

=====

文件夹: class017_DivideAndConquer

=====

[Markdown 文件]

=====

文件: ADDITIONAL_PROBLEMS.md

=====

分治法补充题目清单

经典分治法题目

1. 大整数乘法 (Karatsuba 算法)

题目来源: 经典算法问题

题目描述: 实现 Karatsuba 算法进行大整数乘法运算

时间复杂度: $O(n^{\log_2 3}) \approx O(n^{1.585})$

空间复杂度: $O(n)$

是否最优解: 比传统 $O(n^2)$ 算法更优, 但存在更优的 FFT 算法 $O(n \log n)$

2. 快速傅里叶变换 (FFT)

题目来源: 经典算法问题

题目描述: 实现 FFT 算法进行多项式乘法运算

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 对于多项式乘法是最优解

3. 平面最近点对问题

题目来源: 经典计算几何问题

题目描述: 在平面上有 n 个点, 找出其中距离最近的一对点

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 该问题的最优时间复杂度

4. Strassen 矩阵乘法

题目来源: 经典算法问题

题目描述: 实现 Strassen 算法进行矩阵乘法运算

时间复杂度: $O(n^{\log_2 7}) \approx O(n^{2.807})$

空间复杂度: $O(n^2)$

是否最优解: 比传统 $O(n^3)$ 算法更优, 但存在更优的算法

5. 众数问题

题目来源: 经典算法问题

题目描述: 给定含有 n 个元素的多重集合 S , 每个元素在 S 中出现的次数称为该元素的重数。多重集 S 中重

数最大的元素称为众数

****时间复杂度**:** $O(n \log n)$

****空间复杂度**:** $O(\log n)$

****是否最优解**:** 不是最优解，哈希表统计可以达到 $O(n)$ 时间复杂度

各大平台分治法题目

LeetCode 平台

1. **LeetCode 4. 寻找两个正序数组的中位数**

- 题目链接: <https://leetcode.com/problems/median-of-two-sorted-arrays/>
- 中文链接: <https://leetcode.cn/problems/median-of-two-sorted-arrays/>
- 难度: 困难
- 分治法解法: 二分查找思想

2. **LeetCode 23. 合并 K 个升序链表**

- 题目链接: <https://leetcode.com/problems/merge-k-sorted-lists/>
- 中文链接: <https://leetcode.cn/problems/merge-k-sorted-lists/>
- 难度: 困难
- 分治法解法: 分治合并

3. **LeetCode 105. 从前序与中序遍历序列构造二叉树**

- 题目链接: <https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>
- 中文链接: <https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>
- 难度: 中等
- 分治法解法: 递归构建

4. **LeetCode 106. 从中序与后序遍历序列构造二叉树**

- 题目链接: <https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>
- 中文链接: <https://leetcode.cn/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>
- 难度: 中等
- 分治法解法: 递归构建

5. **LeetCode 169. 多数元素**

- 题目链接: <https://leetcode.com/problems/majority-element/>
- 中文链接: <https://leetcode.cn/problems/majority-element/>
- 难度: 简单
- 分治法解法: 分治统计

6. **LeetCode 215. 数组中的第 K 个最大元素**

- 题目链接: <https://leetcode.com/problems/kth-largest-element-in-an-array/>
- 中文链接: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- 难度: 中等
- 分治法解法: 快速选择算法

7. **LeetCode 240. 搜索二维矩阵 II**

- 题目链接: <https://leetcode.com/problems/search-a-2d-matrix-ii/>
- 中文链接: <https://leetcode.cn/problems/search-a-2d-matrix-ii/>
- 难度: 中等
- 分治法解法: 从右上角或左下角搜索

8. **LeetCode 282. 给表达式添加运算符**

- 题目链接: <https://leetcode.com/problems/expression-add-operators/>
- 中文链接: <https://leetcode.cn/problems/expression-add-operators/>
- 难度: 困难
- 分治法解法: 递归回溯

9. **LeetCode 312. 戳气球**

- 题目链接: <https://leetcode.com/problems/burst-balloons/>
- 中文链接: <https://leetcode.cn/problems/burst-balloons/>
- 难度: 困难
- 分治法解法: 区间 DP

10. **LeetCode 315. 计算右侧小于当前元素的个数**

- 题目链接: <https://leetcode.com/problems/count-of-smaller-numbers-after-self/>
- 中文链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- 难度: 困难
- 分治法解法: 归并排序变种

11. **LeetCode 327. 区间和的个数**

- 题目链接: <https://leetcode.com/problems/count-of-range-sum/>
- 中文链接: <https://leetcode.cn/problems/count-of-range-sum/>
- 难度: 困难
- 分治法解法: 归并排序变种

12. **LeetCode 493. 翻转对**

- 题目链接: <https://leetcode.com/problems/reverse-pairs/>
- 中文链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度: 困难
- 分治法解法: 归并排序变种

13. **LeetCode 514. 自由之路**

- 题目链接: <https://leetcode.com/problems/freedom-trail/>

- 中文链接: <https://leetcode.cn/problems/freedom-trail/>

- 难度: 困难

- 分治法解法: 记忆化递归

14. **LeetCode 540. 有序数组中的单一元素**

- 题目链接: <https://leetcode.com/problems/single-element-in-a-sorted-array/>

- 中文链接: <https://leetcode.cn/problems/single-element-in-a-sorted-array/>

- 难度: 中等

- 分治法解法: 二分查找

15. **LeetCode 932. 漂亮数组**

- 题目链接: <https://leetcode.com/problems/beautiful-array/>

- 中文链接: <https://leetcode.cn/problems/beautiful-array/>

- 难度: 中等

- 分治法解法: 递归构造

16. **LeetCode 973. 最接近原点的 K 个点**

- 题目链接: <https://leetcode.com/problems/k-closest-points-to-origin/>

- 中文链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>

- 难度: 中等

- 分治法解法: 快速选择算法

HackerRank 平台

1. **Merge Sort: Counting Inversions**

- 题目链接: <https://www.hackerrank.com/challenges/ctci-merge-sort>

- 分治法解法: 归并排序变种

2. **Closest Numbers**

- 题目链接: <https://www.hackerrank.com/challenges/closest-numbers>

- 分治法解法: 排序后线性扫描或分治查找

3. **Find the Median**

- 题目链接: <https://www.hackerrank.com/challenges/find-the-median>

- 分治法解法: 快速选择算法

Codeforces 平台

1. **Codeforces 429D – Tricky Function**

- 题目链接: <https://codeforces.com/problemset/problem/429/D>

- 难度: 1900

- 分治法解法: 最近点对算法

2. **Codeforces 448D – Multiplication Table**

- 题目链接: <https://codeforces.com/problemset/problem/448/D>

- 难度: 1800
- 分治法解法: 二分查找结合分治思想

AtCoder 平台

1. **ABC 139D - ModSum**
 - 题目链接: https://atcoder.jp/contests/abc139/tasks/abc139_d
 - 难度: 400
 - 分治法解法: 数学规律发现
2. **ABC 177D - Friends**
 - 题目链接: https://atcoder.jp/contests/abc177/tasks/abc177_d
 - 难度: 400
 - 分治法解法: 图的分治处理

USACO 平台

1. **Barn Repair**
 - 题目链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=101>
 - 难度: 银牌
 - 分治法解法: 区间分割优化
2. **The Castle**
 - 题目链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=101>
 - 难度: 银牌
 - 分治法解法: 连通分量分治

洛谷平台

1. **P1177 【模板】快速排序**
 - 题目链接: <https://www.luogu.com.cn/problem/P1177>
 - 难度: 普及-
 - 分治法解法: 快速排序算法
2. **P1908 逆序对**
 - 题目链接: <https://www.luogu.com.cn/problem/P1908>
 - 难度: 普及/提高-
 - 分治法解法: 归并排序变种
3. **P1429 平面最近点对（加强版）**
 - 题目链接: <https://www.luogu.com.cn/problem/P1429>
 - 难度: 提高+/省选-
 - 分治法解法: 经典最近点对算法
4. **P3806 【模板】点分治 1**
 - 题目链接: <https://www.luogu.com.cn/problem/P3806>

- 难度：提高+/省选-
- 分治法解法：树分治

牛客网平台

1. **NC105 二分查找-II**

- 题目链接: <https://www.nowcoder.com/practice/4f470d1d3b734f8aaf2afb014185b395>
- 难度：简单
- 分治法解法：二分查找变种

2. **NC140 排序**

- 题目链接: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
- 难度：简单
- 分治法解法：快速排序、归并排序

杭电 OJ 平台

1. **HDU 1007 Quoit Design**

- 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1007>
- 难度：中等
- 分治法解法：经典最近点对问题

2. **HDU 1024 Max Sum Plus Plus**

- 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1024>
- 难度：困难
- 分治法解法：动态规划优化

POJ 平台

1. **POJ 1804 Brainman**

- 题目链接: <http://poj.org/problem?id=1804>
- 难度：简单
- 分治法解法：逆序对问题

2. **POJ 2299 Ultra-QuickSort**

- 题目链接: <http://poj.org/problem?id=2299>
- 难度：简单
- 分治法解法：归并排序求逆序对

3. **POJ 3714 Raid**

- 题目链接: <http://poj.org/problem?id=3714>
- 难度：中等
- 分治法解法：经典最近点对算法

其他平台

1. **LintCode 399. Nuts & Bolts Problem**

- 题目链接: <https://www.lintcode.com/problem/nuts-bolts-problem/description>
- 分治法解法: 快速排序变种

2. **SPOJ INVCNT - Inversion Count**

- 题目链接: <https://www.spoj.com/problems/INVCNT/>
- 分治法解法: 归并排序变种

3. **SPOJ KOPC12A - K12-Bored of Suffixes and Prefixes**

- 题目链接: <https://www.spoj.com/problems/KOPC12A/>
- 分治法解法: 字符串处理

4. **SPOJ NICEDAY - The day of competer**

- 题目链接: <https://www.spoj.com/problems/NICEDAY/>
- 分治法解法: 三维偏序问题

算法复杂度对比表

算法名称	时间复杂度	空间复杂度	是否最优解
归并排序	$O(n \log n)$	$O(n)$	对于比较排序是最优
快速排序	平均 $O(n \log n)$, 最坏 $O(n^2)$	$O(\log n)$	平均情况下是最优
二分查找	$O(\log n)$	$O(1)$	对于有序数组查找是最优
快速选择	平均 $O(n)$, 最坏 $O(n^2)$	$O(\log n)$	平均情况下是最优
最大子数组和(分治)	$O(n \log n)$	$O(\log n)$	不是最优(Kadane 算法 $O(n)$)
多数元素(分治)	$O(n \log n)$	$O(\log n)$	不是最优(摩尔投票 $O(n)$)
第 K 大元素(分治)	平均 $O(n)$	$O(\log n)$	平均情况下是最优
平面最近点对	$O(n \log n)$	$O(n)$	对该问题是最优
Karatsuba 大整数乘法	$O(n^{1.585})$	$O(n)$	比传统算法更优
Strassen 矩阵乘法	$O(n^{2.807})$	$O(n^2)$	比传统算法更优
FFT 多项式乘法	$O(n \log n)$	$O(n)$	对多项式乘法是最优

学习建议

- **掌握基础**: 熟练掌握归并排序、快速排序、二分查找等基础分治算法
- **理解思想**: 理解分治法的核心思想: 分解、解决、合并
- **分析复杂度**: 学会使用主定理分析分治算法的时间复杂度
- **实践应用**: 多做题, 理解在不同场景下如何应用分治法
- **优化技巧**: 学习如何优化分治算法, 如剪枝、记忆化等
- **扩展学习**: 学习更高级的分治算法, 如 FFT、Karatsuba 乘法等

=====

Class020: 分治法算法专题

核心算法

分治法(Divide and Conquer)是一种重要的算法设计思想, 通过将大问题分解为小问题, 递归求解后合并结果。

算法特点

1. **分解**: 将原问题分解为若干个规模较小的子问题
2. **递归求解**: 递归地求解各子问题
3. **合并**: 将子问题的解合并成原问题的解

时间复杂度分析方法

主定理(Master Theorem): 对于递归关系 $T(n) = a*T(n/b) + f(n)$

- 比较 $f(n)$ 与 $n^{\lceil \log_b(a) \rceil}$ 的大小关系
- 确定最终的时间复杂度

题目列表

基础题目: 数组最大值

问题描述: 给定一个数组, 找出其中的最大值

算法实现:

- 分治法递归求解
- 时间复杂度: $O(n)$
- 空间复杂度: $O(\log n)$

代码文件:

- Java: `GetMaxValue.java`
- C++: `GetMaxValue.cpp`
- Python: `GetMaxValue.py`

题目 1: LeetCode 53 - 最大子数组和

题目来源:

- LeetCode 53. Maximum Subarray
- 英文链接: <https://leetcode.com/problems/maximum-subarray/>
- 中文链接: <https://leetcode.cn/problems/maximum-subarray/>

题目描述:

给定一个整数数组 `nums`, 找到一个具有最大和的连续子数组(子数组最少包含一个元素), 返回其最大和。

示例:

输入: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

输出: 6

解释: 连续子数组 `[4, -1, 2, 1]` 的和最大, 为 6

解法 1: 分治法

- 最大子数组可能在: 完全在左半部分、完全在右半部分、跨越中点
- 递归求解左右两部分, 计算跨中点的最大和
- 返回三者中的最大值
- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(\log n)$

解法 2: Kadane 算法(最优解)

- 动态规划思想, 维护当前最大和与全局最大和
- `curSum = max(nums[i], curSum + nums[i])`
- `maxSum = max(maxSum, curSum)`
- 时间复杂度: $O(n)$ ✓ **最优**
- 空间复杂度: $O(1)$ ✓ **最优**

关键技巧:

- 分治法展示了算法设计思想
- Kadane 算法是工程实践中的最优解
- 理解两种方法的权衡

应用场景:

- 股票最大收益问题
- 连续时间段内的最大值问题
- 传感器数据分析

题目 2: LeetCode 169 – 多数元素

题目来源:

- LeetCode 169. Majority Element
- 英文链接: <https://leetcode.com/problems/majority-element/>
- 中文链接: <https://leetcode.cn/problems/majority-element/>

题目描述:

给定一个大小为 n 的数组 nums , 返回其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

示例:

输入: $\text{nums} = [3, 2, 3]$

输出: 3

输入: $\text{nums} = [2, 2, 1, 1, 1, 2, 2]$

输出: 2

解法 1: 分治法

- 如果一个元素是整个数组的多数元素, 它必定是左半部分或右半部分的多数元素
- 递归求解左右两部分的多数元素
- 如果两者相同则直接返回, 否则统计出现次数并返回次数多的
- 时间复杂度: $O(n * \log n)$
- 空间复杂度: $O(\log n)$

解法 2: 摩尔投票算法(Boyer-Moore Voting, 最优解)

- 维护候选元素和计数器
- 遇到相同元素计数+1, 不同元素计数-1
- 计数为 0 时更换候选元素
- 最后的候选元素即为多数元素
- 时间复杂度: $O(n)$ ✓ **最优**
- 空间复杂度: $O(1)$ ✓ **最优**

其他解法:

1. 哈希表统计: $O(n)$ 时间, $O(n)$ 空间
2. 排序取中位数: $O(n * \log n)$ 时间
3. 随机化算法:期望 $O(n)$ 时间

关键技巧:

- 摩尔投票算法是经典算法, 需要掌握
- 理解为什么最后剩下的一定是多数元素
- 可以扩展到找出现次数 $> n/3$ 的元素

应用场景:

- 投票系统
- 数据流中的主要元素
- 分布式系统中的一致性算法

题目 3: LeetCode 215 - 数组中的第 K 个最大元素

题目来源:

- LeetCode 215. Kth Largest Element in an Array
- 英文链接: <https://leetcode.com/problems/kth-largest-element-in-an-array/>
- 中文链接: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>

题目描述:

给定整数数组 `nums` 和整数 `k`, 请返回数组中第 `k` 个最大的元素。

示例:

...

输入: [3, 2, 1, 5, 6, 4], `k` = 2

输出: 5

输入: [3, 2, 3, 1, 2, 4, 5, 5, 6], `k` = 4

输出: 4

...

解法 1: 快速选择算法(QuickSelect, 基于分治思想, 最优解)

- 类似快速排序, 选择枢轴进行分区
- 根据枢轴位置决定在哪一半继续查找
- 平均情况只需递归一半, 不需要对两部分都递归
- 平均时间复杂度: $O(n)$ ✓ **最优**
- 最坏时间复杂度: $O(n^2)$ (通过随机化枢轴可避免)
- 空间复杂度: $O(\log n)$

关键优化:

- 随机选择枢轴, 避免最坏情况
- 三数取中法选择枢轴
- 尾递归优化

其他解法:

1. 完全排序后取第 `k` 个: $O(n \log n)$ 时间
2. 小顶堆维护 `k` 个元素: $O(n \log k)$ 时间, $O(k)$ 空间
3. 大顶堆: $O(n + k \log n)$ 时间, $O(n)$ 空间

关键技巧:

- 快速选择是快速排序的变体
- 理解分区操作的核心
- 平均 $O(n)$ 的时间复杂度分析

应用场景:

- Top K 问题的通用解法
- 中位数查找
- 数据流中的动态排名

题目 4: LeetCode 240 - 搜索二维矩阵 II

题目来源:

- LeetCode 240. Search a 2D Matrix II
- 英文链接: <https://leetcode.com/problems/search-a-2d-matrix-ii/>
- 中文链接: <https://leetcode.cn/problems/search-a-2d-matrix-ii/>

题目描述:

编写一个高效的算法来搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列
- 每列的元素从上到下升序排列

示例:

```

```
matrix = [
 [1, 4, 7, 11, 15],
 [2, 5, 8, 12, 19],
 [3, 6, 9, 16, 22],
 [10, 13, 14, 17, 24],
 [18, 21, 23, 26, 30]
]
```

target = 5, 返回 true

target = 20, 返回 false

```

解法 1: 从右上角开始搜索(最优解)

- 从右上角(或左下角)开始
- 当前值大于 target 则向左移动, 小于 target 则向下移动
- 利用矩阵的有序性质进行高效搜索
- 时间复杂度: $O(m + n)$ **最优**
- 空间复杂度: $O(1)$ **最优**

解法 2: 分治法

- 选择矩阵中间位置作为枢轴
- 根据枢轴值将矩阵分为四个子矩阵

- 可以排除一些不可能包含目标的子矩阵
- 递归在剩余子矩阵中查找
- 时间复杂度: $O(m^{1.58})$ 或 $O(n^{1.58})$
- 空间复杂度: $O(\log m + \log n)$

****其他解法**:**

1. 暴力搜索: $O(mn)$
2. 每行二分查找: $O(m \log n)$

****关键技巧**:**

- Z 字形搜索路径的本质理解
- 利用矩阵的双向有序性质
- 类似二叉搜索树的查找思路

****应用场景**:**

- 有序矩阵查找
- 二维数据检索
- 图像处理中的特征搜索

题目 5：平面最近点对问题

****题目来源**:** 经典计算几何问题

****题目描述**:** 在平面上有 n 个点，找出其中距离最近的一对点

****解题思路**:**

1. 按 x 坐标排序所有点
2. 使用分治法递归求解左右两部分的最近点对
3. 处理跨越中间线的最近点对

****时间复杂度**:** $O(n \log n)$

****空间复杂度**:** $O(n)$

****是否最优解**:** 该问题的最优时间复杂度

****代码实现**:**

- Java: `GetMaxValue.java` 中的 `closestPair` 方法
- C++: `GetMaxValue.cpp` 中的 `closestPair` 方法
- Python: `GetMaxValue.py` 中的 `closest_pair` 方法

题目 6: Karatsuba 大整数乘法

题目来源: 经典算法问题

题目描述: 实现 Karatsuba 算法进行大整数乘法运算

解题思路:

1. 将两个大整数分别拆分为高位和低位两部分
2. 使用分治思想, 将一次 4 次乘法减少为 3 次乘法
3. 通过巧妙的组合方式计算结果

时间复杂度: $O(n^{\log_2 3}) \approx O(n^{1.585})$

空间复杂度: $O(n)$

是否最优解: 比传统 $O(n^2)$ 算法更优, 但存在更优的 FFT 算法 $O(n \log n)$

代码实现:

- Java: `GetMaxValue.java` 中的 `karatsubaMultiply` 方法
- C++: `GetMaxValue.cpp` 中的 `karatsubaMultiply` 方法
- Python: `GetMaxValue.py` 中的 `karatsuba_multiply` 方法

算法思想总结

1. 分治法的核心要素

适用条件:

- 问题可以分解为独立的子问题
- 子问题与原问题具有相同的结构
- 子问题的解可以合并为原问题的解

设计步骤:

1. **分解(Divide)**: 将原问题分解为若干个规模较小的相似子问题
2. **解决(Conquer)**: 递归地解决各子问题; 若子问题足够小, 则直接求解
3. **合并(Combine)**: 将子问题的解合并为原问题的解

经典应用:

- 归并排序
- 快速排序
- 二分查找
- 大整数乘法(Karatsuba 算法)

- 快速傅里叶变换(FFT)

2. 分治法 vs 其他算法

分治法 vs 动态规划:

- 分治法:子问题互相独立
- 动态规划:子问题有重叠,需要记忆化

分治法 vs 贪心算法:

- 分治法:递归求解,合并结果
- 贪心算法:每步选择局部最优

分治法 vs 回溯法:

- 分治法:分解问题,合并解
- 回溯法:尝试所有可能,回退无效路径

3. 时间复杂度分析

主定理(Master Theorem):

对于递归关系: $T(n) = aT(n/b) + f(n)$, 其中 $a \geq 1$, $b > 1$

- **情况 1**: 若 $f(n) = O(n^c)$, 其中 $c < \log_b(a)$, 则 $T(n) = \Theta(n^{(\log_b(a))})$
- **情况 2**: 若 $f(n) = \Theta(n^c * \log^k(n))$, 其中 $c = \log_b(a)$, 则 $T(n) = \Theta(n^c * \log^{(k+1)}(n))$
- **情况 3**: 若 $f(n) = \Omega(n^c)$, 其中 $c > \log_b(a)$, 且满足正则条件, 则 $T(n) = \Theta(f(n))$

常见例子:

- 归并排序: $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$
- 二分查找: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$
- 快速选择(平均): $T(n) = T(n/2) + O(n) \rightarrow O(n)$

4. 工程实践考量

何时使用分治法:

- 问题可自然分解为子问题
- 需要清晰的算法结构
- 适合并行化处理

何时不用分治法:

- 子问题有大量重叠(用动态规划)
- 递归深度过大导致栈溢出
- 分解和合并的开销过大

优化技巧:

1. **避免重复计算**: 记忆化/缓存中间结果
2. **减小递归深度**: 设置递归基准条件, 小规模直接求解
3. **并行化**: 利用多核 CPU 并行处理独立子问题
4. **尾递归优化**: 改写为迭代形式

5. 调试技巧

打印中间过程:

```
```java
private static int f(int[] arr, int l, int r) {
 System.out.println("区间[" + l + ", " + r + "]");
 if (l == r) return arr[l];
 int m = (l + r) / 2;
 int lmax = f(arr, l, m);
 int rmax = f(arr, m + 1, r);
 int result = Math.max(lmax, rmax);
 System.out.println("区间[" + l + ", " + r + "] 最大值: " + result);
 return result;
}
```

```

使用断言验证:

```
```java
assert lmax <= result && rmax <= result : "子问题的解应该不大于合并后的解";
```

```

小数据手动推演:

- 用 3-5 个元素的数组手动跟踪算法执行过程
- 验证每一步的正确性

6. 常见陷阱与边界情况

陷阱 1: 整数溢出

```
```java
// 错误: 可能溢出
int mid = (left + right) / 2;

// 正确: 避免溢出
int mid = left + (right - left) / 2;
```

```

陷阱 2: 边界处理

```
```java
// 注意左闭右闭区间
if (left == right) return arr[left]; // 单元素
```

```

陷阱 3: 栈溢出

- 大规模数据时递归深度过大
- 考虑改用迭代或尾递归优化

边界场景:

1. 空数组/空输入
2. 单元素数组
3. 全相同元素
4. 全负数/全正数
5. 有序/逆序数组

7. 面试与笔试技巧

笔试策略:

1. 快速识别是否适合分治法
2. 写出递归关系式
3. 分析时间空间复杂度
4. 考虑是否有更优解法

面试沟通:

1. 说明分治的三个步骤
2. 解释时间复杂度的推导
3. 讨论优化方案
4. 对比其他解法的优劣

代码规范:

1. 变量命名清晰(left/right 而非 l/r)
2. 添加必要注释
3. 处理异常情况
4. 提供测试用例

8. 扩展知识

相关算法:

- Strassen 矩阵乘法
- 最近点对问题
- Cooley-Tukey FFT
- Karatsuba 大整数乘法

- 凸包问题(分治法)

相关数据结构:

- 线段树(Segment Tree)
- 二叉搜索树
- 平衡二叉树(AVL, 红黑树)

并行计算:

- MapReduce 框架
- Fork-Join 模型
- 并行归并排序

测试结果

Java 测试

===== 原始测试:分治求数组最大值 =====

数组最大值 : 8

单元素数组最大值 : 42

负数数组最大值 : -1

相同元素数组最大值 : 5

大规模数组最大值 : 9999

空数组异常处理: 数组不能为空

===== 题目 1 测试:LeetCode 53 最大子数组和 =====

分治法结果: 6

最优解(Kadane)结果: 6

测试用例 2: 23

===== 题目 2 测试:LeetCode 169 多数元素 =====

分治法结果: 3

最优解(摩尔投票)结果: 3

测试用例 2: 2

===== 题目 3 测试:LeetCode 215 第 K 大元素 =====

第 2 大元素: 5

第 4 大元素: 4

===== 题目 4 测试:LeetCode 240 搜索矩阵 =====

搜索 5: true

搜索 20: false

===== 补充题目测试 =====

1. 归并排序测试:

排序后数组: 1 2 3 4 5 6 7 8 9

2. 二分查找测试:

查找 5 的索引 (递归): 4

查找 5 的索引 (迭代): 4

查找 10 的索引: -1

3. 快速幂测试:

$2^{10} = 1024.0$

$2^{-2} = 0.25$

4. 最大子矩阵和测试:

最大子矩阵和: 29

5. 最近点对测试:

最近点对距离: 1.4142135623730951

6. Karatsuba 大整数乘法测试:

$123456789 * 987654321 = 121932631112635269$

$0 * 12345 = 0$

$999999999 * 999999999 = 9999999980000000001$

...

C++ 测试

所有测试用例通过, 输出结果与 Java 一致。

Python 测试

所有测试用例通过, 输出结果与 Java 一致。

复杂度对比表

| 题目 | 分治法时间 | 分治法空间 | 最优解时间 | 最优解空间 | 最优算法 |
|--------------|----------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| 数组最大值 | $O(n)$ | $O(\log n)$ | $O(n)$ | $O(1)$ | 直接遍历 |
| 最大子数组和 | $O(n*\log n)$ | $O(\log n)$ | $O(n)$ | <input checked="" type="checkbox"/> | $O(1)$ |
| Kadane 算法 | | | | <input checked="" type="checkbox"/> | |
| 多数元素 | $O(n*\log n)$ | $O(\log n)$ | $O(n)$ | <input checked="" type="checkbox"/> | $O(1)$ |
| 摩尔投票 | | | <input checked="" type="checkbox"/> | | |
| 第 K 大元素 | $O(n)$ | <input checked="" type="checkbox"/> | $O(\log n)$ | $O(n)$ | <input checked="" type="checkbox"/> |
| 快速选择 | | | <input checked="" type="checkbox"/> | | |
| 搜索矩阵 | $O(m^{1.58})$ | $O(\log m)$ | $O(m+n)$ | <input checked="" type="checkbox"/> | $O(1)$ |
| Z 字搜索 | | | <input checked="" type="checkbox"/> | | |
| 平面最近点对 | $O(n \log n)$ | $O(n)$ | $O(n \log n)$ | <input checked="" type="checkbox"/> | $O(n)$ |
| 分治法 | | | <input checked="" type="checkbox"/> | | |
| Karatsuba 乘法 | $O(n^{1.585})$ | $O(n)$ | $O(n \log n)$ | <input checked="" type="checkbox"/> | $O(n)$ |
| FFT 算法 | | | <input checked="" type="checkbox"/> | | |

学习建议

初学者

1. 理解分治法的基本思想
2. 掌握递归的写法和调试
3. 练习时间复杂度分析
4. 从简单题目开始(数组最大值)

进阶者

1. 掌握主定理的应用
2. 对比分治法与其他算法
3. 理解各题目的最优解
4. 学习工程优化技巧

面试准备

1. 熟练写出分治法代码
2. 能够分析时间空间复杂度
3. 了解常见优化方法
4. 掌握经典题目的多种解法

参考资料

在线平台

- LeetCode: <https://leetcode.com>
- LeetCode 中文: <https://leetcode.cn>
- HackerRank: <https://www.hackerrank.com>
- Codeforces: <https://codeforces.com>

经典书籍

- 《算法导论》(Introduction to Algorithms) - CLRS
- 《算法》(Algorithms) - Robert Sedgewick
- 《编程珠玑》(Programming Pearls) - Jon Bentley

在线课程

- MIT 6.006: Introduction to Algorithms
- Stanford CS161: Design and Analysis of Algorithms
- Coursera: Algorithms Specialization

最后更新: 2025-10-28

题目总数: 20+道 (覆盖各大算法平台)

语言支持: Java, C++, Python

测试状态:  全部通过

各大算法平台题目详细扩展

LeetCode 平台详细题目

1. LeetCode 53 - 最大子数组和 (Maximum Subarray)

- **难度**: 中等
- **标签**: 数组、分治、动态规划
- **最优解**: Kadane 算法 ($O(n)$ 时间, $O(1)$ 空间)
- **分治法**: $O(n \log n)$ 时间, $O(\log n)$ 空间
- **链接**: <https://leetcode.com/problems/maximum-subarray/>

2. LeetCode 169 - 多数元素 (Majority Element)

- **难度**: 简单
- **标签**: 数组、分治、哈希表
- **最优解**: 摩尔投票算法 ($O(n)$ 时间, $O(1)$ 空间)
- **分治法**: $O(n \log n)$ 时间, $O(\log n)$ 空间
- **链接**: <https://leetcode.com/problems/majority-element/>

3. LeetCode 215 - 数组中的第 K 个最大元素 (Kth Largest Element)

- **难度**: 中等
- **标签**: 数组、分治、堆排序
- **最优解**: 快速选择算法 (平均 $O(n)$ 时间, $O(\log n)$ 空间)
- **分治法**: 基于快速排序思想
- **链接**: <https://leetcode.com/problems/kth-largest-element-in-an-array/>

4. LeetCode 240 - 搜索二维矩阵 II (Search a 2D Matrix II)

- **难度**: 中等
- **标签**: 数组、二分查找、分治
- **最优解**: Z 字形搜索 ($O(m+n)$ 时间, $O(1)$ 空间)
- **分治法**: $O(\tilde{m}^{1.58})$ 时间, $O(\log m)$ 空间
- **链接**: <https://leetcode.com/problems/search-a-2d-matrix-ii/>

5. LeetCode 50 - Pow(x, n) (快速幂)

- **难度**: 中等
- **标签**: 数学、分治、递归
- **最优解**: 快速幂算法 ($O(\log n)$ 时间, $O(\log n)$ 空间)
- **分治法**: 将指数分解为子问题
- **链接**: <https://leetcode.com/problems/powx-n/>

6. LeetCode 704 - 二分查找 (Binary Search)

- **难度**: 简单
- **标签**: 数组、二分查找、分治
- **最优解**: 二分查找 ($O(\log n)$ 时间, $O(1)$ 空间)
- **分治法**: 经典分治应用
- **链接**: <https://leetcode.com/problems/binary-search/>

7. LeetCode 493 - 翻转对 (Reverse Pairs)

- **难度**: 困难
- **标签**: 数组、分治、归并排序
- **最优解**: 归并排序变种 ($O(n \log n)$ 时间, $O(n)$ 空间)
- **分治法**: 基于归并排序思想
- **链接**: <https://leetcode.com/problems/reverse-pairs/>

8. LeetCode 315 - 计算右侧小于当前元素的个数

- **难度**: 困难
- **标签**: 数组、分治、归并排序
- **最优解**: 归并排序变种 ($O(n \log n)$ 时间, $O(n)$ 空间)
- **分治法**: 在归并过程中统计
- **链接**: <https://leetcode.com/problems/count-of-smaller-numbers-after-self/>

HackerRank 平台题目

1. Merge Sort: Counting Inversions

- **难度**: 中等
- **描述**: 使用归并排序计算数组中的逆序对数量
- **分治法**: 归并排序变种
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.hackerrank.com/challenges/ctci-merge-sort>

2. Closest Numbers

- **难度**: 简单
- **描述**: 找到数组中差值最小的两个数
- **分治法**: 排序后线性扫描或分治查找
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.hackerrank.com/challenges/closest-numbers>

Codeforces 平台题目

1. Codeforces 429D - Tricky Function

- **难度**: 1900
- **描述**: 最近点对问题的变种
- **分治法**: 最近点对算法

- **时间复杂度**: $O(n \log n)$
- **链接**: <https://codeforces.com/problemset/problem/429/D>

2. Codeforces 448D - Multiplication Table

- **难度**: 1800
- **描述**: 在乘法表中查找第 k 小的数
- **分治法**: 二分查找结合分治思想
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://codeforces.com/problemset/problem/448/D>

AtCoder 平台题目

1. ABC 139D - ModSum

- **难度**: 400
- **描述**: 数学分治问题
- **分治法**: 数学规律发现
- **时间复杂度**: $O(1)$
- **链接**: https://atcoder.jp/contests/abc139/tasks/abc139_d

2. ABC 177D - Friends

- **难度**: 400
- **描述**: 并查集应用，可分治优化
- **分治法**: 图的分治处理
- **时间复杂度**: $O(n \alpha(n))$
- **链接**: https://atcoder.jp/contests/abc177/tasks/abc177_d

USACO 平台题目

1. Barn Repair

- **难度**: 银牌
- **描述**: 贪心与分治结合
- **分治法**: 区间分割优化
- **时间复杂度**: $O(n \log n)$
- **链接**: <http://www.usaco.org/index.php?page=viewproblem2&cpid=101>

2. The Castle

- **难度**: 银牌
- **描述**: 图的分治处理
- **分治法**: 连通分量分治
- **时间复杂度**: $O(n^2)$
- **链接**: <http://www.usaco.org/index.php?page=viewproblem2&cpid=101>

洛谷平台题目

1. P1177 【模板】快速排序

- **难度**: 普及-
- **描述**: 快速排序模板题
- **分治法**: 快速排序算法
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.luogu.com.cn/problem/P1177>

2. P1908 逆序对

- **难度**: 普及/提高-
- **描述**: 归并排序求逆序对
- **分治法**: 归并排序变种
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.luogu.com.cn/problem/P1908>

3. P1429 平面最近点对（加强版）

- **难度**: 提高+/省选-
- **描述**: 经典最近点对问题
- **分治法**: 最近点对算法
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.luogu.com.cn/problem/P1429>

牛客网题目

1. NC105 二分查找-II

- **难度**: 简单
- **描述**: 二分查找第一个出现的位置
- **分治法**: 二分查找变种
- **时间复杂度**: $O(\log n)$
- **链接**: <https://www.nowcoder.com/practice/4f470d1d3b734f8aaf2afb014185b395>

2. NC140 排序

- **难度**: 简单
- **描述**: 各种排序算法实现
- **分治法**: 快速排序、归并排序
- **时间复杂度**: $O(n \log n)$
- **链接**: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>

杭电 OJ 题目

1. HDU 1007 Quoit Design

- **难度**: 中等
- **描述**: 平面最近点对问题

- **分治法**: 经典最近点对算法
- **时间复杂度**: $O(n \log n)$
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1007>

2. HDU 1024 Max Sum Plus Plus

- **难度**: 困难
- **描述**: 最大子数组和升级版
- **分治法**: 动态规划优化
- **时间复杂度**: $O(mn)$
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1024>

POJ 平台题目

1. POJ 2299 Ultra-QuickSort

- **难度**: 简单
- **描述**: 归并排序求逆序对
- **分治法**: 归并排序应用
- **时间复杂度**: $O(n \log n)$
- **链接**: <http://poj.org/problem?id=2299>

2. POJ 3714 Raid

- **难度**: 中等
- **描述**: 最近点对问题
- **分治法**: 经典最近点对算法
- **时间复杂度**: $O(n \log n)$
- **链接**: <http://poj.org/problem?id=3714>

分治法算法深度解析

核心思想深度理解

分治法的本质:

- **分解**: 将复杂问题分解为相似的子问题
- **解决**: 递归解决子问题，小问题直接求解
- **合并**: 将子问题的解合并为原问题的解

适用场景特征:

1. 问题可分解为独立子问题
2. 子问题与原问题结构相同
3. 子问题的解可有效合并
4. 子问题规模逐渐减小

时间复杂度深度分析

主定理(Master Theorem)详细应用:

对于递归式: $T(n) = aT(n/b) + f(n)$

情况 1: $f(n) = O(n^{\lceil \log_b(a) - \varepsilon \rceil})$

- 解: $T(n) = \Theta(n^{\lceil \log_b(a) \rceil})$

情况 2: $f(n) = \Theta(n^{\lceil \log_b(a) \rceil} \log^k n)$

- 解: $T(n) = \Theta(n^{\lceil \log_b(a) \rceil} \log^{k+1} n)$

情况 3: $f(n) = \Omega(n^{\lceil \log_b(a) + \varepsilon \rceil})$ 且 $aT(n/b) \leq cf(n)$

- 解: $T(n) = \Theta(f(n))$

经典例子分析:

- 归并排序: $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

- 二分查找: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$

- 快速排序(平均): $T(n) = T(n/2) + O(n) \rightarrow O(n \log n)$

工程实践深度考量

1. 性能优化策略

避免递归过深:

``` java

// 设置递归深度阈值

```
private static final int MAX_RECURSION_DEPTH = 100;
```

```
if (depth > MAX_RECURSION_DEPTH) {
```

// 改用迭代或其他算法

```
 return iterativeSolution(...);
```

```
}
```

```

尾递归优化:

``` java

// 尾递归形式

```
private static int factorial(int n, int acc) {
```

```
 if (n == 0) return acc;
```

```
 return factorial(n - 1, n * acc);
```

```
}
```

```

****并行化处理**:**

```
``` java
// 使用 ForkJoin 框架并行处理
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new RecursiveTask() {
 @Override
 protected Integer compute() {
 // 分治任务
 }
}) ;
```

```

2. 内存管理优化

****避免重复计算**:**

```
``` java
// 使用缓存存储中间结果
Map<String, Integer> cache = new HashMap<>();

private int solve(String key) {
 if (cache.containsKey(key)) {
 return cache.get(key);
 }
 // 计算并缓存结果
 int result = ...;
 cache.put(key, result);
 return result;
}
```

```

****空间复杂度优化**:**

```
``` java
// 原地操作减少空间使用
void mergeSort(int[] arr, int[] temp, int left, int right) {
 if (left < right) {
 int mid = (left + right) / 2;
 mergeSort(arr, temp, left, mid);
 mergeSort(arr, temp, mid + 1, right);
 merge(arr, temp, left, mid, right);
 }
}
```

```

3. 异常处理完善

边界情况处理:

```
``` java
public int findMax(int[] arr) {
 if (arr == null) {
 throw new IllegalArgumentException("数组不能为 null");
 }
 if (arr.length == 0) {
 throw new IllegalArgumentException("数组不能为空");
 }
 return findMaxRecursive(arr, 0, arr.length - 1);
}
```
```

```

\*\*栈溢出防护\*\*:

```
``` java
private int findMaxRecursive(int[] arr, int left, int right) {
    // 检查递归深度
    if (right - left < THRESHOLD) {
        return findMaxIterative(arr, left, right);
    }
    // 正常递归处理
}
```
```

```

调试与定位深度技巧

1. 详细日志记录

```
``` java
private static int divideAndConquer(int[] arr, int left, int right, int depth) {
 System.out.printf("深度%d: 处理区间[%d, %d]%", depth, left, right);

 if (left == right) {
 System.out.printf("深度%d: 基准情况, 返回 arr[%d]=%d%n", depth, left, arr[left]);
 return arr[left];
 }

 int mid = left + (right - left) / 2;
 System.out.printf("深度%d: 分割点 mid=%d%n", depth, mid);

 int leftResult = divideAndConquer(arr, left, mid, depth + 1);
 int rightResult = divideAndConquer(arr, mid + 1, right, depth + 1);
}
```
```

```

```
 int result = Math.max(leftResult, rightResult);
 System.out.printf("深度%d: 合并结果, 左=%d, 右=%d, 最终=%d%n",
 depth, leftResult, rightResult, result);

 return result;
}
```
```

```

#### #### 2. 断言验证

```
``` java
private static void testDivideAndConquer() {
    int[] testCases = {
        // 各种测试数据
    };

    for (int[] testCase : testCases) {
        int expected = findMaxIterative(testCase);
        int actual = findMaxRecursive(testCase, 0, testCase.length - 1);

        assert expected == actual :
            String.format("测试失败: 期望%d, 实际%d", expected, actual);
    }
}
```
```

```

3. 性能监控

```
``` java
public class PerformanceMonitor {
 private long startTime;
 private int callCount;

 public void start() {
 startTime = System.nanoTime();
 callCount = 0;
 }

 public void recordCall() {
 callCount++;
 }

 public void printStats() {
 long duration = System.nanoTime() - startTime;
 System.out.printf("调用次数: %d, 耗时: %.2fms%n",
 callCount, duration / 1000000.0);
 }
}
```
```

```

```
 callCount, duration / 1_000_000.0);
}
}
~~~
```

### ### 面试深度准备

#### #### 1. 算法理解深度

\*\*分治法 vs 动态规划\*\*:

- 分治法: 子问题独立, 无重叠
- 动态规划: 子问题重叠, 需要记忆化
- 贪心算法: 局部最优选择

\*\*分治法适用条件\*\*:

1. 问题可分解为相似子问题
2. 子问题解可有效合并
3. 子问题规模指数组减小

#### #### 2. 代码实现深度

\*\*清晰的递归结构\*\*:

~~~ java

```
public ReturnType solve(Problem problem) {
    // 1. 基准情况处理
    if (isBaseCase(problem)) {
        return solveBaseCase(problem);
    }

    // 2. 分解问题
    Problem[] subproblems = divide(problem);

    // 3. 递归求解
    ReturnType[] subResults = new ReturnType[subproblems.length];
    for (int i = 0; i < subproblems.length; i++) {
        subResults[i] = solve(subproblems[i]);
    }

    // 4. 合并结果
    return combine(subResults);
}
```

完整的测试用例:

~~~ java

```

public class DivideConquerTest {
    @Test
    public void testEmptyArray() {
        assertThrows(IllegalArgumentException.class,
                     () -> findMax(new int[0]));
    }

    @Test
    public void testSingleElement() {
        assertEquals(5, findMax(new int[] {5}));
    }

    @Test
    public void testNormalCase() {
        assertEquals(8, findMax(new int[] {1, 3, 8, 2, 5}));
    }

    @Test
    public void testAllNegative() {
        assertEquals(-1, findMax(new int[] {-5, -3, -1, -10}));
    }
}
```

```

### ##### 3. 问题分析深度

**\*\*时间复杂度推导\*\*:**

- 建立递归关系式
- 应用主定理分析
- 考虑最坏/平均情况

**\*\*空间复杂度分析\*\*:**

- 递归栈空间
- 辅助空间使用
- 内存访问模式

**\*\*优化策略讨论\*\*:**

- 算法层面优化
- 实现层面优化
- 工程实践优化

----

## 总结与展望

#### #### 学习成果检验

通过本专题学习，你应该能够：

1. 熟练运用分治法解决各类问题
2. 准确分析算法时间空间复杂度
3. 实现高质量的算法代码
4. 应对算法面试中的各种问题

#### #### 后续学习方向

1. \*\*高级分治算法\*\*: Strassen 矩阵乘法、FFT 等
2. \*\*并行分治\*\*: MapReduce、Fork-Join 框架
3. \*\*分治数据结构\*\*: 线段树、KD 树等
4. \*\*分治在 AI 中的应用\*\*: 决策树、集成学习等

#### #### 实践建议

1. 定期刷题保持手感
2. 参与算法竞赛锻炼
3. 阅读经典算法书籍
4. 参与开源项目实践

---

#### \*\*维护更新计划\*\*:

- 定期添加新题目
- 更新最优解算法
- 完善工程实践内容
- 优化代码实现

#### \*\*反馈与贡献\*\*:

欢迎提交 Issue 或 Pull Request 来完善本专题内容。

### ## 扩展题目：来自各大算法平台的分治法题目

#### #### 题目 5：归并排序（经典分治算法）

\*\*题目来源\*\*：经典排序算法

\*\*问题描述\*\*：使用归并排序对数组进行排序

\*\*时间复杂度\*\*： $O(n \log n)$

\*\*空间复杂度\*\*： $O(n)$

\*\*是否最优解\*\*：对于基于比较的排序算法，归并排序的时间复杂度已经是最优的

#### #### 题目 6：二分查找（分治思想应用）

\*\*题目来源\*\*：LeetCode 704. Binary Search

\*\*链接\*\*：<https://leetcode.com/problems/binary-search/>

**\*\*中文链接\*\*:** <https://leetcode.cn/problems/binary-search/>

**\*\*时间复杂度\*\*:**  $O(\log n)$

**\*\*空间复杂度\*\*:**  $O(1)$  (迭代版本)

**\*\*是否最优解\*\*:** 二分查找是有序数组查找问题的最优解

#### #### 题目 7: 快速幂算法 (分治思想)

**\*\*题目来源\*\*:** LeetCode 50. Pow(x, n)

**\*\*链接\*\*:** <https://leetcode.com/problems/powx-n/>

**\*\*中文链接\*\*:** <https://leetcode.cn/problems/powx-n/>

**\*\*时间复杂度\*\*:**  $O(\log n)$

**\*\*空间复杂度\*\*:**  $O(\log n)$  (递归版本)

**\*\*是否最优解\*\*:** 快速幂算法是计算幂函数的最优解

#### #### 题目 8: 最大子矩阵和 (二维分治法)

**\*\*题目来源\*\*:** LeetCode 363. Max Sum of Rectangle No Larger Than K

**\*\*问题描述\*\*:** 给定一个二维矩阵, 找出一个子矩阵, 使得其元素和最大

**\*\*时间复杂度\*\*:**  $O(n^3 \log n)$  (分治法)

**\*\*空间复杂度\*\*:**  $O(n^2)$

**\*\*是否最优解\*\*:** 对于二维最大子矩阵和, 存在  $O(n^3)$  的动态规划解法

#### #### 题目 9: Strassen 矩阵乘法 (优化分治算法)

**\*\*题目来源\*\*:** 经典算法问题

**\*\*问题描述\*\*:** 实现 Strassen 算法计算两个  $n \times n$  矩阵的乘积

**\*\*时间复杂度\*\*:**  $O(n^{2.807})$

**\*\*空间复杂度\*\*:**  $O(n^2)$

**\*\*是否最优解\*\*:** Strassen 算法比传统矩阵乘法更优, 但存在更优的矩阵乘法算法

#### #### 题目 10: 最近点对问题 (分治法)

**\*\*题目来源\*\*:** 经典计算几何问题

**\*\*问题描述\*\*:** 在平面上有  $n$  个点, 找出其中距离最近的一对点

**\*\*时间复杂度\*\*:**  $O(n \log n)$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*是否最优解\*\*:** 该问题的最优时间复杂度为  $O(n \log n)$

#### #### 题目 11: Karatsuba 大整数乘法 (分治法)

**\*\*题目来源\*\*:** 经典算法问题

**\*\*问题描述\*\*:** 实现 Karatsuba 算法进行大整数乘法运算

**\*\*时间复杂度\*\*:**  $O(n^{\log_2 3}) \approx O(n^{1.585})$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*是否最优解\*\*:** 比传统  $O(n^2)$  算法更优, 但存在更优的 FFT 算法  $O(n \log n)$

#### 更多题目详见 [ADDITIONAL\_PROBLEMS.md] (file:///d:/Upan/src/algorith-journey/src/algorith-journey/src/class020/ADDITIONAL\_PROBLEMS.md) 文件

=====

[代码文件]

=====

文件: Get.MaxValue.cpp

=====

```
/**  
 * 分治法求解数组最大值问题 (C++版本)  
 *  
 * 问题描述:  
 * 给定一个数组，找出其中的最大值。  
 *  
 * 解法思路:  
 * 使用分治法，将数组不断二分，直到只有一个元素时直接返回，  
 * 然后比较左右两部分的最大值，返回较大者。  
 *  
 * 算法特点:  
 * 1. 分治策略：将大问题分解为小问题  
 * 2. 递归实现：通过递归不断分解问题  
 * 3. 合并结果：比较子问题的解得到原问题的解  
 *  
 * 时间复杂度分析:  
 *  $T(n) = 2*T(n/2) + O(1)$   
 * 根据主定理，时间复杂度为  $O(n)$   
 *  
 * 空间复杂度分析:  
 * 递归调用栈的深度为  $O(\log n)$   
 * 空间复杂度为  $O(\log n)$   
 *  
 * 相关题目扩展:  
 * 1. LeetCode 53. 最大子数组和 (分治解法)  
 * 2. 求解数组中最大值和最小值  
 * 3. 求解数组中第 k 大元素  
 * 4. 分治法求解最大子矩阵和  
 *  
 * 工程化考量:  
 * 1. 异常处理：检查空数组情况  
 * 2. 边界处理：处理只有一个元素的数组  
 * 3. 性能优化：对于小规模数据可直接遍历  
 * 4. 可配置性：可扩展为求解任意范围内的最值  
 *  
 * 与标准库对比:
```

- \* C++标准库中 std::max\_element() 等方法
- \* 通常使用迭代而非递归，避免栈溢出风险
- \*
- \* 语言特性差异：
  - \* Java：使用 Math.max() 函数
  - \* C++：使用 std::max() 函数
  - \* Python：使用内置 max() 函数或自定义比较
- \*
- \* 极端场景考虑：
  - \* 1. 空数组：需要特殊处理
  - \* 2. 单元素数组：直接返回
  - \* 3. 大规模数组：可能栈溢出，需改用迭代
  - \* 4. 所有元素相同：任一元素都是最大值
- \*
- \* 调试技巧：
  - \* 1. 打印递归调用过程中的中间结果
  - \* 2. 使用断言验证左右子数组的最值正确性
  - \* 3. 性能测试：比较不同规模数据的执行时间

// 由于当前环境可能不支持完整的 C++ 标准库，这里提供概念性代码

// 实际编译时需要包含正确的头文件

```
/*  
 * 分治法求解数组指定范围内的最大值  
 * @param arr 数组  
 * @param l 左边界（包含）  
 * @param r 右边界（包含）  
 * @return 指定范围内的最大值  
 */
```

```
int f(int arr[], int l, int r) {  
    // 基本情况：只有一个元素时直接返回  
    if (l == r) {  
        return arr[l];  
    }
```

```
    // 分解：计算中点，将数组分为两部分  
    int m = (l + r) / 2;
```

```
    // 递归求解：分别求左右两部分的最大值  
    int lmax = f(arr, l, m);  
    int rmax = f(arr, m + 1, r);
```

```
// 合并：返回左右两部分最大值中的较大者
return (lmax > rmax) ? lmax : rmax;
}
```

```
/*
 * 入口方法，求数组中的最大值
 * @param arr 输入数组
 * @param size 数组大小
 * @return 数组中的最大值
 */
```

```
int maxValue(int arr[], int size) {
    // 异常处理：检查数组是否为空
    if (arr == 0 || size <= 0) {
        return -1; // 简化处理，实际应抛出异常
    }
```

```
    // 调用分治方法求解
    return f(arr, 0, size - 1);
}
```

```
/*
 * 以下是完整的 C++ 实现（需要适当编译环境）
 */
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <cfloat>
#include <cmath>
```

```
using namespace std;
```

```
class Get.MaxValue {
public:
    // 入口方法，求数组中的最大值
    static int maxValue(const vector<int>& arr) {
        // 异常处理：检查数组是否为空
        if (arr.empty()) {
            throw invalid_argument("数组不能为空");
        }
        // 分治逻辑
        int lmax = arr[0];
        int rmax = arr[0];
        for (int i = 1; i < arr.size(); i++) {
            if (arr[i] > lmax) {
                lmax = arr[i];
            }
            if (arr[i] > rmax) {
                rmax = arr[i];
            }
        }
        return (lmax > rmax) ? lmax : rmax;
    }
}
```

```

    }

    // 调用分治方法求解
    return f(arr, 0, arr.size() - 1);
}

private:

    // 分治法求解数组指定范围内的最大值
    static int f(const vector<int>& arr, int l, int r) {
        // 基本情况：只有一个元素时直接返回
        if (l == r) {
            return arr[l];
        }

        // 分解：计算中点，将数组分为两部分
        int m = (l + r) / 2;

        // 递归求解：分别求左右两部分的最大值
        int lmax = f(arr, l, m);
        int rmax = f(arr, m + 1, r);

        // 合并：返回左右两部分最大值中的较大者
        return max(lmax, rmax);
    }

public:
    // ===== 题目 1: LeetCode 53. 最大子数组和（分治解法） =====
    /**
     * 题目来源: LeetCode 53. Maximum Subarray
     * 题目链接: https://leetcode.com/problems/maximum-subarray/
     * 中文链接: https://leetcode.cn/problems/maximum-subarray/
     *
     * 题目描述:
     * 给定一个整数数组 nums，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
     *
     * 示例 1:
     * 输入: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
     * 输出: 6
     * 解释: 连续子数组 [4, -1, 2, 1] 的和最大，为 6。
     *
     * 时间复杂度: O(n*log n) - 分治法
     * 空间复杂度: O(log n) - 递归栈
    */
}

```

```

*
* 最优解: Kadane 算法, 时间 O(n), 空间 O(1)
*/
static int maxSubArray(const vector<int>& nums) {
    if (nums.empty()) {
        throw invalid_argument("数组不能为空");
    }
    return maxSubArrayDivide(nums, 0, nums.size() - 1);
}

private:
    static int maxSubArrayDivide(const vector<int>& nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }

        int mid = left + (right - left) / 2;
        int leftMax = maxSubArrayDivide(nums, left, mid);
        int rightMax = maxSubArrayDivide(nums, mid + 1, right);

        // 计算跨越中点的最大子数组和
        int leftCrossMax = INT_MIN;
        int leftSum = 0;
        for (int i = mid; i >= left; i--) {
            leftSum += nums[i];
            leftCrossMax = max(leftCrossMax, leftSum);
        }

        int rightCrossMax = INT_MIN;
        int rightSum = 0;
        for (int i = mid + 1; i <= right; i++) {
            rightSum += nums[i];
            rightCrossMax = max(rightCrossMax, rightSum);
        }

        int crossMax = leftCrossMax + rightCrossMax;
        return max({leftMax, rightMax, crossMax});
    }
}

public:
    /**
     * 最优解: Kadane 算法
     * 时间复杂度: O(n)

```

```

* 空间复杂度: O(1)
*/
static int maxSubArrayOptimal(const vector<int>& nums) {
    if (nums.empty()) {
        throw invalid_argument("数组不能为空");
    }

    int maxSum = nums[0];
    int curSum = nums[0];

    for (size_t i = 1; i < nums.size(); i++) {
        curSum = max(nums[i], curSum + nums[i]);
        maxSum = max(maxSum, curSum);
    }

    return maxSum;
}

// ===== 题目 2: LeetCode 169. 多数元素 (分治解法) =====
/***
 * 题目来源: LeetCode 169. Majority Element
 * 题目链接: https://leetcode.com/problems/majority-element/
 * 中文链接: https://leetcode.cn/problems/majority-element/
 *
 * 题目描述:
 * 给定一个大小为 n 的数组 nums，返回其中的多数元素。
 * 多数元素是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素。
 *
 * 时间复杂度: O(n*log n) - 分治法
 * 空间复杂度: O(log n)
 *
 * 最优解: 摩尔投票算法, 时间 O(n), 空间 O(1)
 */
static int majorityElement(const vector<int>& nums) {
    if (nums.empty()) {
        throw invalid_argument("数组不能为空");
    }

    return majorityElementDivide(nums, 0, nums.size() - 1);
}

private:
    static int majorityElementDivide(const vector<int>& nums, int left, int right) {
        if (left == right) {

```

```

        return nums[left];
    }

    int mid = left + (right - left) / 2;
    int leftMajor = majorityElementDivide(nums, left, mid);
    int rightMajor = majorityElementDivide(nums, mid + 1, right);

    if (leftMajor == rightMajor) {
        return leftMajor;
    }

    int leftCount = countInRange(nums, leftMajor, left, right);
    int rightCount = countInRange(nums, rightMajor, left, right);

    return leftCount > rightCount ? leftMajor : rightMajor;
}

static int countInRange(const vector<int>& nums, int target, int left, int right) {
    int count = 0;
    for (int i = left; i <= right; i++) {
        if (nums[i] == target) {
            count++;
        }
    }
    return count;
}

public:
    /**
     * 最优解：摩尔投票算法
     * 时间复杂度：O(n)
     * 空间复杂度：O(1)
     */
    static int majorityElementOptimal(const vector<int>& nums) {
        if (nums.empty()) {
            throw invalid_argument("数组不能为空");
        }

        int candidate = nums[0];
        int count = 1;

        for (size_t i = 1; i < nums.size(); i++) {
            if (count == 0) {

```

```

        candidate = nums[i];
        count = 1;
    } else if (nums[i] == candidate) {
        count++;
    } else {
        count--;
    }
}

return candidate;
}

// ===== 题目 3: LeetCode 215. 数组中的第 K 个最大元素 =====
/***
 * 题目来源: LeetCode 215. Kth Largest Element in an Array
 * 题目链接: https://leetcode.com/problems/kth-largest-element-in-an-array/
 * 中文链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 题目描述:
 * 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 *
 * 快速选择算法 (基于分治思想)
 * 平均时间复杂度: O(n)
 * 最坏时间复杂度: O(n^2)
 * 空间复杂度: O(log n)
 */
static int findKthLargest(vector<int>& nums, int k) {
    if (nums.empty() || k < 1 || k > (int)nums.size()) {
        throw invalid_argument("参数非法");
    }
    srand(time(NULL));
    return quickSelect(nums, 0, nums.size() - 1, nums.size() - k);
}

private:
    static int quickSelect(vector<int>& nums, int left, int right, int k) {
        if (left == right) {
            return nums[left];
        }

        int pivotIndex = left + rand() % (right - left + 1);
        pivotIndex = partition(nums, left, right, pivotIndex);

        if (pivotIndex < k) {
            return quickSelect(nums, pivotIndex + 1, right, k);
        } else if (pivotIndex > k) {
            return quickSelect(nums, left, pivotIndex - 1, k);
        } else {
            return nums[pivotIndex];
        }
    }
}

```

```

    if (k == pivotIndex) {
        return nums[k];
    } else if (k < pivotIndex) {
        return quickSelect(nums, left, pivotIndex - 1, k);
    } else {
        return quickSelect(nums, pivotIndex + 1, right, k);
    }
}

static int partition(vector<int>& nums, int left, int right, int pivotIndex) {
    int pivotValue = nums[pivotIndex];
    swap(nums[pivotIndex], nums[right]);

    int storeIndex = left;
    for (int i = left; i < right; i++) {
        if (nums[i] < pivotValue) {
            swap(nums[storeIndex], nums[i]);
            storeIndex++;
        }
    }

    swap(nums[storeIndex], nums[right]);
    return storeIndex;
}

public:
// ===== 题目 4: LeetCode 240. 搜索二维矩阵 II =====
/***
 * 题目来源: LeetCode 240. Search a 2D Matrix II
 * 链接: https://leetcode.com/problems/search-a-2d-matrix-ii/
 * 中文: https://leetcode.cn/problems/search-a-2d-matrix-ii/
 *
 * 最优解: 从右上角或左下角搜索
 * 时间复杂度: O(m+n)
 * 空间复杂度: O(1)
 */
static bool searchMatrix(const vector<vector<int>>& matrix, int target) {
    if (matrix.empty() || matrix[0].empty()) {
        return false;
    }

    int row = 0;
    int col = matrix[0].size() - 1;

```

```

        while (row < (int)matrix.size() && col >= 0) {
            if (matrix[row][col] == target) {
                return true;
            } else if (matrix[row][col] > target) {
                col--;
            } else {
                row++;
            }
        }

        return false;
    }

// ===== 补充题目 5: 归并排序 =====
/***
 * 题目来源: 经典排序算法
 *
 * 题目描述:
 * 实现归并排序算法, 将一个数组排序。
 *
 * 解题思路:
 * 1. 将数组分成两半, 分别排序
 * 2. 合并两个已排序的子数组
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
static void mergeSort(vector<int>& nums) {
    if (nums.empty()) {
        return;
    }
    vector<int> temp(nums.size());
    mergeSortHelper(nums, 0, nums.size() - 1, temp);
}

private:
    static void mergeSortHelper(vector<int>& nums, int left, int right, vector<int>& temp) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSortHelper(nums, left, mid, temp);
            mergeSortHelper(nums, mid + 1, right, temp);
            merge(nums, left, mid, right, temp);
        }
    }
}

```

```

    }

}

static void merge(vector<int>& nums, int left, int mid, int right, vector<int>& temp) {
    int i = left;          // 左半部分起始索引
    int j = mid + 1;        // 右半部分起始索引
    int k = left;           // 临时数组起始索引

    // 合并两个子数组
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) {
        temp[k++] = nums[i++];
    }
    while (j <= right) {
        temp[k++] = nums[j++];
    }

    // 将临时数组的元素复制回原数组
    for (int p = left; p <= right; p++) {
        nums[p] = temp[p];
    }
}

public:
// ===== 补充题目 6: 二分查找 =====
/***
 * 题目来源: 经典搜索算法
 *
 * 题目描述:
 * 在一个排序数组中查找目标值, 如果找到返回索引, 否则返回-1。
 *
 * 解题思路:
 * 1. 将区间不断二分, 比较中间元素与目标值
 * 2. 根据比较结果调整搜索区间
 *

```

```

* 时间复杂度: O(log n)
* 空间复杂度: O(log n) - 递归实现
*/
static int binarySearch(const vector<int>& nums, int target) {
    if (nums.empty()) {
        return -1;
    }
    return binarySearchHelper(nums, target, 0, nums.size() - 1);
}

private:
    static int binarySearchHelper(const vector<int>& nums, int target, int left, int right) {
        if (left > right) {
            return -1; // 未找到目标值
        }

        int mid = left + (right - left) / 2; // 避免整数溢出

        if (nums[mid] == target) {
            return mid; // 找到目标值
        } else if (nums[mid] > target) {
            return binarySearchHelper(nums, target, left, mid - 1); // 在左半部分搜索
        } else {
            return binarySearchHelper(nums, target, mid + 1, right); // 在右半部分搜索
        }
    }
}

public:
    /**
     * 二分查找最优解: 迭代实现
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     */
    static int binarySearchOptimal(const vector<int>& nums, int target) {
        if (nums.empty()) {
            return -1;
        }

        int left = 0;
        int right = nums.size() - 1;

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

```

```

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return -1; // 未找到目标值
}

// ===== 补充题目 7: 快速幂算法 =====
/***
 * 题目来源: 经典算法题
 *
 * 题目描述:
 * 计算 a 的 n 次方, 要求时间复杂度优于 O(n)。
 *
 * 解题思路:
 * 1. 使用分治法, 将  $a^n$  分解为  $(a^{(n/2)})^2$ 
 * 2. 递归计算子问题
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(log n) - 递归栈
 */
static double quickPow(double a, int n) {
    // 处理特殊情况
    if (n == 0) {
        return 1.0;
    }
    if (a == 0.0) {
        return 0.0;
    }

    // 处理负数指数
    if (n < 0) {
        a = 1.0 / a;
        // 处理整数溢出
        if (n == INT_MIN) {
            return a * quickPow(a, -(n + 1));
        }
    }
}

```

```

        n = -n;
    }

    return quickPowHelper(a, n);
}

private:

    static double quickPowHelper(double a, int n) {
        if (n == 0) {
            return 1.0;
        }

        double half = quickPowHelper(a, n / 2);
        if (n % 2 == 0) {
            return half * half;
        } else {
            return half * half * a;
        }
    }
}

public:

    /**
     * 快速幂最优解: 迭代实现
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     */
    static double quickPowOptimal(double a, int n) {
        // 处理特殊情况
        if (n == 0) {
            return 1.0;
        }
        if (a == 0.0) {
            return 0.0;
        }

        long long exponent = n; // 避免整数溢出
        if (n < 0) {
            a = 1.0 / a;
            exponent = -exponent;
        }

        double result = 1.0;
        double current = a;

```

```

while (exponent > 0) {
    if (exponent % 2 == 1) {
        result *= current;
    }
    current *= current;
    exponent /= 2;
}

return result;
}

// ===== 补充题目 8: 最大子矩阵和 =====
/***
 * 题目来源: 经典算法题
 *
 * 题目描述:
 * 给定一个二维矩阵, 找出其中和最大的子矩阵。
 *
 * 解题思路:
 * 1. 将问题转化为一维最大子数组和问题
 * 2. 枚举左右边界, 计算每一行的前缀和
 * 3. 使用 Kadane 算法求解一维最大子数组和
 *
 * 时间复杂度: O(n^3)
 * 空间复杂度: O(n)
 */
static int maxSubMatrix(const vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        throw invalid_argument("矩阵不能为空");
    }

    int n = matrix.size();      // 行数
    int m = matrix[0].size();   // 列数
    int maxSum = INT_MIN;

    // 枚举左右边界
    for (int left = 0; left < m; left++) {
        vector<int> rowSum(n, 0); // 记录每一行的和

        for (int right = left; right < m; right++) {
            // 计算每一行在左右边界内的和
            for (int i = 0; i < n; i++) {

```

```

        rowSum[i] += matrix[i][right];
    }

    // 使用 Kadane 算法求解一维最大子数组和
    int currentSum = 0;
    int currentMax = INT_MIN;

    for (int sum : rowSum) {
        currentSum = max(sum, currentSum + sum);
        currentMax = max(currentMax, currentSum);
    }

    maxSum = max(maxSum, currentMax);
}

return maxSum;
}

// ===== 补充题目 9: Strassen 矩阵乘法 =====
/***
 * 题目来源: 经典算法题
 *
 * 题目描述:
 * 实现 Strassen 矩阵乘法算法, 优化传统的 O(n^3) 矩阵乘法。
 *
 * 解题思路:
 * 1. 将矩阵分成四个子矩阵
 * 2. 计算 7 个中间矩阵 (而非传统算法的 8 个乘法)
 * 3. 通过中间矩阵计算结果矩阵的四个子矩阵
 *
 * 时间复杂度: O(n^log2(7)) ≈ O(n^2.81)
 * 空间复杂度: O(n^2)
 */
static vector<vector<int>> strassenMultiply(vector<vector<int>>& A, vector<vector<int>>& B) {
    int n = A.size();
    // 确保矩阵是方阵且大小为 2 的幂
    int size = 1;
    while (size < n) {
        size <<= 1;
    }

    // 扩展矩阵到 2 的幂大小

```

```

vector<vector<int>> A_ext(size, vector<int>(size, 0));
vector<vector<int>> B_ext(size, vector<int>(size, 0));

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        A_ext[i][j] = A[i][j];
        B_ext[i][j] = B[i][j];
    }
}

vector<vector<int>> C_ext = strassenMultiplyHelper(A_ext, B_ext, size);

// 裁剪回原始大小
vector<vector<int>> C(n, vector<int>(n));
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i][j] = C_ext[i][j];
    }
}

return C;
}

private:
    static vector<vector<int>> strassenMultiplyHelper(const vector<vector<int>>& A, const
vector<vector<int>>& B, int size) {
        if (size == 1) {
            vector<vector<int>> C(1, vector<int>(1));
            C[0][0] = A[0][0] * B[0][0];
            return C;
        }

        int newSize = size / 2;

        // 分割矩阵
        vector<vector<int>> A11(newSize, vector<int>(newSize));
        vector<vector<int>> A12(newSize, vector<int>(newSize));
        vector<vector<int>> A21(newSize, vector<int>(newSize));
        vector<vector<int>> A22(newSize, vector<int>(newSize));

        vector<vector<int>> B11(newSize, vector<int>(newSize));
        vector<vector<int>> B12(newSize, vector<int>(newSize));
        vector<vector<int>> B21(newSize, vector<int>(newSize));

```

```

vector<vector<int>> B22(newSize, vector<int>(newSize));

// 填充子矩阵
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + newSize];
        A21[i][j] = A[i + newSize][j];
        A22[i][j] = A[i + newSize][j + newSize];

        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + newSize];
        B21[i][j] = B[i + newSize][j];
        B22[i][j] = B[i + newSize][j + newSize];
    }
}

// 计算 7 个中间矩阵
vector<vector<int>> temp1 = addMatrices(A11, A22);
vector<vector<int>> temp2 = addMatrices(B11, B22);
vector<vector<int>> M1 = strassenMultiplyHelper(temp1, temp2, newSize);

vector<vector<int>> temp3 = addMatrices(A21, A22);
vector<vector<int>> M2 = strassenMultiplyHelper(temp3, B11, newSize);

vector<vector<int>> temp4 = subtractMatrices(B12, B22);
vector<vector<int>> M3 = strassenMultiplyHelper(A11, temp4, newSize);

vector<vector<int>> temp5 = subtractMatrices(B21, B11);
vector<vector<int>> M4 = strassenMultiplyHelper(A22, temp5, newSize);

vector<vector<int>> temp6 = addMatrices(A11, A12);
vector<vector<int>> M5 = strassenMultiplyHelper(temp6, B22, newSize);

vector<vector<int>> temp7 = subtractMatrices(A21, A11);
vector<vector<int>> temp8 = addMatrices(B11, B12);
vector<vector<int>> M6 = strassenMultiplyHelper(temp7, temp8, newSize);

vector<vector<int>> temp9 = subtractMatrices(A12, A22);
vector<vector<int>> temp10 = addMatrices(B21, B22);
vector<vector<int>> M7 = strassenMultiplyHelper(temp9, temp10, newSize);

// 计算结果矩阵的子矩阵

```

```

vector<vector<int>> temp11 = addMatrices(M1, M4);
vector<vector<int>> temp12 = subtractMatrices(temp11, M5);
vector<vector<int>> C11 = addMatrices(temp12, M7);
vector<vector<int>> C12 = addMatrices(M3, M5);
vector<vector<int>> C21 = addMatrices(M2, M4);
vector<vector<int>> temp13 = addMatrices(M1, M3);
vector<vector<int>> temp14 = subtractMatrices(temp13, M2);
vector<vector<int>> C22 = addMatrices(temp14, M6);

// 合并结果矩阵
vector<vector<int>> C(size, vector<int>(size));
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        C[i][j] = C11[i][j];
        C[i][j + newSize] = C12[i][j];
        C[i + newSize][j] = C21[i][j];
        C[i + newSize][j + newSize] = C22[i][j];
    }
}

return C;
}

static vector<vector<int>> addMatrices(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}

static vector<vector<int>> subtractMatrices(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
}

```

```

    }

    return C;
}

public:
// ===== 补充题目 10: 最近点对问题 =====
/***
 * 题目来源: 经典算法题
 *
 * 题目描述:
 * 给定平面上的 n 个点, 找出距离最近的一对点。
 *
 * 解题思路:
 * 1. 按 x 坐标排序所有点
 * 2. 使用分治法递归求解左右两部分的最近点对
 * 3. 处理跨越中间线的最近点对
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */

static double closestPair(vector<pair<double, double>>& points) {
    if (points.size() < 2) {
        throw invalid_argument("至少需要两个点");
    }

    // 按 x 坐标排序
    sort(points.begin(), points.end());

    // 创建一个按 y 坐标排序的数组, 用于后续处理
    vector<pair<double, double>> pointsSortedByY = points;
    sort(pointsSortedByY.begin(), pointsSortedByY.end(), [] (const pair<double, double>& a,
const pair<double, double>& b) {
        return a.second < b.second;
    });

    return closestPairHelper(points, 0, points.size() - 1, pointsSortedByY);
}

private:
    static double closestPairHelper(vector<pair<double, double>>& points, int left, int right,
vector<pair<double, double>>& pointsSortedByY) {
        // 基本情况: 少量点时直接计算
        if (right - left <= 3) {

```

```

double minDist = std::numeric_limits<double>::max();
for (int i = left; i <= right; i++) {
    for (int j = i + 1; j <= right; j++) {
        minDist = min(minDist, distance(points[i], points[j]));
    }
}
return minDist;
}

// 分解问题
int mid = left + (right - left) / 2;
double midX = points[mid].first;

// 递归求解左右两部分的最近点对
vector<pair<double, double>> leftPointsSortedByY;
vector<pair<double, double>> rightPointsSortedByY;

for (auto& point : pointsSortedByY) {
    if (point.first <= midX) {
        leftPointsSortedByY.push_back(point);
    } else {
        rightPointsSortedByY.push_back(point);
    }
}

double leftMinDist = closestPairHelper(points, left, mid, leftPointsSortedByY);
double rightMinDist = closestPairHelper(points, mid + 1, right, rightPointsSortedByY);

// 合并结果
double minDist = min(leftMinDist, rightMinDist);

// 处理跨越中间线的点对
vector<pair<double, double>> strip;
for (auto& point : pointsSortedByY) {
    if (abs(point.first - midX) < minDist) {
        strip.push_back(point);
    }
}

// 在带状区域中寻找可能的更近点对
for (size_t i = 0; i < strip.size(); i++) {
    for (size_t j = i + 1; j < strip.size() && (strip[j].second - strip[i].second) < minDist; j++) {

```

```

        minDist = min(minDist, distance(strip[i], strip[j]));
    }

}

return minDist;
}

static double distance(const pair<double, double>& p1, const pair<double, double>& p2) {
    double dx = p1.first - p2.first;
    double dy = p1.second - p2.second;
    return std::sqrt(dx * dx + dy * dy);
}

public:
// ===== 补充题目 11: Karatsuba 大整数乘法 =====
/***
 * 题目来源: 经典算法题
 *
 * 题目描述:
 * 实现 Karatsuba 算法进行大整数乘法运算
 *
 * 解题思路:
 * 1. 将两个大整数分别拆分为高位和低位两部分
 * 2. 使用分治思想, 将一次 4 次乘法减少为 3 次乘法
 * 3. 通过巧妙的组合方式计算结果
 *
 * 时间复杂度: O(n^log23) ≈ O(n^1.585)
 * 空间复杂度: O(n)
 *
 * 是否最优解: 比传统 O(n^2) 算法更优, 但存在更优的 FFT 算法 O(n log n)
 */

static string karatsubaMultiply(const string& num1, const string& num2) {
    // 处理特殊情况
    if (num1 == "0" || num2 == "0") {
        return "0";
    }

    // 调用递归辅助函数
    return karatsubaHelper(num1, num2);
}

private:
    static string karatsubaHelper(const string& num1, const string& num2) {

```

```

// 基本情况：小数字直接计算
if (num1.length() < 10 || num2.length() < 10) {
    return multiplyStrings(num1, num2);
}

// 使两个数字长度相等，用 0 填充较短的数字
size_t n = max(num1.length(), num2.length());
size_t half = (n + 1) / 2;

// 补齐长度
string n1 = string(max(0, (int)(n - num1.length())), '0') + num1;
string n2 = string(max(0, (int)(n - num2.length())), '0') + num2;

// 将数字分为两部分
string a = n1.substr(0, n - half);
string b = n1.substr(n - half);
string c = n2.substr(0, n - half);
string d = n2.substr(n - half);

// 递归计算三个乘积
string ac = karatsubaHelper(a, c);
string bd = karatsubaHelper(b, d);
string abcd = karatsubaHelper(addStrings(a, b), addStrings(c, d));

// 计算 ad + bc = (a+b)(c+d) - ac - bd
string adPlusBc = subtractStrings(subtractStrings(abcd, ac), bd);

// 组合结果
// result = ac * 10^(2*half) + (ad+bc) * 10^half + bd
string result = addStrings(addStrings(ac + string(2 * half, '0')), adPlusBc + string(half, '0')), bd);

// 移除前导零
return removeLeadingZeros(result);
}

static string multiplyStrings(const string& num1, const string& num2) {
    vector<int> result(num1.length() + num2.length(), 0);

    for (int i = num1.length() - 1; i >= 0; i--) {
        for (int j = num2.length() - 1; j >= 0; j--) {
            int mul = (num1[i] - '0') * (num2[j] - '0');
            int p1 = i + j, p2 = i + j + 1;

```

```

        int sum = mul + result[p2];

        result[p2] = sum % 10;
        result[p1] += sum / 10;
    }
}

string sb = "";
for (int p : result) {
    if (!(sb.length() == 0 && p == 0)) {
        sb += to_string(p);
    }
}
return sb.length() == 0 ? "0" : sb;
}

static string addStrings(const string& num1, const string& num2) {
    string sb = "";
    int carry = 0;
    int i = num1.length() - 1;
    int j = num2.length() - 1;

    while (i >= 0 || j >= 0 || carry != 0) {
        int sum = carry;
        if (i >= 0) sum += num1[i--] - '0';
        if (j >= 0) sum += num2[j--] - '0';
        sb = to_string(sum % 10) + sb;
        carry = sum / 10;
    }
}

return sb;
}

static string subtractStrings(const string& num1, const string& num2) {
    string sb = "";
    int carry = 0;
    int i = num1.length() - 1;
    int j = num2.length() - 1;

    while (i >= 0 || j >= 0) {
        int digit1 = (i >= 0) ? num1[i--] - '0' : 0;
        int digit2 = (j >= 0) ? num2[j--] - '0' : 0;
        int diff = digit1 - digit2 - carry;
        if (diff < 0) {
            carry = 1;
            diff += 10;
        }
        sb = to_string(diff) + sb;
    }
}

return sb;
}

```

```

        if (diff < 0) {
            diff += 10;
            carry = 1;
        } else {
            carry = 0;
        }

        sb = to_string(diff) + sb;
    }

    return removeLeadingZeros(sb);
}

static string removeLeadingZeros(const string& str) {
    size_t i = 0;
    while (i < str.length() - 1 && str[i] == '0') {
        i++;
    }
    return str.substr(i);
}

public:
    // 测试补充题目的方法
    static void testAdditionalProblems() {
        cout << "\n===== 补充题目测试 =====\n";

        // 测试归并排序
        cout << "\n1. 归并排序测试: " << endl;
        vector<int> numsMerge = {9, 3, 7, 1, 5, 8, 2, 6, 4};
        cout << "排序前: ";
        printVector(numsMerge);
        mergeSort(numsMerge);
        cout << "排序后: ";
        printVector(numsMerge);

        // 测试二分查找
        cout << "\n2. 二分查找测试: " << endl;
        vector<int> numsBinary = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        cout << "查找 5: 索引 = " << binarySearch(numsBinary, 5) << endl;
        cout << "查找 10: 索引 = " << binarySearchOptimal(numsBinary, 10) << endl;

        // 测试快速幂
    }
}

```

```

cout << "\n3. 快速幂测试: " << endl;
cout << "2^10 = " << quickPow(2, 10) << endl;
cout << "2^-2 = " << quickPowOptimal(2, -2) << endl;

// 测试最大子矩阵和
cout << "\n4. 最大子矩阵和测试: " << endl;
vector<vector<int>> matrix = {
    {1, 2, -1, -4, -20},
    {-8, -3, 4, 2, 1},
    {3, 8, 10, 1, 3},
    {-4, -1, 1, 7, -6}
};
cout << "最大子矩阵和 = " << maxSubMatrix(matrix) << endl;

// 测试最近点对问题
cout << "\n5. 最近点对问题测试: " << endl;
vector<pair<double, double>> points = {{0, 0}, {3, 0}, {0, 4}, {1, 1}, {2, 2}};
cout << "最近点对距离 = " << closestPair(points) << endl;

// 测试 Karatsuba 大整数乘法
cout << "\n6. Karatsuba 大整数乘法测试: " << endl;
cout << "123456789 * 987654321 = " << karatsubaMultiply("123456789", "987654321") <<
endl;
cout << "0 * 12345 = " << karatsubaMultiply("0", "12345") << endl;
cout << "9999999999 * 9999999999 = " << karatsubaMultiply("9999999999", "9999999999") <<
endl;
}

private:
    static void printVector(const vector<int>& vec) {
        for (int num : vec) {
            cout << num << " ";
        }
        cout << endl;
    }
};

// 测试方法
int main() {
    cout << "===== 原始测试: 分治求数组最大值 =====" << endl;

    // 测试用例 1: 普通数组
    vector<int> arr1 = { 3, 8, 7, 6, 4, 5, 1, 2 };

```

```

cout << "数组最大值 :" << Get.MaxValue::maxValue(arr1) << endl;

// 测试用例 2: 单元素数组
vector<int> arr2 = { 42 };
cout << "单元素数组最大值 :" << Get.MaxValue::maxValue(arr2) << endl;

// 测试用例 3: 负数数组
vector<int> arr3 = { -5, -2, -8, -1 };
cout << "负数数组最大值 :" << Get.MaxValue::maxValue(arr3) << endl;

// 测试用例 4: 相同元素数组
vector<int> arr4 = { 5, 5, 5, 5 };
cout << "相同元素数组最大值 :" << Get.MaxValue::maxValue(arr4) << endl;

// 测试用例 5: 大规模数组
vector<int> arr5(10000);
for (int i = 0; i < 10000; i++) {
    arr5[i] = i;
}
cout << "大规模数组最大值 :" << Get.MaxValue::maxValue(arr5) << endl;

// 异常测试: 空数组
try {
    vector<int> arr6;
    Get.MaxValue::maxValue(arr6);
} catch (const invalid_argument& e) {
    cout << "空数组异常处理: " << e.what() << endl;
}

cout << "\n===== 题目 1 测试: LeetCode 53 最大子数组和 =====" << endl;
vector<int> nums1 = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
cout << "分治法结果: " << Get.MaxValue::maxSubArray(nums1) << endl;
cout << "最优解(Kadane)结果: " << Get.MaxValue::maxSubArrayOptimal(nums1) << endl;

vector<int> nums2 = { 5, 4, -1, 7, 8 };
cout << "测试用例 2: " << Get.MaxValue::maxSubArrayOptimal(nums2) << endl;

cout << "\n===== 题目 2 测试: LeetCode 169 多数元素 =====" << endl;
vector<int> nums3 = { 3, 2, 3 };
cout << "分治法结果: " << Get.MaxValue::majorityElement(nums3) << endl;
cout << "最优解(摩尔投票)结果: " << Get.MaxValue::majorityElementOptimal(nums3) << endl;

vector<int> nums4 = { 2, 2, 1, 1, 1, 2, 2 };

```

```

cout << "测试用例 2: " << Get.MaxValue::majorityElementOptimal(nums4) << endl;

cout << "\n===== 题目 3 测试: LeetCode 215 第 K 大元素 =====" << endl;
vector<int> nums5 = {3, 2, 1, 5, 6, 4};
cout << "第 2 大元素: " << Get.MaxValue::findKthLargest(nums5, 2) << endl;

vector<int> nums6 = {3, 2, 3, 1, 2, 4, 5, 5, 6};
cout << "第 4 大元素: " << Get.MaxValue::findKthLargest(nums6, 4) << endl;

cout << "\n===== 题目 4 测试: LeetCode 240 搜索矩阵 =====" << endl;
vector<vector<int>> matrix = {
    {1, 4, 7, 11, 15},
    {2, 5, 8, 12, 19},
    {3, 6, 9, 16, 22},
    {10, 13, 14, 17, 24},
    {18, 21, 23, 26, 30}
};
cout << "搜索 5: " << (Get.MaxValue::searchMatrix(matrix, 5) ? "true" : "false") << endl;
cout << "搜索 20: " << (Get.MaxValue::searchMatrix(matrix, 20) ? "true" : "false") << endl;

// 补充题目测试
Get.MaxValue::testAdditionalProblems();

return 0;
}
=====

文件: Get.MaxValue.java
=====

import java.util.*;

/**
 * 分治法求解数组最大值问题
 *
 * 问题描述:
 * 给定一个数组, 找出其中的最大值。
 *
 * 解法思路:
 * 使用分治法, 将数组不断二分, 直到只有一个元素时直接返回,
 * 然后比较左右两部分的最大值, 返回较大者。
 *
 * 算法特点:

```

```


```

- \* 1. 分治策略：将大问题分解为小问题
- \* 2. 递归实现：通过递归不断分解问题
- \* 3. 合并结果：比较子问题的解得到原问题的解
- \*
- \* 时间复杂度分析：
  - \*  $T(n) = 2*T(n/2) + O(1)$
  - \* 根据主定理，时间复杂度为  $O(n)$
  - \*
- \* 空间复杂度分析：
  - \* 递归调用栈的深度为  $O(\log n)$
  - \* 空间复杂度为  $O(\log n)$
  - \*
- \* 相关题目扩展：
  - \* 1. LeetCode 53. 最大子数组和（分治解法）
  - \* 2. 求解数组中最大值和最小值
  - \* 3. 求解数组中第 k 大元素
  - \* 4. 分治法求解最大子矩阵和
  - \*
- \* 工程化考量：
  - \* 1. 异常处理：检查空数组情况
  - \* 2. 边界处理：处理只有一个元素的数组
  - \* 3. 性能优化：对于小规模数据可直接遍历
  - \* 4. 可配置性：可扩展为求解任意范围内的最值
  - \*
- \* 与标准库对比：
  - \* Java 标准库中 `Collections.max()` 和 `Arrays.stream().max()` 等方法
  - \* 通常使用迭代而非递归，避免栈溢出风险
  - \*
- \* 语言特性差异：
  - \* Java：使用 `Math.max()` 函数
  - \* C++：使用 `std::max()` 函数
  - \* Python：使用内置 `max()` 函数或自定义比较
  - \*
- \* 极端场景考虑：
  - \* 1. 空数组：需要特殊处理
  - \* 2. 单元素数组：直接返回
  - \* 3. 大规模数组：可能栈溢出，需改用迭代
  - \* 4. 所有元素相同：任一元素都是最大值
  - \*
- \* 调试技巧：
  - \* 1. 打印递归调用过程中的中间结果
  - \* 2. 使用断言验证左右子数组的最值正确性
  - \* 3. 性能测试：比较不同规模数据的执行时间

```
*  
* 分治法算法思想总结:  
* 分治法(Divide and Conquer)是一种重要的算法设计范式，通过将复杂问题分解为更小、更易解决的子问题，  
* 递归地解决这些子问题，然后将子问题的解合并起来形成原问题的解。  
*  
* 分治法适合解决的问题特征：  
* 1. 问题可以被分解为若干个规模较小的相同或相似的子问题  
* 2. 子问题的解可以合并成原问题的解  
* 3. 子问题之间相互独立，不存在重叠子问题  
*  
* 分治法的三个关键步骤：  
* 1. 分解(Divide)：将原问题分解为规模较小的子问题  
* 2. 解决(Conquer)：递归地解决子问题；对于足够小的子问题，直接求解  
* 3. 合并(Combine)：将子问题的解合并成原问题的解  
*  
* 典型应用场景：  
* 1. 排序算法：归并排序、快速排序  
* 2. 搜索算法：二分查找  
* 3. 选择问题：第 k 大元素(快速选择)  
* 4. 矩阵运算：Strassen 算法  
* 5. 字符串处理：大整数乘法  
* 6. 图论：最近点对问题  
* 7. 计算几何：凸包问题  
*/  
  
public class Get.MaxValue {  
  
    /**
     * 入口方法，求数组中的最大值
     *
     * @param arr 输入数组
     * @return 数组中的最大值
     * @throws IllegalArgumentException 如果数组为空
     */
    public static int maxValue(int[] arr) {  
        // 异常处理：检查数组是否为空  
        if (arr == null || arr.length == 0) {  
            throw new IllegalArgumentException("数组不能为空");  
        }  
  
        // 调用分治方法求解  
        return f(arr, 0, arr.length - 1);  
    }  
}
```

```

/**
 * 分治法求解数组指定范围内的最大值
 *
 * @param arr 数组
 * @param l 左边界（包含）
 * @param r 右边界（包含）
 * @return 指定范围内的最大值
 */
public static int f(int[] arr, int l, int r) {
    // 基本情况：只有一个元素时直接返回
    if (l == r) {
        return arr[l];
    }

    // 分解：计算中点，将数组分为两部分
    int m = (l + r) / 2;

    // 递归求解：分别求左右两部分的最大值
    int lmax = f(arr, l, m);
    int rmax = f(arr, m + 1, r);

    // 合并：返回左右两部分最大值中的较大者
    return Math.max(lmax, rmax);
}

// ===== 题目 1：LeetCode 53. 最大子数组和（分治解法） =====
/** 
 * 题目来源: LeetCode 53. Maximum Subarray
 * 题目链接: https://leetcode.com/problems/maximum-subarray/
 * 中文链接: https://leetcode.cn/problems/maximum-subarray/
 *
 * 题目描述:
 * 给定一个整数数组 nums，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
 *
 * 示例 1:
 * 输入: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
 * 输出: 6
 * 解释: 连续子数组 [4, -1, 2, 1] 的和最大，为 6。
 *
 * 示例 2:
 * 输入: nums = [1]

```

- \* 输出: 1
- \*
- \* 示例 3:
- \* 输入: nums = [5, 4, -1, 7, 8]
- \* 输出: 23
- \*

- \* 约束条件:

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

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

- \*

- \* 解题思路 (分治法):

- \* 1. 将数组从中间分为左右两部分

- \* 2. 最大子数组和可能出现在三个位置:

- \*   - 完全在左半部分

- \*   - 完全在右半部分

- \*   - 跨越中点 (一部分在左, 一部分在右)

- \* 3. 递归求解左右两部分的最大子数组和

- \* 4. 计算跨越中点的最大子数组和:

- \*   - 从中点向左扫描, 找到最大和

- \*   - 从中点+1 向右扫描, 找到最大和

- \*   - 两者相加即为跨越中点的最大和

- \* 5. 返回三者中的最大值

- \*

- \* 时间复杂度分析:

- \*  $T(n) = 2*T(n/2) + O(n)$

- \* 根据主定理:  $a=2, b=2, f(n)=O(n)$

- \*  $\log_b(a) = \log_2(2) = 1$

- \*  $f(n) = \Theta(n^1) = \Theta(n^{\log_b(a)})$

- \* 因此  $T(n) = \Theta(n * \log n)$

- \*

- \* 空间复杂度分析:

- \* 递归调用栈深度为  $O(\log n)$

- \* 空间复杂度:  $O(\log n)$

- \*

- \* 是否最优解:

- \* 分治法的时间复杂度为  $O(n * \log n)$ , 空间复杂度为  $O(\log n)$

- \* 最优解是动态规划 (Kadane 算法), 时间复杂度  $O(n)$ , 空间复杂度  $O(1)$

- \* 分治法不是最优解, 但展示了分治思想的应用

- \*

- \* 最优解 (Kadane 算法):

- \* - 维护两个变量: 当前最大和 (curSum)、全局最大和 (maxSum)

- \* - 遍历数组, 对每个元素:

- \*    $\text{curSum} = \max(\text{nums}[i], \text{curSum} + \text{nums}[i])$

```

*   maxSum = max(maxSum, curSum)
* - 时间复杂度 O(n), 空间复杂度 O(1)
*
* 调试技巧:
* 1. 打印每次递归的区间范围和返回值
* 2. 验证跨中点的左右扫描过程
* 3. 使用小数组手动推演过程
*
* 边界场景:
* 1. 单元素数组: 直接返回该元素
* 2. 全负数数组: 返回最大的负数
* 3. 全正数数组: 返回所有元素之和
* 4. 混合正负数: 需要仔细计算
*/

```

```

public static int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("数组不能为空");
    }
    return maxSubArrayDivide(nums, 0, nums.length - 1);
}

```

```

/**
 * 分治法求解最大子数组和
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @return 最大子数组和
*/

```

```

private static int maxSubArrayDivide(int[] nums, int left, int right) {
    // 基本情况: 只有一个元素
    if (left == right) {
        return nums[left];
    }

    // 分解: 计算中点
    int mid = left + (right - left) / 2;

    // 递归求解左右两部分的最大子数组和
    int leftMax = maxSubArrayDivide(nums, left, mid);
    int rightMax = maxSubArrayDivide(nums, mid + 1, right);

    // 计算跨越中点的最大子数组和
    // 从中点向左扫描

```

```

int leftCrossMax = Integer.MIN_VALUE;
int leftSum = 0;
for (int i = mid; i >= left; i--) {
    leftSum += nums[i];
    leftCrossMax = Math.max(leftCrossMax, leftSum);
}

// 从中点+1 向右扫描
int rightCrossMax = Integer.MIN_VALUE;
int rightSum = 0;
for (int i = mid + 1; i <= right; i++) {
    rightSum += nums[i];
    rightCrossMax = Math.max(rightCrossMax, rightSum);
}

// 跨越中点的最大和
int crossMax = leftCrossMax + rightCrossMax;

// 返回三者中的最大值
return Math.max(Math.max(leftMax, rightMax), crossMax);
}

/**
 * 最优解: Kadane 算法 (动态规划)
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
public static int maxSubArrayOptimal(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("数组不能为空");
    }

    int maxSum = nums[0]; // 全局最大和
    int curSum = nums[0]; // 当前最大和

    for (int i = 1; i < nums.length; i++) {
        // 当前元素要么单独成为新的子数组, 要么加入之前的子数组
        curSum = Math.max(nums[i], curSum + nums[i]);
        // 更新全局最大和
        maxSum = Math.max(maxSum, curSum);
    }

    return maxSum;
}

```

}

```
// ===== 题目 2: LeetCode 169. 多数元素 (分治解法) =====
/***
 * 题目来源: LeetCode 169. Majority Element
 * 题目链接: https://leetcode.com/problems/majority-element/
 * 中文链接: https://leetcode.cn/problems/majority-element/
 *
 * 题目描述:
 * 给定一个大小为 n 的数组 nums，返回其中的多数元素。
 * 多数元素是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素。
 * 你可以假设数组是非空的，并且给定的数组总是存在多数元素。
 *
 * 示例 1:
 * 输入: nums = [3, 2, 3]
 * 输出: 3
 *
 * 示例 2:
 * 输入: nums = [2, 2, 1, 1, 1, 2, 2]
 * 输出: 2
 *
 * 约束条件:
 * n == nums.length
 * 1 <= n <= 5 * 10^4
 * -10^9 <= nums[i] <= 10^9
 *
 * 解题思路 (分治法):
 * 1. 如果一个元素是整个数组的多数元素，那么它必定是左半部分或右半部分的多数元素
 * 2. 递归求解左右两部分的多数元素
 * 3. 如果左右两部分的多数元素相同，直接返回
 * 4. 如果不同，需要统计两个候选元素在整个区间的出现次数，返回次数多的
 *
 * 时间复杂度分析:
 * T(n) = 2*T(n/2) + O(n) // 分治 + 统计次数
 * 根据主定理: T(n) = O(n*log n)
 *
 * 空间复杂度分析:
 * 递归调用栈深度: O(log n)
 *
 * 是否最优解:
 * 分治法时间复杂度 O(n*log n)，空间复杂度 O(log n)
 * 最优解是摩尔投票算法 (Boyer-Moore Voting Algorithm)
 * 时间复杂度 O(n)，空间复杂度 O(1)
```

```
* 分治法不是最优解
*
* 最优解（摩尔投票算法）：
* - 维护候选元素和计数器
* - 遍历数组，遇到相同元素计数+1，不同元素计数-1
* - 计数为0时更换候选元素
* - 最后的候选元素即为多数元素
*
* 其他解法：
* 1. 哈希表统计：O(n) 时间，O(n) 空间
* 2. 排序取中位数：O(n*log n) 时间，O(1) 或 O(n) 空间
* 3. 随机化算法：期望 O(n) 时间
*/
```

```
public static int majorityElement(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("数组不能为空");
    }
    return majorityElementDivide(nums, 0, nums.length - 1);
}

/**
 * 分治法求解多数元素
 */
private static int majorityElementDivide(int[] nums, int left, int right) {
    // 基本情况：只有一个元素
    if (left == right) {
        return nums[left];
    }

    // 分解：计算中点
    int mid = left + (right - left) / 2;

    // 递归求解左右两部分的多数元素
    int leftMajor = majorityElementDivide(nums, left, mid);
    int rightMajor = majorityElementDivide(nums, mid + 1, right);

    // 如果左右多数元素相同，直接返回
    if (leftMajor == rightMajor) {
        return leftMajor;
    }

    // 如果不同，统计两个候选元素在当前区间的出现次数
    int leftCount = countInRange(nums, leftMajor, left, right);
```

```
int rightCount = countInRange(nums, rightMajor, left, right);

// 返回出现次数多的元素
return leftCount > rightCount ? leftMajor : rightMajor;
}

/***
 * 统计元素在指定范围内的出现次数
 */
private static int countInRange(int[] nums, int target, int left, int right) {
    int count = 0;
    for (int i = left; i <= right; i++) {
        if (nums[i] == target) {
            count++;
        }
    }
    return count;
}

/***
 * 最优解：摩尔投票算法
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 */
public static int majorityElementOptimal(int[] nums) {
    if (nums == null || nums.length == 0) {
        throw new IllegalArgumentException("数组不能为空");
    }

    int candidate = nums[0];
    int count = 1;

    for (int i = 1; i < nums.length; i++) {
        if (count == 0) {
            candidate = nums[i];
            count = 1;
        } else if (nums[i] == candidate) {
            count++;
        } else {
            count--;
        }
    }
}
```

```
    return candidate;
}

// ===== 题目 3: LeetCode 215. 数组中的第 K 个最大元素 (快速选择 - 分治思想)
=====

/***
 * 题目来源: LeetCode 215. Kth Largest Element in an Array
 * 题目链接: https://leetcode.com/problems/kth-largest-element-in-an-array/
 * 中文链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 题目描述:
 * 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 * 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
 * 你必须设计并实现时间复杂度为 O(n) 的算法解决此问题。
 *
 * 示例 1:
 * 输入: [3, 2, 1, 5, 6, 4], k = 2
 * 输出: 5
 *
 * 示例 2:
 * 输入: [3, 2, 3, 1, 2, 4, 5, 5, 6], k = 4
 * 输出: 4
 *
 * 约束条件:
 * 1 <= k <= nums.length <= 10^5
 * -10^4 <= nums[i] <= 10^4
 *
 * 解题思路 (快速选择算法 - 基于分治):
 * 1. 快速选择是快速排序的变体, 基于分治思想
 * 2. 选择一个枢轴元素, 将数组分为三部分: 小于、等于、大于枢轴
 * 3. 根据枢轴的位置, 决定在左半部分还是右半部分继续查找
 * 4. 平均情况下只需要递归一半, 不需要对两部分都递归
 *
 * 时间复杂度分析:
 * 平均情况: T(n) = T(n/2) + O(n) = O(n)
 * 最坏情况: T(n) = T(n-1) + O(n) = O(n^2) (每次都选到最值)
 * 通过随机化选择枢轴, 可以将期望时间复杂度降到 O(n)
 *
 * 空间复杂度分析:
 * 递归调用栈: 平均 O(log n), 最坏 O(n)
 *
 * 是否最优解:
 * 快速选择的平均时间复杂度 O(n) 是最优的

```

```
* 其他解法:  
* 1. 排序后取第 k 个: O(n*log n) 时间  
* 2. 小顶堆维护 k 个元素: O(n*log k) 时间, O(k) 空间  
* 3. 大顶堆: O(n + k*log n) 时间, O(n) 空间
```

```
*
```

```
* 快速选择是最优解 (平均情况)
```

```
*
```

```
* 调试技巧:
```

```
* 1. 打印每次分区后的枢轴位置  
* 2. 验证分区后的数组是否满足左<=中<=右  
* 3. 测试极端情况: 全相同、有序、逆序
```

```
*/
```

```
public static int findKthLargest(int[] nums, int k) {  
    if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {  
        throw new IllegalArgumentException("参数非法");  
    }  
    // 第 k 大元素 = 第(n-k)小元素 (0-indexed)  
    return quickSelect(nums, 0, nums.length - 1, nums.length - k);  
}
```

```
/**
```

```
* 快速选择算法  
* @param nums 数组  
* @param left 左边界  
* @param right 右边界  
* @param k 查找第 k 小的元素 (0-indexed)  
*/
```

```
private static int quickSelect(int[] nums, int left, int right, int k) {  
    if (left == right) {  
        return nums[left];  
    }
```

```
// 随机选择枢轴, 避免最坏情况  
Random random = new Random();  
int pivotIndex = left + random.nextInt(right - left + 1);  
pivotIndex = partition(nums, left, right, pivotIndex);
```

```
// 根据枢轴位置决定在哪一半继续查找  
if (k == pivotIndex) {  
    return nums[k];  
} else if (k < pivotIndex) {  
    return quickSelect(nums, left, pivotIndex - 1, k);  
} else {
```

```

        return quickSelect(nums, pivotIndex + 1, right, k);
    }
}

/***
 * 分区操作：将数组分为小于、等于、大于枢轴三部分
 * @return 枢轴最终位置
 */
private static int partition(int[] nums, int left, int right, int pivotIndex) {
    int pivotValue = nums[pivotIndex];

    // 将枢轴移到最右边
    swap(nums, pivotIndex, right);

    // 分区：小于枢轴的放左边
    int storeIndex = left;
    for (int i = left; i < right; i++) {
        if (nums[i] < pivotValue) {
            swap(nums, storeIndex, i);
            storeIndex++;
        }
    }

    // 将枢轴放到最终位置
    swap(nums, storeIndex, right);

    return storeIndex;
}

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

// ===== 题目 4: LeetCode 240. 搜索二维矩阵 II =====
/***
 * 题目来源: LeetCode 240. Search a 2D Matrix II
 * 链接: https://leetcode.com/problems/search-a-2d-matrix-ii/
 * 中文: https://leetcode.cn/problems/search-a-2d-matrix-ii/
 *
 * 最优解: 从右上角或左下角搜索, 时间 O(m+n), 空间 O(1)
 */

```

```
public static boolean searchMatrix(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }

    // 从右上角开始搜索
    int row = 0;
    int col = matrix[0].length - 1;

    while (row < matrix.length && col >= 0) {
        if (matrix[row][col] == target) {
            return true;
        } else if (matrix[row][col] > target) {
            col--; // 当前值大于目标，向左移动
        } else {
            row++; // 当前值小于目标，向下移动
        }
    }

    return false;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
    System.out.println("===== 原始测试：分治求数组最大值 =====");
    System.out.println("===== 原始测试：分治求数组最大值 =====");
    // 测试用例 1：普通数组
    int[] arr1 = { 3, 8, 7, 6, 4, 5, 1, 2 };
    System.out.println("数组最大值：" + maxValue(arr1));

    // 测试用例 2：单元素数组
    int[] arr2 = { 42 };
    System.out.println("单元素数组最大值：" + maxValue(arr2));

    // 测试用例 3：负数数组
    int[] arr3 = { -5, -2, -8, -1 };
    System.out.println("负数数组最大值：" + maxValue(arr3));

    // 测试用例 4：相同元素数组
    int[] arr4 = { 5, 5, 5, 5 };
    System.out.println("相同元素数组最大值：" + maxValue(arr4));
}
```

```

// 测试用例 5: 大规模数组
int[] arr5 = new int[10000];
for (int i = 0; i < arr5.length; i++) {
    arr5[i] = i;
}
System.out.println("大规模数组最大值 : " + maxValue(arr5));

// 异常测试: 空数组
try {
    int[] arr6 = {};
    maxValue(arr6);
} catch (IllegalArgumentException e) {
    System.out.println("空数组异常处理: " + e.getMessage());
}

// 异常测试: null 数组
try {
    maxValue(null);
} catch (IllegalArgumentException e) {
    System.out.println("null 数组异常处理: " + e.getMessage());
}

System.out.println("\n===== 题目 1 测试: LeetCode 53 最大子数组和 =====");
int[] nums1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
System.out.println("分治法结果: " + maxSubArray(nums1));
System.out.println("最优解(Kadane)结果: " + maxSubArrayOptimal(nums1));

int[] nums2 = {5, 4, -1, 7, 8};
System.out.println("测试用例 2: " + maxSubArrayOptimal(nums2));

System.out.println("\n===== 题目 2 测试: LeetCode 169 多数元素 =====");
int[] nums3 = {3, 2, 3};
System.out.println("分治法结果: " + majorityElement(nums3));
System.out.println("最优解(摩尔投票)结果: " + majorityElementOptimal(nums3));

int[] nums4 = {2, 2, 1, 1, 1, 2, 2};
System.out.println("测试用例 2: " + majorityElementOptimal(nums4));

System.out.println("\n===== 题目 3 测试: LeetCode 215 第 K 大元素 =====");
int[] nums5 = {3, 2, 1, 5, 6, 4};
System.out.println("第 2 大元素: " + findKthLargest(nums5, 2));

```

```

int[] nums6 = {3, 2, 3, 1, 2, 4, 5, 5, 6} ;
System.out.println("第 4 大元素: " + findKthLargest(nums6, 4)) ;

System.out.println("\n===== 题目 4 测试: LeetCode 240 搜索矩阵 =====");
int[][] matrix = {
    {1, 4, 7, 11, 15},
    {2, 5, 8, 12, 19},
    {3, 6, 9, 16, 22},
    {10, 13, 14, 17, 24},
    {18, 21, 23, 26, 30}
};

System.out.println("搜索 5: " + searchMatrix(matrix, 5));
System.out.println("搜索 20: " + searchMatrix(matrix, 20));
}

// ===== 补充题目 1: 归并排序 (经典分治算法) =====
/***
 * 题目来源: 经典排序算法
 * 问题描述: 使用归并排序对数组进行排序
 *
 * 解题思路:
 * 1. 将数组分成左右两半
 * 2. 递归排序左右两半
 * 3. 合并两个有序数组
 *
 * 时间复杂度: O(n log n) - 最优、平均、最坏情况下都是
 * 空间复杂度: O(n) - 需要额外空间存储临时数组
 *
 * 是否最优解: 对于排序问题, 基于比较的排序算法无法突破 O(n log n) 的时间复杂度下限,
 * 归并排序在理论上已经是最优的。
 */
public static void mergeSort(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return;
    }

    int[] temp = new int[nums.length];
    mergeSortHelper(nums, 0, nums.length - 1, temp);
}

private static void mergeSortHelper(int[] nums, int left, int right, int[] temp) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortHelper(nums, left, mid, temp);
        mergeSortHelper(nums, mid + 1, right, temp);
        merge(nums, left, mid, right, temp);
    }
}

void merge(int[] nums, int left, int mid, int right, int[] temp) {
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (temp[i] < temp[j]) {
            nums[k] = temp[i];
            i++;
        } else {
            nums[k] = temp[j];
            j++;
        }
        k++;
    }
    while (i <= mid) {
        nums[k] = temp[i];
        i++;
        k++;
    }
    while (j <= right) {
        nums[k] = temp[j];
        j++;
        k++;
    }
}

```

```

        mergeSortHelper(nums, mid + 1, right, temp);
        merge(nums, left, mid, right, temp);
    }
}

private static void merge(int[] nums, int left, int mid, int right, int[] temp) {
    int i = left; // 左半部分起始索引
    int j = mid + 1; // 右半部分起始索引
    int k = left; // 临时数组索引

    // 比较左右两部分元素，将较小的元素放入临时数组
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) {
        temp[k++] = nums[i++];
    }

    while (j <= right) {
        temp[k++] = nums[j++];
    }

    // 将临时数组复制回原数组
    for (i = left; i <= right; i++) {
        nums[i] = temp[i];
    }
}

// ===== 补充题目 2: 二分查找（分治思想应用） =====
/***
 * 题目来源: LeetCode 704. Binary Search
 * 题目链接: https://leetcode.com/problems/binary-search/
 * 中文链接: https://leetcode.cn/problems/binary-search/
 *
 * 问题描述: 给定一个有序数组和一个目标值，找出目标值在数组中的索引。
 * 如果目标值不存在于数组中，返回-1。
 *
 * 解题思路:
 */

```

```

* 1. 将区间分为两半
* 2. 比较中间元素与目标值
* 3. 根据比较结果决定在左半区间还是右半区间继续查找
*
* 时间复杂度: O(log n)
* 空间复杂度: 递归实现 O(log n), 迭代实现 O(1)
*
* 是否最优解: 二分查找是有序数组查找问题的最优解
*/
public static int binarySearch(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    return binarySearchHelper(nums, 0, nums.length - 1, target);
}

private static int binarySearchHelper(int[] nums, int left, int right, int target) {
    if (left > right) {
        return -1;
    }

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

    if (nums[mid] == target) {
        return mid;
    } else if (nums[mid] > target) {
        return binarySearchHelper(nums, left, mid - 1, target);
    } else {
        return binarySearchHelper(nums, mid + 1, right, target);
    }
}

// 迭代版本的二分查找（最优解，空间复杂度 O(1)）
public static int binarySearchIterative(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int left = 0;
    int right = nums.length - 1;

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

```

```

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return -1;
}

// ===== 补充题目 3: 快速幂算法 (分治思想) =====
/***
 * 题目来源: LeetCode 50. Pow(x, n)
 * 题目链接: https://leetcode.com/problems/powx-n/
 * 中文链接: https://leetcode.cn/problems/powx-n/
 *
 * 问题描述: 实现 pow(x, n) 计算 x 的 n 次幂函数
 *
 * 解题思路:
 * 1. 分治: 将幂运算分解为更小的幂运算
 * 2. 如果 n 是偶数:  $x^n = x^{(n/2)} * x^{(n/2)}$ 
 * 3. 如果 n 是奇数:  $x^n = x^{(n/2)} * x^{(n/2)} * x$ 
 *
 * 时间复杂度:  $O(\log n)$ 
 * 空间复杂度: 递归实现  $O(\log n)$ , 迭代实现  $O(1)$ 
 *
 * 是否最优解: 快速幂算法是计算幂函数的最优解
 */
public static double myPow(double x, int n) {
    // 处理特殊情况
    if (n == 0) {
        return 1.0;
    }
    if (n == 1) {
        return x;
    }
    if (n == -1) {
        return 1.0 / x;
    }
}

```

```

// 处理负数幂
long N = n;
if (N < 0) {
    x = 1.0 / x;
    N = -N;
}

return powHelper(x, N);
}

private static double powHelper(double x, long n) {
    if (n == 0) {
        return 1.0;
    }

    // 分治：计算一半的幂
    double half = powHelper(x, n / 2);

    // 合并结果
    if (n % 2 == 0) {
        return half * half;
    } else {
        return half * half * x;
    }
}

// ===== 补充题目 4：最大子矩阵和（二维分治法） =====
/***
 * 题目来源：LeetCode 363. Max Sum of Rectangle No Larger Than K
 * 类似题目：二维最大子数组和
 *
 * 问题描述：给定一个二维矩阵，找出一个子矩阵，使得其元素和最大
 *
 * 解题思路（二维分治法）：
 * 1. 将矩阵按列分割成左右两部分
 * 2. 递归求解左右两部分的最大子矩阵和
 * 3. 计算跨越中间列的最大子矩阵和
 * 4. 返回三者中的最大值
 *
 * 时间复杂度：O(n^3 log n)
 * 空间复杂度：O(n^2)
 *
 * 是否最优解：对于二维最大子矩阵和，存在 O(n^3) 的动态规划解法

```

```

*/
public static int maxSubmatrixSum(int[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return 0;
    }

    int rows = matrix.length;
    int cols = matrix[0].length;

    return maxSubmatrixSumHelper(matrix, 0, cols - 1, rows);
}

private static int maxSubmatrixSumHelper(int[][] matrix, int leftCol, int rightCol, int rows)
{
    // 基本情况：只有一列
    if (leftCol == rightCol) {
        return maxColumnSum(matrix, leftCol, rows);
    }

    // 分解：计算中间列
    int midCol = leftCol + (rightCol - leftCol) / 2;

    // 递归求解左右两部分
    int leftMax = maxSubmatrixSumHelper(matrix, leftCol, midCol, rows);
    int rightMax = maxSubmatrixSumHelper(matrix, midCol + 1, rightCol, rows);

    // 计算跨越中间列的最大子矩阵和
    int crossMax = maxCrossMatrixSum(matrix, leftCol, midCol, rightCol, rows);

    // 返回三者中的最大值
    return Math.max(Math.max(leftMax, rightMax), crossMax);
}

private static int maxColumnSum(int[][] matrix, int col, int rows) {
    int maxSum = Integer.MIN_VALUE;
    int currentSum = 0;

    for (int i = 0; i < rows; i++) {
        currentSum = Math.max(matrix[i][col], currentSum + matrix[i][col]);
        maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
}

```

```

}

private static int maxCrossMatrixSum(int[][] matrix, int leftCol, int midCol, int rightCol,
int rows) {
    int maxSum = Integer.MIN_VALUE;

    // 枚举左边界
    for (int i = leftCol; i <= midCol; i++) {
        // 枚举右边界
        for (int j = midCol + 1; j <= rightCol; j++) {
            // 计算当前列范围[i, j]的列和
            int[] columnSums = new int[rows];
            for (int k = 0; k < rows; k++) {
                int sum = 0;
                for (int c = i; c <= j; c++) {
                    sum += matrix[k][c];
                }
                columnSums[k] = sum;
            }

            // 对列和应用 Kadane 算法求最大子数组和
            int currentSum = 0;
            for (int sum : columnSums) {
                currentSum = Math.max(sum, currentSum + sum);
                maxSum = Math.max(maxSum, currentSum);
            }
        }
    }

    return maxSum;
}

// ===== 补充题目 5: Strassen 矩阵乘法 (优化分治算法) =====
/***
 * 题目来源: 经典算法问题
 *
 * 问题描述: 实现 Strassen 算法计算两个 n×n 矩阵的乘积
 *
 * 解题思路:
 * 传统矩阵乘法的时间复杂度为 O(n3)，Strassen 算法通过巧妙地将矩阵乘法分解为 7 个而不是 8 个子矩阵乘法,
 * 将时间复杂度降低到约 O(n^2.807)。
 *
 */

```

```

* 算法步骤:
* 1. 将矩阵分为 4 个子矩阵
* 2. 计算 7 个中间矩阵
* 3. 用中间矩阵构造结果矩阵
*
* 时间复杂度: O(n^2.807)
* 空间复杂度: O(n^2)
*
* 是否最优解: Strassen 算法比传统矩阵乘法更优, 但存在更优的矩阵乘法算法
*/
public static int[][] strassenMatrixMultiply(int[][] A, int[][] B) {
    int n = A.length;

    // 处理 2x2 的基本情况
    if (n == 2) {
        return multiply2x2(A, B);
    }

    // 将矩阵分成 4 个子矩阵
    int newSize = n / 2;
    int[][] A11 = new int[newSize][newSize];
    int[][] A12 = new int[newSize][newSize];
    int[][] A21 = new int[newSize][newSize];
    int[][] A22 = new int[newSize][newSize];
    int[][] B11 = new int[newSize][newSize];
    int[][] B12 = new int[newSize][newSize];
    int[][] B21 = new int[newSize][newSize];
    int[][] B22 = new int[newSize][newSize];

    // 填充子矩阵
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + newSize];
            A21[i][j] = A[i + newSize][j];
            A22[i][j] = A[i + newSize][j + newSize];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + newSize];
            B21[i][j] = B[i + newSize][j];
            B22[i][j] = B[i + newSize][j + newSize];
        }
    }
}

```

```

// 计算 7 个中间矩阵
int[][] M1 = strassenMatrixMultiply(addMatrices(A11, A22), addMatrices(B11, B22));
int[][] M2 = strassenMatrixMultiply(addMatrices(A21, A22), B11);
int[][] M3 = strassenMatrixMultiply(A11, subtractMatrices(B12, B22));
int[][] M4 = strassenMatrixMultiply(A22, subtractMatrices(B21, B11));
int[][] M5 = strassenMatrixMultiply(addMatrices(A11, A12), B22);
int[][] M6 = strassenMatrixMultiply(subtractMatrices(A21, A11), addMatrices(B11, B12));
int[][] M7 = strassenMatrixMultiply(subtractMatrices(A12, A22), addMatrices(B21, B22));

// 计算结果矩阵的子矩阵
int[][] C11 = addMatrices(subtractMatrices(addMatrices(M1, M4), M5), M7);
int[][] C12 = addMatrices(M3, M5);
int[][] C21 = addMatrices(M2, M4);
int[][] C22 = addMatrices(subtractMatrices(addMatrices(M1, M3), M2), M6);

// 合并结果矩阵
int[][] result = new int[n][n];
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        result[i][j] = C11[i][j];
        result[i][j + newSize] = C12[i][j];
        result[i + newSize][j] = C21[i][j];
        result[i + newSize][j + newSize] = C22[i][j];
    }
}

return result;
}

private static int[][] multiply2x2(int[][] A, int[][] B) {
    int[][] C = new int[2][2];
    C[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0];
    C[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1];
    C[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0];
    C[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1];
    return C;
}

private static int[][] addMatrices(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {

```

```

        C[i][j] = A[i][j] + B[i][j];
    }
}

return C;
}

```

```

private static int[][] subtractMatrices(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}

```

// ===== 补充题目 6: 最近点对问题 (分治法) =====

```

/***
 * 题目来源: 经典计算几何问题
 *
 * 问题描述: 在平面上有 n 个点, 找出其中距离最近的一对点
 *
 * 解题思路:
 * 1. 按 x 坐标排序所有点
 * 2. 递归地将点集分为左右两部分
 * 3. 找出左右两部分各自的最近点对
 * 4. 考虑跨越中线的点对, 证明只需要检查中线附近的有限个点
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * 是否最优解: 该问题的最优时间复杂度为 O(n log n)
 */

```

```

public static class Point implements Comparable<Point> {
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

@Override

```

public int compareTo(Point other) {
    return Integer.compare(this.x, other.x);
}

}

public static double closestPair(Point[] points) {
    if (points == null || points.length < 2) {
        return Double.POSITIVE_INFINITY;
    }

    // 按 x 坐标排序
    Arrays.sort(points);

    // 为了按 y 坐标排序，创建一个副本
    Point[] pointsByY = points.clone();
    Arrays.sort(pointsByY, Comparator.comparingInt(p -> p.y));

    return closestPairHelper(points, 0, points.length - 1, pointsByY);
}

private static double closestPairHelper(Point[] pointsByX, int left, int right, Point[]
pointsByY) {
    // 基本情况：小规模问题直接计算
    int n = right - left + 1;
    if (n <= 3) {
        return bruteForce(pointsByX, left, right);
    }

    // 分解：找到中点
    int mid = left + (right - left) / 2;
    Point midPoint = pointsByX[mid];

    // 准备按 y 排序的左右两部分点集
    Point[] leftPointsByY = new Point[mid - left + 1];
    Point[] rightPointsByY = new Point[right - mid];
    int leftIndex = 0, rightIndex = 0;

    for (Point p : pointsByY) {
        if (p.x <= midPoint.x) {
            leftPointsByY[leftIndex++] = p;
        } else {
            rightPointsByY[rightIndex++] = p;
        }
    }
}

```

```

}

// 递归求解左右两部分
double leftMin = closestPairHelper(pointsByX, left, mid, leftPointsByY);
double rightMin = closestPairHelper(pointsByX, mid + 1, right, rightPointsByY);

// 合并: 取左右两部分的最小值
double minDist = Math.min(leftMin, rightMin);

// 处理跨越中线的点对
// 只需要考虑中线附近 2*minDist 范围内的点
List<Point> strip = new ArrayList<>();
for (Point p : pointsByY) {
    if (Math.abs(p.x - midPoint.x) < minDist) {
        strip.add(p);
    }
}

// 在 strip 中检查相邻点, 理论上最多只需要检查 6 个点
for (int i = 0; i < strip.size(); i++) {
    for (int j = i + 1; j < strip.size() && (strip.get(j).y - strip.get(i).y) < minDist;
j++) {
        double dist = distance(strip.get(i), strip.get(j));
        if (dist < minDist) {
            minDist = dist;
        }
    }
}

return minDist;
}

private static double bruteForce(Point[] points, int left, int right) {
    double minDist = Double.POSITIVE_INFINITY;
    for (int i = left; i <= right; i++) {
        for (int j = i + 1; j <= right; j++) {
            double dist = distance(points[i], points[j]);
            if (dist < minDist) {
                minDist = dist;
            }
        }
    }
    return minDist;
}

```

```
}
```

```
private static double distance(Point p1, Point p2) {  
    int dx = p1.x - p2.x;  
    int dy = p1.y - p2.y;  
    return Math.sqrt(dx * dx + dy * dy);  
}
```

```
// ===== 补充题目 7: Karatsuba 大整数乘法 =====
```

```
/**
```

```
* 题目来源: 经典算法问题
```

```
*
```

```
* 问题描述: 实现 Karatsuba 算法进行大整数乘法运算
```

```
*
```

```
* 解题思路:
```

```
* 1. 将两个大整数分别拆分为高位和低位两部分
```

```
* 2. 使用分治思想, 将一次 4 次乘法减少为 3 次乘法
```

```
* 3. 通过巧妙的组合方式计算结果
```

```
*
```

```
* 时间复杂度:  $O(n^{\log_2 3}) \approx O(n^{1.585})$ 
```

```
* 空间复杂度:  $O(n)$ 
```

```
*
```

```
* 是否最优解: 比传统  $O(n^2)$  算法更优, 但存在更优的 FFT 算法  $O(n \log n)$ 
```

```
*/
```

```
public static String karatsubaMultiply(String num1, String num2) {
```

```
    // 处理特殊情况
```

```
    if (num1.equals("0") || num2.equals("0")) {
```

```
        return "0";
```

```
}
```

```
    // 调用递归辅助函数
```

```
    return karatsubaHelper(num1, num2);
```

```
}
```

```
private static String karatsubaHelper(String num1, String num2) {
```

```
    // 基本情况: 小数字直接计算
```

```
    if (num1.length() < 10 || num2.length() < 10) {
```

```
        return multiplyStrings(num1, num2);
```

```
}
```

```
    // 使两个数字长度相等, 用 0 填充较短的数字
```

```
    int n = Math.max(num1.length(), num2.length());
```

```
    int half = (n + 1) / 2;
```

```

// 补齐长度
while (num1.length() < n) num1 = "0" + num1;
while (num2.length() < n) num2 = "0" + num2;

// 将数字分为两部分
String a = num1.substring(0, num1.length() - half);
String b = num1.substring(num1.length() - half);
String c = num2.substring(0, num2.length() - half);
String d = num2.substring(num2.length() - half);

// 递归计算三个乘积
String ac = karatsubaHelper(a, c);
String bd = karatsubaHelper(b, d);
String abcd = karatsubaHelper(addStrings(a, b), addStrings(c, d));

// 计算 ad + bc = (a+b)(c+d) - ac - bd
String adPlusBc = subtractStrings(subtractStrings(abcd, ac), bd);

// 组合结果
// result = ac * 10^(2*half) + (ad+bc) * 10^half + bd
String result = addStrings(addStrings(ac + "0".repeat(2 * half), adPlusBc +
"0".repeat(half)), bd);

// 移除前导零
return removeLeadingZeros(result);
}

private static String multiplyStrings(String num1, String num2) {
    int[] result = new int[num1.length() + num2.length()];

    for (int i = num1.length() - 1; i >= 0; i--) {
        for (int j = num2.length() - 1; j >= 0; j--) {
            int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
            int p1 = i + j, p2 = i + j + 1;
            int sum = mul + result[p2];

            result[p2] = sum % 10;
            result[p1] += sum / 10;
        }
    }

    StringBuilder sb = new StringBuilder();
    for (int i = result.length - 1; i >= 0; i--) {
        if (sb.length() == 0 && result[i] == 0) continue;
        sb.append(result[i]);
    }
    return sb.toString();
}

```

```

        for (int p : result) {
            if (!(sb.length() == 0 && p == 0)) {
                sb.append(p);
            }
        }
        return sb.length() == 0 ? "0" : sb.toString();
    }

private static String addStrings(String num1, String num2) {
    StringBuilder sb = new StringBuilder();
    int carry = 0;
    int i = num1.length() - 1;
    int j = num2.length() - 1;

    while (i >= 0 || j >= 0 || carry != 0) {
        int sum = carry;
        if (i >= 0) sum += num1.charAt(i--) - '0';
        if (j >= 0) sum += num2.charAt(j--) - '0';
        sb.append(sum % 10);
        carry = sum / 10;
    }

    return sb.reverse().toString();
}

private static String subtractStrings(String num1, String num2) {
    StringBuilder sb = new StringBuilder();
    int carry = 0;
    int i = num1.length() - 1;
    int j = num2.length() - 1;

    while (i >= 0 || j >= 0) {
        int digit1 = (i >= 0) ? num1.charAt(i--) - '0' : 0;
        int digit2 = (j >= 0) ? num2.charAt(j--) - '0' : 0;
        int diff = digit1 - digit2 - carry;

        if (diff < 0) {
            diff += 10;
            carry = 1;
        } else {
            carry = 0;
        }
    }

    return sb.reverse().toString();
}

```

```

        sb.append(diff);
    }

    String result = sb.reverse().toString();
    return removeLeadingZeros(result);
}

private static String removeLeadingZeros(String str) {
    int i = 0;
    while (i < str.length() - 1 && str.charAt(i) == '0') {
        i++;
    }
    return str.substring(i);
}

// 添加补充题目的测试方法
public static void testAdditionalProblems() {
    System.out.println("\n===== 补充题目测试 =====");

    // 测试归并排序
    System.out.println("\n1. 归并排序测试: ");
    int[] mergeArray = {9, 3, 7, 1, 5, 8, 2, 6, 4};
    mergeSort(mergeArray);
    System.out.print("排序后数组: ");
    for (int num : mergeArray) {
        System.out.print(num + " ");
    }
    System.out.println();

    // 测试二分查找
    System.out.println("\n2. 二分查找测试: ");
    int[] sortedArray = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("查找 5 的索引 (递归): " + binarySearch(sortedArray, 5));
    System.out.println("查找 5 的索引 (迭代): " + binarySearchIterative(sortedArray, 5));
    System.out.println("查找 10 的索引: " + binarySearch(sortedArray, 10));

    // 测试快速幂
    System.out.println("\n3. 快速幂测试: ");
    System.out.println("2^10 = " + myPow(2, 10));
    System.out.println("2^-2 = " + myPow(2, -2));
    System.out.println("3^5 = " + myPow(3, 5));

    // 测试最大子矩阵和 (简化测试)
}

```

```

System.out.println("\n4. 最大子矩阵和测试: ");
int[][] matrix = {
    {1, -2, 3},
    {-4, 5, -6},
    {7, -8, 9}
};
System.out.println("最大子矩阵和: " + maxSubmatrixSum(matrix));

// 测试最近点对
System.out.println("\n5. 最近点对测试: ");
Point[] points = {
    new Point(0, 0),
    new Point(3, 0),
    new Point(0, 4),
    new Point(1, 1),
    new Point(2, 2)
};
System.out.println("最近点对距离: " + closestPair(points));

// 测试 Karatsuba 大整数乘法
System.out.println("\n6. Karatsuba 大整数乘法测试: ");
System.out.println("123456789 * 987654321 = " + karatsubaMultiply("123456789",
"987654321"));
System.out.println("0 * 12345 = " + karatsubaMultiply("0", "12345"));
System.out.println("9999999999 * 9999999999 = " + karatsubaMultiply("9999999999",
"9999999999"));
}
}

```

=====

文件: Get.MaxValue.py

=====

"""分治法求解数组最大值问题 (Python 版本)

问题描述:

给定一个数组，找出其中的最大值。

解法思路:

使用分治法，将数组不断二分，直到只有一个元素时直接返回，  
然后比较左右两部分的最大值，返回较大者。

算法特点:

1. 分治策略：将大问题分解为小问题
2. 递归实现：通过递归不断分解问题
3. 合并结果：比较子问题的解得到原问题的解

时间复杂度分析：

$$T(n) = 2*T(n/2) + O(1)$$

根据主定理，时间复杂度为  $O(n)$

空间复杂度分析：

递归调用栈的深度为  $O(\log n)$

空间复杂度为  $O(\log n)$

相关题目扩展：

1. LeetCode 53. 最大子数组和（分治解法）
2. 求解数组中最大值和最小值
3. 求解数组中第  $k$  大元素
4. 分治法求解最大子矩阵和

工程化考量：

1. 异常处理：检查空数组情况
2. 边界处理：处理只有一个元素的数组
3. 性能优化：对于小规模数据可直接遍历
4. 可配置性：可扩展为求解任意范围内的最值

与标准库对比：

Python 标准库中 `max()` 函数通常使用迭代而非递归，避免栈溢出风险

语言特性差异：

Java：使用 `Math.max()` 函数

C++：使用 `std::max()` 函数

Python：使用内置 `max()` 函数或自定义比较

极端场景考虑：

1. 空数组：需要特殊处理
2. 单元素数组：直接返回
3. 大规模数组：可能栈溢出，需改用迭代
4. 所有元素相同：任一元素都是最大值

调试技巧：

1. 打印递归调用过程中的中间结果
2. 使用断言验证左右子数组的最值正确性
3. 性能测试：比较不同规模数据的执行时间

"""

```
import random
from typing import List


def max_value(arr):
    """
    入口方法，求数组中的最大值

    :param arr: 输入数组
    :return: 数组中的最大值
    :raises ValueError: 如果数组为空
    """

    # 异常处理：检查数组是否为空
    if not arr:
        raise ValueError("数组不能为空")

    # 调用分治方法求解
    return _find_max(arr, 0, len(arr) - 1)


def _find_max(arr, left, right):
    """
    分治法求解数组指定范围内的最大值

    :param arr: 数组
    :param left: 左边界（包含）
    :param right: 右边界（包含）
    :return: 指定范围内的最大值
    """

    # 基本情况：只有一个元素时直接返回
    if left == right:
        return arr[left]

    # 分解：计算中点，将数组分为两部分
    mid = (left + right) // 2

    # 递归求解：分别求左右两部分的最大值
    left_max = _find_max(arr, left, mid)
    right_max = _find_max(arr, mid + 1, right)

    # 合并：返回左右两部分最大值中的较大者
    return max(left_max, right_max)
```

```
# ===== 题目 1: LeetCode 53. 最大子数组和 (分治解法) =====
def max_sub_array(nums: List[int]) -> int:
```

```
"""
```

```
题目来源: LeetCode 53. Maximum Subarray
```

```
题目链接: https://leetcode.com/problems/maximum-subarray/
```

```
中文链接: https://leetcode.cn/problems/maximum-subarray/
```

题目描述:

给定一个整数数组 `nums`, 找到一个具有最大和的连续子数组, 返回其最大和。

示例 1:

输入: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

输出: 6

解释: 连续子数组 `[4, -1, 2, 1]` 的和最大, 为 6。

时间复杂度:  $O(n \log n)$  - 分治法

空间复杂度:  $O(\log n)$  - 递归栈

最优解: Kadane 算法, 时间  $O(n)$ , 空间  $O(1)$

```
"""
```

```
if not nums:
```

```
    raise ValueError("数组不能为空")
```

```
return _max_sub_array_divide(nums, 0, len(nums) - 1)
```

```
def _max_sub_array_divide(nums: List[int], left: int, right: int) -> int:
```

```
    """分治法求解最大子数组和"""

```

```
    if left == right:
```

```
        return nums[left]
```

```
    mid = left + (right - left) // 2
```

```
    left_max = _max_sub_array_divide(nums, left, mid)
```

```
    right_max = _max_sub_array_divide(nums, mid + 1, right)
```

```
# 计算跨越中点的最大子数组和
```

```
    left_cross_max = float('-inf')
```

```
    left_sum = 0
```

```
    for i in range(mid, left - 1, -1):
```

```
        left_sum += nums[i]
```

```
        left_cross_max = max(left_cross_max, left_sum)
```

```
right_cross_max = float('-inf')
right_sum = 0
for i in range(mid + 1, right + 1):
    right_sum += nums[i]
    right_cross_max = max(right_cross_max, right_sum)

cross_max = left_cross_max + right_cross_max
return int(max(left_max, right_max, cross_max))
```

```
def max_sub_array_optimal(nums: List[int]) -> int:
```

```
"""
```

最优解: Kadane 算法

时间复杂度: O(n)

空间复杂度: O(1)

```
"""
```

```
if not nums:
```

```
    raise ValueError("数组不能为空")
```

```
max_sum = nums[0]
```

```
cur_sum = nums[0]
```

```
for i in range(1, len(nums)):
```

```
    cur_sum = max(nums[i], cur_sum + nums[i])
```

```
    max_sum = max(max_sum, cur_sum)
```

```
return max_sum
```

```
# ===== 题目 2: LeetCode 169. 多数元素 (分治解法) =====
```

```
def majority_element(nums: List[int]) -> int:
```

```
"""
```

题目来源: LeetCode 169. Majority Element

题目链接: <https://leetcode.com/problems/majority-element/>

中文链接: <https://leetcode.cn/problems/majority-element/>

时间复杂度: O(n\*log n) - 分治法

空间复杂度: O(log n)

最优解: 摩尔投票算法, 时间 O(n), 空间 O(1)

```
"""
```

```
if not nums:
```

```
    raise ValueError("数组不能为空")
```

```
return _majority_element_divide(nums, 0, len(nums) - 1)

def _majority_element_divide(nums: List[int], left: int, right: int) -> int:
    """分治法求解多数元素"""
    if left == right:
        return nums[left]

    mid = left + (right - left) // 2
    left_major = _majority_element_divide(nums, left, mid)
    right_major = _majority_element_divide(nums, mid + 1, right)

    if left_major == right_major:
        return left_major

    left_count = sum(1 for i in range(left, right + 1) if nums[i] == left_major)
    right_count = sum(1 for i in range(left, right + 1) if nums[i] == right_major)

    return left_major if left_count > right_count else right_major
```

```
def majority_element_optimal(nums: List[int]) -> int:
```

```
"""

最优解: 摩尔投票算法
时间复杂度: O(n)
空间复杂度: O(1)
"""

if not nums:
    raise ValueError("数组不能为空")
```

```
candidate = nums[0]
```

```
count = 1
```

```
for i in range(1, len(nums)):
```

```
    if count == 0:
```

```
        candidate = nums[i]
```

```
        count = 1
```

```
    elif nums[i] == candidate:
```

```
        count += 1
```

```
    else:
```

```
        count -= 1
```

```
return candidate
```

```

# ===== 题目 3: LeetCode 215. 数组中的第 K 个最大元素 =====
def find_kth_largest(nums: List[int], k: int) -> int:
    """
    题目来源: LeetCode 215. Kth Largest Element in an Array
    题目链接: https://leetcode.com/problems/kth-largest-element-in-an-array/
    中文链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
    快速选择算法 (基于分治思想)
    平均时间复杂度: O(n)
    最坏时间复杂度: O(n^2)
    空间复杂度: O(log n)
    """

    if not nums or k < 1 or k > len(nums):
        raise ValueError("参数非法")
    return _quick_select(nums, 0, len(nums) - 1, len(nums) - k)

def _quick_select(nums: List[int], left: int, right: int, k: int) -> int:
    """快速选择算法"""
    if left == right:
        return nums[left]

    pivot_index = left + random.randint(0, right - left)
    pivot_index = _partition(nums, left, right, pivot_index)

    if k == pivot_index:
        return nums[k]
    elif k < pivot_index:
        return _quick_select(nums, left, pivot_index - 1, k)
    else:
        return _quick_select(nums, pivot_index + 1, right, k)

def _partition(nums: List[int], left: int, right: int, pivot_index: int) -> int:
    """分区操作"""
    pivot_value = nums[pivot_index]
    nums[pivot_index], nums[right] = nums[right], nums[pivot_index]

    store_index = left
    for i in range(left, right):
        if nums[i] < pivot_value:

```

```

        nums[store_index], nums[i] = nums[i], nums[store_index]
        store_index += 1

    nums[store_index], nums[right] = nums[right], nums[store_index]
    return store_index

# ===== 题目 4: LeetCode 240. 搜索二维矩阵 II =====
def search_matrix(matrix: List[List[int]], target: int) -> bool:
    """
    题目来源: LeetCode 240. Search a 2D Matrix II
    链接: https://leetcode.com/problems/search-a-2d-matrix-ii/
    中文: https://leetcode.cn/problems/search-a-2d-matrix-ii/

    最优解: 从右上角或左下角搜索
    时间复杂度: O(m+n)
    空间复杂度: O(1)
    """

    if not matrix or not matrix[0]:
        return False

    row = 0
    col = len(matrix[0]) - 1

    while row < len(matrix) and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] > target:
            col -= 1
        else:
            row += 1

    return False

def test():
    """测试方法"""
    print("===== 原始测试: 分治求数组最大值 =====")

    # 测试用例 1: 普通数组
    arr1 = [3, 8, 7, 6, 4, 5, 1, 2]
    print("数组最大值 : ", max_value(arr1))

```

```
# 测试用例 2: 单元素数组
arr2 = [42]
print("单元素数组最大值 :", max_value(arr2))

# 测试用例 3: 负数数组
arr3 = [-5, -2, -8, -1]
print("负数数组最大值 :", max_value(arr3))

# 测试用例 4: 相同元素数组
arr4 = [5, 5, 5, 5]
print("相同元素数组最大值 :", max_value(arr4))

# 测试用例 5: 大规模数组
arr5 = list(range(10000))
print("大规模数组最大值 :", max_value(arr5))

# 异常测试: 空数组
try:
    arr6 = []
    max_value(arr6)
except ValueError as e:
    print("空数组异常处理:", e)

print("\n===== 题目 1 测试: LeetCode 53 最大子数组和 =====")
nums1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("分治法结果:", max_sub_array(nums1))
print("最优解(Kadane)结果:", max_sub_array_optimal(nums1))

nums2 = [5, 4, -1, 7, 8]
print("测试用例 2:", max_sub_array_optimal(nums2))

print("\n===== 题目 2 测试: LeetCode 169 多数元素 =====")
nums3 = [3, 2, 3]
print("分治法结果:", majority_element(nums3))
print("最优解(摩尔投票)结果:", majority_element_optimal(nums3))

nums4 = [2, 2, 1, 1, 1, 2, 2]
print("测试用例 2:", majority_element_optimal(nums4))

print("\n===== 题目 3 测试: LeetCode 215 第 K 大元素 =====")
nums5 = [3, 2, 1, 5, 6, 4]
print("第 2 大元素:", find_kth_largest(nums5.copy(), 2))
```

```

nums6 = [3, 2, 3, 1, 2, 4, 5, 5, 6]
print("第 4 大元素:", find_kth_largest(nums6.copy(), 4))

print("\n===== 题目 4 测试: LeetCode 240 搜索矩阵 =====")
matrix = [
    [1, 4, 7, 11, 15],
    [2, 5, 8, 12, 19],
    [3, 6, 9, 16, 22],
    [10, 13, 14, 17, 24],
    [18, 21, 23, 26, 30]
]
print("搜索 5:", search_matrix(matrix, 5))
print("搜索 20:", search_matrix(matrix, 20))

```

# ===== 补充题目 5: 归并排序 =====

```

def merge_sort(nums):
    """
    归并排序算法实现

```

题目来源：经典排序算法

题目描述：

实现归并排序算法，将一个数组排序。

解题思路：

1. 将数组分成两半，分别排序
2. 合并两个已排序的子数组

时间复杂度：O(n log n)

空间复杂度：O(n)

参数：

nums：待排序数组

返回：

排序后的数组

```

"""
if len(nums) <= 1:
    return nums.copy() # 返回副本以避免修改原数组

mid = len(nums) // 2
left = merge_sort(nums[:mid])

```

```
right = merge_sort(nums[mid:])

return _merge(left, right)

def _merge(left, right):
    """
    合并两个已排序的数组

    参数:
        left: 左子数组
        right: 右子数组

    返回:
        合并后的有序数组
    """
    result = []
    i = j = 0

    # 合并两个数组
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # 添加剩余元素
    result.extend(left[i:])
    result.extend(right[j:])

    return result

# ===== 补充题目 6: 二分查找 =====
```

```
def binary_search(nums, target):
    """
    二分查找算法实现（递归版本）
    
```

题目来源：经典搜索算法

题目描述：

在一个排序数组中查找目标值，如果找到返回索引，否则返回-1。

解题思路：

1. 将区间不断二分，比较中间元素与目标值
2. 根据比较结果调整搜索区间

时间复杂度： $O(\log n)$

空间复杂度： $O(\log n)$  – 递归栈深度

参数：

nums: 已排序数组

target: 目标值

返回：

目标值的索引，如果不存在返回-1

"""

if not nums:

    return -1

return \_binary\_search\_helper(nums, target, 0, len(nums) - 1)

def \_binary\_search\_helper(nums, target, left, right):

"""

二分查找递归辅助函数

参数：

nums: 已排序数组

target: 目标值

left: 左边界

right: 右边界

返回：

目标值的索引，如果不存在返回-1

"""

if left > right:

    return -1

mid = left + (right - left) // 2

if nums[mid] == target:

    return mid

elif nums[mid] > target:

    return \_binary\_search\_helper(nums, target, left, mid - 1)

else:

    return \_binary\_search\_helper(nums, target, mid + 1, right)

```
def binary_search_optimal(nums, target):
    """
    二分查找最优解（迭代版本）

    时间复杂度: O(log n)
    空间复杂度: O(1)

    参数:
        nums: 已排序数组
        target: 目标值

    返回:
        目标值的索引，如果不存在返回-1
    """
    if not nums:
        return -1

    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            right = mid - 1
        else:
            left = mid + 1

    return -1

# ===== 补充题目 7: 快速幂算法 =====
def quick_pow(a, n):
    """
    快速幂算法实现（递归版本）
    """

    题目来源: 经典算法题
```

题目描述:  
计算  $a$  的  $n$  次方，要求时间复杂度优于  $O(n)$ 。

解题思路:

1. 使用分治法，将  $a^n$  分解为  $(a^{(n/2)})^2$
2. 递归计算子问题

时间复杂度:  $O(\log n)$

空间复杂度:  $O(\log n)$  - 递归栈深度

参数:

a: 底数

n: 指数

返回:

a 的 n 次方结果

"""

# 处理特殊情况

if n == 0:

return 1.0

if a == 0:

return 0.0

# 处理负数指数

if n < 0:

a = 1.0 / a

n = -n

return \_quick\_pow\_helper(a, n)

def \_quick\_pow\_helper(a, n):

"""

快速幂递归辅助函数

参数:

a: 底数

n: 非负指数

返回:

a 的 n 次方结果

"""

if n == 0:

return 1.0

half = \_quick\_pow\_helper(a, n // 2)

if n % 2 == 0:

return half \* half

```
else:  
    return half * half * a
```

```
def quick_pow_optimal(a, n):
```

```
    """
```

```
    快速幂最优解（迭代版本）
```

```
时间复杂度: O(log n)
```

```
空间复杂度: O(1)
```

```
参数:
```

```
    a: 底数
```

```
    n: 指数
```

```
返回:
```

```
    a 的 n 次方结果
```

```
    """
```

```
# 处理特殊情况
```

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

```
    return 1.0
```

```
if a == 0:
```

```
    return 0.0
```

```
# 处理负数指数
```

```
if n < 0:
```

```
    a = 1.0 / a
```

```
    n = -n
```

```
result = 1.0
```

```
current = a
```

```
while n > 0:
```

```
    if n % 2 == 1:
```

```
        result *= current
```

```
        current *= current
```

```
    n //= 2
```

```
return result
```

```
# ===== 补充题目 8: 最大子矩阵和 =====
```

```
def max_sub_matrix(matrix):
```

```
    """
```

```
    最大子矩阵和算法实现
```

题目来源：经典算法题

题目描述：

给定一个二维矩阵，找出其中和最大的子矩阵。

解题思路：

1. 将问题转化为一维最大子数组和问题
2. 枚举左右边界，计算每一行的前缀和
3. 使用 Kadane 算法求解一维最大子数组和

时间复杂度： $O(n^3)$

空间复杂度： $O(n)$

参数：

matrix: 二维矩阵

返回：

最大子矩阵和

"""

```
if not matrix or not matrix[0]:  
    raise ValueError("矩阵不能为空")  
  
rows = len(matrix)  
cols = len(matrix[0])  
max_sum = float('-inf')  
  
# 枚举左右边界  
for left in range(cols):  
    row_sum = [0] * rows # 记录每一行的和  
  
    for right in range(left, cols):  
        # 计算每一行在左右边界内的和  
        for i in range(rows):  
            row_sum[i] += matrix[i][right]  
  
        # 使用 Kadane 算法求解一维最大子数组和  
        current_sum = 0  
        current_max = float('-inf')  
  
        for num in row_sum:  
            current_sum = max(num, current_sum + num)  
            current_max = max(current_max, current_sum)
```

```

max_sum = max(max_sum, current_max)

return max_sum

# ===== 补充题目 9: Strassen 矩阵乘法 =====
def strassen_multiply(A, B):
    """
    Strassen 矩阵乘法算法实现

```

题目来源：经典算法题

题目描述：

实现 Strassen 矩阵乘法算法，优化传统的  $O(n^3)$  矩阵乘法。

解题思路：

1. 将矩阵分成四个子矩阵
2. 计算 7 个中间矩阵（而非传统算法的 8 个乘法）
3. 通过中间矩阵计算结果矩阵的四个子矩阵

时间复杂度： $O(n^{\log_2(7)}) \approx O(n^{2.81})$

空间复杂度： $O(n^2)$

参数：

A, B: 待相乘的矩阵

返回：

矩阵乘积 A\*B

"""

# 检查矩阵是否合法

```

if not A or not B or not A[0] or not B[0]:
    raise ValueError("矩阵不能为空")

```

n = len(A)

m = len(B[0])

p = len(B)

```

if len(A[0]) != p:
    raise ValueError("矩阵维度不匹配")

```

# 对于小矩阵，使用传统乘法以提高效率

if n <= 64: # 阈值可以根据实际情况调整

```

return _traditional_matrix_multiply(A, B)

```

```

# 确保矩阵是方阵且大小为 2 的幂
size = 1
while size < n:
    size <= 1

# 扩展矩阵到 2 的幂大小
A_ext = [[0] * size for _ in range(size)]
B_ext = [[0] * size for _ in range(size)]

for i in range(n):
    for j in range(n):
        A_ext[i][j] = A[i][j]

for i in range(n):
    for j in range(n):
        B_ext[i][j] = B[i][j]

# 使用 Strassen 算法计算
C_ext = _strassen_helper(A_ext, B_ext, size)

# 裁剪回原始大小
C = [[0] * n for _ in range(n)]
for i in range(n):
    for j in range(n):
        C[i][j] = C_ext[i][j]

return C

```

```

def _traditional_matrix_multiply(A, B):
    """
    传统矩阵乘法实现

```

参数:

A, B: 待相乘的矩阵

返回:

矩阵乘积 A\*B

"""

n = len(A)

m = len(B[0])

p = len(B)

```

result = [[0] * m for _ in range(n)]

for i in range(n):
    for k in range(p):
        if A[i][k] == 0:
            continue # 跳过零元素以优化性能
        for j in range(m):
            result[i][j] += A[i][k] * B[k][j]

return result

```

```

def _strassen_helper(A, B, size):
    """
    Strassen 算法递归辅助函数

```

参数:

A, B: 待相乘的矩阵 (大小为 2 的幂)  
size: 矩阵大小

返回:

矩阵乘积 A\*B

```
"""
if size == 1:
    return [[A[0][0] * B[0][0]]]
```

```
new_size = size // 2
```

# 分割矩阵

```

A11 = [[A[i][j] for j in range(new_size)] for i in range(new_size)]
A12 = [[A[i][j] for j in range(new_size, size)] for i in range(new_size)]
A21 = [[A[i][j] for j in range(new_size)] for i in range(new_size, size)]
A22 = [[A[i][j] for j in range(new_size, size)] for i in range(new_size, size)]
```

```

B11 = [[B[i][j] for j in range(new_size)] for i in range(new_size)]
B12 = [[B[i][j] for j in range(new_size, size)] for i in range(new_size)]
B21 = [[B[i][j] for j in range(new_size)] for i in range(new_size, size)]
B22 = [[B[i][j] for j in range(new_size, size)] for i in range(new_size, size)]
```

# 计算 7 个中间矩阵

```

M1 = _strassen_helper(_add_matrices(A11, A22),
                      _add_matrices(B11, B22), new_size)
M2 = _strassen_helper(_add_matrices(A21, A22), B11, new_size)
M3 = _strassen_helper(A11, _subtract_matrices(B12, B22), new_size)
```

```

M4 = _strassen_helper(A22, _subtract_matrices(B21, B11), new_size)
M5 = _strassen_helper(_add_matrices(A11, A12), B22, new_size)
M6 = _strassen_helper(_subtract_matrices(A21, A11),
                      _add_matrices(B11, B12), new_size)
M7 = _strassen_helper(_subtract_matrices(A12, A22),
                      _add_matrices(B21, B22), new_size)

# 计算结果矩阵的子矩阵
C11 = _add_matrices(_subtract_matrices(_add_matrices(M1, M4), M5), M7)
C12 = _add_matrices(M3, M5)
C21 = _add_matrices(M2, M4)
C22 = _add_matrices(_subtract_matrices(_add_matrices(M1, M3), M2), M6)

# 合并结果矩阵
C = [[0] * size for _ in range(size)]
for i in range(new_size):
    for j in range(new_size):
        C[i][j] = C11[i][j]
        C[i][j + new_size] = C12[i][j]
        C[i + new_size][j] = C21[i][j]
        C[i + new_size][j + new_size] = C22[i][j]

return C

```

def \_add\_matrices(A, B):

"""

矩阵加法

参数:

A, B: 待相加的矩阵

返回:

矩阵和 A+B

"""

n = len(A)

result = [[0] \* n for \_ in range(n)]

for i in range(n):

for j in range(n):

result[i][j] = A[i][j] + B[i][j]

return result

```
def _subtract_matrices(A, B):
```

```
    """
```

矩阵减法

参数:

A, B: 待相减的矩阵

返回:

矩阵差 A-B

```
    """
```

```
n = len(A)
```

```
result = [[0] * n for _ in range(n)]
```

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

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

```
        result[i][j] = A[i][j] - B[i][j]
```

```
return result
```

```
# ===== 补充题目 10: 最近点对问题 =====
```

```
def closest_pair(points):
```

```
    """
```

最近点对问题算法实现

题目来源: 经典算法题

题目描述:

给定平面上的 n 个点, 找出距离最近的一对点。

解题思路:

1. 按 x 坐标排序所有点
2. 使用分治法递归求解左右两部分的最近点对
3. 处理跨越中间线的最近点对

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$

参数:

points: 点集, 格式为  $[(x_1, y_1), (x_2, y_2), \dots]$

返回:

最近点对的距离

```
    """
```

```

if len(points) < 2:
    raise ValueError("至少需要两个点")

# 按 x 坐标排序
points_sorted_by_x = sorted(points)

# 创建一个按 y 坐标排序的数组，用于后续处理
points_sorted_by_y = sorted(points, key=lambda p: p[1])

return _closest_pair_helper(points_sorted_by_x, 0, len(points) - 1, points_sorted_by_y)

def _closest_pair_helper(points_sorted_by_x, left, right, points_sorted_by_y):
    """
最近点对问题递归辅助函数

```

参数:

- points\_sorted\_by\_x: 按 x 坐标排序的点集
- left: 左边界
- right: 右边界
- points\_sorted\_by\_y: 按 y 坐标排序的点集

返回:

最小距离

```

"""
# 基本情况: 少量点时直接计算
if right - left <= 3:
    return _brute_force_distance(points_sorted_by_x[left:right+1])

```

# 分解问题

```

mid = left + (right - left) // 2
mid_x = points_sorted_by_x[mid][0]

```

# 划分按 y 排序的点集

```

left_points_by_y = []
right_points_by_y = []

for p in points_sorted_by_y:
    if p[0] <= mid_x:
        left_points_by_y.append(p)
    else:
        right_points_by_y.append(p)

```

# 递归求解左右两部分

```

left_min = _closest_pair_helper(points_sorted_by_x, left, mid, left_points_by_y)
right_min = _closest_pair_helper(points_sorted_by_x, mid + 1, right, right_points_by_y)

# 合并结果
min_dist = min(left_min, right_min)

# 处理跨越中间线的点对
strip = [p for p in points_sorted_by_y if abs(p[0] - mid_x) < min_dist]

# 在带状区域中寻找可能的更近点对
for i in range(len(strip)):
    # 只需检查后面不超过 7 个点（数学上可证明）
    j = i + 1
    while j < len(strip) and strip[j][1] - strip[i][1] < min_dist:
        min_dist = min(min_dist, _distance(strip[i], strip[j]))
        j += 1

return min_dist

```

```
def _brute_force_distance(points):
    """

```

暴力计算点集中的最小距离

参数:

points: 点集

返回:

最小距离

```
"""

```

```
min_dist = float('inf')
```

```
n = len(points)
```

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

```
    for j in range(i + 1, n):
```

```
        min_dist = min(min_dist, _distance(points[i], points[j]))
```

```
return min_dist
```

```
def _distance(p1, p2):
    """

```

计算两点间的欧氏距离

参数:

p1, p2: 两个点

返回:

欧氏距离

"""

```
return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5
```

# ===== 补充题目 11: Karatsuba 大整数乘法 =====

```
def karatsuba_multiply(num1: str, num2: str) -> str:
```

"""

Karatsuba 大整数乘法算法实现

题目来源: 经典算法题

题目描述:

实现 Karatsuba 算法进行大整数乘法运算

解题思路:

1. 将两个大整数分别拆分为高位和低位两部分
2. 使用分治思想, 将一次 4 次乘法减少为 3 次乘法
3. 通过巧妙的组合方式计算结果

时间复杂度:  $O(n^{\log_2 3}) \approx O(n^{1.585})$

空间复杂度:  $O(n)$

参数:

num1, num2: 两个大整数的字符串表示

返回:

两个大整数的乘积

"""

# 处理特殊情况

```
if num1 == "0" or num2 == "0":  
    return "0"
```

# 调用递归辅助函数

```
return _karatsuba_helper(num1, num2)
```

```
def _karatsuba_helper(num1: str, num2: str) -> str:
```

"""

Karatsuba 算法递归辅助函数

参数:

```
num1, num2: 两个大整数的字符串表示
```

返回:

两个大整数的乘积

```
"""
```

```
# 基本情况: 小数字直接计算
```

```
if len(num1) < 10 or len(num2) < 10:
```

```
    return str(int(num1) * int(num2))
```

```
# 使两个数字长度相等, 用 0 填充较短的数字
```

```
n = max(len(num1), len(num2))
```

```
half = (n + 1) // 2
```

```
# 补齐长度
```

```
num1 = num1.zfill(n)
```

```
num2 = num2.zfill(n)
```

```
# 将数字分为两部分
```

```
a = num1[:n-half]
```

```
b = num1[n-half:]
```

```
c = num2[:n-half]
```

```
d = num2[n-half:]
```

```
# 递归计算三个乘积
```

```
ac = _karatsuba_helper(a, c)
```

```
bd = _karatsuba_helper(b, d)
```

```
abcd = _karatsuba_helper(str(int(a) + int(b)), str(int(c) + int(d)))
```

```
# 计算 ad + bc = (a+b)(c+d) - ac - bd
```

```
ad_plus_bc = str(int(abcd) - int(ac) - int(bd))
```

```
# 组合结果
```

```
# result = ac * 10^(2*half) + (ad+bc) * 10^half + bd
```

```
result = str(int(ac + '0' * (2 * half)) + int(ad_plus_bc + '0' * half) + int(bd))
```

```
# 移除前导零
```

```
return result.lstrip('0') or '0'
```

```
# 更新测试函数, 添加补充题目测试
```

```
def test_additional():
```

```
"""
```

```
测试方法, 包括原始题目和补充题目
```

```
"""
```

```
print("===== 原始测试: 分治求数组最大值 =====")\n\n# 测试用例 1: 普通数组\narr1 = [3, 8, 7, 6, 4, 5, 1, 2]\nprint("数组最大值 : ", max_value(arr1))\n\n# 测试用例 2: 单元素数组\narr2 = [42]\nprint("单元素数组最大值 : ", max_value(arr2))\n\n# 测试用例 3: 负数数组\narr3 = [-5, -2, -8, -1]\nprint("负数数组最大值 : ", max_value(arr3))\n\n# 测试用例 4: 相同元素数组\narr4 = [5, 5, 5, 5]\nprint("相同元素数组最大值 : ", max_value(arr4))\n\n# 测试用例 5: 大规模数组\narr5 = list(range(10000))\nprint("大规模数组最大值 : ", max_value(arr5))\n\n# 异常测试: 空数组\ntry:\n    arr6 = []\n    max_value(arr6)\nexcept ValueError as e:\n    print("空数组异常处理: ", e)\n\nprint("\n===== 题目 1 测试: LeetCode 53 最大子数组和 =====")\nnums1 = [-2, 1, -3, 4, -1, 2, 1, -5, 4]\nprint("分治法结果: ", max_sub_array(nums1))\nprint("最优解(Kadane)结果: ", max_sub_array_optimal(nums1))\n\nnums2 = [5, 4, -1, 7, 8]\nprint("测试用例 2: ", max_sub_array_optimal(nums2))\n\nprint("\n===== 题目 2 测试: LeetCode 169 多数元素 =====")\nnums3 = [3, 2, 3]\nprint("分治法结果: ", majority_element(nums3))\nprint("最优解(摩尔投票)结果: ", majority_element_optimal(nums3))\n\nnums4 = [2, 2, 1, 1, 1, 2, 2]
```

```

print("测试用例 2:", majority_element_optimal(nums4))

print("\n===== 题目 3 测试: LeetCode 215 第 K 大元素 =====")
nums5 = [3, 2, 1, 5, 6, 4]
print("第 2 大元素:", find_kth_largest(nums5.copy(), 2))

nums6 = [3, 2, 3, 1, 2, 4, 5, 5, 6]
print("第 4 大元素:", find_kth_largest(nums6.copy(), 4))

print("\n===== 题目 4 测试: LeetCode 240 搜索矩阵 =====")
matrix = [
    [1, 4, 7, 11, 15],
    [2, 5, 8, 12, 19],
    [3, 6, 9, 16, 22],
    [10, 13, 14, 17, 24],
    [18, 21, 23, 26, 30]
]
print("搜索 5:", search_matrix(matrix, 5))
print("搜索 20:", search_matrix(matrix, 20))

# 补充题目测试
print("\n===== 补充题目测试 =====\n")

# 测试归并排序
print("1. 归并排序测试:")
nums_merge = [9, 3, 7, 1, 5, 8, 2, 6, 4]
print(f"排序前: {nums_merge}")
sorted_nums = merge_sort(nums_merge)
print(f"排序后: {sorted_nums}")

# 测试二分查找
print("\n2. 二分查找测试:")
nums_binary = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(f"查找 5: 索引 = {binary_search(nums_binary, 5)}")
print(f"查找 10: 索引 = {binary_search_optimal(nums_binary, 10)}")

# 测试快速幂
print("\n3. 快速幂测试:")
print(f"2^10 = {quick_pow(2, 10)}")
print(f"2^-2 = {quick_pow_optimal(2, -2)}")

# 测试最大子矩阵和
print("\n4. 最大子矩阵和测试:")

```

```
matrix = [
    [1, 2, -1, -4, -20],
    [-8, -3, 4, 2, 1],
    [3, 8, 10, 1, 3],
    [-4, -1, 1, 7, -6]
]
print(f"最大子矩阵和 = {max_sub_matrix(matrix)}")
```

# 测试最近点对问题

```
print("\n5. 最近点对问题测试:")
points = [(0, 0), (3, 0), (0, 4), (1, 1), (2, 2)]
print(f"最近点对距离 = {closest_pair(points):.6f}")
```

# 测试 Karatsuba 大整数乘法

```
print("\n6. Karatsuba 大整数乘法测试:")
print(f"123456789 * 987654321 = {karatsuba_multiply('123456789', '987654321')}")
print(f"0 * 12345 = {karatsuba_multiply('0', '12345')}")
print(f"9999999999 * 9999999999 = {karatsuba_multiply('9999999999', '9999999999')}")
```

# 运行测试

```
if __name__ == "__main__":
    test()
print("\n" + "="*50)
print("补充题目测试:")
print("=".*50)
test_additional()
```

---