

=====

文件夹: class144\_Backtracking

=====

[Markdown 文件]

=====

文件: COMPREHENSIVE\_PROBLEMS.md

=====

# 递归与回溯算法 - 综合题目集锦

## 子序列相关问题

#### 1. 字符串的所有子序列（已实现）

- \*\*题目\*\*: 生成字符串的所有子序列（去重）
- \*\*平台\*\*: NowCoder
- \*\*链接\*\*: <https://www.nowcoder.com/practice/92e6247998294f2c933906fdedbc6e6a>

#### 2. LeetCode 78. 子集

- \*\*题目\*\*: 给定一组不含重复元素的整数数组 `nums`, 返回该数组所有可能的子集（幂集）
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/subsets/>
- \*\*解法\*\*: 回溯算法, 每个元素选择或不选择

#### 3. LeetCode 90. 子集 II（新增）

- \*\*题目\*\*: 给定一个可能包含重复元素的整数数组 `nums`, 返回该数组所有可能的子集（幂集）
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/subsets-ii/>
- \*\*解法\*\*: 回溯算法, 需要去重

#### 4. LeetCode 115. 不同的子序列

- \*\*题目\*\*: 给定一个字符串 `s` 和一个字符串 `t`, 计算在 `s` 的子序列中 `t` 出现的个数
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/distinct-subsequences/>
- \*\*解法\*\*: 动态规划或递归+记忆化

## 组合相关问题

#### 1. 数组组合去重（已实现）

- \*\*题目\*\*: 给你一个整数数组 `nums` , 其中可能包含重复元素, 请你返回该数组所有可能的组合
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/subsets-ii/>

#### 2. LeetCode 77. 组合（新增）

- **题目**: 给定两个整数  $n$  和  $k$ , 返回  $1 \dots n$  中所有可能的  $k$  个数的组合
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/combinations/>
- **解法**: 回溯算法

#### #### 3. LeetCode 39. 组合总和 (已实现)

- **题目**: 给定一个无重复元素的数组  $\text{candidates}$  和一个目标数  $\text{target}$ , 找出  $\text{candidates}$  中所有可以使数字和为  $\text{target}$  的组合
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/combination-sum/>
- **解法**: 回溯算法, 元素可重复使用

#### #### 4. LeetCode 40. 组合总和 II (新增)

- **题目**: 给定一个数组  $\text{candidates}$  和一个目标数  $\text{target}$ , 找出  $\text{candidates}$  中所有可以使数字和为  $\text{target}$  的组合
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/combination-sum-ii/>
- **解法**: 回溯算法, 元素不可重复使用

#### #### 5. LeetCode 216. 组合总和 III (新增)

- **题目**: 找出所有相加之和为  $n$  的  $k$  个数的组合。组合中只允许含有  $1 - 9$  的正整数, 且每种组合中不存在重复的数字
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/combination-sum-iii/>
- **解法**: 回溯算法

## ## 排列相关问题

#### #### 1. 无重复数字全排列 (已实现)

- **题目**: 没有重复项数字的全排列
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/permutations/>

#### #### 2. 有重复数字全排列 (已实现)

- **题目**: 有重复项数组的去重全排列
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/permutations-ii/>

#### #### 3. LeetCode 46. 全排列 (已实现)

- **题目**: 给定一个没有重复数字的序列, 返回其所有可能的全排列
- **平台**: LeetCode
- **链接**: <https://leetcode.cn/problems/permutations/>
- **解法**: 回溯算法

#### #### 4. LeetCode 47. 全排列 II (新增)

- \*\*题目\*\*: 给定一个可包含重复数字的序列，返回所有不重复的全排列
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/permutations-ii/>
- \*\*解法\*\*: 回溯算法，需要去重

#### #### 5. LeetCode 60. 排列序列 (新增)

- \*\*题目\*\*: 给出集合  $[1, 2, 3, \dots, n]$ ，其所有元素共有  $n!$  种排列。按大小顺序列出所有排列情况
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/permute/>
- \*\*解法\*\*: 数学方法，康托展开

### ## 栈操作相关问题

#### #### 1. 递归逆序栈 (已实现)

- \*\*题目\*\*: 用递归函数逆序栈
- \*\*平台\*\*: 自定义
- \*\*解法\*\*: 递归

#### #### 2. 递归排序栈 (已实现)

- \*\*题目\*\*: 用递归函数排序栈
- \*\*平台\*\*: 自定义
- \*\*解法\*\*: 递归

#### #### 3. LeetCode 155. 最小栈

- \*\*题目\*\*: 设计一个支持 push, pop, top 操作，并能在常数时间内检索到最小元素的栈
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/min-stack/>
- \*\*解法\*\*: 辅助栈

#### #### 4. LeetCode 232. 用栈实现队列

- \*\*题目\*\*: 使用栈实现队列的下列操作：push、pop、peek、empty
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/implement-queue-using-stacks/>
- \*\*解法\*\*: 双栈

#### #### 5. LeetCode 225. 用队列实现栈

- \*\*题目\*\*: 使用队列实现栈的下列操作：push、pop、top、empty
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/implement-stack-using-queues/>
- \*\*解法\*\*: 单队列或双队列

## ## 汉诺塔问题

### #### 1. 汉诺塔移动（已实现）

- \*\*题目\*\*: 打印 n 层汉诺塔问题的最优移动轨迹
- \*\*平台\*\*: 自定义
- \*\*解法\*\*: 递归

### #### 2. LeetCode 面试题 08.06. 汉诺塔问题

- \*\*题目\*\*: 在经典汉诺塔问题中，有 3 根柱子及 N 个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。请编写程序，用栈将所有盘子从第一根柱子移到最后一根柱子
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/hanota-lcci/>
- \*\*解法\*\*: 递归

## ## 其他递归与回溯经典问题

### #### 1. LeetCode 17. 电话号码的字母组合（已实现）

- \*\*题目\*\*: 给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/letter-combinations-of-a-phone-number/>
- \*\*解法\*\*: 回溯算法

### #### 2. LeetCode 22. 括号生成（已实现）

- \*\*题目\*\*: 给出 n 代表生成括号的对数，生成所有可能的并且有效的括号组合
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/generate-parentheses/>
- \*\*解法\*\*: 回溯算法

### #### 3. LeetCode 37. 解数独（已实现）

- \*\*题目\*\*: 编写一个程序，通过已填充的空格来解决数独问题
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/sudoku-solver/>
- \*\*解法\*\*: 回溯算法

### #### 4. LeetCode 51. N 皇后（已实现）

- \*\*题目\*\*: n 皇后问题研究的是如何将 n 个皇后放置在  $n \times n$  的棋盘上，并且使皇后彼此之间不能相互攻击
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/n-queens/>
- \*\*解法\*\*: 回溯算法

### #### 5. LeetCode 93. 复原 IP 地址（新增）

- \*\*题目\*\*: 有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔

- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/restore-ip-addresses/>
- \*\*解法\*\*: 回溯算法

#### ### 6. LeetCode 131. 分割回文串 (已实现)

- \*\*题目\*\*: 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/palindrome-partitioning/>
- \*\*解法\*\*: 回溯算法

#### ### 7. LeetCode 140. 单词拆分 II (新增)

- \*\*题目\*\*: 给定一个字符串 s 和一个字符串字典 wordDict，在字符串 s 中增加空格来构建一个句子，使得句子中所有的单词都在词典中
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/word-break-ii/>
- \*\*解法\*\*: 回溯算法 + 记忆化搜索

#### ### 8. LeetCode 212. 单词搜索 II (已实现)

- \*\*题目\*\*: 给定一个二维网格 board 和一个字典中的单词列表 words，找出所有同时在二维网格和字典中出现的单词
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/word-search-ii/>
- \*\*解法\*\*: 回溯算法 + Trie 树

#### ### 9. LeetCode 306. 累加数 (新增)

- \*\*题目\*\*: 累加数是一个字符串，组成它的数字可以形成累加序列。一个有效的累加序列必须至少包含 3 个数
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/additive-number/>
- \*\*解法\*\*: 回溯算法

#### ### 10. LeetCode 401. 二进制手表 (新增)

- \*\*题目\*\*: 二进制手表顶部有 4 个 LED 代表小时 (0-11)，底部的 6 个 LED 代表分钟 (0-59)。给你一个整数 turnedOn，表示当前亮着的 LED 的数量，返回二进制手表可以表示的所有可能时间
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/binary-watch/>
- \*\*解法\*\*: 回溯算法

#### ### 11. LeetCode 473. 火柴拼正方形 (新增)

- \*\*题目\*\*: 你将得到一个整数数组 matchsticks，其中 matchsticks[i] 是第 i 个火柴棒的长度。你要用所有的火柴棍拼成一个正方形
- \*\*平台\*\*: LeetCode

- \*\*链接\*\*: <https://leetcode.cn/problems/matchsticks-to-square/>
- \*\*解法\*\*: 回溯算法

#### #### 12. LeetCode 494. 目标和 (已实现)

- \*\*题目\*\*: 给定一个非负整数数组,  $a_1, a_2, \dots, a_n$ , 和一个目标数,  $S$ 。现在你有两个符号 + 和 -。对于数组中的每个数字, 你都可以选择一个符号
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/target-sum/>
- \*\*解法\*\*: 回溯算法或动态规划

#### #### 13. LeetCode 526. 优美的排列 (新增)

- \*\*题目\*\*: 假设有从 1 到  $n$  的  $n$  个整数。用这些整数构造一个数组  $\text{perm}$  (下标从 1 开始), 只要满足下述条件之一, 该数组就是一个优美的排列
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/beautiful-arrangement/>
- \*\*解法\*\*: 回溯算法

#### #### 14. LeetCode 698. 划分为 $k$ 个相等的子集 (新增)

- \*\*题目\*\*: 给定一个整数数组  $\text{nums}$  和一个正整数  $k$ , 找出是否有可能把数组分成  $k$  个非空子集, 其总和都相等
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/>
- \*\*解法\*\*: 回溯算法

#### #### 15. LeetCode 784. 字母大小写全排列 (新增)

- \*\*题目\*\*: 给定一个字符串  $S$ , 通过将字符串  $S$  中的每个字母转变大小写, 我们可以获得一个新的字符串。返回所有可能得到的字符串集合
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/letter-case-permutation/>
- \*\*解法\*\*: 回溯算法

#### #### 16. LeetCode 79. 单词搜索 (已实现)

- \*\*题目\*\*: 给定一个二维网格和一个单词, 找出该单词是否存在于网格中
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/word-search/>
- \*\*解法\*\*: 回溯算法

#### #### 17. LeetCode 996. 正方形数组的数目 (新增)

- \*\*题目\*\*: 给定一个非负整数数组  $A$ , 如果该数组任意两个相邻元素的和都可以表示为某个完全平方数, 那么这个数组就称为正方形数组。返回  $A$  的所有可能的排列中, 正方形数组的数目
- \*\*平台\*\*: LeetCode
- \*\*链接\*\*: <https://leetcode.cn/problems/number-of-squareful-arrays/>
- \*\*解法\*\*: 回溯算法 + 去重

### #### 18. POJ 1011 Sticks (新增)

- \*\*题目\*\*: 给定 n 根火柴棍，每根火柴棍都有一定的长度。要求将这些火柴棍拼成若干根长度相等的火柴棍，且每根新火柴棍的长度要尽可能大
- \*\*平台\*\*: POJ
- \*\*链接\*\*: <http://poj.org/problem?id=1011>
- \*\*解法\*\*: 回溯算法 + 剪枝优化

## ## 新增题目详细说明

### #### Code18\_SubsetsII - 子集 II (LeetCode 90)

- \*\*问题类型\*\*: 子集生成 + 去重
- \*\*关键技巧\*\*: 排序后跳过重复元素
- \*\*时间复杂度\*\*:  $O(n * 2^n)$
- \*\*空间复杂度\*\*:  $O(n)$

### #### Code19\_Combinations - 组合 (LeetCode 77)

- \*\*问题类型\*\*: 组合生成
- \*\*关键技巧\*\*: 控制起始位置避免重复
- \*\*时间复杂度\*\*:  $O(C(n, k) * k)$
- \*\*空间复杂度\*\*:  $O(k)$

### #### Code20\_PermutationsII - 全排列 II (LeetCode 47)

- \*\*问题类型\*\*: 排列生成 + 去重
- \*\*关键技巧\*\*: 排序后确保相同元素的相对顺序
- \*\*时间复杂度\*\*:  $O(n * n!)$
- \*\*空间复杂度\*\*:  $O(n)$

### #### Code21\_CombinationSumII - 组合总和 II (LeetCode 40)

- \*\*问题类型\*\*: 组合求和 + 去重
- \*\*关键技巧\*\*: 排序后跳过重复元素，每个数字只能使用一次
- \*\*时间复杂度\*\*:  $O(2^n)$
- \*\*空间复杂度\*\*:  $O(n)$

### #### Code22\_CombinationSumIII - 组合总和 III (LeetCode 216)

- \*\*问题类型\*\*: 组合求和
- \*\*关键技巧\*\*: 数字范围限制在 1-9，不能重复使用
- \*\*时间复杂度\*\*:  $O(C(9, k))$
- \*\*空间复杂度\*\*:  $O(k)$

### #### Code23\_PermutationSequence - 排列序列 (LeetCode 60)

- \*\*问题类型\*\*: 数学排列
- \*\*关键技巧\*\*: 康托展开，直接计算第 k 个排列

- \*\*时间复杂度\*\*:  $O(n^2)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code24\_RestoreIPAddresses - 复原 IP 地址 (LeetCode 93)

- \*\*问题类型\*\*: 字符串分割

- \*\*关键技巧\*\*: 回溯分割, 检查 IP 地址段合法性

- \*\*时间复杂度\*\*:  $O(3^n) = O(81)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code25\_WordBreakII - 单词拆分 II (LeetCode 140)

- \*\*问题类型\*\*: 字符串分割 + 字典匹配

- \*\*关键技巧\*\*: 回溯 + 记忆化搜索优化

- \*\*时间复杂度\*\*:  $O(2^n * n)$

- \*\*空间复杂度\*\*:  $O(n^2)$

#### Code26\_BeautifulArrangement - 优美的排列 (LeetCode 526)

- \*\*问题类型\*\*: 排列生成 + 条件约束

- \*\*关键技巧\*\*: 提前剪枝, 只有满足条件的数字才被选择

- \*\*时间复杂度\*\*:  $O(n!)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code27\_MatchsticksToSquare - 火柴拼正方形 (LeetCode 473)

- \*\*问题类型\*\*: 分区问题

- \*\*关键技巧\*\*: 回溯分配火柴棒到四条边, 剪枝优化

- \*\*时间复杂度\*\*:  $O(4^n)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code28\_PartitionToKEqualSumSubsets - 划分为 k 个相等的子集 (LeetCode 698)

- \*\*问题类型\*\*: 分区问题

- \*\*关键技巧\*\*: 回溯分配元素到 k 个子集, 剪枝优化

- \*\*时间复杂度\*\*:  $O(k^n)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code29\_AdditiveNumber - 累加数 (LeetCode 306)

- \*\*问题类型\*\*: 字符串验证

- \*\*关键技巧\*\*: 回溯验证累加关系, 处理大数问题

- \*\*时间复杂度\*\*:  $O(n^3)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code30\_BinaryWatch - 二进制手表 (LeetCode 401)

- \*\*问题类型\*\*: 枚举 + 回溯

- \*\*关键技巧\*\*: 使用回溯算法枚举所有可能的 LED 点亮组合

- \*\*时间复杂度\*\*:  $O(2^{10}) = O(1024)$

- \*\*空间复杂度\*\*:  $O(1)$

#### Code31\_LetterCasePermutation - 字母大小写全排列 (LeetCode 784)

- \*\*问题类型\*\*: 字符串变换

- \*\*关键技巧\*\*: 对每个字母字符尝试大小写两种情况

- \*\*时间复杂度\*\*:  $O(2^n * n)$

- \*\*空间复杂度\*\*:  $O(2^n * n)$

#### Code32\_NumSquarefulPerms - 正方形数组的数目 (LeetCode 996)

- \*\*问题类型\*\*: 排列生成 + 条件验证

- \*\*关键技巧\*\*: 生成所有排列并在过程中验证相邻元素和是否为完全平方数

- \*\*时间复杂度\*\*:  $O(n! * n)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code33\_Sticks - 火柴拼接 (POJ 1011)

- \*\*问题类型\*\*: 分区问题

- \*\*关键技巧\*\*: 从大到小尝试可能的长度，使用回溯分配火柴棍

- \*\*时间复杂度\*\*:  $O(2^n * n)$

- \*\*空间复杂度\*\*:  $O(n)$

#### Code34\_GenerateParenthesesII - 括号生成增强版 (LeetCode 22)

- \*\*问题类型\*\*: 括号生成 + 连续性计算

- \*\*关键技巧\*\*: 在生成括号的同时计算最大连续括号长度

- \*\*时间复杂度\*\*:  $O(4^n / \sqrt{n})$

- \*\*空间复杂度\*\*:  $O(4^n / \sqrt{n})$

## 算法技巧总结

#### 1. 回溯算法通用模板

``` java

void backtrack(参数) {

if (终止条件) {

存放结果;

return;

}

for (选择: 本层集合中元素) {

处理节点;

backtrack(路径, 选择列表); // 递归

回溯, 撤销处理结果

}

}

```

#### #### 2. 去重技巧

- **排序去重**: 先排序，然后跳过重复元素
- **相对顺序**: 确保相同元素的相对顺序，避免生成重复结果
- **哈希去重**: 使用 Set 存储已访问的状态

#### #### 3. 剪枝优化

- **提前终止**: 当当前路径不可能得到解时提前返回
- **排序剪枝**: 从大到小排序，便于提前发现不可能的情况
- **状态压缩**: 使用位运算减少空间使用

#### #### 4. 记忆化搜索

- **存储中间结果**: 避免重复计算相同子问题
- **状态表示**: 使用合适的数据结构表示状态

#### #### 5. 工程化考虑

- **异常处理**: 空输入、非法输入检查
- **边界条件**: 极端值、边界值处理
- **性能优化**: 选择合适的算法和数据结构
- **代码可读性**: 清晰的命名和注释

### ## 适用场景总结

1. **组合优化问题**: 需要找出所有满足条件的组合
2. **排列生成问题**: 需要生成所有可能的排列
3. **分区问题**: 需要将元素分配到多个组中
4. **字符串分割问题**: 需要将字符串分割成多个部分
5. **棋盘问题**: 需要在棋盘上放置棋子
6. **路径搜索问题**: 需要在网格或图中搜索路径

通过掌握这些经典的回溯算法题目和技巧，可以更好地应对各种算法面试和实际开发中的组合优化问题。

---

文件: README.md

---

# 递归与回溯算法详解

### ## 概述

递归和回溯是解决组合、排列、子集等问题的重要算法思想。递归通过函数自身调用来解决问题，而回溯则是在递归过程中通过“选择-探索-撤销”的方式来遍历所有可能的解空间。

## ## 核心知识点

### #### 1. 递归三要素

- **递归函数定义**: 明确函数的输入输出
- **递归终止条件**: 确定何时停止递归
- **递归关系**: 如何通过子问题求解原问题

### #### 2. 回溯算法框架

```

```
def backtrack(路径, 选择列表):
```

```
    if 满足结束条件:
```

```
        result.add(路径)
```

```
        return
```

```
    for 选择 in 选择列表:
```

```
        做选择
```

```
        backtrack(路径, 选择列表)
```

```
        撤销选择
```

```

### #### 3. 常见题型分类

1. **子序列问题**: 生成所有子序列
2. **组合问题**: 从  $n$  个数中选出  $k$  个数的所有组合
3. **排列问题**:  $n$  个数的所有排列
4. **栈操作问题**: 用递归实现栈的逆序和排序
5. **汉诺塔问题**: 经典的递归问题
6. **字符串组合问题**: 电话号码字母组合、括号生成等
7. **棋盘问题**: N 皇后、解数独等
8. **路径搜索问题**: 单词搜索、目标和等
9. **分区问题**: 火柴拼正方形、划分为  $k$  个相等的子集等
10. **字符串分割问题**: 复原 IP 地址、单词拆分等

## ## 算法复杂度分析

### #### 时间复杂度

- 子序列问题:  $O(2^n)$
- 组合问题:  $O(C(n, k))$
- 排列问题:  $O(n!)$
- 栈操作问题:  $O(n^2)$
- 汉诺塔问题:  $O(2^n)$
- 字符串组合问题:  $O(3^m * 4^n)$
- 棋盘问题:  $O(N!)$  或  $O(9^{(n*n)})$
- 路径搜索问题:  $O(m*n*4^L)$

- 分区问题:  $O(k^n)$  或  $O(4^n)$
- 字符串分割问题:  $O(2^n * n)$

#### #### 空间复杂度

- 递归调用栈:  $O(n)$
- 存储结果: 根据具体问题而定

#### ## 工程化考虑

##### #### 异常处理

- 输入为空的处理
- 边界条件检查
- 非法输入的验证

##### #### 性能优化

- 剪枝优化: 提前终止无效分支
- 记忆化搜索: 避免重复计算
- 迭代替代递归: 避免栈溢出

##### #### 代码可读性

- 函数命名清晰
- 添加详细注释
- 模块化设计

#### ## 适用场景

1. \*\*组合优化问题\*\*: 需要找出所有满足条件的组合
2. \*\*搜索问题\*\*: 在解空间中搜索满足条件的解
3. \*\*游戏 AI\*\*: 如棋类游戏的走法生成
4. \*\*编译器设计\*\*: 语法分析
5. \*\*人工智能\*\*: 决策树搜索
6. \*\*字符串处理\*\*: 模式匹配、文本分割等
7. \*\*资源分配\*\*: 任务调度、资源分区等

#### ## 面试重点

1. 理解递归的本质和执行过程
2. 掌握回溯算法的模板和应用
3. 能够分析时间和空间复杂度
4. 熟悉常见的变种题目
5. 能够进行代码优化和边界处理
6. 掌握剪枝技巧和记忆化搜索
7. 理解去重策略和状态压缩

## ## 文件说明

本目录包含以下 Java、C++ 和 Python 实现文件：

### #### 基础题目（已存在于原仓库）

1. \*\*Code01\_Subsequences\*\* - 字符串的所有子序列
2. \*\*Code02\_Combinations\*\* - 数组组合去重
3. \*\*Code03\_Permutations\*\* - 无重复数字全排列
4. \*\*Code04\_PermutationWithoutRepetition\*\* - 有重复数字全排列
5. \*\*Code05\_ReverseStackWithRecursive\*\* - 递归逆序栈
6. \*\*Code06\_SortStackWithRecursive\*\* - 递归排序栈
7. \*\*Code07\_TowerOfHanoi\*\* - 汉诺塔移动

### #### 经典题目（已实现）

8. \*\*Code08\_LetterCombinations\*\* - 电话号码的字母组合 (LeetCode 17)
9. \*\*Code09\_GenerateParentheses\*\* - 括号生成 (LeetCode 22)
10. \*\*Code10\_SudokuSolver\*\* - 解数独 (LeetCode 37)
11. \*\*Code11\_NQueens\*\* - N 皇后 (LeetCode 51)
12. \*\*Code12\_TargetSum\*\* - 目标和 (LeetCode 494)
13. \*\*Code13\_WordSearch\*\* - 单词搜索 (LeetCode 79)
14. \*\*Code14\_PalindromePartitioning\*\* - 分割回文串 (LeetCode 131)
15. \*\*Code15\_WordSearchII\*\* - 单词搜索 II (LeetCode 212)
16. \*\*Code16\_Permutations\*\* - 全排列 (LeetCode 46)
17. \*\*Code17\_CombinationSum\*\* - 组合总和 (LeetCode 39)

### #### 新增题目（本次补充）

18. \*\*Code18\_SubsetsII\*\* - 子集 II (LeetCode 90)
19. \*\*Code19\_Combinations\*\* - 组合 (LeetCode 77)
20. \*\*Code20\_PermutationsII\*\* - 全排列 II (LeetCode 47)
21. \*\*Code21\_CombinationSumII\*\* - 组合总和 II (LeetCode 40)
22. \*\*Code22\_CombinationSumIII\*\* - 组合总和 III (LeetCode 216)
23. \*\*Code23\_PermutationSequence\*\* - 排列序列 (LeetCode 60)
24. \*\*Code24\_RestoreIPAddresses\*\* - 复原 IP 地址 (LeetCode 93)
25. \*\*Code25\_WordBreakII\*\* - 单词拆分 II (LeetCode 140)
26. \*\*Code26\_BeautifulArrangement\*\* - 优美的排列 (LeetCode 526)
27. \*\*Code27\_MatchsticksToSquare\*\* - 火柴拼正方形 (LeetCode 473)
28. \*\*Code28\_PartitionToKEqualSumSubsets\*\* - 划分为 k 个相等的子集 (LeetCode 698)
29. \*\*Code29\_AdditiveNumber\*\* - 累加数 (LeetCode 306)

## ## 算法技巧总结

### #### 1. 回溯算法模板

```
``` java
void backtrack(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择: 本层集合中元素) {
        处理节点;
        backtrack(路径, 选择列表); // 递归
        回溯, 撤销处理结果
    }
}
```
```
```

#### ### 2. 去重技巧

- \*\*排序去重\*\*: 先排序, 然后跳过重复元素
- \*\*相对顺序\*\*: 确保相同元素的相对顺序, 避免生成重复结果
- \*\*哈希去重\*\*: 使用 Set 存储已访问的状态

#### ### 3. 剪枝优化

- \*\*提前终止\*\*: 当当前路径不可能得到解时提前返回
- \*\*排序剪枝\*\*: 从大到小排序, 便于提前发现不可能的情况
- \*\*状态压缩\*\*: 使用位运算减少空间使用

#### ### 4. 记忆化搜索

- \*\*存储中间结果\*\*: 避免重复计算相同子问题
- \*\*状态表示\*\*: 使用合适的数据结构表示状态

#### ### 5. 工程化考虑

- \*\*异常处理\*\*: 空输入、非法输入检查
- \*\*边界条件\*\*: 极端值、边界值处理
- \*\*性能优化\*\*: 选择合适的算法和数据结构
- \*\*代码可读性\*\*: 清晰的命名和注释

## ## 使用说明

### ### 编译和运行

#### #### Java

```
``` bash
```

```
javac class038/CodeXX_XXX.java
java -cp . class038.CodeXX_XXX
```

```
```
##### C++
```bash
g++ -std=c++11 class038/CodeXX_XXX.cpp -o class038/CodeXX_XXX.exe
class038/CodeXX_XXX.exe
```
```

```
##### Python
```bash
python class038/CodeXX_XXX.py
```
```

## ## 总结

本项目完整实现了递归与回溯算法的经典题目，涵盖了从基础到高级的各种应用场景。通过多语言实现和详细注释，帮助学习者深入理解算法本质，掌握工程化实现技巧，为算法面试和实际开发打下坚实基础。

有关完整项目结构和实现细节的总结，请参阅 [SUMMARY.md] (SUMMARY.md) 文件。

有关所有题目的详细说明和链接，请参阅 [COMPREHENSIVE\_PROBLEMS.md] (COMPREHENSIVE\_PROBLEMS.md) 文件。

有关每道题目的详细题解，请参阅 [SOLUTIONS.md] (SOLUTIONS.md) 文件。

---

文件: README\_1.md

---

## # 剪枝算法详解

### ## 1. 算法概述

剪枝是一种优化技术，通过提前终止不可能产生最优解的搜索分支来减少搜索空间，从而提高算法效率。剪枝技术广泛应用于回溯算法、博弈树搜索、分支限界等场景。

#### ### 1.1 算法特点

- 减少搜索空间，提高算法效率
- 不影响最终结果的正确性
- 需要设计合适的剪枝条件
- 剪枝效果与问题特性密切相关

#### ### 1.2 应用场景

- 回溯算法（N皇后、数独等）
- 博弈树搜索（Alpha-Beta 剪枝）

- 分支限界算法
- 组合优化问题

## ## 2. 剪枝类型

### ### 2.1 可行性剪枝

**\*\*定义\*\*:** 提前判断当前分支是否可能产生可行解

**\*\*应用场景\*\*:**

- N 皇后问题中检查皇后是否冲突
- 数独问题中检查数字是否符合规则
- 组合问题中检查是否超出目标值

**\*\*示例代码\*\* (N 皇后问题):**

```
``` java
// 检查在(row, col)位置放置皇后是否合法
private static boolean isValid(int[] queens, int row, int col) {
    // 检查之前行的皇后是否与当前位置冲突
    for (int i = 0; i < row; i++) {
        // 检查列冲突
        if (queens[i] == col) {
            return false;
        }
    }

    // 检查对角线冲突
    if (Math.abs(queens[i] - col) == Math.abs(i - row)) {
        return false;
    }
}

return true;
}
```

```

### ### 2.2 最优性剪枝

**\*\*定义\*\*:** 提前判断当前分支是否可能产生更优解

**\*\*应用场景\*\*:**

- 0-1 背包问题中计算上界
- 最短路径问题中评估潜在路径
- 组合优化问题中估算最优值

**\*\*示例代码\*\* (0-1 背包问题):**

```

```java
// 计算 0-1 背包问题的上界（用于最优性剪枝）
private static int calculateUpperBound(List<KnapsackItem> items, int capacity,
                                       int currentIndex, int currentWeight,
                                       int currentValue) {
    int remainingCapacity = capacity - currentWeight;
    int bound = currentValue;

    // 贪心法计算上界：按价值密度选择物品
    for (int i = currentIndex; i < items.size() && remainingCapacity > 0; i++) {
        KnapsackItem item = items.get(i);
        if (item.weight <= remainingCapacity) {
            // 完全装入
            bound += item.value;
            remainingCapacity -= item.weight;
        } else {
            // 部分装入
            bound += (int) ((double) item.value / item.weight * remainingCapacity);
            remainingCapacity = 0;
        }
    }

    return bound;
}
```
```

```

### ### 2.3 记忆化剪枝

**\*\*定义\*\*:** 避免重复计算相同子问题

**\*\*应用场景\*\*:**

- 斐波那契数列计算
- 动态规划问题
- 递归问题中的重叠子问题

**\*\*示例代码\*\* (记忆化斐波那契):**

```

```java
static class FibonacciWithMemoization {
    private Map<Integer, Long> memo = new HashMap<>();

    public long fibonacci(int n) {
        // 基础情况
        if (n <= 1) {
            return n;
        }
```

```

```

    }

    // 记忆化剪枝：如果已经计算过，直接返回
    if (memo.containsKey(n)) {
        return memo.get(n);
    }

    // 递归计算并存储结果
    long result = fibonacci(n - 1) + fibonacci(n - 2);
    memo.put(n, result);

    return result;
}

}
```

```

#### ### 2.4 Alpha-Beta 剪枝

**\*\*定义\*\*：**博弈树搜索中的剪枝技术

**\*\*应用场景\*\*：**

- 井字棋、五子棋等博弈问题
- 决策系统
- 对抗性问题求解

**\*\*示例代码\*\*** (Alpha-Beta 剪枝)：

```

``` java
public static int alphaBetaSearch(int[][] board, int depth, int alpha, int beta,
                                  boolean isMaximizingPlayer) {
    // 终止条件：达到最大深度或游戏结束
    if (depth == 0 || isGameOver(board)) {
        return evaluateBoard(board);
    }

    if (isMaximizingPlayer) {
        int maxEval = Integer.MIN_VALUE;
        List<int[]> moves = generateMoves(board, true);

        for (int[] move : moves) {
            // 执行移动
            makeMove(board, move, true);

            // 递归搜索
            int eval = alphaBetaSearch(board, depth - 1, alpha, beta, false);
            if (eval > maxEval) {
                maxEval = eval;
            }
        }
        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;
        List<int[]> moves = generateMoves(board, false);

        for (int[] move : moves) {
            // 执行移动
            makeMove(board, move, false);

            // 递归搜索
            int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);
            if (eval < minEval) {
                minEval = eval;
            }
        }
        return minEval;
    }
}
```

```

```

// 撤销移动
undoMove(board, move);

maxEval = Math.max(maxEval, eval);
alpha = Math.max(alpha, eval);

// Alpha-Beta 剪枝
if (beta <= alpha) {
    break; // beta 剪枝
}

}

return maxEval;
} else {
    int minEval = Integer.MAX_VALUE;
    List<int[]> moves = generateMoves(board, false);

    for (int[] move : moves) {
        // 执行移动
        makeMove(board, move, false);

        // 递归搜索
        int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);

        // 撤销移动
        undoMove(board, move);

        minEval = Math.min(minEval, eval);
        beta = Math.min(beta, eval);

        // Alpha-Beta 剪枝
        if (beta <= alpha) {
            break; // alpha 剪枝
        }
    }

    return minEval;
}
```
```

```

### ## 3. 剪枝策略与技巧

### #### 3.1 搜索顺序优化

**\*\*原则\*\*:** 优先搜索更有可能产生最优解的分支

**\*\*技巧\*\*:**

- 对于最大化问题，优先搜索评估值较高的分支
- 对于最小化问题，优先搜索评估值较低的分支
- 在组合问题中，按价值密度排序

### #### 3.2 边界条件处理

**\*\*原则\*\*:** 及时终止不可能产生更优解的分支

**\*\*技巧\*\*:**

- 设置合理的终止条件
- 预先计算上下界
- 使用启发式函数评估

### #### 3.3 数据结构优化

**\*\*原则\*\*:** 选择合适的数据结构提高剪枝效率

**\*\*技巧\*\*:**

- 使用位运算优化状态表示
- 使用哈希表缓存中间结果
- 使用优先队列优化搜索顺序

## ## 4. 经典问题与实现

### #### 4.1 N 皇后问题

**\*\*问题描述\*\*:** 在  $N \times N$  棋盘上放置  $N$  个皇后，使它们互不攻击

**\*\*剪枝策略\*\*:**

- 可行性剪枝：检查皇后是否冲突
- 约束传播：一旦某行无法放置皇后立即回溯

**\*\*时间复杂度\*\*:**  $O(N!)$

### #### 4.2 0-1 背包问题

**\*\*问题描述\*\*:** 在容量限制下选择物品使价值最大

**\*\*剪枝策略\*\*:**

- 最优性剪枝：计算上界评估潜在解
- 可行性剪枝：检查容量约束

**\*\*时间复杂度\*\*:** 取决于剪枝效果

#### #### 4.3 组合总和问题

**\*\*问题描述\*\*:** 找出数组中所有和为目标值的组合

**\*\*剪枝策略\*\*:**

- 可行性剪枝: 当前和超过目标值时剪枝
- 有序剪枝: 排序后利用单调性剪枝

**\*\*时间复杂度\*\*:**  $O(2^n)$

#### #### 4.4 单词搜索问题

**\*\*问题描述\*\*:** 在二维网格中查找单词

**\*\*剪枝策略\*\*:**

- 可行性剪枝: 检查字符匹配
- 约束传播: 标记已访问位置

**\*\*时间复杂度\*\*:**  $O(m \cdot n \cdot 4^L)$

### ## 5. 工程化考量

#### #### 5.1 性能优化

- **\*\*缓存机制\*\*:** 使用记忆化避免重复计算
- **\*\*预处理\*\*:** 提前排序或计算辅助数据
- **\*\*早期终止\*\*:** 及时发现无解情况

#### #### 5.2 内存管理

- **\*\*状态压缩\*\*:** 使用位运算减少内存占用
- **\*\*对象复用\*\*:** 避免频繁创建销毁对象
- **\*\*及时释放\*\*:** 清理不需要的中间结果

#### #### 5.3 代码质量

- **\*\*模块化设计\*\*:** 将剪枝逻辑独立封装
- **\*\*参数验证\*\*:** 检查输入参数的有效性
- **\*\*异常处理\*\*:** 处理边界情况和异常输入

### ## 6. 实现语言对比

#### #### 6.1 Java 实现特点

- 面向对象设计, 代码结构清晰
- 丰富的集合框架支持
- 自动内存管理

### ### 6.2 Python 实现特点

- 语法简洁，易于理解
- 强大的内置函数和库
- 动态类型，灵活性高

### ### 6.3 C++实现特点

- 性能优异，控制精细
- 模板支持泛型编程
- 手动内存管理，效率更高

## ## 7. 学习建议

### ### 7.1 掌握基础

1. 理解回溯算法原理
2. 熟悉各种搜索策略
3. 掌握基本数据结构

### ### 7.2 实践提升

1. 从简单问题开始练习
2. 分析经典问题的剪枝策略
3. 对比不同实现的性能差异

### ### 7.3 进阶学习

1. 研究高级剪枝技术
2. 学习博弈论相关算法
3. 探索机器学习中的剪枝应用

## ## 8. 参考资源

### ### 8.1 经典题目

- [LeetCode 37. 解数独] (LeetCode37\_SudokuSolver. java)
- [LeetCode 51. N皇后] (LeetCode51\_NQueens. java)
- [LeetCode 52. N皇后 II] (LeetCode52\_NQueensII. java)
- [LeetCode 39. 组合总和] (LeetCode39\_CombinationSum. java)
- [LeetCode 40. 组合总和 II] (LeetCode40\_CombinationSumII. java)
- [LeetCode 46. 全排列] (LeetCode46\_Permutations. java)
- [LeetCode 47. 全排列 II] (LeetCode47\_PermutationsII. java)
- [LeetCode 78. 子集] (LeetCode78\_Subsets. java)
- [LeetCode 90. 子集 II] (LeetCode90\_SubsetsII. java)
- [LeetCode 79. 单词搜索] (LeetCode79\_WordSearch. java)
- [LeetCode 131. 分割回文串] (LeetCode131\_PalindromePartitioning. java)
- [LeetCode 93. 复原 IP 地址] (LeetCode93\_RestoreIPAddresses. java)

- [LeetCode 329. 矩阵中的最长递增路径] (LeetCode329\_LongestIncreasingPath.java)

- [Alpha-Beta 剪枝] (AlphaBetaPruning.java)

### ### 8.2 进阶题目

- [LeetCode 216. 组合总和 III] (LeetCode216\_CombinationSumIII.java)

- [LeetCode 847. 访问所有节点的最短路径] (LeetCode847\_ShortestPathVisitingAllNodes.java)

### ### 8.3 算法复杂度分析

- 时间复杂度：根据具体问题和剪枝效果分析

- 空间复杂度：通常为  $O(n)$  递归栈深度

文件: SOLUTIONS.md

# 递归与回溯算法 - 详细题解

## ## 概述

本文档提供了 class038 目录中所有递归与回溯算法题目的详细解答，包括算法思路、复杂度分析、关键技巧和工程化考虑。

## ## 基础题目

#### Code01\_Subsequences - 字符串的所有子序列

**题目描述：**生成字符串的所有子序列（去重）

**算法思路：**

1. 使用回溯算法生成所有子序列
2. 每个字符有两种选择：选择或不选择
3. 使用 Set 去重，确保结果唯一

**时间复杂度：**  $O(2^n * n)$ ，其中  $n$  是字符串长度

**空间复杂度：**  $O(n)$ ，递归栈深度

**关键技巧：**

- 使用 Set 自动去重
- 每个字符的选择/不选择决策

#### Code02\_Combinations - 数组组合去重

**题目描述：**给定可能包含重复元素的数组，返回所有不重复的组合

**\*\*算法思路\*\*:**

1. 先排序使相同元素相邻
2. 回溯生成所有组合
3. 跳过重复元素避免重复组合

**\*\*时间复杂度\*\*:**  $O(2^n * n)$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*关键技巧\*\*:**

- 排序后跳过重复元素
- 控制起始位置避免重复

#### Code03\_Permutations – 无重复数字全排列

**\*\*题目描述\*\*:** 生成无重复数字的所有全排列

**\*\*算法思路\*\*:**

1. 回溯算法生成所有排列
2. 使用布尔数组标记已使用的元素
3. 每次选择一个未使用的元素

**\*\*时间复杂度\*\*:**  $O(n * n!)$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*关键技巧\*\*:**

- 使用 used 数组标记已使用元素
- 递归回溯选择路径

## 经典题目

#### Code08\_LetterCombinations – 电话号码的字母组合 (LeetCode 17)

**\*\*题目描述\*\*:** 给定数字字符串，返回所有可能的字母组合

**\*\*算法思路\*\*:**

1. 建立数字到字母的映射表
2. 回溯生成所有组合
3. 每个数字对应多个字母选择

**\*\*时间复杂度\*\*:**  $O(3^m * 4^n)$ ，其中  $m$  是 3 字母数字个数， $n$  是 4 字母数字个数

**\*\*空间复杂度\*\*:**  $O(m+n)$

## \*\*关键技巧\*\*:

- 数字到字母的映射处理
- 字符串拼接优化

#### Code09\_GenerateParentheses – 括号生成 (LeetCode 22)

\*\*题目描述\*\*: 生成所有有效的括号组合

## \*\*算法思路\*\*:

1. 回溯生成所有括号组合
2. 使用计数器确保括号有效性
3. 左括号数 $\geq$ 右括号数

\*\*时间复杂度\*\*:  $O(4^n / \sqrt{n})$

\*\*空间复杂度\*\*:  $O(n)$

## \*\*关键技巧\*\*:

- 左右括号计数控制
- 提前剪枝无效组合

#### Code10\_SudokuSolver – 解数独 (LeetCode 37)

\*\*题目描述\*\*: 解决  $9 \times 9$  数独问题

## \*\*算法思路\*\*:

1. 回溯尝试每个空格的数字
2. 检查行、列、 $3 \times 3$  宫格的数字有效性
3. 找到解后立即返回

\*\*时间复杂度\*\*:  $O(9^{(n \times n)})$ , 最坏情况

\*\*空间复杂度\*\*:  $O(n \times n)$

## \*\*关键技巧\*\*:

- 有效性检查优化
- 提前终止找到解

## 新增题目详细解答

#### Code18\_SubsetsII – 子集 II (LeetCode 90)

\*\*题目链接\*\*: <https://leetcode.cn/problems/subsets-ii/>

\*\*问题描述\*\*: 给定可能包含重复元素的数组，返回所有不重复的子集

**\*\*算法思路\*\*:**

1. 排序数组使相同元素相邻
  2. 回溯生成所有子集
  3. 跳过重复元素避免重复子集

**\*\*关键代码\*\*:**

```
```java
if (i > start && nums[i] == nums[i - 1]) {
    continue; // 跳过重复元素
}
```

```

### \*\*复杂度分析\*\*:

- 时间复杂度:  $O(n \times 2^n)$
  - 空间复杂度:  $O(n)$

\*\*工程化考慮\*\*:

- 输入验证: 空数组处理
  - 边界条件: 单个元素数组
  - 性能优化: 排序后剪枝

## ### Code19 Combinations - 组合 (LeetCode 77)

**\*\*题目链接\*\*:** <https://leetcode.cn/problems/combinations/>

**\*\*问题描述\*\*:** 从 1 到 n 中选出 k 个数的所有组合

**\*\*算法思路\*\*:**

1. 回溯生成所有组合
  2. 控制起始位置避免重复
  3. 剪枝优化：剩余数字不足时提前终止

**\*\*关键代码\*\*:**

```
```java
for (int i = start; i <= n - (k - path.size()) + 1; i++) {
    // 剪枝优化
}
```

```

### \*\*复杂度分析\*\*:

- 时间复杂度:  $O(C(n, k) \times k)$
  - 空间复杂度:  $O(k)$

## \*\*面试技巧\*\*:

- 解释剪枝优化的数学原理
- 对比组合与排列的区别

#### Code20\_PermutationsII - 全排列 II (LeetCode 47)

\*\*题目链接\*\*: <https://leetcode.cn/problems/permutations-ii/>

\*\*问题描述\*\*: 生成包含重复元素数组的所有不重复全排列

## \*\*算法思路\*\*:

1. 排序使相同元素相邻
2. 回溯生成排列
3. 跳过重复排列: 相同元素且前一个未使用时跳过

## \*\*关键代码\*\*:

```
``` java
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
    continue;
}
```
```

```

## \*\*复杂度分析\*\*:

- 时间复杂度:  $O(n \times n!)$
- 空间复杂度:  $O(n)$

## \*\*去重原理\*\*:

- 确保相同元素的相对顺序
- 避免生成重复排列

#### Code21\_CombinationSumII - 组合总和 II (LeetCode 40)

\*\*题目链接\*\*: <https://leetcode.cn/problems/combination-sum-ii/>

\*\*问题描述\*\*: 找出数组中所有和为 target 的组合, 每个数字只能使用一次

## \*\*算法思路\*\*:

1. 排序数组
2. 回溯生成组合
3. 跳过重复元素
4. 剪枝: 当前和超过 target 时终止

## \*\*复杂度分析\*\*:

- 时间复杂度:  $O(2^n)$
- 空间复杂度:  $O(n)$

## \*\*关键区别\*\*:

- 与组合总和 I 的区别: 数字不能重复使用
- 去重处理更复杂

#### Code22\_CombinationSumIII – 组合总和 III (LeetCode 216)

\*\*题目链接\*\*: <https://leetcode.cn/problems/combination-sum-iii/>

\*\*问题描述\*\*: 从 1-9 中找出 k 个数的组合, 和为 n

## \*\*算法思路\*\*:

1. 数字范围限制在 1-9
2. 回溯生成组合
3. 双重约束: 个数 k 与和 n

## \*\*复杂度分析\*\*:

- 时间复杂度:  $O(C(9, k))$
- 空间复杂度:  $O(k)$

## \*\*特殊约束\*\*:

- 数字范围固定
- 个数与和的双重限制

#### Code23\_PermutationSequence – 排列序列 (LeetCode 60)

\*\*题目链接\*\*: <https://leetcode.cn/problems/permute/>

\*\*问题描述\*\*: 直接计算第 k 个排列, 而不生成所有排列

## \*\*算法思路\*\*:

1. 数学方法: 康托展开
2. 计算阶乘数组
3. 逐位确定数字

## \*\*关键代码\*\*:

```
``` python
index = k // factorial[i]
result.append(str(numbers[index]))
numbers.pop(index)
```

```
k %= factorial[i]
...

```

**\*\*复杂度分析\*\*:**

- 时间复杂度:  $O(n^2)$
- 空间复杂度:  $O(n)$

**\*\*算法优势\*\*:**

- 避免生成所有排列
- 直接定位目标排列

### Code24\_RestoreIPAddresses – 复原 IP 地址 (LeetCode 93)

**\*\*题目链接\*\*:** <https://leetcode.cn/problems/restore-ip-addresses/>

**\*\*问题描述\*\*:** 将数字字符串分割成有效的 IP 地址

**\*\*算法思路\*\*:**

1. 回溯分割字符串
2. 检查每段有效性: 0-255, 无前导 0
3. 确保分成 4 段

**\*\*复杂度分析\*\*:**

- 时间复杂度:  $O(3^4) = O(81)$
- 空间复杂度:  $O(n)$

**\*\*有效性检查\*\*:**

- 数值范围: 0-255
- 前导 0 处理
- 段数限制: 必须 4 段

### Code25\_WordBreakII – 单词拆分 II (LeetCode 140)

**\*\*题目链接\*\*:** <https://leetcode.cn/problems/word-break-ii/>

**\*\*问题描述\*\*:** 将字符串分割成字典中的单词, 返回所有可能的分割

**\*\*算法思路\*\*:**

1. 回溯分割字符串
2. 记忆化搜索优化
3. 检查单词是否在字典中

**\*\*复杂度分析\*\*:**

- 时间复杂度:  $O(2^n \times n)$
- 空间复杂度:  $O(n^2)$

#### \*\*优化技巧\*\*:

- 记忆化搜索避免重复计算
- 提前终止无效分割

### Code26\_BeautifulArrangement – 优美的排列 (LeetCode 526)

\*\*题目链接\*\*: <https://leetcode.cn/problems/beautiful-arrangement/>

\*\*问题描述\*\*: 计算满足特定条件的排列数量

#### \*\*算法思路\*\*:

1. 回溯生成排列
2. 提前剪枝: 不满足条件时跳过
3. 条件: 数字能被位置整除或位置能被数字整除

#### \*\*复杂度分析\*\*:

- 时间复杂度:  $O(n!)$
- 空间复杂度:  $O(n)$

#### \*\*剪枝优化\*\*:

- 提前检查排列条件
- 减少无效搜索

### Code27\_MatchsticksToSquare – 火柴拼正方形 (LeetCode 473)

\*\*题目链接\*\*: <https://leetcode.cn/problems/matchsticks-to-square/>

\*\*问题描述\*\*: 用所有火柴拼成正方形

#### \*\*算法思路\*\*:

1. 计算总长度, 检查是否能被 4 整除
2. 排序火柴, 优先使用长火柴
3. 回溯分配到四条边

#### \*\*复杂度分析\*\*:

- 时间复杂度:  $O(4^n)$
- 空间复杂度:  $O(n)$

#### \*\*剪枝策略\*\*:

- 边长超过目标时终止

- 跳过相同长度的边

### Code28\_PartitionToKEqualSumSubsets - 划分为 k 个相等的子集 (LeetCode 698)

\*\*题目链接\*\*: <https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/>

\*\*问题描述\*\*: 将数组分成 k 个和相等的子集

\*\*算法思路\*\*:

1. 计算目标和 = 总和 / k
2. 排序数组，优先使用大数字
3. 回溯分配到 k 个子集

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(k^n)$
- 空间复杂度:  $O(n)$

\*\*与火柴问题的区别\*\*:

- 子集数量 k 可变
- 数字范围更大

### Code29\_AdditiveNumber - 累加数 (LeetCode 306)

\*\*题目链接\*\*: <https://leetcode.cn/problems/additive-number/>

\*\*问题描述\*\*: 验证字符串是否是累加序列

\*\*算法思路\*\*:

1. 尝试所有可能的前两个数字分割
2. 验证剩余部分是否满足累加关系
3. 处理大数字符串加法

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(n^3)$
- 空间复杂度:  $O(n)$

\*\*大数处理\*\*:

- 字符串加法实现
- 避免整数溢出

## 算法技巧总结

### 1. 回溯算法模板

```
``` python
def backtrack(路径, 选择列表):
    if 满足结束条件:
        结果. add(路径)
        return

    for 选择 in 选择列表:
        if 不满足条件: continue # 剪枝

        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

```

### ### 2. 常见优化技巧

#### \*\*剪枝优化\*\*:

- 提前终止无效分支
- 排序后优先处理大元素
- 跳过重复计算

#### \*\*记忆化搜索\*\*:

- 存储中间结果
- 避免重复计算子问题

#### \*\*数学优化\*\*:

- 利用数学性质减少搜索空间
- 康托展开直接计算排列

### ### 3. 工程化考虑

#### \*\*异常处理\*\*:

- 空输入处理
- 边界条件检查
- 非法输入验证

#### \*\*性能优化\*\*:

- 选择合适的数据结构
- 减少不必要的复制
- 利用语言特性优化

#### \*\*代码可读性\*\*:

- 清晰的变量命名

- 模块化函数设计
- 详细的注释说明

## ## 面试准备建议

### #### 1. 掌握核心算法

- 理解回溯算法的本质
- 熟练应用剪枝技巧
- 掌握复杂度分析方法

### #### 2. 问题分析能力

- 快速识别问题类型
- 设计合适的算法方案
- 分析时间空间复杂度

### #### 3. 编码实现能力

- 熟练编写回溯代码
- 处理边界条件
- 进行代码优化

### #### 4. 沟通表达能力

- 清晰解释算法思路
- 分析算法优缺点
- 讨论优化方案

通过系统学习这些题目，您将能够熟练掌握递归与回溯算法的各种应用场景，为算法面试和实际开发工作打下坚实基础。

文件: SUMMARY.md

## # 递归与回溯算法完整实现总结

### ## 项目概述

本项目对递归与回溯算法进行了全面的实现和分析，涵盖了从基础到高级的各种经典问题。所有代码都提供了 Java、C++ 和 Python 三种语言的实现，并包含详细的注释和复杂度分析。

### ## 已实现的题目

#### #### 1. 基础题目（已存在于原仓库）

1. \*\*Code01\_Subsequences\*\* - 字符串的所有子序列

2. \*\*Code02\_Combinations\*\* - 数组组合去重
3. \*\*Code03\_Permutations\*\* - 无重复数字全排列
4. \*\*Code04\_PermutationWithoutRepetition\*\* - 有重复数字全排列
5. \*\*Code05\_ReverseStackWithRecursive\*\* - 递归逆序栈
6. \*\*Code06\_SortStackWithRecursive\*\* - 递归排序栈
7. \*\*Code07\_TowerOfHanoi\*\* - 汉诺塔移动

#### ### 2. 经典题目（已实现）

8. \*\*Code08\_LetterCombinations\*\* - 电话号码的字母组合 (LeetCode 17)
9. \*\*Code09\_GenerateParentheses\*\* - 括号生成 (LeetCode 22)
10. \*\*Code10\_SudokuSolver\*\* - 解数独 (LeetCode 37)
11. \*\*Code11\_NQueens\*\* - N 皇后 (LeetCode 51)
12. \*\*Code12\_TargetSum\*\* - 目标和 (LeetCode 494)
13. \*\*Code13\_WordSearch\*\* - 单词搜索 (LeetCode 79)
14. \*\*Code14\_PalindromePartitioning\*\* - 分割回文串 (LeetCode 131)
15. \*\*Code15\_WordSearchII\*\* - 单词搜索 II (LeetCode 212)
16. \*\*Code16\_Permutations\*\* - 全排列 (LeetCode 46)
17. \*\*Code17\_CombinationSum\*\* - 组合总和 (LeetCode 39)

#### ### 3. 新增题目（本次补充）

18. \*\*Code18\_SubsetsII\*\* - 子集 II (LeetCode 90)
19. \*\*Code19\_Combinations\*\* - 组合 (LeetCode 77)
20. \*\*Code20\_PermutationsII\*\* - 全排列 II (LeetCode 47)
21. \*\*Code21\_CombinationSumII\*\* - 组合总和 II (LeetCode 40)
22. \*\*Code22\_CombinationSumIII\*\* - 组合总和 III (LeetCode 216)
23. \*\*Code23\_PermutationSequence\*\* - 排列序列 (LeetCode 60)
24. \*\*Code24\_RestoreIPAddresses\*\* - 复原 IP 地址 (LeetCode 93)
25. \*\*Code25\_WordBreakII\*\* - 单词拆分 II (LeetCode 140)
26. \*\*Code26\_BeautifulArrangement\*\* - 优美的排列 (LeetCode 526)
27. \*\*Code27\_MatchsticksToSquare\*\* - 火柴拼正方形 (LeetCode 473)
28. \*\*Code28\_PartitionToKEqualSumSubsets\*\* - 划分为 k 个相等的子集 (LeetCode 698)
29. \*\*Code29\_AdditiveNumber\*\* - 累加数 (LeetCode 306)

## ## 文件结构

```
```
class038/
├── README.md                      # 递归与回溯算法详解
├── COMPREHENSIVE_PROBLEMS.md      # 综合题目集锦
├── SOLUTIONS.md                   # 详细题解
├── SUMMARY.md                     # 项目总结
├── Code01_Subsequences.java/cpp/py # 字符串的所有子序列
└── Code02_Combinations.java/cpp/py # 数组组合去重
```

```
└── Code03_Permutations.java/cpp/py    # 无重复数字全排列
└── Code04_PermutationWithoutRepetition.java/cpp/py  # 有重复数字全排列
└── Code05_ReverseStackWithRecursive.java/cpp/py      # 递归逆序栈
└── Code06_SortStackWithRecursive.java/cpp/py        # 递归排序栈
└── Code07_TowerOfHanoi.java/cpp/py                 # 汉诺塔移动
└── Code08_LetterCombinations.java/cpp/py          # 电话号码的字母组合
└── Code09_GenerateParentheses.java/cpp/py         # 括号生成
└── Code10_SudokuSolver.java/cpp/py                # 解数独
└── Code11_NQueens.java/cpp/py                    # N皇后
└── Code12_TargetSum.java/cpp/py                  # 目标和
└── Code13_WordSearch.java/cpp/py                 # 单词搜索
└── Code14_PalindromePartitioning.java/cpp/py    # 分割回文串
└── Code15_WordSearchII.java/cpp/py               # 单词搜索 II
└── Code16_Permutations.java/cpp/py              # 全排列
└── Code17_CombinationSum.java/cpp/py            # 组合总和
└── Code18_SubsetsII.java/cpp/py                 # 子集 II
└── Code19_Combinations.java/cpp/py             # 组合
└── Code20_PermutationsII.java/cpp/py           # 全排列 II
└── Code21_CombinationSumII.java/cpp/py         # 组合总和 II
└── Code22_CombinationSumIII.java/cpp/py        # 组合总和 III
└── Code23_PermutationSequence.java/cpp/py       # 排列序列
└── Code24_RestoreIPAddresses.java/cpp/py        # 复原 IP 地址
└── Code25_WordBreakII.java/cpp/py              # 单词拆分 II
└── Code26_BeautifulArrangement.java/cpp/py    # 优美的排列
└── Code27_MatchsticksToSquare.java/cpp/py       # 火柴拼正方形
└── Code28_PartitionToKEqualSumSubsets.java/cpp/py # 划分为 k 个相等的子集
└── Code29_AdditiveNumber.java/cpp/py           # 累加数
...
```

```

## ## 技术特点

### #### 1. 多语言实现

- 所有题目均提供 Java、C++、Python 三种语言实现
- 保持各语言版本算法逻辑一致性
- 遵循各语言最佳实践和编码规范

### #### 2. 详细注释

- 每个文件都包含完整的题目描述
- 算法思路详细解释
- 时间复杂度和空间复杂度分析
- 关键步骤注释说明

### #### 3. 完整测试

- 每个文件都包含测试用例
- 覆盖边界条件和特殊情况
- 提供清晰的输入输出示例

#### #### 4. 工程化考虑

- 异常处理：空输入、非法输入检查
- 性能优化：剪枝、提前终止等优化策略
- 代码可读性：清晰的变量命名和模块化设计

### ## 算法分类

#### #### 1. 子序列问题

- 字符串的所有子序列
- LeetCode 78. 子集
- LeetCode 90. 子集 II

#### #### 2. 组合问题

- 数组组合去重
- LeetCode 77. 组合
- LeetCode 39. 组合总和
- LeetCode 40. 组合总和 II
- LeetCode 216. 组合总和 III

#### #### 3. 排列问题

- 无重复数字全排列
- 有重复数字全排列
- LeetCode 46. 全排列
- LeetCode 47. 全排列 II
- LeetCode 60. 排列序列

#### #### 4. 栈操作问题

- 递归逆序栈
- 递归排序栈

#### #### 5. 汉诺塔问题

- 汉诺塔移动

#### #### 6. 字符串组合问题

- 电话号码的字母组合
- 括号生成

#### #### 7. 棋盘问题

- 解数独

- N 皇后

#### #### 8. 路径搜索问题

- 目标和
- 单词搜索
- 单词搜索 II

#### #### 9. 字符串分割问题

- 分割回文串
- 复原 IP 地址
- 单词拆分 II
- 累加数

#### #### 10. 分区问题

- 火柴拼正方形
- 划分为 k 个相等的子集

#### #### 11. 约束满足问题

- 优美的排列

### ## 复杂度分析

#### #### 时间复杂度

- 子序列问题:  $O(2^n * n)$
- 组合问题:  $O(C(n, k) * k)$
- 排列问题:  $O(n! * n)$
- 栈操作问题:  $O(n^2)$
- 汉诺塔问题:  $O(2^n)$
- 电话号码字母组合:  $O(3^m * 4^n)$
- 括号生成:  $O(4^n / \sqrt{n})$
- 解数独:  $O(9^{(n*n)})$
- N 皇后:  $O(N!)$
- 目标和:  $O(2^n)$  回溯 /  $O(n * sum)$  动态规划
- 单词搜索:  $O(m*n*4^L)$
- 分割回文串:  $O(N * 2^N)$
- 单词搜索 II:  $O(m*n*4^L)$
- 全排列:  $O(N * N!)$
- 组合总和:  $O(N^(T/M))$
- 子集 II:  $O(n * 2^n)$
- 组合:  $O(C(n, k) * k)$
- 全排列 II:  $O(n * n!)$
- 组合总和 II:  $O(2^n)$
- 组合总和 III:  $O(C(9, k))$

- 排列序列:  $O(n^2)$
- 复原 IP 地址:  $O(3^4) = O(81)$
- 单词拆分 II:  $O(2^n * n)$
- 优美的排列:  $O(n!)$
- 火柴拼正方形:  $O(4^n)$
- 划分为 k 个相等的子集:  $O(k^n)$
- 累加数:  $O(n^3)$

#### #### 空间复杂度

- 递归调用栈:  $O(n)$  到  $O(n^2)$
- 存储结果: 根据具体问题而定

#### ## 面试重点

1. **理解递归的本质**: 函数调用栈、递归基条件
2. **掌握回溯算法模板**: 选择-探索-撤销选择
3. **复杂度分析能力**: 时间复杂度和空间复杂度计算
4. **优化技巧**: 剪枝、去重、记忆化搜索
5. **边界处理**: 空输入、极端值处理
6. **工程化思维**: 异常处理、代码可读性、测试覆盖
7. **多解法对比**: 能够分析不同解法的优缺点
8. **实际问题应用**: 能够将算法应用于实际问题解决

#### ## 使用说明

#### #### 编译和运行

```
##### Java
```bash
javac class038/CodeXX_XXX.java
java -cp . class038.CodeXX_XXX
```

```

```
##### C++
```bash
g++ -std=c++11 class038/CodeXX_XXX.cpp -o class038/CodeXX_XXX.exe
class038/CodeXX_XXX.exe
```

```

```
##### Python
```bash
python class038/CodeXX_XXX.py
```

```

## ## 总结

本项目完整实现了递归与回溯算法的经典题目，涵盖了从基础到高级的各种应用场景。通过多语言实现和详细注释，帮助学习者深入理解算法本质，掌握工程化实现技巧，为算法面试和实际开发打下坚实基础。

通过系统学习本项目，您将能够：

1. 熟练掌握递归与回溯算法的核心思想
2. 理解各种经典问题的解决思路
3. 掌握算法复杂度分析和优化技巧
4. 具备解决实际工程问题的能力
5. 在算法面试中表现出色

本项目不仅提供了完整的代码实现，还包含了详细的算法分析和工程化考虑，是学习递归与回溯算法的理想资源。

---

[代码文件]

---

文件：AlphaBetaPruning.java

---

/\*\*

\* Alpha-Beta 剪枝算法实现（以井字棋为例）

\*

\* 算法原理：

\* Alpha-Beta 剪枝是一种优化极小极大搜索算法的技术，通过剪掉搜索树中不会影响最终结果的分支来减少搜索节点数量。

\*

\* 算法特点：

- \* 1. 减少搜索空间，提高搜索效率
- \* 2. 不影响最终结果的正确性
- \* 3. 剪枝效果与搜索顺序密切相关
- \* 4. 最好情况下可以将时间复杂度从  $O(b^d)$  降低到  $O(b^{(d/2)})$

\*

\* 算法步骤：

- \* 1. 在极小极大搜索过程中维护 alpha 和 beta 值
- \* 2. alpha 表示最大化玩家能够保证的最小收益
- \* 3. beta 表示最小化玩家能够保证的最大损失
- \* 4. 当  $\alpha \geq \beta$  时，剪枝剩余分支

\*

\* 应用场景：

- \* - 博弈树搜索（井字棋、五子棋、象棋等）

```
* - 决策系统
* - 对抗性问题求解
*
* 时间复杂度:
* - 最坏情况:  $O(b^d)$ 
* - 最好情况:  $O(b^{(d/2)})$ 
* - 平均情况:  $O(b^{(3d/4)})$ 
*
* 空间复杂度:  $O(d)$ , 递归栈深度
*
* 工程化考量:
* 1. 移动排序: 优先搜索较优的移动可以提高剪枝效果
* 2. 迭代加深: 结合迭代加深搜索使用
* 3. 转置表: 缓存已计算的节点避免重复计算
* 4. 启发式评估: 设计合理的评估函数
* 5. 边界处理: 处理游戏结束状态
* 6. 性能优化: 通过剪枝减少不必要的计算
*/

```

```
import java.util.ArrayList;
import java.util.List;

public class AlphaBetaPruning {

    // 井字棋棋盘状态
    public static final int EMPTY = 0;
    public static final int PLAYER_X = 1;
    public static final int PLAYER_O = 2;

    // 评估分数
    public static final int WIN_SCORE = 10000;
    public static final int LOSE_SCORE = -10000;
    public static final int TIE_SCORE = 0;

    // 棋盘大小
    private static final int BOARD_SIZE = 3;

    /**
     * Alpha-Beta 剪枝搜索函数
     *
     * @param board 当前棋盘状态
     * @param depth 当前搜索深度
     * @param alpha Alpha 值 (最大化玩家的最小保证收益)
     */
}
```

```

* @param beta Beta 值（最小化玩家的最大保证损失）
* @param isMaximizingPlayer 是否为最大化玩家（PLAYER_X）
* @return 最佳评估分数
*/
public static int alphaBetaSearch(int[][] board, int depth, int alpha, int beta, boolean
isMaximizingPlayer) {
    // 检查游戏是否结束或达到最大深度
    int gameResult = evaluateGame(board);
    if (gameResult != -2 || depth == 0) { // -2 表示游戏继续
        return gameResult;
    }

    if (isMaximizingPlayer) {
        int maxEval = Integer.MIN_VALUE;
        List<int[]> moves = generateMoves(board);

        // 启发式排序：优先考虑中心和角落位置
        moves.sort((a, b) -> {
            // 中心位置优先
            if (a[0] == 1 && a[1] == 1) return -1;
            if (b[0] == 1 && b[1] == 1) return 1;
            // 角落位置次之
            if ((a[0] == 0 || a[0] == 2) && (a[1] == 0 || a[1] == 2)) return -1;
            if ((b[0] == 0 || b[0] == 2) && (b[1] == 0 || b[1] == 2)) return 1;
            return 0;
        });

        for (int[] move : moves) {
            // 执行移动
            board[move[0]][move[1]] = PLAYER_X;

            // 递归搜索
            int eval = alphaBetaSearch(board, depth - 1, alpha, beta, false);

            // 撤销移动
            board[move[0]][move[1]] = EMPTY;

            maxEval = Math.max(maxEval, eval);
            alpha = Math.max(alpha, eval);

            // Alpha-Beta 剪枝
            if (beta <= alpha) {
                break; // beta 剪枝
            }
        }
    }
}
```

```

        }

    }

    return maxEval;
} else {
    int minEval = Integer.MAX_VALUE;
    List<int[]> moves = generateMoves(board);

    // 启发式排序: 优先考虑中心和角落位置
    moves.sort((a, b) -> {
        // 中心位置优先
        if (a[0] == 1 && a[1] == 1) return -1;
        if (b[0] == 1 && b[1] == 1) return 1;
        // 角落位置次之
        if ((a[0] == 0 || a[0] == 2) && (a[1] == 0 || a[1] == 2)) return -1;
        if ((b[0] == 0 || b[0] == 2) && (b[1] == 0 || b[1] == 2)) return 1;
        return 0;
    });

    for (int[] move : moves) {
        // 执行移动
        board[move[0]][move[1]] = PLAYER_0;

        // 递归搜索
        int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);

        // 撤销移动
        board[move[0]][move[1]] = EMPTY;

        minEval = Math.min(minEval, eval);
        beta = Math.min(beta, eval);

        // Alpha-Beta 剪枝
        if (beta <= alpha) {
            break; // alpha 剪枝
        }
    }

    return minEval;
}
}

/**
```

```

* 获取最佳移动
*
* @param board 当前棋盘状态
* @param depth 搜索深度
* @param isMaximizingPlayer 是否为最大化玩家
* @return 最佳移动位置 [row, col]
*/
public static int[] getBestMove(int[][] board, int depth, boolean isMaximizingPlayer) {
    int bestValue = isMaximizingPlayer ? Integer.MIN_VALUE : Integer.MAX_VALUE;
    int[] bestMove = {-1, -1};
    List<int[]> moves = generateMoves(board);

    // 启发式排序：优先考虑中心和角落位置
    moves.sort((a, b) -> {
        // 中心位置优先
        if (a[0] == 1 && a[1] == 1) return -1;
        if (b[0] == 1 && b[1] == 1) return 1;
        // 角落位置次之
        if ((a[0] == 0 || a[0] == 2) && (a[1] == 0 || a[1] == 2)) return -1;
        if ((b[0] == 0 || b[0] == 2) && (b[1] == 0 || b[1] == 2)) return 1;
        return 0;
    });

    for (int[] move : moves) {
        // 执行移动
        board[move[0]][move[1]] = isMaximizingPlayer ? PLAYER_X : PLAYER_O;

        // 评估移动
        int moveValue = alphaBetaSearch(board, depth - 1,
   isMaximizingPlayer ? Integer.MIN_VALUE : bestValue,
   isMaximizingPlayer ? bestValue : Integer.MAX_VALUE,
   !isMaximizingPlayer);

        // 撤销移动
        board[move[0]][move[1]] = EMPTY;

        // 更新最佳移动
        if (isMaximizingPlayer && moveValue > bestValue) {
            bestValue = moveValue;
            bestMove[0] = move[0];
            bestMove[1] = move[1];
        } else if (!isMaximizingPlayer && moveValue < bestValue) {
            bestValue = moveValue;
        }
    }
}

```

```

        bestMove[0] = move[0];
        bestMove[1] = move[1];
    }

}

return bestMove;
}

/***
 * 生成所有可能的移动
 *
 * @param board 当前棋盘状态
 * @return 所有可能的移动列表
 */
private static List<int[]> generateMoves(int[][] board) {
    List<int[]> moves = new ArrayList<>();

    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            if (board[i][j] == EMPTY) {
                moves.add(new int[] {i, j});
            }
        }
    }

    return moves;
}

/***
 * 评估游戏状态
 *
 * @param board 当前棋盘状态
 * @return 游戏结果: WIN_SCORE (X 胜)、LOSE_SCORE (O 胜)、TIE_SCORE (平局)、-2 (游戏继续)
 */
private static int evaluateGame(int[][] board) {
    // 检查行
    for (int i = 0; i < BOARD_SIZE; i++) {
        if (board[i][0] != EMPTY && board[i][0] == board[i][1] && board[i][1] == board[i][2]) {
            return board[i][0] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
        }
    }
}

```

```

// 检查列
for (int j = 0; j < BOARD_SIZE; j++) {
    if (board[0][j] != EMPTY && board[0][j] == board[1][j] && board[1][j] == board[2][j])
    {
        return board[0][j] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
    }
}

// 检查对角线
if (board[0][0] != EMPTY && board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
    return board[0][0] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
}

if (board[0][2] != EMPTY && board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
    return board[0][2] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
}

// 检查是否平局
boolean isFull = true;
for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
        if (board[i][j] == EMPTY) {
            isFull = false;
            break;
        }
    }
    if (!isFull) break;
}

if (isFull) {
    return TIE_SCORE; // 平局
}

return -2; // 游戏继续
}

/***
 * 打印棋盘
 *
 * @param board 棋盘状态
 */
public static void printBoard(int[][] board) {
    System.out.println("Current Board:");

```

```

for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
        switch (board[i][j]) {
            case EMPTY:
                System.out.print(".");
                break;
            case PLAYER_X:
                System.out.print("X ");
                break;
            case PLAYER_O:
                System.out.print("O ");
                break;
        }
    }
    System.out.println();
}
System.out.println();

}

/***
 * 检查移动是否合法
 *
 * @param board 棋盘状态
 * @param row 行索引
 * @param col 列索引
 * @return 是否合法
 */
public static boolean isValidMove(int[][] board, int row, int col) {
    return row >= 0 && row < BOARD_SIZE && col >= 0 && col < BOARD_SIZE && board[row][col] == EMPTY;
}

// 测试方法
public static void main(String[] args) {
    System.out.println("== Alpha-Beta 剪枝算法测试（井字棋） ==");

    // 初始化空棋盘
    int[][] board = new int[BOARD_SIZE][BOARD_SIZE];

    // 简单测试：评估空棋盘
    int eval = alphaBetaSearch(board, 9, Integer.MIN_VALUE, Integer.MAX_VALUE, true);
    System.out.println("空棋盘评估值：" + eval);
}

```

```

// 测试: X 在中心位置
board[1][1] = PLAYER_X;
printBoard(board);
eval = alphaBetaSearch(board, 8, Integer.MIN_VALUE, Integer.MAX_VALUE, false);
System.out.println("X 在中心位置的评估值: " + eval);

// 获取最佳移动
int[] bestMove = getBestMove(board, 7, false);
System.out.println("O 的最佳移动: [" + bestMove[0] + ", " + bestMove[1] + "]");

// 模拟一局游戏
System.out.println("\n==== 模拟游戏 ===");
// 重置棋盘
for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
        board[i][j] = EMPTY;
    }
}

boolean isXMove = true;
int movesCount = 0;

while (evaluateGame(board) == -2 && movesCount < 9) {
    printBoard(board);
    int[] move = getBestMove(board, 9 - movesCount, isXMove);

    if (move[0] != -1 && move[1] != -1) {
        board[move[0]][move[1]] = isXMove ? PLAYER_X : PLAYER_O;
        System.out.println((isXMove ? "X" : "O") + " 下在 [" + move[0] + ", " + move[1] +
"]");
        isXMove = !isXMove;
        movesCount++;
    } else {
        break;
    }
}

// 短暂延迟以便观察
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}

```

```

    printBoard(board);
    int result = evaluateGame(board);
    if (result == WIN_SCORE) {
        System.out.println("X 获胜！");
    } else if (result == LOSE_SCORE) {
        System.out.println("O 获胜！");
    } else {
        System.out.println("平局！");
    }
}
=====
```

文件: alpha\_beta\_pruning.cpp

```
=====
```

```

/***
 * Alpha-Beta 剪枝算法实现（以井字棋为例）
 *
 * 算法原理:
 * Alpha-Beta 剪枝是一种优化极小极大搜索算法的技术，通过剪掉搜索树中不会影响最终结果的分支来减少搜索节点数量。
 *
 * 算法特点:
 * 1. 减少搜索空间，提高搜索效率
 * 2. 不影响最终结果的正确性
 * 3. 剪枝效果与搜索顺序密切相关
 * 4. 最好情况下可以将时间复杂度从  $O(b^d)$  降低到  $O(b^{(d/2)})$ 
 *
 * 算法步骤:
 * 1. 在极小极大搜索过程中维护 alpha 和 beta 值
 * 2. alpha 表示最大化玩家能够保证的最小收益
 * 3. beta 表示最小化玩家能够保证的最大损失
 * 4. 当  $\alpha \geq \beta$  时，剪枝剩余分支
 *
 * 应用场景:
 * - 博弈树搜索（井字棋、五子棋、象棋等）
 * - 决策系统
 * - 对抗性问题求解
 *
 * 时间复杂度:
 * - 最坏情况:  $O(b^d)$ 
```

```

* - 最好情况: O(b^(d/2))
* - 平均情况: O(b^(3d/4))
*
* 空间复杂度: O(d), 递归栈深度
*
* 工程化考量:
* 1. 移动排序: 优先搜索较优的移动可以提高剪枝效果
* 2. 迭代加深: 结合迭代加深搜索使用
* 3. 转置表: 缓存已计算的节点避免重复计算
* 4. 启发式评估: 设计合理的评估函数
* 5. 边界处理: 处理游戏结束状态
* 6. 性能优化: 通过剪枝减少不必要的计算
*/

```

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <utility>
using namespace std;

class AlphaBetaPruning {
public:
    // 井字棋棋盘状态
    static const int EMPTY = 0;
    static const int PLAYER_X = 1;
    static const int PLAYER_O = 2;

    // 评估分数
    static const int WIN_SCORE = 10000;
    static const int LOSE_SCORE = -10000;
    static const int TIE_SCORE = 0;

    // 棋盘大小
    static const int BOARD_SIZE = 3;

    /**
     * Alpha-Beta 剪枝搜索函数
     *
     * @param board 当前棋盘状态
     * @param depth 当前搜索深度
     * @param alpha Alpha 值 (最大化玩家的最小保证收益)
     * @param beta Beta 值 (最小化玩家的最大保证损失)
     */

```

```

* @param isMaximizingPlayer 是否为最大化玩家 (PLAYER_X)
* @return 最佳评估分数
*/
static int alphaBetaSearch(vector<vector<int>>& board, int depth, int alpha, int beta, bool
isMaximizingPlayer) {
    // 检查游戏是否结束或达到最大深度
    int gameResult = evaluateGame(board);
    if (gameResult != -2 || depth == 0) { // -2 表示游戏继续
        return gameResult;
    }

    if (isMaximizingPlayer) {
        int maxEval = INT_MIN;
        vector<pair<int, int>> moves = generateMoves(board);

        // 启发式排序: 优先考虑中心和角落位置
        sort(moves.begin(), moves.end(), [] (const pair<int, int>& a, const pair<int, int>& b)
{
            // 中心位置优先
            if (a.first == 1 && a.second == 1) return true;
            if (b.first == 1 && b.second == 1) return false;
            // 角落位置次之
            if ((a.first == 0 || a.first == 2) && (a.second == 0 || a.second == 2)) return
true;
            if ((b.first == 0 || b.first == 2) && (b.second == 0 || b.second == 2)) return
false;
            return false;
        });

        for (const auto& move : moves) {
            // 执行移动
            board[move.first][move.second] = PLAYER_X;

            // 递归搜索
            int eval = alphaBetaSearch(board, depth - 1, alpha, beta, false);

            // 撤销移动
            board[move.first][move.second] = EMPTY;

            maxEval = max(maxEval, eval);
            alpha = max(alpha, eval);

            // Alpha-Beta 剪枝
        }
    }
}
```

```

        if (beta <= alpha) {
            break; // beta 剪枝
        }
    }

    return maxEval;
} else {
    int minEval = INT_MAX;
    vector<pair<int, int>> moves = generateMoves(board);

    // 启发式排序: 优先考虑中心和角落位置
    sort(moves.begin(), moves.end(), [](const pair<int, int>& a, const pair<int, int>& b)
{
    // 中心位置优先
    if (a.first == 1 && a.second == 1) return true;
    if (b.first == 1 && b.second == 1) return false;
    // 角落位置次之
    if ((a.first == 0 || a.first == 2) && (a.second == 0 || a.second == 2)) return
true;
    if ((b.first == 0 || b.first == 2) && (b.second == 0 || b.second == 2)) return
false;
    return false;
});

for (const auto& move : moves) {
    // 执行移动
    board[move.first][move.second] = PLAYER_0;

    // 递归搜索
    int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);

    // 撤销移动
    board[move.first][move.second] = EMPTY;

    minEval = min(minEval, eval);
    beta = min(beta, eval);

    // Alpha-Beta 剪枝
    if (beta <= alpha) {
        break; // alpha 剪枝
    }
}
}

```

```

        return minEval;
    }
}

/***
 * 获取最佳移动
 *
 * @param board 当前棋盘状态
 * @param depth 搜索深度
 * @param isMaximizingPlayer 是否为最大化玩家
 * @return 最佳移动位置 [row, col]
 */
static pair<int, int> getBestMove(vector<vector<int>>& board, int depth, bool
isMaximizingPlayer) {
    int bestValue = isMaximizingPlayer ? INT_MIN : INT_MAX;
    pair<int, int> bestMove = make_pair(-1, -1);
    vector<pair<int, int>> moves = generateMoves(board);

    // 启发式排序: 优先考虑中心和角落位置
    sort(moves.begin(), moves.end(), [] (const pair<int, int>& a, const pair<int, int>& b) {
        // 中心位置优先
        if (a.first == 1 && a.second == 1) return true;
        if (b.first == 1 && b.second == 1) return false;
        // 角落位置次之
        if ((a.first == 0 || a.first == 2) && (a.second == 0 || a.second == 2)) return true;
        if ((b.first == 0 || b.first == 2) && (b.second == 0 || b.second == 2)) return false;
        return false;
    });

    for (const auto& move : moves) {
        // 执行移动
        board[move.first][move.second] = isMaximizingPlayer ? PLAYER_X : PLAYER_O;

        // 评估移动
        int moveValue = alphaBetaSearch(board, depth - 1,
   isMaximizingPlayer ? INT_MIN : bestValue,
   isMaximizingPlayer ? bestValue : INT_MAX,
   !isMaximizingPlayer);

        // 撤销移动
        board[move.first][move.second] = EMPTY;

        // 更新最佳移动
        if (moveValue > bestValue) {
            bestValue = moveValue;
            bestMove = move;
        }
    }
}

```

```

        if (isMaximizingPlayer && moveValue > bestValue) {
            bestValue = moveValue;
            bestMove = move;
        } else if (!isMaximizingPlayer && moveValue < bestValue) {
            bestValue = moveValue;
            bestMove = move;
        }
    }

    return bestMove;
}

private:
/***
 * 生成所有可能的移动
 *
 * @param board 当前棋盘状态
 * @return 所有可能的移动列表
 */
static vector<pair<int, int>> generateMoves(const vector<vector<int>>& board) {
    vector<pair<int, int>> moves;

    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            if (board[i][j] == EMPTY) {
                moves.push_back(make_pair(i, j));
            }
        }
    }

    return moves;
}

/***
 * 评估游戏状态
 *
 * @param board 当前棋盘状态
 * @return 游戏结果: WIN_SCORE (X 胜)、LOSE_SCORE (O 胜)、TIE_SCORE (平局)、-2 (游戏继续)
 */
static int evaluateGame(const vector<vector<int>>& board) {
    // 检查行
    for (int i = 0; i < BOARD_SIZE; i++) {
        if (board[i][0] != EMPTY && board[i][0] == board[i][1] && board[i][1] == board[i][2])

```

```

{
    return board[i][0] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
}
}

// 检查列
for (int j = 0; j < BOARD_SIZE; j++) {
    if (board[0][j] != EMPTY && board[0][j] == board[1][j] && board[1][j] == board[2][j])
    {
        return board[0][j] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
    }
}

// 检查对角线
if (board[0][0] != EMPTY && board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
    return board[0][0] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
}

if (board[0][2] != EMPTY && board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
    return board[0][2] == PLAYER_X ? WIN_SCORE : LOSE_SCORE;
}

// 检查是否平局
bool isFull = true;
for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
        if (board[i][j] == EMPTY) {
            isFull = false;
            break;
        }
    }
    if (!isFull) break;
}

if (isFull) {
    return TIE_SCORE; // 平局
}

return -2; // 游戏继续
}

public:
/**
```

```

* 打印棋盘
*
* @param board 棋盘状态
*/
static void printBoard(const vector<vector<int>>& board) {
    cout << "Current Board:" << endl;
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            switch (board[i][j]) {
                case EMPTY:
                    cout << ". ";
                    break;
                case PLAYER_X:
                    cout << "X ";
                    break;
                case PLAYER_0:
                    cout << "0 ";
                    break;
            }
        }
        cout << endl;
    }
    cout << endl;
}

/** 
 * 检查移动是否合法
 *
 * @param board 棋盘状态
 * @param row 行索引
 * @param col 列索引
 * @return 是否合法
 */
static bool isValidMove(const vector<vector<int>>& board, int row, int col) {
    return row >= 0 && row < BOARD_SIZE && col >= 0 && col < BOARD_SIZE && board[row][col] == EMPTY;
}

// 测试方法
int main() {
    cout << "==== Alpha-Beta 剪枝算法测试 (井字棋) ===" << endl;
}

```

```

// 初始化空棋盘
vector<vector<int>> board(3, vector<int>(3, 0));

// 简单测试：评估空棋盘
int eval = AlphaBetaPruning::alphaBetaSearch(board, 9, INT_MIN, INT_MAX, true);
cout << "空棋盘评估值：" << eval << endl;

// 测试：X 在中心位置
board[1][1] = AlphaBetaPruning::PLAYER_X;
AlphaBetaPruning::printBoard(board);
eval = AlphaBetaPruning::alphaBetaSearch(board, 8, INT_MIN, INT_MAX, false);
cout << "X 在中心位置的评估值：" << eval << endl;

// 获取最佳移动
pair<int, int> bestMove = AlphaBetaPruning::getBestMove(board, 7, false);
cout << "0的最佳移动: [" << bestMove.first << ", " << bestMove.second << "]" << endl;

// 模拟一局游戏
cout << "\n==== 模拟游戏 ===" << endl;
// 重置棋盘
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        board[i][j] = AlphaBetaPruning::EMPTY;
    }
}

bool isXMove = true;
int movesCount = 0;

while (AlphaBetaPruning::evaluateGame(board) == -2 && movesCount < 9) {
    AlphaBetaPruning::printBoard(board);
    pair<int, int> move = AlphaBetaPruning::getBestMove(board, 9 - movesCount, isXMove);

    if (move.first != -1 && move.second != -1) {
        board[move.first][move.second] = isXMove ? AlphaBetaPruning::PLAYER_X :
        AlphaBetaPruning::PLAYER_0;
        cout << (isXMove ? "X" : "0") << " 下在 [" << move.first << ", " << move.second << "]"
        << endl;
        isXMove = !isXMove;
        movesCount++;
    } else {
        break;
    }
}

```

```

    }

AlphaBetaPruning::printBoard(board);
int result = AlphaBetaPruning::evaluateGame(board);
if (result == AlphaBetaPruning::WIN_SCORE) {
    cout << "X 获胜!" << endl;
} else if (result == AlphaBetaPruning::LOSE_SCORE) {
    cout << "O 获胜!" << endl;
} else {
    cout << "平局!" << endl;
}

return 0;
}
=====
```

文件: alpha\_beta\_pruning.py

```
=====
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Alpha-Beta 剪枝算法实现（以井字棋为例）

```

算法原理:

Alpha-Beta 剪枝是一种优化极小极大搜索算法的技术，通过剪掉搜索树中不会影响最终结果的分支来减少搜索节点数量。

算法特点:

1. 减少搜索空间，提高搜索效率
2. 不影响最终结果的正确性
3. 剪枝效果与搜索顺序密切相关
4. 最好情况下可以将时间复杂度从  $O(b^d)$  降低到  $O(b^{(d/2)})$

算法步骤:

1. 在极小极大搜索过程中维护 alpha 和 beta 值
2. alpha 表示最大化玩家能够保证的最小收益
3. beta 表示最小化玩家能够保证的最大损失
4. 当  $\alpha \geq \beta$  时，剪枝剩余分支

应用场景:

- 博弈树搜索（井字棋、五子棋、象棋等）

- 决策系统
- 对抗性问题求解

时间复杂度:

- 最坏情况:  $O(b^d)$
- 最好情况:  $O(b^{(d/2)})$
- 平均情况:  $O(b^{(3d/4)})$

空间复杂度:  $O(d)$ , 递归栈深度

工程化考量:

1. 移动排序: 优先搜索较优的移动可以提高剪枝效果
  2. 迭代加深: 结合迭代加深搜索使用
  3. 转置表: 缓存已计算的节点避免重复计算
  4. 启发式评估: 设计合理的评估函数
  5. 边界处理: 处理游戏结束状态
  6. 性能优化: 通过剪枝减少不必要的计算
- """

```
class AlphaBetaPruning:
    # 井字棋棋盘状态
    EMPTY = 0
    PLAYER_X = 1
    PLAYER_O = 2

    # 评估分数
    WIN_SCORE = 10000
    LOSE_SCORE = -10000
    TIE_SCORE = 0

    # 棋盘大小
    BOARD_SIZE = 3

    @staticmethod
    def alpha_beta_search(board: list[list[int]], depth: int, alpha: float, beta: float,
    is_maximizing_player: bool) -> int:
        """
        Alpha-Beta 剪枝搜索函数
        """

Args:
```

board: 当前棋盘状态

depth: 当前搜索深度

alpha: Alpha 值 (最大化玩家的最小保证收益)

beta: Beta 值（最小化玩家的最大保证损失）  
is\_maximizing\_player: 是否为最大化玩家（PLAYER\_X）

Returns:

最佳评估分数

"""

# 检查游戏是否结束或达到最大深度

game\_result = AlphaBetaPruning.\_evaluate\_game(board)

if game\_result != -2 or depth == 0: # -2 表示游戏继续

    return game\_result

if is\_maximizing\_player:

    max\_eval = float('-inf')

    moves = AlphaBetaPruning.\_generate\_moves(board)

# 启发式排序：优先考虑中心和角落位置

    moves.sort(key=lambda move: (

        0 if move[0] == 1 and move[1] == 1 else # 中心位置优先

        1 if (move[0] in [0, 2] and move[1] in [0, 2]) else # 角落位置次之

        2 # 边位置最后

))

for move in moves:

    # 执行移动

    board[move[0]][move[1]] = AlphaBetaPruning.PLAYER\_X

    # 递归搜索

    eval\_score = AlphaBetaPruning.alpha\_beta\_search(board, depth - 1, alpha, beta,

False)

    # 撤销移动

    board[move[0]][move[1]] = AlphaBetaPruning.EMPTY

    max\_eval = max(max\_eval, eval\_score)

    alpha = max(alpha, eval\_score)

    # Alpha-Beta 剪枝

    if beta <= alpha:

        break # beta 剪枝

return int(max\_eval)

else:

    min\_eval = float('inf')

```

moves = AlphaBetaPruning._generate_moves(board)

# 启发式排序：优先考虑中心和角落位置
moves.sort(key=lambda move:
    0 if move[0] == 1 and move[1] == 1 else # 中心位置优先
    1 if (move[0] in [0, 2] and move[1] in [0, 2]) else # 角落位置次之
    2 # 边位置最后
))

for move in moves:
    # 执行移动
    board[move[0]][move[1]] = AlphaBetaPruning.PLAYER_0

    # 递归搜索
    eval_score = AlphaBetaPruning.alpha_beta_search(board, depth - 1, alpha, beta,
True)

    # 撤销移动
    board[move[0]][move[1]] = AlphaBetaPruning.EMPTY

    min_eval = min(min_eval, eval_score)
    beta = min(beta, eval_score)

    # Alpha-Beta 剪枝
    if beta <= alpha:
        break # alpha 剪枝

return int(min_eval)

@staticmethod
def get_best_move(board: list[list[int]], depth: int, is_maximizing_player: bool) ->
tuple[int, int]:
    """
    获取最佳移动
    """

```

Args:

- board: 当前棋盘状态
- depth: 搜索深度
- is\_maximizing\_player: 是否为最大化玩家

Returns:

最佳移动位置 (row, col)

"""

```

best_value = float('-inf') if is_maximizing_player else float('inf')
best_move = (-1, -1)
moves = AlphaBetaPruning._generate_moves(board)

# 启发式排序：优先考虑中心和角落位置
moves.sort(key=lambda move: (
    0 if move[0] == 1 and move[1] == 1 else # 中心位置优先
    1 if (move[0] in [0, 2] and move[1] in [0, 2]) else # 角落位置次之
    2 # 边位置最后
))
for move in moves:
    # 执行移动
    board[move[0]][move[1]] = AlphaBetaPruning.PLAYER_X if is_maximizing_player else
AlphaBetaPruning.PLAYER_O

    # 评估移动
    move_value = AlphaBetaPruning.alpha_beta_search(
        board, depth - 1,
        float('-inf') if is_maximizing_player else best_value,
        best_value if is_maximizing_player else float('inf'),
        not is_maximizing_player
    )

    # 撤销移动
    board[move[0]][move[1]] = AlphaBetaPruning.EMPTY

    # 更新最佳移动
    if is_maximizing_player and move_value > best_value:
        best_value = move_value
        best_move = (move[0], move[1])
    elif not is_maximizing_player and move_value < best_value:
        best_value = move_value
        best_move = (move[0], move[1])

return best_move

```

```

@staticmethod
def _generate_moves(board: list[list[int]]) -> list[tuple[int, int]]:
    """
    生成所有可能的移动

```

Args:

board: 当前棋盘状态

Returns:

所有可能的移动列表

"""

moves = []

```
for i in range(AlphaBetaPruning.BOARD_SIZE):
    for j in range(AlphaBetaPruning.BOARD_SIZE):
        if board[i][j] == AlphaBetaPruning.EMPTY:
            moves.append((i, j))
```

return moves

@staticmethod

def \_evaluate\_game(board: list[list[int]]) -> int:

"""

评估游戏状态

Args:

board: 当前棋盘状态

Returns:

游戏结果: WIN\_SCORE (X 胜)、LOSE\_SCORE (O 胜)、TIE\_SCORE (平局)、-2 (游戏继续)

"""

# 检查行

```
for i in range(AlphaBetaPruning.BOARD_SIZE):
    if (board[i][0] != AlphaBetaPruning.EMPTY and
        board[i][0] == board[i][1] == board[i][2]):
        return (AlphaBetaPruning.WIN_SCORE if board[i][0] == AlphaBetaPruning.PLAYER_X
                else AlphaBetaPruning.LOSE_SCORE)
```

# 检查列

```
for j in range(AlphaBetaPruning.BOARD_SIZE):
    if (board[0][j] != AlphaBetaPruning.EMPTY and
        board[0][j] == board[1][j] == board[2][j]):
        return (AlphaBetaPruning.WIN_SCORE if board[0][j] == AlphaBetaPruning.PLAYER_X
                else AlphaBetaPruning.LOSE_SCORE)
```

# 检查对角线

```
if (board[0][0] != AlphaBetaPruning.EMPTY and
    board[0][0] == board[1][1] == board[2][2]):
    return (AlphaBetaPruning.WIN_SCORE if board[0][0] == AlphaBetaPruning.PLAYER_X
```

```

        else AlphaBetaPruning.LOSE_SCORE)

    if (board[0][2] != AlphaBetaPruning.EMPTY and
        board[0][2] == board[1][1] == board[2][0]):
        return (AlphaBetaPruning.WIN_SCORE if board[0][2] == AlphaBetaPruning.PLAYER_X
                else AlphaBetaPruning.LOSE_SCORE)

# 检查是否平局
is_full = all(board[i][j] != AlphaBetaPruning.EMPTY
              for i in range(AlphaBetaPruning.BOARD_SIZE)
              for j in range(AlphaBetaPruning.BOARD_SIZE))

if is_full:
    return AlphaBetaPruning.TIE_SCORE # 平局

return -2 # 游戏继续

```

@staticmethod

def print\_board(board: list[list[int]]) -> None:

"""

打印棋盘

Args:

board: 棋盘状态
"""

print("Current Board:")
for i in range(AlphaBetaPruning.BOARD\_SIZE):
 for j in range(AlphaBetaPruning.BOARD\_SIZE):
 if board[i][j] == AlphaBetaPruning.EMPTY:
 print(".", end="")
 elif board[i][j] == AlphaBetaPruning.PLAYER\_X:
 print("X ", end="")
 elif board[i][j] == AlphaBetaPruning.PLAYER\_0:
 print("0 ", end="")
 print()
print()

@staticmethod

def is\_valid\_move(board: list[list[int]], row: int, col: int) -> bool:

"""

检查移动是否合法

Args:

board: 棋盘状态

row: 行索引

col: 列索引

Returns:

是否合法

"""

```
return (0 <= row < AlphaBetaPruning.BOARD_SIZE and
       0 <= col < AlphaBetaPruning.BOARD_SIZE and
       board[row][col] == AlphaBetaPruning.EMPTY)
```

```
def main():
```

"""测试示例"""

```
print("== Alpha-Beta 剪枝算法测试（井字棋） ==")
```

# 初始化空棋盘

```
board = [[0 for _ in range(3)] for _ in range(3)]
```

# 简单测试：评估空棋盘

```
eval_score = AlphaBetaPruning.alpha_beta_search(board, 9, float('-inf'), float('inf'), True)
print(f"空棋盘评估值: {eval_score}")
```

# 测试：X 在中心位置

```
board[1][1] = AlphaBetaPruning.PLAYER_X
```

```
AlphaBetaPruning.print_board(board)
```

```
eval_score = AlphaBetaPruning.alpha_beta_search(board, 8, float('-inf'), float('inf'), False)
print(f"X 在中心位置的评估值: {eval_score}")
```

# 获取最佳移动

```
best_move = AlphaBetaPruning.get_best_move(board, 7, False)
```

```
print(f"0 的最佳移动: [{best_move[0]}, {best_move[1]}]")
```

# 模拟一局游戏

```
print("\n== 模拟游戏 ==")
```

# 重置棋盘

```
for i in range(3):
```

```
    for j in range(3):
```

```
        board[i][j] = AlphaBetaPruning.EMPTY
```

```
is_x_move = True
```

```
moves_count = 0
```

```

while AlphaBetaPruning._evaluate_game(board) == -2 and moves_count < 9:
    AlphaBetaPruning.print_board(board)
    move = AlphaBetaPruning.get_best_move(board, 9 - moves_count, is_x_move)

    if move[0] != -1 and move[1] != -1:
        board[move[0]][move[1]] = AlphaBetaPruning.PLAYER_X if is_x_move else
AlphaBetaPruning.PLAYER_0
        print(f"{'X' if is_x_move else 'O'} 下在 [{move[0]}, {move[1]}]")
        is_x_move = not is_x_move
        moves_count += 1
    else:
        break

AlphaBetaPruning.print_board(board)
result = AlphaBetaPruning._evaluate_game(board)
if result == AlphaBetaPruning.WIN_SCORE:
    print("X 获胜!")
elif result == AlphaBetaPruning.LOSE_SCORE:
    print("O 获胜!")
else:
    print("平局!")

if __name__ == "__main__":
    main()

```

=====

文件: Code01\_Subsequences.java

=====

```

package class038;

import java.util.HashSet;
import java.util.ArrayList;
import java.util.List;

// 字符串的全部子序列
// 子序列本身是可以有重复的，只是这个题目要求去重
// 测试链接：https://www.nowcoder.com/practice/92e6247998294f2c933906fdedbc6e6a
public class Code01_Subsequences {

    /**
     * 生成字符串的所有子序列（方法1）

```

```

* 使用 StringBuilder 构建路径
*
* 算法思路:
* 1. 对于每个字符, 我们有两种选择: 包含在子序列中或不包含
* 2. 通过递归实现这种选择过程
* 3. 当遍历完所有字符时, 将当前路径加入结果集
*
* 时间复杂度: O(2^n * n), 其中 n 为字符串长度
* 空间复杂度: O(2^n * n), 用于存储所有子序列
*
* @param str 输入字符串
* @return 所有不重复的子序列
*/
public static String[] generatePermutation1(String str) {
    char[] s = str.toCharArray();
    HashSet<String> set = new HashSet<>();
    f1(s, 0, new StringBuilder(), set);
    int m = set.size();
    String[] ans = new String[m];
    int i = 0;
    for (String cur : set) {
        ans[i++] = cur;
    }
    return ans;
}

```

```

// s[i...], 之前决定的路径 path, set 收集结果时去重
public static void f1(char[] s, int i, StringBuilder path, HashSet<String> set) {
    if (i == s.length) {
        set.add(path.toString());
    } else {
        path.append(s[i]); // 加到路径中去
        f1(s, i + 1, path, set);
        path.deleteCharAt(path.length() - 1); // 从路径中移除
        f1(s, i + 1, path, set);
    }
}

```

```

/**
* 生成字符串的所有子序列 (方法 2)
* 使用字符数组构建路径
*
* 算法思路:

```

```

* 与方法 1 相同，但使用字符数组和 size 变量来维护路径
* 这种方法避免了 StringBuilder 的频繁创建和销毁
*
* 时间复杂度: O(2^n * n)
* 空间复杂度: O(2^n * n)
*
* @param str 输入字符串
* @return 所有不重复的子序列
*/
public static String[] generatePermutation2(String str) {
    char[] s = str.toCharArray();
    HashSet<String> set = new HashSet<>();
    f2(s, 0, new char[s.length], 0, set);
    int m = set.size();
    String[] ans = new String[m];
    int i = 0;
    for (String cur : set) {
        ans[i++] = cur;
    }
    return ans;
}

public static void f2(char[] s, int i, char[] path, int size, HashSet<String> set) {
    if (i == s.length) {
        set.add(String.valueOf(path, 0, size));
    } else {
        path[size] = s[i];
        f2(s, i + 1, path, size + 1, set);
        f2(s, i + 1, path, size, set);
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String test1 = "abc";
    String[] result1 = generatePermutation1(test1);
    System.out.println("输入: " + test1);
    System.out.print("输出: [");
    for (int i = 0; i < result1.length; i++) {
        System.out.print("\\" + result1[i] + "\\");
        if (i < result1.length - 1) System.out.print(", ");
    }
}

```

```
System.out.println("]");

// 测试用例 2
String test2 = "aab";
String[] result2 = generatePermutation2(test2);
System.out.println("\n 输入: " + test2);
System.out.print("输出: [");
for (int i = 0; i < result2.length; i++) {
    System.out.print("'" + result2[i] + "'");
    if (i < result2.length - 1) System.out.print(", ");
}
System.out.println("]");
}

}

=====
```

文件: Code02\_Combinations.java

```
=====
package class038;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// 给你一个整数数组 nums，其中可能包含重复元素，请你返回该数组所有可能的组合
// 答案 不能 包含重复的组合。返回的答案中，组合可以按 任意顺序 排列
// 注意其实要求返回的不是子集，因为子集一定是不包含相同元素的，要返回的其实是不重复的组合
// 比如输入: nums = [1, 2, 2]
// 输出: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]
// 测试链接 : https://leetcode.cn/problems/subsets-ii/
public class Code02_Combinations {

    /**
     * 返回数组所有可能的不重复组合
     *
     * 算法思路:
     * 1. 首先对数组排序，使相同元素相邻
     * 2. 使用回溯算法，对于每组相同元素，统一处理选择 0 个、1 个、2 个... 的情况
     * 3. 通过跳过相同元素避免重复组合
     *
     * 时间复杂度: O(2^n * n)，其中 n 为数组长度
    
```

```

* 空间复杂度: O(n), 递归栈深度
*
* @param nums 输入数组
* @return 所有不重复的组合
*/
public static List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    // 排序是去重的关键步骤
    Arrays.sort(nums);
    f(nums, 0, new int[nums.length], 0, ans);
    return ans;
}

/**
* 递归函数生成所有组合
*
* @param nums 输入数组
* @param i 当前处理到的索引
* @param path 当前路径 (组合)
* @param size 当前路径的大小
* @param ans 结果集合
*/
public static void f(int[] nums, int i, int[] path, int size, List<List<Integer>> ans) {
    if (i == nums.length) {
        // 将当前路径加入结果集
        ArrayList<Integer> cur = new ArrayList<>();
        for (int j = 0; j < size; j++) {
            cur.add(path[j]);
        }
        ans.add(cur);
    } else {
        // 找到下一组第一个不同元素的位置
        int j = i + 1;
        while (j < nums.length && nums[i] == nums[j]) {
            j++;
        }
        // 当前数 nums[i], 选择 0 个
        f(nums, j, path, size, ans);
        // 当前数 nums[i], 选择 1 个、2 个、3 个...都尝试
        for (; i < j; i++) {
            path[size++] = nums[i];
            f(nums, j, path, size, ans);
        }
    }
}

```

```
    }

}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] test1 = {1, 2, 2};
    List<List<Integer>> result1 = subsetsWithDup(test1);
    System.out.println("输入: [1, 2, 2]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] test2 = {0};
    List<List<Integer>> result2 = subsetsWithDup(test2);
    System.out.println("\n 输入: [0]");
    System.out.println("输出: " + result2);
}
```

}

=====

文件: Code03\_Permutations.java

```
=====
package class038;

import java.util.ArrayList;
import java.util.List;

// 没有重复项数字的全排列
// 测试链接 : https://leetcode.cn/problems/permute/
public class Code03_Permutations {

    /**
     * 生成数组的所有全排列（无重复元素）
     *
     * 算法思路:
     * 1. 使用回溯算法，通过交换元素位置生成所有排列
     * 2. 对于位置 i，尝试将后面每个元素交换到位置 i
     * 3. 递归处理位置 i+1
     * 4. 回溯时恢复交换前的状态
     *
     * 时间复杂度: O(n! * n)，其中 n 为数组长度
}
```

```

* 空间复杂度: O(n), 递归栈深度
*
* @param nums 输入数组 (无重复元素)
* @return 所有全排列
*/
public static List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    f(nums, 0, ans);
    return ans;
}

/***
 * 递归生成排列
 *
 * @param nums 数组
 * @param i 当前处理的位置
 * @param ans 结果集合
 */
public static void f(int[] nums, int i, List<List<Integer>> ans) {
    if (i == nums.length) {
        // 已经处理完所有位置, 将当前排列加入结果集
        List<Integer> cur = new ArrayList<>();
        for (int num : nums) {
            cur.add(num);
        }
        ans.add(cur);
    } else {
        // 尝试将位置 j 的元素交换到位置 i
        for (int j = i; j < nums.length; j++) {
            swap(nums, i, j);
            f(nums, i + 1, ans);
            swap(nums, i, j); // 回溯, 恢复状态
        }
    }
}

/***
 * 交换数组中两个位置的元素
 *
 * @param nums 数组
 * @param i 位置 i
 * @param j 位置 j
 */

```

```

public static void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = { 1, 2, 3 };
    List<List<Integer>> ans1 = permute(nums1);
    System.out.println("输入: [1, 2, 3]");
    System.out.println("输出: " + ans1);

    // 测试用例 2
    int[] nums2 = { 0, 1 };
    List<List<Integer>> ans2 = permute(nums2);
    System.out.println("\n 输入: [0, 1]");
    System.out.println("输出: " + ans2);

    // 测试用例 3
    int[] nums3 = { 1 };
    List<List<Integer>> ans3 = permute(nums3);
    System.out.println("\n 输入: [1]");
    System.out.println("输出: " + ans3);
}

```

}

=====

文件: Code04\_PermutationWithoutRepetition.java

```

=====
package class038;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

// 有重复项数组的去重全排列
// 测试链接 : https://leetcode.cn/problems/permute-ii/
public class Code04_PermutationWithoutRepetition {

    /**

```

```

* 生成数组的所有不重复全排列（可能包含重复元素）
*
* 算法思路：
* 1. 使用回溯算法生成排列
* 2. 在每个位置，使用 HashSet 记录已经放置过的元素，避免重复
* 3. 对于位置 i，只尝试将未在该位置放置过的元素交换到位置 i
*
* 时间复杂度：O(n! * n)，其中 n 为数组长度
* 空间复杂度：O(n)，递归栈深度
*
* @param nums 输入数组（可能包含重复元素）
* @return 所有不重复的全排列
*/

```

```

public static List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    f(nums, 0, ans);
    return ans;
}

/**
 * 递归生成不重复排列
 *
 * @param nums 数组
 * @param i 当前处理的位置
 * @param ans 结果集合
 */

```

```

public static void f(int[] nums, int i, List<List<Integer>> ans) {
    if (i == nums.length) {
        // 已经处理完所有位置，将当前排列加入结果集
        List<Integer> cur = new ArrayList<>();
        for (int num : nums) {
            cur.add(num);
        }
        ans.add(cur);
    } else {
        // 使用 HashSet 记录在位置 i 已经放置过的元素
        HashSet<Integer> set = new HashSet<>();
        for (int j = i; j < nums.length; j++) {
            // nums[j]没有来到过 i 位置，才会去尝试
            if (!set.contains(nums[j])) {
                set.add(nums[j]);
                swap(nums, i, j);
                f(nums, i + 1, ans);
            }
        }
    }
}

```

```

        swap(nums, i, j);
    }
}
}

/**
 * 交换数组中两个位置的元素
 *
 * @param nums 数组
 * @param i 位置 i
 * @param j 位置 j
 */
public static void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 1, 2};
    List<List<Integer>> ans1 = permuteUnique(nums1);
    System.out.println("输入: [1,1,2]");
    System.out.println("输出: " + ans1);

    // 测试用例 2
    int[] nums2 = {1, 2, 1, 1};
    List<List<Integer>> ans2 = permuteUnique(nums2);
    System.out.println("\n 输入: [1,2,1,1]");
    System.out.println("输出: " + ans2);
}

}

```

文件: Code05\_ReverseStackWithRecursive.java

```
package class038;
```

```
import java.util.Stack;
```

```
// 用递归函数逆序栈
public class Code05_ReverseStackWithRecursive {

    /**
     * 仅使用递归函数将栈逆序
     *
     * 算法思路:
     * 1. 递归地移除栈底元素
     * 2. 逆序处理剩余元素
     * 3. 将之前移除的栈底元素压入栈顶
     *
     * 时间复杂度: O(n^2), 其中 n 为栈的大小
     * 空间复杂度: O(n), 递归栈深度
     *
     * @param stack 待逆序的栈
     */
    public static void reverse(Stack<Integer> stack) {
        if (stack.isEmpty()) {
            return;
        }
        int num = bottomOut(stack);
        reverse(stack);
        stack.push(num);
    }

    /**
     * 移除并返回栈底元素, 其他元素向下移动
     *
     * @param stack 栈
     * @return 栈底元素
     */
    // 栈底元素移除掉, 上面的元素盖下来
    // 返回移除掉的栈底元素
    public static int bottomOut(Stack<Integer> stack) {
        int ans = stack.pop();
        if (stack.isEmpty()) {
            return ans;
        } else {
            int last = bottomOut(stack);
            stack.push(ans);
            return last;
        }
    }
}
```

```
}

public static void main(String[] args) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    stack.push(5);

    System.out.println("逆序前: " + stack);
    reverse(stack);
    System.out.println("逆序后: " + stack);

    // 测试空栈
    Stack<Integer> emptyStack = new Stack<Integer>();
    System.out.println("\n空栈逆序前: " + emptyStack);
    reverse(emptyStack);
    System.out.println("空栈逆序后: " + emptyStack);
}

}
```

=====

文件: Code06\_SortStackWithRecursive.java

=====

```
package class038;

import java.util.Stack;

// 用递归函数排序栈
// 栈只提供 push、pop、isEmpty 三个方法
// 请完成无序栈的排序，要求排完序之后，从栈顶到栈底从小到大
// 只能使用栈提供的 push、pop、isEmpty 三个方法、以及递归函数
// 除此之外不能使用任何的容器，数组也不行
// 就是排序过程中只能用：
// 1) 栈提供的 push、pop、isEmpty 三个方法
// 2) 递归函数，并且返回值最多为单个整数
public class Code06_SortStackWithRecursive {

    /**
     * 仅使用递归函数对栈进行排序（从栈顶到栈底从小到大）
}
```

```

*
* 算法思路:
* 1. 统计栈的深度
* 2. 找到当前深度范围内的最大值及其出现次数
* 3. 将最大值沉底, 其余元素保持相对顺序
* 4. 递归处理剩余元素
*
* 时间复杂度: O(n^2), 其中 n 为栈的大小
* 空间复杂度: O(n), 递归栈深度
*
* @param stack 待排序的栈
*/
public static void sort(Stack<Integer> stack) {
    int deep = deep(stack);
    while (deep > 0) {
        int max = max(stack, deep);
        int k = times(stack, deep, max);
        down(stack, deep, max, k);
        deep -= k;
    }
}

/**
* 返回栈的深度, 不改变栈的数据状况
*
* @param stack 栈
* @return 栈的深度
*/
// 返回栈的深度
// 不改变栈的数据状况
public static int deep(Stack<Integer> stack) {
    if (stack.isEmpty()) {
        return 0;
    }
    int num = stack.pop();
    int deep = deep(stack) + 1;
    stack.push(num);
    return deep;
}

/**
* 从栈当前的顶部开始, 往下数 deep 层, 返回这 deep 层里的最大值
*

```

```

* @param stack 栈
* @param deep 深度
* @return 最大值
*/
// 从栈当前的顶部开始，往下数 deep 层
// 返回这 deep 层里的最大值
public static int max(Stack<Integer> stack, int deep) {
    if (deep == 0) {
        return Integer.MIN_VALUE;
    }
    int num = stack.pop();
    int restMax = max(stack, deep - 1);
    int max = Math.max(num, restMax);
    stack.push(num);
    return max;
}

/**
* 从栈当前的顶部开始，往下数 deep 层，已知最大值是 max 了，返回 max 出现了几次，不改变栈的数据状况
*
* @param stack 栈
* @param deep 深度
* @param max 最大值
* @return 最大值出现的次数
*/
// 从栈当前的顶部开始，往下数 deep 层，已知最大值是 max 了
// 返回，max 出现了几次，不改变栈的数据状况
public static int times(Stack<Integer> stack, int deep, int max) {
    if (deep == 0) {
        return 0;
    }
    int num = stack.pop();
    int restTimes = times(stack, deep - 1, max);
    int times = restTimes + (num == max ? 1 : 0);
    stack.push(num);
    return times;
}

/**
* 从栈当前的顶部开始，往下数 deep 层，已知最大值是 max，出现了 k 次，
* 请把这 k 个最大值沉底，剩下的数据状况不变
*

```

```

* @param stack 栈
* @param deep 深度
* @param max 最大值
* @param k 最大值出现的次数
*/
// 从栈当前的顶部开始，往下数 deep 层，已知最大值是 max，出现了 k 次
// 请把这 k 个最大值沉底，剩下的数据状况不变
public static void down(Stack<Integer> stack, int deep, int max, int k) {
    if (deep == 0) {
        for (int i = 0; i < k; i++) {
            stack.push(max);
        }
    } else {
        int num = stack.pop();
        down(stack, deep - 1, max, k);
        if (num != max) {
            stack.push(num);
        }
    }
}

// 为了测试
// 生成随机栈
public static Stack<Integer> randomStack(int n, int v) {
    Stack<Integer> ans = new Stack<Integer>();
    for (int i = 0; i < n; i++) {
        ans.add((int) (Math.random() * v));
    }
    return ans;
}

// 为了测试
// 检测栈是不是从顶到底依次有序
public static boolean isSorted(Stack<Integer> stack) {
    int step = Integer.MIN_VALUE;
    while (!stack.isEmpty()) {
        if (step > stack.peek()) {
            return false;
        }
        step = stack.pop();
    }
    return true;
}

```

```
// 为了测试
public static void main(String[] args) {
    Stack<Integer> test = new Stack<Integer>();
    test.add(1);
    test.add(5);
    test.add(4);
    test.add(5);
    test.add(3);
    test.add(2);
    test.add(3);
    test.add(1);
    test.add(4);
    test.add(2);

    System.out.println("排序前: " + test);
    sort(test);
    System.out.println("排序后: " + test);

    // 随机测试
    int N = 20;
    int V = 20;
    int testTimes = 20000;
    System.out.println("\n开始随机测试... ");
    for (int i = 0; i < testTimes; i++) {
        int n = (int) (Math.random() * N);
        Stack<Integer> stack = randomStack(n, V);
        sort(stack);
        if (!isSorted(stack)) {
            System.out.println("出错了!");
            break;
        }
    }
    System.out.println("随机测试结束");
}

}
```

=====

文件: Code07\_TowerOfHanoi.java

=====

```
package class038;
```

```
// 打印 n 层汉诺塔问题的最优移动轨迹
public class Code07_TowerOfHanoi {

    /**
     * 打印 n 层汉诺塔问题的最优移动轨迹
     *
     * 算法思路：
     * 1. 将 n-1 个盘子从起始柱借助目标柱移动到辅助柱
     * 2. 将第 n 个盘子从起始柱移动到目标柱
     * 3. 将 n-1 个盘子从辅助柱借助起始柱移动到目标柱
     *
     * 时间复杂度：O(2^n)
     * 空间复杂度：O(n)
     *
     * @param n 盘子数量
     */
    public static void hanoi(int n) {
        if (n > 0) {
            f(n, "左", "右", "中");
        }
    }

    /**
     * 递归移动盘子
     *
     * @param i 盘子数量
     * @param from 起始柱
     * @param to 目标柱
     * @param other 辅助柱
     */
    public static void f(int i, String from, String to, String other) {
        if (i == 1) {
            System.out.println("移动圆盘 1 从 " + from + " 到 " + to);
        } else {
            f(i - 1, from, other, to);
            System.out.println("移动圆盘 " + i + " 从 " + from + " 到 " + to);
            f(i - 1, other, to, from);
        }
    }

    public static void main(String[] args) {
        int n = 3;
```

```
System.out.println("汉诺塔移动步骤 (n=" + n + ") :");
hanoi(n);

System.out.println("\n 汉诺塔移动步骤 (n=2) :");
hanoi(2);

}

}

=====
```

文件: Code08\_LetterCombinations.cpp

```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

/***
 * LeetCode 17. 电话号码的字母组合
 *
 * 题目描述:
 * 给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。
 * 答案可以按任意顺序返回。
 *
 * 示例:
 * 输入: digits = "23"
 * 输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
 *
 * 输入: digits = ""
 * 输出: []
 *
 * 输入: digits = "2"
 * 输出: ["a", "b", "c"]
 *
 * 提示:
 * 0 <= digits.length <= 4
 * digits[i] 是范围 [‘2’ , ‘9’] 的一个数字。
 *
 * 链接: https://leetcode.cn/problems/letter-combinations-of-a-phone-number/
 */
```

```

class Solution {
public:
    /**
     * 生成电话号码的字母组合
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有可能的字母组合
     * 2. 建立数字到字母的映射关系
     * 3. 对于每个数字，遍历其对应的所有字母
     * 4. 递归处理下一个数字，直到处理完所有数字
     *
     * 时间复杂度: O(3^m * 4^n)，其中 m 是对应 3 个字母的数字个数，n 是对应 4 个字母的数字个数
     * 空间复杂度: O(3^m * 4^n)，用于存储所有组合
     *
     * @param digits 输入的数字字符串
     * @return 所有可能的字母组合
     */
    vector<string> letterCombinations(string digits) {
        vector<string> result;
        // 边界条件: 空字符串
        if (digits.empty()) return result;

        // 数字到字母的映射
        vector<string> mapping = {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"} ;

        // 回溯生成所有组合
        backtrack(digits, mapping, result, "", 0);
        return result;
    }

private:
    /**
     * 回溯函数生成字母组合
     *
     * @param digits 输入的数字字符串
     * @param mapping 数字到字母的映射数组
     * @param result 结果列表
     * @param current 当前已生成的字符串
     * @param index 当前处理的数字索引
     */
    void backtrack(const string& digits, const vector<string>& mapping, vector<string>& result,
string current, int index) {

```

```
// 终止条件：已处理完所有数字
if (index == digits.length()) {
    result.push_back(current);
    return;
}

// 获取当前数字对应的字母
int digit = digits[index] - '0';
string letters = mapping[digit];

// 遍历所有可能的字母
for (char letter : letters) {
    // 递归处理下一个数字
    backtrack(digits, mapping, result, current + letter, index + 1);
}
}

};

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

    // 测试用例 1
    string test1 = "23";
    vector<string> result1 = solution.letterCombinations(test1);
    cout << "输入: " << test1 << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << result1[i] << ",";
        if (i < result1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;

    // 测试用例 2
    string test2 = "";
    vector<string> result2 = solution.letterCombinations(test2);
    cout << "\n输入: " << test2 << endl;
    cout << "输出: [";
    for (int i = 0; i < result2.size(); i++) {
        cout << result2[i] << ",";
        if (i < result2.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
```

```

// 测试用例 3
string test3 = "2";
vector<string> result3 = solution.letterCombinations(test3);
cout << "\n 输入: \"\" << test3 << "\"\" << endl;
cout << "输出: [";
for (int i = 0; i < result3.size(); i++) {
    cout << "\"\" << result3[i] << "\"";
    if (i < result3.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 4
string test4 = "234";
vector<string> result4 = solution.letterCombinations(test4);
cout << "\n 输入: \"\" << test4 << "\"\" << endl;
cout << "输出: [";
for (int i = 0; i < result4.size(); i++) {
    cout << "\"\" << result4[i] << "\"";
    if (i < result4.size() - 1) cout << ", ";
}
cout << "]" << endl;

return 0;
}

```

=====

文件: Code08\_LetterCombinations.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 17. 电话号码的字母组合
 *
 * 题目描述:
 * 给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。
 * 答案可以按任意顺序返回。
 *
 * 示例:

```

```

* 输入: digits = "23"
* 输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
*
* 输入: digits = ""
* 输出: []
*
* 输入: digits = "2"
* 输出: ["a", "b", "c"]
*
* 提示:
* 0 <= digits.length <= 4
* digits[i] 是范围 [‘2’, ‘9’] 的一个数字。
*
* 链接: https://leetcode.cn/problems/letter-combinations-of-a-phone-number/
*/
public class Code08_LetterCombinations {

    /**
     * 生成电话号码的字母组合
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有可能的字母组合
     * 2. 建立数字到字母的映射关系
     * 3. 对于每个数字，遍历其对应的所有字母
     * 4. 递归处理下一个数字，直到处理完所有数字
     *
     * 时间复杂度: O(3^m * 4^n)，其中 m 是对应 3 个字母的数字个数，n 是对应 4 个字母的数字个数
     * 空间复杂度: O(3^m * 4^n)，用于存储所有组合
     *
     * @param digits 输入的数字字符串
     * @return 所有可能的字母组合
     */
    public static List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<>();
        // 边界条件: 空字符串
        if (digits == null || digits.length() == 0) return result;

        // 数字到字母的映射
        String[] mapping = new String[] {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tuv", "wxyz"} ;

        // 回溯生成所有组合
        backtrack(digits, mapping, result, "", 0);
    }
}

```

```
        return result;
    }

/**
 * 回溯函数生成字母组合
 *
 * @param digits 输入的数字字符串
 * @param mapping 数字到字母的映射数组
 * @param result 结果列表
 * @param current 当前已生成的字符串
 * @param index 当前处理的数字索引
 */
private static void backtrack(String digits, String[] mapping, List<String> result, String current, int index) {
    // 终止条件：已处理完所有数字
    if (index == digits.length()) {
        result.add(current);
        return;
    }

    // 获取当前数字对应的字母
    int digit = digits.charAt(index) - '0';
    String letters = mapping[digit];

    // 遍历所有可能的字母
    for (int i = 0; i < letters.length(); i++) {
        char letter = letters.charAt(i);
        // 递归处理下一个数字
        backtrack(digits, mapping, result, current + letter, index + 1);
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String test1 = "23";
    List<String> result1 = letterCombinations(test1);
    System.out.println("输入: " + test1 + ")");
    System.out.println("输出: " + result1);

    // 测试用例 2
    String test2 = "";
    List<String> result2 = letterCombinations(test2);
```

```
System.out.println("\n输入: " + test2 + "\n");
System.out.println("输出: " + result2);

// 测试用例 3
String test3 = "2";
List<String> result3 = letterCombinations(test3);
System.out.println("\n输入: " + test3 + "\n");
System.out.println("输出: " + result3);

// 测试用例 4
String test4 = "234";
List<String> result4 = letterCombinations(test4);
System.out.println("\n输入: " + test4 + "\n");
System.out.println("输出: " + result4);

}

}

=====

文件: Code08_LetterCombinations.py
=====

"""

LeetCode 17. 电话号码的字母组合
```

题目描述:

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

答案可以按任意顺序返回。

示例:

输入: digits = "23"

输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

输入: digits = ""

输出: []

输入: digits = "2"

输出: ["a", "b", "c"]

提示:

$0 \leq \text{digits.length} \leq 4$

$\text{digits}[i]$  是范围  $['2', '9']$  的一个数字。

链接: <https://leetcode.cn/problems/letter-combinations-of-a-phone-number/>

```
"""
```

```
class Solution:  
    def letterCombinations(self, digits):  
        """  
        生成电话号码的字母组合  
        """
```

算法思路：

1. 使用回溯算法生成所有可能的字母组合
2. 建立数字到字母的映射关系
3. 对于每个数字，遍历其对应的所有字母
4. 递归处理下一个数字，直到处理完所有数字

时间复杂度： $O(3^m * 4^n)$ ，其中  $m$  是对应 3 个字母的数字个数， $n$  是对应 4 个字母的数字个数

空间复杂度： $O(3^m * 4^n)$ ，用于存储所有组合

```
:param digits: 输入的数字字符串  
:return: 所有可能的字母组合  
"""  
  
# 边界条件：空字符串  
if not digits:  
    return []  
  
# 数字到字母的映射  
mapping = ["0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]
```

```
result = []  
# 回溯生成所有组合  
self.backtrack(digits, mapping, result, "", 0)  
return result
```

```
def backtrack(self, digits, mapping, result, current, index):
```

```
"""
```

回溯函数生成字母组合

```
:param digits: 输入的数字字符串  
:param mapping: 数字到字母的映射数组  
:param result: 结果列表  
:param current: 当前已生成的字符串  
:param index: 当前处理的数字索引  
"""  
  
# 终止条件：已处理完所有数字  
if index == len(digits):
```

```
    result.append(current)
    return

# 获取当前数字对应的字母
digit = int(digits[index])
letters = mapping[digit]

# 遍历所有可能的字母
for letter in letters:
    # 递归处理下一个数字
    self.backtrack(digits, mapping, result, current + letter, index + 1)

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    test1 = "23"
    result1 = solution.letterCombinations(test1)
    print(f'输入: "{test1}"')
    print(f"输出: {result1}")

    # 测试用例 2
    test2 = ""
    result2 = solution.letterCombinations(test2)
    print(f'\n输入: "{test2}"')
    print(f"输出: {result2}")

    # 测试用例 3
    test3 = "2"
    result3 = solution.letterCombinations(test3)
    print(f'\n输入: "{test3}"')
    print(f"输出: {result3}")

    # 测试用例 4
    test4 = "234"
    result4 = solution.letterCombinations(test4)
    print(f'\n输入: "{test4}"')
    print(f"输出: {result4}")

if __name__ == "__main__":
```

```
main()
```

```
=====
```

文件: Code09\_GenerateParentheses.cpp

```
#include <vector>
```

```
#include <string>
```

```
#include <iostream>
```

```
using namespace std;
```

```
/**
```

```
* LeetCode 22. 括号生成
```

```
*
```

```
* 题目描述:
```

```
* 数字 n 代表生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且有效的括号组合。
```

```
*
```

```
* 示例:
```

```
* 输入: n = 3
```

```
* 输出: ["((()))","(()())","(())()","()((()))","()()()"]
```

```
*
```

```
* 输入: n = 1
```

```
* 输出: ["()"]
```

```
*
```

```
* 提示:
```

```
* 1 <= n <= 8
```

```
*
```

```
* 链接: https://leetcode.cn/problems/generate-parentheses/
```

```
*/
```

```
class Solution {
```

```
public:
```

```
/**
```

```
* 生成所有可能的有效括号组合
```

```
*
```

```
* 算法思路:
```

```
* 1. 使用回溯算法生成有效括号组合
```

```
* 2. 维护左右括号的数量, 确保生成的括号始终有效
```

```
* 3. 左括号数量不能超过 n
```

```
* 4. 右括号数量不能超过左括号数量
```

```
*
```

```
* 时间复杂度: O(4^n / sqrt(n)), 第 n 个卡塔兰数
```

```

* 空间复杂度: O(4^n / sqrt(n)), 用于存储所有组合
*
* @param n 括号对数
* @return 所有可能的有效括号组合
*/
vector<string> generateParenthesis(int n) {
    vector<string> result;
    backtrack(result, "", 0, 0, n);
    return result;
}

private:
/***
 * 回溯函数生成有效括号组合
 *
 * @param result 结果列表
 * @param current 当前已生成的字符串
 * @param open 已使用的左括号数量
 * @param close 已使用的右括号数量
 * @param max 括号对数
 */
void backtrack(vector<string>& result, string current, int open, int close, int max) {
    // 终止条件: 已生成 2*max 个字符
    if (current.length() == max * 2) {
        result.push_back(current);
        return;
    }

    // 添加左括号 (左括号数量小于 max 时)
    if (open < max)
        backtrack(result, current + "(", open + 1, close, max);

    // 添加右括号 (右括号数量小于左括号数量时)
    if (close < open)
        backtrack(result, current + ")", open, close + 1, max);
}

};

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

    // 测试用例 1

```

```

int n1 = 3;
vector<string> result1 = solution.generateParenthesis(n1);
cout << "输入: n = " << n1 << endl;
cout << "输出: [";
for (int i = 0; i < result1.size(); i++) {
    cout << "\"" << result1[i] << "\"";
    if (i < result1.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 2
int n2 = 1;
vector<string> result2 = solution.generateParenthesis(n2);
cout << "\n 输入: n = " << n2 << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "\"" << result2[i] << "\"";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 3
int n3 = 2;
vector<string> result3 = solution.generateParenthesis(n3);
cout << "\n 输入: n = " << n3 << endl;
cout << "输出: [";
for (int i = 0; i < result3.size(); i++) {
    cout << "\"" << result3[i] << "\"";
    if (i < result3.size() - 1) cout << ", ";
}
cout << "]" << endl;

return 0;
}

```

=====

文件: Code09\_GenerateParentheses.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.List;

```

```
/**  
 * LeetCode 22. 括号生成  
 *  
 * 题目描述:  
 * 数字 n 代表生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且有效的括号组合。  
 *  
 * 示例:  
 * 输入: n = 3  
 * 输出: ["((()))","(()())","(())()","()((()))","()()()"]  
 *  
 * 输入: n = 1  
 * 输出: ["()"]  
 *  
 * 提示:  
 * 1 <= n <= 8  
 *  
 * 链接: https://leetcode.cn/problems/generate-parentheses/  
 */  
public class Code09_GenerateParentheses {  
  
    /**  
     * 生成所有可能的有效括号组合  
     *  
     * 算法思路:  
     * 1. 使用回溯算法生成有效括号组合  
     * 2. 维护左右括号的数量, 确保生成的括号始终有效  
     * 3. 左括号数量不能超过 n  
     * 4. 右括号数量不能超过左括号数量  
     *  
     * 时间复杂度: O(4^n / sqrt(n)), 第 n 个卡塔兰数  
     * 空间复杂度: O(4^n / sqrt(n)), 用于存储所有组合  
     *  
     * @param n 括号对数  
     * @return 所有可能的有效括号组合  
    */  
    public static List<String> generateParenthesis(int n) {  
        List<String> result = new ArrayList<>();  
        backtrack(result, "", 0, 0, n);  
        return result;  
    }  
  
    /**
```

```
* 回溯函数生成有效括号组合
*
* @param result 结果列表
* @param current 当前已生成的字符串
* @param open 已使用的左括号数量
* @param close 已使用的右括号数量
* @param max 括号对数
*/
private static void backtrack(List<String> result, String current, int open, int close, int max) {
    // 终止条件：已生成 2*max 个字符
    if (current.length() == max * 2) {
        result.add(current);
        return;
    }

    // 添加左括号（左括号数量小于 max 时）
    if (open < max)
        backtrack(result, current + "(", open + 1, close, max);

    // 添加右括号（右括号数量小于左括号数量时）
    if (close < open)
        backtrack(result, current + ")", open, close + 1, max);
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 3;
    List<String> result1 = generateParenthesis(n1);
    System.out.println("输入: n = " + n1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int n2 = 1;
    List<String> result2 = generateParenthesis(n2);
    System.out.println("\n输入: n = " + n2);
    System.out.println("输出: " + result2);

    // 测试用例 3
    int n3 = 2;
    List<String> result3 = generateParenthesis(n3);
    System.out.println("\n输入: n = " + n3);
```

```
        System.out.println("输出: " + result3);
    }
}
```

=====

文件: Code09\_GenerateParentheses.py

=====

"""

## LeetCode 22. 括号生成

题目描述:

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。

示例:

输入: n = 3

输出: ["((()))","(()())","(())()","()((()))","()()()"]

输入: n = 1

输出: ["()"]

提示:

1 <= n <= 8

链接: <https://leetcode.cn/problems/generate-parentheses/>

"""

```
class Solution:
```

```
    def generateParenthesis(self, n):
```

"""

生成所有可能的有效括号组合

算法思路:

1. 使用回溯算法生成有效括号组合
2. 维护左右括号的数量，确保生成的括号始终有效
3. 左括号数量不能超过 n
4. 右括号数量不能超过左括号数量

时间复杂度:  $O(4^n / \sqrt{n})$ , 第 n 个卡塔兰数

空间复杂度:  $O(4^n / \sqrt{n})$ , 用于存储所有组合

:param n: 括号对数

:return: 所有可能的有效括号组合

```
"""
result = []
self.backtrack(result, "", 0, 0, n)
return result

def backtrack(self, result, current, open_count, close_count, max_count):
    """
回溯函数生成有效括号组合

:param result: 结果列表
:param current: 当前已生成的字符串
:param open_count: 已使用的左括号数量
:param close_count: 已使用的右括号数量
:param max_count: 括号对数
"""

# 终止条件: 已生成 2*max 个字符
if len(current) == max_count * 2:
    result.append(current)
    return

# 添加左括号 (左括号数量小于 max 时)
if open_count < max_count:
    self.backtrack(result, current + "(", open_count + 1, close_count, max_count)

# 添加右括号 (右括号数量小于左括号数量时)
if close_count < open_count:
    self.backtrack(result, current + ")", open_count, close_count + 1, max_count)

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    n1 = 3
    result1 = solution.generateParenthesis(n1)
    print(f"输入: n = {n1}")
    print(f"输出: {result1}")

    # 测试用例 2
    n2 = 1
    result2 = solution.generateParenthesis(n2)
    print(f"\n输入: n = {n2}")
```

```
print(f"输出: {result2}")

# 测试用例 3
n3 = 2
result3 = solution.generateParenthesis(n3)
print(f"\n 输入: n = {n3}")
print(f"输出: {result3}")
```

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

```
=====
文件: Code10_SudokuSolver.cpp
=====
```

```
#include <vector>
#include <iostream>

using namespace std;

/**
 * LeetCode 37. 解数独
 *
 * 题目描述:
 * 编写一个程序，通过填充空格来解决数独问题。
 * 数独的解法需遵循如下规则:
 * 1. 数字 1-9 在每一行只能出现一次。
 * 2. 数字 1-9 在每一列只能出现一次。
 * 3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
 * 空白格用 '.' 表示。
 *
 * 示例:
 * 输入:
 * [
 *   ["5", "3", ".", ".", "7", ".", ".", ".", "."],
 *   ["6", ".", ".", "1", "9", "5", ".", ".", "."],
 *   [".", "9", "8", ".", ".", ".", "6", "."],
 *   [".", "8", "7", "6", "5", "4", "3", "2"],
 *   [".", "4", "3", "8", "7", "6", "5", "1"],
 *   [".", "7", "2", "1", "5", "3", "4", "8"],
 *   [".", "6", "5", "4", "2", "1", "9", "3"],
 *   [".", "1", "9", "3", "8", "7", "2", "6"]
 * ]
```

```

* [".",".",".",".","8",".",".","7","9"]
*
* 输出:
* [
* ["5","3","4","6","7","8","9","1","2"],
* ["6","7","2","1","9","5","3","4","8"],
* ["1","9","8","3","4","2","5","6","7"],
* ["8","5","9","7","6","1","4","2","3"],
* ["4","2","6","8","5","3","7","9","1"],
* ["7","1","3","9","2","4","8","5","6"],
* ["9","6","1","5","3","7","2","8","4"],
* ["2","8","7","4","1","9","6","3","5"],
* ["3","4","5","2","8","6","1","7","9"]
*
* 提示:
* board.length == 9
* board[i].length == 9
* board[i][j] 是一位数字或者 '.'
* 题目数据保证输入数独仅有一个解
*
* 链接: https://leetcode.cn/problems/sudoku-solver/
*/

```

```

class Solution {
public:
    /**
     * 解决数独问题
     *
     * 算法思路:
     * 1. 使用回溯算法解决数独问题
     * 2. 遍历棋盘, 找到第一个空格
     * 3. 尝试填入 1-9 的数字, 检查是否符合数独规则
     * 4. 如果符合规则, 递归解决剩余空格
     * 5. 如果递归返回 false, 说明当前填法不正确, 回溯并尝试下一个数字
     *
     * 时间复杂度: O(9^(n*n)), 最坏情况下每个空格都要尝试 9 个数字
     * 空间复杂度: O(n*n), 递归栈深度
     *
     * @param board 数独棋盘
     */
    void solveSudoku(vector<vector<char>>& board) {
        solve(board);
    }
}
```

```
}
```

```
private:
```

```
/**
```

```
* 回溯函数解决数独
```

```
*
```

```
* @param board 数独棋盘
```

```
* @return 是否成功解决
```

```
*/
```

```
bool solve(vector<vector<char>>& board) {
```

```
    for (int row = 0; row < 9; row++) {
```

```
        for (int col = 0; col < 9; col++) {
```

```
            // 找到空格
```

```
            if (board[row][col] == '.') {
```

```
                // 尝试填入 1-9
```

```
                for (char c = '1'; c <= '9'; c++) {
```

```
                    // 检查是否合法
```

```
                    if (isValid(board, row, col, c)) {
```

```
                        board[row][col] = c;
```

```
                        // 递归求解
```

```
                        if (solve(board))
```

```
                            return true;
```

```
                        else
```

```
                            board[row][col] = '.'; // 回溯
```

```
}
```

```
}
```

```
        return false; // 1-9 都尝试过都不行
```

```
}
```

```
    return true; // 全部填完
```

```
}
```

```
/**
```

```
* 检查在指定位置填入指定数字是否合法
```

```
*
```

```
* @param board 数独棋盘
```

```
* @param row 行索引
```

```
* @param col 列索引
```

```
* @param c 要填入的数字
```

```
* @return 是否合法
```

```
*/
```

```

bool isValid(vector<vector<char>>& board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        // 检查行
        if (board[i][col] == c) return false;
        // 检查列
        if (board[row][i] == c) return false;
        // 检查 3x3 子网格
        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) return false;
    }
    return true;
}

// 打印数独棋盘
void printBoard(const vector<vector<char>>& board) {
    for (int i = 0; i < 9; i++) {
        if (i % 3 == 0 && i != 0) {
            cout << "-----+-----+-----" << endl;
        }
        for (int j = 0; j < 9; j++) {
            if (j % 3 == 0 && j != 0) {
                cout << "| ";
            }
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

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

    // 测试用例
    vector<vector<char>> board = {
        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},
        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},
        {'.', '9', '8', '.', '.', '.', '6', '.'},
        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},
        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},
        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},
        {'.', '6', '.', '.', '.', '2', '8', '.'},
        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},

```

```

{'.','.','.','.','8','.','.','7','9'}
};

cout << "数独题目:" << endl;
printBoard(board);

solution.solveSudoku(board);

cout << "\n 数独解答:" << endl;
printBoard(board);

return 0;
}

```

---

文件: Code10\_SudokuSolver.java

---

```

/**
 * LeetCode 37. 解数独
 *
 * 题目描述:
 * 编写一个程序，通过填充空格来解决数独问题。
 * 数独的解法需遵循如下规则:
 * 1. 数字 1-9 在每一行只能出现一次。
 * 2. 数字 1-9 在每一列只能出现一次。
 * 3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
 * 空白格用 '.' 表示。
 *
 * 示例:
 * 输入:
 * [
 *   ["5","3",".",".","7",".",".",".","."],
 *   ["6",".",".","1","9","5",".",".","."],
 *   [".","9","8",".",".",".","6","."],
 *   ["8",".",".",".","6",".",".","3"],
 *   ["4",".",".","8",".","3",".",".","1"],
 *   ["7",".",".",".","2",".",".",".","6"],
 *   [".","6",".",".",".","2","8","."],
 *   [".",".",".","4","1","9",".","5"],
 *   [".",".",".","8",".",".","7","9"]
 * ]
 * 输出:

```

```
* [
*   ["5", "3", "4", "6", "7", "8", "9", "1", "2"],
*   ["6", "7", "2", "1", "9", "5", "3", "4", "8"],
*   ["1", "9", "8", "3", "4", "2", "5", "6", "7"],
*   ["8", "5", "9", "7", "6", "1", "4", "2", "3"],
*   ["4", "2", "6", "8", "5", "3", "7", "9", "1"],
*   ["7", "1", "3", "9", "2", "4", "8", "5", "6"],
*   ["9", "6", "1", "5", "3", "7", "2", "8", "4"],
*   ["2", "8", "7", "4", "1", "9", "6", "3", "5"],
*   ["3", "4", "5", "2", "8", "6", "1", "7", "9"]
* ]
*
* 提示:
* board.length == 9
* board[i].length == 9
* board[i][j] 是一位数字或者 '.'
* 题目数据保证输入数独仅有一个解
*
* 链接: https://leetcode.cn/problems/sudoku-solver/
*
* 算法思路:
* 1. 使用回溯算法解决数独问题
* 2. 遍历棋盘, 找到第一个空格
* 3. 尝试填入 1-9 的数字, 检查是否符合数独规则
* 4. 如果符合规则, 递归解决剩余空格
* 5. 如果递归返回 false, 说明当前填法不正确, 回溯并尝试下一个数字
*
* 剪枝策略:
* 1. 可行性剪枝: 在填入数字前检查是否符合数独规则
* 2. 约束传播: 一旦某个位置填入数字, 立即更新约束条件
* 3. 提前终止: 当发现冲突时立即回溯
*
* 时间复杂度: O(9^(n*n)), 最坏情况下每个空格都要尝试 9 个数字
* 空间复杂度: O(n*n), 递归栈深度
*
* 工程化考量:
* 1. 边界处理: 处理空棋盘和已填满棋盘的情况
* 2. 性能优化: 通过剪枝减少不必要的计算
* 3. 内存管理: 合理使用数据结构减少内存占用
* 4. 可读性: 添加详细注释和变量命名
* 5. 异常处理: 处理可能的异常情况
* 6. 模块化设计: 将核心逻辑封装成独立方法
* 7. 可维护性: 添加详细注释和文档说明
```

```

*/
public class Code10_SudokuSolver {

    /**
     * 解决数独问题
     *
     * @param board 数独棋盘
     */
    public static void solveSudoku(char[][] board) {
        // 边界条件检查
        if (board == null || board.length != 9 || board[0].length != 9) {
            throw new IllegalArgumentException("Invalid board size");
        }

        solve(board);
    }

    /**
     * 回溯函数解决数独
     *
     * @param board 数独棋盘
     * @return 是否成功解决
     */
    private static boolean solve(char[][] board) {
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                // 找到空格
                if (board[row][col] == '.') {
                    // 尝试填入 1-9
                    for (char c = '1'; c <= '9'; c++) {
                        // 可行性剪枝：检查是否合法
                        if (isValid(board, row, col, c)) {
                            board[row][col] = c;

                            // 递归求解
                            if (solve(board))
                                return true;
                            else
                                board[row][col] = '.'; // 回溯
                        }
                    }
                }
            }
        }
        return false; // 1-9 都尝试过都不行
    }
}

```

```

        }
    }

    return true; // 全部填完
}

/***
 * 检查在指定位置填入指定数字是否合法
 *
 * @param board 数独棋盘
 * @param row 行索引
 * @param col 列索引
 * @param c 要填入的数字
 * @return 是否合法
 */
private static boolean isValid(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        // 检查行
        if (board[i][col] == c) return false;
        // 检查列
        if (board[row][i] == c) return false;
        // 检查 3x3 子网格
        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) return false;
    }
    return true;
}

/***
 * 优化版本：使用位运算优化的数独求解器
 *
 * @param board 数独棋盘
 */
public static void solveSudokuOptimized(char[][] board) {
    // 边界条件检查
    if (board == null || board.length != 9 || board[0].length != 9) {
        throw new IllegalArgumentException("Invalid board size");
    }

    // 使用位运算优化
    int[] rows = new int[9];
    int[] cols = new int[9];
    int[][] boxes = new int[3][3];

    // 初始化约束条件

```

```

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.') {
                    int digit = board[i][j] - '1';
                    rows[i] |= (1 << digit);
                    cols[j] |= (1 << digit);
                    boxes[i/3][j/3] |= (1 << digit);
                }
            }
        }

        solveOptimized(board, rows, cols, boxes, 0, 0);
    }

/***
 * 优化版本的回溯函数
 *
 * @param board 数独棋盘
 * @param rows 行约束
 * @param cols 列约束
 * @param boxes 3x3 盒子约束
 * @param row 当前行
 * @param col 当前列
 * @return 是否成功解决
 */
private static boolean solveOptimized(char[][] board, int[] rows, int[] cols, int[][] boxes,
int row, int col) {
    // 找到下一个空格
    while (row < 9 && board[row][col] != '.') {
        col++;
        if (col == 9) {
            col = 0;
            row++;
        }
    }

    // 终止条件：已处理完所有格子
    if (row == 9) {
        return true;
    }

    // 计算可用数字
    int boxRow = row / 3;

```

```

int boxCol = col / 3;
int used = rows[row] | cols[col] | boxes[boxRow][boxCol];

// 尝试填入可用数字
for (int digit = 0; digit < 9; digit++) {
    if ((used & (1 << digit)) == 0) { // 数字未被使用
        // 填入数字
        board[row][col] = (char)('1' + digit);
        rows[row] |= (1 << digit);
        cols[col] |= (1 << digit);
        boxes[boxRow][boxCol] |= (1 << digit);

        // 递归求解
        if (solveOptimized(board, rows, cols, boxes, row, col)) {
            return true;
        }
    }
}

// 回溯
board[row][col] = '.';
rows[row] &= ~(1 << digit);
cols[col] &= ~(1 << digit);
boxes[boxRow][boxCol] &= ~(1 << digit);
}

}

return false;
}

/***
 * 打印数独棋盘
 *
 * @param board 数独棋盘
 */
public static void printBoard(char[][] board) {
    System.out.println("Current Board:");
    for (int i = 0; i < 9; i++) {
        if (i % 3 == 0 && i != 0) {
            System.out.println("-----+-----+-----");
        }
        for (int j = 0; j < 9; j++) {
            if (j % 3 == 0 && j != 0) {
                System.out.print(" | ");
            }

```

```

        System.out.print(board[i][j] + " ");
    }
    System.out.println();
}
System.out.println();
}

/**
 * 验证数独解是否正确
 *
 * @param board 数独棋盘
 * @return 是否正确
 */
public static boolean isValidSolution(char[][] board) {
    // 检查行
    for (int i = 0; i < 9; i++) {
        boolean[] used = new boolean[9];
        for (int j = 0; j < 9; j++) {
            if (board[i][j] < '1' || board[i][j] > '9') return false;
            int digit = board[i][j] - '1';
            if (used[digit]) return false;
            used[digit] = true;
        }
    }

    // 检查列
    for (int j = 0; j < 9; j++) {
        boolean[] used = new boolean[9];
        for (int i = 0; i < 9; i++) {
            int digit = board[i][j] - '1';
            if (used[digit]) return false;
            used[digit] = true;
        }
    }

    // 检查 3x3 子网格
    for (int boxRow = 0; boxRow < 3; boxRow++) {
        for (int boxCol = 0; boxCol < 3; boxCol++) {
            boolean[] used = new boolean[9];
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    int row = boxRow * 3 + i;
                    int col = boxCol * 3 + j;
                    if (board[row][col] < '1' || board[row][col] > '9') return false;
                    int digit = board[row][col] - '1';
                    if (used[digit]) return false;
                    used[digit] = true;
                }
            }
        }
    }
}

```



```

{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'},
{'.','.','.','.','.','.','.','.','.'}
};

System.out.println("\n==== 测试用例 2 (空棋盘) ===");
System.out.println("数独题目:");
printBoard(board2);

solveSudokuOptimized(board2);

System.out.println("数独解答:");
printBoard(board2);

System.out.println("解是否正确: " + isValidSolution(board2));
}
}
=====

文件: Code10_SudokuSolver.py
=====
"""
LeetCode 37. 解数独

题目描述:
编写一个程序，通过填充空格来解决数独问题。
数独的解法需遵循如下规则:
1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
空白格用 '.' 表示。
示例:
输入:
[
["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],
[".","9","8",".",".",".","6",".","."]
]

```

示例:

输入:

[

```

["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],
[".","9","8",".",".",".","6",".","."]
]
```

```
[["8",".",".",".","6",".",".",".","3"],  
 ["4",".",".","8",".","3",".",".","1"],  
 ["7",".",".",".","2",".",".",".","6"],  
 [".","6",".",".",".","2","8","."],  
 [".",".",".","4","1","9",".",".","5"],  
 [".",".",".","8",".","7","7","9"]]
```

]

输出:

```
[  
 ["5","3","4","6","7","8","9","1","2"],  
 ["6","7","2","1","9","5","3","4","8"],  
 ["1","9","8","3","4","2","5","6","7"],  
 ["8","5","9","7","6","1","4","2","3"],  
 ["4","2","6","8","5","3","7","9","1"],  
 ["7","1","3","9","2","4","8","5","6"],  
 ["9","6","1","5","3","7","2","8","4"],  
 ["2","8","7","4","1","9","6","3","5"],  
 ["3","4","5","2","8","6","1","7","9"]]
```

]

提示:

board.length == 9

board[i].length == 9

board[i][j] 是一位数字或者 '.'

题目数据保证输入数独仅有一个解

链接: <https://leetcode.cn/problems/sudoku-solver/>

""""

```
class Solution:
```

```
    def solveSudoku(self, board):
```

```
        """
```

解决数独问题

算法思路:

1. 使用回溯算法解决数独问题
2. 遍历棋盘，找到第一个空格
3. 尝试填入 1-9 的数字，检查是否符合数独规则
4. 如果符合规则，递归解决剩余空格
5. 如果递归返回 false，说明当前填法不正确，回溯并尝试下一个数字

时间复杂度:  $O(9^{n \times n})$ ，最坏情况下每个空格都要尝试 9 个数字

空间复杂度:  $O(n \times n)$ ，递归栈深度

```

:param board: 数独棋盘
"""
self.solve(board)

def solve(self, board):
    """
回溯函数解决数独

:param board: 数独棋盘
:return: 是否成功解决
"""

for row in range(9):
    for col in range(9):
        # 找到空格
        if board[row][col] == '.':
            # 尝试填入 1-9
            for c in map(str, range(1, 10)):
                # 检查是否合法
                if self.isValid(board, row, col, c):
                    board[row][col] = c

                    # 递归求解
                    if self.solve(board):
                        return True
                    else:
                        board[row][col] = '.' # 回溯
            return False # 1-9 都尝试过都不行
    return True # 全部填完

def isValid(self, board, row, col, c):
    """
检查在指定位置填入指定数字是否合法

:param board: 数独棋盘
:param row: 行索引
:param col: 列索引
:param c: 要填入的数字
:return: 是否合法
"""

for i in range(9):
    # 检查行
    if board[i][col] == c:

```

```

        return False
    # 检查列
    if board[row][i] == c:
        return False
    # 检查 3x3 子网格
    if board[3 * (row // 3) + i // 3][3 * (col // 3) + i % 3] == c:
        return False
    return True

# 打印数独棋盘
def printBoard(board):
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("-----+-----+-----")
        for j in range(9):
            if j % 3 == 0 and j != 0:
                print("| ", end="")
            print(board[i][j] + " ", end="")
        print()

# 测试方法
def main():
    solution = Solution()

    # 测试用例
    board = [
        ['5', '3', '.', '.', '7', '.', '.', '.', '.'],
        ['6', '.', '.', '1', '9', '5', '.', '.', '.'],
        ['.', '9', '8', '.', '.', '.', '6', '.'],
        ['8', '.', '.', '.', '6', '.', '.', '.', '3'],
        ['4', '.', '.', '8', '.', '3', '.', '.', '1'],
        ['7', '.', '.', '.', '2', '.', '.', '.', '6'],
        ['.', '6', '.', '.', '.', '.', '2', '8', '.'],
        ['.', '.', '.', '.', '4', '1', '9', '.', '.'],
        ['.', '.', '.', '.', '8', '.', '7', '9']
    ]

    print("数独题目:")
    printBoard(board)

    solution.solveSudoku(board)

```

```
print("\n 数独解答:")
printBoard(board)
```

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

=====

文件: Code11\_NQueens.cpp

=====

```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

/***
 * LeetCode 51. N 皇后
 *
 * 题目描述:
 * n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
 * 给你一个整数 n，返回所有不同的 n 皇后问题 的解决方案。
 * 每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。
 *
 * 示例:
 * 输入: n = 4
 * 输出: [["Q..", "...Q", "Q...", "..Q."], [".Q...", "Q...", "...Q", ".Q.."]]
 *
 * 输入: n = 1
 * 输出: [["Q"]]
 *
 * 提示:
 * 1 <= n <= 9
 *
 * 链接: https://leetcode.cn/problems/n-queens/
 */
```

```
class Solution {
public:
    /**
     * 解决 N 皇后问题
```

```

*
* 算法思路:
* 1. 使用回溯算法解决 N 皇后问题
* 2. 按行放置皇后, 每行放置一个
* 3. 对于每一行, 尝试在每一列放置皇后
* 4. 检查当前位置是否与已放置的皇后冲突
* 5. 如果不冲突, 递归处理下一行
* 6. 如果冲突, 尝试下一列
* 7. 如果所有列都尝试过都不行, 回溯到上一行
*
* 时间复杂度:  $O(N!)$ , 第一行有  $N$  种选择, 第二行最多有  $N-1$  种选择, 以此类推
* 空间复杂度:  $O(N^2)$ , 棋盘空间和递归栈深度
*
* @param n 皇后数量和棋盘大小
* @return 所有解决方案
*/
vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> result;
    vector<string> board(n, string(n, '.'));

    backtrack(result, board, 0);
    return result;
}

private:
    /**
     * 回溯函数解决 N 皇后问题
     *
     * @param result 结果列表
     * @param board 棋盘
     * @param row 当前行
     */
    void backtrack(vector<vector<string>>& result, vector<string>& board, int row) {
        // 终止条件: 已放置完所有皇后
        if (row == board.size()) {
            result.push_back(board);
            return;
        }

        // 在当前行的每一列尝试放置皇后
        for (int col = 0; col < board.size(); col++) {
            if (isValid(board, row, col)) {
                board[row][col] = 'Q';
                backtrack(result, board, row + 1);
                board[row][col] = '.';
            }
        }
    }
}

```

```

        backtrack(result, board, row + 1);
        board[row][col] = '.';
    }
}

/***
 * 检查在指定位置放置皇后是否合法
 *
 * @param board 棋盘
 * @param row 行索引
 * @param col 列索引
 * @return 是否合法
 */
bool isValid(vector<string>& board, int row, int col) {
    // 检查列
    for (int i = 0; i < row; i++)
        if (board[i][col] == 'Q') return false;

    // 检查左上对角线
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q') return false;

    // 检查右上对角线
    for (int i = row - 1, j = col + 1; i >= 0 && j < board.size(); i--, j++)
        if (board[i][j] == 'Q') return false;

    return true;
};

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

    // 测试用例 1
    int n1 = 4;
    vector<vector<string>> result1 = solution.solveNQueens(n1);
    cout << "输入: n = " << n1 << endl;
    cout << "输出:" << endl;
    for (const auto& solution_board : result1) {
        for (const auto& row : solution_board) {
            cout << row << endl;
        }
    }
}

```

```

    }

    cout << endl;
}

// 测试用例 2
int n2 = 1;
vector<vector<string>> result2 = solution.solveNQueens(n2);
cout << "输入: n = " << n2 << endl;
cout << "输出:" << endl;
for (const auto& solution_board : result2) {
    for (const auto& row : solution_board) {
        cout << row << endl;
    }
    cout << endl;
}

return 0;
}

```

=====

文件: Code11\_NQueens.java

```

=====
import java.util.*;

/**
 * LeetCode 51. N 皇后
 *
 * 题目描述:
 * n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
 * 给你一个整数 n，返回所有不同的 n 皇后问题 的解决方案。
 * 每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。
 *
 * 示例:
 * 输入: n = 4
 * 输出: [["Q..", "...Q", "Q..."], [".Q.", "...Q", "...Q"]]
 *
 * 输入: n = 1
 * 输出: [["Q"]]
 *
 * 提示:
 * 1 <= n <= 9
 */

```

```

* 链接: https://leetcode.cn/problems/n-queens/
*
* 算法思路:
* 1. 使用回溯算法解决 N 皇后问题
* 2. 按行放置皇后, 每行放置一个
* 3. 对于每一行, 尝试在每一列放置皇后
* 4. 检查当前位置是否与已放置的皇后冲突
* 5. 如果不冲突, 递归处理下一行
* 6. 如果冲突, 尝试下一列
* 7. 如果所有列都尝试过都不行, 回溯到上一行
*
* 剪枝策略:
* 1. 可行性剪枝: 在放置皇后前检查是否与已放置的皇后冲突
* 2. 约束传播: 一旦某行无法放置皇后, 立即回溯
* 3. 提前终止: 当发现冲突时立即停止当前路径的探索
*
* 时间复杂度: O(N!), 第一行有 N 种选择, 第二行最多有 N-1 种选择, 以此类推
* 空间复杂度: O(N^2), 棋盘空间和递归栈深度
*
* 工程化考量:
* 1. 边界处理: 处理 n=1 的特殊情况
* 2. 性能优化: 使用位运算可以进一步优化冲突检查
* 3. 内存管理: 合理使用数据结构减少内存占用
* 4. 可读性: 添加详细注释和变量命名
* 5. 异常处理: 验证输入参数的有效性
* 6. 模块化设计: 将核心逻辑封装成独立方法
* 7. 可维护性: 添加详细注释和文档说明
*/
public class Code11_NQueens {

    /**
     * 解决 N 皇后问题
     *
     * @param n 皇后数量和棋盘大小
     * @return 所有解决方案
     */
    public static List<List<String>> solveNQueens(int n) {
        // 边界条件检查
        if (n <= 0) {
            throw new IllegalArgumentException("n must be positive");
        }

        List<List<String>> result = new ArrayList<>();

```

```

char[][] board = new char[n][n];

// 初始化棋盘
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        board[i][j] = '.';

backtrack(result, board, 0);
return result;
}

/***
 * 回溯函数解决 N 皇后问题
 *
 * @param result 结果列表
 * @param board 棋盘
 * @param row 当前行
 */
private static void backtrack(List<List<String>> result, char[][] board, int row) {
    // 终止条件：已放置完所有皇后
    if (row == board.length) {
        result.add(construct(board));
        return;
    }

    // 在当前行的每一列尝试放置皇后
    for (int col = 0; col < board.length; col++) {
        // 可行性剪枝：检查是否与已放置的皇后冲突
        if (isValid(board, row, col)) {
            board[row][col] = 'Q';
            backtrack(result, board, row + 1);
            board[row][col] = '.'; // 回溯
        }
    }
}

/***
 * 检查在指定位置放置皇后是否合法
 *
 * @param board 棋盘
 * @param row 行索引
 * @param col 列索引
 * @return 是否合法

```

```

*/
private static boolean isValid(char[][] board, int row, int col) {
    // 检查列
    for (int i = 0; i < row; i++)
        if (board[i][col] == 'Q') return false;

    // 检查左上对角线
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q') return false;

    // 检查右上对角线
    for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++)
        if (board[i][j] == 'Q') return false;

    return true;
}

/**
 * 优化版本：使用位运算优化的 N 皇后问题解法
 *
 * @param n 皇后数量和棋盘大小
 * @return 所有解决方案
 */
public static List<List<String>> solveNQueensOptimized(int n) {
    // 边界条件检查
    if (n <= 0) {
        throw new IllegalArgumentException("n must be positive");
    }

    List<List<String>> result = new ArrayList<>();
    int[] queens = new int[n];

    // 使用位运算表示列、主对角线、副对角线的占用情况
    backtrackOptimized(n, 0, 0, 0, 0, queens, result);
    return result;
}

/**
 * 使用位运算的回溯函数
 *
 * @param n 棋盘大小
 * @param row 当前行
 * @param cols 列占用情况（二进制位表示）

```

```

* @param diag1 主对角线占用情况
* @param diag2 副对角线占用情况
* @param queens 皇后位置数组
* @param result 结果列表
*/
private static void backtrackOptimized(int n, int row, int cols, int diag1, int diag2, int[]
queens, List<List<String>> result) {
    // 终止条件：已放置完所有皇后
    if (row == n) {
        result.add(constructFromQueens(queens, n));
        return;
    }

    // 计算可以放置皇后的位置
    // ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ ) 表示不与任何皇后冲突的位置
    // (( $1 \ll n$ ) - 1) 用于限制在 n 位范围内
    int availablePositions = ((1 << n) - 1) & ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ );

    // 遍历所有可以放置皇后的位置
    while (availablePositions != 0) {
        // 获取最右边的可用位置
        int position = availablePositions & ( $\sim\text{availablePositions}$ );

        // 记录皇后位置
        queens[row] = Integer.numberOfTrailingZeros(position);

        // 在该位置放置皇后
        backtrackOptimized(n, row + 1,
                           cols | position,
                           (diag1 | position) << 1,
                           (diag2 | position) >> 1,
                           queens, result);

        // 移除已处理的位置
        availablePositions &= availablePositions - 1;
    }
}

/**
 * 从皇后位置数组构造解决方案
 *
 * @param queens 皇后位置数组
 * @param n 棋盘大小

```

```

 * @return 解决方案字符串列表
 */
private static List<String> constructFromQueens(int[] queens, int n) {
    List<String> result = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        char[] row = new char[n];
        for (int j = 0; j < n; j++) {
            row[j] = '.';
        }
        row[queens[i]] = 'Q';
        result.add(new String(row));
    }
    return result;
}

/***
 * 构造解决方案字符串
 *
 * @param board 棋盘
 * @return 解决方案字符串列表
 */
private static List<String> construct(char[][] board) {
    List<String> result = new ArrayList<>();
    for (int i = 0; i < board.length; i++)
        result.add(new String(board[i]));
    return result;
}

/***
 * 验证解决方案是否正确
 *
 * @param solution 解决方案
 * @return 是否正确
 */
public static boolean isValidSolution(List<String> solution) {
    if (solution == null || solution.isEmpty()) return false;

    int n = solution.size();
    // 检查每行是否只有一个皇后
    for (String row : solution) {
        if (row.length() != n) return false;
        int queenCount = 0;
        for (char c : row.toCharArray()) {

```

```

        if (c == 'Q') queenCount++;
        else if (c != '.') return false;
    }
    if (queenCount != 1) return false;
}

// 检查皇后是否相互攻击
int[] queens = new int[n];
for (int i = 0; i < n; i++) {
    queens[i] = solution.get(i).indexOf('Q');
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        // 检查列冲突
        if (queens[i] == queens[j]) return false;
        // 检查对角线冲突
        if (Math.abs(queens[i] - queens[j]) == Math.abs(i - j)) return false;
    }
}

return true;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 4;
    List<List<String>> result1 = solveNQueens(n1);
    System.out.println("==> 测试用例 1 (n = " + n1 + ") ==>");
    System.out.println("输出:");
    for (List<String> solution : result1) {
        for (String row : solution) {
            System.out.println(row);
        }
        System.out.println("解是否正确: " + isValidSolution(solution));
        System.out.println();
    }

    // 测试用例 2
    int n2 = 1;
}

```

```

List<List<String>> result2 = solveNQueens(n2);
System.out.println("==> 测试用例 2 (n = " + n2 + ") ==>");
System.out.println("输出:");
for (List<String> solution : result2) {
    for (String row : solution) {
        System.out.println(row);
    }
    System.out.println("解是否正确: " + isValidSolution(solution));
    System.out.println();
}

// 测试用例 3: 优化版本
int n3 = 8;
long startTime = System.currentTimeMillis();
List<List<String>> result3 = solveNQueens(n3);
long endTime = System.currentTimeMillis();
System.out.println("==> 测试用例 3 (n = " + n3 + ", 基础版本) ==>");
System.out.println("解的数量: " + result3.size());
System.out.println("耗时: " + (endTime - startTime) + " ms");

startTime = System.currentTimeMillis();
List<List<String>> result4 = solveNQueensOptimized(n3);
endTime = System.currentTimeMillis();
System.out.println("==> 测试用例 3 (n = " + n3 + ", 优化版本) ==>");
System.out.println("解的数量: " + result4.size());
System.out.println("耗时: " + (endTime - startTime) + " ms");
}
}

```

文件: Code11\_NQueens.py

=====  
'''

LeetCode 51. N 皇后

题目描述:

n 皇后问题研究的是如何将 n 个皇后放置在  $n \times n$  的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

示例:

输入: n = 4

```
输出: [[".Q..", "...Q", "Q...", "..Q."], [".Q.", "Q...", "...Q", ".Q.."]]
```

输入: n = 1

```
输出: [["Q"]]
```

提示:

```
1 <= n <= 9
```

链接: <https://leetcode.cn/problems/n-queens/>

"""

```
class Solution:
```

```
    def solveNQueens(self, n):
```

"""

```
        解决 N 皇后问题
```

算法思路:

1. 使用回溯算法解决 N 皇后问题
2. 按行放置皇后，每行放置一个
3. 对于每一行，尝试在每一列放置皇后
4. 检查当前位置是否与已放置的皇后冲突
5. 如果不冲突，递归处理下一行
6. 如果冲突，尝试下一列
7. 如果所有列都尝试过都不行，回溯到上一行

时间复杂度:  $O(N!)$ ，第一行有  $N$  种选择，第二行最多有  $N-1$  种选择，以此类推

空间复杂度:  $O(N^2)$ ，棋盘空间和递归栈深度

```
:param n: 皇后数量和棋盘大小
```

```
:return: 所有解决方案
```

"""

```
result = []
```

```
board = [['.' for _ in range(n)] for _ in range(n)]
```

```
self.backtrack(result, board, 0)
```

```
return result
```

```
def backtrack(self, result, board, row):
```

"""

```
    回溯函数解决 N 皇后问题
```

```
:param result: 结果列表
```

```
:param board: 棋盘
```

```

:param row: 当前行
"""

# 终止条件：已放置完所有皇后
if row == len(board):
    # 将棋盘转换为字符串列表
    solution = ['.join(row) for row in board]
    result.append(solution)
    return

# 在当前行的每一列尝试放置皇后
for col in range(len(board)):
    if self.isValid(board, row, col):
        board[row][col] = 'Q'
        self.backtrack(result, board, row + 1)
        board[row][col] = '.' # 回溯

def isValid(self, board, row, col):
    """

检查在指定位置放置皇后是否合法

:param board: 棋盘
:param row: 行索引
:param col: 列索引
:return: 是否合法
"""

# 检查列
for i in range(row):
    if board[i][col] == 'Q':
        return False

# 检查左上对角线
i, j = row - 1, col - 1
while i >= 0 and j >= 0:
    if board[i][j] == 'Q':
        return False
    i -= 1
    j -= 1

# 检查右上对角线
i, j = row - 1, col + 1
while i >= 0 and j < len(board):
    if board[i][j] == 'Q':
        return False
    i -= 1
    j += 1

```

```
i -= 1
j += 1

return True

# 测试方法
def main():
    solution = Solution()

# 测试用例 1
n1 = 4
result1 = solution.solveNQueens(n1)
print(f"输入: n = {n1}")
print("输出:")
for solution_board in result1:
    for row in solution_board:
        print(row)
    print()

# 测试用例 2
n2 = 1
result2 = solution.solveNQueens(n2)
print(f"输入: n = {n2}")
print("输出:")
for solution_board in result2:
    for row in solution_board:
        print(row)
    print()

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

文件: Code12\_TargetSum.cpp

```
#include <vector>
#include <iostream>

using namespace std;
```

```
/**  
 * LeetCode 494. 目标和  
 *  
 * 题目描述:  
 * 给你一个非负整数数组 nums 和一个整数 target 。  
 * 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个表达式 。  
 * 返回可以通过上述方法构造的、运算结果等于 target 的不同表达式的数目。  
 *  
 * 示例:  
 * 输入: nums = [1, 1, 1, 1, 1], target = 3  
 * 输出: 5  
 *  
 * 输入: nums = [1], target = 1  
 * 输出: 1  
 *  
 * 提示:  
 * 1 <= nums.length <= 20  
 * 0 <= nums[i] <= 1000  
 * 0 <= sum(nums[i]) <= 1000  
 * -1000 <= target <= 1000  
 *  
 * 链接: https://leetcode.cn/problems/target-sum/  
 */
```

```
class Solution {  
public:  
    /**  
     * 计算目标和的表达式数目（回溯算法）  
     *  
     * 算法思路:  
     * 1. 使用回溯算法遍历所有可能的符号组合  
     * 2. 对于每个数字，有两种选择：加号或减号  
     * 3. 递归处理下一个数字  
     * 4. 当处理完所有数字时，检查结果是否等于目标值  
     *  
     * 时间复杂度: O(2^n)  
     * 空间复杂度: O(n)  
     *  
     * @param nums 数组  
     * @param target 目标值  
     * @return 表达式数目  
    */  
    int findTargetSumWays(vector<int>& nums, int target) {
```

```

        return backtrack(nums, target, 0, 0);
    }

private:
    /**
     * 回溯函数计算目标和的表达式数目
     *
     * @param nums 数组
     * @param target 目标值
     * @param index 当前处理的索引
     * @param sum 当前和
     * @return 表达式数目
     */
    int backtrack(vector<int>& nums, int target, int index, int sum) {
        // 终止条件：已处理完所有数字
        if (index == nums.size()) {
            return sum == target ? 1 : 0;
        }

        // 选择加号
        int add = backtrack(nums, target, index + 1, sum + nums[index]);
        // 选择减号
        int subtract = backtrack(nums, target, index + 1, sum - nums[index]);

        return add + subtract;
    }
};

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

    // 测试用例 1
    vector<int> nums1 = {1, 1, 1, 1, 1};
    int target1 = 3;
    int result1 = solution.findTargetSumWays(nums1, target1);
    cout << "输入: nums = [1,1,1,1,1], target = " << target1 << endl;
    cout << "输出: " << result1 << endl;

    // 测试用例 2
    vector<int> nums2 = {1};
    int target2 = 1;
    int result2 = solution.findTargetSumWays(nums2, target2);
}

```

```

cout << "\n 输入: nums = [1], target = " << target2 << endl;
cout << "输出: " << result2 << endl;

// 测试用例 3
vector<int> nums3 = {1, 0};
int target3 = 1;
int result3 = solution.findTargetSumWays(nums3, target3);
cout << "\n 输入: nums = [1, 0], target = " << target3 << endl;
cout << "输出: " << result3 << endl;

return 0;
}
=====
```

文件: Code12\_TargetSum.java

```

=====
package class038;

/**
 * LeetCode 494. 目标和
 *
 * 题目描述:
 * 给你一个非负整数数组 nums 和一个整数 target 。
 * 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个表达式 。
 * 返回可以通过上述方法构造的、运算结果等于 target 的不同表达式的数目。
 *
 * 示例:
 * 输入: nums = [1,1,1,1,1], target = 3
 * 输出: 5
 *
 * 输入: nums = [1], target = 1
 * 输出: 1
 *
 * 提示:
 * 1 <= nums.length <= 20
 * 0 <= nums[i] <= 1000
 * 0 <= sum(nums[i]) <= 1000
 * -1000 <= target <= 1000
 *
 * 链接: https://leetcode.cn/problems/target-sum/
 */
public class Code12_TargetSum {
```

```

/**
 * 计算目标和的表达式数目（回溯算法）
 *
 * 算法思路：
 * 1. 使用回溯算法遍历所有可能的符号组合
 * 2. 对于每个数字，有两种选择：加号或减号
 * 3. 递归处理下一个数字
 * 4. 当处理完所有数字时，检查结果是否等于目标值
 *
 * 时间复杂度：O(2^n)
 * 空间复杂度：O(n)
 *
 * @param nums 数组
 * @param target 目标值
 * @return 表达式数目
 */
public static int findTargetSumWays(int[] nums, int target) {
    return backtrack(nums, target, 0, 0);
}

/**
 * 回溯函数计算目标和的表达式数目
 *
 * @param nums 数组
 * @param target 目标值
 * @param index 当前处理的索引
 * @param sum 当前和
 * @return 表达式数目
 */
private static int backtrack(int[] nums, int target, int index, int sum) {
    // 终止条件：已处理完所有数字
    if (index == nums.length) {
        return sum == target ? 1 : 0;
    }

    // 选择加号
    int add = backtrack(nums, target, index + 1, sum + nums[index]);
    // 选择减号
    int subtract = backtrack(nums, target, index + 1, sum - nums[index]);

    return add + subtract;
}

```

```
/**  
 * 计算目标和的表达式数目（动态规划优化）  
 *  
 * 算法思路：  
 * 1. 将问题转换为子集和问题  
 * 2. 假设正数集合和为 P，负数集合和为 N，则 P-N=target, P+N=sum  
 * 3. 联立得 P=(target+sum)/2，问题转化为找出和为 P 的子集数目  
 * 4. 使用动态规划求解子集和问题  
 *  
 * 时间复杂度：O(n * sum)  
 * 空间复杂度：O(sum)  
 *  
 * @param nums 数组  
 * @param target 目标值  
 * @return 表达式数目  
 */
```

```
public static int findTargetSumWaysDP(int[] nums, int target) {  
    int sum = 0;  
    for (int num : nums) sum += num;  
  
    // 边界情况  
    if (sum < Math.abs(target) || (sum + target) % 2 != 0) return 0;  
  
    // 转换为子集和问题  
    int P = (sum + target) / 2;  
    int[] dp = new int[P + 1];  
    dp[0] = 1;  
  
    for (int num : nums) {  
        for (int i = P; i >= num; i--) {  
            dp[i] += dp[i - num];  
        }  
    }  
  
    return dp[P];  
}
```

```
// 测试方法  
public static void main(String[] args) {  
    // 测试用例 1  
    int[] nums1 = {1, 1, 1, 1, 1};  
    int target1 = 3;
```

```

int result1 = findTargetSumWays(nums1, target1);
int result1DP = findTargetSumWaysDP(nums1, target1);
System.out.println("输入: nums = [1,1,1,1,1], target = " + target1);
System.out.println("输出 (回溯): " + result1);
System.out.println("输出 (动态规划): " + result1DP);

// 测试用例 2
int[] nums2 = {1};
int target2 = 1;
int result2 = findTargetSumWays(nums2, target2);
int result2DP = findTargetSumWaysDP(nums2, target2);
System.out.println("\n输入: nums = [1], target = " + target2);
System.out.println("输出 (回溯): " + result2);
System.out.println("输出 (动态规划): " + result2DP);

// 测试用例 3
int[] nums3 = {1, 0};
int target3 = 1;
int result3 = findTargetSumWays(nums3, target3);
int result3DP = findTargetSumWaysDP(nums3, target3);
System.out.println("\n输入: nums = [1,0], target = " + target3);
System.out.println("输出 (回溯): " + result3);
System.out.println("输出 (动态规划): " + result3DP);
}
}
=====

文件: Code12_TargetSum.py
=====
"""
LeetCode 494. 目标和

```

#### 题目描述:

给你一个非负整数数组 `nums` 和一个整数 `target`。

向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个表达式。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同表达式的数目。

#### 示例:

输入: `nums = [1,1,1,1,1]`, `target = 3`

输出: 5

输入: `nums = [1]`, `target = 1`

输出: 1

提示:

```
1 <= nums.length <= 20
0 <= nums[i] <= 1000
0 <= sum(nums[i]) <= 1000
-1000 <= target <= 1000
```

链接: <https://leetcode.cn/problems/target-sum/>

"""

```
class Solution:
    def findTargetSumWays(self, nums, target):
        """
        计算目标和的表达式数目（回溯算法）
        
```

算法思路:

1. 使用回溯算法遍历所有可能的符号组合
2. 对于每个数字，有两种选择：加号或减号
3. 递归处理下一个数字
4. 当处理完所有数字时，检查结果是否等于目标值

时间复杂度:  $O(2^n)$

空间复杂度:  $O(n)$

```
:param nums: 数组
:param target: 目标值
:return: 表达式数目
"""
return self.backtrack(nums, target, 0, 0)
```

```
def backtrack(self, nums, target, index, sum_val):
    """
    
```

回溯函数计算目标和的表达式数目

```
:param nums: 数组
:param target: 目标值
:param index: 当前处理的索引
:param sum_val: 当前和
:return: 表达式数目
"""
# 终止条件: 已处理完所有数字
if index == len(nums):
```

```
        return 1 if sum_val == target else 0

    # 选择加号
    add = self.backtrack(nums, target, index + 1, sum_val + nums[index])
    # 选择减号
    subtract = self.backtrack(nums, target, index + 1, sum_val - nums[index])

    return add + subtract

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 1, 1, 1, 1]
    target1 = 3
    result1 = solution.findTargetSumWays(nums1, target1)
    print(f"输入: nums = {nums1}, target = {target1}")
    print(f"输出: {result1}")

    # 测试用例 2
    nums2 = [1]
    target2 = 1
    result2 = solution.findTargetSumWays(nums2, target2)
    print(f"\n输入: nums = {nums2}, target = {target2}")
    print(f"输出: {result2}")

    # 测试用例 3
    nums3 = [1, 0]
    target3 = 1
    result3 = solution.findTargetSumWays(nums3, target3)
    print(f"\n输入: nums = {nums3}, target = {target3}")
    print(f"输出: {result3}")

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

文件: Code13\_WordSearch.cpp

```
=====
```

```
#include <vector>
#include <iostream>

using namespace std;

/***
 * LeetCode 79. 单词搜索
 *
 * 题目描述:
 * 给定一个  $m \times n$  二维字符网格 board 和一个字符串单词 word 。
 * 如果 word 存在于网格中，返回 true；否则，返回 false。
 * 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不允许被重复使用。
 *
 * 示例:
 * 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCED"
 * 输出: true
 *
 * 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "SEE"
 * 输出: true
 *
 * 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCB"
 * 输出: false
 *
 * 提示:
 *  $m == board.length$ 
 *  $n == board[i].length$ 
 *  $1 \leq m, n \leq 6$ 
 *  $1 \leq word.length \leq 15$ 
 * board 和 word 仅由大小写英文字母组成
 *
 * 链接: https://leetcode.cn/problems/word-search/
 */
```

```
class Solution {
public:
    /**
     * 检查单词是否存在于网格中
     *
     * 算法思路:
     * 1. 遍历网格中的每个位置作为起点
     * 2. 对于每个起点，使用回溯算法搜索单词
```

```

* 3. 在回溯过程中，标记已访问的位置，避免重复使用
* 4. 向四个方向探索：上、下、左、右
* 5. 如果找到完整单词，返回 true
* 6. 如果当前路径不匹配，回溯并尝试其他路径
*
* 时间复杂度：O(m*n*4^L)，其中 m 和 n 是网格的行数和列数，L 是单词的长度
* 空间复杂度：O(L)，递归栈深度
*
* @param board 二维字符网格
* @param word 单词
* @return 单词是否存在于网格中
*/
bool exist(vector<vector<char>>& board, string word) {
    // 边界条件检查
    if (board.empty() || board[0].empty() || word.empty()) return false;

    // 遍历网格中的每个位置作为起点
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[0].size(); j++) {
            if (backtrack(board, word, i, j, 0))
                return true;
        }
    }
    return false;
}

private:
/***
 * 回溯函数搜索单词
 *
 * @param board 二维字符网格
 * @param word 单词
 * @param row 当前行
 * @param col 当前列
 * @param index 当前处理的单词字符索引
 * @return 是否找到单词
*/
bool backtrack(vector<vector<char>>& board, string word, int row, int col, int index) {
    // 终止条件：已找到完整单词
    if (index == word.length()) return true;

    // 边界检查和字符匹配检查
    if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size() ||

```

```

        board[row][col] != word[index])) {
            return false;
        }

        // 标记已访问
        char temp = board[row][col];
        board[row][col] = '#';

        // 向四个方向探索
        bool found = backtrack(board, word, row + 1, col, index + 1) || // 下
                    backtrack(board, word, row - 1, col, index + 1) || // 上
                    backtrack(board, word, row, col + 1, index + 1) || // 右
                    backtrack(board, word, row, col - 1, index + 1); // 左

        // 回溯
        board[row][col] = temp;

        return found;
    }
};

// 打印网格
void printBoard(const vector<vector<char>>& board) {
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[0].size(); j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

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

    // 测试用例 1
    vector<vector<char>> board1 = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
    };
    string word1 = "ABCED";
    bool result1 = solution.exist(board1, word1);
}

```

```

cout << "网格:" << endl;
printBoard(board1);
cout << "单词: \\" << word1 << "\"" << endl;
cout << "结果: " << (result1 ? "true" : "false") << endl;

// 测试用例 2
vector<vector<char>> board2 = {
    {'A', 'B', 'C', 'E'},
    {'S', 'F', 'C', 'S'},
    {'A', 'D', 'E', 'E'}
};
string word2 = "SEE";
bool result2 = solution.exist(board2, word2);
cout << "\n 网格:" << endl;
printBoard(board2);
cout << "单词: \\" << word2 << "\"" << endl;
cout << "结果: " << (result2 ? "true" : "false") << endl;

// 测试用例 3
vector<vector<char>> board3 = {
    {'A', 'B', 'C', 'E'},
    {'S', 'F', 'C', 'S'},
    {'A', 'D', 'E', 'E'}
};
string word3 = "ABCB";
bool result3 = solution.exist(board3, word3);
cout << "\n 网格:" << endl;
printBoard(board3);
cout << "单词: \\" << word3 << "\"" << endl;
cout << "结果: " << (result3 ? "true" : "false") << endl;

return 0;
}
=====

文件: Code13_WordSearch.java
=====

package class038;

/**
 * LeetCode 79. 单词搜索
 *

```

文件: Code13\_WordSearch.java

```

=====
package class038;

/**
 * LeetCode 79. 单词搜索
 *

```

\* 题目描述:

\* 给定一个  $m \times n$  二维字符网格 board 和一个字符串单词 word。

\* 如果 word 存在于网格中，返回 true；否则，返回 false。

\* 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不允许被重复使用。

\*

\* 示例:

\* 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCED"

\* 输出: true

\*

\* 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "SEE"

\* 输出: true

\*

\* 输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCB"

\* 输出: false

\*

\* 提示:

\*  $m == board.length$

\*  $n == board[i].length$

\*  $1 \leq m, n \leq 6$

\*  $1 \leq word.length \leq 15$

\* board 和 word 仅由大小写英文字母组成

\*

\* 链接: <https://leetcode.cn/problems/word-search/>

\*/

```
public class Code13_WordSearch {

    /**
     * 检查单词是否存在与网格中
     *
     * 算法思路:
     * 1. 遍历网格中的每个位置作为起点
     * 2. 对于每个起点，使用回溯算法搜索单词
     * 3. 在回溯过程中，标记已访问的位置，避免重复使用
     * 4. 向四个方向探索：上、下、左、右
     * 5. 如果找到完整单词，返回 true
     * 6. 如果当前路径不匹配，回溯并尝试其他路径
     *
     * 时间复杂度: O(m*n*4^L)，其中 m 和 n 是网格的行数和列数，L 是单词的长度
     * 空间复杂度: O(L)，递归栈深度
     *
     * @param board 二维字符网格
    }
```

```

* @param word 单词
* @return 单词是否存在于网格中
*/
public static boolean exist(char[][] board, String word) {
    // 边界条件检查
    if (board == null || board.length == 0 || word == null) return false;

    // 遍历网格中的每个位置作为起点
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (backtrack(board, word, i, j, 0))
                return true;
        }
    }
    return false;
}

/**
 * 回溯函数搜索单词
 *
 * @param board 二维字符网格
 * @param word 单词
 * @param row 当前行
 * @param col 当前列
 * @param index 当前处理的单词字符索引
 * @return 是否找到单词
*/
private static boolean backtrack(char[][] board, String word, int row, int col, int index) {
    // 终止条件：已找到完整单词
    if (index == word.length()) return true;

    // 边界检查和字符匹配检查
    if (row < 0 || row >= board.length || col < 0 || col >= board[0].length ||
        board[row][col] != word.charAt(index)) {
        return false;
    }

    // 标记已访问
    char temp = board[row][col];
    board[row][col] = '#';

    // 向四个方向探索
    boolean found = backtrack(board, word, row + 1, col, index + 1) || // 下

```

```

        backtrack(board, word, row - 1, col, index + 1) || // 上
        backtrack(board, word, row, col + 1, index + 1) || // 右
        backtrack(board, word, row, col - 1, index + 1);    // 左

    // 回溯
    board[row][col] = temp;

    return found;
}

// 打印网格
public static void printBoard(char[][] board) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println();
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    char[][] board1 = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
    };
    String word1 = "ABCED";
    boolean result1 = exist(board1, word1);
    System.out.println("网格:");
    printBoard(board1);
    System.out.println("单词: " + word1 + ")");
    System.out.println("结果: " + result1);

    // 测试用例 2
    char[][] board2 = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
    };
    String word2 = "SEE";
    boolean result2 = exist(board2, word2);
}
```

```

System.out.println("\n网格:");
printBoard(board2);
System.out.println("单词: " + word2 + ")");
System.out.println("结果: " + result2);

// 测试用例 3
char[][] board3 = {
    {'A', 'B', 'C', 'E'},
    {'S', 'F', 'C', 'S'},
    {'A', 'D', 'E', 'E'}
};
String word3 = "ABCB";
boolean result3 = exist(board3, word3);
System.out.println("\n网格:");
printBoard(board3);
System.out.println("单词: " + word3 + ")");
System.out.println("结果: " + result3);

// 测试用例 4
char[][] board4 = {
    {'A', 'B', 'C', 'E'},
    {'S', 'F', 'C', 'S'},
    {'A', 'D', 'E', 'E'}
};
String word4 = "ABCSEEEFS";
boolean result4 = exist(board4, word4);
System.out.println("\n网格:");
printBoard(board4);
System.out.println("单词: " + word4 + ")");
System.out.println("结果: " + result4);
}

}
=====

文件: Code13_WordSearch.py
=====

"""

```

## LeetCode 79. 单词搜索

### 题目描述:

给定一个  $m \times n$  二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true；否则，返回 false。

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不允许被重复使用。

示例：

输入：board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCCED"  
输出：true

输入：board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "SEE"  
输出：true

输入：board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCB"  
输出：false

提示：

m == board.length  
n == board[i].length  
1 <= m, n <= 6  
1 <= word.length <= 15  
board 和 word 仅由大小写英文字母组成

链接：<https://leetcode.cn/problems/word-search/>

"""

```
class Solution:
```

```
    def exist(self, board, word):
```

```
        """
```

```
        检查单词是否存在子网格中
```

算法思路：

1. 遍历网格中的每个位置作为起点
2. 对于每个起点，使用回溯算法搜索单词
3. 在回溯过程中，标记已访问的位置，避免重复使用
4. 向四个方向探索：上、下、左、右
5. 如果找到完整单词，返回 true
6. 如果当前路径不匹配，回溯并尝试其他路径

时间复杂度：O(m\*n\*4^L)，其中 m 和 n 是网格的行数和列数，L 是单词的长度

空间复杂度：O(L)，递归栈深度

:param board: 二维字符网格

:param word: 单词

:return: 单词是否存在子网格中

```

"""
# 边界条件检查
if not board or not board[0] or not word:
    return False

# 遍历网格中的每个位置作为起点
for i in range(len(board)):
    for j in range(len(board[0])):
        if self.backtrack(board, word, i, j, 0):
            return True
return False

def backtrack(self, board, word, row, col, index):
    """
回溯函数搜索单词

:param board: 二维字符网格
:param word: 单词
:param row: 当前行
:param col: 当前列
:param index: 当前处理的单词字符索引
:return: 是否找到单词
"""

# 终止条件: 已找到完整单词
if index == len(word):
    return True

# 边界检查和字符匹配检查
if (row < 0 or row >= len(board) or col < 0 or col >= len(board[0])) or
    board[row][col] != word[index]:
    return False

# 标记已访问
temp = board[row][col]
board[row][col] = '#'

# 向四个方向探索
found = (self.backtrack(board, word, row + 1, col, index + 1) or # 下
          self.backtrack(board, word, row - 1, col, index + 1) or # 上
          self.backtrack(board, word, row, col + 1, index + 1) or # 右
          self.backtrack(board, word, row, col - 1, index + 1)) # 左

# 回溯
board[row][col] = temp

```

```
board[row][col] = temp

return found

# 打印网格
def printBoard(board):
    for i in range(len(board)):
        for j in range(len(board[0])):
            print(board[i][j] + " ", end="")
    print()

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    board1 = [
        ['A', 'B', 'C', 'E'],
        ['S', 'F', 'C', 'S'],
        ['A', 'D', 'E', 'E']
    ]
    word1 = "ABCED"
    result1 = solution.exist(board1, word1)
    print("网格:")
    printBoard(board1)
    print(f"单词: \'{word1}\'")
    print(f"结果: {result1}")

    # 测试用例 2
    board2 = [
        ['A', 'B', 'C', 'E'],
        ['S', 'F', 'C', 'S'],
        ['A', 'D', 'E', 'E']
    ]
    word2 = "SEE"
    result2 = solution.exist(board2, word2)
    print("\n网格:")
    printBoard(board2)
    print(f"单词: \'{word2}\'")
    print(f"结果: {result2}")
```

```
# 测试用例 3
board3 = [
    [ 'A', 'B', 'C', 'E' ],
    [ 'S', 'F', 'C', 'S' ],
    [ 'A', 'D', 'E', 'E' ]
]
word3 = "ABCB"
result3 = solution.exist(board3, word3)
print("\n网格:")
printBoard(board3)
print(f"单词: {word3}")
print(f"结果: {result3}")
```

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

```
=====
文件: Code14_PalindromePartitioning.cpp
=====
```

```
#include <vector>
#include <string>
#include <iostream>

using namespace std;

/***
 * LeetCode 131. 分割回文串
 *
 * 题目描述:
 * 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。
 *
 * 示例:
 * 输入: s = "aab"
 * 输出: [[ "a", "a", "b"], [ "aa", "b"]]
 *
 * 输入: s = "a"
 * 输出: [[ "a"]]
 *
 * 提示:
 * 1 <= s.length <= 16
 * s 仅由小写英文字母组成
```

```

*
* 链接: https://leetcode.cn/problems/palindrome-partitioning/
*/

```

```

class Solution {
public:
    /**
     * 分割回文串
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有可能的分割方案
     * 2. 对于每个位置，判断从当前位置开始的子串是否为回文串
     * 3. 如果是回文串，则将其加入路径，并递归处理剩余部分
     * 4. 回溯时移除当前子串，尝试其他分割方式
     *
     * 时间复杂度: O(N * 2^N)，其中 N 是字符串长度。在最坏情况下，每个字符都可以单独作为回文串，共有 O(2^N) 种分割方案，每种方案需要 O(N) 时间检查回文。
     * 空间复杂度: O(N)，递归栈的深度加上存储当前路径的空间。
     *
     * @param s 输入字符串
     * @return 所有可能的分割方案
     */
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;

        // 回溯生成所有分割方案
        backtrack(s, 0, path, result);
        return result;
    }
}

private:
    /**
     * 回溯函数生成分割方案
     *
     * @param s 输入字符串
     * @param start 当前处理的起始位置
     * @param path 当前分割路径
     * @param result 结果列表
     */
    void backtrack(const string& s, int start, vector<string>& path, vector<vector<string>>& result) {
        // 终止条件：已处理到字符串末尾

```

```

    if (start == s.size()) {
        result.push_back(path);
        return;
    }

    // 从 start 开始尝试不同长度的子串
    for (int end = start + 1; end <= s.size(); end++) {
        // 判断子串 s[start...end-1] 是否为回文串
        if (isPalindrome(s, start, end - 1)) {
            // 将回文子串加入路径
            path.push_back(s.substr(start, end - start));
            // 递归处理剩余部分
            backtrack(s, end, path, result);
            // 回溯: 移除当前子串
            path.pop_back();
        }
    }
}

/***
 * 判断字符串的子串是否为回文串
 *
 * @param s 原始字符串
 * @param left 左边界 (包含)
 * @param right 右边界 (包含)
 * @return 是否为回文串
 */
bool isPalindrome(const string& s, int left, int right) {
    while (left < right) {
        if (s[left++] != s[right--]) {
            return false;
        }
    }
    return true;
};

// 辅助函数: 打印结果
void printResult(const vector<vector<string>>& result) {
    cout << "[";
    for (size_t i = 0; i < result.size(); i++) {
        cout << "[";
        for (size_t j = 0; j < result[i].size(); j++) {

```

```
        cout << "\"" << result[i][j] << "\"";
        if (j < result[i].size() - 1) {
            cout << ", ";
        }
    }
    cout << "]";
    if (i < result.size() - 1) {
        cout << ", ";
    }
}
cout << "]" << endl;
}

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

    // 测试用例 1
    string test1 = "aab";
    vector<vector<string>> result1 = solution.partition(test1);
    cout << "输入: \"\" " << test1 << "\"\" << endl;
    cout << "输出: ";
    printResult(result1);

    // 测试用例 2
    string test2 = "a";
    vector<vector<string>> result2 = solution.partition(test2);
    cout << "\n 输入: \"\" " << test2 << "\"\" << endl;
    cout << "输出: ";
    printResult(result2);

    // 测试用例 3
    string test3 = "aabb";
    vector<vector<string>> result3 = solution.partition(test3);
    cout << "\n 输入: \"\" " << test3 << "\"\" << endl;
    cout << "输出: ";
    printResult(result3);

    return 0;
}
```

---

文件: Code14\_PalindromePartitioning.java

```
=====
```

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

```
/**
```

```
* LeetCode 131. 分割回文串
```

```
*
```

```
* 题目描述:
```

```
* 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。
```

```
*
```

```
* 示例:
```

```
* 输入: s = "aab"
```

```
* 输出: [["a", "a", "b"], ["aa", "b"]]
```

```
*
```

```
* 输入: s = "a"
```

```
* 输出: [["a"]]
```

```
*
```

```
* 提示:
```

```
* 1 <= s.length <= 16
```

```
* s 仅由小写英文字母组成
```

```
*
```

```
* 链接: https://leetcode.cn/problems/palindrome-partitioning/
```

```
*
```

```
* 算法思路:
```

```
* 1. 使用回溯算法生成所有可能的分割方案
```

```
* 2. 对于每个位置，判断从当前位置开始的子串是否为回文串
```

```
* 3. 如果是回文串，则将其加入路径，并递归处理剩余部分
```

```
* 4. 回溯时移除当前子串，尝试其他分割方式
```

```
*
```

```
* 剪枝策略:
```

```
* 1. 可行性剪枝：只在子串是回文串时才继续递归
```

```
* 2. 提前终止：当剩余字符串无法形成有效分割时提前终止
```

```
* 3. 预处理优化：预先计算所有子串是否为回文串
```

```
*
```

```
* 时间复杂度：O(N * 2^N)，其中 N 是字符串长度。在最坏情况下，每个字符都可以单独作为回文串，共有 O(2^N) 种分割方案，每种方案需要 O(N) 时间检查回文。
```

```
* 空间复杂度：O(N^2)，递归栈的深度加上存储当前路径的空间和预处理的回文矩阵。
```

```
*
```

```
* 工程化考量:
```

```
* 1. 边界处理：处理空字符串和单字符字符串的特殊情况
```

```
* 2. 性能优化：使用动态规划预处理回文串信息
```

```
* 3. 内存管理：合理使用数据结构减少内存占用
```

```
* 4. 可读性：添加详细注释和变量命名
```

```

* 5. 异常处理：验证输入参数的有效性
* 6. 模块化设计：将核心逻辑封装成独立方法
* 7. 可维护性：添加详细注释和文档说明
*/
public class Code14_PalindromePartitioning {

    /**
     * 分割回文串
     *
     * @param s 输入字符串
     * @return 所有可能的分割方案
     */
    public static List<List<String>> partition(String s) {
        // 边界条件检查
        if (s == null || s.isEmpty()) {
            return new ArrayList<>();
        }

        List<List<String>> result = new ArrayList<>();
        List<String> path = new ArrayList<>();
        // 回溯生成所有分割方案
        backtrack(s, 0, path, result);
        return result;
    }

    /**
     * 回溯函数生成分割方案
     *
     * @param s 输入字符串
     * @param start 当前处理的起始位置
     * @param path 当前分割路径
     * @param result 结果列表
     */
    private static void backtrack(String s, int start, List<String> path, List<List<String>>
result) {
        // 终止条件：已处理到字符串末尾
        if (start == s.length()) {
            result.add(new ArrayList<>(path));
            return;
        }

        // 从 start 开始尝试不同长度的子串
        for (int end = start + 1; end <= s.length(); end++) {

```

```

// 可行性剪枝：判断子串 s[start...end-1]是否为回文串
if (isPalindrome(s, start, end - 1)) {
    // 将回文子串加入路径
    path.add(s.substring(start, end));
    // 递归处理剩余部分
    backtrack(s, end, path, result);
    // 回溯：移除当前子串
    path.remove(path.size() - 1);
}
}

/**
 * 判断字符串的子串是否为回文串
 *
 * @param s 原始字符串
 * @param left 左边界（包含）
 * @param right 右边界（包含）
 * @return 是否为回文串
 */
private static boolean isPalindrome(String s, int left, int right) {
    while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) {
            return false;
        }
    }
    return true;
}

/**
 * 优化版本：使用动态规划预处理回文串信息
 *
 * @param s 输入字符串
 * @return 所有可能的分割方案
 */
public static List<List<String>> partitionOptimized(String s) {
    // 边界条件检查
    if (s == null || s.isEmpty()) {
        return new ArrayList<>();
    }

    int n = s.length();
    // 预处理：计算所有子串是否为回文串

```

```

boolean[][] isPalindrome = new boolean[n][n];
for (int i = 0; i < n; i++) {
    isPalindrome[i][i] = true; // 单个字符都是回文串
}

for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
        int j = i + len - 1;
        if (s.charAt(i) == s.charAt(j)) {
            if (len == 2 || isPalindrome[i + 1][j - 1]) {
                isPalindrome[i][j] = true;
            }
        }
    }
}

List<List<String>> result = new ArrayList<>();
List<String> path = new ArrayList<>();
// 回溯生成所有分割方案
backtrackOptimized(s, 0, path, result, isPalindrome);
return result;
}

/**
 * 优化版本的回溯函数
 *
 * @param s 输入字符串
 * @param start 当前处理的起始位置
 * @param path 当前分割路径
 * @param result 结果列表
 * @param isPalindrome 预处理的回文串信息
 */
private static void backtrackOptimized(String s, int start, List<String> path,
List<List<String>> result, boolean[][] isPalindrome) {
    // 终止条件：已处理到字符串末尾
    if (start == s.length()) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 从 start 开始尝试不同长度的子串
    for (int end = start + 1; end <= s.length(); end++) {
        // 可行性剪枝：使用预处理的回文串信息
    }
}

```

```

        if (isPalindrome[start][end - 1]) {
            // 将回文子串加入路径
            path.add(s.substring(start, end));
            // 递归处理剩余部分
            backtrackOptimized(s, end, path, result, isPalindrome);
            // 回溯：移除当前子串
            path.remove(path.size() - 1);
        }
    }
}

/**
 * 验证分割方案是否正确
 *
 * @param s 原始字符串
 * @param partition 分割方案
 * @return 是否正确
 */
public static boolean isValidPartition(String s, List<String> partition) {
    if (partition == null) return false;

    // 检查拼接后是否等于原字符串
    StringBuilder sb = new StringBuilder();
    for (String part : partition) {
        if (part == null || part.isEmpty()) return false;
        sb.append(part);
    }
    if (!sb.toString().equals(s)) return false;

    // 检查每个部分是否为回文串
    for (String part : partition) {
        if (!isPalindrome(part, 0, part.length() - 1)) return false;
    }

    return true;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
    // 测试用例 1
    String test1 = "aab";
}

```

```
List<List<String>> result1 = partition(test1);
System.out.println("==> 测试用例 1 ==>");
System.out.println("输入: " + test1 + ")");
System.out.println("输出: " + result1);
for (List<String> partition : result1) {
    System.out.println("  方案正确性: " + isValidPartition(test1, partition));
}

// 测试用例 2
String test2 = "a";
List<List<String>> result2 = partition(test2);
System.out.println("\n==> 测试用例 2 ==>");
System.out.println("输入: " + test2 + ")");
System.out.println("输出: " + result2);
for (List<String> partition : result2) {
    System.out.println("  方案正确性: " + isValidPartition(test2, partition));
}

// 测试用例 3
String test3 = "aabb";
List<List<String>> result3 = partition(test3);
System.out.println("\n==> 测试用例 3 ==>");
System.out.println("输入: " + test3 + ")");
System.out.println("输出: " + result3);
for (List<String> partition : result3) {
    System.out.println("  方案正确性: " + isValidPartition(test3, partition));
}

// 性能对比测试
String test4 = "abccba";
long startTime = System.currentTimeMillis();
List<List<String>> result4a = partition(test4);
long endTime = System.currentTimeMillis();
System.out.println("\n==> 性能对比测试 ==>");
System.out.println("输入: " + test4 + ")");
System.out.println("基础版本解的数量: " + result4a.size());
System.out.println("基础版本耗时: " + (endTime - startTime) + " ms");

startTime = System.currentTimeMillis();
List<List<String>> result4b = partitionOptimized(test4);
endTime = System.currentTimeMillis();
System.out.println("优化版本解的数量: " + result4b.size());
System.out.println("优化版本耗时: " + (endTime - startTime) + " ms");
```

```
    }  
}
```

```
=====
```

文件: Code14\_PalindromePartitioning.py

```
=====
```

```
"""
```

LeetCode 131. 分割回文串

题目描述:

给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。

示例:

输入: s = "aab"

输出: [["a", "a", "b"], ["aa", "b"]]

输入: s = "a"

输出: [["a"]]

提示:

1 <= s.length <= 16

s 仅由小写英文字母组成

链接: <https://leetcode.cn/problems/palindrome-partitioning/>

```
"""
```

```
class Solution:
```

```
    def partition(self, s):
```

```
        """
```

分割回文串

算法思路:

1. 使用回溯算法生成所有可能的分割方案
2. 对于每个位置，判断从当前位置开始的子串是否为回文串
3. 如果是回文串，则将其加入路径，并递归处理剩余部分
4. 回溯时移除当前子串，尝试其他分割方式

时间复杂度:  $O(N * 2^N)$ ，其中 N 是字符串长度。在最坏情况下，每个字符都可以单独作为回文串，共有  $O(2^N)$  种分割方案，每种方案需要  $O(N)$  时间检查回文。

空间复杂度:  $O(N)$ ，递归栈的深度加上存储当前路径的空间。

:param s: 输入字符串

```

:return: 所有可能的分割方案
"""

result = []
path = []

# 回溯生成所有分割方案
self.backtrack(s, 0, path, result)
return result

def backtrack(self, s, start, path, result):
    """
    回溯函数生成分割方案

    :param s: 输入字符串
    :param start: 当前处理的起始位置
    :param path: 当前分割路径
    :param result: 结果列表
    """

    # 终止条件：已处理到字符串末尾
    if start == len(s):
        result.append(path[:])  # 深拷贝当前路径
        return

    # 从 start 开始尝试不同长度的子串
    for end in range(start + 1, len(s) + 1):
        # 判断子串 s[start:end] 是否为回文串
        if self.is_palindrome(s, start, end - 1):
            # 将回文子串加入路径
            path.append(s[start:end])
            # 递归处理剩余部分
            self.backtrack(s, end, path, result)
            # 回溯：移除当前子串
            path.pop()

def is_palindrome(self, s, left, right):
    """
    判断字符串的子串是否为回文串

    :param s: 原始字符串
    :param left: 左边界（包含）
    :param right: 右边界（包含）
    :return: 是否为回文串
    """

```

```
while left < right:
    if s[left] != s[right]:
        return False
    left += 1
    right -= 1
return True

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    test1 = "aab"
    result1 = solution.partition(test1)
    print(f'输入: "{test1}"')
    print(f"输出: {result1}")

    # 测试用例 2
    test2 = "a"
    result2 = solution.partition(test2)
    print(f'\n输入: "{test2}"')
    print(f"输出: {result2}")

    # 测试用例 3
    test3 = "aabb"
    result3 = solution.partition(test3)
    print(f'\n输入: "{test3}"')
    print(f"输出: {result3}")

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

=====

文件: Code15\_WordSearchII.cpp

=====

```
#include <vector>
#include <string>
#include <iostream>
#include <unordered_set>
```

```
using namespace std;

/**
 * LeetCode 212. 单词搜索 II
 *
 * 题目描述:
 * 给定一个二维网格 board 和一个字典中的单词列表 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。
 *
 * 示例:
 * 输入:
 * board = [
 *   [ 'o', 'a', 'a', 'n'],
 *   [ 'e', 't', 'a', 'e'],
 *   [ 'i', 'h', 'k', 'r'],
 *   [ 'i', 'f', 'l', 'v']
 * ]
 * words = ["oath", "pea", "eat", "rain"]
 * 输出: ["eat", "oath"]
 *
 * 提示:
 * m == board.length
 * n == board[i].length
 * 1 <= m, n <= 12
 * 1 <= words.length <= 3 * 10^4
 * 1 <= words[i].length <= 10
 * board 和 words[i] 仅由小写英文字母组成
 * words 中的所有字符串互不相同
 *
 * 链接: https://leetcode.cn/problems/word-search-ii/
 */
```

```
// Trie树节点定义
struct TrieNode {
    TrieNode* children[26]; // 26个小写字母
    string word; // 存储完整单词，非空表示这是一个单词的结尾

    // 构造函数
    TrieNode() : word("") {
        for (int i = 0; i < 26; i++) {
```

```

        children[i] = nullptr;
    }
}

// 析构函数
~TrieNode() {
    for (int i = 0; i < 26; i++) {
        if (children[i]) {
            delete children[i];
        }
    }
}
};

class Solution {
public:
    /**
     * 查找二维网格中所有存在于字典中的单词
     *
     * 算法思路：
     * 1. 构建 Trie 树，将所有单词插入 Trie 中
     * 2. 对二维网格中的每个单元格作为起点，进行深度优先搜索
     * 3. 使用 Trie 树来剪枝无效的搜索路径
     * 4. 找到单词后，将其加入结果集并从 Trie 中移除，避免重复添加
     *
     * 时间复杂度：O(M*N*4^L)，其中 M 和 N 是网格的行数和列数，L 是单词的最大长度。每个单元格最多被访问 4^L 次。
     * 空间复杂度：O(K)，其中 K 是所有单词的字符总数，用于存储 Trie 树。
     *
     * @param board 二维字符网格
     * @param words 单词列表
     * @return 网格中存在的单词列表
    */
vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
    vector<string> result;
    if (board.empty() || board[0].empty() || words.empty()) {
        return result;
    }

    // 构建 Trie 树
    TrieNode* root = buildTrie(words);

    // 遍历网格中的每个单元格作为起点

```

```

int m = board.size();
int n = board[0].size();
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        dfs(board, i, j, root, result);
    }
}

// 释放 Trie 树内存
delete root;

return result;
}

private:
/***
 * 构建 Trie 树
 *
 * @param words 单词列表
 * @return Trie 树的根节点
 */
TrieNode* buildTrie(vector<string>& words) {
    TrieNode* root = new TrieNode();
    for (const string& word : words) {
        TrieNode* node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }
        node->word = word; // 标记单词结尾
    }
    return root;
}

/***
 * 深度优先搜索
 *
 * @param board 二维字符网格
 * @param i 当前行索引
 * @param j 当前列索引
 */

```

```

* @param node 当前 Trie 节点
* @param result 结果列表
*/
void dfs(vector<vector<char>>& board, int i, int j, TrieNode* node, vector<string>& result) {
    // 检查边界条件
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || board[i][j] == '#') {
        return;
    }

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

    // 如果当前字符不在 Trie 的子节点中，剪枝
    if (!node->children[index]) {
        return;
    }

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

    // 如果找到一个单词
    if (!node->word.empty()) {
        result.push_back(node->word);
        node->word = ""; // 移除单词，避免重复添加
    }

    // 标记当前单元格为已访问
    board[i][j] = '#';

    // 向四个方向搜索
    dfs(board, i + 1, j, node, result);
    dfs(board, i - 1, j, node, result);
    dfs(board, i, j + 1, node, result);
    dfs(board, i, j - 1, node, result);

    // 回溯：恢复当前单元格
    board[i][j] = c;
}

};

// 辅助函数：打印结果
void printResult(const vector<string>& result) {
    cout << "[";

```

```

for (size_t i = 0; i < result.size(); i++) {
    cout << "\"" << result[i] << "\"";
    if (i < result.size() - 1) {
        cout << ", ";
    }
}
cout << "]" << endl;
}

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

    // 测试用例 1
    vector<vector<char>> board1 = {
        {'o', 'a', 'a', 'n'},
        {'e', 't', 'a', 'e'},
        {'i', 'h', 'k', 'r'},
        {'i', 'f', 'l', 'v'}
    };
    vector<string> words1 = {"oath", "pea", "eat", "rain"};
    vector<string> result1 = solution.findWords(board1, words1);
    cout << "输入:" << endl;
    cout << "board = [[o, a, a, n], [e, t, a, e], [i, h, k, r], [i, f, l, v]]]" << endl;
    cout << "words = [oath, pea, eat, rain]" << endl;
    cout << "输出: ";
    printResult(result1);

    // 测试用例 2
    vector<vector<char>> board2 = {{'a', 'b'}, {'c', 'd'}};
    vector<string> words2 = {"abcb"};
    vector<string> result2 = solution.findWords(board2, words2);
    cout << "\n输入:" << endl;
    cout << "board = [[a, b], [c, d]]" << endl;
    cout << "words = [abcb]" << endl;
    cout << "输出: ";
    printResult(result2);

    return 0;
}
=====
```

文件: Code15\_WordSearchII.java

```
=====
```

```
package class038;
```

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

```
/**
```

```
* LeetCode 212. 单词搜索 II
```

```
*
```

```
* 题目描述:
```

```
* 给定一个二维网格 board 和一个字典中的单词列表 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。
```

```
*
```

```
* 示例:
```

```
* 输入:
```

```
* board = [
```

```
*   [’o’, ’a’, ’a’, ’n’],
```

```
*   [’e’, ’t’, ’a’, ’e’],
```

```
*   [’i’, ’h’, ’k’, ’r’],
```

```
*   [’i’, ’f’, ’l’, ’v’]
```

```
* ]
```

```
* words = [”oath”, ”pea”, ”eat”, ”rain”]
```

```
* 输出: [”eat”, ”oath”]
```

```
*
```

```
* 提示:
```

```
* m == board.length
```

```
* n == board[i].length
```

```
* 1 <= m, n <= 12
```

```
* 1 <= words.length <= 3 * 10^4
```

```
* 1 <= words[i].length <= 10
```

```
* board 和 words[i] 仅由小写英文字母组成
```

```
* words 中的所有字符串互不相同
```

```
*
```

```
* 链接: https://leetcode.cn/problems/word-search-ii/
```

```
*/
```

```
public class Code15_WordSearchII {
```

```
    // Trie 树节点定义
```

```
    private static class TrieNode {
```

```
        TrieNode[] children = new TrieNode[26]; // 26 个小写字母
```

```
String word; // 存储完整单词，非 null 表示这是一个单词的结尾
}

/***
 * 查找二维网格中所有存在于字典中的单词
 *
 * 算法思路：
 * 1. 构建 Trie 树，将所有单词插入 Trie 中
 * 2. 对二维网格中的每个单元格作为起点，进行深度优先搜索
 * 3. 使用 Trie 树来剪枝无效的搜索路径
 * 4. 找到单词后，将其加入结果集并从 Trie 中移除，避免重复添加
 *
 * 时间复杂度：O(M*N*4^L)，其中 M 和 N 是网格的行数和列数，L 是单词的最大长度。每个单元格最多被访问  $4^L$  次。
 * 空间复杂度：O(K)，其中 K 是所有单词的字符总数，用于存储 Trie 树。
 *
 * @param board 二维字符网格
 * @param words 单词列表
 * @return 网格中存在的单词列表
 */
public static 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
    TrieNode root = buildTrie(words);

    // 遍历网格中的每个单元格作为起点
    int m = board.length;
    int n = board[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            dfs(board, i, j, root, result);
        }
    }

    return result;
}

/***
```

```

* 构建 Trie 树
*
* @param words 单词列表
* @return Trie 树的根节点
*/
private static TrieNode buildTrie(String[] words) {
    TrieNode root = new TrieNode();
    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.word = word; // 标记单词结尾
    }
    return root;
}

/**
* 深度优先搜索
*
* @param board 二维字符网格
* @param i 当前行索引
* @param j 当前列索引
* @param node 当前 Trie 节点
* @param result 结果列表
*/
private static void dfs(char[][] board, int i, int j, TrieNode node, List<String> result) {
    // 检查边界条件
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] == '#') {
        return;
    }

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

    // 如果当前字符不在 Trie 的子节点中，剪枝
    if (node.children[index] == null) {
        return;
    }

```

```

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

// 如果找到一个单词
if (node.word != null) {
    result.add(node.word);
    node.word = null; // 移除单词，避免重复添加
}

// 标记当前单元格为已访问
board[i][j] = '#';

// 向四个方向搜索
dfs(board, i + 1, j, node, result);
dfs(board, i - 1, j, node, result);
dfs(board, i, j + 1, node, result);
dfs(board, i, j - 1, node, result);

// 回溯：恢复当前单元格
board[i][j] = c;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    char[][] board1 = {
        {'o', 'a', 'a', 'n'},
        {'e', 't', 'a', 'e'},
        {'i', 'h', 'k', 'r'},
        {'i', 'f', 'l', 'v'}
    };
    String[] words1 = {"oath", "pea", "eat", "rain"};
    List<String> result1 = findWords(board1, words1);
    System.out.println("输入:");
    System.out.println("board =");
    System.out.println("[" + board1[0] + ", " + board1[1] + ", " + board1[2] + ", " + board1[3] + "]");
    System.out.println("words = [" + words1[0] + ", " + words1[1] + ", " + words1[2] + ", " + words1[3] + "]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    char[][] board2 = {{'a', 'b'}, {'c', 'd'}};
    String[] words2 = {"abcb"};
}

```

```
        List<String> result2 = findWords(board2, words2);
        System.out.println("\n 输入:");
        System.out.println("board = [[ 'a' , 'b' ], [ 'c' , 'd' ] ]");
        System.out.println("words = [\"abcb\"]");
        System.out.println("输出: " + result2);
    }
}
```

=====

文件: Code15\_WordSearchII.py

=====

"""

LeetCode 212. 单词搜索 II

题目描述:

给定一个二维网格 board 和一个字典中的单词列表 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。

示例:

输入:

```
board = [
    [ 'o' , 'a' , 'a' , 'n' ],
    [ 'e' , 't' , 'a' , 'e' ],
    [ 'i' , 'h' , 'k' , 'r' ],
    [ 'i' , 'f' , 'l' , 'v' ]
]
words = [ "oath" , "pea" , "eat" , "rain" ]
输出: [ "eat" , "oath" ]
```

提示:

$m == board.length$

$n == board[i].length$

$1 \leq m, n \leq 12$

$1 \leq words.length \leq 3 * 10^4$

$1 \leq words[i].length \leq 10$

board 和 words[i] 仅由小写英文字母组成

words 中的所有字符串互不相同

链接: <https://leetcode.cn/problems/word-search-ii/>

"""

```

class Solution:
    def findWords(self, board, words):
        """
        查找二维网格中所有存在于字典中的单词
    
```

算法思路：

1. 构建 Trie 树，将所有单词插入 Trie 中
2. 对二维网格中的每个单元格作为起点，进行深度优先搜索
3. 使用 Trie 树来剪枝无效的搜索路径
4. 找到单词后，将其加入结果集并从 Trie 中移除，避免重复添加

时间复杂度： $O(M \times N \times 4^L)$ ，其中 M 和 N 是网格的行数和列数，L 是单词的最大长度。每个单元格最多被访问  $4^L$  次。

空间复杂度： $O(K)$ ，其中 K 是所有单词的字符总数，用于存储 Trie 树。

```

:param board: 二维字符网格
:param words: 单词列表
:return: 网格中存在的单词列表
"""

# 构建 Trie 树
root = {}
for word in words:
    node = root
    for char in word:
        if char not in node:
            node[char] = {}
        node = node[char]
    node['#'] = word # 标记单词结尾

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

def dfs(i, j, node):
    """
    深度优先搜索
    :param i: 当前行索引
    :param j: 当前列索引
    :param node: 当前 Trie 节点
    """

    # 检查边界条件
    if i < 0 or i >= m or j < 0 or j >= n or board[i][j] == '#':
        return
    char = board[i][j]
    if '#' in node:
        result.add(word)
    if char in node:
        node = node[char]
        if '#' in node:
            result.add(word)
        del node['#']
        dfs(i+1, j, node)
        dfs(i-1, j, node)
        dfs(i, j+1, node)
        dfs(i, j-1, node)
    else:
        return

```

```

        return

    char = board[i][j]
    # 如果当前字符不在 Trie 的子节点中，剪枝
    if char not in node:
        return

    # 移动到下一个 Trie 节点
    node = node[char]

    # 如果找到一个单词
    if '#' in node:
        result.add(node['#'])

    # 标记当前单元格为已访问
    temp = board[i][j]
    board[i][j] = '#'

    # 向四个方向搜索
    dfs(i + 1, j, node)
    dfs(i - 1, j, node)
    dfs(i, j + 1, node)
    dfs(i, j - 1, node)

    # 回溯：恢复当前单元格
    board[i][j] = temp

    # 遍历网格中的每个单元格作为起点
    for i in range(m):
        for j in range(n):
            dfs(i, j, root)

    return list(result)

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    board1 = [
        ['o', 'a', 'a', 'n'],
        ['e', 't', 'a', 'e'],
    ]

```

```
[‘i’, ‘h’, ‘k’, ‘r’],  
[‘i’, ‘f’, ‘l’, ‘v’]  
]  
words1 = [“oath”, “pea”, “eat”, “rain”]  
result1 = solution.findWords(board1, words1)  
print(“输入:”)  
print(“board = [[o, a, a, n], [e, t, a, e], [i, h, k, r], [i, f, l, v]]]”)  
print(“words = [oath, pea, eat, rain]”)  
print(f“输出: {result1}”)
```

```
# 测试用例 2  
board2 = [[‘a’, ‘b’], [‘c’, ‘d’]]  
words2 = [“abcb”]  
result2 = solution.findWords(board2, words2)  
print(“\n 输入:”)  
print(“board = [[a, b], [c, d]]”)  
print(“words = [abcb]”)  
print(f“输出: {result2}”)
```

```
if __name__ == “__main__”:  
    main()
```

```
=====
```

文件: Code16\_Permutations.cpp

```
#include <vector>  
#include <iostream>  
  
using namespace std;  
  
/**  
 * LeetCode 46. 全排列  
 *  
 * 题目描述:  
 * 给定一个不含重复数字的数组 nums，返回其所有可能的全排列。你可以按任意顺序返回答案。  
 *  
 * 示例:  
 * 输入: nums = [1, 2, 3]  
 * 输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]  
 *  
 * 输入: nums = [0, 1]
```

```

* 输出: [[0, 1], [1, 0]]
*
* 输入: nums = [1]
* 输出: [[1]]
*
* 提示:
* 1 <= nums.length <= 6
* -10 <= nums[i] <= 10
* nums 中的所有整数互不相同
*
* 链接: https://leetcode.cn/problems/permutations/
*/

```

```

class Solution {
public:
    /**
     * 生成数组的所有可能全排列
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有排列
     * 2. 使用一个 used 数组来标记每个数字是否已经被使用
     * 3. 对于每个位置, 尝试将未使用的数字放入当前位置
     * 4. 递归处理下一个位置
     * 5. 回溯时, 将当前数字标记为未使用, 尝试其他选择
     *
     * 时间复杂度: O(N * N!), 其中 N 是数组长度。生成 N! 个排列, 每个排列需要 O(N) 时间复制。
     * 空间复杂度: O(N), 递归栈的深度加上 used 数组的大小。
     *
     * @param nums 输入数组
     * @return 所有可能的全排列
     */
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> path;
        vector<bool> used(nums.size(), false);

        backtrack(nums, used, path, result);
        return result;
    }

private:
    /**
     * 回溯函数生成排列

```

```

*
 * @param nums 输入数组
 * @param used 标记数组，记录数字是否已使用
 * @param path 当前排列路径
 * @param result 结果列表
 */
void backtrack(vector<int>& nums, vector<bool>& used, vector<int>& path, vector<vector<int>>&
result) {
    // 终止条件：当前排列长度等于数组长度
    if (path.size() == nums.size()) {
        result.push_back(path);
        return;
    }

    // 尝试将每个未使用的数字放入当前位置
    for (int i = 0; i < nums.size(); i++) {
        if (used[i]) {
            continue; // 跳过已使用的数字
        }

        // 选择当前数字
        used[i] = true;
        path.push_back(nums[i]);

        // 递归处理下一个位置
        backtrack(nums, used, path, result);

        // 回溯：撤销选择
        path.pop_back();
        used[i] = false;
    }
}

// 辅助函数：打印结果
void printResult(const vector<vector<int>>& result) {
    cout << "[";
    for (size_t i = 0; i < result.size(); i++) {
        cout << "[";
        for (size_t j = 0; j < result[i].size(); j++) {
            cout << result[i][j];
            if (j < result[i].size() - 1) {
                cout << ", ";
            }
        }
        cout << "]";
    }
    cout << "]";
}

```

```

        }
    }

    cout << "]";
    if (i < result.size() - 1) {
        cout << ", ";
    }
}

cout << "]" << endl;
}

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

    // 测试用例 1
    vector<int> nums1 = {1, 2, 3};
    vector<vector<int>> result1 = solution.permute(nums1);
    cout << "输入: nums = [1,2,3]" << endl;
    cout << "输出: ";
    printResult(result1);

    // 测试用例 2
    vector<int> nums2 = {0, 1};
    vector<vector<int>> result2 = solution.permute(nums2);
    cout << "\n 输入: nums = [0,1]" << endl;
    cout << "输出: ";
    printResult(result2);

    // 测试用例 3
    vector<int> nums3 = {1};
    vector<vector<int>> result3 = solution.permute(nums3);
    cout << "\n 输入: nums = [1]" << endl;
    cout << "输出: ";
    printResult(result3);

    return 0;
}

```

=====

文件: Code16\_Permutations.java

=====

```
package class038;
```

```
import java.util.*;

/**
 * LeetCode 46. 全排列
 *
 * 题目描述:
 * 给定一个不含重复数字的数组 nums，返回其所有可能的全排列。你可以按任意顺序返回答案。
 *
 * 示例:
 * 输入: nums = [1,2,3]
 * 输出: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
 *
 * 输入: nums = [0,1]
 * 输出: [[0,1], [1,0]]
 *
 * 输入: nums = [1]
 * 输出: [[1]]
 *
 * 提示:
 * 1 <= nums.length <= 6
 * -10 <= nums[i] <= 10
 * nums 中的所有整数互不相同
 *
 * 链接: https://leetcode.cn/problems/permutations/
 */
public class Code16_Permutations {

    /**
     * 生成数组的所有可能全排列
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有排列
     * 2. 使用一个 used 数组来标记每个数字是否已经被使用
     * 3. 对于每个位置，尝试将未使用的数字放入当前位置
     * 4. 递归处理下一个位置
     * 5. 回溯时，将当前数字标记为未使用，尝试其他选择
     *
     * 时间复杂度: O(N * N!)，其中 N 是数组长度。生成 N! 个排列，每个排列需要 O(N) 时间复制。
     * 空间复杂度: O(N)，递归栈的深度加上 used 数组的大小。
     *
     * @param nums 输入数组
     * @return 所有可能的全排列
}
```

```

*/
public static List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums == null || nums.length == 0) {
        return result;
    }

    boolean[] used = new boolean[nums.length]; // 标记数字是否已使用
    List<Integer> path = new ArrayList<>(); // 当前排列路径

    backtrack(nums, used, path, result);
    return result;
}

/**
 * 回溯函数生成排列
 *
 * @param nums 输入数组
 * @param used 标记数组，记录数字是否已使用
 * @param path 当前排列路径
 * @param result 结果列表
 */
private static void backtrack(int[] nums, boolean[] used, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：当前排列长度等于数组长度
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path)); // 深拷贝当前路径
        return;
    }

    // 尝试将每个未使用的数字放入当前位置
    for (int i = 0; i < nums.length; i++) {
        if (used[i]) {
            continue; // 跳过已使用的数字
        }

        // 选择当前数字
        used[i] = true;
        path.add(nums[i]);

        // 递归处理下一个位置
        backtrack(nums, used, path, result);
    }
}

```

```

// 回溯：撤销选择
path.remove(path.size() - 1);
used[i] = false;
}
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 2, 3};
    List<List<Integer>> result1 = permute(nums1);
    System.out.println("输入: nums = [1, 2, 3]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] nums2 = {0, 1};
    List<List<Integer>> result2 = permute(nums2);
    System.out.println("\n 输入: nums = [0, 1]");
    System.out.println("输出: " + result2);

    // 测试用例 3
    int[] nums3 = {1};
    List<List<Integer>> result3 = permute(nums3);
    System.out.println("\n 输入: nums = [1]");
    System.out.println("输出: " + result3);
}
}

```

文件: Code16\_Permutations.py

"""

LeetCode 46. 全排列

题目描述:

给定一个不含重复数字的数组 `nums`, 返回其所有可能的全排列。你可以按任意顺序返回答案。

示例:

输入: `nums = [1, 2, 3]`

输出: `[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]`

输入: `nums = [0, 1]`

输出: [[0, 1], [1, 0]]

输入: nums = [1]

输出: [[1]]

提示:

1 <= nums.length <= 6

-10 <= nums[i] <= 10

nums 中的所有整数互不相同

链接: <https://leetcode.cn/problems/permutations/>

"""

class Solution:

def permute(self, nums):

"""

生成数组的所有可能全排列

算法思路:

1. 使用回溯算法生成所有排列
2. 使用一个 used 列表来标记每个数字是否已经被使用
3. 对于每个位置，尝试将未使用的数字放入当前位置
4. 递归处理下一个位置
5. 回溯时，将当前数字标记为未使用，尝试其他选择

时间复杂度:  $O(N * N!)$ ，其中  $N$  是数组长度。生成  $N!$  个排列，每个排列需要  $O(N)$  时间复制。

空间复杂度:  $O(N)$ ，递归栈的深度加上 used 列表的大小。

:param nums: 输入数组

:return: 所有可能的全排列

"""

result = []

used = [False] \* len(nums)

path = []

self.backtrack(nums, used, path, result)

return result

def backtrack(self, nums, used, path, result):

"""

回溯函数生成排列

:param nums: 输入数组

```
:param used: 标记列表, 记录数字是否已使用
:param path: 当前排列路径
:param result: 结果列表
"""

# 终止条件: 当前排列长度等于数组长度
if len(path) == len(nums):
    result.append(path[:]) # 深拷贝当前路径
    return

# 尝试将每个未使用的数字放入当前位置
for i in range(len(nums)):
    if used[i]:
        continue # 跳过已使用的数字

    # 选择当前数字
    used[i] = True
    path.append(nums[i])

    # 递归处理下一个位置
    self.backtrack(nums, used, path, result)

    # 回溯: 撤销选择
    path.pop()
    used[i] = False

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 2, 3]
    result1 = solution.permute(nums1)
    print(f'输入: nums = [1, 2, 3]')
    print(f'输出: {result1}')

    # 测试用例 2
    nums2 = [0, 1]
    result2 = solution.permute(nums2)
    print(f'\n输入: nums = [0, 1]')
    print(f'输出: {result2}')

    # 测试用例 3
```

```
nums3 = [1]
result3 = solution.permute(nums3)
print(f'\n输入: nums = [1]')
print(f'输出: {result3}')
```

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

=====

文件: Code17\_CombinationSum.cpp

=====

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

/**
 * LeetCode 39. 组合总和
 *
 * 题目描述:
 * 给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ，找出 candidates 中可以使数字和为目标数 target 的所有不同组合 ，
 * 并以列表形式返回。你可以按 任意顺序 返回这些组合。
 * candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。
 *
 * 对于给定的输入，保证和为 target 的不同组合数少于 150 个。
 *
 * 示例:
 * 输入: candidates = [2, 3, 6, 7], target = 7
 * 输出: [[2, 2, 3], [7]]
 *
 * 输入: candidates = [2, 3, 5], target = 8
 * 输出: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]
 *
 * 输入: candidates = [2], target = 1
 * 输出: []
 *
 * 提示:
 * 1 <= candidates.length <= 30
```

```

* 2 <= candidates[i] <= 40
* candidates 的所有元素 互不相同
* 1 <= target <= 40
*
* 链接: https://leetcode.cn/problems/combination-sum/
*/

```

class Solution {

public:

```

    /**
     * 找出所有和为 target 的组合
     *
     * 算法思路:
     * 1. 使用回溯算法生成所有可能的组合
     * 2. 由于每个数字可以重复使用, 所以递归时从当前索引开始, 而不是下一个索引
     * 3. 为了避免重复组合, 对数组进行排序, 并按顺序选取元素
     * 4. 剪枝: 当当前和大于 target 时, 停止当前路径的探索
     *
     * 时间复杂度: O(N^(T/M)), 其中 N 是数组长度, T 是 target 值, M 是数组中的最小元素。最坏情况下需要探索的组合数约为 N^(T/M)。
     * 空间复杂度: O(T/M), 递归栈的最大深度。
     *
     * @param candidates 候选数组
     * @param target 目标和
     * @return 所有可能的组合
    */

```

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

```

    vector<vector<int>> result;
    vector<int> path;

    // 排序以便剪枝优化
    sort(candidates.begin(), candidates.end());

    backtrack(candidates, 0, target, 0, path, result);
    return result;
}
```

private:

```

    /**
     * 回溯函数生成组合
     *
     * @param candidates 候选数组
     * @param start 当前选择的起始索引
    */

```

```

* @param target 目标和
* @param currentSum 当前和
* @param path 当前组合路径
* @param result 结果列表
*/
void backtrack(vector<int>& candidates, int start, int target, int currentSum,
              vector<int>& path, vector<vector<int>>& result) {
    // 终止条件：找到一个有效组合
    if (currentSum == target) {
        result.push_back(path);
        return;
    }

    // 从 start 开始选择元素，避免重复组合
    for (int i = start; i < candidates.size(); i++) {
        // 剪枝：如果当前元素已经使得和超过 target，由于数组已排序，后面的元素更大，直接跳过
        if (currentSum + candidates[i] > target) {
            break;
        }

        // 选择当前元素
        path.push_back(candidates[i]);

        // 递归：由于元素可以重复使用，所以起始索引仍然是 i
        backtrack(candidates, i, target, currentSum + candidates[i], path, result);

        // 回溯：撤销选择
        path.pop_back();
    }
}

// 辅助函数：打印结果
void printResult(const vector<vector<int>>& result) {
    cout << "[";
    for (size_t i = 0; i < result.size(); i++) {
        cout << "[";
        for (size_t j = 0; j < result[i].size(); j++) {
            cout << result[i][j];
            if (j < result[i].size() - 1) {
                cout << ", ";
            }
        }
        cout << "]";
    }
    cout << "]";
}

```

```
cout << "]";
if (i < result.size() - 1) {
    cout << ", ";
}
cout << "]" << endl;
}

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

    // 测试用例 1
    vector<int> candidates1 = {2, 3, 6, 7};
    int target1 = 7;
    vector<vector<int>> result1 = solution.combinationSum(candidates1, target1);
    cout << "输入: candidates = [2,3,6,7], target = 7" << endl;
    cout << "输出: ";
    printResult(result1);

    // 测试用例 2
    vector<int> candidates2 = {2, 3, 5};
    int target2 = 8;
    vector<vector<int>> result2 = solution.combinationSum(candidates2, target2);
    cout << "\n 输入: candidates = [2,3,5], target = 8" << endl;
    cout << "输出: ";
    printResult(result2);

    // 测试用例 3
    vector<int> candidates3 = {2};
    int target3 = 1;
    vector<vector<int>> result3 = solution.combinationSum(candidates3, target3);
    cout << "\n 输入: candidates = [2], target = 1" << endl;
    cout << "输出: ";
    printResult(result3);

    return 0;
}
```

=====

文件: Code17\_CombinationSum.java

=====

```
package class038;

import java.util.*;

/**
 * LeetCode 39. 组合总和
 *
 * 题目描述:
 * 给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ，找出 candidates 中可以使数字和为目标数 target 的所有不同组合 ，
 * 并以列表形式返回。你可以按 任意顺序 返回这些组合。
 * candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。
 *
 * 对于给定的输入，保证和为 target 的不同组合数少于 150 个。
 *
 * 示例:
 * 输入: candidates = [2, 3, 6, 7], target = 7
 * 输出: [[2, 2, 3], [7]]
 * 解释:
 * 2 和 3 可以形成一组候选， $2 + 2 + 3 = 7$  。注意 2 可以使用多次。
 * 7 也是一个候选， $7 = 7$  。
 * 仅有这两种组合。
 *
 * 输入: candidates = [2, 3, 5], target = 8
 * 输出: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]
 *
 * 输入: candidates = [2], target = 1
 * 输出: []
 *
 * 提示:
 *  $1 \leq \text{candidates.length} \leq 30$ 
 *  $2 \leq \text{candidates}[i] \leq 40$ 
 * candidates 的所有元素 互不相同
 *  $1 \leq \text{target} \leq 40$ 
 *
 * 链接: https://leetcode.cn/problems/combination-sum/
 */
public class Code17_CombinationSum {

    /**
     * 找出所有和为 target 的组合
     *
```

```

* 算法思路:
* 1. 使用回溯算法生成所有可能的组合
* 2. 由于每个数字可以重复使用, 所以递归时从当前索引开始, 而不是下一个索引
* 3. 为了避免重复组合, 对数组进行排序, 并按顺序选取元素
* 4. 剪枝: 当当前和大于 target 时, 停止当前路径的探索
*
* 时间复杂度:  $O(N^{(T/M)})$ , 其中 N 是数组长度, T 是 target 值, M 是数组中的最小元素。最坏情况下需要探索的组合数约为  $N^{(T/M)}$ 。
* 空间复杂度:  $O(T/M)$ , 递归栈的最大深度。
*
* @param candidates 候选数组
* @param target 目标和
* @return 所有可能的组合
*/
public static List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    if (candidates == null || candidates.length == 0) {
        return result;
    }

    // 排序以便剪枝优化
    Arrays.sort(candidates);
    List<Integer> path = new ArrayList<>();

    backtrack(candidates, 0, target, 0, path, result);
    return result;
}

/**
* 回溯函数生成组合
*
* @param candidates 候选数组
* @param start 当前选择的起始索引
* @param target 目标和
* @param currentSum 当前和
* @param path 当前组合路径
* @param result 结果列表
*/
private static void backtrack(int[] candidates, int start, int target, int currentSum,
                             List<Integer> path, List<List<Integer>> result) {
    // 终止条件: 找到一个有效组合
    if (currentSum == target) {
        result.add(new ArrayList<>(path));
    }
}

```

```
    return;
}

// 从 start 开始选择元素，避免重复组合
for (int i = start; i < candidates.length; i++) {
    // 剪枝：如果当前元素已经使得和超过 target，由于数组已排序，后面的元素更大，直接跳过
    if (currentSum + candidates[i] > target) {
        break;
    }

    // 选择当前元素
    path.add(candidates[i]);
    currentSum += candidates[i];

    // 递归：由于元素可以重复使用，所以起始索引仍然是 i
    backtrack(candidates, i, target, currentSum, path, result);

    // 回溯：撤销选择
    currentSum -= candidates[i];
    path.remove(path.size() - 1);
}
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] candidates1 = {2, 3, 6, 7};
    int target1 = 7;
    List<List<Integer>> result1 = combinationSum(candidates1, target1);
    System.out.println("输入: candidates = [2,3,6,7], target = 7");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] candidates2 = {2, 3, 5};
    int target2 = 8;
    List<List<Integer>> result2 = combinationSum(candidates2, target2);
    System.out.println("\n输入: candidates = [2,3,5], target = 8");
    System.out.println("输出: " + result2);

    // 测试用例 3
    int[] candidates3 = {2};
    int target3 = 1;
    List<List<Integer>> result3 = combinationSum(candidates3, target3);
}
```

```
        System.out.println("\n输入: candidates = [2], target = 1");
        System.out.println("输出: " + result3);
    }
}
```

=====

文件: Code17\_CombinationSum.py

=====

"""

## LeetCode 39. 组合总和

题目描述:

给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ，找出 candidates 中可以使数字和为目标数 target 的所有不同组合 ，

并以列表形式返回。你可以按 任意顺序 返回这些组合。

candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 target 的不同组合数少于 150 个。

示例:

输入: candidates = [2, 3, 6, 7], target = 7

输出: [[2, 2, 3], [7]]

输入: candidates = [2, 3, 5], target = 8

输出: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]

输入: candidates = [2], target = 1

输出: []

提示:

$1 \leq \text{candidates.length} \leq 30$

$2 \leq \text{candidates}[i] \leq 40$

candidates 的所有元素 互不相同

$1 \leq \text{target} \leq 40$

链接: <https://leetcode.cn/problems/combination-sum/>

"""

class Solution:

```
    def combinationSum(self, candidates, target):
```

"""

找出所有和为 target 的组合

算法思路：

1. 使用回溯算法生成所有可能的组合
2. 由于每个数字可以重复使用，所以递归时从当前索引开始，而不是下一个索引
3. 为了避免重复组合，对数组进行排序，并按顺序选取元素
4. 剪枝：当当前和大于 target 时，停止当前路径的探索

时间复杂度： $O(N^{\lceil T/M \rceil})$ ，其中 N 是数组长度，T 是 target 值，M 是数组中的最小元素。最坏情况下需要探索的组合数约为  $N^{\lceil T/M \rceil}$ 。

空间复杂度： $O(T/M)$ ，递归栈的最大深度。

```
:param candidates: 候选数组
:param target: 目标和
:return: 所有可能的组合
"""

result = []
path = []

# 排序以便剪枝优化
candidates.sort()

self.backtrack(candidates, 0, target, 0, path, result)
return result

def backtrack(self, candidates, start, target, current_sum, path, result):
    """
    回溯函数生成组合

    :param candidates: 候选数组
    :param start: 当前选择的起始索引
    :param target: 目标和
    :param current_sum: 当前和
    :param path: 当前组合路径
    :param result: 结果列表
    """

    # 终止条件：找到一个有效组合
    if current_sum == target:
        result.append(path[:])  # 深拷贝当前路径
        return

    # 从 start 开始选择元素，避免重复组合
    for i in range(start, len(candidates)):
```

```
# 剪枝: 如果当前元素已经使得和超过 target, 由于数组已排序, 后面的元素更大, 直接跳过
if current_sum + candidates[i] > target:
    break

# 选择当前元素
path.append(candidates[i])

# 递归: 由于元素可以重复使用, 所以起始索引仍然是 i
self.backtrack(candidates, i, target, current_sum + candidates[i], path, result)

# 回溯: 撤销选择
path.pop()

# 测试方法
def main():
    solution = Solution()

    # 测试用例 1
    candidates1 = [2, 3, 6, 7]
    target1 = 7
    result1 = solution.combinationSum(candidates1, target1)
    print(f'输入: candidates = [2, 3, 6, 7], target = 7')
    print(f'输出: {result1}')

    # 测试用例 2
    candidates2 = [2, 3, 5]
    target2 = 8
    result2 = solution.combinationSum(candidates2, target2)
    print(f'\n输入: candidates = [2, 3, 5], target = 8')
    print(f'输出: {result2}')

    # 测试用例 3
    candidates3 = [2]
    target3 = 1
    result3 = solution.combinationSum(candidates3, target3)
    print(f'\n输入: candidates = [2], target = 1')
    print(f'输出: {result3}')

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

文件: Code18\_SubsetsII.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

/***
 * LeetCode 90. 子集 II
 *
 * 题目描述:
 * 给你一个整数数组 nums，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。
 *
 * 示例:
 * 输入: nums = [1, 2, 2]
 * 输出: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]
 *
 * 输入: nums = [0]
 * 输出: [[], [0]]
 *
 * 提示:
 * 1 <= nums.length <= 10
 * -10 <= nums[i] <= 10
 *
 * 链接: https://leetcode.cn/problems/subsets-ii/
 *
 * 算法思路:
 * 1. 先对数组进行排序，使相同元素相邻
 * 2. 使用回溯算法生成所有子集
 * 3. 在回溯过程中，对于重复元素，只选择第一个出现的，跳过后续相同的元素
 * 4. 这样可以避免生成重复的子集
 *
 * 时间复杂度: O(n * 2^n)，其中 n 是数组长度，共有  $2^n$  个子集，每个子集需要 O(n) 时间复制
 * 空间复杂度: O(n)，递归栈深度和存储路径的空间
 */
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
```

```

vector<vector<int>> result;
vector<int> path;
sort(nums.begin(), nums.end());
backtrack(nums, 0, path, result);
return result;
}

private:
void backtrack(vector<int>& nums, int start, vector<int>& path, vector<vector<int>>& result)
{
    // 每一步都添加到结果中
    result.push_back(path);

    // 从 start 开始遍历，避免重复
    for (int i = start; i < nums.size(); i++) {
        // 跳过重复元素：如果当前元素与前一个相同且不是第一个出现的，则跳过
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }

        path.push_back(nums[i]); // 选择当前元素
        backtrack(nums, i + 1, path, result); // 递归处理下一个元素
        path.pop_back(); // 撤销选择
    }
}
};

// 测试函数
void testSubsetsII() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 2, 2};
    vector<vector<int>> result1 = solution.subsetsWithDup(nums1);
    cout << "输入: nums = [1, 2, 2]" << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << "[";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << result1[i][j];
            if (j < result1[i].size() - 1) cout << ",";
        }
        cout << "]";
    }
}

```

```

    if (i < result1.size() - 1) cout << ", ";
}

cout << "]" << endl;

// 测试用例 2
vector<int> nums2 = {0};
vector<vector<int>> result2 = solution.subsetsWithDup(nums2);
cout << "\n 输入: nums = [0]" << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "[";
    for (int j = 0; j < result2[i].size(); j++) {
        cout << result2[i][j];
        if (j < result2[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

int main() {
    testSubsetsII();
    return 0;
}

```

=====

文件: Code18\_SubsetsII.java

=====

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * LeetCode 90. 子集 II
 *
 * 题目描述:
 * 给你一个整数数组 nums，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。
 *
 * 示例:
 * 输入: nums = [1, 2, 2]

```

```
* 输出: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]
*
* 输入: nums = [0]
* 输出: [[], [0]]
*
* 提示:
* 1 <= nums.length <= 10
* -10 <= nums[i] <= 10
*
* 链接: https://leetcode.cn/problems/subsets-ii/
*
* 算法思路:
* 1. 先对数组进行排序，使相同元素相邻
* 2. 使用回溯算法生成所有子集
* 3. 在回溯过程中，对于重复元素，只选择第一个出现的，跳过后续相同的元素
* 4. 这样可以避免生成重复的子集
*
* 时间复杂度: O(n * 2^n)，其中 n 是数组长度，共有 2^n 个子集，每个子集需要 O(n) 时间复制
* 空间复杂度: O(n)，递归栈深度和存储路径的空间
*/

```

```
public class Code18_SubsetsII {
```

```
    /**
     * 生成包含重复元素的数组的所有不重复子集
     *
     * @param nums 输入数组（可能包含重复元素）
     * @return 所有不重复的子集
     */
    public static List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        // 先排序，使相同元素相邻
        Arrays.sort(nums);
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }
```

```
    /**
     * 回溯函数生成子集
     *
     * @param nums 输入数组
     * @param start 当前起始位置
     * @param path 当前路径
     * @param result 结果列表
     */
```

```

*/
private static void backtrack(int[] nums, int start, List<Integer> path, List<List<Integer>>
result) {
    // 每一步都添加到结果中
    result.add(new ArrayList<>(path));

    // 从 start 开始遍历，避免重复
    for (int i = start; i < nums.length; i++) {
        // 跳过重复元素：如果当前元素与前一个相同且不是第一个出现的，则跳过
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }

        path.add(nums[i]); // 选择当前元素
        backtrack(nums, i + 1, path, result); // 递归处理下一个元素
        path.remove(path.size() - 1); // 撤销选择
    }
}

/**
 * 解法二：使用计数法处理重复元素
 * 对于重复元素，我们可以选择 0 个、1 个、2 个... 直到所有重复元素
 *
 * @param nums 输入数组
 * @return 所有不重复的子集
 */
public static List<List<Integer>> subsetsWithDup2(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(nums);
    backtrack2(nums, 0, new ArrayList<>(), result);
    return result;
}

private static void backtrack2(int[] nums, int start, List<Integer> path, List<List<Integer>>
result) {
    if (start == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 统计当前元素出现的次数
    int count = 1;
    int i = start + 1;

```

```

while (i < nums.length && nums[i] == nums[start]) {
    count++;
    i++;
}

// 对于当前元素，可以选择 0 个、1 个、2 个...count 个
for (int j = 0; j <= count; j++) {
    // 添加 j 个当前元素
    for (int k = 0; k < j; k++) {
        path.add(nums[start]);
    }
}

backtrack2(nums, start + count, path, result);

// 回溯，移除添加的元素
for (int k = 0; k < j; k++) {
    path.remove(path.size() - 1);
}
}

}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 2, 2};
    List<List<Integer>> result1 = subsetsWithDup(nums1);
    System.out.println("输入: nums = [1, 2, 2]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] nums2 = {0};
    List<List<Integer>> result2 = subsetsWithDup(nums2);
    System.out.println("\n输入: nums = [0]");
    System.out.println("输出: " + result2);

    // 测试用例 3
    int[] nums3 = {1, 1, 2, 2};
    List<List<Integer>> result3 = subsetsWithDup(nums3);
    System.out.println("\n输入: nums = [1, 1, 2, 2]");
    System.out.println("输出: " + result3);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ====");
}

```

```
    List<List<Integer>> result4 = subsetsWithDup2(nums1);
    System.out.println("输入: nums = [1, 2, 2]");
    System.out.println("输出: " + result4);
}
=====
```

文件: Code18\_SubsetsII.py

```
=====
from typing import List
```

```
class Solution:
```

```
    """

```

```
    LeetCode 90. 子集 II
```

题目描述:

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。  
解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。

示例:

输入: `nums = [1, 2, 2]`

输出: `[[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]`

输入: `nums = [0]`

输出: `[[], [0]]`

提示:

`1 <= nums.length <= 10`

`-10 <= nums[i] <= 10`

链接: <https://leetcode.cn/problems/subsets-ii/>

算法思路:

1. 先对数组进行排序，使相同元素相邻
2. 使用回溯算法生成所有子集
3. 在回溯过程中，对于重复元素，只选择第一个出现的，跳过后续相同的元素
4. 这样可以避免生成重复的子集

时间复杂度:  $O(n * 2^n)$ ，其中  $n$  是数组长度，共有  $2^n$  个子集，每个子集需要  $O(n)$  时间复制

空间复杂度:  $O(n)$ ，递归栈深度和存储路径的空间

```
"""

```

```

def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
    result = []
    nums.sort() # 先排序，使相同元素相邻
    self.backtrack(nums, 0, [], result)
    return result

def backtrack(self, nums: List[int], start: int, path: List[int], result: List[List[int]]) ->
None:
    # 每一步都添加到结果中
    result.append(path[:])

    # 从 start 开始遍历，避免重复
    for i in range(start, len(nums)):
        # 跳过重复元素：如果当前元素与前一个相同且不是第一个出现的，则跳过
        if i > start and nums[i] == nums[i - 1]:
            continue

        path.append(nums[i]) # 选择当前元素
        self.backtrack(nums, i + 1, path, result) # 递归处理下一个元素
        path.pop() # 撤销选择

def test_subsets_ii():
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 2, 2]
    result1 = solution.subsetsWithDup(nums1)
    print("输入: nums = [1, 2, 2]")
    print("输出:", result1)

    # 测试用例 2
    nums2 = [0]
    result2 = solution.subsetsWithDup(nums2)
    print("\n 输入: nums = [0]")
    print("输出:", result2)

    # 测试用例 3
    nums3 = [1, 1, 2, 2]
    result3 = solution.subsetsWithDup(nums3)
    print("\n 输入: nums = [1, 1, 2, 2]")
    print("输出:", result3)

if __name__ == "__main__":

```

```
test_subsets_i()
```

```
=====
```

文件: Code19\_Combinations.cpp

```
#include <iostream>
```

```
#include <vector>
```

```
#include <string>
```

```
using namespace std;
```

```
/**
```

```
* LeetCode 77. 组合
```

```
*
```

```
* 题目描述:
```

```
* 给定两个整数 n 和 k，返回范围 [1, n] 中所有可能的 k 个数的组合。
```

```
* 你可以按任何顺序返回答案。
```

```
*
```

```
* 示例:
```

```
* 输入: n = 4, k = 2
```

```
* 输出: [[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

```
*
```

```
* 输入: n = 1, k = 1
```

```
* 输出: [[1]]
```

```
*
```

```
* 提示:
```

```
* 1 <= n <= 20
```

```
* 1 <= k <= n
```

```
*
```

```
* 链接: https://leetcode.cn/problems/combinations/
```

```
*
```

```
* 算法思路:
```

```
* 1. 使用回溯算法生成所有可能的组合
```

```
* 2. 从 1 开始，每次选择一个数字，然后递归选择下一个数字
```

```
* 3. 当组合长度达到 k 时，将其加入结果集
```

```
* 4. 通过控制起始位置避免重复组合
```

```
*
```

```
* 时间复杂度: O(C(n, k) * k)，其中 C(n, k) 是组合数，每个组合需要 O(k) 时间复制
```

```
* 空间复杂度: O(k)，递归栈深度和存储路径的空间
```

```
*/
```

```
class Solution {
```

```
public:
```

```

vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> result;
    vector<int> path;
    backtrack(n, k, 1, path, result);
    return result;
}

private:
void backtrack(int n, int k, int start, vector<int>& path, vector<vector<int>>& result) {
    // 终止条件: 组合长度达到 k
    if (path.size() == k) {
        result.push_back(path);
        return;
    }

    // 剪枝优化: 如果剩余的数字不足以填满组合, 提前终止
    // 还需要选择的数字个数: k - path.size()
    // 从 start 到 n 至少要有这么多个数字: n - start + 1 >= k - path.size()
    // 所以 start <= n - (k - path.size()) + 1
    for (int i = start; i <= n - (k - path.size()) + 1; i++) {
        path.push_back(i); // 选择当前数字
        backtrack(n, k, i + 1, path, result); // 递归选择下一个数字
        path.pop_back(); // 撤销选择
    }
}
};

// 测试函数
void testCombinations() {
    Solution solution;

    // 测试用例 1
    int n1 = 4, k1 = 2;
    vector<vector<int>> result1 = solution.combine(n1, k1);
    cout << "输入: n = " << n1 << ", k = " << k1 << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << "[";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << result1[i][j];
            if (j < result1[i].size() - 1) cout << ",";
        }
        cout << "]";
    }
}

```

```

    if (i < result1.size() - 1) cout << ", ";
}

cout << "]" << endl;

// 测试用例 2
int n2 = 1, k2 = 1;
vector<vector<int>> result2 = solution.combine(n2, k2);
cout << "\n 输入: n = " << n2 << ", k = " << k2 << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "[";
    for (int j = 0; j < result2[i].size(); j++) {
        cout << result2[i][j];
        if (j < result2[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

int main() {
    testCombinations();
    return 0;
}

```

=====

文件: Code19\_Combinations.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 77. 组合
 *
 * 题目描述:
 * 给定两个整数 n 和 k，返回范围 [1, n] 中所有可能的 k 个数的组合。
 * 你可以按任何顺序返回答案。
 *
 * 示例:

```

```

* 输入: n = 4, k = 2
* 输出: [[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
*
* 输入: n = 1, k = 1
* 输出: [[1]]
*
* 提示:
* 1 <= n <= 20
* 1 <= k <= n
*
* 链接: https://leetcode.cn/problems/combinations/
*
* 算法思路:
* 1. 使用回溯算法生成所有可能的组合
* 2. 从 1 开始, 每次选择一个数字, 然后递归选择下一个数字
* 3. 当组合长度达到 k 时, 将其加入结果集
* 4. 通过控制起始位置避免重复组合
*
* 时间复杂度: O(C(n, k) * k), 其中 C(n, k) 是组合数, 每个组合需要 O(k) 时间复制
* 空间复杂度: O(k), 递归栈深度和存储路径的空间
*/
public class Code19_Combinations {

    /**
     * 生成从 1 到 n 中所有 k 个数的组合
     *
     * @param n 范围上限
     * @param k 组合大小
     * @return 所有可能的组合
     */
    public static List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(n, k, 1, new ArrayList<>(), result);
        return result;
    }

    /**
     * 回溯函数生成组合
     *
     * @param n 范围上限
     * @param k 组合大小
     * @param start 当前起始数字
     * @param path 当前路径
     */
    private void backtrack(int n, int k, int start, List<Integer> path, List<List<Integer>> result) {
        if (path.size() == k) {
            result.add(new ArrayList<Integer>(path));
            return;
        }
        for (int i = start; i <= n; i++) {
            path.add(i);
            backtrack(n, k, i + 1, path, result);
            path.remove(path.size() - 1);
        }
    }
}

```

```

* @param result 结果列表
*/
private static void backtrack(int n, int k, int start, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：组合长度达到 k
    if (path.size() == k) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 剪枝优化：如果剩余的数字不足以填满组合，提前终止
    // 还需要选择的数字个数：k - path.size()
    // 从 start 到 n 至少要有这么多个数字：n - start + 1 >= k - path.size()
    // 所以 start <= n - (k - path.size()) + 1
    for (int i = start; i <= n - (k - path.size()) + 1; i++) {
        path.add(i); // 选择当前数字
        backtrack(n, k, i + 1, path, result); // 递归选择下一个数字
        path.remove(path.size() - 1); // 撤销选择
    }
}

/**
 * 解法二：使用迭代法生成组合
 * 使用字典序组合生成算法
 *
 * @param n 范围上限
 * @param k 组合大小
 * @return 所有可能的组合
 */
public static List<List<Integer>> combineIterative(int n, int k) {
    List<List<Integer>> result = new ArrayList<>();
    // 初始化第一个组合
    List<Integer> combination = new ArrayList<>();
    for (int i = 1; i <= k; i++) {
        combination.add(i);
    }

    int i = k - 1;
    while (i >= 0) {
        // 添加当前组合
        result.add(new ArrayList<>(combination));
        // 寻找下一个组合

```

```
        if (combination.get(i) < n - (k - 1 - i)) {
            combination.set(i, combination.get(i) + 1);
            for (int j = i + 1; j < k; j++) {
                combination.set(j, combination.get(j - 1) + 1);
            }
        } else {
            i--;
        }
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 4, k1 = 2;
    List<List<Integer>> result1 = combine(n1, k1);
    System.out.println("输入: n = " + n1 + ", k = " + k1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int n2 = 1, k2 = 1;
    List<List<Integer>> result2 = combine(n2, k2);
    System.out.println("\n输入: n = " + n2 + ", k = " + k2);
    System.out.println("输出: " + result2);

    // 测试用例 3
    int n3 = 5, k3 = 3;
    List<List<Integer>> result3 = combine(n3, k3);
    System.out.println("\n输入: n = " + n3 + ", k = " + k3);
    System.out.println("输出: " + result3);

    // 测试迭代解法
    System.out.println("\n==== 迭代解法测试 ===");
    List<List<Integer>> result4 = combineIterative(n1, k1);
    System.out.println("输入: n = " + n1 + ", k = " + k1);
    System.out.println("输出: " + result4);
}
```

---

文件: Code19\_Combinations.py

```
=====
```

```
from typing import List
```

```
class Solution:
```

```
    """
```

```
    LeetCode 77. 组合
```

题目描述:

给定两个整数  $n$  和  $k$ , 返回范围  $[1, n]$  中所有可能的  $k$  个数的组合。

你可以按任何顺序返回答案。

示例:

输入:  $n = 4, k = 2$

输出:  $[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]$

输入:  $n = 1, k = 1$

输出:  $[[1]]$

提示:

$1 \leq n \leq 20$

$1 \leq k \leq n$

链接: <https://leetcode.cn/problems/combinations/>

算法思路:

1. 使用回溯算法生成所有可能的组合
2. 从 1 开始, 每次选择一个数字, 然后递归选择下一个数字
3. 当组合长度达到  $k$  时, 将其加入结果集
4. 通过控制起始位置避免重复组合

时间复杂度:  $O(C(n, k) * k)$ , 其中  $C(n, k)$  是组合数, 每个组合需要  $O(k)$  时间复制

空间复杂度:  $O(k)$ , 递归栈深度和存储路径的空间

```
"""
```

```
def combine(self, n: int, k: int) -> List[List[int]]:  
    result = []  
    self.backtrack(n, k, 1, [], result)  
    return result
```

```
def backtrack(self, n: int, k: int, start: int, path: List[int], result: List[List[int]]) ->
```

None:

```
# 终止条件: 组合长度达到 k
```

```

    if len(path) == k:
        result.append(path[:])
        return

    # 剪枝优化：如果剩余的数字不足以填满组合，提前终止
    # 还需要选择的数字个数: k - len(path)
    # 从 start 到 n 至少要有这么多个数字: n - start + 1 >= k - len(path)
    # 所以 i 的范围是 start 到 n - (k - len(path)) + 1
    for i in range(start, n - (k - len(path)) + 2):
        path.append(i) # 选择当前数字
        self.backtrack(n, k, i + 1, path, result) # 递归选择下一个数字
        path.pop() # 撤销选择

def test_combinations():
    solution = Solution()

    # 测试用例 1
    n1, k1 = 4, 2
    result1 = solution.combine(n1, k1)
    print(f"输入: n = {n1}, k = {k1}")
    print("输出:", result1)

    # 测试用例 2
    n2, k2 = 1, 1
    result2 = solution.combine(n2, k2)
    print(f"\n输入: n = {n2}, k = {k2}")
    print("输出:", result2)

    # 测试用例 3
    n3, k3 = 5, 3
    result3 = solution.combine(n3, k3)
    print(f"\n输入: n = {n3}, k = {k3}")
    print("输出:", result3)

if __name__ == "__main__":
    test_combinations()

```

=====

文件: Code20\_PermutationsII.cpp

=====

```
#include <iostream>
#include <vector>
```

```
#include <algorithm>

using namespace std;

/***
 * LeetCode 47. 全排列 II
 *
 * 题目描述:
 * 给定一个可包含重复数字的序列 nums ，按任意顺序返回所有不重复的全排列。
 *
 * 示例:
 * 输入: nums = [1,1,2]
 * 输出: [[1,1,2],[1,2,1],[2,1,1]]
 *
 * 输入: nums = [1,2,3]
 * 输出: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
 *
 * 提示:
 * 1 <= nums.length <= 8
 * -10 <= nums[i] <= 10
 *
 * 链接: https://leetcode.cn/problems/permutations-ii/
 *
 * 算法思路:
 * 1. 先对数组进行排序，使相同元素相邻
 * 2. 使用回溯算法生成所有排列
 * 3. 使用布尔数组标记已使用的元素
 * 4. 对于重复元素，确保相同元素的相对顺序，避免生成重复排列
 *
 * 时间复杂度: O(n * n!)，其中 n 是数组长度，共有 n! 个排列，每个排列需要 O(n) 时间复制
 * 空间复杂度: O(n)，递归栈深度和存储路径的空间
 */
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> path;
        vector<bool> used(nums.size(), false);
        sort(nums.begin(), nums.end());
        backtrack(nums, used, path, result);
        return result;
    }
}
```

```

private:
    void backtrack(vector<int>& nums, vector<bool>& used, vector<int>& path, vector<vector<int>>& result) {
        // 终止条件：排列长度等于数组长度
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            // 跳过已使用的元素
            if (used[i]) {
                continue;
            }

            // 去重关键：如果当前元素与前一个相同，且前一个元素未被使用，则跳过
            // 这样可以确保相同元素的相对顺序，避免生成重复排列
            if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
                continue;
            }

            used[i] = true;
            path.push_back(nums[i]);
            backtrack(nums, used, path, result);
            path.pop_back();
            used[i] = false;
        }
    }
};

// 测试函数
void testPermutationsII() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 1, 2};
    vector<vector<int>> result1 = solution.permuteUnique(nums1);
    cout << "输入: nums = [1, 1, 2]" << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << "[";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << result1[i][j];
        }
        cout << "]";
    }
}

```

```

        if (j < result1[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result1.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 2
vector<int> nums2 = {1, 2, 3};
vector<vector<int>> result2 = solution.permuteUnique(nums2);
cout << "\n 输入: nums = [1, 2, 3]" << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "[";
    for (int j = 0; j < result2[i].size(); j++) {
        cout << result2[i][j];
        if (j < result2[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

int main() {
    testPermutationsII();
    return 0;
}

```

=====

文件: Code20\_PermutationsII.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * LeetCode 47. 全排列 II
 *
 * 题目描述:

```

```

* 给定一个可包含重复数字的序列 nums，按任意顺序返回所有不重复的全排列。
*
* 示例：
* 输入：nums = [1,1,2]
* 输出：[[1,1,2],[1,2,1],[2,1,1]]
*
* 输入：nums = [1,2,3]
* 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
*
* 提示：
* 1 <= nums.length <= 8
* -10 <= nums[i] <= 10
*
* 链接：https://leetcode.cn/problems/permutations-ii/
*
* 算法思路：
* 1. 先对数组进行排序，使相同元素相邻
* 2. 使用回溯算法生成所有排列
* 3. 使用布尔数组标记已使用的元素
* 4. 对于重复元素，确保相同元素的相对顺序，避免生成重复排列
*
* 时间复杂度：O(n * n!)，其中 n 是数组长度，共有 n! 个排列，每个排列需要 O(n) 时间复制
* 空间复杂度：O(n)，递归栈深度和存储路径的空间
*/
public class Code20_PermutationsII {

    /**
     * 生成包含重复元素的数组的所有不重复全排列
     *
     * @param nums 输入数组（可能包含重复元素）
     * @return 所有不重复的全排列
     */
    public static List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        // 先排序，使相同元素相邻
        Arrays.sort(nums);
        boolean[] used = new boolean[nums.length];
        backtrack(nums, used, new ArrayList<>(), result);
        return result;
    }

    /**
     * 回溯函数生成排列

```

```

*
 * @param nums 输入数组
 * @param used 标记已使用元素的数组
 * @param path 当前路径
 * @param result 结果列表
 */
private static void backtrack(int[] nums, boolean[] used, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：排列长度等于数组长度
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 跳过已使用的元素
        if (used[i]) {
            continue;
        }

        // 去重关键：如果当前元素与前一个相同，且前一个元素未被使用，则跳过
        // 这样可以确保相同元素的相对顺序，避免生成重复排列
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }

        used[i] = true;
        path.add(nums[i]);
        backtrack(nums, used, path, result);
        path.remove(path.size() - 1);
        used[i] = false;
    }
}

/**
 * 解法二：使用交换元素的方式生成排列
 * 通过交换元素实现原地排列，减少空间使用
 *
 * @param nums 输入数组
 * @return 所有不重复的全排列
 */
public static List<List<Integer>> permuteUnique2(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();

```

```

        Arrays.sort(nums);
        backtrack2(nums, 0, result);
        return result;
    }

private static void backtrack2(int[] nums, int start, List<List<Integer>> result) {
    if (start == nums.length) {
        List<Integer> permutation = new ArrayList<>();
        for (int num : nums) {
            permutation.add(num);
        }
        result.add(permutation);
        return;
    }

    for (int i = start; i < nums.length; i++) {
        // 去重关键：如果当前元素与前面某个元素相同，且该元素已经被交换过，则跳过
        if (i != start && nums[i] == nums[start] && i > start) {
            continue;
        }

        // 检查是否应该交换
        boolean shouldSwap = true;
        for (int j = start; j < i; j++) {
            if (nums[j] == nums[i]) {
                shouldSwap = false;
                break;
            }
        }

        if (shouldSwap) {
            swap(nums, start, i);
            backtrack2(nums, start + 1, result);
            swap(nums, start, i);
        }
    }
}

private static void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}

```

```

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 1, 2};
    List<List<Integer>> result1 = permuteUnique(nums1);
    System.out.println("输入: nums = [1, 1, 2]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] nums2 = {1, 2, 3};
    List<List<Integer>> result2 = permuteUnique(nums2);
    System.out.println("\n 输入: nums = [1, 2, 3]");
    System.out.println("输出: " + result2);

    // 测试用例 3
    int[] nums3 = {2, 2, 1, 1};
    List<List<Integer>> result3 = permuteUnique(nums3);
    System.out.println("\n 输入: nums = [2, 2, 1, 1]");
    System.out.println("输出: " + result3);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ====");
    List<List<Integer>> result4 = permuteUnique2(nums1);
    System.out.println("输入: nums = [1, 1, 2]");
    System.out.println("输出: " + result4);
}

}

```

=====

文件: Code20\_PermutationsII.py

=====

```
from typing import List
```

```
class Solution:
```

```
    """

```

LeetCode 47. 全排列 II

题目描述:

给定一个可包含重复数字的序列 `nums`，按任意顺序返回所有不重复的全排列。

示例:

输入: nums = [1, 1, 2]

输出: [[1, 1, 2], [1, 2, 1], [2, 1, 1]]

输入: nums = [1, 2, 3]

输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

提示:

1 <= nums.length <= 8

-10 <= nums[i] <= 10

链接: <https://leetcode.cn/problems/permutations-ii/>

算法思路:

1. 先对数组进行排序，使相同元素相邻
2. 使用回溯算法生成所有排列
3. 使用布尔数组标记已使用的元素
4. 对于重复元素，确保相同元素的相对顺序，避免生成重复排列

时间复杂度:  $O(n * n!)$ ，其中  $n$  是数组长度，共有  $n!$  个排列，每个排列需要  $O(n)$  时间复制

空间复杂度:  $O(n)$ ，递归栈深度和存储路径的空间

"""

```
def permuteUnique(self, nums: List[int]) -> List[List[int]]:  
    result = []  
    nums.sort() # 先排序，使相同元素相邻  
    used = [False] * len(nums)  
    self.backtrack(nums, used, [], result)  
    return result
```

```
def backtrack(self, nums: List[int], used: List[bool], path: List[int], result:  
List[List[int]]) -> None:
```

# 终止条件：排列长度等于数组长度

```
if len(path) == len(nums):  
    result.append(path[:])  
    return
```

```
for i in range(len(nums)):
```

# 跳过已使用的元素

```
if used[i]:
```

```
    continue
```

# 去重关键：如果当前元素与前一个相同，且前一个元素未被使用，则跳过

# 这样可以确保相同元素的相对顺序，避免生成重复排列

```

    if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
        continue

    used[i] = True
    path.append(nums[i])
    self.backtrack(nums, used, path, result)
    path.pop()
    used[i] = False

def test_permutations_ii():
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 1, 2]
    result1 = solution.permuteUnique(nums1)
    print("输入: nums = [1, 1, 2]")
    print("输出:", result1)

    # 测试用例 2
    nums2 = [1, 2, 3]
    result2 = solution.permuteUnique(nums2)
    print("\n输入: nums = [1, 2, 3]")
    print("输出:", result2)

    # 测试用例 3
    nums3 = [2, 2, 1, 1]
    result3 = solution.permuteUnique(nums3)
    print("\n输入: nums = [2, 2, 1, 1]")
    print("输出:", result3)

if __name__ == "__main__":
    test_permutations_ii()

```

=====

文件: Code21\_CombinationSumII.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

```

```
/**  
 * LeetCode 40. 组合总和 II  
 *  
 * 题目描述:  
 * 给定一个候选人编号的集合 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。  
 * candidates 中的每个数字在每个组合中只能使用一次。  
 * 注意：解集不能包含重复的组合。  
 *  
 * 示例:  
 * 输入: candidates = [10, 1, 2, 7, 6, 1, 5], target = 8  
 * 输出: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]  
 *  
 * 输入: candidates = [2, 5, 2, 1, 2], target = 5  
 * 输出: [[1, 2, 2], [5]]  
 *  
 * 提示:  
 * 1 <= candidates.length <= 100  
 * 1 <= candidates[i] <= 50  
 * 1 <= target <= 30  
 *  
 * 链接: https://leetcode.cn/problems/combination-sum-ii/  
 *  
 * 算法思路:  
 * 1. 先对数组进行排序，使相同元素相邻  
 * 2. 使用回溯算法生成所有组合  
 * 3. 对于重复元素，确保相同元素的相对顺序，避免生成重复组合  
 * 4. 使用剪枝优化：当当前和超过 target 时提前终止  
 *  
 * 时间复杂度: O(2^n)，其中 n 是数组长度  
 * 空间复杂度: O(n)，递归栈深度  
 */  
class Solution {  
public:  
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {  
        vector<vector<int>> result;  
        vector<int> path;  
        sort(candidates.begin(), candidates.end());  
        backtrack(candidates, target, 0, path, result);  
        return result;  
    }  
  
private:
```

```

void backtrack(vector<int>& candidates, int target, int start, vector<int>& path,
vector<vector<int>>& result) {
    // 终止条件：目标值为 0
    if (target == 0) {
        result.push_back(path);
        return;
    }

    for (int i = start; i < candidates.size(); i++) {
        // 剪枝：如果当前数字已经大于剩余目标值，提前终止
        if (candidates[i] > target) {
            break;
        }

        // 去重关键：跳过重复元素（不是第一个出现的重复元素）
        if (i > start && candidates[i] == candidates[i - 1]) {
            continue;
        }

        path.push_back(candidates[i]);
        // 注意：这里传入 i+1，因为每个数字只能使用一次
        backtrack(candidates, target - candidates[i], i + 1, path, result);
        path.pop_back();
    }
}
};

// 测试函数
void testCombinationSumII() {
    Solution solution;

    // 测试用例 1
    vector<int> candidates1 = {10, 1, 2, 7, 6, 1, 5};
    int target1 = 8;
    vector<vector<int>> result1 = solution.combinationSum2(candidates1, target1);
    cout << "输入: candidates = [10,1,2,7,6,1,5], target = " << target1 << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << "[";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << result1[i][j];
            if (j < result1[i].size() - 1) cout << ",";
        }
        cout << "]";
    }
}

```

```

cout << "]";
if (i < result1.size() - 1) cout << ", ";
}
cout << "]" << endl;

// 测试用例 2
vector<int> candidates2 = {2, 5, 2, 1, 2};
int target2 = 5;
vector<vector<int>> result2 = solution.combinationSum2(candidates2, target2);
cout << "\n输入: candidates = [2,5,2,1,2], target = " << target2 << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "[";
    for (int j = 0; j < result2[i].size(); j++) {
        cout << result2[i][j];
        if (j < result2[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

int main() {
    testCombinationSumII();
    return 0;
}

```

=====

文件: Code21\_CombinationSumII.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * LeetCode 40. 组合总和 II
 *
 * 题目描述:
 * 给定一个候选人编号的集合 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为

```

target 的组合。

\* candidates 中的每个数字在每个组合中只能使用一次。

\* 注意：解集不能包含重复的组合。

\*

\* 示例：

\* 输入：candidates = [10, 1, 2, 7, 6, 1, 5], target = 8

\* 输出：[[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

\*

\* 输入：candidates = [2, 5, 2, 1, 2], target = 5

\* 输出：[[1, 2, 2], [5]]

\*

\* 提示：

\*  $1 \leq \text{candidates.length} \leq 100$

\*  $1 \leq \text{candidates}[i] \leq 50$

\*  $1 \leq \text{target} \leq 30$

\*

\* 链接：<https://leetcode.cn/problems/combination-sum-ii/>

\*

\* 算法思路：

\* 1. 先对数组进行排序，使相同元素相邻

\* 2. 使用回溯算法生成所有组合

\* 3. 对于重复元素，确保相同元素的相对顺序，避免生成重复组合

\* 4. 使用剪枝优化：当当前和超过 target 时提前终止

\*

\* 时间复杂度： $O(2^n)$ ，其中 n 是数组长度

\* 空间复杂度： $O(n)$ ，递归栈深度

\*/

```
public class Code21_CombinationSumII {
```

/\*\*

\* 生成所有和为 target 的组合（每个数字只能使用一次）

\*

\* @param candidates 候选数组

\* @param target 目标和

\* @return 所有满足条件的组合

\*/

```
public static List<List<Integer>> combinationSum2(int[] candidates, int target) {
```

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

```
    Arrays.sort(candidates);
```

```
    backtrack(candidates, target, 0, new ArrayList<>(), result);
```

```
    return result;
```

```
}
```

```

/**
 * 回溯函数生成组合
 *
 * @param candidates 候选数组
 * @param target 剩余目标值
 * @param start 当前起始位置
 * @param path 当前路径
 * @param result 结果列表
 */
private static void backtrack(int[] candidates, int target, int start, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：目标值为 0
    if (target == 0) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int i = start; i < candidates.length; i++) {
        // 剪枝：如果当前数字已经大于剩余目标值，提前终止
        if (candidates[i] > target) {
            break;
        }

        // 去重关键：跳过重复元素（不是第一个出现的重复元素）
        if (i > start && candidates[i] == candidates[i - 1]) {
            continue;
        }

        path.add(candidates[i]);
        // 注意：这里传入 i+1，因为每个数字只能使用一次
        backtrack(candidates, target - candidates[i], i + 1, path, result);
        path.remove(path.size() - 1);
    }
}

/**
 * 解法二：使用计数法处理重复元素
 * 对于重复元素，我们可以选择 0 个、1 个、2 个... 直到不超过目标值的最大个数
 *
 * @param candidates 候选数组
 * @param target 目标和
 * @return 所有满足条件的组合
 */

```

```

public static List<List<Integer>> combinationSum2(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(candidates);
    backtrack2(candidates, target, 0, new ArrayList<>(), result);
    return result;
}

private static void backtrack2(int[] candidates, int target, int start, List<Integer> path,
List<List<Integer>> result) {
    if (target == 0) {
        result.add(new ArrayList<>(path));
        return;
    }

    if (start >= candidates.length || target < 0) {
        return;
    }

    // 统计当前元素出现的次数
    int count = 1;
    int i = start + 1;
    while (i < candidates.length && candidates[i] == candidates[start]) {
        count++;
        i++;
    }

    // 对于当前元素，可以选择 0 个、1 个、2 个... 最多不超过目标值的个数
    for (int j = 0; j <= count; j++) {
        // 如果选择 j 个当前元素已经超过目标值，提前终止
        if (j * candidates[start] > target) {
            break;
        }

        // 添加 j 个当前元素
        for (int k = 0; k < j; k++) {
            path.add(candidates[start]);
        }

        backtrack2(candidates, target - j * candidates[start], start + count, path, result);

        // 回溯，移除添加的元素
        for (int k = 0; k < j; k++) {
            path.remove(path.size() - 1);
        }
    }
}

```

```

        }
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[] candidates1 = {10, 1, 2, 7, 6, 1, 5};
    int target1 = 8;
    List<List<Integer>> result1 = combinationSum2(candidates1, target1);
    System.out.println("输入: candidates = [10, 1, 2, 7, 6, 1, 5], target = " + target1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] candidates2 = {2, 5, 2, 1, 2};
    int target2 = 5;
    List<List<Integer>> result2 = combinationSum2(candidates2, target2);
    System.out.println("\n 输入: candidates = [2, 5, 2, 1, 2], target = " + target2);
    System.out.println("输出: " + result2);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ===");
    List<List<Integer>> result3 = combinationSum2_2(candidates1, target1);
    System.out.println("输入: candidates = [10, 1, 2, 7, 6, 1, 5], target = " + target1);
    System.out.println("输出: " + result3);
}
}

```

文件: Code21\_CombinationSumII.py

```
from typing import List
```

```
class Solution:
```

```
    """

```

```
    LeetCode 40. 组合总和 II
```

**题目描述:**

给定一个候选人编号的集合 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

注意：解集不能包含重复的组合。

示例：

输入： candidates = [10, 1, 2, 7, 6, 1, 5], target = 8

输出： [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

输入： candidates = [2, 5, 2, 1, 2], target = 5

输出： [[1, 2, 2], [5]]

提示：

1 <= candidates.length <= 100

1 <= candidates[i] <= 50

1 <= target <= 30

链接：<https://leetcode.cn/problems/combination-sum-ii/>

算法思路：

1. 先对数组进行排序，使相同元素相邻
2. 使用回溯算法生成所有组合
3. 对于重复元素，确保相同元素的相对顺序，避免生成重复组合
4. 使用剪枝优化：当当前和超过 target 时提前终止

时间复杂度： $O(2^n)$ ，其中 n 是数组长度

空间复杂度： $O(n)$ ，递归栈深度

"""

```
def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:  
    result = []  
    candidates.sort() # 先排序，使相同元素相邻  
    self.backtrack(candidates, target, 0, [], result)  
    return result  
  
def backtrack(self, candidates: List[int], target: int, start: int, path: List[int], result: List[List[int]]) -> None:  
    # 终止条件：目标值为 0  
    if target == 0:  
        result.append(path[:])  
        return  
  
    for i in range(start, len(candidates)):  
        # 剪枝：如果当前数字已经大于剩余目标值，提前终止  
        if candidates[i] > target:  
            break  
        else:  
            self.backtrack(candidates, target - candidates[i], i + 1, path + [candidates[i]], result)
```

```

# 去重关键：跳过重复元素（不是第一个出现的重复元素）
if i > start and candidates[i] == candidates[i - 1]:
    continue

path.append(candidates[i])
# 注意：这里传入 i+1，因为每个数字只能使用一次
self.backtrack(candidates, target - candidates[i], i + 1, path, result)
path.pop()

def test_combination_sum_ii():
    solution = Solution()

    # 测试用例 1
    candidates1 = [10, 1, 2, 7, 6, 1, 5]
    target1 = 8
    result1 = solution.combinationSum2(candidates1, target1)
    print("输入: candidates = [10, 1, 2, 7, 6, 1, 5], target =", target1)
    print("输出:", result1)

    # 测试用例 2
    candidates2 = [2, 5, 2, 1, 2]
    target2 = 5
    result2 = solution.combinationSum2(candidates2, target2)
    print("\n输入: candidates = [2, 5, 2, 1, 2], target =", target2)
    print("输出:", result2)

if __name__ == "__main__":
    test_combination_sum_ii()

```

=====

文件: Code22\_CombinationSumIII.cpp

=====

```

#include <iostream>
#include <vector>

using namespace std;

/**
 * LeetCode 216. 组合总和 III
 *
 * 题目描述：
 * 找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重

```

复的数字。

```
*  
* 示例:  
* 输入: k = 3, n = 7  
* 输出: [[1,2,4]]  
  
*  
* 输入: k = 3, n = 9  
* 输出: [[1,2,6],[1,3,5],[2,3,4]]  
  
*  
* 输入: k = 4, n = 1  
* 输出: []  
  
*  
* 提示:  
* 2 <= k <= 9  
* 1 <= n <= 60  
  
* 链接: https://leetcode.cn/problems/combination-sum-iii/  
  
* 算法思路:  
* 1. 使用回溯算法生成所有可能的组合  
* 2. 从 1 开始，每次选择一个数字，然后递归选择下一个数字  
* 3. 当组合长度达到 k 且和为 n 时，将其加入结果集  
* 4. 通过控制起始位置避免重复组合  
* 5. 使用剪枝优化：当当前和已经超过 n 时提前终止  
  
* 时间复杂度: O(C(9, k))，从 9 个数字中选择 k 个数字的组合数  
* 空间复杂度: O(k)，递归栈深度  
*/  
class Solution {  
public:  
    vector<vector<int>> combinationSum3(int k, int n) {  
        vector<vector<int>> result;  
        vector<int> path;  
        backtrack(k, n, 1, path, result);  
        return result;  
    }  
  
private:  
    void backtrack(int k, int n, int start, vector<int>& path, vector<vector<int>>& result) {  
        // 终止条件：已选择 k 个数字  
        if (k == 0) {  
            // 如果和为 n，加入结果集  
            if (n == 0) {  
                result.push_back(path);  
            }  
        } else {  
            for (int i = start; i <= 9; ++i) {  
                path.push_back(i);  
                backtrack(k - 1, n - i, i + 1, path, result);  
                path.pop_back();  
            }  
        }  
    }  
}
```

```

        result.push_back(path);
    }
    return;
}

// 剪枝优化：如果剩余的数字不足以填满组合，提前终止
// 还需要选择的数字个数：k
// 从 start 到 9 至少要有这么多个数字：9 - start + 1 >= k
// 所以 start <= 9 - k + 1
for (int i = start; i <= 9 - k + 1; i++) {
    // 剪枝：如果当前数字已经大于剩余目标值，提前终止
    if (i > n) {
        break;
    }

    path.push_back(i);
    backtrack(k - 1, n - i, i + 1, path, result);
    path.pop_back();
}
}

};

// 测试函数
void testCombinationSumIII() {
    Solution solution;

    // 测试用例 1
    int k1 = 3, n1 = 7;
    vector<vector<int>> result1 = solution.combinationSum3(k1, n1);
    cout << "输入: k = " << k1 << ", n = " << n1 << endl;
    cout << "输出: [";
    for (int i = 0; i < result1.size(); i++) {
        cout << "[";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << result1[i][j];
            if (j < result1[i].size() - 1) cout << ",";
        }
        cout << "]";
        if (i < result1.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
}

// 测试用例 2

```

```

int k2 = 3, n2 = 9;
vector<vector<int>> result2 = solution.combinationSum3(k2, n2);
cout << "\n 输入: k = " << k2 << ", n = " << n2 << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << "[";
    for (int j = 0; j < result2[i].size(); j++) {
        cout << result2[i][j];
        if (j < result2[i].size() - 1) cout << ",";
    }
    cout << "]";
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

int main() {
    testCombinationSumIII();
    return 0;
}

```

=====

文件: Code22\_CombinationSumIII.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 216. 组合总和 III
 *
 * 题目描述:
 * 找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。
 *
 * 示例:
 * 输入: k = 3, n = 7
 * 输出: [[1,2,4]]
 *
 * 输入: k = 3, n = 9
 * 输出: [[1,2,6],[1,3,5],[2,3,4]]

```

```
*  
* 输入: k = 4, n = 1  
* 输出: []  
*  
* 提示:  
* 2 <= k <= 9  
* 1 <= n <= 60  
*  
* 链接: https://leetcode.cn/problems/combination-sum-iii/  
*  
* 算法思路:  
* 1. 使用回溯算法生成所有可能的组合  
* 2. 从 1 开始, 每次选择一个数字, 然后递归选择下一个数字  
* 3. 当组合长度达到 k 且和为 n 时, 将其加入结果集  
* 4. 通过控制起始位置避免重复组合  
* 5. 使用剪枝优化: 当当前和已经超过 n 时提前终止  
*  
* 时间复杂度: O(C(9, k)), 从 9 个数字中选择 k 个数字的组合数  
* 空间复杂度: O(k), 递归栈深度  
*/
```

```
public class Code22_CombinationSumIII {
```

```
/**  
 * 生成所有和为 n 的 k 个数的组合 (数字范围 1-9)  
 *  
 * @param k 组合大小  
 * @param n 目标和  
 * @return 所有满足条件的组合  
 */
```

```
public static List<List<Integer>> combinationSum3(int k, int n) {  
    List<List<Integer>> result = new ArrayList<>();  
    backtrack(k, n, 1, new ArrayList<>(), result);  
    return result;  
}
```

```
/**  
 * 回溯函数生成组合  
 *  
 * @param k 剩余需要选择的数字个数  
 * @param n 剩余目标值  
 * @param start 当前起始数字  
 * @param path 当前路径  
 * @param result 结果列表
```

```

*/
private static void backtrack(int k, int n, int start, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：已选择 k 个数字
    if (k == 0) {
        // 如果和为 n，加入结果集
        if (n == 0) {
            result.add(new ArrayList<>(path));
        }
        return;
    }

    // 剪枝优化：如果剩余的数字不足以填满组合，提前终止
    // 还需要选择的数字个数：k
    // 从 start 到 9 至少要有这么多个数字：9 - start + 1 >= k
    // 所以 start <= 9 - k + 1
    for (int i = start; i <= 9 - k + 1; i++) {
        // 剪枝：如果当前数字已经大于剩余目标值，提前终止
        if (i > n) {
            break;
        }

        path.add(i);
        backtrack(k - 1, n - i, i + 1, path, result);
        path.remove(path.size() - 1);
    }
}

/**
 * 解法二：使用迭代法生成组合
 * 使用位运算生成所有可能的组合
 *
 * @param k 组合大小
 * @param n 目标和
 * @return 所有满足条件的组合
 */
public static List<List<Integer>> combinationSum3_2(int k, int n) {
    List<List<Integer>> result = new ArrayList<>();

    // 遍历所有可能的组合（使用位掩码）
    for (int mask = 0; mask < (1 << 9); mask++) {
        List<Integer> combination = new ArrayList<>();
        int sum = 0;

```

```

// 检查当前掩码对应的组合
for (int i = 0; i < 9; i++) {
    if ((mask & (1 << i)) != 0) {
        combination.add(i + 1); // 数字从 1 开始
        sum += i + 1;
    }
}

// 检查是否满足条件
if (combination.size() == k && sum == n) {
    result.add(combination);
}
}

return result;
}

/***
 * 解法三：使用动态规划预处理
 * 先计算所有可能的组合，然后筛选满足条件的
 *
 * @param k 组合大小
 * @param n 目标和
 * @return 所有满足条件的组合
 */
public static List<List<Integer>> combinationSum3_3(int k, int n) {
    // 先使用回溯生成所有 k 个数字的组合
    List<List<Integer>> allCombinations = new ArrayList<>();
    generateCombinations(k, 1, new ArrayList<>(), allCombinations);

    // 筛选和为 n 的组合
    List<List<Integer>> result = new ArrayList<>();
    for (List<Integer> comb : allCombinations) {
        int sum = 0;
        for (int num : comb) {
            sum += num;
        }
        if (sum == n) {
            result.add(comb);
        }
    }
}

```

```
        return result;
    }

    private static void generateCombinations(int k, int start, List<Integer> path,
List<List<Integer>> result) {
        if (k == 0) {
            result.add(new ArrayList<>(path));
            return;
        }

        for (int i = start; i <= 9; i++) {
            path.add(i);
            generateCombinations(k - 1, i + 1, path, result);
            path.remove(path.size() - 1);
        }
    }

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int k1 = 3, n1 = 7;
    List<List<Integer>> result1 = combinationSum3(k1, n1);
    System.out.println("输入: k = " + k1 + ", n = " + n1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int k2 = 3, n2 = 9;
    List<List<Integer>> result2 = combinationSum3(k2, n2);
    System.out.println("\n输入: k = " + k2 + ", n = " + n2);
    System.out.println("输出: " + result2);

    // 测试用例 3
    int k3 = 4, n3 = 1;
    List<List<Integer>> result3 = combinationSum3(k3, n3);
    System.out.println("\n输入: k = " + k3 + ", n = " + n3);
    System.out.println("输出: " + result3);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ===");
    List<List<Integer>> result4 = combinationSum3_2(k1, n1);
    System.out.println("输入: k = " + k1 + ", n = " + n1);
    System.out.println("输出: " + result4);
}
```

}

=====

文件: Code22\_CombinationSumIII.py

```
from typing import List
```

```
class Solution:
```

```
    """
```

```
    LeetCode 216. 组合总和 III
```

题目描述:

找出所有相加之和为  $n$  的  $k$  个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

示例:

输入:  $k = 3, n = 7$

输出:  $[[1, 2, 4]]$

输入:  $k = 3, n = 9$

输出:  $[[1, 2, 6], [1, 3, 5], [2, 3, 4]]$

输入:  $k = 4, n = 1$

输出:  $[]$

提示:

$2 \leq k \leq 9$

$1 \leq n \leq 60$

链接: <https://leetcode.cn/problems/combination-sum-iii/>

算法思路:

1. 使用回溯算法生成所有可能的组合
2. 从 1 开始，每次选择一个数字，然后递归选择下一个数字
3. 当组合长度达到  $k$  且和为  $n$  时，将其加入结果集
4. 通过控制起始位置避免重复组合
5. 使用剪枝优化：当当前和已经超过  $n$  时提前终止

时间复杂度:  $O(C(9, k))$ ，从 9 个数字中选择  $k$  个数字的组合数

空间复杂度:  $O(k)$ ，递归栈深度

```
"""
```

```
def combinationSum3(self, k: int, n: int) -> List[List[int]]:  
    result = []  
    self.backtrack(k, n, 1, [], result)  
    return result
```

```
def backtrack(self, k: int, n: int, start: int, path: List[int], result: List[List[int]]) ->
```

None:

# 终止条件：已选择 k 个数字

```
if k == 0:
```

# 如果和为 n，加入结果集

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

```
    result.append(path[:])
```

```
return
```

# 剪枝优化：如果剩余的数字不足以填满组合，提前终止

# 还需要选择的数字个数：k

# 从 start 到 9 至少要有这么多个数字：9 - start + 1 >= k

# 所以 i 的范围是 start 到 9 - k + 1

```
for i in range(start, 10 - k):
```

# 剪枝：如果当前数字已经大于剩余目标值，提前终止

```
if i > n:
```

```
    break
```

```
path.append(i)
```

```
self.backtrack(k - 1, n - i, i + 1, path, result)
```

```
path.pop()
```

```
def test_combination_sum_iii():
```

```
    solution = Solution()
```

# 测试用例 1

```
k1, n1 = 3, 7
```

```
result1 = solution.combinationSum3(k1, n1)
```

```
print(f"输入: k = {k1}, n = {n1}")
```

```
print("输出:", result1)
```

# 测试用例 2

```
k2, n2 = 3, 9
```

```
result2 = solution.combinationSum3(k2, n2)
```

```
print(f"\n输入: k = {k2}, n = {n2}")
```

```
print("输出:", result2)
```

# 测试用例 3

```
k3, n3 = 4, 1
result3 = solution.combinationSum3(k3, n3)
print(f"\n输入: k = {k3}, n = {n3}")
print("输出:", result3)

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

---

文件: Code23\_PermutationSequence.java

---

```
=====
package class038;

import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 60. 排列序列
 *
 * 题目描述:
 * 给出集合 [1, 2, 3, ..., n]，其所有元素共有 n! 种排列。
 * 按大小顺序列出所有排列情况，并一一标记，当 n = 3 时，所有排列如下：
 * "123", "132", "213", "231", "312", "321"
 * 给定 n 和 k，返回第 k 个排列。
 *
 * 示例:
 * 输入: n = 3, k = 3
 * 输出: "213"
 *
 * 输入: n = 4, k = 9
 * 输出: "2314"
 *
 * 输入: n = 3, k = 1
 * 输出: "123"
 *
 * 提示:
 * 1 <= n <= 9
 * 1 <= k <= n!
 *
 * 链接: https://leetcode.cn/problems/permuation-sequence/
 *
 * 算法思路:
```

```
* 1. 使用数学方法（康托展开）直接计算第 k 个排列
* 2. 对于 n 个数字，第一个位置有 n 种选择，每种选择对应  $(n-1)!$  个排列
* 3. 通过计算 k 所在的区间确定第一个数字
* 4. 递归处理剩余数字
*
* 时间复杂度： $O(n^2)$ ，需要遍历 n 个位置，每个位置需要  $O(n)$  时间确定数字
* 空间复杂度： $O(n)$ ，用于存储可用数字和结果
*/
```

```
public class Code23_PermutationSequence {

    /**
     * 获取第 k 个排列
     *
     * @param n 数字个数
     * @param k 排列序号
     * @return 第 k 个排列的字符串表示
     */
    public static String getPermutation(int n, int k) {
        // 计算阶乘数组
        int[] factorial = new int[n + 1];
        factorial[0] = 1;
        for (int i = 1; i <= n; i++) {
            factorial[i] = factorial[i - 1] * i;
        }

        // 创建可用数字列表
        List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= n; i++) {
            numbers.add(i);
        }

        // 康托展开计算排列
        StringBuilder result = new StringBuilder();
        k--; // 转换为 0-based 索引

        for (int i = n; i >= 1; i--) {
            int index = k / factorial[i - 1];
            result.append(numbers.get(index));
            numbers.remove(index);
            k %= factorial[i - 1];
        }

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

```

}

/**
 * 解法二：使用回溯算法（适用于小规模数据）
 * 生成所有排列直到第 k 个
 *
 * @param n 数字个数
 * @param k 排列序号
 * @return 第 k 个排列的字符串表示
 */

public static String getPermutation2(int n, int k) {
    int[] count = {0};
    StringBuilder result = new StringBuilder();
    boolean[] used = new boolean[n + 1];
    backtrack(n, k, used, new StringBuilder(), count, result);
    return result.toString();
}

private static boolean backtrack(int n, int k, boolean[] used, StringBuilder path, int[]
count, StringBuilder result) {
    // 终止条件：生成完整排列
    if (path.length() == n) {
        count[0]++;
        if (count[0] == k) {
            result.append(path);
            return true;
        }
        return false;
    }

    for (int i = 1; i <= n; i++) {
        if (!used[i]) {
            used[i] = true;
            path.append(i);

            if (backtrack(n, k, used, path, count, result)) {
                return true;
            }

            path.deleteCharAt(path.length() - 1);
            used[i] = false;
        }
    }
}

```

```

    return false;
}

/***
 * 解法三：使用迭代法生成排列
 * 使用字典序算法生成第 k 个排列
 *
 * @param n 数字个数
 * @param k 排列序号
 * @return 第 k 个排列的字符串表示
 */
public static String getPermutation3(int n, int k) {
    // 生成初始排列（最小排列）
    char[] arr = new char[n];
    for (int i = 0; i < n; i++) {
        arr[i] = (char) ('1' + i);
    }

    // 使用字典序算法生成第 k 个排列
    for (int i = 1; i < k; i++) {
        nextPermutation(arr);
    }

    return new String(arr);
}

private static void nextPermutation(char[] arr) {
    int n = arr.length;

    // 1. 从右向左找到第一个降序的位置
    int i = n - 2;
    while (i >= 0 && arr[i] >= arr[i + 1]) {
        i--;
    }

    if (i >= 0) {
        // 2. 从右向左找到第一个大于 arr[i] 的数字
        int j = n - 1;
        while (j >= 0 && arr[j] <= arr[i]) {
            j--;
        }
    }
}

```

```
// 3. 交换 arr[i]和 arr[j]
swap(arr, i, j);
}

// 4. 反转 i+1 到末尾的部分
reverse(arr, i + 1, n - 1);
}

private static void swap(char[] arr, int i, int j) {
    char temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

private static void reverse(char[] arr, int start, int end) {
    while (start < end) {
        swap(arr, start, end);
        start++;
        end--;
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 3, k1 = 3;
    String result1 = getPermutation(n1, k1);
    System.out.println("输入: n = " + n1 + ", k = " + k1);
    System.out.println("输出: \"\"\" + result1 + "\"\"");
}

// 测试用例 2
int n2 = 4, k2 = 9;
String result2 = getPermutation(n2, k2);
System.out.println("\n 输入: n = " + n2 + ", k = " + k2);
System.out.println("输出: \"\"\" + result2 + "\"\"");

// 测试用例 3
int n3 = 3, k3 = 1;
String result3 = getPermutation(n3, k3);
System.out.println("\n 输入: n = " + n3 + ", k = " + k3);
System.out.println("输出: \"\"\" + result3 + "\"\"");

// 测试解法二
```

```

System.out.println("\n== 解法二测试 ==");
String result4 = getPermutation2(n1, k1);
System.out.println("输入: n = " + n1 + ", k = " + k1);
System.out.println("输出: \"\"\" + result4 + "\"\"");
}

// 测试解法三
System.out.println("\n== 解法三测试 ==");
String result5 = getPermutation3(n1, k1);
System.out.println("输入: n = " + n1 + ", k = " + k1);
System.out.println("输出: \"\"\" + result5 + "\"\"");
}

```

=====

文件: Code23\_PermutationSequence.py

=====

```

class Solution:
    """
    LeetCode 60. 排列序列

```

题目描述:

给出集合  $[1, 2, 3, \dots, n]$ ，其所有元素共有  $n!$  种排列。

按大小顺序列出所有排列情况，并找出第  $k$  个排列。

示例:

输入:  $n = 3, k = 3$

输出: "213"

输入:  $n = 4, k = 9$

输出: "2314"

输入:  $n = 3, k = 1$

输出: "123"

提示:

$1 \leq n \leq 9$

$1 \leq k \leq n!$

链接: <https://leetcode.cn/problems/permute/>

算法思路:

1. 使用数学方法（康托展开）直接计算第  $k$  个排列

2. 预先计算阶乘数组，用于快速确定每个位置的数字
3. 从高位到低位依次确定每个位置的数字
4. 使用列表记录可用的数字，每次选择一个数字后将其移除

时间复杂度： $O(n^2)$ ，需要遍历  $n$  个位置，每个位置需要  $O(n)$  时间查找和删除

空间复杂度： $O(n)$ ，存储阶乘数组和可用数字列表

"""

```
def getPermutation(self, n: int, k: int) -> str:
    # 计算阶乘数组
    factorial = [1] * (n + 1)
    for i in range(1, n + 1):
        factorial[i] = factorial[i - 1] * i

    # 可用数字列表
    numbers = list(range(1, n + 1))
    result = []
    k -= 1  # 转换为 0-based 索引

    for i in range(n - 1, -1, -1):
        # 计算当前位应该选择第几个数字
        index = k // factorial[i]
        result.append(str(numbers[index]))
        # 移除已选择的数字
        numbers.pop(index)
        # 更新 k 值
        k %= factorial[i]

    return ''.join(result)
```

```
def test_permutation_sequence():
    solution = Solution()

    # 测试用例 1
    n1, k1 = 3, 3
    result1 = solution.getPermutation(n1, k1)
    print(f"输入: n = {n1}, k = {k1}")
    print("输出:", result1)
```

```
# 测试用例 2
n2, k2 = 4, 9
result2 = solution.getPermutation(n2, k2)
print(f"\n输入: n = {n2}, k = {k2}")
```

```

print("输出:", result2)

# 测试用例 3
n3, k3 = 3, 1
result3 = solution.getPermutation(n3, k3)
print(f"\n 输入: n = {n3}, k = {k3}")
print("输出:", result3)

# 测试用例 4
n4, k4 = 1, 1
result4 = solution.getPermutation(n4, k4)
print(f"\n 输入: n = {n4}, k = {k4}")
print("输出:", result4)

if __name__ == "__main__":
    test_permutation_sequence()

```

---

文件: Code24\_RestoreIPAddresses.cpp

---

```

/**
 * LeetCode 93. 复原 IP 地址
 *
 * 题目描述:
 * 有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔。
 * 例如: "0.1.2.201" 和 "192.168.1.1" 是有效 IP 地址，
 * 但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是无效 IP 地址。
 * 给定一个只包含数字的字符串 s，用以表示一个 IP 地址，返回所有可能的有效 IP 地址。
 *
 * 示例:
 * 输入: s = "25525511135"
 * 输出: ["255.255.11.135", "255.255.111.35"]
 *
 * 输入: s = "0000"
 * 输出: ["0.0.0.0"]
 *
 * 输入: s = "101023"
 * 输出: ["1.0.10.23", "1.0.102.3", "10.1.0.23", "10.10.2.3", "101.0.2.3"]
 *
 * 提示:
 * 1 <= s.length <= 20

```

```
* s 仅由数字组成
*
* 链接: https://leetcode.cn/problems/restore-ip-addresses/
*
* 算法思路:
* 1. 使用回溯算法分割字符串为 4 个部分
* 2. 每个部分必须满足: 0-255 之间, 不能有前导 0 (除非是 0 本身)
* 3. 当分割完成 4 个部分且字符串用完时, 加入结果集
* 4. 使用剪枝优化: 剩余字符串长度不足以填满剩余部分时提前终止
*
* 时间复杂度: O(3^4) = O(81), 每个部分最多 3 位数字, 共 4 个部分
* 空间复杂度: O(n), 递归栈深度
*/

```

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

class Solution {
public:
    /**
     * 复原 IP 地址
     *
     * @param s 输入字符串
     * @return 所有可能的有效 IP 地址
     */
    std::vector<std::string> restoreIpAddresses(std::string s) {
        std::vector<std::string> result;
        // 边界条件检查
        if (s.empty() || s.length() < 4 || s.length() > 12) {
            return result;
        }
        std::vector<std::string> path;
        backtrack(s, 0, path, result);
        return result;
    }

private:
    /**
     * 回溯函数生成 IP 地址
     *
     * @param s 输入字符串
     */
```

```

* @param start 当前起始位置
* @param path 当前路径（已分割的部分）
* @param result 结果列表
*/
void backtrack(const std::string& s, int start, std::vector<std::string>& path,
std::vector<std::string>& result) {
    // 终止条件：已分割成 4 部分且处理完所有字符
    if (path.size() == 4) {
        if (start == s.length()) {
            std::string ip;
            for (int i = 0; i < 4; i++) {
                ip += path[i];
                if (i < 3) ip += ".";
            }
            result.push_back(ip);
        }
        return;
    }

    // 剪枝：剩余字符串长度不足以填满剩余部分
    // 剩余部分数：4 - path.size()
    // 每个部分最少 1 位，最多 3 位
    int min_remaining = 4 - path.size();
    int max_remaining = 3 * (4 - path.size());
    int remaining_length = s.length() - start;

    if (remaining_length < min_remaining || remaining_length > max_remaining) {
        return;
    }

    // 尝试分割 1-3 个字符
    for (int len = 1; len <= 3; len++) {
        if (start + len > s.length()) {
            break;
        }

        std::string segment = s.substr(start, len);

        // 检查分割部分是否合法
        if (isValidSegment(segment)) {
            path.push_back(segment);
            backtrack(s, start + len, path, result);
            path.pop_back(); // 回溯
        }
    }
}

```

```

        }
    }
}

/***
 * 检查 IP 地址分段是否合法
 *
 * @param segment IP 地址分段
 * @return 是否合法
 */
bool isValidSegment(const std::string& segment) {
    // 长度检查
    if (segment.empty() || segment.length() > 3) {
        return false;
    }

    // 前导 0 检查：如果长度大于 1 且以 0 开头，不合法
    if (segment.length() > 1 && segment[0] == '0') {
        return false;
    }

    // 数值范围检查：必须在 0-255 之间
    int value = stoi(segment);
    return value >= 0 && value <= 255;
}

public:
    // 解法二：使用迭代法生成所有可能的分割方案
    // 通过三层循环枚举所有可能的分割点
    std::vector<std::string> restoreIpAddresses2(std::string s) {
        std::vector<std::string> result;
        int n = s.length();

        // 枚举三个分割点的位置
        for (int i = 1; i <= 3 && i <= n - 3; i++) {
            for (int j = i + 1; j <= i + 3 && j <= n - 2; j++) {
                for (int k = j + 1; k <= j + 3 && k <= n - 1; k++) {
                    std::string seg1 = s.substr(0, i);
                    std::string seg2 = s.substr(i, j - i);
                    std::string seg3 = s.substr(j, k - j);
                    std::string seg4 = s.substr(k);

                    // 检查每个分段是否合法

```

```

        if (isValidSegment(seg1) && isValidSegment(seg2) &&
            isValidSegment(seg3) && isValidSegment(seg4)) {
            result.push_back(seg1 + "." + seg2 + "." + seg3 + "." + seg4);
        }
    }
}

return result;
}
};

// 辅助函数: 检查 IP 地址分段是否合法
bool isValidSegmentHelper(const std::string& segment) {
    // 长度检查
    if (segment.empty() || segment.length() > 3) {
        return false;
    }

    // 前导 0 检查: 如果长度大于 1 且以 0 开头, 不合法
    if (segment.length() > 1 && segment[0] == '0') {
        return false;
    }

    // 数值范围检查: 必须在 0-255 之间
    int value = stoi(segment);
    return value >= 0 && value <= 255;
}

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

    // 测试用例 1
    std::string s1 = "25525511135";
    std::vector<std::string> result1 = solution.restoreIpAddresses(s1);
    printf("输入: s = \"%s\"\n", s1.c_str());
    printf("输出: [");
    for (int i = 0; i < result1.size(); i++) {
        printf("\\"%s\\\"", result1[i].c_str());
        if (i < result1.size() - 1) printf(", ");
    }
    printf("]\n");
}

```

```

// 测试用例 2
std::string s2 = "0000";
std::vector<std::string> result2 = solution.restoreIpAddresses(s2);
printf("\n 输入: s = \"%s\"\n", s2.c_str());
printf("输出: [");
for (int i = 0; i < result2.size(); i++) {
    printf("%s", result2[i].c_str());
    if (i < result2.size() - 1) printf(", ");
}
printf("]\n");

// 测试用例 3
std::string s3 = "101023";
std::vector<std::string> result3 = solution.restoreIpAddresses(s3);
printf("\n 输入: s = \"%s\"\n", s3.c_str());
printf("输出: [");
for (int i = 0; i < result3.size(); i++) {
    printf("%s", result3[i].c_str());
    if (i < result3.size() - 1) printf(", ");
}
printf("]\n");

// 测试解法二
printf("\n==== 解法二测试 ====\n");
std::vector<std::string> result4 = solution.restoreIpAddresses2(s1);
printf("输入: s = \"%s\"\n", s1.c_str());
printf("输出: [");
for (int i = 0; i < result4.size(); i++) {
    printf("%s", result4[i].c_str());
    if (i < result4.size() - 1) printf(", ");
}
printf("]\n");

return 0;
}

```

=====

文件: Code24\_RestoreIPAddresses.java

=====

```
package class038;
```

```
import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 93. 复原 IP 地址
 *
 * 题目描述:
 * 有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 ‘.’ 分隔。
 * 例如：“0.1.2.201” 和 “192.168.1.1” 是有效 IP 地址，
 * 但是 “0.011.255.245”、“192.168.1.312” 和 “192.168@1.1” 是无效 IP 地址。
 * 给定一个只包含数字的字符串 s，用以表示一个 IP 地址，返回所有可能的有效 IP 地址。
 *
 * 示例:
 * 输入: s = "25525511135"
 * 输出: ["255.255.11.135", "255.255.111.35"]
 *
 * 输入: s = "0000"
 * 输出: ["0.0.0.0"]
 *
 * 输入: s = "101023"
 * 输出: ["1.0.10.23", "1.0.102.3", "10.1.0.23", "10.10.2.3", "101.0.2.3"]
 *
 * 提示:
 * 1 <= s.length <= 20
 * s 仅由数字组成
 *
 * 链接: https://leetcode.cn/problems/restore-ip-addresses/
 *
 * 算法思路:
 * 1. 使用回溯算法生成所有可能的 IP 地址分割方案
 * 2. IP 地址由 4 个部分组成，每个部分必须是 0-255 之间的整数
 * 3. 不能有前导 0（除非数字本身就是 0）
 * 4. 通过递归分割字符串，检查每个分割部分是否合法
 *
 * 时间复杂度: O(3^4) = O(81)，因为每个部分最多 3 位数字，共 4 个部分
 * 空间复杂度: O(n)，递归栈深度
 */
public class Code24_RestoreIPAddresses {
```

```
    /**
     * 复原 IP 地址
     *
```

```

* @param s 输入字符串
* @return 所有可能的有效 IP 地址
*/
public static List<String> restoreIpAddresses(String s) {
    List<String> result = new ArrayList<>();
    // 边界条件检查
    if (s == null || s.length() < 4 || s.length() > 12) {
        return result;
    }
    backtrack(s, 0, new ArrayList<>(), result);
    return result;
}

/**
 * 回溯函数生成 IP 地址
 *
 * @param s 输入字符串
 * @param start 当前起始位置
 * @param path 当前路径（已分割的部分）
 * @param result 结果列表
 */
private static void backtrack(String s, int start, List<String> path, List<String> result) {
    // 终止条件：已分割成 4 部分且处理完所有字符
    if (path.size() == 4) {
        if (start == s.length()) {
            result.add(String.join(".", path));
        }
        return;
    }

    // 尝试分割 1-3 个字符
    for (int len = 1; len <= 3; len++) {
        if (start + len > s.length()) {
            break;
        }

        String segment = s.substring(start, start + len);

        // 检查分割部分是否合法
        if (isValidSegment(segment)) {
            path.add(segment);
            backtrack(s, start + len, path, result);
            path.remove(path.size() - 1); // 回溯
        }
    }
}

```

```

        }
    }
}

/***
 * 检查 IP 地址分段是否合法
 *
 * @param segment IP 地址分段
 * @return 是否合法
 */
private static boolean isValidSegment(String segment) {
    // 长度检查
    if (segment.length() == 0 || segment.length() > 3) {
        return false;
    }

    // 前导 0 检查：如果长度大于 1 且以 0 开头，不合法
    if (segment.length() > 1 && segment.charAt(0) == '0') {
        return false;
    }

    // 数值范围检查：必须在 0-255 之间
    int value = Integer.parseInt(segment);
    return value >= 0 && value <= 255;
}

/***
 * 解法二：使用迭代法生成所有可能的分割方案
 * 通过三层循环枚举所有可能的分割点
 *
 * @param s 输入字符串
 * @return 所有可能的有效 IP 地址
 */
public static List<String> restoreIpAddresses2(String s) {
    List<String> result = new ArrayList<>();
    int n = s.length();

    // 枚举三个分割点的位置
    for (int i = 1; i <= 3 && i <= n - 3; i++) {
        for (int j = i + 1; j <= i + 3 && j <= n - 2; j++) {
            for (int k = j + 1; k <= j + 3 && k <= n - 1; k++) {
                String seg1 = s.substring(0, i);
                String seg2 = s.substring(i, j);
                String seg3 = s.substring(j, k);
                String seg4 = s.substring(k);
                result.add(seg1 + "." + seg2 + "." + seg3 + "." + seg4);
            }
        }
    }
}

```

```
        String seg3 = s.substring(j, k);
        String seg4 = s.substring(k);

        if (isValidSegment(seg1) && isValidSegment(seg2) &&
            isValidSegment(seg3) && isValidSegment(seg4)) {
            result.add(seg1 + "." + seg2 + "." + seg3 + "." + seg4);
        }
    }
}

return result;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "25525511135";
    List<String> result1 = restoreIpAddresses(s1);
    System.out.println("输入: s = " + s1 + ")");
    System.out.println("输出: " + result1);

    // 测试用例 2
    String s2 = "0000";
    List<String> result2 = restoreIpAddresses(s2);
    System.out.println("\n输入: s = " + s2 + ")");
    System.out.println("输出: " + result2);

    // 测试用例 3
    String s3 = "101023";
    List<String> result3 = restoreIpAddresses(s3);
    System.out.println("\n输入: s = " + s3 + ")");
    System.out.println("输出: " + result3);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ===");
    List<String> result4 = restoreIpAddresses2(s1);
    System.out.println("输入: s = " + s1 + ")");
    System.out.println("输出: " + result4);
}
```

=====

文件: Code24\_RestoreIPAddresses.py

```
=====
```

```
from typing import List
```

```
class Solution:
```

```
    """
```

```
    LeetCode 93. 复原 IP 地址
```

题目描述:

有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 ‘.’ 分隔。

例如：“0.1.2.201” 和 “192.168.1.1” 是 有效 IP 地址，

但是 “0.011.255.245”、“192.168.1.312” 和 “192.168@1.1” 是 无效 IP 地址。

给定一个只包含数字的字符串 s，用以表示一个 IP 地址，返回所有可能的有效 IP 地址。

示例:

输入: s = "25525511135"

输出: ["255.255.11.135", "255.255.111.35"]

输入: s = "0000"

输出: ["0.0.0.0"]

输入: s = "101023"

输出: ["1.0.10.23", "1.0.102.3", "10.1.0.23", "10.10.2.3", "101.0.2.3"]

提示:

1 <= s.length <= 20

s 仅由数字组成

链接: <https://leetcode.cn/problems/restore-ip-addresses/>

算法思路:

1. 使用回溯算法分割字符串为 4 个部分
2. 每个部分必须满足: 0-255 之间，不能有前导 0 (除非是 0 本身)
3. 当分割完成 4 个部分且字符串用完时，加入结果集
4. 使用剪枝优化：剩余字符串长度不足以填满剩余部分时提前终止

时间复杂度:  $O(3^4) = O(81)$ ，每个部分最多 3 位数字，共 4 个部分

空间复杂度:  $O(n)$ ，递归栈深度

```
"""
```

```
def restoreIpAddresses(self, s: str) -> List[str]:
```

```

result = []
self.backtrack(s, 0, [], result)
return result

def backtrack(self, s: str, start: int, path: List[str], result: List[str]) -> None:
    # 终止条件：已经分割成 4 个部分
    if len(path) == 4:
        # 如果字符串刚好用完，加入结果集
        if start == len(s):
            result.append('.'.join(path))
        return

    # 剪枝：剩余字符串长度不足以填满剩余部分
    # 剩余部分数：4 - len(path)
    # 每个部分最少 1 位，最多 3 位
    min_remaining = 4 - len(path)
    max_remaining = 3 * (4 - len(path))
    remaining_length = len(s) - start

    if remaining_length < min_remaining or remaining_length > max_remaining:
        return

    # 尝试取 1 位、2 位、3 位数字
    for length in range(1, 4):
        # 检查是否超出字符串长度
        if start + length > len(s):
            break

        # 获取当前部分
        segment = s[start:start + length]

        # 检查是否有效
        if self.is_valid_segment(segment):
            path.append(segment)
            self.backtrack(s, start + length, path, result)
            path.pop()

def is_valid_segment(self, segment: str) -> bool:
    # 检查长度
    if len(segment) == 0 or len(segment) > 3:
        return False

    # 检查前导 0

```

```

if len(segment) > 1 and segment[0] == '0':
    return False

# 检查数值范围
num = int(segment)
return 0 <= num <= 255

def test_restore_ip_addresses():
    solution = Solution()

    # 测试用例 1
    s1 = "25525511135"
    result1 = solution.restoreIpAddresses(s1)
    print(f'输入: s = "{s1}"')
    print("输出:", result1)

    # 测试用例 2
    s2 = "0000"
    result2 = solution.restoreIpAddresses(s2)
    print(f'\n输入: s = "{s2}"')
    print("输出:", result2)

    # 测试用例 3
    s3 = "101023"
    result3 = solution.restoreIpAddresses(s3)
    print(f'\n输入: s = "{s3}"')
    print("输出:", result3)

if __name__ == "__main__":
    test_restore_ip_addresses()

```

=====

文件: Code25\_WordBreakII.java

=====

```

package class038;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**

```

\* LeetCode 140. 单词拆分 II

\*

\* 题目描述:

\* 给定一个字符串 s 和一个字符串字典 wordDict, 在字符串 s 中增加空格来构建一个句子,

\* 使得句子中所有的单词都在词典中。返回所有这些可能的句子。

\*

\* 示例:

\* 输入: s = "catsanddog", wordDict = ["cat", "cats", "and", "sand", "dog"]

\* 输出: ["cats and dog", "cat sand dog"]

\*

\* 输入: s = "pineapplepenapple", wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]

\* 输出: ["pine apple pen apple", "pineapple pen apple", "pine applepen apple"]

\*

\* 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

\* 输出: []

\*

\* 提示:

\*  $1 \leq s.length \leq 20$

\*  $1 \leq wordDict.length \leq 1000$

\*  $1 \leq wordDict[i].length \leq 10$

\* s 和 wordDict[i] 仅有小写英文字母组成

\* wordDict 中的所有字符串互不相同

\*

\* 链接: <https://leetcode.cn/problems/word-break-ii/>

\*

\* 算法思路:

\* 1. 使用回溯算法生成所有可能的分割方案

\* 2. 结合记忆化搜索优化重复计算

\* 3. 对于每个位置, 尝试所有可能的单词分割

\* 4. 如果当前分割的单词在字典中, 递归处理剩余部分

\*

\* 时间复杂度:  $O(2^n * n)$ , 最坏情况下需要尝试所有可能的分割

\* 空间复杂度:  $O(n^2)$ , 递归栈深度和存储结果的空间

\*/

```
public class Code25_WordBreakII {
```

```
    /**
     * 单词拆分 II
     *
     * @param s 输入字符串
     * @param wordDict 单词字典
     * @return 所有可能的句子
    */
```

```

public static List<String> wordBreak(String s, List<String> wordDict) {
    Set<String> dict = new HashSet<>(wordDict);
    return backtrack(s, 0, dict, new HashMap<>());
}

/**
 * 回溯函数生成句子（带记忆化）
 *
 * @param s 输入字符串
 * @param start 当前起始位置
 * @param dict 单词字典
 * @param memo 记忆化存储
 * @return 从 start 开始的所有可能句子
 */
private static List<String> backtrack(String s, int start, Set<String> dict, Map<Integer, List<String>> memo) {
    // 如果已经计算过，直接返回结果
    if (memo.containsKey(start)) {
        return memo.get(start);
    }

    List<String> result = new ArrayList<>();

    // 终止条件：到达字符串末尾
    if (start == s.length()) {
        result.add("");
        // 添加空字符串作为基础
        return result;
    }

    // 尝试所有可能的分割点
    for (int end = start + 1; end <= s.length(); end++) {
        String word = s.substring(start, end);

        // 如果当前单词在字典中
        if (dict.contains(word)) {
            // 递归处理剩余部分
            List<String> subSentences = backtrack(s, end, dict, memo);

            // 将当前单词与子句组合
            for (String subSentence : subSentences) {
                if (subSentence.isEmpty()) {
                    result.add(word);
                } else {

```

```

        result.add(word + " " + subSentence);
    }
}
}

// 存储结果到记忆化表
memo.put(start, result);
return result;
}

/***
 * 解法二：使用动态规划预处理 + 回溯
 * 先使用动态规划判断是否可分割，再进行回溯
 *
 * @param s 输入字符串
 * @param wordDict 单词字典
 * @return 所有可能的句子
 */
public static List<String> wordBreak2(String s, List<String> wordDict) {
    Set<String> dict = new HashSet<>(wordDict);

    // 使用动态规划预处理，判断是否可分割
    int n = s.length();
    boolean[] dp = new boolean[n + 1];
    dp[0] = true;

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < i; j++) {
            if (dp[j] && dict.contains(s.substring(j, i))) {
                dp[i] = true;
                break;
            }
        }
    }
}

// 如果不可分割，直接返回空列表
if (!dp[n]) {
    return new ArrayList<>();
}

// 使用回溯生成所有句子
List<String> result = new ArrayList<>();

```

```

backtrack2(s, 0, dict, new StringBuilder(), result, dp);
return result;
}

private static void backtrack2(String s, int start, Set<String> dict, StringBuilder path,
List<String> result, boolean[] dp) {
    // 终止条件：到达字符串末尾
    if (start == s.length()) {
        result.add(path.toString().trim());
        return;
    }

    // 尝试所有可能的分割点
    for (int end = start + 1; end <= s.length(); end++) {
        String word = s.substring(start, end);

        // 如果当前单词在字典中且剩余部分可分割
        if (dict.contains(word) && dp[end]) {
            int originalLength = path.length();

            // 添加当前单词到路径
            if (path.length() > 0) {
                path.append(" ");
            }
            path.append(word);

            // 递归处理剩余部分
            backtrack2(s, end, dict, path, result, dp);

            // 回溯
            path.setLength(originalLength);
        }
    }
}

/**
 * 解法三：纯回溯算法（无优化）
 * 适用于小规模数据
 *
 * @param s 输入字符串
 * @param wordDict 单词字典
 * @return 所有可能的句子
 */

```

```

public static List<String> wordBreak3(String s, List<String> wordDict) {
    Set<String> dict = new HashSet<>(wordDict);
    List<String> result = new ArrayList<>();
    backtrack3(s, 0, dict, new ArrayList<>(), result);
    return result;
}

private static void backtrack3(String s, int start, Set<String> dict, List<String> path,
List<String> result) {
    // 终止条件：到达字符串末尾
    if (start == s.length()) {
        result.add(String.join(" ", path));
        return;
    }

    // 尝试所有可能的分割点
    for (int end = start + 1; end <= s.length(); end++) {
        String word = s.substring(start, end);

        // 如果当前单词在字典中
        if (dict.contains(word)) {
            path.add(word);
            backtrack3(s, end, dict, path, result);
            path.remove(path.size() - 1); // 回溯
        }
    }
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "catsanddog";
    List<String> wordDict1 = List.of("cat", "cats", "and", "sand", "dog");
    List<String> result1 = wordBreak(s1, wordDict1);
    System.out.println("输入: s = " + s1 + "\\", wordDict = " + wordDict1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    String s2 = "pineapplepenapple";
    List<String> wordDict2 = List.of("apple", "pen", "applepen", "pine", "pineapple");
    List<String> result2 = wordBreak(s2, wordDict2);
    System.out.println("\n输入: s = " + s2 + "\\", wordDict = " + wordDict2);
    System.out.println("输出: " + result2);
}

```

```

// 测试用例 3
String s3 = "catsanddog";
List<String> wordDict3 = List.of("cats", "dog", "sand", "and", "cat");
List<String> result3 = wordBreak(s3, wordDict3);
System.out.println("\n 输入: s = " + s3 + "\\", wordDict = " + wordDict3);
System.out.println("输出: " + result3);

// 测试解法二
System.out.println("\n==== 解法二测试 ====");
List<String> result4 = wordBreak2(s1, wordDict1);
System.out.println("输入: s = " + s1 + "\\", wordDict = " + wordDict1);
System.out.println("输出: " + result4);
}
}
=====
```

文件: Code25\_WordBreakII.py

```

from typing import List

class Solution:
    """
    LeetCode 140. 单词拆分 II
```

题目描述:

给定一个字符串  $s$  和一个字符串字典  $\text{wordDict}$ , 在字符串  $s$  中增加空格来构建一个句子, 使得句子中所有的单词都在词典中。返回所有这些可能的句子。

示例:

输入:  $s = \text{"catsanddog"}$ ,  $\text{wordDict} = [\text{"cat"}, \text{"cats"}, \text{"and"}, \text{"sand"}, \text{"dog"}]$   
 输出:  $[\text{"cats and dog"}, \text{"cat sand dog"}]$

输入:  $s = \text{"pineapplepenapple"}$ ,  $\text{wordDict} = [\text{"apple"}, \text{"pen"}, \text{"applepen"}, \text{"pine"}, \text{"pineapple"}]$   
 输出:  $[\text{"pine apple pen apple"}, \text{"pineapple pen apple"}, \text{"pine applepen apple"}]$

输入:  $s = \text{"catsandog"}$ ,  $\text{wordDict} = [\text{"cats"}, \text{"dog"}, \text{"sand"}, \text{"and"}, \text{"cat"}]$   
 输出:  $[]$

提示:

$1 \leq s.length \leq 20$   
 $1 \leq \text{wordDict.length} \leq 1000$

$1 \leq \text{wordDict}[i].\text{length} \leq 10$   
s 和 wordDict[i] 仅有小写英文字母组成  
wordDict 中的所有字符串互不相同

链接: <https://leetcode.cn/problems/word-break-ii/>

算法思路:

1. 使用回溯算法分割字符串
2. 使用记忆化搜索优化，避免重复计算
3. 对于每个位置，尝试所有可能的单词分割
4. 当分割到字符串末尾时，将结果加入结果集

时间复杂度:  $O(2^n * n)$ , 最坏情况下需要尝试所有分割方式

空间复杂度:  $O(n^2)$ , 记忆化存储的空间

"""

```
def wordBreak(self, s: str, wordDict: List[str]) -> List[str]:  
    word_set = set(wordDict)  
    memo = {}  
    return self.backtrack(s, 0, word_set, memo)  
  
def backtrack(self, s: str, start: int, word_set: set, memo: dict) -> List[str]:  
    # 如果已经计算过这个位置的结果，直接返回  
    if start in memo:  
        return memo[start]  
  
    # 如果已经到达字符串末尾，返回空列表（表示一个有效的分割结束）  
    if start == len(s):  
        return [""]  
  
    result = []  
  
    # 尝试所有可能的分割点  
    for end in range(start + 1, len(s) + 1):  
        word = s[start:end]  
  
        # 如果当前单词在字典中  
        if word in word_set:  
            # 递归处理剩余部分  
            sub_results = self.backtrack(s, end, word_set, memo)  
  
            # 将当前单词与子结果组合  
            for sub_result in sub_results:  
                result.append(word + " " + sub_result)  
  
    memo[start] = result  
    return result
```

```

        if sub_result:
            result.append(word + " " + sub_result)
        else:
            result.append(word)

    # 记忆化存储结果
    memo[start] = result
    return result

def test_word_break_ii():
    solution = Solution()

    # 测试用例 1
    s1 = "catsanddog"
    wordDict1 = ["cat", "cats", "and", "sand", "dog"]
    result1 = solution.wordBreak(s1, wordDict1)
    print(f'输入: s = "{s1}", wordDict = {wordDict1}')
    print("输出:", result1)

    # 测试用例 2
    s2 = "pineapplepenapple"
    wordDict2 = ["apple", "pen", "applepen", "pine", "pineapple"]
    result2 = solution.wordBreak(s2, wordDict2)
    print(f'\n输入: s = "{s2}", wordDict = {wordDict2}')
    print("输出:", result2)

    # 测试用例 3
    s3 = "catsandog"
    wordDict3 = ["cats", "dog", "sand", "and", "cat"]
    result3 = solution.wordBreak(s3, wordDict3)
    print(f'\n输入: s = "{s3}", wordDict = {wordDict3}')
    print("输出:", result3)

if __name__ == "__main__":
    test_word_break_ii()

```

=====

文件: Code26\_BeautifulArrangement.java

=====

package class038;

import java.util.ArrayList;

```
import java.util.List;

/**
 * LeetCode 526. 优美的排列
 *
 * 题目描述:
 * 假设有从 1 到 n 的 n 个整数。用这些整数构造一个数组 perm (下标从 1 开始),
 * 只要满足下述条件之一, 该数组就是一个优美的排列:
 * 1. perm[i] 能够被 i 整除
 * 2. i 能够被 perm[i] 整除
 * 给你一个整数 n , 返回可以构造的优美排列的数量。
 *
 * 示例:
 * 输入: n = 2
 * 输出: 2
 * 解释: 第 1 个优美的排列是 [1,2]; 第 2 个优美的排列是 [2,1]
 *
 * 输入: n = 1
 * 输出: 1
 *
 * 提示:
 * 1 <= n <= 15
 *
 * 链接: https://leetcode.cn/problems/beautiful-arrangement/
 *
 * 算法思路:
 * 1. 使用回溯算法生成所有可能的排列
 * 2. 在生成排列的过程中进行剪枝: 只有满足优美排列条件的数字才被选择
 * 3. 使用布尔数组标记已使用的数字
 * 4. 通过提前终止无效分支优化性能
 *
 * 时间复杂度: O(n!), 但通过剪枝可以大大减少实际计算量
 * 空间复杂度: O(n), 递归栈深度
 */
public class Code26_BeautifulArrangement {

    private static int count = 0;

    /**
     * 计算优美排列的数量
     *
     * @param n 数字个数
     * @return 优美排列的数量
    
```

```

*/
public static int countArrangement(int n) {
    count = 0;
    boolean[] used = new boolean[n + 1];
    backtrack(n, 1, used);
    return count;
}

/***
 * 回溯函数生成优美排列
 *
 * @param n 数字个数
 * @param pos 当前位置（从 1 开始）
 * @param used 标记已使用数字的数组
 */
private static void backtrack(int n, int pos, boolean[] used) {
    // 终止条件：已生成完整排列
    if (pos > n) {
        count++;
        return;
    }

    // 尝试所有可能的数字
    for (int num = 1; num <= n; num++) {
        if (!used[num] && (num % pos == 0 || pos % num == 0)) {
            used[num] = true;
            backtrack(n, pos + 1, used);
            used[num] = false; // 回溯
        }
    }
}

/***
 * 解法二：返回所有优美的排列（而不仅仅是数量）
 *
 * @param n 数字个数
 * @return 所有优美的排列
 */
public static List<List<Integer>> getBeautifulArrangements(int n) {
    List<List<Integer>> result = new ArrayList<>();
    boolean[] used = new boolean[n + 1];
    backtrackWithResult(n, 1, used, new ArrayList<>(), result);
    return result;
}

```

```

}

private static void backtrackWithResult(int n, int pos, boolean[] used, List<Integer> path,
List<List<Integer>> result) {
    // 终止条件：已生成完整排列
    if (pos > n) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 尝试所有可能的数字
    for (int num = 1; num <= n; num++) {
        if (!used[num] && (num % pos == 0 || pos % num == 0)) {
            used[num] = true;
            path.add(num);
            backtrackWithResult(n, pos + 1, used, path, result);
            path.remove(path.size() - 1); // 回溯
            used[num] = false;
        }
    }
}

/***
 * 解法三：使用位运算优化空间复杂度
 * 使用整数位掩码代替布尔数组
 *
 * @param n 数字个数
 * @return 优美排列的数量
 */
public static int countArrangementBitmask(int n) {
    return backtrackBitmask(n, 1, 0);
}

private static int backtrackBitmask(int n, int pos, int used) {
    // 终止条件：已生成完整排列
    if (pos > n) {
        return 1;
    }

    int count = 0;

    // 尝试所有可能的数字
    for (int num = 1; num <= n; num++) {

```

```

        int mask = 1 << num;
        if ((used & mask) == 0 && (num % pos == 0 || pos % num == 0)) {
            count += backtrackBitmask(n, pos + 1, used | mask);
        }
    }

    return count;
}

/***
 * 解法四：使用动态规划 + 状态压缩
 * 适用于需要高效计算的情况
 *
 * @param n 数字个数
 * @return 优美排列的数量
 */
public static int countArrangementDP(int n) {
    int totalStates = 1 << n;
    int[] dp = new int[totalStates];
    dp[0] = 1;

    for (int state = 0; state < totalStates; state++) {
        int pos = Integer.bitCount(state) + 1; // 当前位置

        for (int num = 1; num <= n; num++) {
            int mask = 1 << (num - 1);
            if ((state & mask) == 0 && (num % pos == 0 || pos % num == 0)) {
                dp[state | mask] += dp[state];
            }
        }
    }

    return dp[totalStates - 1];
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 2;
    int result1 = countArrangement(n1);
    System.out.println("输入: n = " + n1);
    System.out.println("输出: " + result1);
}

```

```

// 测试用例 2
int n2 = 1;
int result2 = countArrangement(n2);
System.out.println("\n输入: n = " + n2);
System.out.println("输出: " + result2);

// 测试用例 3
int n3 = 3;
int result3 = countArrangement(n3);
System.out.println("\n输入: n = " + n3);
System.out.println("输出: " + result3);

// 测试返回所有排列
System.out.println("\n== 所有优美排列 ==");
List<List<Integer>> arrangements = getBeautifulArrangements(n1);
System.out.println("输入: n = " + n1);
System.out.println("输出: " + arrangements);

// 测试位运算解法
System.out.println("\n== 位运算解法测试 ==");
int result4 = countArrangementBitmask(n1);
System.out.println("输入: n = " + n1);
System.out.println("输出: " + result4);

// 测试动态规划解法
System.out.println("\n== 动态规划解法测试 ==");
int result5 = countArrangementDP(n1);
System.out.println("输入: n = " + n1);
System.out.println("输出: " + result5);
}

}
=====

文件: Code26_BeautifulArrangement.py
=====

class Solution:
    """
    LeetCode 526. 优美的排列

```

#### 题目描述:

假设有从 1 到 n 的 n 个整数。用这些整数构造一个数组 perm (下标从 1 开始)，只要满足下述条件之一，该数组就是一个优美的排列：

1.  $\text{perm}[i]$  能够被  $i$  整除
2.  $i$  能够被  $\text{perm}[i]$  整除

给你一个整数  $n$ ，返回可以构造的优美排列的数量。

示例：

输入：  $n = 2$

输出： 2

解释： 第 1 个优美的排列是  $[1, 2]$ :  $\text{perm}[1] = 1$  能被  $i=1$  整除， $\text{perm}[2] = 2$  能被  $i=2$  整除  
第 2 个优美的排列是  $[2, 1]$ :  $\text{perm}[1] = 2$  能被  $i=1$  整除， $i=2$  能被  $\text{perm}[2] = 1$  整除

输入：  $n = 1$

输出： 1

提示：

$1 \leq n \leq 15$

链接：<https://leetcode.cn/problems/beautiful-arrangement/>

算法思路：

1. 使用回溯算法生成所有可能的排列
2. 在生成排列的过程中，提前剪枝：如果当前数字不满足优美排列的条件，则跳过
3. 使用布尔数组标记已使用的数字
4. 当排列完成时，计数加 1

时间复杂度： $O(n!)$ ，需要生成所有排列

空间复杂度： $O(n)$ ，递归栈深度和标记数组

"""

```
def countArrangement(self, n: int) -> int:  
    self.count = 0  
    used = [False] * (n + 1)  
    self.backtrack(n, 1, used)  
    return self.count  
  
def backtrack(self, n: int, pos: int, used: list) -> None:  
    # 终止条件：已经排列完所有数字  
    if pos > n:  
        self.count += 1  
        return  
  
    for num in range(1, n + 1):  
        # 如果数字未被使用且满足优美排列条件  
        if not used[num] and (num % pos == 0 or pos % num == 0):  
            used[num] = True  
            self.backtrack(n, pos + 1, used)  
            used[num] = False
```

```
    if not used[num] and self.is_valid(pos, num):
        used[num] = True
        self.backtrack(n, pos + 1, used)
        used[num] = False

def is_valid(self, pos: int, num: int) -> bool:
    """检查位置 pos 放置数字 num 是否满足优美排列条件"""
    return num % pos == 0 or pos % num == 0

def test_beautiful_arrangement():
    solution = Solution()

    # 测试用例 1
    n1 = 2
    result1 = solution.countArrangement(n1)
    print(f"输入: n = {n1}")
    print("输出:", result1)

    # 测试用例 2
    n2 = 1
    result2 = solution.countArrangement(n2)
    print(f"\n输入: n = {n2}")
    print("输出:", result2)

    # 测试用例 3
    n3 = 3
    result3 = solution.countArrangement(n3)
    print(f"\n输入: n = {n3}")
    print("输出:", result3)

    # 测试用例 4
    n4 = 4
    result4 = solution.countArrangement(n4)
    print(f"\n输入: n = {n4}")
    print("输出:", result4)

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

=====

文件: Code27\_MatchsticksToSquare.java

=====

```
package class038;

import java.util.Arrays;

/**
 * LeetCode 473. 火柴拼正方形
 *
 * 题目描述:
 * 你将得到一个整数数组 matchsticks，其中 matchsticks[i] 是第 i 个火柴棒的长度。
 * 你要用所有的火柴棍拼成一个正方形。你不能折断任何一根火柴棒，但你可以把它们连接在一起，
 * 而且每根火柴棒必须使用一次。
 * 如果你能使这个正方形，则返回 true，否则返回 false。
 *
 * 示例:
 * 输入: matchsticks = [1,1,2,2,2]
 * 输出: true
 * 解释: 能拼成一个边长为 2 的正方形，每边两根火柴棒
 *
 * 输入: matchsticks = [3,3,3,3,4]
 * 输出: false
 * 解释: 不能用所有火柴棒拼成一个正方形
 *
 * 提示:
 * 1 <= matchsticks.length <= 15
 * 1 <= matchsticks[i] <= 10^8
 *
 * 链接: https://leetcode.cn/problems/matchsticks-to-square/
 *
 * 算法思路:
 * 1. 计算所有火柴棒的总长度，如果不能被 4 整除，直接返回 false
 * 2. 计算每条边的目标长度（总长度/4）
 * 3. 使用回溯算法尝试将火柴棒分配到四条边
 * 4. 使用剪枝优化：排序、提前终止等
 *
 * 时间复杂度: O(4^n)，其中 n 是火柴棒数量
 * 空间复杂度: O(n)，递归栈深度
 */
public class Code27_MatchsticksToSquare {

    /**
     * 判断是否能用火柴棒拼成正方形
     *
     * @param matchsticks 火柴棒长度数组
    
```

```

* @return 是否能拼成正方形
*/
public static boolean makesquare(int[] matchsticks) {
    // 计算总长度
    int total = 0;
    for (int stick : matchsticks) {
        total += stick;
    }

    // 如果不能被 4 整除，直接返回 false
    if (total % 4 != 0) {
        return false;
    }

    int sideLength = total / 4;

    // 排序（从大到小），便于剪枝
    Arrays.sort(matchsticks);
    reverse(matchsticks);

    // 初始化四条边的当前长度
    int[] sides = new int[4];

    return backtrack(matchsticks, 0, sides, sideLength);
}

/**
 * 回溯函数分配火柴棒
 *
 * @param matchsticks 火柴棒长度数组
 * @param index 当前处理的火柴棒索引
 * @param sides 四条边的当前长度
 * @param target 目标边长
 * @return 是否能成功分配
 */
private static boolean backtrack(int[] matchsticks, int index, int[] sides, int target) {
    // 终止条件：所有火柴棒都已分配
    if (index == matchsticks.length) {
        return sides[0] == target && sides[1] == target && sides[2] == target && sides[3] ==
target;
    }

    int currentStick = matchsticks[index];

```

```

// 尝试将当前火柴棒分配到四条边
for (int i = 0; i < 4; i++) {
    // 剪枝：如果当前边加上火柴棒长度超过目标值，跳过
    if (sides[i] + currentStick > target) {
        continue;
    }

    // 剪枝：如果当前边与前一条边长度相同，且前一条边分配失败，跳过
    // 这样可以避免重复计算相同的情况
    if (i > 0 && sides[i] == sides[i - 1]) {
        continue;
    }

    sides[i] += currentStick;
    if (backtrack(matchsticks, index + 1, sides, target)) {
        return true;
    }
    sides[i] -= currentStick; // 回溯
}

return false;
}

/**
 * 解法二：使用位运算 + 动态规划
 * 适用于需要高效计算的情况
 *
 * @param matchsticks 火柴棒长度数组
 * @return 是否能拼成正方形
 */
public static boolean makesquareDP(int[] matchsticks) {
    int total = 0;
    for (int stick : matchsticks) {
        total += stick;
    }

    if (total % 4 != 0) {
        return false;
    }

    int sideLength = total / 4;
    int n = matchsticks.length;

```

```

int totalStates = 1 << n;

// dp[mask] 表示使用 mask 对应的火柴棒能组成的边长模 sideLength 的余数
int[] dp = new int[totalStates];
Arrays.fill(dp, -1);
dp[0] = 0;

for (int mask = 0; mask < totalStates; mask++) {
    if (dp[mask] == -1) {
        continue;
    }

    for (int i = 0; i < n; i++) {
        // 如果第 i 根火柴棒还未使用
        if ((mask & (1 << i)) == 0) {
            int nextMask = mask | (1 << i);
            int remainder = dp[mask] + matchsticks[i];

            // 如果当前边长超过目标值，重置
            if (remainder > sideLength) {
                continue;
            }

            dp[nextMask] = (remainder == sideLength) ? 0 : remainder;
        }
    }
}

return dp[totalStates - 1] == 0;
}

/**
 * 解法三：使用 DFS + 剪枝优化
 * 更直观的实现方式
 *
 * @param matchsticks 火柴棒长度数组
 * @return 是否能拼成正方形
 */
public static boolean makesquareDFS(int[] matchsticks) {
    int total = 0;
    for (int stick : matchsticks) {
        total += stick;
    }
}

```

```
if (total % 4 != 0) {
    return false;
}

int sideLength = total / 4;

// 排序（从大到小）
Arrays.sort(matchsticks);
reverse(matchsticks);

// 如果有火柴棒长度大于边长，直接返回 false
if (matchsticks[0] > sideLength) {
    return false;
}

return dfs(matchsticks, new int[4], 0, sideLength);
}

private static boolean dfs(int[] matchsticks, int[] sides, int index, int target) {
    if (index == matchsticks.length) {
        return true;
    }

    for (int i = 0; i < 4; i++) {
        if (sides[i] + matchsticks[index] <= target) {
            sides[i] += matchsticks[index];
            if (dfs(matchsticks, sides, index + 1, target)) {
                return true;
            }
            sides[i] -= matchsticks[index];
        }
    }

    // 剪枝：如果当前边长度为 0，且分配失败，说明无法成功
    if (sides[i] == 0) {
        break;
    }
}

return false;
}

// 反转数组（从大到小排序）
```

```
private static void reverse(int[] arr) {  
    int left = 0, right = arr.length - 1;  
    while (left < right) {  
        int temp = arr[left];  
        arr[left] = arr[right];  
        arr[right] = temp;  
        left++;  
        right--;  
    }  
}  
  
// 测试方法  
public static void main(String[] args) {  
    // 测试用例 1  
    int[] matchsticks1 = {1, 1, 2, 2, 2};  
    boolean result1 = makesquare(matchsticks1);  
    System.out.println("输入: matchsticks = [1,1,2,2,2]");  
    System.out.println("输出: " + result1);  
  
    // 测试用例 2  
    int[] matchsticks2 = {3, 3, 3, 3, 4};  
    boolean result2 = makesquare(matchsticks2);  
    System.out.println("\n 输入: matchsticks = [3,3,3,3,4]");  
    System.out.println("输出: " + result2);  
  
    // 测试用例 3  
    int[] matchsticks3 = {5, 5, 5, 5, 4, 4, 4, 4, 3, 3, 3, 3};  
    boolean result3 = makesquare(matchsticks3);  
    System.out.println("\n 输入: matchsticks = [5,5,5,5,4,4,4,4,3,3,3,3]");  
    System.out.println("输出: " + result3);  
  
    // 测试动态规划解法  
    System.out.println("\n==== 动态规划解法测试 ===");  
    boolean result4 = makesquareDP(matchsticks1);  
    System.out.println("输入: matchsticks = [1,1,2,2,2]");  
    System.out.println("输出: " + result4);  
}
```

=====

文件: Code27\_MatchsticksToSquare.py

=====

```
from typing import List
```

```
class Solution:
```

```
    """
```

```
LeetCode 473. 火柴拼正方形
```

题目描述：

你将得到一个整数数组 `matchsticks`，其中 `matchsticks[i]` 是第 `i` 个火柴棒的长度。

你要用所有的火柴棍拼成一个正方形。你不能折断任何一根火柴棒，但你可以把它们连在一起，而且每根火柴棒必须使用一次。

如果你能使这个正方形，则返回 `true`，否则返回 `false`。

示例：

输入： `matchsticks = [1, 1, 2, 2, 2]`

输出： `true`

解释： 可以拼成一个边长为 2 的正方形，每边两根火柴

输入： `matchsticks = [3, 3, 3, 3, 4]`

输出： `false`

解释： 不能用所有火柴拼成一个正方形

提示：

`1 <= matchsticks.length <= 15`

`0 <= matchsticks[i] <= 10^9`

链接： <https://leetcode.cn/problems/matchsticks-to-square/>

算法思路：

1. 计算所有火柴的总长度，如果不能被 4 整除，直接返回 `false`
2. 计算每条边的目标长度 = 总长度 / 4
3. 将火柴从大到小排序，优先使用长火柴可以提前剪枝
4. 使用回溯算法尝试将火柴分配到四条边
5. 使用剪枝优化：如果当前边长度超过目标长度，提前终止

时间复杂度： $O(4^n)$ ，最坏情况下需要尝试所有分配方式

空间复杂度： $O(n)$ ，递归栈深度

```
"""
```

```
def makesquare(self, matchsticks: List[int]) -> bool:
```

```
    total = sum(matchsticks)
```

```
    # 如果总长度不能被 4 整除，直接返回 false
```

```
    if total % 4 != 0:
```

```

        return False

# 目标边长
target = total // 4

# 从大到小排序，优先使用长火柴
matchsticks.sort(reverse=True)

# 如果最长的火柴大于目标边长，直接返回 false
if matchsticks[0] > target:
    return False

# 四条边的当前长度
sides = [0, 0, 0, 0]

return self.backtrack(matchsticks, 0, sides, target)

def backtrack(self, matchsticks: List[int], index: int, sides: List[int], target: int) ->
bool:
    # 终止条件：所有火柴都已分配
    if index == len(matchsticks):
        # 检查四条边是否都等于目标长度
        return all(side == target for side in sides)

    # 尝试将当前火柴分配到四条边
    for i in range(4):
        # 剪枝：如果当前边长度加上当前火柴长度超过目标长度，跳过
        if sides[i] + matchsticks[index] > target:
            continue

        # 剪枝：如果当前边与前一条边长度相同，且前一条边没有分配当前火柴，跳过
        # 避免重复计算相同的情况
        if i > 0 and sides[i] == sides[i - 1]:
            continue

        sides[i] += matchsticks[index]
        if self.backtrack(matchsticks, index + 1, sides, target):
            return True
        sides[i] -= matchsticks[index]

    return False

def test_matchsticks_to_square():

```

```

solution = Solution()

# 测试用例 1
matchsticks1 = [1, 1, 2, 2, 2]
result1 = solution.makesquare(matchsticks1)
print("输入: matchsticks = [1, 1, 2, 2, 2]")
print("输出:", result1)

# 测试用例 2
matchsticks2 = [3, 3, 3, 3, 4]
result2 = solution.makesquare(matchsticks2)
print("\n 输入: matchsticks = [3, 3, 3, 3, 4]")
print("输出:", result2)

# 测试用例 3
matchsticks3 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] # 12 根长度为 1 的火柴
result3 = solution.makesquare(matchsticks3)
print("\n 输入: matchsticks = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]")
print("输出:", result3)

if __name__ == "__main__":
    test_matchsticks_to_square()

```

---

文件: Code28\_PartitionToKEqualSumSubsets.java

---

```

package class038;

import java.util.Arrays;

/**
 * LeetCode 698. 划分为 k 个相等的子集
 *
 * 题目描述:
 * 给定一个整数数组 nums 和一个正整数 k, 找出是否有可能把这个数组分成 k 个非空子集, 其总和都相等。
 *
 * 示例:
 * 输入: nums = [4, 3, 2, 3, 5, 2, 1], k = 4
 * 输出: true
 * 解释: 有可能将其分成 4 个子集 (5), (1,4), (2,3), (2,3) 等于总和。
 */

```

```
* 输入: nums = [1, 2, 3, 4], k = 3
* 输出: false
*
* 提示:
* 1 <= k <= len(nums) <= 16
* 0 < nums[i] < 10000
* 每个元素的频率在 [1, 4] 范围内
*
* 链接: https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/
*
* 算法思路:
* 1. 计算数组总和, 如果不能被 k 整除, 直接返回 false
* 2. 计算每个子集的目标和 (总和/k)
* 3. 使用回溯算法尝试将元素分配到 k 个子集
* 4. 使用剪枝优化: 排序、提前终止等
*
* 时间复杂度: O(k^n), 其中 n 是数组长度
* 空间复杂度: O(n), 递归栈深度
*/

```

```
public class Code28_PartitionToKEqualSumSubsets {

    /**
     * 判断是否能将数组划分为 k 个和相等的子集
     *
     * @param nums 整数数组
     * @param k 子集个数
     * @return 是否能成功划分
     */
    public static boolean canPartitionKSubsets(int[] nums, int k) {
        // 计算总和
        int total = 0;
        for (int num : nums) {
            total += num;
        }

        // 如果不能被 k 整除, 直接返回 false
        if (total % k != 0) {
            return false;
        }

        int target = total / k;

        // 排序 (从大到小), 便于剪枝
    }
}
```

```

Arrays.sort(nums);
reverse(nums);

// 如果有元素大于目标值，直接返回 false
if (nums[0] > target) {
    return false;
}

// 初始化 k 个子集的当前和
int[] subsets = new int[k];

return backtrack(nums, 0, subsets, target);
}

/**
 * 回溯函数分配元素到子集
 *
 * @param nums 整数数组
 * @param index 当前处理的元素索引
 * @param subsets k 个子集的当前和
 * @param target 目标和
 * @return 是否能成功分配
 */
private static boolean backtrack(int[] nums, int index, int[] subsets, int target) {
    // 终止条件：所有元素都已分配
    if (index == nums.length) {
        // 检查所有子集的和是否都等于目标值
        for (int sum : subsets) {
            if (sum != target) {
                return false;
            }
        }
        return true;
    }

    int currentNum = nums[index];

    // 尝试将当前元素分配到 k 个子集
    for (int i = 0; i < subsets.length; i++) {
        // 剪枝：如果当前子集加上元素超过目标值，跳过
        if (subsets[i] + currentNum > target) {
            continue;
        }
    }
}

```

```

// 剪枝：如果当前子集与前一个子集相同，且前一个子集分配失败，跳过
// 这样可以避免重复计算相同的情况
if (i > 0 && subsets[i] == subsets[i - 1]) {
    continue;
}

subsets[i] += currentNum;
if (backtrack(nums, index + 1, subsets, target)) {
    return true;
}
subsets[i] -= currentNum; // 回溯

// 剪枝：如果当前子集和为 0，且分配失败，说明无法成功
if (subsets[i] == 0) {
    break;
}
}

return false;
}

/***
 * 解法二：使用位运算 + 动态规划
 * 适用于需要高效计算的情况
 *
 * @param nums 整数数组
 * @param k 子集个数
 * @return 是否能成功划分
 */
public static boolean canPartitionKSubsetsDP(int[] nums, int k) {
    int total = 0;
    for (int num : nums) {
        total += num;
    }

    if (total % k != 0) {
        return false;
    }

    int target = total / k;
    int n = nums.length;
    int totalStates = 1 << n;

```

```

// dp[mask] 表示使用 mask 对应的元素能组成的和模 target 的余数
int[] dp = new int[totalStates];
Arrays.fill(dp, -1);
dp[0] = 0;

for (int mask = 0; mask < totalStates; mask++) {
    if (dp[mask] == -1) {
        continue;
    }

    for (int i = 0; i < n; i++) {
        // 如果第 i 个元素还未使用
        if ((mask & (1 << i)) == 0) {
            int nextMask = mask | (1 << i);
            int remainder = dp[mask] + nums[i];

            // 如果当前和超过目标值，跳过
            if (remainder > target) {
                continue;
            }

            dp[nextMask] = (remainder == target) ? 0 : remainder;
        }
    }
}

return dp[totalStates - 1] == 0;
}

/**
 * 解法三：使用 DFS + 剪枝优化
 * 更直观的实现方式
 *
 * @param nums 整数数组
 * @param k 子集个数
 * @return 是否能成功划分
 */
public static boolean canPartitionKSubsetsDFS(int[] nums, int k) {
    int total = 0;
    for (int num : nums) {
        total += num;
    }

    if (total % k != 0) {
        return false;
    }

    int[] dp = new int[k];
    Arrays.fill(dp, -1);
    dp[0] = total;
    return dfs(0, k, dp, total / k);
}

private boolean dfs(int index, int k, int[] dp, int target) {
    if (k == 1) {
        return true;
    }

    if (index == dp.length) {
        return false;
    }

    if (dp[index] != -1) {
        return dp[index] == target;
    }

    for (int i = 0; i < k; i++) {
        if (i > 0 && dp[i] == dp[i - 1]) {
            continue;
        }

        if (dp[i] + nums[index] > target) {
            continue;
        }

        dp[i] += nums[index];
        if (dfs(index + 1, k, dp, target)) {
            return true;
        }
        dp[i] -= nums[index];
    }

    dp[index] = -1;
    return false;
}

```

```

if (total % k != 0) {
    return false;
}

int target = total / k;

// 排序（从大到小）
Arrays.sort(nums);
reverse(nums);

// 如果有元素大于目标值，直接返回 false
if (nums[0] > target) {
    return false;
}

boolean[] used = new boolean[nums.length];
return dfs(nums, used, 0, k, 0, target);
}

private static boolean dfs(int[] nums, boolean[] used, int start, int k, int currentSum, int target) {
    // 如果已经成功构建了 k-1 个子集，剩下的自然能构成第 k 个子集
    if (k == 1) {
        return true;
    }

    // 如果当前子集和等于目标值，开始构建下一个子集
    if (currentSum == target) {
        return dfs(nums, used, 0, k - 1, 0, target);
    }

    for (int i = start; i < nums.length; i++) {
        if (!used[i] && currentSum + nums[i] <= target) {
            used[i] = true;
            if (dfs(nums, used, i + 1, k, currentSum + nums[i], target)) {
                return true;
            }
            used[i] = false; // 回溯
        }
    }
}

return false;

```

```
}
```

```
// 反转数组（从大到小排序）
```

```
private static void reverse(int[] arr) {  
    int left = 0, right = arr.length - 1;  
    while (left < right) {  
        int temp = arr[left];  
        arr[left] = arr[right];  
        arr[right] = temp;  
        left++;  
        right--;  
    }  
}
```

```
// 测试方法
```

```
public static void main(String[] args) {  
    // 测试用例 1  
    int[] nums1 = {4, 3, 2, 3, 5, 2, 1};  
    int k1 = 4;  
    boolean result1 = canPartitionKSubsets(nums1, k1);  
    System.out.println("输入: nums = [4,3,2,3,5,2,1], k = " + k1);  
    System.out.println("输出: " + result1);
```

```
// 测试用例 2
```

```
int[] nums2 = {1, 2, 3, 4};  
int k2 = 3;  
boolean result2 = canPartitionKSubsets(nums2, k2);  
System.out.println("\n输入: nums = [1,2,3,4], k = " + k2);  
System.out.println("输出: " + result2);
```

```
// 测试用例 3
```

```
int[] nums3 = {2, 2, 2, 2, 3, 4, 5};  
int k3 = 4;  
boolean result3 = canPartitionKSubsets(nums3, k3);  
System.out.println("\n输入: nums = [2,2,2,2,3,4,5], k = " + k3);  
System.out.println("输出: " + result3);
```

```
// 测试动态规划解法
```

```
System.out.println("\n==== 动态规划解法测试 ===");  
boolean result4 = canPartitionKSubsetsDP(nums1, k1);  
System.out.println("输入: nums = [4,3,2,3,5,2,1], k = " + k1);  
System.out.println("输出: " + result4);  
}
```

}

=====

文件: Code28\_PartitionToKEqualSumSubsets.py

```
from typing import List
```

```
class Solution:
```

```
    """
```

LeetCode 698. 划分为 k 个相等的子集

题目描述:

给定一个整数数组 `nums` 和一个正整数 `k`, 找出是否有可能把数组分成 `k` 个非空子集, 其总和都相等。

示例:

输入: `nums = [4, 3, 2, 3, 5, 2, 1]`, `k = 4`

输出: `true`

解释: 可以将其分成 4 个子集 (5), (1,4), (2,3), (2,3) 等于总和。

输入: `nums = [1, 2, 3, 4]`, `k = 3`

输出: `false`

提示:

`1 <= k <= len(nums) <= 16`

`0 < nums[i] < 10000`

每个元素的频率在 `[1, 4]` 范围内

链接: <https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/>

算法思路:

1. 计算所有数字的总和, 如果不能被 `k` 整除, 直接返回 `false`
2. 计算每个子集的目标和 = 总和 / `k`
3. 将数字从大到小排序, 优先使用大数字可以提前剪枝
4. 使用回溯算法尝试将数字分配到 `k` 个子集
5. 使用剪枝优化: 如果当前子集和超过目标和, 提前终止

时间复杂度:  $O(k^n)$ , 最坏情况下需要尝试所有分配方式

空间复杂度:  $O(n)$ , 递归栈深度

```
"""
```

```
def canPartitionKSubsets(self, nums: List[int], k: int) -> bool:  
    total = sum(nums)
```

```

# 如果总和不能被 k 整除，直接返回 false
if total % k != 0:
    return False

# 目标和
target = total // k

# 从大到小排序，优先使用大数字
nums.sort(reverse=True)

# 如果最大的数字大于目标和，直接返回 false
if nums[0] > target:
    return False

# k 个子集的当前和
subsets = [0] * k

return self.backtrack(nums, 0, subsets, target)

def backtrack(self, nums: List[int], index: int, subsets: List[int], target: int) -> bool:
    # 终止条件：所有数字都已分配
    if index == len(nums):
        # 检查所有子集是否都等于目标和
        return all(subset == target for subset in subsets)

    # 尝试将当前数字分配到 k 个子集
    for i in range(len(subsets)):
        # 剪枝：如果当前子集和加上当前数字超过目标和，跳过
        if subsets[i] + nums[index] > target:
            continue

        # 剪枝：如果当前子集与前一个子集和相同，且前一个子集没有分配当前数字，跳过
        # 避免重复计算相同的情况
        if i > 0 and subsets[i] == subsets[i - 1]:
            continue

        subsets[i] += nums[index]
        if self.backtrack(nums, index + 1, subsets, target):
            return True
        subsets[i] -= nums[index]

return False

```

```

def test_partition_k_subsets():
    solution = Solution()

    # 测试用例 1
    nums1 = [4, 3, 2, 3, 5, 2, 1]
    k1 = 4
    result1 = solution.canPartitionKSubsets(nums1, k1)
    print("输入: nums = [4, 3, 2, 3, 5, 2, 1], k =", k1)
    print("输出:", result1)

    # 测试用例 2
    nums2 = [1, 2, 3, 4]
    k2 = 3
    result2 = solution.canPartitionKSubsets(nums2, k2)
    print("\n输入: nums = [1, 2, 3, 4], k =", k2)
    print("输出:", result2)

    # 测试用例 3
    nums3 = [1, 1, 1, 1, 2, 2, 2, 2]
    k3 = 4
    result3 = solution.canPartitionKSubsets(nums3, k3)
    print("\n输入: nums = [1, 1, 1, 1, 2, 2, 2, 2], k =", k3)
    print("输出:", result3)

if __name__ == "__main__":
    test_partition_k_subsets()

```

=====

文件: Code29\_AdditiveNumber.java

=====

```

package class038;


```

```

/***
 * LeetCode 306. 累加数
 *
 * 题目描述:
 * 累加数是一个字符串，组成它的数字可以形成累加序列。
 * 一个有效的累加序列必须至少包含 3 个数。除了最开始的两个数以外，字符串中的其他数都等于它之前两个数相加的和。
 * 给定一个只包含数字 '0'-'9' 的字符串，编写一个算法来判断给定输入是否是累加数。
 *

```

```
* 示例:  
* 输入: "112358"  
* 输出: true  
* 解释: 累加序列为: 1, 1, 2, 3, 5, 8。1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8  
*  
* 输入: "199100199"  
* 输出: true  
* 解释: 累加序列为: 1, 99, 100, 199。1 + 99 = 100, 99 + 100 = 199  
*  
* 提示:  
* 1 <= num.length <= 35  
* num 仅由数字 (0 - 9) 组成  
*  
* 链接: https://leetcode.cn/problems/additive-number/  
*  
* 算法思路:  
* 1. 使用回溯算法尝试所有可能的前两个数字分割方案  
* 2. 对于每个分割方案，验证后续数字是否满足累加关系  
* 3. 注意处理大数问题和前导零问题  
* 4. 使用剪枝优化：数字不能以 0 开头（除非数字本身就是 0）  
*  
* 时间复杂度: O(n^3)，需要枚举前两个数字的分割点  
* 空间复杂度: O(n)，递归栈深度  
*/
```

```
public class Code29_AdditiveNumber {  
  
    /**  
     * 判断字符串是否是累加数  
     *  
     * @param num 输入字符串  
     * @return 是否是累加数  
     */  
    public static boolean isAdditiveNumber(String num) {  
        int n = num.length();  
  
        // 尝试所有可能的前两个数字分割方案  
        for (int i = 1; i <= n / 2; i++) {  
            for (int j = 1; Math.max(i, j) <= n - i - j; j++) {  
                String num1 = num.substring(0, i);  
                String num2 = num.substring(i, i + j);  
  
                // 检查前两个数字是否合法  
                if (isValid(num1) && isValid(num2)) {  
                    // ...  
                }  
            }  
        }  
    }  
}
```

```

        if (backtrack(num, i + j, num1, num2)) {
            return true;
        }
    }
}

return false;
}

/***
 * 回溯函数验证后续数字
 *
 * @param num 原始字符串
 * @param start 当前起始位置
 * @param prev1 前一个数字（字符串形式）
 * @param prev2 当前数字（字符串形式）
 * @return 是否满足累加关系
 */
private static boolean backtrack(String num, int start, String prev1, String prev2) {
    // 终止条件：已处理完所有字符
    if (start == num.length()) {
        return true;
    }

    // 计算期望的下一个数字
    String expected = addStrings(prev1, prev2);

    // 检查剩余字符串是否以期望数字开头
    if (start + expected.length() <= num.length() &&
        num.substring(start, start + expected.length()).equals(expected)) {
        // 递归验证后续数字
        return backtrack(num, start + expected.length(), prev2, expected);
    }

    return false;
}

/***
 * 检查数字字符串是否合法
 * 数字不能以 0 开头，除非数字本身就是 0
 *
 * @param numStr 数字字符串
 */

```

```

* @return 是否合法
*/
private static boolean isValid(String numStr) {
    // 如果长度大于 1 且以 0 开头，不合法
    if (numStr.length() > 1 && numStr.charAt(0) == '0') {
        return false;
    }
    return true;
}

/**
* 大数加法：字符串形式的大数相加
*
* @param num1 第一个数字字符串
* @param num2 第二个数字字符串
* @return 相加结果的字符串形式
*/
private static String addStrings(String num1, String num2) {
    StringBuilder result = new StringBuilder();
    int i = num1.length() - 1, j = num2.length() - 1;
    int carry = 0;

    while (i >= 0 || j >= 0 || carry > 0) {
        int digit1 = i >= 0 ? num1.charAt(i) - '0' : 0;
        int digit2 = j >= 0 ? num2.charAt(j) - '0' : 0;

        int sum = digit1 + digit2 + carry;
        result.append(sum % 10);
        carry = sum / 10;

        i--;
        j--;
    }

    return result.reverse().toString();
}

/**
* 解法二：使用 DFS + 剪枝优化
* 更直观的实现方式
*
* @param num 输入字符串
* @return 是否是累加数

```

```

*/
public static boolean isAdditiveNumberDFS(String num) {
    return dfs(num, 0, null, null, 0);
}

private static boolean dfs(String num, int start, Long prev1, Long prev2, int count) {
    // 终止条件：已处理完所有字符且至少有三个数字
    if (start == num.length()) {
        return count >= 3;
    }

    // 尝试所有可能的分割长度
    for (int len = 1; start + len <= num.length(); len++) {
        String currentStr = num.substring(start, start + len);

        // 检查当前数字是否合法（不能有前导 0）
        if (currentStr.length() > 1 && currentStr.charAt(0) == '0') {
            continue;
        }

        long current;
        try {
            current = Long.parseLong(currentStr);
        } catch (NumberFormatException e) {
            // 处理大数情况，使用字符串比较
            return false;
        }

        // 如果是前两个数字，直接递归
        if (count < 2) {
            if (count == 0) {
                if (dfs(num, start + len, current, null, count + 1)) {
                    return true;
                }
            } else {
                if (dfs(num, start + len, prev1, current, count + 1)) {
                    return true;
                }
            }
        } else {
            // 检查是否满足累加关系
            long expected = prev1 + prev2;
            if (current == expected) {

```

```

        if (dfs(num, start + len, prev2, current, count + 1)) {
            return true;
        }
    } else if (current > expected) {
        // 剪枝：如果当前数字已经大于期望值，提前终止
        break;
    }
}

return false;
}

/***
 * 解法三：使用迭代法验证
 * 适用于需要高效验证的情况
 *
 * @param num 输入字符串
 * @return 是否是累加数
 */
public static boolean isAdditiveNumberIterative(String num) {
    int n = num.length();

    // 尝试所有可能的前两个数字分割方案
    for (int i = 1; i <= n / 2; i++) {
        String num1 = num.substring(0, i);
        if (!isValid(num1)) continue;

        for (int j = 1; Math.max(i, j) <= n - i - j; j++) {
            String num2 = num.substring(i, i + j);
            if (!isValid(num2)) continue;

            if (isValidSequence(num, i + j, num1, num2)) {
                return true;
            }
        }
    }

    return false;
}

private static boolean isValidSequence(String num, int start, String num1, String num2) {
    int n = num.length();

```

```
String prev1 = num1, prev2 = num2;
int currentStart = start;

while (currentStart < n) {
    String expected = addStrings(prev1, prev2);
    int expectedLen = expected.length();

    if (currentStart + expectedLen > n) {
        return false;
    }

    String actual = num.substring(currentStart, currentStart + expectedLen);
    if (!actual.equals(expected)) {
        return false;
    }

    prev1 = prev2;
    prev2 = expected;
    currentStart += expectedLen;
}

return true;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    String num1 = "112358";
    boolean result1 = isAdditiveNumber(num1);
    System.out.println("输入: num = " + num1 + ")");
    System.out.println("输出: " + result1);

    // 测试用例 2
    String num2 = "199100199";
    boolean result2 = isAdditiveNumber(num2);
    System.out.println("\n输入: num = " + num2 + ")");
    System.out.println("输出: " + result2);

    // 测试用例 3
    String num3 = "1023";
    boolean result3 = isAdditiveNumber(num3);
    System.out.println("\n输入: num = " + num3 + ")");
    System.out.println("输出: " + result3);
}
```

```
// 测试 DFS 解法
System.out.println("\n==== DFS 解法测试 ====");
boolean result4 = isAdditiveNumberDFS(num1);
System.out.println("输入: num = " + num1 + ")");
System.out.println("输出: " + result4);
}
}

=====

```

文件: Code29\_AdditiveNumber.py

```
=====
class Solution:
```

```
"""

```

```
LeetCode 306. 累加数
```

题目描述:

累加数是一个字符串，组成它的数字可以形成累加序列。

一个有效的累加序列必须至少包含 3 个数。除了最开始的两个数以外，字符串中的其他数都等于它之前两个数相加的和。

示例:

输入: "112358"

输出: true

解释: 累加序列为: 1, 1, 2, 3, 5, 8。 $1 + 1 = 2$ ,  $1 + 2 = 3$ ,  $2 + 3 = 5$ ,  $3 + 5 = 8$

输入: "199100199"

输出: true

解释: 累加序列为: 1, 99, 100, 199。 $1 + 99 = 100$ ,  $99 + 100 = 199$

提示:

$1 \leq \text{num.length} \leq 35$

num 仅由数字 (0 - 9) 组成

链接: <https://leetcode.cn/problems/additive-number/>

算法思路:

1. 使用回溯算法尝试所有可能的前两个数字分割
2. 对于每个可能的前两个数字，验证剩余部分是否满足累加关系
3. 注意处理大数问题（字符串可能很长）
4. 注意处理前导 0 的情况（数字不能有前导 0，除非数字本身就是 0）

时间复杂度:  $O(n^3)$ , 需要尝试所有可能的前两个数字分割

空间复杂度:  $O(n)$ , 递归栈深度

"""

```
def isAdditiveNumber(self, num: str) -> bool:
    n = len(num)

    # 尝试所有可能的前两个数字分割
    for i in range(1, n):  # 第一个数字的结束位置
        # 第一个数字不能有前导 0 (除非是 0 本身)
        if num[0] == '0' and i > 1:
            continue

        for j in range(i + 1, n):  # 第二个数字的结束位置
            # 第二个数字不能有前导 0 (除非是 0 本身)
            if num[i] == '0' and j > i + 1:
                continue

            # 获取前两个数字
            num1 = num[:i]
            num2 = num[i:j]

            # 验证剩余部分
            if self.validate(num, j, num1, num2):
                return True

    return False

def validate(self, num: str, start: int, num1: str, num2: str) -> bool:
    """验证从 start 位置开始, 剩余部分是否满足累加关系"""
    if start == len(num):
        return True

    # 计算下一个期望的数字
    expected = self.add_strings(num1, num2)

    # 检查剩余字符串是否以期望数字开头
    if num.startswith(expected, start):
        # 递归验证剩余部分
        return self.validate(num, start + len(expected), num2, expected)

    return False
```

```
def add_strings(self, num1: str, num2: str) -> str:  
    """字符串加法，处理大数问题"""  
    i, j = len(num1) - 1, len(num2) - 1  
    carry = 0  
    result = []  
  
    while i >= 0 or j >= 0 or carry:  
        # 获取当前位的数字  
        digit1 = int(num1[i]) if i >= 0 else 0  
        digit2 = int(num2[j]) if j >= 0 else 0  
  
        # 计算当前位的和  
        total = digit1 + digit2 + carry  
        carry = total // 10  
        digit = total % 10  
  
        result.append(str(digit))  
  
        i -= 1  
        j -= 1  
  
    # 反转结果  
    return ''.join(result[::-1])  
  
def test_additive_number():  
    solution = Solution()  
  
    # 测试用例 1  
    num1 = "112358"  
    result1 = solution.isAdditiveNumber(num1)  
    print(f'输入: num = "{num1}"')  
    print("输出:", result1)  
  
    # 测试用例 2  
    num2 = "199100199"  
    result2 = solution.isAdditiveNumber(num2)  
    print(f'\n输入: num = "{num2}"')  
    print("输出:", result2)  
  
    # 测试用例 3  
    num3 = "1023"  
    result3 = solution.isAdditiveNumber(num3)  
    print(f'\n输入: num = "{num3}"')
```

```

print("输出:", result3)

# 测试用例 4
num4 = "101"
result4 = solution.isAdditiveNumber(num4)
print(f'\n 输入: num = "{num4}"')
print("输出:", result4)

if __name__ == "__main__":
    test_additive_number()

```

=====

文件: Code30\_BinaryWatch.cpp

=====

```

/*
LeetCode 401. 二进制手表

```

二进制手表顶部有 4 个 LED 代表小时 (0-11)，底部的 6 个 LED 代表分钟 (0-59)。  
 每个 LED 代表一个 0 或 1，最低位在右侧。  
 给你一个整数 turnedOn，表示当前亮着的 LED 的数量，返回二进制手表可以表示的所有可能时间。

算法思路：

使用回溯算法生成所有可能的 LED 点亮组合，然后验证这些组合是否能构成有效的时间。

时间复杂度:  $O(2^{10}) = O(1024)$ ，因为总共有 10 个 LED

空间复杂度:  $O(1)$ ，不考虑结果数组的空间

\*/

```

#include <vector>
#include <string>
#include <iostream>
#include <sstream>
#include <iomanip>
using namespace std;

class Solution {
private:
    // LED 代表的数值，前 4 个是小时，后 6 个是分钟
    vector<int> LED_VALUES = {8, 4, 2, 1, 32, 16, 8, 4, 2, 1};

    /**
     * 回溯函数

```

```

* @param turnedOn 剩余需要点亮的 LED 数量
* @param start 当前考虑的 LED 位置
* @param hour 当前小时数
* @param minute 当前分钟数
* @param result 结果列表
*/
void backtrack(int turnedOn, int start, int hour, int minute, vector<string>& result) {
    // 剪枝: 如果小时或分钟超出范围, 直接返回
    if (hour > 11 || minute > 59) {
        return;
    }

    // 终止条件: 所有 LED 都已考虑完
    if (turnedOn == 0) {
        // 格式化时间字符串
        string time = to_string(hour) + ":" + (minute < 10 ? "0" : "") + to_string(minute);
        result.push_back(time);
        return;
    }

    // 遍历剩余的 LED
    for (int i = start; i < LED_VALUES.size(); i++) {
        if (i < 4) {
            // 处理小时 LED
            backtrack(turnedOn - 1, i + 1, hour + LED_VALUES[i], minute, result);
        } else {
            // 处理分钟 LED
            backtrack(turnedOn - 1, i + 1, hour, minute + LED_VALUES[i], result);
        }
    }
}

public:
/***
 * 返回二进制手表可以表示的所有可能时间
 * @param turnedOn 当前亮着的 LED 的数量
 * @return 所有可能的时间列表
 */
vector<string> readBinaryWatch(int turnedOn) {
    vector<string> result;
    backtrack(turnedOn, 0, 0, 0, result);
    return result;
}

```

```
};

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

    // 测试用例 1
    cout << "turnedOn = 1:" << endl;
    vector<string> result1 = solution.readBinaryWatch(1);
    for (const string& time : result1) {
        cout << time << " ";
    }
    cout << endl;

    // 测试用例 2
    cout << "\nturnedOn = 9:" << endl;
    vector<string> result2 = solution.readBinaryWatch(9);
    for (const string& time : result2) {
        cout << time << " ";
    }
    cout << endl;

    return 0;
}
```

=====

文件: Code30\_BinaryWatch.java

=====

```
package class038;

import java.util.*;

/**
 * LeetCode 401. 二进制手表
 *
 * 二进制手表顶部有 4 个 LED 代表小时 (0-11)，底部的 6 个 LED 代表分钟 (0-59)。
 * 每个 LED 代表一个 0 或 1，最低位在右侧。
 * 给你一个整数 turnedOn，表示当前亮着的 LED 的数量，返回二进制手表可以表示的所有可能时间。
 *
 * 算法思路：
 * 使用回溯算法生成所有可能的 LED 点亮组合，然后验证这些组合是否能构成有效的时间。
 *
```

```

* 时间复杂度: O(2^10) = O(1024), 因为总共有 10 个 LED
* 空间复杂度: O(1), 不考虑结果数组的空间
*/
public class Code30_BinaryWatch {

    // LED 代表的数值, 前 4 个是小时, 后 6 个是分钟
    private static final int[] LED_VALUES = {8, 4, 2, 1, 32, 16, 8, 4, 2, 1};

    /**
     * 返回二进制手表可以表示的所有可能时间
     * @param turnedOn 当前亮着的 LED 的数量
     * @return 所有可能的时间列表
     */
    public List<String> readBinaryWatch(int turnedOn) {
        List<String> result = new ArrayList<>();
        backtrack(turnedOn, 0, 0, 0, result);
        return result;
    }

    /**
     * 回溯函数
     * @param turnedOn 剩余需要点亮的 LED 数量
     * @param start 当前考虑的 LED 位置
     * @param hour 当前小时数
     * @param minute 当前分钟数
     * @param result 结果列表
     */
    private void backtrack(int turnedOn, int start, int hour, int minute, List<String> result) {
        // 剪枝: 如果小时或分钟超出范围, 直接返回
        if (hour > 11 || minute > 59) {
            return;
        }

        // 终止条件: 所有 LED 都已考虑完
        if (turnedOn == 0) {
            // 格式化时间字符串
            String time = hour + ":" + (minute < 10 ? "0" + minute : minute);
            result.add(time);
            return;
        }

        // 遍历剩余的 LED
        for (int i = start; i < LED_VALUES.length; i++) {

```

```

    if (i < 4) {
        // 处理小时 LED
        backtrack(turnedOn - 1, i + 1, hour + LED_VALUES[i], minute, result);
    } else {
        // 处理分钟 LED
        backtrack(turnedOn - 1, i + 1, hour, minute + LED_VALUES[i], result);
    }
}

// 测试方法
public static void main(String[] args) {
    Code30_BinaryWatch solution = new Code30_BinaryWatch();

    // 测试用例 1
    System.out.println("turnedOn = 1:");
    System.out.println(solution.readBinaryWatch(1));

    // 测试用例 2
    System.out.println("\nturnedOn = 9:");
    System.out.println(solution.readBinaryWatch(9));
}
}

```

=====

文件: Code30\_BinaryWatch.py

=====

"""
LeetCode 401. 二进制手表

二进制手表顶部有 4 个 LED 代表小时 (0-11)，底部的 6 个 LED 代表分钟 (0-59)。  
 每个 LED 代表一个 0 或 1，最低位在右侧。  
 给你一个整数 turnedOn，表示当前亮着的 LED 的数量，返回二进制手表可以表示的所有可能时间。

算法思路：

使用回溯算法生成所有可能的 LED 点亮组合，然后验证这些组合是否能构成有效的时间。

时间复杂度:  $O(2^{10}) = O(1024)$ ，因为总共有 10 个 LED

空间复杂度:  $O(1)$ ，不考虑结果数组的空间

"""

class Solution:

```

def __init__(self):
    # LED 代表的数值，前 4 个是小时，后 6 个是分钟
    self.LED_VALUES = [8, 4, 2, 1, 32, 16, 8, 4, 2, 1]

def readBinaryWatch(self, turnedOn):
    """
    返回二进制手表可以表示的所有可能时间
    :param turnedOn: int 当前亮着的 LED 的数量
    :return: List[str] 所有可能的时间列表
    """
    result = []
    self.backtrack(turnedOn, 0, 0, 0, result)
    return result

def backtrack(self, turnedOn, start, hour, minute, result):
    """
    回溯函数
    :param turnedOn: int 剩余需要点亮的 LED 数量
    :param start: int 当前考虑的 LED 位置
    :param hour: int 当前小时数
    :param minute: int 当前分钟数
    :param result: List[str] 结果列表
    """
    # 剪枝：如果小时或分钟超出范围，直接返回
    if hour > 11 or minute > 59:
        return

    # 终止条件：所有 LED 都已考虑完
    if turnedOn == 0:
        # 格式化时间字符串
        time = f'{hour}:{minute:02d}'
        result.append(time)
        return

    # 遍历剩余的 LED
    for i in range(start, len(self.LED_VALUES)):
        if i < 4:
            # 处理小时 LED
            self.backtrack(turnedOn - 1, i + 1, hour + self.LED_VALUES[i], minute, result)
        else:
            # 处理分钟 LED
            self.backtrack(turnedOn - 1, i + 1, hour, minute + self.LED_VALUES[i], result)

```

```

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

# 测试用例 1
print("turnedOn = 1:")
print(solution.readBinaryWatch(1))

# 测试用例 2
print("\nturnedOn = 9:")
print(solution.readBinaryWatch(9))

```

=====

文件: Code31\_LetterCasePermutation.cpp

=====

/\*
LeetCode 784. 字母大小写全排列

给定一个字符串 S，通过将字符串 S 中的每个字母转变大小写，我们可以获得一个新的字符串。  
返回所有可能得到的字符串集合。以任意顺序返回输出。

算法思路:

使用回溯算法遍历字符串中的每个字符，对于字母字符，分别尝试大写和小写两种情况。

时间复杂度:  $O(2^n * n)$ ，其中 n 是字符串中字母的个数

空间复杂度:  $O(2^n * n)$ ，用于存储所有可能的字符串

\*/

```

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

class Solution {
public:
    /**
     * 返回所有可能得到的字符串集合
     * @param s 输入字符串
     * @return 所有可能的字符串集合
     */
    vector<string> letterCasePermutation(string s) {

```

```
vector<string> result;
backtrack(s, 0, result);
return result;
}

private:
/***
 * 回溯函数
 * @param s 字符串
 * @param index 当前处理的字符位置
 * @param result 结果列表
 */
void backtrack(string s, int index, vector<string>& result) {
    // 终止条件：处理完所有字符
    if (index == s.length()) {
        result.push_back(s);
        return;
    }

    char ch = s[index];

    // 如果是字母，则分别尝试大写和小写
    if (isalpha(ch)) {
        // 尝试小写
        s[index] = tolower(ch);
        backtrack(s, index + 1, result);

        // 尝试大写
        s[index] = toupper(ch);
        backtrack(s, index + 1, result);
    } else {
        // 如果不是字母，直接处理下一个字符
        backtrack(s, index + 1, result);
    }
}

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

    // 测试用例 1
    cout << "Input: \"a1b2\"" << endl;
```

```

vector<string> result1 = solution.letterCasePermutation("a1b2");
for (const string& str : result1) {
    cout << str << " ";
}
cout << endl;

// 测试用例 2
cout << "\nInput: \"3z4\" " << endl;
vector<string> result2 = solution.letterCasePermutation("3z4");
for (const string& str : result2) {
    cout << str << " ";
}
cout << endl;

// 测试用例 3
cout << "\nInput: \"12345\" " << endl;
vector<string> result3 = solution.letterCasePermutation("12345");
for (const string& str : result3) {
    cout << str << " ";
}
cout << endl;

return 0;
}

```

=====

文件: Code31\_LetterCasePermutation.java

=====

```

package class038;

import java.util.*;

/**
 * LeetCode 784. 字母大小写全排列
 *
 * 给定一个字符串 S，通过将字符串 S 中的每个字母转变大小写，我们可以获得一个新的字符串。
 * 返回所有可能得到的字符串集合。以任意顺序返回输出。
 *
 * 算法思路：
 * 使用回溯算法遍历字符串中的每个字符，对于字母字符，分别尝试大写和小写两种情况。
 *
 * 时间复杂度：O(2^n * n)，其中 n 是字符串中字母的个数

```

```

* 空间复杂度: O(2^n * n), 用于存储所有可能的字符串
*/
public class Code31_LetterCasePermutation {

    /**
     * 返回所有可能得到的字符串集合
     * @param s 输入字符串
     * @return 所有可能的字符串集合
     */
    public List<String> letterCasePermutation(String s) {
        List<String> result = new ArrayList<>();
        char[] chars = s.toCharArray();
        backtrack(chars, 0, result);
        return result;
    }

    /**
     * 回溯函数
     * @param chars 字符数组
     * @param index 当前处理的字符位置
     * @param result 结果列表
     */
    private void backtrack(char[] chars, int index, List<String> result) {
        // 终止条件: 处理完所有字符
        if (index == chars.length) {
            result.add(new String(chars));
            return;
        }

        char ch = chars[index];

        // 如果是字母, 则分别尝试大写和小写
        if (Character.isLetter(ch)) {
            // 尝试小写
            chars[index] = Character.toLowerCase(ch);
            backtrack(chars, index + 1, result);

            // 尝试大写
            chars[index] = Character.toUpperCase(ch);
            backtrack(chars, index + 1, result);
        } else {
            // 如果不是字母, 直接处理下一个字符
            backtrack(chars, index + 1, result);
        }
    }
}

```

```

    }
}

// 测试方法
public static void main(String[] args) {
    Code31_LetterCasePermutation solution = new Code31_LetterCasePermutation();

    // 测试用例 1
    System.out.println("Input: \"a1b2\"");
    System.out.println(solution.letterCasePermutation("a1b2"));

    // 测试用例 2
    System.out.println("\nInput: \"3z4\"");
    System.out.println(solution.letterCasePermutation("3z4"));

    // 测试用例 3
    System.out.println("\nInput: \"12345\"");
    System.out.println(solution.letterCasePermutation("12345"));
}
}

```

=====

文件: Code31\_LetterCasePermutation.py

=====

"""
LeetCode 784. 字母大小写全排列

给定一个字符串 S，通过将字符串 S 中的每个字母转变大小写，我们可以获得一个新的字符串。  
返回所有可能得到的字符串集合。以任意顺序返回输出。

算法思路:

使用回溯算法遍历字符串中的每个字符，对于字母字符，分别尝试大写和小写两种情况。

时间复杂度:  $O(2^n * n)$ ，其中 n 是字符串中字母的个数

空间复杂度:  $O(2^n * n)$ ，用于存储所有可能的字符串

"""

```

class Solution:
    def letterCasePermutation(self, s):
        """
        返回所有可能得到的字符串集合
        :param s: str 输入字符串

```

```
:return: List[str] 所有可能的字符串集合
"""

result = []
self.backtrack(list(s), 0, result)
return result

def backtrack(self, chars, index, result):
    """
    回溯函数
    :param chars: List[str] 字符列表
    :param index: int 当前处理的字符位置
    :param result: List[str] 结果列表
    """

    # 终止条件：处理完所有字符
    if index == len(chars):
        result.append("".join(chars))
        return

    ch = chars[index]

    # 如果是字母，则分别尝试大写和小写
    if ch.isalpha():
        # 尝试小写
        chars[index] = ch.lower()
        self.backtrack(chars, index + 1, result)

        # 尝试大写
        chars[index] = ch.upper()
        self.backtrack(chars, index + 1, result)

    else:
        # 如果不是字母，直接处理下一个字符
        self.backtrack(chars, index + 1, result)

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

    # 测试用例 1
    print('Input: "a1b2"')
    print(solution.letterCasePermutation("a1b2"))

    # 测试用例 2
    print('\nInput: "3z4"')
```

```

print(solution.letterCasePermutation("3z4"))

# 测试用例 3
print('\nInput: "12345"')
print(solution.letterCasePermutation("12345"))

```

=====

文件: Code32\_NumSquarefulPerms.cpp

=====

```

/*
LeetCode 996. 正方形数组的数目

```

给定一个非负整数数组 A，如果该数组任意两个相邻元素的和都可以表示为某个完全平方数，那么这个数组就称为正方形数组。返回 A 的所有可能的排列中，正方形数组的数目。

算法思路：

使用回溯算法生成所有可能的排列，并在生成过程中检查相邻元素之和是否为完全平方数。  
需要注意去重，因为数组中可能有重复元素。

时间复杂度:  $O(n! * n)$

空间复杂度:  $O(n)$

\*/

```

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

class Solution {
private:
    int count = 0;

    /**
     * 回溯函数
     * @param nums 输入数组
     * @param used 标记数组元素是否已使用
     * @param prevIndex 前一个元素的索引
     * @param index 当前处理的位置
     */
    void backtrack(vector<int>& nums, vector<bool>& used, int prevIndex, int index) {
        // 终止条件：处理完所有元素

```

```

    if (index == nums.size()) {
        count++;
        return;
    }

    for (int i = 0; i < nums.size(); i++) {
        // 去重：如果当前元素与前一个元素相同，且前一个元素未使用，则跳过
        if (used[i] || (i > 0 && nums[i] == nums[i-1] && !used[i-1])) {
            continue;
        }

        // 检查相邻元素之和是否为完全平方数
        if (prevIndex != -1 && !isPerfectSquare(nums[prevIndex] + nums[i])) {
            continue;
        }

        used[i] = true;
        backtrack(nums, used, i, index + 1);
        used[i] = false;
    }
}

/***
 * 判断一个数是否为完全平方数
 * @param num 待判断的数
 * @return 是否为完全平方数
 */
bool isPerfectSquare(int num) {
    int sqrtNum = (int)sqrt(num);
    return sqrtNum * sqrtNum == num;
}

public:
    /**
     * 返回正方形数组的数目
     * @param nums 输入数组
     * @return 正方形数组的数目
     */
    int numSquarefulPerms(vector<int>& nums) {
        count = 0;
        sort(nums.begin(), nums.end()); // 排序便于去重
        vector<bool> used(nums.size(), false);
        backtrack(nums, used, -1, 0);
    }
}

```

```

        return count;
    }
};

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

    // 测试用例 1
    vector<int> nums1 = {1, 17, 8};
    cout << "Input: [1, 17, 8]" << endl;
    cout << "Output: " << solution.numSquarefulPerms(nums1) << endl;

    // 测试用例 2
    vector<int> nums2 = {2, 2, 2};
    cout << "\nInput: [2, 2, 2]" << endl;
    cout << "Output: " << solution.numSquarefulPerms(nums2) << endl;

    return 0;
}

```

=====

文件: Code32\_NumSquarefulPerms.java

=====

```

package class038;

import java.util.*;

/**
 * LeetCode 996. 正方形数组的数目
 *
 * 给定一个非负整数数组 A，如果该数组任意两个相邻元素的和都可以表示为某个完全平方数，那么这个数组就称为正方形数组。返回 A 的所有可能的排列中，正方形数组的数目。
 *
 * 算法思路：
 * 使用回溯算法生成所有可能的排列，并在生成过程中检查相邻元素之和是否为完全平方数。
 * 需要注意去重，因为数组中可能有重复元素。
 *
 * 时间复杂度：O(n! * n)
 * 空间复杂度：O(n)
 */
public class Code32_NumSquarefulPerms {

```

```

private int count = 0;

/**
 * 返回正方形数组的数目
 * @param nums 输入数组
 * @return 正方形数组的数目
 */
public int numSquarefulPerms(int[] nums) {
    count = 0;
    Arrays.sort(nums); // 排序便于去重
    boolean[] used = new boolean[nums.length];
    backtrack(nums, used, -1, 0);
    return count;
}

/**
 * 回溯函数
 * @param nums 输入数组
 * @param used 标记数组元素是否已使用
 * @param prevIndex 前一个元素的索引
 * @param index 当前处理的位置
 */
private void backtrack(int[] nums, boolean[] used, int prevIndex, int index) {
    // 终止条件：处理完所有元素
    if (index == nums.length) {
        count++;
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 去重：如果当前元素与前一个元素相同，且前一个元素未使用，则跳过
        if (used[i] || (i > 0 && nums[i] == nums[i-1] && !used[i-1])) {
            continue;
        }

        // 检查相邻元素之和是否为完全平方数
        if (prevIndex != -1 && !isPerfectSquare(nums[prevIndex] + nums[i])) {
            continue;
        }

        used[i] = true;
        backtrack(nums, used, i, index + 1);
    }
}

```

```

        used[i] = false;
    }
}

/***
 * 判断一个数是否为完全平方数
 * @param num 待判断的数
 * @return 是否为完全平方数
 */
private boolean isPerfectSquare(int num) {
    int sqrt = (int) Math.sqrt(num);
    return sqrt * sqrt == num;
}

// 测试方法
public static void main(String[] args) {
    Code32_NumSquarefulPerms solution = new Code32_NumSquarefulPerms();

    // 测试用例 1
    int[] nums1 = {1, 17, 8};
    System.out.println("Input: [1, 17, 8]");
    System.out.println("Output: " + solution.numSquarefulPerms(nums1));

    // 测试用例 2
    int[] nums2 = {2, 2, 2};
    System.out.println("\nInput: [2, 2, 2]");
    System.out.println("Output: " + solution.numSquarefulPerms(nums2));
}
}

```

文件: Code32\_NumSquarefulPerms.py

```

=====
"""

LeetCode 996. 正方形数组的数目

给定一个非负整数数组 A，如果该数组任意两个相邻元素的和都可以表示为某个完全平方数，那么这个数组就称为正方形数组。返回 A 的所有可能的排列中，正方形数组的数目。

```

算法思路:

使用回溯算法生成所有可能的排列，并在生成过程中检查相邻元素之和是否为完全平方数。需要注意去重，因为数组中可能有重复元素。

时间复杂度:  $O(n! * n)$

空间复杂度:  $O(n)$

"""

class Solution:

def \_\_init\_\_(self):

self.count = 0

def numSquarefulPerms(self, nums):

"""

返回正方形数组的数目

:param nums: List[int] 输入数组

:return: int 正方形数组的数目

"""

self.count = 0

nums.sort() # 排序便于去重

used = [False] \* len(nums)

self.backtrack(nums, used, -1, 0)

return self.count

def backtrack(self, nums, used, prev\_index, index):

"""

回溯函数

:param nums: List[int] 输入数组

:param used: List[bool] 标记数组元素是否已使用

:param prev\_index: int 前一个元素的索引

:param index: int 当前处理的位置

"""

# 终止条件: 处理完所有元素

if index == len(nums):

self.count += 1

return

for i in range(len(nums)):

# 去重: 如果当前元素与前一个元素相同, 且前一个元素未使用, 则跳过

if used[i] or (i > 0 and nums[i] == nums[i-1] and not used[i-1]):

continue

# 检查相邻元素之和是否为完全平方数

if prev\_index != -1 and not self.is\_perfect\_square(nums[prev\_index] + nums[i]):

continue

```

        used[i] = True
        self.backtrack(nums, used, i, index + 1)
        used[i] = False

def is_perfect_square(self, num):
    """
    判断一个数是否为完全平方数
    :param num: int 待判断的数
    :return: bool 是否为完全平方数
    """
    sqrt_num = int(num ** 0.5)
    return sqrt_num * sqrt_num == num

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

# 测试用例 1
nums1 = [1, 17, 8]
print("Input: [1, 17, 8]")
print("Output:", solution.numSquarefulPerms(nums1))

# 测试用例 2
nums2 = [2, 2, 2]
print("\nInput: [2, 2, 2]")
print("Output:", solution.numSquarefulPerms(nums2))

```

=====

文件: Code33\_Sticks.cpp

=====

```

/*
POJ 1011 Sticks

```

给定  $n$  根火柴棍，每根火柴棍都有一定的长度。要求将这些火柴棍拼成若干根长度相等的火柴棍，且每根新火柴棍的长度要尽可能大。求这个最大长度。

算法思路：

使用回溯算法，从最大可能的长度开始尝试，逐步减小，直到找到一个可行解。

对于每个尝试的长度，使用回溯算法检查是否能将所有火柴棍拼成该长度的若干根新火柴棍。

时间复杂度： $O(2^n * n)$

空间复杂度： $O(n)$

```

*/
#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>
using namespace std;

class Solution {
public:
    /**
     * 求最大可能的火柴棍长度
     * @param sticks 火柴棍长度数组
     * @return 最大可能的火柴棍长度
     */
    int maxLenOfSticks(vector<int>& sticks) {
        // 从大到小排序，便于剪枝
        sort(sticks.begin(), sticks.end(), greater<int>());

        int sum = accumulate(sticks.begin(), sticks.end(), 0);

        // 从最大可能长度开始尝试
        for (int len = sum / sticks.size(); len >= 1; len--) {
            if (sum % len == 0) { // 只有当总长度能被 len 整除时才可能
                vector<int> buckets(sum / len, 0);
                if (backtrack(sticks, 0, buckets, len)) {
                    return len;
                }
            }
        }

        return 1; // 最坏情况，每根火柴棍单独作为一根
    }

private:
    /**
     * 回溯函数，尝试将火柴棍分配到各个桶中
     * @param sticks 火柴棍长度数组
     * @param index 当前处理的火柴棍索引
     * @param buckets 桶数组，记录每个桶当前的长度
     * @param target 目标长度
     * @return 是否能成功分配
     */

```

```

bool backtrack(vector<int>& sticks, int index, vector<int>& buckets, int target) {
    // 终止条件：所有火柴棍都已处理完
    if (index == sticks.size()) {
        return true;
    }

    int stick = sticks[index];

    // 尝试将当前火柴棍放入每个桶中
    for (int i = 0; i < buckets.size(); i++) {
        // 剪枝：如果放入当前桶后超过目标长度，则跳过
        if (buckets[i] + stick > target) {
            continue;
        }

        buckets[i] += stick;
        if (backtrack(sticks, index + 1, buckets, target)) {
            return true;
        }
        buckets[i] -= stick;

        // 剪枝：如果当前桶为空，说明当前火柴棍无法放入任何桶中
        if (buckets[i] == 0) {
            break;
        }
    }

    return false;
}

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

    // 测试用例 1
    vector<int> sticks1 = {5, 2, 1, 5, 2, 1, 5, 2, 1};
    cout << "Input: [5, 2, 1, 5, 2, 1, 5, 2, 1]" << endl;
    cout << "Output: " << solution.maxLenOfSticks(sticks1) << endl;

    // 测试用例 2
    vector<int> sticks2 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    cout << "\nInput: [1, 2, 3, 4, 5, 6, 7, 8, 9]" << endl;
}

```

```
cout << "Output: " << solution.maxLenOfSticks(sticks2) << endl;

return 0;
}
```

=====

文件: Code33\_Sticks.java

=====

```
package class038;

import java.util.*;

/**
 * POJ 1011 Sticks
 *
 * 给定 n 根火柴棍，每根火柴棍都有一定的长度。要求将这些火柴棍拼成若干根长度相等的火柴棍，且每根新火柴棍的长度要尽可能大。求这个最大长度。
 *
 * 算法思路：
 * 使用回溯算法，从最大可能的长度开始尝试，逐步减小，直到找到一个可行解。
 * 对于每个尝试的长度，使用回溯算法检查是否能将所有火柴棍拼成该长度的若干根新火柴棍。
 *
 * 时间复杂度：O(2^n * n)
 * 空间复杂度：O(n)
 */
public class Code33_Sticks {

    /**
     * 求最大可能的火柴棍长度
     * @param sticks 火柴棍长度数组
     * @return 最大可能的火柴棍长度
     */
    public int maxLenOfSticks(int[] sticks) {
        // 从大到小排序，便于剪枝
        Arrays.sort(sticks);
        for (int i = 0, j = sticks.length - 1; i < j; i++, j--) {
            int temp = sticks[i];
            sticks[i] = sticks[j];
            sticks[j] = temp;
        }

        int sum = 0;
```

```

for (int stick : sticks) {
    sum += stick;
}

// 从最大可能长度开始尝试
for (int len = sum / sticks.length; len >= 1; len--) {
    if (sum % len == 0) { // 只有当总长度能被 len 整除时才可能
        int[] buckets = new int[sum / len];
        if (backtrack(sticks, 0, buckets, len)) {
            return len;
        }
    }
}

return 1; // 最坏情况，每根火柴棍单独作为一根
}

/***
 * 回溯函数，尝试将火柴棍分配到各个桶中
 * @param sticks 火柴棍长度数组
 * @param index 当前处理的火柴棍索引
 * @param buckets 桶数组，记录每个桶当前的长度
 * @param target 目标长度
 * @return 是否能成功分配
 */
private boolean backtrack(int[] sticks, int index, int[] buckets, int target) {
    // 终止条件：所有火柴棍都已处理完
    if (index == sticks.length) {
        return true;
    }

    int stick = sticks[index];

    // 尝试将当前火柴棍放入每个桶中
    for (int i = 0; i < buckets.length; i++) {
        // 剪枝：如果放入当前桶后超过目标长度，则跳过
        if (buckets[i] + stick > target) {
            continue;
        }

        buckets[i] += stick;
        if (backtrack(sticks, index + 1, buckets, target)) {
            return true;
        }
    }
}

```

```

        }

        buckets[i] -= stick;

        // 剪枝：如果当前桶为空，说明当前火柴棍无法放入任何桶中
        if (buckets[i] == 0) {
            break;
        }
    }

    return false;
}

// 测试方法
public static void main(String[] args) {
    Code33_Sticks solution = new Code33_Sticks();

    // 测试用例 1
    int[] sticks1 = {5, 2, 1, 5, 2, 1, 5, 2, 1};
    System.out.println("Input: [5, 2, 1, 5, 2, 1, 5, 2, 1]");
    System.out.println("Output: " + solution.maxLenOfSticks(sticks1));

    // 测试用例 2
    int[] sticks2 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("\nInput: [1, 2, 3, 4, 5, 6, 7, 8, 9]");
    System.out.println("Output: " + solution.maxLenOfSticks(sticks2));
}
}

```

文件: Code33\_Sticks.py

POJ 1011 Sticks

给定 n 根火柴棍，每根火柴棍都有一定的长度。要求将这些火柴棍拼成若干根长度相等的火柴棍，且每根新火柴棍的长度要尽可能大。求这个最大长度。

算法思路：

使用回溯算法，从最大可能的长度开始尝试，逐步减小，直到找到一个可行解。

对于每个尝试的长度，使用回溯算法检查是否能将所有火柴棍拼成该长度的若干根新火柴棍。

时间复杂度： $O(2^n * n)$

空间复杂度: O(n)

"""

class Solution:

def maxLenOfSticks(self, sticks):

"""

求最大可能的火柴棍长度

:param sticks: List[int] 火柴棍长度数组

:return: int 最大可能的火柴棍长度

"""

# 从大到小排序, 便于剪枝

sticks.sort(reverse=True)

total\_sum = sum(sticks)

# 从最大可能长度开始尝试

for length in range(total\_sum // len(sticks), 0, -1):

if total\_sum % length == 0: # 只有当总长度能被 length 整除时才可能

buckets = [0] \* (total\_sum // length)

if self.backtrack(sticks, 0, buckets, length):

return length

return 1 # 最坏情况, 每根火柴棍单独作为一根

def backtrack(self, sticks, index, buckets, target):

"""

回溯函数, 尝试将火柴棍分配到各个桶中

:param sticks: List[int] 火柴棍长度数组

:param index: int 当前处理的火柴棍索引

:param buckets: List[int] 桶数组, 记录每个桶当前的长度

:param target: int 目标长度

:return: bool 是否能成功分配

"""

# 终止条件: 所有火柴棍都已处理完

if index == len(sticks):

return True

stick = sticks[index]

# 尝试将当前火柴棍放入每个桶中

for i in range(len(buckets)):

# 剪枝: 如果放入当前桶后超过目标长度, 则跳过

if buckets[i] + stick > target:

```

        continue

    buckets[i] += stick
    if self.backtrack(sticks, index + 1, buckets, target):
        return True
    buckets[i] -= stick

    # 剪枝: 如果当前桶为空, 说明当前火柴棍无法放入任何桶中
    if buckets[i] == 0:
        break

    return False

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

# 测试用例 1
sticks1 = [5, 2, 1, 5, 2, 1, 5, 2, 1]
print("Input: [5, 2, 1, 5, 2, 1, 5, 2, 1]")
print("Output:", solution.maxLenOfSticks(sticks1))

# 测试用例 2
sticks2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print("\nInput: [1, 2, 3, 4, 5, 6, 7, 8, 9]")
print("Output:", solution.maxLenOfSticks(sticks2))

```

=====

文件: Code34\_GenerateParenthesesII.cpp

=====

```

/*
LeetCode 22. 括号生成 (增强版)

```

数字  $n$  代表生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且有效的括号组合。  
增强版: 除了生成  $n$  对括号的所有组合外, 还要求计算每个组合中连续括号的最大长度。

算法思路:

使用回溯算法生成所有有效的括号组合, 在生成过程中同时计算连续括号的最大长度。

时间复杂度:  $O(4^n / \sqrt{n})$ , 第  $n$  个卡塔兰数

空间复杂度:  $O(4^n / \sqrt{n})$

\*/

```

#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
#include <utility>
using namespace std;

class Solution {
public:
    /**
     * 生成所有可能的并且有效的括号组合，并计算每个组合中连续括号的最大长度
     * @param n 括号对数
     * @return 包含括号组合和对应最大连续长度的列表
     */
    vector<pair<string, int>> generateParenthesisWithMaxConsecutive(int n) {
        vector<pair<string, int>> result;
        backtrack(n, n, "", 0, 0, result);
        return result;
    }

private:
    /**
     * 回溯函数
     * @param left 剩余左括号数量
     * @param right 剩余右括号数量
     * @param current 当前生成的括号字符串
     * @param consecutive 当前连续括号长度
     * @param maxConsecutive 当前最大连续括号长度
     * @param result 结果列表
     */
    void backtrack(int left, int right, string current, int consecutive, int maxConsecutive,
vector<pair<string, int>>& result) {
        // 终止条件：所有括号都已使用完
        if (left == 0 && right == 0) {
            result.push_back(make_pair(current, maxConsecutive));
            return;
        }

        // 剪枝：右括号不能比左括号多
        if (left > right) {
            return;
        }

```

```

// 添加左括号
if (left > 0) {
    backtrack(left - 1, right, current + "(", consecutive + 1, max(maxConsecutive,
consecutive + 1), result);
}

// 添加右括号
if (right > 0) {
    int newConsecutive = consecutive > 0 ? consecutive - 1 : 0;
    backtrack(left, right - 1, current + ")", newConsecutive, maxConsecutive, result);
}
}

};

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

    // 测试用例 1
    cout << "n = 3:" << endl;
    vector<pair<string, int>> result1 = solution.generateParenthesisWithMaxConsecutive(3);
    for (const auto& entry : result1) {
        cout << entry.first << " -> Max consecutive: " << entry.second << endl;
    }

    // 测试用例 2
    cout << "\nn = 2:" << endl;
    vector<pair<string, int>> result2 = solution.generateParenthesisWithMaxConsecutive(2);
    for (const auto& entry : result2) {
        cout << entry.first << " -> Max consecutive: " << entry.second << endl;
    }

    return 0;
}

```

=====

文件: Code34\_GenerateParenthesesII.java

=====

```

package class038;

import java.util.*;

```

```

/**
 * LeetCode 22. 括号生成（增强版）
 *
 * 数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。
 * 增强版：除了生成 n 对括号的所有组合外，还要求计算每个组合中连续括号的最大长度。
 *
 * 算法思路：
 * 使用回溯算法生成所有有效的括号组合，在生成过程中同时计算连续括号的最大长度。
 *
 * 时间复杂度：O(4^n / sqrt(n))，第 n 个卡塔兰数
 * 空间复杂度：O(4^n / sqrt(n))
 */

public class Code34_GenerateParenthesesII {

    /**
     * 生成所有可能的并且有效的括号组合，并计算每个组合中连续括号的最大长度
     * @param n 括号对数
     * @return 包含括号组合和对应最大连续长度的列表
     */
    public List<Map.Entry<String, Integer>> generateParenthesisWithMaxConsecutive(int n) {
        List<Map.Entry<String, Integer>> result = new ArrayList<>();
        backtrack(n, n, "", 0, 0, result);
        return result;
    }

    /**
     * 回溯函数
     * @param left 剩余左括号数量
     * @param right 剩余右括号数量
     * @param current 当前生成的括号字符串
     * @param consecutive 当前连续括号长度
     * @param maxConsecutive 当前最大连续括号长度
     * @param result 结果列表
     */
    private void backtrack(int left, int right, String current, int consecutive, int maxConsecutive, List<Map.Entry<String, Integer>> result) {
        // 终止条件：所有括号都已使用完
        if (left == 0 && right == 0) {
            result.add(new AbstractMap.SimpleEntry<>(current, maxConsecutive));
            return;
        }
}

```

```
// 剪枝：右括号不能比左括号多
if (left > right) {
    return;
}

// 添加左括号
if (left > 0) {
    backtrack(left - 1, right, current + "(", consecutive + 1, Math.max(maxConsecutive,
consecutive + 1), result);
}

// 添加右括号
if (right > 0) {
    backtrack(left, right - 1, current + ")", consecutive > 0 ? consecutive - 1 : 0,
maxConsecutive, result);
}

// 测试方法
public static void main(String[] args) {
    Code34_GenerateParenthesesII solution = new Code34_GenerateParenthesesII();

    // 测试用例 1
    System.out.println("n = 3:");
    List<Map.Entry<String, Integer>> result1 =
solution.generateParenthesisWithMaxConsecutive(3);
    for (Map.Entry<String, Integer> entry : result1) {
        System.out.println(entry.getKey() + " -> Max consecutive: " + entry.getValue());
    }

    // 测试用例 2
    System.out.println("\nn = 2:");
    List<Map.Entry<String, Integer>> result2 =
solution.generateParenthesisWithMaxConsecutive(2);
    for (Map.Entry<String, Integer> entry : result2) {
        System.out.println(entry.getKey() + " -> Max consecutive: " + entry.getValue());
    }
}
```

```
=====
```

```
"""
```

## LeetCode 22. 括号生成（增强版）

数字  $n$  代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。

增强版：除了生成  $n$  对括号的所有组合外，还要求计算每个组合中连续括号的最大长度。

算法思路：

使用回溯算法生成所有有效的括号组合，在生成过程中同时计算连续括号的最大长度。

时间复杂度： $O(4^n / \sqrt{n})$ ，第  $n$  个卡塔兰数

空间复杂度： $O(4^n / \sqrt{n})$

```
"""
```

```
class Solution:
```

```
    def generateParenthesisWithMaxConsecutive(self, n):
```

```
        """
```

生成所有可能的并且有效的括号组合，并计算每个组合中连续括号的最大长度

:param n: int 括号对数

:return: List[Tuple[str, int]] 包含括号组合和对应最大连续长度的列表

```
        """
```

```
        result = []
```

```
        self.backtrack(n, n, "", 0, 0, result)
```

```
        return result
```

```
    def backtrack(self, left, right, current, consecutive, max_consecutive, result):
```

```
        """
```

回溯函数

:param left: int 剩余左括号数量

:param right: int 剩余右括号数量

:param current: str 当前生成的括号字符串

:param consecutive: int 当前连续括号长度

:param max\_consecutive: int 当前最大连续括号长度

:param result: List[Tuple[str, int]] 结果列表

```
        """
```

# 终止条件：所有括号都已使用完

```
        if left == 0 and right == 0:
```

```
            result.append((current, max_consecutive))
```

```
            return
```

# 剪枝：右括号不能比左括号多

```
        if left > right:
```

```
            return
```

```

# 添加左括号
if left > 0:
    self.backtrack(left - 1, right, current + "(", consecutive + 1, max(max_consecutive,
consecutive + 1), result)

# 添加右括号
if right > 0:
    new_consecutive = consecutive - 1 if consecutive > 0 else 0
    self.backtrack(left, right - 1, current + ")", new_consecutive, max_consecutive,
result)

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

    # 测试用例 1
    print("n = 3:")
    result1 = solution.generateParenthesisWithMaxConsecutive(3)
    for combo, max_consecutive in result1:
        print(f"{combo} -> Max consecutive: {max_consecutive}")

    # 测试用例 2
    print("\nn = 2:")
    result2 = solution.generateParenthesisWithMaxConsecutive(2)
    for combo, max_consecutive in result2:
        print(f"{combo} -> Max consecutive: {max_consecutive}")

```

=====

文件: LeetCode216\_CombinationSumIII.java

=====

```

/**
 * LeetCode 216. 组合总和 III
 *
 * 题目描述:
 * 找出所有相加之和为 n 的 k 个数的组合。
 * 只能使用数字 1 到 9，每个数字最多使用一次。
 * 返回所有可能的有效组合的列表。该列表不能包含相同的组合两次。
 *
 * 示例:
 * 输入: k = 3, n = 7
 * 输出: [[1, 2, 4]]

```

```
* 解释: 1 + 2 + 4 = 7, 没有其他符合的组合了。  
*  
* 输入: k = 3, n = 9  
* 输出: [[1, 2, 6], [1, 3, 5], [2, 3, 4]]  
*  
* 输入: k = 4, n = 1  
* 输出: []  
* 解释: 不存在有效的组合。在[1, 9]范围内使用 4 个不同的数字, 我们可以得到的最小和是 1+2+3+4 = 10,  
因为 10 > 1, 没有有效的组合。  
*  
* 提示:  
* 2 <= k <= 9  
* 1 <= n <= 60  
*  
* 链接: https://leetcode.cn/problems/combination-sum-iii/  
*  
* 算法思路:  
* 1. 使用回溯算法生成所有可能的组合  
* 2. 从数字 1 开始, 依次尝试每个数字  
* 3. 对于每个数字, 有两种选择: 选择或不选择  
* 4. 通过递归和回溯生成所有满足条件的组合  
* 5. 剪枝优化: 提前终止不可能产生有效解的分支  
*  
* 剪枝策略:  
* 1. 可行性剪枝: 当当前和超过目标值时剪枝  
* 2. 最优性剪枝: 当已选择的数字个数超过 k 时剪枝  
* 3. 边界剪枝: 当剩余可选数字不足时剪枝  
* 4. 范围剪枝: 只在 1-9 范围内选择数字  
*  
* 时间复杂度: O(C(9, k)), 其中 C(9, k) 是组合数  
* 空间复杂度: O(k), 递归栈深度和存储路径的空间  
*  
* 工程化考量:  
* 1. 边界处理: 处理 k=0 或 n=0 的特殊情况  
* 2. 参数验证: 验证输入参数的有效性  
* 3. 性能优化: 通过剪枝减少不必要的计算  
* 4. 内存管理: 合理使用数据结构减少内存占用  
* 5. 可读性: 添加详细注释和变量命名  
* 6. 异常处理: 处理可能的异常情况  
* 7. 模块化设计: 将核心逻辑封装成独立方法  
* 8. 可维护性: 添加详细注释和文档说明  
*/  
import java.util.ArrayList;
```

```
import java.util.List;

public class LeetCode216_CombinationSumIII {

    /**
     * 找出所有相加之和为 n 的 k 个数的组合
     *
     * @param k 组合中数字的个数
     * @param n 目标和
     * @return 所有满足条件的组合
     */
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> result = new ArrayList<>();

        // 边界条件检查
        if (k <= 0 || n <= 0 || k > 9 || n > 45) { // 1-9 的最大和是 45
            return result;
        }

        List<Integer> path = new ArrayList<>();
        backtrack(k, n, 1, 0, path, result);
        return result;
    }

    /**
     * 回溯函数生成组合
     *
     * @param k 组合中数字的个数
     * @param n 目标和
     * @param start 当前起始数字
     * @param currentSum 当前和
     * @param path 当前路径
     * @param result 结果列表
     */
    private void backtrack(int k, int n, int start, int currentSum, List<Integer> path,
                          List<List<Integer>> result) {
        // 终止条件：已选择 k 个数字
        if (path.size() == k) {
            // 检查和是否等于目标值
            if (currentSum == n) {
                result.add(new ArrayList<>(path));
            }
        }
        return;
    }
}
```

```

}

// 剪枝：如果已选择的数字个数超过 k 或当前和超过目标值，提前终止
if (path.size() > k || currentSum > n) {
    return;
}

// 从 start 开始尝试数字 1-9
for (int i = start; i <= 9; i++) {
    // 剪枝：如果加上当前数字后和超过目标值，由于数字递增，后面的数字更大，直接跳出循环
    if (currentSum + i > n) {
        break;
    }

    // 剪枝：如果剩余可选数字不足，提前终止
    if (9 - i + 1 < k - path.size()) {
        break;
    }

    // 选择当前数字
    path.add(i);

    // 递归处理下一个数字
    backtrack(k, n, i + 1, currentSum + i, path, result);

    // 回溯：撤销选择
    path.remove(path.size() - 1);
}

}

/***
 * 解法二：使用位运算枚举所有可能的组合
 *
 * @param k 组合中数字的个数
 * @param n 目标和
 * @return 所有满足条件的组合
 */
public List<List<Integer>> combinationSum3_2(int k, int n) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件检查
    if (k <= 0 || n <= 0 || k > 9 || n > 45) {
        return result;
    }

    // 使用位运算枚举所有可能的组合
    for (int i = 1; i <= 1000000000; i++) {
        if (countSetBits(i) == k && sumOfSetBits(i) == n) {
            result.add(convertToPath(i));
        }
    }
}

private List<Integer> convertToPath(int num) {
    List<Integer> path = new ArrayList<>();
    for (int i = 1; i <= 9; i++) {
        if ((num & (1 << i)) != 0) {
            path.add(i);
        }
    }
    return path;
}

private int countSetBits(int num) {
    int count = 0;
    while (num != 0) {
        if ((num & 1) == 1) {
            count++;
        }
        num = num >> 1;
    }
    return count;
}

private int sumOfSetBits(int num) {
    int sum = 0;
    for (int i = 1; i <= 9; i++) {
        if ((num & (1 << i)) != 0) {
            sum += i;
        }
    }
    return sum;
}

```

```

}

// 枚举所有可能的组合 (1-9 的子集)
for (int mask = 0; mask < (1 << 9); mask++) {
    // 检查组合中数字的个数是否为 k
    if (Integer.bitCount(mask) == k) {
        List<Integer> combination = new ArrayList<>();
        int sum = 0;

        // 构造组合并计算和
        for (int i = 0; i < 9; i++) {
            if ((mask & (1 << i)) != 0) {
                combination.add(i + 1);
                sum += i + 1;
            }
        }
    }

    // 检查和是否等于目标值
    if (sum == n) {
        result.add(combination);
    }
}

return result;
}

// 测试方法
public static void main(String[] args) {
    LeetCode216_CombinationSumIII solution = new LeetCode216_CombinationSumIII();

    // 测试用例 1
    int k1 = 3, n1 = 7;
    List<List<Integer>> result1 = solution.combinationSum3(k1, n1);
    System.out.println("输入: k = " + k1 + ", n = " + n1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int k2 = 3, n2 = 9;
    List<List<Integer>> result2 = solution.combinationSum3(k2, n2);
    System.out.println("\n输入: k = " + k2 + ", n = " + n2);
    System.out.println("输出: " + result2);
}

```

```

// 测试用例 3
int k3 = 4, n3 = 1;
List<List<Integer>> result3 = solution.combinationSum3(k3, n3);
System.out.println("\n 输入: k = " + k3 + ", n = " + n3);
System.out.println("输出: " + result3);

// 测试解法二
System.out.println("\n==== 解法二测试 ===");
List<List<Integer>> result4 = solution.combinationSum3_2(k1, n1);
System.out.println("输入: k = " + k1 + ", n = " + n1);
System.out.println("输出: " + result4);
}

}

```

=====

文件: leetcode216\_combination\_sum\_iii.cpp

=====

```

/**
 * LeetCode 216. 组合总和 III
 *
 * 题目描述:
 * 找出所有相加之和为 n 的 k 个数的组合。
 * 只能使用数字 1 到 9，每个数字最多使用一次。
 * 返回所有可能的有效组合的列表。该列表不能包含相同的组合两次。
 *
 * 示例:
 * 输入: k = 3, n = 7
 * 输出: [[1,2,4]]
 * 解释: 1 + 2 + 4 = 7, 没有其他符合的组合了。
 *
 * 输入: k = 3, n = 9
 * 输出: [[1,2,6], [1,3,5], [2,3,4]]
 *
 * 输入: k = 4, n = 1
 * 输出: []
 * 解释: 不存在有效的组合。在[1,9]范围内使用 4 个不同的数字，我们可以得到的最小和是 1+2+3+4 = 10，因为 10 > 1，没有有效的组合。
 *
 * 提示:
 * 2 <= k <= 9
 * 1 <= n <= 60
 *

```

- \* 链接: <https://leetcode.cn/problems/combination-sum-iii/>
- \*
- \* 算法思路:
  - \* 1. 使用回溯算法生成所有可能的组合
  - \* 2. 从数字 1 开始, 依次尝试每个数字
  - \* 3. 对于每个数字, 有两种选择: 选择或不选择
  - \* 4. 通过递归和回溯生成所有满足条件的组合
  - \* 5. 剪枝优化: 提前终止不可能产生有效解的分支
- \*
- \* 剪枝策略:
  - \* 1. 可行性剪枝: 当当前和超过目标值时剪枝
  - \* 2. 最优性剪枝: 当已选择的数字个数超过 k 时剪枝
  - \* 3. 边界剪枝: 当剩余可选数字不足时剪枝
  - \* 4. 范围剪枝: 只在 1-9 范围内选择数字
- \*
- \* 时间复杂度:  $O(C(9, k))$ , 其中  $C(9, k)$  是组合数
- \* 空间复杂度:  $O(k)$ , 递归栈深度和存储路径的空间
- \*
- \* 工程化考量:
  - \* 1. 边界处理: 处理  $k=0$  或  $n=0$  的特殊情况
  - \* 2. 参数验证: 验证输入参数的有效性
  - \* 3. 性能优化: 通过剪枝减少不必要的计算
  - \* 4. 内存管理: 合理使用数据结构减少内存占用
  - \* 5. 可读性: 添加详细注释和变量命名
  - \* 6. 异常处理: 处理可能的异常情况
  - \* 7. 模块化设计: 将核心逻辑封装成独立方法
  - \* 8. 可维护性: 添加详细注释和文档说明
- \*/

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

class LeetCode216_CombinationSumIII {
public:
    /**
     * 找出所有相加之和为 n 的 k 个数的组合
     *
     * @param k 组合中数字的个数
     * @param n 目标和
     * @return 所有满足条件的组合
     */
    vector<vector<int>> combinationSum3(int k, int n) {
```

```

vector<vector<int>> result;

// 边界条件检查
if (k <= 0 || n <= 0 || k > 9 || n > 45) { // 1-9 的最大和是 45
    return result;
}

vector<int> path;
backtrack(k, n, 1, 0, path, result);
return result;
}

private:
/***
 * 回溯函数生成组合
 *
 * @param k 组合中数字的个数
 * @param n 目标和
 * @param start 当前起始数字
 * @param currentSum 当前和
 * @param path 当前路径
 * @param result 结果列表
 */
void backtrack(int k, int n, int start, int currentSum, vector<int>& path,
vector<vector<int>>& result) {
    // 终止条件：已选择 k 个数字
    if (path.size() == k) {
        // 检查和是否等于目标值
        if (currentSum == n) {
            result.push_back(path);
        }
        return;
    }

    // 剪枝：如果已选择的数字个数超过 k 或当前和超过目标值，提前终止
    if (path.size() > k || currentSum > n) {
        return;
    }

    // 从 start 开始尝试数字 1-9
    for (int i = start; i <= 9; i++) {
        // 剪枝：如果加上当前数字后和超过目标值，由于数字递增，后面的数字更大，直接跳出循环
        if (currentSum + i > n) {

```

```

        break;
    }

    // 剪枝：如果剩余可选数字不足，提前终止
    if (9 - i + 1 < k - path.size()) {
        break;
    }

    // 选择当前数字
    path.push_back(i);

    // 递归处理下一个数字
    backtrack(k, n, i + 1, currentSum + i, path, result);

    // 回溯：撤销选择
    path.pop_back();
}

}

public:
/***
 * 解法二：使用位运算枚举所有可能的组合
 *
 * @param k 组合中数字的个数
 * @param n 目标和
 * @return 所有满足条件的组合
 */
vector<vector<int>> combinationSum3_2(int k, int n) {
    vector<vector<int>> result;

    // 边界条件检查
    if (k <= 0 || n <= 0 || k > 9 || n > 45) {
        return result;
    }

    // 枚举所有可能的组合（1-9 的子集）
    for (int mask = 0; mask < (1 << 9); mask++) {
        // 检查组合中数字的个数是否为 k
        if (__builtin_popcount(mask) == k) {
            vector<int> combination;
            int sum = 0;

            // 构造组合并计算和

```

```

        for (int i = 0; i < 9; i++) {
            if (mask & (1 << i)) {
                combination.push_back(i + 1);
                sum += i + 1;
            }
        }

        // 检查和是否等于目标值
        if (sum == n) {
            result.push_back(combination);
        }
    }

    return result;
}

};

// 测试方法
int main() {
    LeetCode216_CombinationSumIII solution;

    // 测试用例 1
    int k1 = 3, n1 = 7;
    vector<vector<int>> result1 = solution.combinationSum3(k1, n1);
    printf("输入: k = %d, n = %d\n", k1, n1);
    printf("输出: [");
    for (int i = 0; i < result1.size(); i++) {
        printf("[");
        for (int j = 0; j < result1[i].size(); j++) {
            printf("%d", result1[i][j]);
            if (j < result1[i].size() - 1) printf(",");
        }
        printf("]");
        if (i < result1.size() - 1) printf(",");
    }
    printf("]\n");
}

// 测试用例 2
int k2 = 3, n2 = 9;
vector<vector<int>> result2 = solution.combinationSum3(k2, n2);
printf("\n 输入: k = %d, n = %d\n", k2, n2);
printf("输出: [");

```

```

for (int i = 0; i < result2.size(); i++) {
    printf("[");
    for (int j = 0; j < result2[i].size(); j++) {
        printf("%d", result2[i][j]);
        if (j < result2[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result2.size() - 1) printf(",");
}
printf("]\n");

// 测试用例 3
int k3 = 4, n3 = 1;
vector<vector<int>> result3 = solution.combinationSum3(k3, n3);
printf("\n输入: k = %d, n = %d\n", k3, n3);
printf("输出: [");
for (int i = 0; i < result3.size(); i++) {
    printf("[");
    for (int j = 0; j < result3[i].size(); j++) {
        printf("%d", result3[i][j]);
        if (j < result3[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result3.size() - 1) printf(",");
}
printf("]\n");

// 测试解法二
printf("\n==== 解法二测试 ===\n");
vector<vector<int>> result4 = solution.combinationSum3_2(k1, n1);
printf("输入: k = %d, n = %d\n", k1, n1);
printf("输出: [");
for (int i = 0; i < result4.size(); i++) {
    printf("[");
    for (int j = 0; j < result4[i].size(); j++) {
        printf("%d", result4[i][j]);
        if (j < result4[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result4.size() - 1) printf(",");
}
printf("]\n");

```

```
    return 0;  
}
```

---

文件: leetcode216\_combination\_sum\_iii.py

---

```
#!/usr/bin/env python3  
# -*- coding: utf-8 -*-
```

```
"""
```

LeetCode 216. 组合总和 III

题目描述:

找出所有相加之和为 n 的 k 个数的组合。

只能使用数字 1 到 9，每个数字最多使用一次。

返回所有可能的有效组合的列表。该列表不能包含相同的组合两次。

示例:

输入: k = 3, n = 7

输出: [[1, 2, 4]]

解释: 1 + 2 + 4 = 7, 没有其他符合的组合了。

输入: k = 3, n = 9

输出: [[1, 2, 6], [1, 3, 5], [2, 3, 4]]

输入: k = 4, n = 1

输出: []

解释: 不存在有效的组合。在[1, 9]范围内使用 4 个不同的数字，我们可以得到的最小和是  $1+2+3+4 = 10$ ，因为  $10 > 1$ ，没有有效的组合。

提示:

$2 \leq k \leq 9$

$1 \leq n \leq 60$

链接: <https://leetcode.cn/problems/combination-sum-iii/>

算法思路:

1. 使用回溯算法生成所有可能的组合
2. 从数字 1 开始，依次尝试每个数字
3. 对于每个数字，有两种选择：选择或不选择
4. 通过递归和回溯生成所有满足条件的组合
5. 剪枝优化：提前终止不可能产生有效解的分支

剪枝策略：

1. 可行性剪枝：当当前和超过目标值时剪枝
2. 最优性剪枝：当已选择的数字个数超过 k 时剪枝
3. 边界剪枝：当剩余可选数字不足时剪枝
4. 范围剪枝：只在 1-9 范围内选择数字

时间复杂度： $O(C(9, k))$ ，其中  $C(9, k)$  是组合数

空间复杂度： $O(k)$ ，递归栈深度和存储路径的空间

工程化考量：

1. 边界处理：处理  $k=0$  或  $n=0$  的特殊情况
2. 参数验证：验证输入参数的有效性
3. 性能优化：通过剪枝减少不必要的计算
4. 内存管理：合理使用数据结构减少内存占用
5. 可读性：添加详细注释和变量命名
6. 异常处理：处理可能的异常情况
7. 模块化设计：将核心逻辑封装成独立方法
8. 可维护性：添加详细注释和文档说明

"""

```
class LeetCode216_CombinationSumIII:  
    def combination_sum3(self, k: int, n: int) -> list[list[int]]:  
        """  
        找出所有相加之和为 n 的 k 个数的组合  
        """
```

Args:

    k: 组合中数字的个数  
    n: 目标和

Returns:

    所有满足条件的组合  
    """

```
    result = []
```

```
    # 边界条件检查  
    if k <= 0 or n <= 0 or k > 9 or n > 45: # 1-9 的最大和是 45  
        return result
```

```
    path = []  
    self._backtrack(k, n, 1, 0, path, result)  
    return result
```

```
def _backtrack(self, k: int, n: int, start: int, current_sum: int, path: list, result: list)
-> None:
    """
    回溯函数生成组合

    Args:
        k: 组合中数字的个数
        n: 目标和
        start: 当前起始数字
        current_sum: 当前和
        path: 当前路径
        result: 结果列表
    """

    # 终止条件: 已选择 k 个数字
    if len(path) == k:
        # 检查和是否等于目标值
        if current_sum == n:
            result.append(path[:]) # 添加路径的副本
        return

    # 剪枝: 如果已选择的数字个数超过 k 或当前和超过目标值, 提前终止
    if len(path) > k or current_sum > n:
        return

    # 从 start 开始尝试数字 1-9
    for i in range(start, 10):
        # 剪枝: 如果加上当前数字后和超过目标值, 由于数字递增, 后面的数字更大, 直接跳出循环
        if current_sum + i > n:
            break

        # 剪枝: 如果剩余可选数字不足, 提前终止
        if 9 - i + 1 < k - len(path):
            break

        # 选择当前数字
        path.append(i)

        # 递归处理下一个数字
        self._backtrack(k, n, i + 1, current_sum + i, path, result)

        # 回溯: 撤销选择
        path.pop()
```

```
def combination_sum3_2(self, k: int, n: int) -> list[list[int]]:  
    """
```

解法二：使用位运算枚举所有可能的组合

Args:

k: 组合中数字的个数

n: 目标和

Returns:

所有满足条件的组合

```
"""
```

```
result = []
```

# 边界条件检查

```
if k <= 0 or n <= 0 or k > 9 or n > 45:  
    return result
```

# 枚举所有可能的组合（1-9 的子集）

```
for mask in range(1 << 9):  
    # 检查组合中数字的个数是否为 k  
    if bin(mask).count('1') == k:  
        combination = []  
        total = 0
```

# 构造组合并计算和

```
for i in range(9):  
    if mask & (1 << i):  
        combination.append(i + 1)  
        total += i + 1
```

# 检查和是否等于目标值

```
if total == n:  
    result.append(combination)
```

```
return result
```

```
def main():
```

```
    """测试示例"""
```

```
solution = LeetCode216_CombinationSumIII()
```

# 测试用例 1

```
k1, n1 = 3, 7
```

```

result1 = solution.combination_sum3(k1, n1)
print(f"输入: k = {k1}, n = {n1}")
print(f"输出: {result1}")

# 测试用例 2
k2, n2 = 3, 9
result2 = solution.combination_sum3(k2, n2)
print(f"\n 输入: k = {k2}, n = {n2}")
print(f"输出: {result2}")

# 测试用例 3
k3, n3 = 4, 1
result3 = solution.combination_sum3(k3, n3)
print(f"\n 输入: k = {k3}, n = {n3}")
print(f"输出: {result3}")

# 测试解法二
print("\n==== 解法二测试 ===")
result4 = solution.combination_sum3_2(k1, n1)
print(f"输入: k = {k1}, n = {n1}")
print(f"输出: {result4}")

if __name__ == "__main__":
    main()

```

=====

文件: leetcode46\_permutations.cpp

=====

```

/**
 * LeetCode 46. 全排列
 *
 * 题目描述:
 * 给定一个不含重复数字的数组 nums ，返回其所有可能的全排列。你可以按任意顺序返回答案。
 *
 * 示例:
 * 输入: nums = [1, 2, 3]
 * 输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
 *
 * 输入: nums = [0, 1]
 * 输出: [[0, 1], [1, 0]]
 *

```

```
* 输入: nums = [1]
* 输出: [[1]]
*
* 提示:
* 1 <= nums.length <= 6
* -10 <= nums[i] <= 10
* nums 中的所有整数互不相同
*
* 链接: https://leetcode.cn/problems/permutations/
*
* 算法思路:
* 1. 使用回溯算法生成所有可能的排列
* 2. 对于每个位置, 尝试放置每个未使用的数字
* 3. 通过递归和回溯生成所有满足条件的排列
* 4. 使用布尔数组标记数字是否已被使用
*
* 剪枝策略:
* 1. 可行性剪枝: 使用布尔数组避免重复使用数字
* 2. 最优性剪枝: 当已选择的数字个数等于数组长度时终止
* 3. 约束传播: 一旦某个数字被使用, 立即标记为已使用
*
* 时间复杂度: O(n! * n), 其中 n 是数组长度, 共有 n! 种排列, 每种排列需要 O(n) 时间复制
* 空间复杂度: O(n), 递归栈深度和存储路径的空间
*
* 工程化考量:
* 1. 边界处理: 处理空数组和单元素数组的特殊情况
* 2. 参数验证: 验证输入参数的有效性
* 3. 性能优化: 通过剪枝减少不必要的计算
* 4. 内存管理: 合理使用数据结构减少内存占用
* 5. 可读性: 添加详细注释和变量命名
* 6. 异常处理: 处理可能的异常情况
* 7. 模块化设计: 将核心逻辑封装成独立方法
* 8. 可维护性: 添加详细注释和文档说明
*/

```

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

class LeetCode46_Permutations {
public:
    /**

```

```

* 生成数组的所有全排列
*
* @param nums 输入数组
* @return 所有可能的全排列
*/
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> result;

    // 边界条件检查
    if (nums.empty()) {
        return result;
    }

    vector<int> path;
    vector<bool> used(nums.size(), false);
    backtrack(nums, path, used, result);
    return result;
}

private:
    /**
     * 回溯函数生成排列
     *
     * @param nums 输入数组
     * @param path 当前路径
     * @param used 标记数字是否已被使用的数组
     * @param result 结果列表
     */
    void backtrack(vector<int>& nums, vector<int>& path, vector<bool>& used, vector<vector<int>>& result) {
        // 终止条件：已选择所有数字
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        // 尝试每个未使用的数字
        for (int i = 0; i < nums.size(); i++) {
            // 可行性剪枝：如果数字已被使用，跳过
            if (used[i]) {
                continue;
            }

```

```

        // 选择当前数字
        path.push_back(nums[i]);
        used[i] = true;

        // 递归处理下一个位置
        backtrack(nums, path, used, result);

        // 回溯：撤销选择
        path.pop_back();
        used[i] = false;
    }
}

public:
    /**
     * 解法二：交换元素法生成排列
     * 通过交换数组元素生成所有排列，避免使用额外的标记数组
     *
     * @param nums 输入数组
     * @return 所有可能的全排列
     */
    vector<vector<int>> permute2(vector<int>& nums) {
        vector<vector<int>> result;

        // 边界条件检查
        if (nums.empty()) {
            return result;
        }

        backtrack2(nums, 0, result);
        return result;
    }
}

private:
    /**
     * 通过交换元素生成排列的回溯函数
     *
     * @param nums 数字列表
     * @param start 当前处理的位置
     * @param result 结果列表
     */
    void backtrack2(vector<int>& nums, int start, vector<vector<int>>& result) {
        // 终止条件：已处理完所有位置

```

```

    if (start == nums.size()) {
        result.push_back(nums);
        return;
    }

    // 尝试将每个后续元素交换到当前位置
    for (int i = start; i < nums.size(); i++) {
        // 交换元素
        swap(nums[start], nums[i]);

        // 递归处理下一个位置
        backtrack2(nums, start + 1, result);

        // 回溯：恢复交换前的状态
        swap(nums[start], nums[i]);
    }
}

public:
/***
 * 解法三：使用 STL 的 next_permutation 函数
 * 先对数组排序，然后使用 STL 函数生成所有排列
 *
 * @param nums 输入数组
 * @return 所有可能的全排列
 */
vector<vector<int>> permute3(vector<int>& nums) {
    vector<vector<int>> result;

    // 边界条件检查
    if (nums.empty()) {
        return result;
    }

    // 排序以确保从最小排列开始
    sort(nums.begin(), nums.end());

    // 添加第一个排列
    result.push_back(nums);

    // 生成所有后续排列
    while (next_permutation(nums.begin(), nums.end())) {
        result.push_back(nums);
    }
}

```

```
}

    return result;
}

};

// 测试方法
int main() {
    LeetCode46_Permutations solution;

    // 测试用例 1
    vector<int> nums1 = {1, 2, 3};
    vector<vector<int>> result1 = solution.permute(nums1);
    printf("输入: nums = [1,2,3]\n");
    printf("输出: [");
    for (int i = 0; i < result1.size(); i++) {
        printf("[");
        for (int j = 0; j < result1[i].size(); j++) {
            printf("%d", result1[i][j]);
            if (j < result1[i].size() - 1) printf(",");
        }
        printf("]");
        if (i < result1.size() - 1) printf(",");
    }
    printf("]\n");
}

// 测试用例 2
vector<int> nums2 = {0, 1};
vector<vector<int>> result2 = solution.permute(nums2);
printf("\n输入: nums = [0,1]\n");
printf("输出: [");
for (int i = 0; i < result2.size(); i++) {
    printf("[");
    for (int j = 0; j < result2[i].size(); j++) {
        printf("%d", result2[i][j]);
        if (j < result2[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result2.size() - 1) printf(",");
}
printf("]\n");

// 测试用例 3
```

```

vector<int> nums3 = {1};
vector<vector<int>> result3 = solution.permute(nums3);
printf("\n 输入: nums = [1]\n");
printf("输出: [");
for (int i = 0; i < result3.size(); i++) {
    printf("[");
    for (int j = 0; j < result3[i].size(); j++) {
        printf("%d", result3[i][j]);
        if (j < result3[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result3.size() - 1) printf(",");
}
printf("]\n");

// 测试解法二
printf("\n==== 解法二测试 ====\n");
vector<vector<int>> result4 = solution.permute2(nums1);
printf("输入: nums = [1, 2, 3]\n");
printf("输出: [");
for (int i = 0; i < result4.size(); i++) {
    printf("[");
    for (int j = 0; j < result4[i].size(); j++) {
        printf("%d", result4[i][j]);
        if (j < result4[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result4.size() - 1) printf(",");
}
printf("]\n");

return 0;
}

```

=====

文件: LeetCode46\_Permutations.java

=====

```

import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 46. 全排列

```

\*

\* 题目描述:

\* 给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。你可以按任意顺序返回答案。

\*

\* 示例:

\* 输入: `nums = [1, 2, 3]`

\* 输出: `[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]`

\*

\* 输入: `nums = [0, 1]`

\* 输出: `[[[0, 1], [1, 0]]`

\*

\* 输入: `nums = [1]`

\* 输出: `[[[1]]]`

\*

\* 提示:

\*  $1 \leq \text{nums.length} \leq 6$

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

\* `nums` 中的所有整数互不相同

\*

\* 链接: <https://leetcode.cn/problems/permutations/>

\*

\* 算法思路:

\* 1. 使用回溯算法生成所有可能的排列

\* 2. 对于每个位置，尝试放置每个未使用的数字

\* 3. 通过递归和回溯生成所有满足条件的排列

\* 4. 使用布尔数组标记数字是否已被使用

\*

\* 剪枝策略:

\* 1. 可行性剪枝: 使用布尔数组避免重复使用数字

\* 2. 最优性剪枝: 当已选择的数字个数等于数组长度时终止

\* 3. 约束传播: 一旦某个数字被使用，立即标记为已使用

\*

\* 时间复杂度:  $O(n! * n)$ ，其中  $n$  是数组长度，共有  $n!$  种排列，每种排列需要  $O(n)$  时间复制

\* 空间复杂度:  $O(n)$ ，递归栈深度和存储路径的空间

\*

\* 工程化考量:

\* 1. 边界处理: 处理空数组和单元素数组的特殊情况

\* 2. 参数验证: 验证输入参数的有效性

\* 3. 性能优化: 通过剪枝减少不必要的计算

\* 4. 内存管理: 合理使用数据结构减少内存占用

\* 5. 可读性: 添加详细注释和变量命名

\* 6. 异常处理: 处理可能的异常情况

\* 7. 模块化设计: 将核心逻辑封装成独立方法

```
* 8. 可维护性：添加详细注释和文档说明
*/
public class LeetCode46_Permutations {

    /**
     * 生成数组的所有全排列
     *
     * @param nums 输入数组
     * @return 所有可能的全排列
     */
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();

        // 边界条件检查
        if (nums == null || nums.length == 0) {
            return result;
        }

        List<Integer> path = new ArrayList<>();
        boolean[] used = new boolean[nums.length];
        backtrack(nums, path, used, result);
        return result;
    }

    /**
     * 回溯函数生成排列
     *
     * @param nums 输入数组
     * @param path 当前路径
     * @param used 标记数字是否已被使用的数组
     * @param result 结果列表
     */
    private void backtrack(int[] nums, List<Integer> path, boolean[] used, List<List<Integer>> result) {
        // 终止条件：已选择所有数字
        if (path.size() == nums.length) {
            result.add(new ArrayList<>(path));
            return;
        }

        // 尝试每个未使用的数字
        for (int i = 0; i < nums.length; i++) {
            // 可行性剪枝：如果数字已被使用，跳过

```

```

    if (used[i]) {
        continue;
    }

    // 选择当前数字
    path.add(nums[i]);
    used[i] = true;

    // 递归处理下一个位置
    backtrack(nums, path, used, result);

    // 回溯：撤销选择
    path.remove(path.size() - 1);
    used[i] = false;
}

}

/***
 * 解法二：交换元素法生成排列
 * 通过交换数组元素生成所有排列，避免使用额外的标记数组
 *
 * @param nums 输入数组
 * @return 所有可能的全排列
 */
public List<List<Integer>> permute2(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return result;
    }

    // 将数组转换为列表以便操作
    List<Integer> numList = new ArrayList<>();
    for (int num : nums) {
        numList.add(num);
    }

    backtrack2(numList, 0, result);
    return result;
}

/***

```

```

* 通过交换元素生成排列的回溯函数
*
* @param nums 数字列表
* @param start 当前处理的位置
* @param result 结果列表
*/
private void backtrack2(List<Integer> nums, int start, List<List<Integer>> result) {
    // 终止条件：已处理完所有位置
    if (start == nums.size()) {
        result.add(new ArrayList<>(nums));
        return;
    }

    // 尝试将每个后续元素交换到当前位置
    for (int i = start; i < nums.size(); i++) {
        // 交换元素
        swap(nums, start, i);

        // 递归处理下一个位置
        backtrack2(nums, start + 1, result);

        // 回溯：恢复交换前的状态
        swap(nums, start, i);
    }
}

/**
 * 交换列表中两个位置的元素
 *
* @param nums 数字列表
* @param i 位置 i
* @param j 位置 j
*/
private void swap(List<Integer> nums, int i, int j) {
    int temp = nums.get(i);
    nums.set(i, nums.get(j));
    nums.set(j, temp);
}

// 测试方法
public static void main(String[] args) {
    LeetCode46_Permutations solution = new LeetCode46_Permutations();
}

```

```

// 测试用例 1
int[] nums1 = {1, 2, 3};
List<List<Integer>> result1 = solution.permute(nums1);
System.out.println("输入: nums = [1,2,3]");
System.out.println("输出: " + result1);

// 测试用例 2
int[] nums2 = {0, 1};
List<List<Integer>> result2 = solution.permute(nums2);
System.out.println("\n 输入: nums = [0,1]");
System.out.println("输出: " + result2);

// 测试用例 3
int[] nums3 = {1};
List<List<Integer>> result3 = solution.permute(nums3);
System.out.println("\n 输入: nums = [1]");
System.out.println("输出: " + result3);

// 测试解法二
System.out.println("\n==== 解法二测试 ===");
List<List<Integer>> result4 = solution.permute2(nums1);
System.out.println("输入: nums = [1,2,3]");
System.out.println("输出: " + result4);
}

}

```

文件: leetcode46\_permutations.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

LeetCode 46. 全排列

题目描述:

给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。你可以按任意顺序返回答案。

示例:

输入: `nums` = [1, 2, 3]

输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

输入: nums = [0, 1]

输出: [[0, 1], [1, 0]]

输入: nums = [1]

输出: [[1]]

提示:

1 <= nums.length <= 6

-10 <= nums[i] <= 10

nums 中的所有整数互不相同

链接: <https://leetcode.cn/problems/permutations/>

算法思路:

1. 使用回溯算法生成所有可能的排列
2. 对于每个位置，尝试放置每个未使用的数字
3. 通过递归和回溯生成所有满足条件的排列
4. 使用布尔数组标记数字是否已被使用

剪枝策略:

1. 可行性剪枝: 使用布尔数组避免重复使用数字
2. 最优性剪枝: 当已选择的数字个数等于数组长度时终止
3. 约束传播: 一旦某个数字被使用, 立即标记为已使用

时间复杂度:  $O(n! * n)$ , 其中  $n$  是数组长度, 共有  $n!$  种排列, 每种排列需要  $O(n)$  时间复制

空间复杂度:  $O(n)$ , 递归栈深度和存储路径的空间

工程化考量:

1. 边界处理: 处理空数组和单元素数组的特殊情况
2. 参数验证: 验证输入参数的有效性
3. 性能优化: 通过剪枝减少不必要的计算
4. 内存管理: 合理使用数据结构减少内存占用
5. 可读性: 添加详细注释和变量命名
6. 异常处理: 处理可能的异常情况
7. 模块化设计: 将核心逻辑封装成独立方法
8. 可维护性: 添加详细注释和文档说明

"""

```
class LeetCode46_Permutations:
```

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

```
        """
```

生成数组的所有全排列

Args:

    nums: 输入数组

Returns:

    所有可能的全排列

"""

result = []

# 边界条件检查

if not nums:

    return result

path = []

used = [False] \* len(nums)

self.\_backtrack(nums, path, used, result)

return result

```
def _backtrack(self, nums: list[int], path: list[int], used: list[bool], result: list[list[int]]) -> None:
```

"""

回溯函数生成排列

Args:

    nums: 输入数组

    path: 当前路径

    used: 标记数字是否已被使用的数组

    result: 结果列表

"""

# 终止条件: 已选择所有数字

if len(path) == len(nums):

    result.append(path[:]) # 添加路径的副本

    return

# 尝试每个未使用的数字

for i in range(len(nums)):

    # 可行性剪枝: 如果数字已被使用, 跳过

    if used[i]:

        continue

    # 选择当前数字

    path.append(nums[i])

    used[i] = True

```
# 递归处理下一个位置
self._backtrack(nums, path, used, result)

# 回溯：撤销选择
path.pop()
used[i] = False
```

```
def permute2(self, nums: list[int]) -> list[list[int]]:
    """
```

解法二：交换元素法生成排列

通过交换数组元素生成所有排列，避免使用额外的标记数组

Args:

nums: 输入数组

Returns:

所有可能的全排列

```
"""
```

```
result = []
```

# 边界条件检查

```
if not nums:
    return result
```

```
self._backtrack2(nums, 0, result)
```

```
return result
```

```
def _backtrack2(self, nums: list[int], start: int, result: list[list[int]]) -> None:
    """
```

通过交换元素生成排列的回溯函数

Args:

nums: 数字列表

start: 当前处理的位置

result: 结果列表

```
"""
```

# 终止条件：已处理完所有位置

```
if start == len(nums):
    result.append(nums[:]) # 添加当前排列的副本
    return
```

# 尝试将每个后续元素交换到当前位置

```
for i in range(start, len(nums)):
```

```
# 交换元素
nums[start], nums[i] = nums[i], nums[start]

# 递归处理下一个位置
self._backtrack2(nums, start + 1, result)

# 回溯：恢复交换前的状态
nums[start], nums[i] = nums[i], nums[start]

def permute3(self, nums: list[int]) -> list[list[int]]:
    """
    解法三：使用 itertools.permutations
    利用 Python 标准库生成排列
    """
```

Args:

nums: 输入数组

Returns:

所有可能的全排列

"""

```
import itertools
return [list(p) for p in itertools.permutations(nums)]
```

```
def main():
    """测试示例"""
    solution = LeetCode46_Permutations()
```

```
# 测试用例 1
nums1 = [1, 2, 3]
result1 = solution.permute(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result1}")
```

```
# 测试用例 2
nums2 = [0, 1]
result2 = solution.permute(nums2)
print(f"\n输入: nums = {nums2}")
print(f"输出: {result2}")
```

```
# 测试用例 3
nums3 = [1]
result3 = solution.permute(nums3)
```

```
print(f"\n输入: nums = {nums3}")
print(f"输出: {result3}")

# 测试解法二
print("\n==== 解法二测试 ===")
result4 = solution.permute2(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result4}")

# 测试解法三
print("\n==== 解法三测试 ===")
result5 = solution.permute3(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result5}")

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

```
=====
```

文件: LeetCode47\_PermutationsII.java

```
=====
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * LeetCode 47. 全排列 II
 *
 * 题目描述:
 * 给定一个可包含重复数字的序列 nums ，按任意顺序返回所有不重复的全排列。
 *
 * 示例:
 * 输入: nums = [1,1,2]
 * 输出:
 * [[1,1,2],
 *  [1,2,1],
 *  [2,1,1]]
 *
 * 输入: nums = [1,2,3]
 * 输出: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
 *
```

```
* 提示:  
* 1 <= nums.length <= 8  
* -10 <= nums[i] <= 10  
*  
* 链接: https://leetcode.cn/problems/permutations-ii/  
*  
* 算法思路:  
* 1. 使用回溯算法生成所有可能的排列  
* 2. 先对数组进行排序，使相同元素相邻  
* 3. 对于每个位置，尝试放置每个未使用的数字  
* 4. 通过递归和回溯生成所有满足条件的排列  
* 5. 使用布尔数组标记数字是否已被使用  
* 6. 通过剪枝避免生成重复的排列  
*  
* 剪枝策略:  
* 1. 可行性剪枝：使用布尔数组避免重复使用数字  
* 2. 最优性剪枝：当已选择的数字个数等于数组长度时终止  
* 3. 约束传播：一旦某个数字被使用，立即标记为已使用  
* 4. 重复剪枝：对于相同元素，只允许第一个未使用的元素被选择  
*  
* 时间复杂度：O(n! * n)，其中 n 是数组长度，共有 n! 种排列，每种排列需要 O(n) 时间复制  
* 空间复杂度：O(n)，递归栈深度和存储路径的空间  
*  
* 工程化考量:  
* 1. 边界处理：处理空数组和单元素数组的特殊情况  
* 2. 参数验证：验证输入参数的有效性  
* 3. 性能优化：通过剪枝减少不必要的计算  
* 4. 内存管理：合理使用数据结构减少内存占用  
* 5. 可读性：添加详细注释和变量命名  
* 6. 异常处理：处理可能的异常情况  
* 7. 模块化设计：将核心逻辑封装成独立方法  
* 8. 可维护性：添加详细注释和文档说明  
*/  
  
public class LeetCode47_PermutationsII {  
  
    /**  
     * 生成数组的所有不重复全排列  
     *  
     * @param nums 输入数组（可能包含重复元素）  
     * @return 所有不重复的全排列  
     */  
  
    public List<List<Integer>> permuteUnique(int[] nums) {  
        List<List<Integer>> result = new ArrayList<>();
```

```

// 边界条件检查
if (nums == null || nums.length == 0) {
    return result;
}

// 排序使相同元素相邻，便于剪枝
Arrays.sort(nums);

List<Integer> path = new ArrayList<>();
boolean[] used = new boolean[nums.length];
backtrack(nums, path, used, result);
return result;
}

/**
 * 回溯函数生成不重复排列
 *
 * @param nums 输入数组
 * @param path 当前路径
 * @param used 标记数字是否已被使用的数组
 * @param result 结果列表
 */
private void backtrack(int[] nums, List<Integer> path, boolean[] used, List<List<Integer>>
result) {
    // 终止条件：已选择所有数字
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 尝试每个未使用的数字
    for (int i = 0; i < nums.length; i++) {
        // 可行性剪枝：如果数字已被使用，跳过
        if (used[i]) {
            continue;
        }

        // 重复剪枝：对于相同元素，只允许第一个未使用的元素被选择
        // 如果当前元素与前一个元素相同，且前一个元素未被使用，则跳过当前元素
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }
    }
}

```

```

// 选择当前数字
path.add(nums[i]);
used[i] = true;

// 递归处理下一个位置
backtrack(nums, path, used, result);

// 回溯：撤销选择
path.remove(path.size() - 1);
used[i] = false;
}

}

/***
 * 解法二：使用计数法处理重复元素
 * 统计每个元素的出现次数，然后基于计数生成排列
 *
 * @param nums 输入数组（可能包含重复元素）
 * @return 所有不重复的全排列
 */
public List<List<Integer>> permuteUnique2(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return result;
    }

    // 统计每个元素的出现次数
    int[] counts = new int[21]; // 数值范围是[-10, 10]，偏移10映射到[0, 20]
    for (int num : nums) {
        counts[num + 10]++;
    }

    List<Integer> path = new ArrayList<>();
    backtrack2(counts, nums.length, path, result);
    return result;
}

/***
 * 基于计数的回溯函数
 *

```

```

* @param counts 每个元素的出现次数
* @param remaining 剩余需要选择的元素个数
* @param path 当前路径
* @param result 结果列表
*/
private void backtrack2(int[] counts, int remaining, List<Integer> path, List<List<Integer>>
result) {
    // 终止条件：已选择所有数字
    if (remaining == 0) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 尝试每个可用的数字
    for (int i = 0; i < 21; i++) {
        // 如果当前数字还有剩余
        if (counts[i] > 0) {
            // 选择当前数字
            path.add(i - 10); // 偏移回原始值
            counts[i]--;
            // 递归处理剩余位置
            backtrack2(counts, remaining - 1, path, result);

            // 回溯：撤销选择
            path.remove(path.size() - 1);
            counts[i]++;
        }
    }
}

// 测试方法
public static void main(String[] args) {
    LeetCode47_PermutationsII solution = new LeetCode47_PermutationsII();

    // 测试用例 1
    int[] nums1 = {1, 1, 2};
    List<List<Integer>> result1 = solution.permuteUnique(nums1);
    System.out.println("输入: nums = [1, 1, 2]");
    System.out.println("输出: " + result1);

    // 测试用例 2
    int[] nums2 = {1, 2, 3};
}

```

```

List<List<Integer>> result2 = solution.permuteUnique(nums2);
System.out.println("\n输入: nums = [1, 2, 3]");
System.out.println("输出: " + result2);

// 测试用例 3
int[] nums3 = {2, 2, 1, 1};
List<List<Integer>> result3 = solution.permuteUnique(nums3);
System.out.println("\n输入: nums = [2, 2, 1, 1]");
System.out.println("输出: " + result3);

// 测试解法二
System.out.println("\n==== 解法二测试 ===");
List<List<Integer>> result4 = solution.permuteUnique2(nums1);
System.out.println("输入: nums = [1, 1, 2]");
System.out.println("输出: " + result4);
}

}
=====
```

文件: leetcode47\_permutations\_ii.cpp

```

=====
/** 
 * LeetCode 47. 全排列 II
 *
 * 题目描述:
 * 给定一个可包含重复数字的序列 nums，按任意顺序返回所有不重复的全排列。
 *
 * 示例:
 * 输入: nums = [1, 1, 2]
 * 输出:
 * [[1, 1, 2],
 *  [1, 2, 1],
 *  [2, 1, 1]]
 *
 * 输入: nums = [1, 2, 3]
 * 输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
 *
 * 提示:
 * 1 <= nums.length <= 8
 * -10 <= nums[i] <= 10
 *
 * 链接: https://leetcode.cn/problems/permutations-ii/

```

\*

- \* 算法思路:
  - \* 1. 使用回溯算法生成所有可能的排列
  - \* 2. 先对数组进行排序，使相同元素相邻
  - \* 3. 对于每个位置，尝试放置每个未使用的数字
  - \* 4. 通过递归和回溯生成所有满足条件的排列
  - \* 5. 使用布尔数组标记数字是否已被使用
  - \* 6. 通过剪枝避免生成重复的排列
- \*
- \* 剪枝策略:
  - \* 1. 可行性剪枝：使用布尔数组避免重复使用数字
  - \* 2. 最优性剪枝：当已选择的数字个数等于数组长度时终止
  - \* 3. 约束传播：一旦某个数字被使用，立即标记为已使用
  - \* 4. 重复剪枝：对于相同元素，只允许第一个未使用的元素被选择
- \*
- \* 时间复杂度： $O(n! * n)$ ，其中  $n$  是数组长度，共有  $n!$  种排列，每种排列需要  $O(n)$  时间复制
- \* 空间复杂度： $O(n)$ ，递归栈深度和存储路径的空间
- \*
- \* 工程化考量:
  - \* 1. 边界处理：处理空数组和单元素数组的特殊情况
  - \* 2. 参数验证：验证输入参数的有效性
  - \* 3. 性能优化：通过剪枝减少不必要的计算
  - \* 4. 内存管理：合理使用数据结构减少内存占用
  - \* 5. 可读性：添加详细注释和变量命名
  - \* 6. 异常处理：处理可能的异常情况
  - \* 7. 模块化设计：将核心逻辑封装成独立方法
  - \* 8. 可维护性：添加详细注释和文档说明
- \*/

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

class LeetCode47_PermutationsII {
public:
    /**
     * 生成数组的所有不重复全排列
     *
     * @param nums 输入数组（可能包含重复元素）
     * @return 所有不重复的全排列
     */
}
```

```

vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<vector<int>> result;

    // 边界条件检查
    if (nums.empty()) {
        return result;
    }

    // 排序使相同元素相邻，便于剪枝
    sort(nums.begin(), nums.end());

    vector<int> path;
    vector<bool> used(nums.size(), false);
    backtrack(nums, path, used, result);
    return result;
}

private:
    /**
     * 回溯函数生成不重复排列
     *
     * @param nums 输入数组
     * @param path 当前路径
     * @param used 标记数字是否已被使用的数组
     * @param result 结果列表
     */
    void backtrack(vector<int>& nums, vector<int>& path, vector<bool>& used, vector<vector<int>>& result) {
        // 终止条件：已选择所有数字
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        // 尝试每个未使用的数字
        for (int i = 0; i < nums.size(); i++) {
            // 可行性剪枝：如果数字已被使用，跳过
            if (used[i]) {
                continue;
            }

            // 重复剪枝：对于相同元素，只允许第一个未使用的元素被选择
            // 如果当前元素与前一个元素相同，且前一个元素未被使用，则跳过当前元素

```

```

        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }

        // 选择当前数字
        path.push_back(nums[i]);
        used[i] = true;

        // 递归处理下一个位置
        backtrack(nums, path, used, result);

        // 回溯：撤销选择
        path.pop_back();
        used[i] = false;
    }
}

public:
/***
 * 解法二：使用计数法处理重复元素
 * 统计每个元素的出现次数，然后基于计数生成排列
 *
 * @param nums 输入数组（可能包含重复元素）
 * @return 所有不重复的全排列
 */
vector<vector<int>> permuteUnique2(vector<int>& nums) {
    vector<vector<int>> result;

    // 边界条件检查
    if (nums.empty()) {
        return result;
    }

    // 统计每个元素的出现次数
    map<int, int> counts;
    for (int num : nums) {
        counts[num]++;
    }

    vector<int> path;
    backtrack2(counts, nums.size(), path, result);
    return result;
}

```

```
private:
    /**
     * 基于计数的回溯函数
     *
     * @param counts 每个元素的出现次数
     * @param remaining 剩余需要选择的元素个数
     * @param path 当前路径
     * @param result 结果列表
     */
    void backtrack2(map<int, int>& counts, int remaining, vector<int>& path, vector<vector<int>>& result) {
        // 终止条件：已选择所有数字
        if (remaining == 0) {
            result.push_back(path);
            return;
        }

        // 尝试每个可用的数字
        for (auto& pair : counts) {
            int num = pair.first;
            int count = pair.second;

            // 如果当前数字还有剩余
            if (count > 0) {
                // 选择当前数字
                path.push_back(num);
                counts[num]--;
                backtrack2(counts, remaining - 1, path, result);

                // 回溯：撤销选择
                path.pop_back();
                counts[num]++;
            }
        }
    }

};

// 测试方法
int main() {
    LeetCode47_PermutationsII solution;
```

```
// 测试用例 1
vector<int> nums1 = {1, 1, 2};
vector<vector<int>> result1 = solution.permuteUnique(nums1);
printf("输入: nums = [1,1,2]\n");
printf("输出: [");
for (int i = 0; i < result1.size(); i++) {
    printf("[");
    for (int j = 0; j < result1[i].size(); j++) {
        printf("%d", result1[i][j]);
        if (j < result1[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result1.size() - 1) printf(",");
}
printf("]\n");
```

```
// 测试用例 2
vector<int> nums2 = {1, 2, 3};
vector<vector<int>> result2 = solution.permuteUnique(nums2);
printf("\n 输入: nums = [1,2,3]\n");
printf("输出: [");
for (int i = 0; i < result2.size(); i++) {
    printf("[");
    for (int j = 0; j < result2[i].size(); j++) {
        printf("%d", result2[i][j]);
        if (j < result2[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result2.size() - 1) printf(",");
}
printf("]\n");
```

```
// 测试用例 3
vector<int> nums3 = {2, 2, 1, 1};
vector<vector<int>> result3 = solution.permuteUnique(nums3);
printf("\n 输入: nums = [2,2,1,1]\n");
printf("输出: [");
for (int i = 0; i < result3.size(); i++) {
    printf("[");
    for (int j = 0; j < result3[i].size(); j++) {
        printf("%d", result3[i][j]);
        if (j < result3[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result3.size() - 1) printf(",");
}
```

```

    }
    printf("]");
    if (i < result3.size() - 1) printf(",");
}
printf("]\n");

// 测试解法二
printf("\n==== 解法二测试 ====\n");
vector<vector<int>> result4 = solution.permuteUnique2(nums1);
printf("输入: nums = [1, 1, 2]\n");
printf("输出: [");
for (int i = 0; i < result4.size(); i++) {
    printf("[");
    for (int j = 0; j < result4[i].size(); j++) {
        printf("%d", result4[i][j]);
        if (j < result4[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result4.size() - 1) printf(",");
}
printf("]\n");

return 0;
}

```

文件: leetcode47\_permutations\_ii.py

```
=====
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

LeetCode 47. 全排列 II

题目描述:

给定一个可包含重复数字的序列 `nums`，按任意顺序返回所有不重复的全排列。

示例:

输入: `nums = [1, 1, 2]`

输出:

`[[1, 1, 2],  
 [1, 2, 1],`

[2, 1, 1]]

输入: nums = [1, 2, 3]  
输出: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

提示:

1 <= nums.length <= 8  
-10 <= nums[i] <= 10

链接: <https://leetcode.cn/problems/permutations-ii/>

算法思路:

1. 使用回溯算法生成所有可能的排列
2. 先对数组进行排序，使相同元素相邻
3. 对于每个位置，尝试放置每个未使用的数字
4. 通过递归和回溯生成所有满足条件的排列
5. 使用布尔数组标记数字是否已被使用
6. 通过剪枝避免生成重复的排列

剪枝策略:

1. 可行性剪枝: 使用布尔数组避免重复使用数字
2. 最优性剪枝: 当已选择的数字个数等于数组长度时终止
3. 约束传播: 一旦某个数字被使用，立即标记为已使用
4. 重复剪枝: 对于相同元素，只允许第一个未使用的元素被选择

时间复杂度:  $O(n! * n)$ ，其中  $n$  是数组长度，共有  $n!$  种排列，每种排列需要  $O(n)$  时间复制

空间复杂度:  $O(n)$ ，递归栈深度和存储路径的空间

工程化考量:

1. 边界处理: 处理空数组和单元素数组的特殊情况
2. 参数验证: 验证输入参数的有效性
3. 性能优化: 通过剪枝减少不必要的计算
4. 内存管理: 合理使用数据结构减少内存占用
5. 可读性: 添加详细注释和变量命名
6. 异常处理: 处理可能的异常情况
7. 模块化设计: 将核心逻辑封装成独立方法
8. 可维护性: 添加详细注释和文档说明

"""

```
class LeetCode47_PermutationsII:  
    def permute_unique(self, nums: list[int]) -> list[list[int]]:  
        """  
        生成数组的所有不重复全排列  
        """
```

Args:

    nums: 输入数组（可能包含重复元素）

Returns:

    所有不重复的全排列

"""

result = []

# 边界条件检查

if not nums:

    return result

# 排序使相同元素相邻，便于剪枝

nums.sort()

path = []

used = [False] \* len(nums)

self.\_backtrack(nums, path, used, result)

return result

def \_backtrack(self, nums: list[int], path: list[int], used: list[bool], result: list[list[int]]) -> None:

"""

回溯函数生成不重复排列

Args:

    nums: 输入数组

    path: 当前路径

    used: 标记数字是否已被使用的数组

    result: 结果列表

"""

# 终止条件：已选择所有数字

if len(path) == len(nums):

    result.append(path[:]) # 添加路径的副本

    return

# 尝试每个未使用的数字

for i in range(len(nums)):

    # 可行性剪枝：如果数字已被使用，跳过

    if used[i]:

        continue

```

# 重复剪枝：对于相同元素，只允许第一个未使用的元素被选择
# 如果当前元素与前一个元素相同，且前一个元素未被使用，则跳过当前元素
if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
    continue

# 选择当前数字
path.append(nums[i])
used[i] = True

# 递归处理下一个位置
self._backtrack(nums, path, used, result)

# 回溯：撤销选择
path.pop()
used[i] = False

```

```
def permute_unique2(self, nums: list[int]) -> list[list[int]]:
    """

```

解法二：使用计数法处理重复元素

统计每个元素的出现次数，然后基于计数生成排列

Args:

nums: 输入数组（可能包含重复元素）

Returns:

所有不重复的全排列

"""

result = []

# 边界条件检查

if not nums:

return result

# 统计每个元素的出现次数

from collections import Counter

counter = Counter(nums)

path = []

self.\_backtrack2(counter, len(nums), path, result)

return result

```
def _backtrack2(self, counter: dict, remaining: int, path: list[int], result:
list[list[int]]) -> None:
```

```
"""
```

## 基于计数的回溯函数

Args:

counter: 每个元素的出现次数

remaining: 剩余需要选择的元素个数

path: 当前路径

result: 结果列表

```
"""
```

# 终止条件: 已选择所有数字

```
if remaining == 0:
```

```
    result.append(path[:]) # 添加路径的副本
```

```
    return
```

# 尝试每个可用的数字

```
for num in list(counter.keys()):
```

# 如果当前数字还有剩余

```
if counter[num] > 0:
```

# 选择当前数字

```
path.append(num)
```

```
counter[num] -= 1
```

# 递归处理剩余位置

```
self._backtrack2(counter, remaining - 1, path, result)
```

# 回溯: 撤销选择

```
path.pop()
```

```
counter[num] += 1
```

```
def permute_unique3(self, nums: list[int]) -> list[list[int]]:
```

```
"""
```

解法三: 使用 set 去重

生成所有排列后使用 set 去除重复

Args:

nums: 输入数组 (可能包含重复元素)

Returns:

所有不重复的全排列

```
"""
```

```
from itertools import permutations
```

# 使用 set 去除重复排列, 然后转换为列表格式

```
return [list(p) for p in set(permutations(nums))]
```

```
def main():
    """测试示例"""
    solution = LeetCode47_PermutationsII()

    # 测试用例 1
    nums1 = [1, 1, 2]
    result1 = solution.permute_unique(nums1)
    print(f"输入: nums = {nums1}")
    print(f"输出: {result1}")

    # 测试用例 2
    nums2 = [1, 2, 3]
    result2 = solution.permute_unique(nums2)
    print(f"\n 输入: nums = {nums2}")
    print(f"输出: {result2}")

    # 测试用例 3
    nums3 = [2, 2, 1, 1]
    result3 = solution.permute_unique(nums3)
    print(f"\n 输入: nums = {nums3}")
    print(f"输出: {result3}")

    # 测试解法二
    print("\n== 解法二测试 ==")
    result4 = solution.permute_unique2(nums1)
    print(f"输入: nums = {nums1}")
    print(f"输出: {result4}")

    # 测试解法三
    print("\n== 解法三测试 ==")
    result5 = solution.permute_unique3(nums1)
    print(f"输入: nums = {nums1}")
    print(f"输出: {result5}")

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

```
=====
/*
 * LeetCode 52. N皇后 II
 *
 * 题目描述:
 * n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
 * 给定一个整数 n，返回 n 皇后不同的解决方案的数量。
 *
 * 示例:
 * 输入: 4
 * 输出: 2
 * 解释: 4 皇后问题存在两个不同的解法。
 *
 * 输入: 1
 * 输出: 1
 *
 * 提示:
 * 1 <= n <= 9
 *
 * 链接: https://leetcode.cn/problems/n-queens-ii/
 *
 * 算法思路:
 * 1. 使用回溯算法解决 N 皇后问题
 * 2. 按行放置皇后，每行放置一个
 * 3. 对于每一行，尝试在每一列放置皇后
 * 4. 检查当前位置是否与已放置的皇后冲突
 * 5. 如果不冲突，递归处理下一行
 * 6. 如果冲突，尝试下一列
 * 7. 如果所有列都尝试过都不行，回溯到上一行
 * 8. 与 N-Queens I 不同的是，这里只需要统计解的数量，不需要记录具体解
 *
 * 剪枝策略:
 * 1. 可行性剪枝：在放置皇后时检查是否与已放置的皇后冲突
 * 2. 约束传播：一旦某一行无法放置皇后，立即回溯
 *
 * 时间复杂度: O(N!)，第一行有 N 种选择，第二行最多有 N-1 种选择，以此类推
 * 空间复杂度: O(N)，递归栈深度和三个布尔数组的空间
 *
 * 工程化考量:
 * 1. 使用三个布尔数组优化冲突检查:
 *      - cols[i]: 第 i 列是否有皇后
```

```

*      - diag1[i]: 第 i 条主对角线是否已有皇后
*      - diag2[i]: 第 i 条副对角线是否已有皇后
* 2. 主对角线标识: row - col + n - 1 (避免负数索引)
* 3. 副对角线标识: row + col
* 4. 边界处理: 处理 n=1 的特殊情况
* 5. 性能优化: 使用位运算可以进一步优化空间
* 6. 异常处理: 验证输入参数的有效性
*/
public class LeetCode52_NQueensII {

    private int count = 0;

    /**
     * 计算 N 皇后问题的不同解决方案数量
     *
     * @param n 皇后数量和棋盘大小
     * @return 不同解决方案的数量
     */
    public int totalNQueens(int n) {
        // 边界条件检查
        if (n <= 0) {
            return 0;
        }

        // 优化的冲突检查数组
        boolean[] cols = new boolean[n];           // 列冲突检查
        boolean[] diag1 = new boolean[2 * n - 1];   // 主对角线冲突检查
        boolean[] diag2 = new boolean[2 * n - 1];   // 副对角线冲突检查

        count = 0;
        backtrack(n, 0, cols, diag1, diag2);
        return count;
    }

    /**
     * 回溯函数计算 N 皇后问题的解数量
     *
     * @param n 棋盘大小
     * @param row 当前行
     * @param cols 列冲突检查数组
     * @param diag1 主对角线冲突检查数组
     * @param diag2 副对角线冲突检查数组
     */

```

```

private void backtrack(int n, int row, boolean[] cols, boolean[] diag1, boolean[] diag2) {
    // 终止条件：已放置完所有皇后
    if (row == n) {
        count++;
        return;
    }

    // 在当前行的每一列尝试放置皇后
    for (int col = 0; col < n; col++) {
        // 计算对角线索引
        int d1 = row - col + n - 1; // 主对角线索引（避免负数）
        int d2 = row + col; // 副对角线索引

        // 可行性剪枝：检查是否与已放置的皇后冲突
        if (!cols[col] && !diag1[d1] && !diag2[d2]) {
            // 放置皇后
            cols[col] = true;
            diag1[d1] = true;
            diag2[d2] = true;

            // 递归处理下一行
            backtrack(n, row + 1, cols, diag1, diag2);

            // 回溯：撤销放置
            cols[col] = false;
            diag1[d1] = false;
            diag2[d2] = false;
        }
    }
}

/**
 * 解法二：使用位运算优化的 N 皇后问题解法
 *
 * @param n 皇后数量和棋盘大小
 * @return 不同解决方案的数量
 */
public int totalNQueens2(int n) {
    if (n <= 0) {
        return 0;
    }

    count = 0;
}

```

```

// 使用位运算表示列、主对角线、副对角线的占用情况
backtrack2(n, 0, 0, 0, 0);
return count;
}

/***
 * 使用位运算的回溯函数
 *
 * @param n 棋盘大小
 * @param row 当前行
 * @param cols 列占用情况（二进制位表示）
 * @param diag1 主对角线占用情况
 * @param diag2 副对角线占用情况
 */
private void backtrack2(int n, int row, int cols, int diag1, int diag2) {
    // 终止条件：已放置完所有皇后
    if (row == n) {
        count++;
        return;
    }

    // 计算可以放置皇后的位置
    // ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ ) 表示不与任何皇后冲突的位置
    // ((1 << n) - 1) 用于限制在 n 位范围内
    int availablePositions = ((1 << n) - 1) & ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ );

    // 遍历所有可以放置皇后的位置
    while (availablePositions != 0) {
        // 获取最右边的可用位置
        int position = availablePositions & (-availablePositions);

        // 在该位置放置皇后
        backtrack2(n, row + 1,
                   cols | position,
                   (diag1 | position) << 1,
                   (diag2 | position) >> 1);

        // 移除已处理的位置
        availablePositions &= availablePositions - 1;
    }
}

// 测试方法

```

```

public static void main(String[] args) {
    LeetCode52_NQueensII solution = new LeetCode52_NQueensII();

    // 测试用例 1
    int n1 = 4;
    int result1 = solution.totalNQueens(n1);
    System.out.println("输入: n = " + n1);
    System.out.println("输出: " + result1);

    // 测试用例 2
    int n2 = 1;
    int result2 = solution.totalNQueens(n2);
    System.out.println("\n 输入: n = " + n2);
    System.out.println("输出: " + result2);

    // 测试用例 3
    int n3 = 8;
    int result3 = solution.totalNQueens(n3);
    System.out.println("\n 输入: n = " + n3);
    System.out.println("输出: " + result3);

    // 测试解法二
    System.out.println("\n==== 解法二测试 ====");
    int result4 = solution.totalNQueens2(n1);
    System.out.println("输入: n = " + n1);
    System.out.println("输出: " + result4);
}

}

```

文件: leetcode52\_n\_queens\_ii.cpp

```

=====
/** 
 * LeetCode 52. N 皇后 II
 *
 * 题目描述:
 * n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
 * 给定一个整数 n，返回 n 皇后不同的解决方案的数量。
 *
 * 示例:
 * 输入: 4
 * 输出: 2

```

```
* 解释: 4 皇后问题存在两个不同的解法。  
*  
* 输入: 1  
* 输出: 1  
*  
* 提示:  
* 1 <= n <= 9  
*  
* 链接: https://leetcode.cn/problems/n-queens-ii/  
*  
* 算法思路:  
* 1. 使用回溯算法解决 N 皇后问题  
* 2. 按行放置皇后, 每行放置一个  
* 3. 对于每一行, 尝试在每一列放置皇后  
* 4. 检查当前位置是否与已放置的皇后冲突  
* 5. 如果不冲突, 递归处理下一行  
* 6. 如果冲突, 尝试下一列  
* 7. 如果所有列都尝试过都不行, 回溯到上一行  
* 8. 与 N-Queens I 不同的是, 这里只需要统计解的数量, 不需要记录具体解  
*  
* 剪枝策略:  
* 1. 可行性剪枝: 在放置皇后时检查是否与已放置的皇后冲突  
* 2. 约束传播: 一旦某一行无法放置皇后, 立即回溯  
*  
* 时间复杂度: O(N!), 第一行有 N 种选择, 第二行最多有 N-1 种选择, 以此类推  
* 空间复杂度: O(N), 递归栈深度和三个布尔数组的空间  
*  
* 工程化考量:  
* 1. 使用三个布尔数组优化冲突检查:  
*   - cols[i]: 第 i 列是否已有皇后  
*   - diag1[i]: 第 i 条主对角线是否已有皇后  
*   - diag2[i]: 第 i 条副对角线是否已有皇后  
* 2. 主对角线标识: row - col + n - 1 (避免负数索引)  
* 3. 副对角线标识: row + col  
* 4. 边界处理: 处理 n=1 的特殊情况  
* 5. 性能优化: 使用位运算可以进一步优化空间  
* 6. 异常处理: 验证输入参数的有效性  
*/
```

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

```

class LeetCode52_NQueensII {
private:
    int count;

    /**
     * 回溯函数计算 N 皇后问题的解数量
     *
     * @param n 棋盘大小
     * @param row 当前行
     * @param cols 列冲突检查数组
     * @param diag1 主对角线冲突检查数组
     * @param diag2 副对角线冲突检查数组
     */
    void backtrack(int n, int row, vector<bool>& cols, vector<bool>& diag1, vector<bool>& diag2)
    {
        // 终止条件：已放置完所有皇后
        if (row == n) {
            count++;
            return;
        }

        // 在当前行的每一列尝试放置皇后
        for (int col = 0; col < n; col++) {
            // 计算对角线索引
            int d1 = row - col + n - 1; // 主对角线索引（避免负数）
            int d2 = row + col; // 副对角线索引

            // 可行性剪枝：检查是否与已放置的皇后冲突
            if (!cols[col] && !diag1[d1] && !diag2[d2]) {
                // 放置皇后
                cols[col] = true;
                diag1[d1] = true;
                diag2[d2] = true;

                // 递归处理下一行
                backtrack(n, row + 1, cols, diag1, diag2);

                // 回溯：撤销放置
                cols[col] = false;
                diag1[d1] = false;
                diag2[d2] = false;
            }
        }
    }
}

```

```

    }

}

/***
 * 使用位运算的回溯函数
 *
 * @param n 棋盘大小
 * @param row 当前行
 * @param cols 列占用情况（二进制位表示）
 * @param diag1 主对角线占用情况
 * @param diag2 副对角线占用情况
 */
void backtrack2(int n, int row, int cols, int diag1, int diag2) {
    // 终止条件：已放置完所有皇后
    if (row == n) {
        count++;
        return;
    }

    // 计算可以放置皇后的位置
    // ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ ) 表示不与任何皇后冲突的位置
    // ((1 << n) - 1) 用于限制在 n 位范围内
    int availablePositions = ((1 << n) - 1) & ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ );

    // 遍历所有可以放置皇后的位置
    while (availablePositions != 0) {
        // 获取最右边的可用位置
        int position = availablePositions & (-availablePositions);

        // 在该位置放置皇后
        backtrack2(n, row + 1,
                   cols | position,
                   (diag1 | position) << 1,
                   (diag2 | position) >> 1);

        // 移除已处理的位置
        availablePositions &= availablePositions - 1;
    }
}

public:
/***
 * 计算 N 皇后问题的不同解决方案数量
*/

```

```

/*
 * @param n 皇后数量和棋盘大小
 * @return 不同解决方案的数量
 */
int totalNQueens(int n) {
    // 边界条件检查
    if (n <= 0) {
        return 0;
    }

    // 优化的冲突检查数组
    vector<bool> cols(n, false);           // 列冲突检查
    vector<bool> diag1(2 * n - 1, false); // 主对角线冲突检查
    vector<bool> diag2(2 * n - 1, false); // 副对角线冲突检查

    count = 0;
    backtrack(n, 0, cols, diag1, diag2);
    return count;
}

/***
 * 解法二：使用位运算优化的 N 皇后问题解法
 *
 * @param n 皇后数量和棋盘大小
 * @return 不同解决方案的数量
 */
int totalNQueens2(int n) {
    if (n <= 0) {
        return 0;
    }

    count = 0;
    // 使用位运算表示列、主对角线、副对角线的占用情况
    backtrack2(n, 0, 0, 0, 0);
    return count;
}

// 测试方法
int main() {
    LeetCode52_NQueensII solution;

    // 测试用例 1

```

```

int n1 = 4;
int result1 = solution.totalNQueens(n1);
printf("输入: n = %d\n", n1);
printf("输出: %d\n", result1);

// 测试用例 2
int n2 = 1;
int result2 = solution.totalNQueens(n2);
printf("\n 输入: n = %d\n", n2);
printf("输出: %d\n", result2);

// 测试用例 3
int n3 = 8;
int result3 = solution.totalNQueens(n3);
printf("\n 输入: n = %d\n", n3);
printf("输出: %d\n", result3);

// 测试解法二
printf("\n==== 解法二测试 ====\n");
int result4 = solution.totalNQueens2(n1);
printf("输入: n = %d\n", n1);
printf("输出: %d\n", result4);

return 0;
}

```

=====

文件: leetcode52\_n\_queens\_i.py

=====

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

LeetCode 52. N皇后 II

**题目描述:**

n 皇后问题研究的是如何将 n 个皇后放置在  $n \times n$  的棋盘上，并且使皇后彼此之间不能相互攻击。给定一个整数 n，返回 n 皇后不同的解决方案的数量。

**示例:**

输入: 4

输出: 2

解释：4 皇后问题存在两个不同的解法。

输入：1

输出：1

提示：

$1 \leq n \leq 9$

链接：<https://leetcode.cn/problems/n-queens-ii/>

算法思路：

1. 使用回溯算法解决 N 皇后问题
2. 按行放置皇后，每行放置一个
3. 对于每一行，尝试在每一列放置皇后
4. 检查当前位置是否与已放置的皇后冲突
5. 如果不冲突，递归处理下一行
6. 如果冲突，尝试下一列
7. 如果所有列都尝试过都不行，回溯到上一行
8. 与 N-Queens I 不同的是，这里只需要统计解的数量，不需要记录具体解

剪枝策略：

1. 可行性剪枝：在放置皇后时检查是否与已放置的皇后冲突
2. 约束传播：一旦某一行无法放置皇后，立即回溯

时间复杂度： $O(N!)$ ，第一行有  $N$  种选择，第二行最多有  $N-1$  种选择，以此类推

空间复杂度： $O(N)$ ，递归栈深度和三个布尔数组的空间

工程化考量：

1. 使用三个布尔数组优化冲突检查：
  - `cols[i]`: 第  $i$  列是否已有皇后
  - `diag1[i]`: 第  $i$  条主对角线是否已有皇后
  - `diag2[i]`: 第  $i$  条副对角线是否已有皇后
2. 主对角线标识： $\text{row} - \text{col} + n - 1$  (避免负数索引)
3. 副对角线标识： $\text{row} + \text{col}$
4. 边界处理：处理  $n=1$  的特殊情况
5. 性能优化：使用位运算可以进一步优化空间
6. 异常处理：验证输入参数的有效性

"""

```
class LeetCode52_NQueensII:  
    def __init__(self):  
        self.count = 0
```

```

def total_n_queens(self, n: int) -> int:
    """
    计算 N 皇后问题的不同解决方案数量

    Args:
        n: 皇后数量和棋盘大小

    Returns:
        不同解决方案的数量
    """

    # 边界条件检查
    if n <= 0:
        return 0

    # 优化的冲突检查数组
    cols = [False] * n           # 列冲突检查
    diag1 = [False] * (2 * n - 1) # 主对角线冲突检查
    diag2 = [False] * (2 * n - 1) # 副对角线冲突检查

    self.count = 0
    self._backtrack(n, 0, cols, diag1, diag2)
    return self.count

def _backtrack(self, n: int, row: int, cols: list, diag1: list, diag2: list) -> None:
    """
    回溯函数计算 N 皇后问题的解数量

    Args:
        n: 棋盘大小
        row: 当前行
        cols: 列冲突检查数组
        diag1: 主对角线冲突检查数组
        diag2: 副对角线冲突检查数组
    """

    # 终止条件: 已放置完所有皇后
    if row == n:
        self.count += 1
        return

    # 在当前行的每一列尝试放置皇后
    for col in range(n):
        # 计算对角线索引
        d1 = row - col + n - 1 # 主对角线索引 (避免负数)
        d2 = row + col - 1      # 副对角线索引

        if not (cols[col] or diag1[d1] or diag2[d2]):
            cols[col] = True
            diag1[d1] = True
            diag2[d2] = True

            self._backtrack(n, row + 1, cols, diag1, diag2)

            cols[col] = False
            diag1[d1] = False
            diag2[d2] = False

```

```

d2 = row + col          # 副对角线索引

# 可行性剪枝：检查是否与已放置的皇后冲突
if not cols[col] and not diag1[d1] and not diag2[d2]:
    # 放置皇后
    cols[col] = True
    diag1[d1] = True
    diag2[d2] = True

    # 递归处理下一行
    self._backtrack(n, row + 1, cols, diag1, diag2)

    # 回溯：撤销放置
    cols[col] = False
    diag1[d1] = False
    diag2[d2] = False

```

def total\_n\_queens2(self, n: int) -> int:

"""

解法二：使用位运算优化的 N 皇后问题解法

Args:

n: 皇后数量和棋盘大小

Returns:

不同解决方案的数量

"""

if n <= 0:

return 0

self.count = 0

# 使用位运算表示列、主对角线、副对角线的占用情况

self.\_backtrack2(n, 0, 0, 0, 0)

return self.count

def \_backtrack2(self, n: int, row: int, cols: int, diag1: int, diag2: int) -> None:

"""

使用位运算的回溯函数

Args:

n: 棋盘大小

row: 当前行

cols: 列占用情况（二进制位表示）

```

diag1: 主对角线占用情况
diag2: 副对角线占用情况
"""
# 终止条件: 已放置完所有皇后
if row == n:
    self.count += 1
    return

# 计算可以放置皇后的位置
# ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ ) 表示不与任何皇后冲突的位置
# ((1 << n) - 1) 用于限制在 n 位范围内
available_positions = ((1 << n) - 1) & ( $\sim(\text{cols} \mid \text{diag1} \mid \text{diag2})$ )

# 遍历所有可以放置皇后的位置
while available_positions != 0:
    # 获取最右边的可用位置
    position = available_positions & ( $\sim\text{available\_positions}$ )
    # 在该位置放置皇后
    self._backtrack2(n, row + 1,
                      cols | position,
                      (diag1 | position) << 1,
                      (diag2 | position) >> 1)

    # 移除已处理的位置
    available_positions &= available_positions - 1

def main():
    """测试示例"""
    solution = LeetCode52_NQueensII()

    # 测试用例 1
    n1 = 4
    result1 = solution.total_n_queens(n1)
    print(f"输入: n = {n1}")
    print(f"输出: {result1}")

    # 测试用例 2
    n2 = 1
    result2 = solution.total_n_queens(n2)
    print(f"\n输入: n = {n2}")
    print(f"输出: {result2}")

```

```
# 测试用例 3
n3 = 8
result3 = solution.total_n_queens(n3)
print(f"\n 输入: n = {n3}")
print(f"输出: {result3}")

# 测试解法二
print("\n==== 解法二测试 ===")
result4 = solution.total_n_queens2(n1)
print(f"输入: n = {n1}")
print(f"输出: {result4}")

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

=====

文件: leetcode78\_subsets.cpp

=====

```
/***
 * LeetCode 78. 子集
 *
 * 题目描述:
 * 给你一个整数数组 nums，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。你可以按任意顺序返回解集。
 *
 * 示例:
 * 输入: nums = [1, 2, 3]
 * 输出: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
 *
 * 输入: nums = [0]
 * 输出: [[], [0]]
 *
 * 提示:
 * 1 <= nums.length <= 10
 * -10 <= nums[i] <= 10
 * nums 中的所有元素互不相同
 *
 * 链接: https://leetcode.cn/problems/subsets/
 *
 * 算法思路:
```

- \* 1. 使用回溯算法生成所有可能的子集
- \* 2. 对于每个元素，有两种选择：选择或不选择
- \* 3. 通过递归和回溯生成所有满足条件的子集
- \* 4. 每一步递归都将当前路径加入结果集
- \*
- \* 剪枝策略：
  - \* 1. 可行性剪枝：通过起始索引避免重复选择元素
  - \* 2. 最优性剪枝：当已选择的元素个数等于数组长度时终止
  - \* 3. 约束传播：一旦某个元素被选择，后续只能选择后面的元素
  - \*
  - \* 时间复杂度： $O(n * 2^n)$ ，其中 n 是数组长度，共有  $2^n$  个子集，每个子集需要  $O(n)$  时间复制
  - \* 空间复杂度： $O(n)$ ，递归栈深度和存储路径的空间
  - \*
- \* 工程化考量：
  - \* 1. 边界处理：处理空数组和单元素数组的特殊情况
  - \* 2. 参数验证：验证输入参数的有效性
  - \* 3. 性能优化：通过剪枝减少不必要的计算
  - \* 4. 内存管理：合理使用数据结构减少内存占用
  - \* 5. 可读性：添加详细注释和变量命名
  - \* 6. 异常处理：处理可能的异常情况
  - \* 7. 模块化设计：将核心逻辑封装成独立方法
  - \* 8. 可维护性：添加详细注释和文档说明
- \*/

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

class LeetCode78_Subsets {
public:
    /**
     * 生成数组的所有子集
     *
     * @param nums 输入数组
     * @return 所有可能的子集
     */
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> result;

        // 边界条件检查
        if (nums.empty()) {
            return result;
        }
    }
}
```

```

vector<int> path;
backtrack(nums, 0, path, result);
return result;
}

private:
/***
 * 回溯函数生成子集
 *
 * @param nums 输入数组
 * @param start 当前起始索引
 * @param path 当前路径
 * @param result 结果列表
 */
void backtrack(vector<int>& nums, int start, vector<int>& path, vector<vector<int>>& result)
{
    // 每一步都添加到结果中（空集也在其中）
    result.push_back(path);

    // 从 start 开始遍历，避免重复
    for (int i = start; i < nums.size(); i++) {
        // 选择当前元素
        path.push_back(nums[i]);

        // 递归处理下一个元素
        backtrack(nums, i + 1, path, result);

        // 回溯：撤销选择
        path.pop_back();
    }
}

public:
/***
 * 解法二：使用位运算枚举所有可能的子集
 * 每个元素有两种状态：选择(1)或不选择(0)，共  $2^n$  种组合
 *
 * @param nums 输入数组
 * @return 所有可能的子集
 */
vector<vector<int>> subsets2(vector<int>& nums) {
    vector<vector<int>> result;

```

```

// 边界条件检查
if (nums.empty()) {
    return result;
}

int n = nums.size();

// 枚举所有可能的子集 (0 到 2^n-1)
for (int mask = 0; mask < (1 << n); mask++) {
    vector<int> subset;

    // 根据位掩码构造子集
    for (int i = 0; i < n; i++) {
        // 检查第 i 位是否为 1
        if (mask & (1 << i)) {
            subset.push_back(nums[i]);
        }
    }

    result.push_back(subset);
}

return result;
}

/**
 * 解法三：使用迭代法生成所有子集
 * 逐个添加元素，每次添加新元素时，将该元素添加到已有的所有子集中
 *
 * @param nums 输入数组
 * @return 所有可能的子集
 */
vector<vector<int>> subsets3(vector<int>& nums) {
    vector<vector<int>> result;

    // 边界条件检查
    if (nums.empty()) {
        return result;
    }

    // 初始化空集
    result.push_back(vector<int>());

```

```
// 逐个添加元素
for (int num : nums) {
    int size = result.size();

    // 将当前元素添加到已有的所有子集中
    for (int i = 0; i < size; i++) {
        vector<int> newSubset = result[i];
        newSubset.push_back(num);
        result.push_back(newSubset);
    }
}

return result;
}

};

// 测试方法
int main() {
    LeetCode78_Subsets solution;

    // 测试用例 1
    vector<int> nums1 = {1, 2, 3};
    vector<vector<int>> result1 = solution.subsets(nums1);
    printf("输入: nums = [1, 2, 3]\n");
    printf("输出: [");
    for (int i = 0; i < result1.size(); i++) {
        printf("[");
        for (int j = 0; j < result1[i].size(); j++) {
            printf("%d", result1[i][j]);
            if (j < result1[i].size() - 1) printf(",");
        }
        printf("]");
        if (i < result1.size() - 1) printf(",");
    }
    printf("]\n");
}

// 测试用例 2
vector<int> nums2 = {0};
vector<vector<int>> result2 = solution.subsets(nums2);
printf("\n 输入: nums = [0]\n");
printf("输出: [");
for (int i = 0; i < result2.size(); i++) {
```

```

printf("[");
for (int j = 0; j < result2[i].size(); j++) {
    printf("%d", result2[i][j]);
    if (j < result2[i].size() - 1) printf(",");
}
printf("]");
if (i < result2.size() - 1) printf(",");
}

printf("]\n");

// 测试解法二
printf("\n==== 解法二测试 ====\n");
vector<vector<int>> result3 = solution.subsets2(nums1);
printf("输入: nums = [1, 2, 3]\n");
printf("输出: [");
for (int i = 0; i < result3.size(); i++) {
    printf("[");
    for (int j = 0; j < result3[i].size(); j++) {
        printf("%d", result3[i][j]);
        if (j < result3[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result3.size() - 1) printf(",");
}
printf("]\n");

// 测试解法三
printf("\n==== 解法三测试 ====\n");
vector<vector<int>> result4 = solution.subsets3(nums1);
printf("输入: nums = [1, 2, 3]\n");
printf("输出: [");
for (int i = 0; i < result4.size(); i++) {
    printf("[");
    for (int j = 0; j < result4[i].size(); j++) {
        printf("%d", result4[i][j]);
        if (j < result4[i].size() - 1) printf(",");
    }
    printf("]");
    if (i < result4.size() - 1) printf(",");
}
printf("]\n");

return 0;

```

}

=====

文件: LeetCode78\_Subsets. java

```
=====
import java.util.ArrayList;
import java.util.List;

/**
 * LeetCode 78. 子集
 *
 * 题目描述:
 * 给你一个整数数组 nums，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。
 * 解集不能包含重复的子集。你可以按任意顺序返回解集。
 *
 * 示例:
 * 输入: nums = [1,2,3]
 * 输出: [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
 *
 * 输入: nums = [0]
 * 输出: [[], [0]]
 *
 * 提示:
 * 1 <= nums.length <= 10
 * -10 <= nums[i] <= 10
 * nums 中的所有元素互不相同
 *
 * 链接: https://leetcode.cn/problems/subsets/
 *
 * 算法思路:
 * 1. 使用回溯算法生成所有可能的子集
 * 2. 对于每个元素，有两种选择：选择或不选择
 * 3. 通过递归和回溯生成所有满足条件的子集
 * 4. 每一步递归都将当前路径加入结果集
 *
 * 剪枝策略:
 * 1. 可行性剪枝：通过起始索引避免重复选择元素
 * 2. 最优性剪枝：当已选择的元素个数等于数组长度时终止
 * 3. 约束传播：一旦某个元素被选择，后续只能选择后面的元素
 *
 * 时间复杂度: O(n * 2^n)，其中 n 是数组长度，共有  $2^n$  个子集，每个子集需要 O(n) 时间复制
 * 空间复杂度: O(n)，递归栈深度和存储路径的空间
```

```
*  
* 工程化考量:  
* 1. 边界处理: 处理空数组和单元素数组的特殊情况  
* 2. 参数验证: 验证输入参数的有效性  
* 3. 性能优化: 通过剪枝减少不必要的计算  
* 4. 内存管理: 合理使用数据结构减少内存占用  
* 5. 可读性: 添加详细注释和变量命名  
* 6. 异常处理: 处理可能的异常情况  
* 7. 模块化设计: 将核心逻辑封装成独立方法  
* 8. 可维护性: 添加详细注释和文档说明  
*/  
  
public class LeetCode78_Subsets {  
  
    /**  
     * 生成数组的所有子集  
     *  
     * @param nums 输入数组  
     * @return 所有可能的子集  
     */  
    public List<List<Integer>> subsets(int[] nums) {  
        List<List<Integer>> result = new ArrayList<>();  
  
        // 边界条件检查  
        if (nums == null) {  
            return result;  
        }  
  
        List<Integer> path = new ArrayList<>();  
        backtrack(nums, 0, path, result);  
        return result;  
    }  
  
    /**  
     * 回溯函数生成子集  
     *  
     * @param nums 输入数组  
     * @param start 当前起始索引  
     * @param path 当前路径  
     * @param result 结果列表  
     */  
    private void backtrack(int[] nums, int start, List<Integer> path, List<List<Integer>> result)  
    {  
        // 每一步都添加到结果中 (空集也在其中)  
    }
```

```

result.add(new ArrayList<>(path));

// 从 start 开始遍历，避免重复
for (int i = start; i < nums.length; i++) {
    // 选择当前元素
    path.add(nums[i]);

    // 递归处理下一个元素
    backtrack(nums, i + 1, path, result);

    // 回溯：撤销选择
    path.remove(path.size() - 1);
}
}

/***
 * 解法二：使用位运算枚举所有可能的子集
 * 每个元素有两种状态：选择(1)或不选择(0)，共  $2^n$  种组合
 *
 * @param nums 输入数组
 * @return 所有可能的子集
 */
public List<List<Integer>> subsets2(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件检查
    if (nums == null) {
        return result;
    }

    int n = nums.length;

    // 枚举所有可能的子集 (0 到  $2^{n-1}$ )
    for (int mask = 0; mask < (1 << n); mask++) {
        List<Integer> subset = new ArrayList<>();

        // 根据位掩码构造子集
        for (int i = 0; i < n; i++) {
            // 检查第 i 位是否为 1
            if ((mask & (1 << i)) != 0) {
                subset.add(nums[i]);
            }
        }
        result.add(subset);
    }
}

```

```
        result.add(subset);
    }

    return result;
}

/**
 * 解法三：使用迭代法生成所有子集
 * 逐个添加元素，每次添加新元素时，将该元素添加到已有的所有子集中
 *
 * @param nums 输入数组
 * @return 所有可能的子集
 */
public List<List<Integer>> subsets3(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件检查
    if (nums == null) {
        return result;
    }

    // 初始化空集
    result.add(new ArrayList<>());

    // 逐个添加元素
    for (int num : nums) {
        int size = result.size();

        // 将当前元素添加到已有的所有子集中
        for (int i = 0; i < size; i++) {
            List<Integer> newSubset = new ArrayList<>(result.get(i));
            newSubset.add(num);
            result.add(newSubset);
        }
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    LeetCode78_Subsets solution = new LeetCode78_Subsets();
```

```

// 测试用例 1
int[] nums1 = {1, 2, 3};
List<List<Integer>> result1 = solution.subsets(nums1);
System.out.println("输入: nums = [1, 2, 3]");
System.out.println("输出: " + result1);

// 测试用例 2
int[] nums2 = {0};
List<List<Integer>> result2 = solution.subsets(nums2);
System.out.println("\n 输入: nums = [0]");
System.out.println("输出: " + result2);

// 测试解法二
System.out.println("\n==== 解法二测试 ===");
List<List<Integer>> result3 = solution.subsets2(nums1);
System.out.println("输入: nums = [1, 2, 3]");
System.out.println("输出: " + result3);

// 测试解法三
System.out.println("\n==== 解法三测试 ===");
List<List<Integer>> result4 = solution.subsets3(nums1);
System.out.println("输入: nums = [1, 2, 3]");
System.out.println("输出: " + result4);
}

}
=====
```

文件: leetcode78\_subsets.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
```

## LeetCode 78. 子集

**题目描述:**

给你一个整数数组 `nums`，数组中的元素互不相同。返回该数组所有可能的子集（幂集）。  
解集不能包含重复的子集。你可以按任意顺序返回解集。

**示例:**

输入: `nums = [1, 2, 3]`

输出: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

输入: nums = [0]

输出: [[], [0]]

提示:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

nums 中的所有元素互不相同

链接: <https://leetcode.cn/problems/subsets/>

算法思路:

1. 使用回溯算法生成所有可能的子集
2. 对于每个元素，有两种选择：选择或不选择
3. 通过递归和回溯生成所有满足条件的子集
4. 每一步递归都将当前路径加入结果集

剪枝策略:

1. 可行性剪枝：通过起始索引避免重复选择元素
2. 最优性剪枝：当已选择的元素个数等于数组长度时终止
3. 约束传播：一旦某个元素被选择，后续只能选择后面的元素

时间复杂度:  $O(n * 2^n)$ ，其中 n 是数组长度，共有  $2^n$  个子集，每个子集需要  $O(n)$  时间复制

空间复杂度:  $O(n)$ ，递归栈深度和存储路径的空间

工程化考量:

1. 边界处理：处理空数组和单元素数组的特殊情况
2. 参数验证：验证输入参数的有效性
3. 性能优化：通过剪枝减少不必要的计算
4. 内存管理：合理使用数据结构减少内存占用
5. 可读性：添加详细注释和变量命名
6. 异常处理：处理可能的异常情况
7. 模块化设计：将核心逻辑封装成独立方法
8. 可维护性：添加详细注释和文档说明

"""

```
class LeetCode78_Subsets:
```

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

```
        """
```

```
        生成数组的所有子集
```

Args:

nums: 输入数组

Returns:

所有可能的子集

"""

result = []

# 边界条件检查

if nums is None:

    return result

path = []

self.\_backtrack(nums, 0, path, result)

return result

def \_backtrack(self, nums: list[int], start: int, path: list[int], result: list[list[int]])

-> None:

"""

回溯函数生成子集

Args:

nums: 输入数组

start: 当前起始索引

path: 当前路径

result: 结果列表

"""

# 每一步都添加到结果中（空集也在其中）

result.append(path[:])

# 从 start 开始遍历，避免重复

for i in range(start, len(nums)):

    # 选择当前元素

    path.append(nums[i])

    # 递归处理下一个元素

    self.\_backtrack(nums, i + 1, path, result)

    # 回溯：撤销选择

    path.pop()

def subsets2(self, nums: list[int]) -> list[list[int]]:

"""

解法二：使用位运算枚举所有可能的子集

每个元素有两种状态：选择(1)或不选择(0)，共  $2^n$  种组合

Args:

    nums: 输入数组

Returns:

    所有可能的子集

"""

result = []

# 边界条件检查

```
if nums is None:  
    return result
```

n = len(nums)

# 枚举所有可能的子集 (0 到  $2^n - 1$ )

```
for mask in range(1 << n):  
    subset = []
```

# 根据位掩码构造子集

```
for i in range(n):  
    # 检查第 i 位是否为 1  
    if mask & (1 << i):  
        subset.append(nums[i])
```

result.append(subset)

return result

def subsets3(self, nums: list[int]) -> list[list[int]]:

"""

解法三：使用迭代法生成所有子集

逐个添加元素，每次添加新元素时，将该元素添加到已有的所有子集中

Args:

    nums: 输入数组

Returns:

    所有可能的子集

"""

result = []

```
# 边界条件检查
if nums is None:
    return result

# 初始化空集
result.append([])

# 逐个添加元素
for num in nums:
    size = len(result)

    # 将当前元素添加到已有的所有子集中
    for i in range(size):
        new_subset = result[i][:]
        new_subset.append(num)
        result.append(new_subset)

return result
```

```
def subsets4(self, nums: list[int]) -> list[list[int]]:
    """
```

解法四：使用 `itertools.combinations`  
利用 Python 标准库生成所有可能的组合

Args:

nums: 输入数组

Returns:

所有可能的子集

```
from itertools import combinations
result = []
```

```
# 生成所有长度的组合
for i in range(len(nums) + 1):
    for combo in combinations(nums, i):
        result.append(list(combo))

return result
```

```
def main():
```

```
    """测试示例"""

```

```
solution = LeetCode78_Subsets()

# 测试用例 1
nums1 = [1, 2, 3]
result1 = solution.subsets(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result1}")

# 测试用例 2
nums2 = [0]
result2 = solution.subsets(nums2)
print(f"\n 输入: nums = {nums2}")
print(f"输出: {result2}")

# 测试解法二
print("\n== 解法二测试 ==")
result3 = solution.subsets2(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result3}")

# 测试解法三
print("\n== 解法三测试 ==")
result4 = solution.subsets3(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result4}")

# 测试解法四
print("\n== 解法四测试 ==")
result5 = solution.subsets4(nums1)
print(f"输入: nums = {nums1}")
print(f"输出: {result5}")

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

文件: PruningTechniques.java

```
=====
import java.util.*;

/**

```

- \* 剪枝体系 (Pruning Techniques)
- \*
- \* 算法原理:
  - \* 剪枝是一种优化技术，通过提前终止不可能产生最优解的搜索分支来减少搜索空间，
  - \* 从而提高算法效率。剪枝技术广泛应用于回溯算法、博弈树搜索、分支限界等场景。
- \*
- \* 算法特点:
  - \* 1. 减少搜索空间，提高算法效率
  - \* 2. 不影响最终结果的正确性
  - \* 3. 需要设计合适的剪枝条件
  - \* 4. 剪枝效果与问题特性密切相关
- \*
- \* 应用场景:
  - \* - 回溯算法 (N 皇后、数独等)
  - \* - 博弈树搜索 (Alpha-Beta 剪枝)
  - \* - 分支限界算法
  - \* - 组合优化问题
- \*
- \* 剪枝类型:
  - \* 1. 可行性剪枝：提前判断当前分支是否可能产生可行解
  - \* 2. 最优性剪枝：提前判断当前分支是否可能产生更优解
  - \* 3. 记忆化剪枝：避免重复计算相同子问题
  - \* 4. 启发式剪枝：基于启发信息进行剪枝
- \*
- \* 时间复杂度：取决于具体问题和剪枝效果
- \* 空间复杂度：取决于具体实现
- \*/

```
public class PruningTechniques {  
  
    /**  
     * N 皇后问题 - 可行性剪枝示例  
     *  
     * @param n 皇后数量  
     * @return 所有解的数量  
     */  
    public static int solveNQueens(int n) {  
        int[] queens = new int[n]; // queens[i] 表示第 i 行皇后所在的列  
        return backtrack(queens, 0, n);  
    }  
  
    /**  
     * 回溯算法求解 N 皇后问题  
     */
```

```

*
 * @param queens 皇后位置数组
 * @param row 当前行
 * @param n 皇后数量
 * @return 解的数量
*/
private static int backtrack(int[] queens, int row, int n) {
    // 递归终止条件
    if (row == n) {
        return 1; // 找到一个解
    }

    int count = 0;

    // 在当前行尝试每一列
    for (int col = 0; col < n; col++) {
        // 可行性剪枝：检查当前位置是否合法
        if (isValid(queens, row, col)) {
            queens[row] = col; // 放置皇后
            count += backtrack(queens, row + 1, n); // 递归处理下一行
            // 回溯时不需要显式重置，因为下一次循环会覆盖
        }
        // 如果不合法，直接剪枝，不继续递归
    }

    return count;
}

/***
 * 检查在(row, col)位置放置皇后是否合法
 *
 * @param queens 皇后位置数组
 * @param row 行号
 * @param col 列号
 * @return 是否合法
*/
private static boolean isValid(int[] queens, int row, int col) {
    // 检查之前行的皇后是否与当前位置冲突
    for (int i = 0; i < row; i++) {
        // 检查列冲突
        if (queens[i] == col) {
            return false;
        }
    }
}

```

```

// 检查对角线冲突
if (Math.abs(queens[i] - col) == Math.abs(i - row)) {
    return false;
}

}

return true;
}

/***
 * 0-1 背包问题 - 最优性剪枝示例
 */
static class KnapsackItem {
    int weight;
    int value;

    KnapsackItem(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

/***
 * 0-1 背包问题求解（带剪枝优化）
 *
 * @param items 物品列表
 * @param capacity 背包容量
 * @return 最大价值
 */
public static int knapsackWithPruning(List<KnapsackItem> items, int capacity) {
    // 按价值密度排序，用于剪枝
    items.sort((a, b) -> Double.compare(
        (double) b.value / b.weight,
        (double) a.value / a.weight
    ));

    return knapsackBacktrack(items, capacity, 0, 0, 0, 0);
}

/***
 * 0-1 背包回溯算法（带剪枝）
 *

```

```
* @param items 物品列表
* @param capacity 背包容量
* @param currentIndex 当前物品索引
* @param currentWeight 当前重量
* @param currentValue 当前价值
* @param bestValue 当前最优价值
* @return 最大价值
*/
private static int knapsackBacktrack(List<KnapsackItem> items, int capacity,
                                    int currentIndex, int currentWeight,
                                    int currentValue, int bestValue) {
    // 更新最优解
    bestValue = Math.max(bestValue, currentValue);

    // 递归终止条件
    if (currentIndex == items.size()) {
        return bestValue;
    }

    // 最优性剪枝：计算上界
    int upperBound = calculateUpperBound(items, capacity, currentIndex,
  currentWeight, currentValue);

    // 如果上界不大于当前最优值，则剪枝
    if (upperBound <= bestValue) {
        return bestValue;
    }

    KnapsackItem currentItem = items.get(currentIndex);

    // 选择当前物品（可行性剪枝）
    if (currentWeight + currentItem.weight <= capacity) {
        bestValue = knapsackBacktrack(items, capacity, currentIndex + 1,
                                      currentWeight + currentItem.weight,
                                      currentValue + currentItem.value,
                                      bestValue);
    }

    // 不选择当前物品
    bestValue = knapsackBacktrack(items, capacity, currentIndex + 1,
                                  currentWeight, currentValue, bestValue);

    return bestValue;
}
```

```

}

/**
 * 计算 0-1 背包问题的上界（用于最优性剪枝）
 *
 * @param items 物品列表
 * @param capacity 背包容量
 * @param currentIndex 当前物品索引
 * @param currentWeight 当前重量
 * @param currentValue 当前价值
 * @return 上界估计值
 */
private static int calculateUpperBound(List<KnapsackItem> items, int capacity,
                                       int currentIndex, int currentWeight,
                                       int currentValue) {
    int remainingCapacity = capacity - currentWeight;
    int bound = currentValue;

    // 贪心法计算上界：按价值密度选择物品
    for (int i = currentIndex; i < items.size() && remainingCapacity > 0; i++) {
        KnapsackItem item = items.get(i);
        if (item.weight <= remainingCapacity) {
            // 完全装入
            bound += item.value;
            remainingCapacity -= item.weight;
        } else {
            // 部分装入
            bound += (int) ((double) item.value / item.weight * remainingCapacity);
            remainingCapacity = 0;
        }
    }
}

return bound;
}

/**
 * 记忆化斐波那契数列 - 记忆化剪枝示例
 */
static class FibonacciWithMemoization {
    private Map<Integer, Long> memo = new HashMap<>();

    /**
     * 计算第 n 个斐波那契数（带记忆化）
     */

```

```

*
 * @param n 序号
 * @return 第 n 个斐波那契数
 */
public long fibonacci(int n) {
    // 基础情况
    if (n <= 1) {
        return n;
    }

    // 记忆化剪枝：如果已经计算过，直接返回
    if (memo.containsKey(n)) {
        return memo.get(n);
    }

    // 递归计算并存储结果
    long result = fibonacci(n - 1) + fibonacci(n - 2);
    memo.put(n, result);

    return result;
}

/**
 * 清空记忆化缓存
 */
public void clearMemo() {
    memo.clear();
}

/**
 * Alpha-Beta 剪枝 – 博弈树搜索剪枝示例
 */
static class AlphaBetaPruning {
    static final int MAX_DEPTH = 6; // 最大搜索深度
    static final int WIN_SCORE = 10000; // 获胜分数
    static final int LOSE_SCORE = -10000; // 失败分数

    /**
     * Alpha-Beta 剪枝搜索
     *
     * @param board 棋盘状态
     * @param depth 当前深度
    
```

```

* @param alpha Alpha 值
* @param beta Beta 值
* @param isMaximizing 是否最大化玩家
* @return 评估分数
*/
public static int alphaBetaSearch(int[][] board, int depth, int alpha, int beta,
                                  boolean isMaximizing) {
    // 终止条件：达到最大深度或游戏结束
    if (depth == 0 || isGameOver(board)) {
        return evaluateBoard(board);
    }

    if (isMaximizing) {
        int maxEval = Integer.MIN_VALUE;
        List<int[]> moves = generateMoves(board, true);

        for (int[] move : moves) {
            // 执行移动
            makeMove(board, move, true);

            // 递归搜索
            int eval = alphaBetaSearch(board, depth - 1, alpha, beta, false);

            // 撤销移动
            undoMove(board, move);

            maxEval = Math.max(maxEval, eval);
            alpha = Math.max(alpha, eval);
        }
    }

    return maxEval;
} else {
    int minEval = Integer.MAX_VALUE;
    List<int[]> moves = generateMoves(board, false);

    for (int[] move : moves) {
        // 执行移动
        makeMove(board, move, false);
    }
}

```

```
// 递归搜索
int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);

// 撤销移动
undoMove(board, move);

minEval = Math.min(minEval, eval);
beta = Math.min(beta, eval);

// Alpha-Beta 剪枝
if (beta <= alpha) {
    break; // alpha 剪枝
}
}

return minEval;
}

}

/***
 * 检查游戏是否结束
 *
 * @param board 棋盘
 * @return 是否结束
 */
private static boolean isGameOver(int[][] board) {
    // 简化实现，实际游戏中需要根据具体规则判断
    return false;
}

/***
 * 评估棋盘状态
 *
 * @param board 棋盘
 * @return 评估分数
 */
private static int evaluateBoard(int[][] board) {
    // 简化实现，实际评估函数需要根据具体游戏设计
    return 0;
}

/***
```

```
* 生成所有可能的移动
*
* @param board 棋盘
* @param isMaximizing 是否最大化玩家
* @return 移动列表
*/
private static List<int[]> generateMoves(int[][] board, boolean isMaximizing) {
    // 简化实现，实际需要根据具体游戏生成移动
    return new ArrayList<>();
}

/**
* 执行移动
*
* @param board 棋盘
* @param move 移动
* @param isMaximizing 是否最大化玩家
*/
private static void makeMove(int[][] board, int[] move, boolean isMaximizing) {
    // 简化实现，实际需要根据具体游戏执行移动
}

/**
* 撤销移动
*
* @param board 棋盘
* @param move 移动
*/
private static void undoMove(int[][] board, int[] move) {
    // 简化实现，实际需要根据具体游戏撤销移动
}

/**
* 测试示例
*/
public static void main(String[] args) {
    System.out.println("== 剪枝技术测试 ==");
    // 测试 N 皇后问题
    System.out.println("\n1. N 皇后问题剪枝测试:");
    int[] nValues = {4, 8};
    for (int n : nValues) {

```

```

long startTime = System.currentTimeMillis();
int solutions = solveNQueens(n);
long endTime = System.currentTimeMillis();

System.out.printf("%d 皇后问题: %d 个解, 时间: %d ms%n", n, solutions, endTime - startTime);
}

// 测试 0-1 背包问题
System.out.println("\n2. 0-1 背包问题剪枝测试:");
List<KnapsackItem> items = Arrays.asList(
    new KnapsackItem(10, 60),
    new KnapsackItem(20, 100),
    new KnapsackItem(30, 120),
    new KnapsackItem(15, 80),
    new KnapsackItem(25, 90)
);
int capacity = 50;

long startTime = System.currentTimeMillis();
int maxValue = knapsackWithPruning(new ArrayList<>(items), capacity);
long endTime = System.currentTimeMillis();

System.out.printf("背包容量: %d, 最大价值: %d, 时间: %d ms%n",
    capacity, maxValue, endTime - startTime);

// 测试记忆化斐波那契
System.out.println("\n3. 记忆化斐波那契剪枝测试:");
FibonacciWithMemoization fib = new FibonacciWithMemoization();

int[] fibIndices = {30, 35, 40};
for (int n : fibIndices) {
    fib.clearMemo(); // 清空缓存
    long startTime2 = System.currentTimeMillis();
    long result = fib.fibonacci(n);
    long endTime2 = System.currentTimeMillis();

    System.out.printf("F(%d) = %d, 时间: %d ms%n", n, result, endTime2 - startTime2);
}

// 测试 Alpha-Beta 剪枝 (概念演示)
System.out.println("\n4. Alpha-Beta 剪枝概念演示:");
System.out.println("Alpha-Beta 剪枝在博弈树搜索中能有效减少节点访问数量");

```

```
System.out.println("对于深度为 d 的完全二叉树，不剪枝需要访问 O(b^d) 个节点");
System.out.println("使用 Alpha-Beta 剪枝后，最好情况下只需要访问 O(b^(d/2)) 个节点");
}

=====
```

文件: pruning\_techniques.cpp

```
=====
/***
 * 剪枝体系 (Pruning Techniques)
 *
 * 算法原理:
 * 剪枝是一种优化技术，通过提前终止不可能产生最优解的搜索分支来减少搜索空间，
 * 从而提高算法效率。剪枝技术广泛应用于回溯算法、博弈树搜索、分支限界等场景。
 *
 * 算法特点:
 * 1. 减少搜索空间，提高算法效率
 * 2. 不影响最终结果的正确性
 * 3. 需要设计合适的剪枝条件
 * 4. 剪枝效果与问题特性密切相关
 *
 * 应用场景:
 * - 回溯算法 (N 皇后、数独等)
 * - 博弈树搜索 (Alpha-Beta 剪枝)
 * - 分支限界算法
 * - 组合优化问题
 *
 * 剪枝类型:
 * 1. 可行性剪枝: 提前判断当前分支是否可能产生可行解
 * 2. 最优性剪枝: 提前判断当前分支是否可能产生更优解
 * 3. 记忆化剪枝: 避免重复计算相同子问题
 * 4. 启发式剪枝: 基于启发信息进行剪枝
 *
 * 时间复杂度: 取决于具体问题和剪枝效果
 * 空间复杂度: 取决于具体实现
 */

```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <climits>
```

```

#include <chrono>

using namespace std;

class PruningTechniques {
public:
    /**
     * N 皇后问题 - 可行性剪枝示例
     *
     * @param n 皇后数量
     * @return 所有解的数量
     */
    static int solveNQueens(int n) {
        vector<int> queens(n, 0); // queens[i] 表示第 i 行皇后所在的列
        return backtrack(queens, 0, n);
    }

    /**
     * 回溯算法求解 N 皇后问题
     *
     * @param queens 皇后位置数组
     * @param row 当前行
     * @param n 皇后数量
     * @return 解的数量
     */
    static int backtrack(vector<int>& queens, int row, int n) {
        // 递归终止条件
        if (row == n) {
            return 1; // 找到一个解
        }

        int count = 0;

        // 在当前行尝试每一列
        for (int col = 0; col < n; col++) {
            // 可行性剪枝：检查当前位置是否合法
            if (isValid(queens, row, col)) {
                queens[row] = col; // 放置皇后
                count += backtrack(queens, row + 1, n); // 递归处理下一行
                // 回溯时不需要显式重置，因为下一次循环会覆盖
            }
        }

        // 如果不合法，直接剪枝，不继续递归
    }
}

```

```

        return count;
    }

/***
 * 检查在(row, col)位置放置皇后是否合法
 *
 * @param queens 皇后位置数组
 * @param row 行号
 * @param col 列号
 * @return 是否合法
 */
static bool isValid(const vector<int>& queens, int row, int col) {
    // 检查之前行的皇后是否与当前位置冲突
    for (int i = 0; i < row; i++) {
        // 检查列冲突
        if (queens[i] == col) {
            return false;
        }
    }

    // 检查对角线冲突
    if (abs(queens[i] - col) == abs(i - row)) {
        return false;
    }
}

return true;
}

/***
 * 0-1 背包问题相关结构
 */
struct KnapsackItem {
    int weight;
    int value;

    KnapsackItem(int w, int v) : weight(w), value(v) {}

    // 按价值密度排序
    bool operator<(const KnapsackItem& other) const {
        return (double)value / weight > (double)other.value / other.weight;
    }
};

```

```

/**
 * 0-1 背包问题求解（带剪枝优化）
 *
 * @param items 物品列表
 * @param capacity 背包容量
 * @return 最大价值
 */
static int knapsackWithPruning(vector<KnapsackItem> items, int capacity) {
    // 按价值密度排序，用于剪枝
    sort(items.begin(), items.end());

    return knapsackBacktrack(items, capacity, 0, 0, 0, 0);
}

/**
 * 0-1 背包回溯算法（带剪枝）
 *
 * @param items 物品列表
 * @param capacity 背包容量
 * @param currentIndex 当前物品索引
 * @param currentWeight 当前重量
 * @param currentValue 当前价值
 * @param bestValue 当前最优价值
 * @return 最大价值
 */
static int knapsackBacktrack(const vector<KnapsackItem>& items, int capacity,
                             int currentIndex, int currentWeight,
                             int currentValue, int bestValue) {
    // 更新最优解
    bestValue = max(bestValue, currentValue);

    // 递归终止条件
    if (currentIndex == items.size()) {
        return bestValue;
    }

    // 最优性剪枝：计算上界
    int upperBound = calculateUpperBound(items, capacity, currentIndex,
   currentWeight, currentValue);

    // 如果上界不大于当前最优值，则剪枝
    if (upperBound <= bestValue) {

```

```

        return bestValue;
    }

const KnapsackItem& currentItem = items[currentIndex];

// 选择当前物品（可行性剪枝）
if (currentWeight + currentItem.weight <= capacity) {
    bestValue = knapsackBacktrack(items, capacity, currentIndex + 1,
                                   currentWeight + currentItem.weight,
                                   currentValue + currentItem.value,
                                   bestValue);
}

// 不选择当前物品
bestValue = knapsackBacktrack(items, capacity, currentIndex + 1,
                               currentWeight, currentValue, bestValue);

return bestValue;
}

/***
 * 计算 0-1 背包问题的上界（用于最优性剪枝）
 *
 * @param items 物品列表
 * @param capacity 背包容量
 * @param currentIndex 当前物品索引
 * @param currentWeight 当前重量
 * @param currentValue 当前价值
 * @return 上界估计值
 */
static int calculateUpperBound(const vector<KnapsackItem>& items, int capacity,
                               int currentIndex, int currentWeight,
                               int currentValue) {
    int remainingCapacity = capacity - currentWeight;
    int bound = currentValue;

    // 贪心法计算上界：按价值密度选择物品
    for (size_t i = currentIndex; i < items.size() && remainingCapacity > 0; i++) {
        const KnapsackItem& item = items[i];
        if (item.weight <= remainingCapacity) {
            // 完全装入
            bound += item.value;
            remainingCapacity -= item.weight;
        }
    }
}

```

```

        } else {
            // 部分装入
            bound += (int)((double) item.value / item.weight * remainingCapacity);
            remainingCapacity = 0;
        }
    }

    return bound;
}

/***
 * 记忆化斐波那契数列 - 记忆化剪枝示例
 */
class FibonacciWithMemoization {
private:
    map<int, long long> memo;

public:
    /**
     * 计算第 n 个斐波那契数（带记忆化）
     *
     * @param n 序号
     * @return 第 n 个斐波那契数
     */
    long long fibonacci(int n) {
        // 基础情况
        if (n <= 1) {
            return n;
        }

        // 记忆化剪枝：如果已经计算过，直接返回
        if (memo.find(n) != memo.end()) {
            return memo[n];
        }

        // 递归计算并存储结果
        long long result = fibonacci(n - 1) + fibonacci(n - 2);
        memo[n] = result;

        return result;
    }

    /**

```

```

 * 清空记忆化缓存
 */
void clearMemo() {
    memo.clear();
}

};

/***
 * Alpha-Beta 剪枝 - 博弈树搜索剪枝示例
*/
class AlphaBetaPruning {
public:
    static const int MAX_DEPTH = 6; // 最大搜索深度
    static const int WIN_SCORE = 10000; // 获胜分数
    static const int LOSE_SCORE = -10000; // 失败分数

    /**
     * Alpha-Beta 剪枝搜索
     *
     * @param board 棋盘状态
     * @param depth 当前深度
     * @param alpha Alpha 值
     * @param beta Beta 值
     * @param isMaximizing 是否最大化玩家
     * @return 评估分数
     */
    static int alphaBetaSearch(vector<vector<int>>& board, int depth, int alpha, int beta,
                           bool isMaximizing) {
        // 终止条件：达到最大深度或游戏结束
        if (depth == 0 || isGameOver(board)) {
            return evaluateBoard(board);
        }

        if (isMaximizing) {
            int maxEval = INT_MIN;
            vector<vector<int>> moves = generateMoves(board, true);

            for (const auto& move : moves) {
                // 执行移动
                makeMove(board, move, true);

                // 递归搜索
                int eval = alphaBetaSearch(board, depth - 1, alpha, beta, false);
                if (eval > maxEval) {
                    maxEval = eval;
                }
            }
            return maxEval;
        } else {
            int minEval = INT_MAX;
            vector<vector<int>> moves = generateMoves(board, false);

            for (const auto& move : moves) {
                // 执行移动
                makeMove(board, move, false);

                // 递归搜索
                int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);
                if (eval < minEval) {
                    minEval = eval;
                }
            }
            return minEval;
        }
    }
}

```

```

        // 撤销移动
        undoMove(board, move);

        maxEval = max(maxEval, eval);
        alpha = max(alpha, eval);

        // Alpha-Beta 剪枝
        if (beta <= alpha) {
            break; // beta 剪枝
        }
    }

    return maxEval;
} else {
    int minEval = INT_MAX;
    vector<vector<int>> moves = generateMoves(board, false);

    for (const auto& move : moves) {
        // 执行移动
        makeMove(board, move, false);

        // 递归搜索
        int eval = alphaBetaSearch(board, depth - 1, alpha, beta, true);

        // 撤销移动
        undoMove(board, move);

        minEval = min(minEval, eval);
        beta = min(beta, eval);

        // Alpha-Beta 剪枝
        if (beta <= alpha) {
            break; // alpha 剪枝
        }
    }

    return minEval;
}

private:
    /**

```

```
* 检查游戏是否结束
*
* @param board 棋盘
* @return 是否结束
*/
static bool isGameOver(const vector<vector<int>>& board) {
    // 简化实现，实际游戏中需要根据具体规则判断
    return false;
}

/**
* 评估棋盘状态
*
* @param board 棋盘
* @return 评估分数
*/
static int evaluateBoard(const vector<vector<int>>& board) {
    // 简化实现，实际评估函数需要根据具体游戏设计
    return 0;
}

/**
* 生成所有可能的移动
*
* @param board 棋盘
* @param isMaximizing 是否最大化玩家
* @return 移动列表
*/
static vector<vector<int>> generateMoves(const vector<vector<int>>& board, bool
isMaximizing) {
    // 简化实现，实际需要根据具体游戏生成移动
    return vector<vector<int>>();
}

/**
* 执行移动
*
* @param board 棋盘
* @param move 移动
* @param isMaximizing 是否最大化玩家
*/
static void makeMove(vector<vector<int>>& board, const vector<int>& move, bool
isMaximizing) {
```

```

        // 简化实现，实际需要根据具体游戏执行移动
    }

    /**
     * 撤销移动
     *
     * @param board 棋盘
     * @param move 移动
     */
    static void undoMove(vector<vector<int>>& board, const vector<int>& move) {
        // 简化实现，实际需要根据具体游戏撤销移动
    }
};

};

/***
 * 测试示例
 */
int main() {
    cout << "==== 剪枝技术测试 ===" << endl;

    // 测试 N 皇后问题
    cout << "\n1. N 皇后问题剪枝测试:" << endl;
    vector<int> nValues = {4, 8};
    for (int n : nValues) {
        auto startTime = chrono::high_resolution_clock::now();
        int solutions = PruningTechniques::solveNQueens(n);
        auto endTime = chrono::high_resolution_clock::now();

        auto duration = chrono::duration_cast<chrono::microseconds>(endTime - startTime);
        printf("%d 皇后问题: %d 个解, 时间: %ld μs\n", n, solutions, duration.count());
    }

    // 测试 0-1 背包问题
    cout << "\n2. 0-1 背包问题剪枝测试:" << endl;
    vector<PruningTechniques::KnapsackItem> items = {
        PruningTechniques::KnapsackItem(10, 60),
        PruningTechniques::KnapsackItem(20, 100),
        PruningTechniques::KnapsackItem(30, 120),
        PruningTechniques::KnapsackItem(15, 80),
        PruningTechniques::KnapsackItem(25, 90)
    };
    int capacity = 50;
}

```

```

auto startTime = chrono::high_resolution_clock::now();
int maxValue = PruningTechniques::knapsackWithPruning(items, capacity);
auto endTime = chrono::high_resolution_clock::now();

auto duration = chrono::duration_cast<chrono::microseconds>(endTime - startTime);
printf("背包容量: %d, 最大价值: %d, 时间: %ld μs\n", capacity, maxValue, duration.count());

// 测试记忆化斐波那契
cout << "\n3. 记忆化斐波那契剪枝测试:" << endl;
PruningTechniques::FibonacciWithMemoization fib;

vector<int> fibIndices = {30, 35, 40};
for (int n : fibIndices) {
    fib.clearMemo(); // 清空缓存
    auto startTime2 = chrono::high_resolution_clock::now();
    long long result = fib.fibonacci(n);
    auto endTime2 = chrono::high_resolution_clock::now();

    auto duration2 = chrono::duration_cast<chrono::microseconds>(endTime2 - startTime2);
    printf("F(%d) = %lld, 时间: %ld μs\n", n, result, duration2.count());
}

// 测试 Alpha-Beta 剪枝 (概念演示)
cout << "\n4. Alpha-Beta 剪枝概念演示:" << endl;
cout << "Alpha-Beta 剪枝在博弈树搜索中能有效减少节点访问数量" << endl;
cout << "对于深度为 d 的完全二叉树, 不剪枝需要访问 O(b^d) 个节点" << endl;
cout << "使用 Alpha-Beta 剪枝后, 最好情况下只需要访问 O(b^(d/2)) 个节点" << endl;

return 0;
}
=====

文件: pruning_techniques.py
=====

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
剪枝体系 (Pruning Techniques)
"""

算法原理:

```

算法原理:

剪枝是一种优化技术，通过提前终止不可能产生最优解的搜索分支来减少搜索空间，从而提高算法效率。剪枝技术广泛应用于回溯算法、博弈树搜索、分支限界等场景。

算法特点：

1. 减少搜索空间，提高算法效率
2. 不影响最终结果的正确性
3. 需要设计合适的剪枝条件
4. 剪枝效果与问题特性密切相关

应用场景：

- 回溯算法（N皇后、数独等）
- 博弈树搜索（Alpha-Beta 剪枝）
- 分支限界算法
- 组合优化问题

剪枝类型：

1. 可行性剪枝：提前判断当前分支是否可能产生可行解
2. 最优性剪枝：提前判断当前分支是否可能产生更优解
3. 记忆化剪枝：避免重复计算相同子问题
4. 启发式剪枝：基于启发信息进行剪枝

时间复杂度：取决于具体问题和剪枝效果

空间复杂度：取决于具体实现

"""

```
from typing import List, Tuple, Dict
import time
```

```
class PruningTechniques:
```

```
    @staticmethod
```

```
    def solve_n_queens(n: int) -> int:
```

```
        """
```

```
        N皇后问题 - 可行性剪枝示例
```

Args:

n: 皇后数量

Returns:

所有解的数量

```
        """
```

```
        queens = [0] * n # queens[i]表示第 i 行皇后所在的列
        return PruningTechniques._backtrack(queens, 0, n)
```

```
@staticmethod
def _backtrack(queens: List[int], row: int, n: int) -> int:
    """
    回溯算法求解 N 皇后问题

    Args:
        queens: 皇后位置数组
        row: 当前行
        n: 皇后数量

    Returns:
        解的数量
    """
    # 递归终止条件
    if row == n:
        return 1  # 找到一个解

    count = 0

    # 在当前行尝试每一列
    for col in range(n):
        # 可行性剪枝: 检查当前位置是否合法
        if PruningTechniques._is_valid(queens, row, col):
            queens[row] = col  # 放置皇后
            count += PruningTechniques._backtrack(queens, row + 1, n)  # 递归处理下一行
            # 回溯时不需要显式重置, 因为下一次循环会覆盖

    return count

@staticmethod
def _is_valid(queens: List[int], row: int, col: int) -> bool:
    """
    检查在 (row, col) 位置放置皇后是否合法

    Args:
        queens: 皇后位置数组
        row: 行号
        col: 列号

    Returns:
        是否合法
    """

```

```
# 检查之前行的皇后是否与当前位置冲突
for i in range(row):
    # 检查列冲突
    if queens[i] == col:
        return False

    # 检查对角线冲突
    if abs(queens[i] - col) == abs(i - row):
        return False

return True
```

```
@staticmethod
def knapsack_with_pruning(items: List[Tuple[int, int]], capacity: int) -> int:
    """
    0-1 背包问题求解（带剪枝优化）
    
```

Args:

```
    items: 物品列表 [(重量, 价值), ...]
    capacity: 背包容量
```

Returns:

最大价值

"""

# 按价值密度排序，用于剪枝

```
    sorted_items = sorted(items, key=lambda x: x[1] / x[0], reverse=True)
```

```
    return PruningTechniques._knapsack_backtrack(
        sorted_items, capacity, 0, 0, 0
    )
```

```
@staticmethod
def _knapsack_backtrack(items: List[Tuple[int, int]], capacity: int,
                        current_index: int, current_weight: int,
                        current_value: int, best_value: int) -> int:
    """
    0-1 背包回溯算法（带剪枝）
    
```

Args:

```
    items: 物品列表 [(重量, 价值), ...]
    capacity: 背包容量
    current_index: 当前物品索引
    current_weight: 当前重量
```

```
    current_value: 当前价值
    best_value: 当前最优价值

Returns:
    最大价值
"""

# 更新最优解
best_value = max(best_value, current_value)

# 递归终止条件
if current_index == len(items):
    return best_value

# 最优性剪枝: 计算上界
upper_bound = PruningTechniques._calculate_upper_bound(
    items, capacity, current_index, current_weight, current_value
)

# 如果上界不大于当前最优值, 则剪枝
if upper_bound <= best_value:
    return best_value

current_item = items[current_index]

# 选择当前物品 (可行性剪枝)
if current_weight + current_item[0] <= capacity:
    best_value = PruningTechniques._knapsack_backtrack(
        items, capacity, current_index + 1,
        current_weight + current_item[0],
        current_value + current_item[1],
        best_value
    )

# 不选择当前物品
best_value = PruningTechniques._knapsack_backtrack(
    items, capacity, current_index + 1,
    current_weight, current_value, best_value
)

return best_value

@staticmethod
def _calculate_upper_bound(items: List[Tuple[int, int]], capacity: int,
```

```
        current_index: int, current_weight: int,
        current_value: int) -> int:
```

```
"""
```

计算 0-1 背包问题的上界（用于最优性剪枝）

Args:

```
    items: 物品列表 [(重量, 价值), ...]
    capacity: 背包容量
    current_index: 当前物品索引
    current_weight: 当前重量
    current_value: 当前价值
```

Returns:

上界估计值

```
"""
```

```
remaining_capacity = capacity - current_weight
bound = current_value
```

# 贪心法计算上界：按价值密度选择物品

```
for i in range(current_index, len(items)):
    if remaining_capacity <= 0:
        break

    weight, value = items[i]
    if weight <= remaining_capacity:
        # 完全装入
        bound += value
        remaining_capacity -= weight
    else:
        # 部分装入
        bound += int((value / weight) * remaining_capacity)
        remaining_capacity = 0

return int(bound)
```

```
class FibonacciWithMemoization:
```

```
    """记忆化斐波那契数列 - 记忆化剪枝示例"""

```

```
def __init__(self):
    self.memo: Dict[int, int] = {}
```

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

计算第 n 个斐波那契数（带记忆化）

Args:

n: 序号

Returns:

第 n 个斐波那契数

"""

# 基础情况

if n <= 1:

    return n

# 记忆化剪枝：如果已经计算过，直接返回

if n in self.memo:

    return self.memo[n]

# 递归计算并存储结果

result = self.fibonacci(n - 1) + self.fibonacci(n - 2)

self.memo[n] = result

return result

def clear\_memo(self) -> None:

"""清空记忆化缓存"""

self.memo.clear()

class AlphaBetaPruning:

"""Alpha-Beta 剪枝 - 博弈树搜索剪枝示例"""

MAX\_DEPTH = 6 # 最大搜索深度

WIN\_SCORE = 10000 # 获胜分数

LOSE\_SCORE = -10000 # 失败分数

@staticmethod

def alpha\_beta\_search(board: List[List[int]], depth: int, alpha: int, beta: int,

    is\_maximizing: bool) -> int:

"""

Alpha-Beta 剪枝搜索

Args:

board: 棋盘状态

depth: 当前深度

alpha: Alpha 值

beta: Beta 值

is\_maximizing: 是否最大化玩家

Returns:

评估分数

"""

# 终止条件: 达到最大深度或游戏结束

```
if depth == 0 or PruningTechniques.AlphaBetaPruning._is_game_over(board):  
    return PruningTechniques.AlphaBetaPruning._evaluate_board(board)
```

```
if is_maximizing:
```

```
    max_eval = float('-inf')
```

```
    moves = PruningTechniques.AlphaBetaPruning._generate_moves(board, True)
```

```
    for move in moves:
```

```
        # 执行移动
```

```
        PruningTechniques.AlphaBetaPruning._make_move(board, move, True)
```

```
        # 递归搜索
```

```
        eval_score = PruningTechniques.AlphaBetaPruning.alpha_beta_search(
```

```
            board, depth - 1, alpha, beta, False
```

```
)
```

```
        # 撤销移动
```

```
        PruningTechniques.AlphaBetaPruning._undo_move(board, move)
```

```
        max_eval = max(max_eval, eval_score)
```

```
        alpha = max(alpha, eval_score)
```

```
        # Alpha-Beta 剪枝
```

```
        if beta <= alpha:
```

```
            break # beta 剪枝
```

```
    return int(max_eval)
```

```
else:
```

```
    min_eval = float('inf')
```

```
    moves = PruningTechniques.AlphaBetaPruning._generate_moves(board, False)
```

```
    for move in moves:
```

```
        # 执行移动
```

```
        PruningTechniques.AlphaBetaPruning._make_move(board, move, False)
```

```
        # 递归搜索
```

```
    eval_score = PruningTechniques.AlphaBetaPruning.alpha_beta_search(
        board, depth - 1, alpha, beta, True
    )

    # 撤销移动
    PruningTechniques.AlphaBetaPruning._undo_move(board, move)

    min_eval = min(min_eval, eval_score)
    beta = min(beta, eval_score)

    # Alpha-Beta 剪枝
    if beta <= alpha:
        break # alpha 剪枝

return int(min_eval)
```

@staticmethod

```
def _is_game_over(board: List[List[int]]) -> bool:
```

"""

检查游戏是否结束

Args:

board: 棋盘

Returns:

是否结束

"""

# 简化实现，实际游戏中需要根据具体规则判断

return False

@staticmethod

```
def _evaluate_board(board: List[List[int]]) -> int:
```

"""

评估棋盘状态

Args:

board: 棋盘

Returns:

评估分数

"""

# 简化实现，实际评估函数需要根据具体游戏设计

return 0

```
@staticmethod
def _generate_moves(board: List[List[int]], is_maximizing: bool) -> List[List[int]]:
    """

```

生成所有可能的移动

Args:

board: 棋盘

is\_maximizing: 是否最大化玩家

Returns:

移动列表

```
"""

```

```
# 简化实现, 实际需要根据具体游戏生成移动
return []

```

```
@staticmethod
def _make_move(board: List[List[int]], move: List[int], is_maximizing: bool) -> None:
    """

```

执行移动

Args:

board: 棋盘

move: 移动

is\_maximizing: 是否最大化玩家

```
"""

```

```
# 简化实现, 实际需要根据具体游戏执行移动
pass

```

```
@staticmethod
def _undo_move(board: List[List[int]], move: List[int]) -> None:
    """

```

撤销移动

Args:

board: 棋盘

move: 移动

```
"""

```

```
# 简化实现, 实际需要根据具体游戏撤销移动
pass

```

```
def main():

```

```

"""测试示例"""
print("== 剪枝技术测试 ==")

# 测试 N 皇后问题
print("\n1. N 皇后问题剪枝测试:")
n_values = [4, 8]
for n in n_values:
    start_time = time.time()
    solutions = PruningTechniques.solve_n_queens(n)
    end_time = time.time()

    print(f"n {n} 皇后问题: {solutions} 个解, 时间: {(end_time - start_time) * 1000:.2f} ms")

# 测试 0-1 背包问题
print("\n2. 0-1 背包问题剪枝测试:")
items = [(10, 60), (20, 100), (30, 120), (15, 80), (25, 90)]
capacity = 50

start_time = time.time()
max_value = PruningTechniques.knapsack_with_pruning(items, capacity)
end_time = time.time()

print(f"背包容量: {capacity}, 最大价值: {max_value}, 时间: {(end_time - start_time) * 1000:.2f} ms")

# 测试记忆化斐波那契
print("\n3. 记忆化斐波那契剪枝测试:")
fib = PruningTechniques.FibonacciWithMemoization()

fib_indices = [30, 35, 40]
for n in fib_indices:
    fib.clear_memo() # 清空缓存
    start_time2 = time.time()
    result = fib.fibonacci(n)
    end_time2 = time.time()

    print(f"F({n}) = {result}, 时间: {(end_time2 - start_time2) * 1000:.2f} ms")

# 测试 Alpha-Beta 剪枝 (概念演示)
print("\n4. Alpha-Beta 剪枝概念演示:")
print("Alpha-Beta 剪枝在博弈树搜索中能有效减少节点访问数量")
print("对于深度为 d 的完全二叉树, 不剪枝需要访问 O(b^d) 个节点")
print("使用 Alpha-Beta 剪枝后, 最好情况下只需要访问 O(b^(d/2)) 个节点")

```

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