: {results}")

    print("\n=== 测试 SPOJPhoneList ===")
    phones1 = ["911", "97625999", "91125426"]
    phones2 = ["113", "12340", "123440", "12345", "98346"]
    spl = solution.SPOJPhoneList()
    print(f"电话号码列表 1 是否一致: {spl.has_consistent_list(phones1)}")  # False
    print(f"电话号码列表 2 是否一致: {spl.has_consistent_list(phones2)}")  # True

    print("\n=== 测试 MinNumber ===")
    nums = [3, 30, 34, 5, 9]
    mn = solution.MinNumber()
    min_num = mn.min_number(nums)
    print(f"最小数字: {min_num}")

    print("\n=== 测试 StatisticalProblem ===")
    words6 = ["banana", "band", "bee", "absolute", "acm"]
    stat = solution.StatisticalProblem(words6)
    print(f"前缀'ba'的数量: {stat.prefix_count('ba')}")  # 2
    print(f"前缀'b'的数量: {stat.prefix_count('b')}")   # 3
    print(f"前缀'abc'的数量: {stat.prefix_count('abc')}") # 0


if __name__ == "__main__":
    test()

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

文件: Code06_ExtendedTrieProblems_Simple.cpp
================================================
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <map>
using namespace std;
```

```
/**
 * Trie 树扩展题目合集 - 简化版本
 * 避免使用可能引起编译问题的现代 C++特性
 */

class ExtendedTrieProblemsSimple {
public:
    /*
     * 题目 1: LeetCode 745. 前缀和后缀搜索
     */
    class WordFilter {
    private:
        struct TrieNode {
            TrieNode* children[27]; // 26 个字母 + 1 个分隔符
            int weight;

            TrieNode() : weight(0) {
                for (int i = 0; i < 27; i++) children[i] = nullptr;
            }
        };

        TrieNode* root;

    public:
        WordFilter(vector<string>& words) {
            root = new TrieNode();
            for (int weight = 0; weight < words.size(); weight++) {
                string word = words[weight];
                for (int i = 0; i <= word.length(); i++) {
                    string key = word.substr(i) + "{" + word;
                    TrieNode* node = root;
                    for (char c : key) {
                        int index = c - 'a';
                        if (c == '{') index = 26;
                        if (!node->children[index]) {
                            node->children[index] = new TrieNode();
                        }
                        node = node->children[index];
                        node->weight = weight;
                    }
                }
            }
```

```
    }

    int f(string prefix, string suffix) {
        string key = suffix + "{" + prefix;
        TrieNode* node = root;
        for (char c : key) {
            int index = c - 'a';
            if (c == '{') index = 26;
            if (!node->children[index]) return -1;
            node = node->children[index];
        }
        return node->weight;
    }
};

/*
 * 题目 2: 最长公共前缀
 */
class LongestCommonPrefix {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if (strs.empty()) return "";
        if (strs.size() == 1) return strs[0];

        string prefix = "";
        // 直接比较法，避免 Trie 树的内存问题
        for (int i = 0; i < strs[0].length(); i++) {
            char c = strs[0][i];
            for (int j = 1; j < strs.size(); j++) {
                if (i >= strs[j].length() || strs[j][i] != c) {
                    return prefix;
                }
            }
            prefix += c;
        }
        return prefix;
    }
};

/*
 * 题目 3: 电话号码列表检查
 */
class PhoneListChecker {
```

```
private:
    struct TrieNode {
        TrieNode* children[10];
        bool isEnd;

        TrieNode() : isEnd(false) {
            for (int i = 0; i < 10; i++) children[i] = nullptr;
        }
    };

public:
    bool hasConsistentList(vector<string>& phoneNumbers) {
        // 按长度排序
        sort(phoneNumbers.begin(), phoneNumbers.end(),
             [](const string& a, const string& b) { return a.length() < b.length(); });

        TrieNode* root = new TrieNode();

        for (const string& phone : phoneNumbers) {
            TrieNode* node = root;
            bool createdNew = false;

            for (int i = 0; i < phone.length(); i++) {
                int digit = phone[i] - '0';

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

                node = node->children[digit];

                if (node->isEnd) {
                    return false;
                }
            }

            if (!createdNew) return false;
            node->isEnd = true;
        }

        return true;
    }
```

```
};

/*
 * 题目 4: 统计前缀数量
 */
class StatisticalProblem {
private:
    struct TrieNode {
        TrieNode* children[26];
        int count;

        TrieNode() : count(0) {
            for (int i = 0; i < 26; i++) children[i] = nullptr;
        }
    };

    TrieNode* root;

    void insert(const string& word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
            node->count++;
        }
    }

public:
    StatisticalProblem(vector<string>& words) {
        root = new TrieNode();
        for (const string& word : words) {
            insert(word);
        }
    }

    int prefixCount(const string& prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index]) return 0;
```

```
            node = node->children[index];
        }
        return node->count;
    }
};

/*
 * 题目 5: 把数组排成最小的数
 */
class MinNumber {
public:
    string minNumber(vector<int>& nums) {
        vector<string> strNums;
        for (int num : nums) {
            strNums.push_back(to_string(num));
        }

        sort(strNums.begin(), strNums.end(), [](const string& a, const string& b) {
            return a + b < b + a;
        });

        string result;
        for (const string& str : strNums) {
            result += str;
        }
        return result;
    }
};

// 测试方法
static void test() {
    ExtendedTrieProblemsSimple solution;

    cout << "=== 测试最长公共前缀 ===" << endl;
    vector<string> words1 = {"flower", "flow", "flight"};
    auto lcp = LongestCommonPrefix();
    cout << "最长公共前缀: " << lcp.longestCommonPrefix(words1) << endl;

    cout << "\n=== 测试电话号码列表 ===" << endl;
    vector<string> phones1 = {"911", "97625999", "91125426"};
    vector<string> phones2 = {"113", "12340", "123440", "12345", "98346"};
    auto plc = PhoneListChecker();
    cout << "列表 1 是否一致: " << (plc.hasConsistentList(phones1) ? "是" : "否") << endl;
```

```
        cout << "列表2是否一致: " << (plc.hasConsistentList(phones2) ? "是" : "否") << endl;

        cout << "\n=== 测试统计前缀数量 ===" << endl;
        vector<string> words2 = {"banana", "band", "bee", "absolute", "acm"};
        auto stat = StatisticalProblem(words2);
        cout << "前缀'ba'的数量: " << stat.prefixCount("ba") << endl;
        cout << "前缀'b'的数量: " << stat.prefixCount("b") << endl;

        cout << "\n=== 测试最小数字 ===" << endl;
        vector<int> nums = {3, 30, 34, 5, 9};
        auto mn = MinNumber();
        cout << "最小数字: " << mn.minNumber(nums) << endl;
    }
};

int main() {
    ExtendedTrieProblemsSimple::test();
    return 0;
}
```

==========

文件: fix_file.py

==========

```
# 修复Code01_TrieTree.java文件的脚本
with open('Code01_TrieTree.java', 'r', encoding='utf-8') as f:
    lines = f.readlines()

# 删除最后一行（多余的右花括号）
lines = lines[:-1]

# 写回文件
with open('Code01_TrieTree.java', 'w', encoding='utf-8') as f:
    f.writelines(lines)

print("文件修复完成")
```

==========