

=====

文件夹: class016_MaximumSubmatrixSum

=====

[Markdown 文件]

=====

文件: README.md

=====

class019 - 子矩阵的最大累加和及相关算法

概述

本文件夹包含与子矩阵最大累加和问题相关的多种实现方式，展示了不同的编程风格和优化技巧。所有代码都经过详细注释，便于理解和学习。

文件说明

Java 实现

1. Code01_FillFunction. java

- **类型**: 填函数风格实现
- **特点**: 只需实现核心算法逻辑，无需处理输入输出
- **适用场景**: 在线评测平台的函数式接口

2. Code02_SpecifyAmount. java

- **类型**: ACM 风格实现（指定数据规模）
- **特点**: 使用 StreamTokenizer 处理输入，PrintWriter 处理输出
- **适用场景**: 算法竞赛中需要自己处理输入输出的情况

3. Code03_StaticSpace. java

- **类型**: ACM 风格实现（静态空间优化）
- **特点**: 预分配静态空间，提高内存使用效率
- **适用场景**: 对内存使用有严格要求的场景

4. Code04_ReadByLine. java

- **类型**: 按行读取输入
- **特点**: 适用于没有明确数据规模的输入格式
- **适用场景**: 每行数据格式简单的场景

5. Code05_Kattio. java

- **类型**: Kattio 类实现
- **特点**: 可以正确处理大整数和科学计数法
- **适用场景**: 需要处理特殊数字格式的场景

6. Code06_FastReaderWriter.java

- **类型**: 快速读写类实现
- **特点**: 提供最快的输入输出方式
- **适用场景**: 大数据量处理, 对 I/O 效率要求极高的场景

Python 实现

Code01_FillFunction.py

- **类型**: Python 版本的最大子矩阵和实现
- **特点**:
 - 使用列表推导式简化代码
 - 使用 `float(' -inf')` 代替 Java 的 `Integer.MIN_VALUE`
 - 包含详细的中文注释和算法解析
 - 提供多个相关题目的实现和解析

C++实现

Code01_FillFunction.cpp

- **类型**: C++版本的最大子矩阵和实现
- **特点**:
 - 使用 `vector<vector<int>>` 代替二维数组, 更安全
 - 使用 `INT_MIN` (定义在`<climits>`) 代替 Java 的 `Integer.MIN_VALUE`
 - 包含详细的中文注释和算法解析
 - 提供多个相关题目的实现和解析

核心算法

所有实现都基于相同的算法思想:

1. **二维压缩技术**: 将二维问题转化为一维最大子数组和问题
2. **枚举上下边界**: 遍历所有可能的行区间组合
3. **列压缩**: 将选定行区间的每列元素相加, 形成一维数组
4. **Kadane 算法**: 对压缩后的一维数组求最大子数组和

时间复杂度

- $O(n^2 \times m)$, 其中 n 是行数, m 是列数

空间复杂度

- $O(m)$, 用于存储压缩后的数组

相关题目

1. LeetCode 1074. 元素和为目标值的子矩阵数量

- **链接**: <https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/>
- **解法**: 二维压缩 + 前缀和 + 哈希表
- **时间复杂度**: $O(n^2 \times m)$

2. LeetCode 363. 矩形区域不超过 K 的最大数值和

- **链接**: <https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/>
- **解法**: 二维压缩 + 前缀和 + 有序集合
- **时间复杂度**: $O(n^2 \times m \times \log m)$

3. UVA 108 Maximum Sum

- **链接**:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=44

- **解法**: 标准的最大子矩阵和问题
- **时间复杂度**: $O(n^4)$ 或优化版 $O(n^3)$

4. 洛谷 P1719 最大加权矩形

- **链接**: <https://www.luogu.com.cn/problem/P1719>
- **解法**: 与本题相同
- **时间复杂度**: $O(n^3)$

5. 牛客网 BM97 子矩阵最大和

- **链接**: <https://www.nowcoder.com/practice/840eee05dccd4ffd8f9433ce8085946b>
- **解法**: 与本题相同
- **时间复杂度**: $O(n^3)$

6. CodeChef MAXREC

- **链接**: <https://www.codechef.com/problems/MAXREC>
- **解法**: 与本题相同
- **时间复杂度**: $O(n^3)$

7. SPOJ MAXSUBM

- **链接**: <https://www.spoj.com/problems/MAXSUBM/>
- **解法**: 与本题相同
- **时间复杂度**: $O(n^3)$

8. LeetCode 152. 乘积最大子数组

- **链接**: <https://leetcode.com/problems/maximum-product-subarray/>
- **解法**: 动态规划，同时维护最大值和最小值
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$

9. LeetCode 918. 环形子数组的最大和

- ****链接****: <https://leetcode.com/problems/maximum-sum-circular-subarray/>
- ****解法****: 分两种情况讨论（正常情况和环形情况）
- ****时间复杂度****: $O(n)$
- ****空间复杂度****: $O(1)$

学习建议

1. ****理解核心思想****: 掌握二维压缩和 Kadane 算法的结合使用
2. ****比较不同实现****: 理解各种 I/O 处理方式的优缺点和适用场景
3. ****扩展练习****: 尝试实现相关题目中提到的变种问题
4. ****性能优化****: 学习静态空间分配和快速 I/O 的实现原理

编译和运行

Java 文件编译

所有 Java 文件都可以通过以下命令编译:

```
```bash
javac FileName.java
```
```

Python 文件检查

Python 文件可以通过以下命令检查语法:

```
```bash
python -m py_compile FileName.py
```
```

C++文件编译

C++文件可以通过以下命令编译:

```
```bash
g++ -std=c++11 FileName.cpp -o FileName.exe
```
```

文档说明

SUMMARY.md

- 包含所有题目的概览和学习路径建议

MAXIMUM_SUBMATRIX_SUM.md

- 详细解析最大子矩阵和问题及其变种

ADDITIONAL_PROBLEMS.md

- 包含更多相关题目和解答

README.md

- 当前文件，提供整体说明

=====

[代码文件]

=====

文件: Code01_FillFunction.cpp

=====

```
// =====  
// 题目 1: 子矩阵的最大累加和 (填函数风格 - C++版本)  
// =====  
// 题目来源: 牛客网 (NowCoder)  
// 题目链接: https://www.nowcoder.com/practice/840eee05dccc4ffd8f9433ce8085946b  
// 难度等级: 中等  
//  
// =====  
// C++实现特点  
// =====  
// 1. 使用 vector<vector<int>>代替二维数组, 更安全  
// 2. 使用 INT_MIN (定义在<climits>) 代替 Java 的 Integer.MIN_VALUE  
// 3. 使用 std::max 代替 Math.max  
// 4. C++没有内置数组越界检查, 需要自己小心  
//  
// =====  
// C++编译命令  
// =====  
// g++ -std=c++11 -O2 Code01_FillFunction.cpp -o solution  
// ./solution  
//  
// 或使用在线评测系统自动编译  
//  
// =====
```

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <climits> // INT_MIN, INT_MAX  
#include <set> // 用于 LeetCode 363  
#include <map> // 用于 LeetCode 1074  
#include <unordered_map> // 用于 Codeforces 977F  
#include <string> // 用于字符串操作  
#include <sstream> // 用于字符串流
```

```

#include <functional> // 用于 greater
#include <queue>    // 用于优先队列
#include <stack>    // 用于栈操作
using namespace std;

class Solution {
public:
    // =====
    // 主要函数: 子矩阵最大累加和 (对外接口)
    // =====
    // 这是填函数风格的对外接口, 在线评测平台会调用此函数
    //
    // 参数:
    // - mat: 输入矩阵
    // - n: 矩阵行数 (在这个版本中, n 也表示列数, 即方阵)
    //
    // 返回值:
    // - 最大子矩阵和
    //
    // 注意: C++ 版本中, 我们使用 vector<vector<int>> 以获得更好的安全性
    int sumOfSubMatrix(vector<vector<int>>& mat, int n) {
        if (mat.empty() || mat[0].empty()) return 0;

        int rows = mat.size();
        int cols = mat[0].size();
        int maxSum = INT_MIN;

        // 枚举所有可能的上下边界
        for (int top = 0; top < rows; top++) {
            vector<int> colSum(cols, 0); // 存储每列在 [top, bottom] 范围内的和

            for (int bottom = top; bottom < rows; bottom++) {
                // 更新列和
                for (int col = 0; col < cols; col++) {
                    colSum[col] += mat[bottom][col];
                }
            }

            // 对列和数组应用 Kadane 算法
            int currentSum = 0;
            for (int col = 0; col < cols; col++) {
                currentSum = max(colSum[col], currentSum + colSum[col]);
                maxSum = max(maxSum, currentSum);
            }
        }
    }
}

```

```
        }
    }

    return maxSum;
}

// =====
// 题目 2: 最大子数组乘积 (LeetCode 152)
// =====
// 题目链接: https://leetcode.com/problems/maximum-product-subarray/
// 难度: 中等
//
// 题目描述:
// 给你一个整数数组 nums, 请你找出数组中乘积最大的连续子数组 (该子数组中至少包含一个数字),
// 并返回该子数组所对应的乘积。
//
// 示例:
// 输入: [2, 3, -2, 4]
// 输出: 6
// 解释: [2, 3] 有最大乘积 6
//
// =====
// 核心算法: 动态规划 (维护最大值和最小值)
// =====
// 【算法思想】
// 由于存在负数, 最大值可能变成最小值, 最小值可能变成最大值
// 因此需要同时维护当前的最大值和最小值
//
// 【时间复杂度】O(n)
// - 只需一次遍历数组
//
// 【空间复杂度】O(1)
// - 只使用常数个变量
//
// 【是否为最优解】是的!
// - 这是解决最大子数组乘积问题的最优解法
int maxProduct(vector<int>& nums) {
    if (nums.empty()) return 0;

    int maxProd = nums[0];
    int minProd = nums[0];
    int result = nums[0];
```

```

for (int i = 1; i < nums.size(); i++) {
    // 如果当前数是负数，交换最大值和最小值
    if (nums[i] < 0) {
        swap(maxProd, minProd);
    }

    // 更新最大值和最小值
    maxProd = max(nums[i], maxProd * nums[i]);
    minProd = min(nums[i], minProd * nums[i]);

    // 更新结果
    result = max(result, maxProd);
}

return result;
}

// =====
// 题目3：环形数组的最大子数组和（LeetCode 918）
// =====
// 题目链接: https://leetcode.com/problems/maximum-sum-circular-subarray/
// 难度：中等
//
// 题目描述：
// 给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。
// 环形数组意味着数组的末端将会与开头相连呈环状。
//
// 示例：
// 输入: [5,-3,5]
// 输出: 10
// 解释: 子数组 [5,5] 有最大和 5 + 5 = 10
//
// =====
// 核心算法: Kadane 算法 + 环形数组技巧
// =====
// 【算法思想】
// 环形数组的最大子数组和有两种情况:
// 1. 正常情况: 最大子数组在数组中间 (使用普通 Kadane 算法)
// 2. 环形情况: 最大子数组跨越数组首尾 (总和 - 最小子数组和)
//
// 【时间复杂度】O(n)
// - 需要两次遍历数组
//

```

```

// 【空间复杂度】O(1)
// - 只使用常数个变量
//
// 【是否为最优解】是的!
// - 这是解决环形数组最大子数组和问题的最优解法
int maxSubarraySumCircular(vector<int>& nums) {
    if (nums.empty()) return 0;

    int total = 0;
    int maxSum = INT_MIN;
    int minSum = INT_MAX;
    int currentMax = 0;
    int currentMin = 0;

    for (int num : nums) {
        total += num;

        // 普通 Kadane 算法求最大子数组和
        currentMax = max(num, currentMax + num);
        maxSum = max(maxSum, currentMax);

        // 求最小子数组和（用于环形情况）
        currentMin = min(num, currentMin + num);
        minSum = min(minSum, currentMin);
    }

    // 如果所有数都是负数，返回最大单个元素
    if (maxSum < 0) return maxSum;

    // 返回两种情况的最大值
    return max(maxSum, total - minSum);
}

// =====
// 题目 4：和为目标值的子矩阵数量 (LeetCode 1074)
// =====
// 题目链接: https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/
// 难度：困难
//
// 题目描述：
// 给出矩阵 matrix 和目标值 target，返回元素总和等于目标值的非空子矩阵的数量。
//
// 示例：

```

```

// 输入: matrix = [[0, 1, 0], [1, 1, 1], [0, 1, 0]], target = 0
// 输出: 4
// 解释: 四个只含 0 的 1x1 子矩阵。
//
// =====
// 核心算法: 前缀和 + 哈希表
// =====
// 【算法思想】
// 1. 枚举所有可能的上下边界
// 2. 计算每列在上下边界范围内的前缀和
// 3. 使用哈希表记录前缀和出现的次数
// 4. 对于每个右边界, 查找满足条件的左边界
//
// 【时间复杂度】O(m^2 * n)
// - m 是行数, n 是列数
//
// 【空间复杂度】O(n)
// - 需要哈希表存储前缀和
//
// 【是否为最优解】是的!
// - 这是解决子矩阵和等于目标值问题的最优解法
int numSubmatrixSumTarget(vector<vector<int>>& matrix, int target) {
    if (matrix.empty() || matrix[0].empty()) return 0;

    int rows = matrix.size();
    int cols = matrix[0].size();
    int count = 0;

    // 枚举上边界
    for (int top = 0; top < rows; top++) {
        vector<int> colSum(cols, 0);

        // 枚举下边界
        for (int bottom = top; bottom < rows; bottom++) {
            // 更新列和
            for (int col = 0; col < cols; col++) {
                colSum[col] += matrix[bottom][col];
            }

            // 使用哈希表记录前缀和
            unordered_map<int, int> prefixSumCount;
            prefixSumCount[0] = 1; // 前缀和为 0 出现 1 次
            int currentSum = 0;

```

```
// 遍历所有列（相当于一维数组）
for (int col = 0; col < cols; col++) {
    currentSum += colSum[col];

    // 查找满足 currentSum - prefixSum = target 的前缀和
    if (prefixSumCount.find(currentSum - target) != prefixSumCount.end()) {
        count += prefixSumCount[currentSum - target];
    }

    // 更新前缀和出现次数
    prefixSumCount[currentSum]++;
}

}

return count;
}

// =====
// 题目 5: 最大权矩形 (洛谷 P1719)
// =====
// 题目链接: https://www.luogu.com.cn/problem/P1719
// 难度: 普及/提高-
//
// 题目描述:
// 给定一个 N*N 的矩阵, 求其最大子矩阵和。
//
// 示例:
// 输入:
// 4
// 0 -2 -7 0
// 9 2 -6 2
// -4 1 -4 1
// -1 8 0 -2
// 输出: 15
//
// =====
// 核心算法: 二维压缩 + Kadane 算法
// =====
// 【算法思想】
// 与题目 1 相同, 都是最大子矩阵和问题
//
```

```

// 【时间复杂度】O(n3)
// - 需要三重循环
//
// 【空间复杂度】O(n)
// - 需要存储列和数组
//
// 【是否为最优解】是的！
// - 对于最大子矩阵和问题，这是最优解法
int maxWeightRectangle(vector<vector<int>>& matrix) {
    return sumOfSubMatrix(matrix, matrix.size());
}

};

// =====
// 测试主函数
// =====
int main() {
    Solution solution;

    cout << "开始测试最大子矩阵和相关算法..." << endl;

    // 测试 1：最大子矩阵和
    vector<vector<int>> matrix1 = {{1, 2, 3}, {-4, 5, -6}, {7, 8, 9}};
    int result1 = solution.sumOfSubMatrix(matrix1, 3);
    cout << "测试 1 - 最大子矩阵和：" << result1 << " (期望: 27)" << endl;

    // 测试 2：最大子数组乘积
    vector<int> nums2 = {2, 3, -2, 4};
    int result2 = solution.maxProduct(nums2);
    cout << "测试 2 - 最大子数组乘积：" << result2 << " (期望: 6)" << endl;

    // 测试 3：环形数组最大子数组和
    vector<int> nums3 = {5, -3, 5};
    int result3 = solution.maxSubarraySumCircular(nums3);
    cout << "测试 3 - 环形数组最大子数组和：" << result3 << " (期望: 10)" << endl;

    // 测试 4：和为目标值的子矩阵数量
    vector<vector<int>> matrix4 = {{0, 1, 0}, {1, 1, 1}, {0, 1, 0}};
    int result4 = solution.numSubmatrixSumTarget(matrix4, 0);
    cout << "测试 4 - 和为目标值的子矩阵数量：" << result4 << " (期望: 4)" << endl;

    cout << "所有测试完成！" << endl;
}

```

```
    return 0;  
}
```

文件: Code01_FillFunction.java

```
package class019;  
  
import java.util.Map;  
import java.util.HashMap;  
  
// =====  
// 题目 1: 子矩阵的最大累加和 (填函数风格)  
// =====  
// 题目来源: 牛客网 (NowCoder)  
// 题目链接: https://www.nowcoder.com/practice/840eee05dccd4ffd8f9433ce8085946b  
// 难度等级: 中等  
//  
// =====  
// 题目描述  
// =====  
// 给定一个矩阵, 求其所有子矩阵中元素和的最大值  
//  
// 输入格式:  
// - 第一行包含两个整数 n 和 m( $1 \leq n, m \leq 100$ ), 表示矩阵的行数和列数  
// - 接下来 n 行, 每行包含 m 个整数, 表示矩阵中的元素 ( $-1000 \leq matrix[i][j] \leq 1000$ )  
//  
// 输出格式:  
// - 一个整数, 表示所有子矩阵中元素和的最大值  
//  
// 示例 1:  
// 输入:  
// 3 3  
// 1 2 3  
// -4 5 -6  
// 7 8 9  
// 输出:  
// 27  
// 解释: 选择整个矩阵, 和为  $1+2+3-4+5-6+7+8+9=27$   
//  
// =====  
// 核心算法: 二维压缩 + Kadane 算法 (动态规划)
```

```

// =====
// 【算法思想】
// 本题是经典的「最大子矩阵和」问题，核心思想是：
// 1. 将二维问题降维为一维问题
// 2. 枚举所有可能的上下边界
// 3. 将选定行范围内的每一列压缩为一个值（列和）
// 4. 在压缩后的一维数组上应用 Kadane 算法求最大子数组和
//
// 【详细步骤】
// 1. 固定上边界 i (第 i 行)
// 2. 枚举下边界 j (第 j 行, j >= i)
// 3. 对于每个(i, j)组合，计算每一列在[i, j]行范围内的和，得到一维数组 arr[]
// 4. 对 arr[] 应用 Kadane 算法，求出该行范围内的最大子矩阵和
// 5. 更新全局最大值
//
// 【Kadane 算法核心】
// - 用于求解一维数组的最大子数组和
// - 动态规划思想：当前位置的最大和 = max(当前值, 前面的最大和+当前值)
// - 简化形式：累加当前值，记录最大值，若累加和变负则重置为 0
//
// =====
// 时间复杂度分析（详细推导）
// =====
// 设矩阵规模为 n×m (n 行 m 列)
//
// 1. 枚举上边界 i: 需要 n 次循环
// 2. 对于每个 i, 枚举下边界 j: 需要(n-i)次循环，平均为 O(n)
// 3. 对于每个(i, j)，计算压缩数组：需要遍历 m 列，O(m)
// 4. 对压缩数组应用 Kadane 算法：O(m)
//
// 总时间复杂度：
// T(n, m) = Σ (i=0 to n-1) Σ (j=i to n-1) O(m)
//           = Σ (i=0 to n-1) (n-i) × O(m)
//           = O(m) × [n + (n-1) + ... + 1]
//           = O(m) × n(n+1)/2
//           = O(n² × m)
//
// 当 n=m 时，时间复杂度为 O(n³)
//
// =====
// 空间复杂度分析（详细推导）
// =====

```

```
// 1. 输入矩阵: O(n × m) - 但这是输入, 不计入额外空间
// 2. 辅助数组 arr[]: O(m) - 用于存储列压缩结果
// 3. 其他变量: O(1)
//
// 总空间复杂度: O(m)
//
// -----
// 是否为最优解?
// -----
// 【结论】是的, 这是最优解!
//
// 【理论下界分析】
// - 任何正确的算法至少需要读取所有输入: O(n × m)
// - 需要枚举所有可能的子矩阵: 共有 O(n2 × m2) 个
// - 但通过巧妙的压缩, 本算法将复杂度降至 O(n2 × m)
//
// 【是否存在更优解?】
// - 对于一般情况, 不存在低于 O(n2 × m) 的算法
// - 已有理论证明: 在比较模型下, 该问题的下界就是 Ω(n2 × m)
// - 特殊情况优化: 如果矩阵有特殊性质 (如全正、单调等), 可能有更快算法
//
// 算法适用场景与题型识别
//
// -----
// 【何时使用此算法】
// 1. 题目要求在二维矩阵中找最大/最小子矩阵和
// 2. 题目涉及矩阵区域求和问题
// 3. 题目可以转化为“固定某些维度, 对其他维度降维”的问题
//
// 【识别关键词】
// - “子矩阵” + “最大和/最小和”
// - “矩阵区域” + “求和”
// - “二维数组” + “连续子数组”
//
// 【相关题型】
// 1. LeetCode 363. Max Sum of Rectangle No Larger Than K (加了上界限制)
// 2. 最大子矩阵 (各种变形)
// 3. 前缀和 + 二维压缩的组合应用
//
// -----
// 边界情况与异常处理
//
// -----
// 【需要考虑的边界情况】
```

```
// 1. 空矩阵: n=0 或 m=0 → 应该返回 0 或报错
// 2. 单元素矩阵: n=1, m=1 → 返回该元素值
// 3. 全负数矩阵: 选择最大的单个元素
// 4. 全正数矩阵: 选择整个矩阵
// 5. 混合正负数: 需要算法正确处理
//
// 【整数溢出问题】
// - 题目限制: -1000 <= matrix[i][j] <= 1000, n, m <= 100
// - 最坏情况: 100×100×1000 = 10,000,000 (千万级)
// - int 范围: -2^31 ~ 2^31-1 (约±21亿)
// - 结论: int 类型足够, 无需使用 long
//
// =====
// 代码可读性优化建议
// =====
// 1. 变量命名: 使用有意义的名称 (如 rowStart, rowEnd 代替 i, j)
// 2. 函数拆分: 将 Kadane 算法独立成函数, 提高复用性
// 3. 注释: 关键步骤添加注释
// 4. 空行: 逻辑块之间添加空行, 增强可读性
//
// =====
// 调试技巧
// =====
// 【中间过程打印】
// - 打印每次压缩后的数组 arr[]
// - 打印每次 Kadane 算法的结果
// - 打印当前的(i, j)边界
//
// 【断言验证】
// - 验证压缩数组长度是否为 m
// - 验证最大值是否持续更新
//
// 【小数据测试】
// - 先用 2×2 矩阵手动计算验证
// - 逐步扩大到 3×3, 4×4
//
// =====
// 填函数风格说明
// =====
// 【什么是填函数风格?】
// - 在线评测平台提供一个函数签名
// - 你只需要实现函数体内的逻辑
// - 不需要处理输入输出, 测试框架会自动调用你的函数
```

```
//  
// 【优点】  
// - 专注于算法逻辑本身  
// - 不用担心 I/O 格式问题  
// - 代码简洁，易于测试  
  
//  
// 【缺点】  
// - 不适合真实的 ACM 竞赛（需要自己处理 I/O）  
// - 无法练习 I/O 优化技巧  
  
//  
// ======  
// 与其他 I/O 风格的对比  
// ======  
// 1. 填函数风格（本文件）：只写核心逻辑，平台提供测试框架  
// 2. ACM 风格-动态分配（Code02）：每组数据动态创建空间  
// 3. ACM 风格-静态空间（Code03）：预分配最大空间，复用  
  
// 推荐顺序：先掌握填函数风格 → 再学习 ACM 风格  
  
//  
// ======  
// 工程化考量  
// ======  
// 【从代码片段到可复用组件】  
// 1. 将 maxSumSubmatrix 封装为工具类  
// 2. 提供多种输入形式（二维数组、List<List<Integer>> 等）  
// 3. 添加参数校验（null 检查、维度检查）  
  
//  
// 【异常抛出】  
// - 输入为 null → 抛出 IllegalArgumentException  
// - 维度不一致 → 抛出 IllegalArgumentException  
  
//  
// 【单元测试】  
// - 测试全正数矩阵  
// - 测试全负数矩阵  
// - 测试混合矩阵  
// - 测试边界情况 ( $1 \times 1$ ,  $1 \times n$ ,  $n \times 1$ )  
  
//  
// 【性能优化】  
// - 对于稀疏矩阵，可以使用稀疏表示  
// - 对于大规模矩阵，可以考虑并行化（枚举上下边界可以并行）  
  
//  
// ======  
// 跨语言实现注意事项
```

```
// =====
// 【Java】
// - 数组越界会抛出 ArrayIndexOutOfBoundsException
// - Integer.MIN_VALUE 用于初始化最大值
//
// 【C++】
// - 使用 vector<vector<int>>更安全
// - INT_MIN 定义在<climits>
// - 注意内存管理（使用 vector 自动管理）
//
// 【Python】
// - 使用 float('-inf') 初始化最大值
// - 列表推导式可以简化代码
// - 注意缩进（Python 语法要求）
//
// =====
// 常见错误与避坑指南
// =====
// 【错误 1】忘记重置辅助数组
// - 问题：每次计算新的(i, j)前，arr[]还保留上次的值
// - 解决：在 j 循环开始时初始化 arr[]为全 0
//
// 【错误 2】Kadane 算法实现错误
// - 问题：没有正确处理全负数情况
// - 解决：max 初始化为 Integer.MIN_VALUE，而不是 0
//
// 【错误 3】边界条件处理不当
// - 问题：i <= j 的边界条件写错
// - 解决：仔细检查循环条件
//
// =====
// 扩展阅读与相关知识
// =====
// 【相关算法】
// 1. Kadane 算法（一维最大子数组和）
// 2. 前缀和（快速计算区间和）
// 3. 分治法（另一种求最大子数组和的方法）
//
// 【数学基础】
// 1. 动态规划原理
// 2. 降维思想
// 3. 组合数学（子矩阵个数计算）
//
```

```
// 【机器学习相关】
// - 图像处理中的区域特征提取
// - 卷积神经网络中的池化操作
// - 感兴趣区域(ROI)的特征计算
//
// =====
public class Code01_FillFunction {

    // =====
    // 主要函数: 子矩阵最大累加和 (对外接口)
    // =====
    // 这是填函数风格的对外接口, 在线评测平台会调用此函数
    //
    // 参数:
    // - mat: 输入矩阵
    // - n: 矩阵行数 (在这个版本中, n 也表示列数, 即方阵)
    //
    // 返回值:
    // - 最大子矩阵和
    public int sumOfSubMatrix(int[][] mat, int n) {
        return maxSumSubmatrix(mat, n, n);
    }

    // =====
    // 核心函数: 求子矩阵的最大累加和
    // =====
    // 算法思路:
    // 1. 枚举所有可能的上边界 (i)
    // 2. 对于每个上边界 i, 枚举下边界 (j), j >= i
    // 3. 将第 i 行到第 j 行的每一列相加, 形成一个一维数组
    // 4. 对这个一维数组求最大子数组和 (Kadane 算法)
    // 5. 记录所有情况下的最大值
    //
    // 参数:
    // - mat: 输入矩阵
    // - n: 行数
    // - m: 列数
    //
    // 返回值:
    // - 最大子矩阵和
    //
    // 时间复杂度: O(n2 × m)
    // - 外层循环: 枚举上边界 i, 共 n 次
```

```

// - 中层循环：枚举下边界 j，平均 n 次
// - 内层循环 1：更新压缩数组，m 次
// - 内层循环 2：Kadane 算法，m 次
// - 总计：n × n × m = O(n2m)
//
// 空间复杂度：O(m)
// 辅助数组 arr[]：O(m)
// 其他变量：O(1)
//
// 是否为最优解：是的！
// 这是经典的最优解，无法在一般情况下继续优化
// 已有理论证明：基于比较模型，该问题的下界为 Ω(n2m)
public static int maxSumSubmatrix(int[][] mat, int n, int m) {
    // 初始化最大值为最小整数，以应对全负数矩阵
    int max = Integer.MIN_VALUE;

    // 外层循环：枚举上边界（起始行）
    for (int i = 0; i < n; i++) {
        // 辅助数组，用于存储列压缩结果
        // arr[k] 表示第 k 列在 [i, j] 行范围内的和
        int[] arr = new int[m];

        // 中层循环：枚举下边界（结束行）
        for (int j = i; j < n; j++) {
            // 将第 j 行的每一列加入压缩数组
            // 这样 arr[k] 就代表 [i, j] 行范围内第 k 列的总和
            for (int k = 0; k < m; k++) {
                arr[k] += mat[j][k];
            }

            // 对压缩后的一维数组应用 Kadane 算法
            // 找到当前行范围 [i, j] 内的最大子矩阵和
            max = Math.max(max, maxSumSubarray(arr, m));
        }
    }

    return max;
}

// =====
// 辅助函数：求子数组的最大累加和（Kadane 算法）
// =====
// 这是经典的 Kadane 算法（卡德内算法），用于解决一维数组的最大子数组和问题

```

```
//  
// 【算法原理】  
// 动态规划思想:  
// - 状态定义: cur = 以当前位置结尾的最大子数组和  
// - 状态转移:  
//   - 如果 cur > 0, 那么将当前元素加入子数组  
//   - 如果 cur <= 0, 那么从当前元素重新开始  
// - 简化形式: cur = cur < 0 ? arr[i] : cur + arr[i]  
// - 更简化: cur += arr[i]; if(cur < 0) cur = 0;  
//  
// 【算法步骤】  
// 1. 初始化: max = Integer.MIN_VALUE, cur = 0  
// 2. 遍历数组:  
//   a. cur += arr[i] (将当前元素加入)  
//   b. max = Math.max(max, cur) (更新全局最大值)  
//   c. if(cur < 0) cur = 0 (若变负, 重新开始)  
// 3. 返回 max  
//  
// 【为什么这样做是对的?】  
// - 如果当前 cur > 0, 说明之前的子数组对后续有贡献, 应该保留  
// - 如果 cur <= 0, 之前的子数组只会拖累, 应该从下一个元素重新开始  
// - 这是贪心+动态规划的完美结合  
//  
// 参数:  
// - arr: 一维数组  
// - m: 数组长度  
//  
// 返回值:  
// - 最大子数组和  
//  
// 时间复杂度: O(m)  
// - 只需一次遍历  
//  
// 空间复杂度: O(1)  
// - 只需要两个变量: max 和 cur  
//  
// 是否为最优解: 是的!  
// - O(m)时间复杂度已经是最优 (必须遍历每个元素)  
// - O(1)空间复杂度已经是最优 (只需常数空间)  
//  
// 【注意事项】  
// 1. 必须将 max 初始化为 Integer.MIN_VALUE, 而不是 0  
//   - 原因: 如果所有元素都是负数, max=0 会导致错误
```

```

//      - 例如: [-3, -1, -2], 正确答案是-1, 但如果 max=0 会返回 0
// 2. cur 初始化为 0 是正确的
//      - 因为第一个元素加入后 cur=arr[0], 然后判断是否重置
// 3. 必须先更新 max, 再判断 cur 是否重置
//      - 如果顺序颠倒, 会漏掉某些情况
public static int maxSumSubarray(int[] arr, int m) {
    // 初始化最大值为最小整数, 以处理全负数数组
    int max = Integer.MIN_VALUE;

    // cur 表示以当前位置结尾的最大子数组和
    int cur = 0;

    // 遍历数组
    for (int i = 0; i < m; i++) {
        // 将当前元素加入子数组
        cur += arr[i];

        // 更新全局最大值 (注意: 必须在重置 cur 之前更新)
        max = Math.max(max, cur);

        // 如果当前 cur 变为负数, 从下一个元素重新开始
        // 因为负数只会拖累后续的累加
        cur = cur < 0 ? 0 : cur;
    }

    return max;
}

}

// -----
// 题目 2: LeetCode 363. 矩形区域不超过 K 的最大数值和
// -----
// 题目来源: LeetCode (力扣)
// 题目链接: https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/
// 难度等级: 困难
// 
// -----
// 题目描述
// -----
// 给定一个非空二维矩阵 matrix 和一个整数 k, 找到这个矩阵中矩形区域的不超过 k 的最大数值和。
// 
// 输入:

```

```

// - matrix: 二维整数数组
// - k: 整数
//
// 输出:
// - 不超过 k 的最大矩形区域和
//
// 示例:
// 输入: matrix = [[1, 0, 1], [0, -2, 3]], k = 2
// 输出: 2
// 解释: 矩形区域 [[0,-2],[1,1]] 的数值和是 2, 且 2 是不超过 k 的最大数字 (k = 2)。
//
// =====
// 核心算法: 二维压缩 + Kadane 算法 + 前缀和 + TreeSet
// =====

// 【算法思想】
// 1. 基础思想仍然是二维压缩, 枚举上下边界
// 2. 对于压缩后的一维数组, 需要找到子数组和不超过 k 的最大值
// 3. 使用前缀和 + TreeSet (有序集合) 来高效查询
//
// 【时间复杂度】 $O(n^2 \times m \times \log m)$ 
// - 枚举上下边界:  $O(n^2)$ 
// - 每次压缩数组:  $O(m)$ 
// - 使用 TreeSet 查询:  $O(m \times \log m)$ 
//
// 【空间复杂度】 $O(m)$ 
// - 压缩数组:  $O(m)$ 
// - TreeSet 存储前缀和:  $O(m)$ 
//
// =====

class LeetCode363_MaxSumSubmatrixNoLargerThanK {
    // 主函数: 求不超过 k 的最大矩形和
    public int maxSumSubmatrix(int[][] matrix, int k) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int n = matrix.length;      // 行数
        int m = matrix[0].length; // 列数
        int max = Integer.MIN_VALUE;

        // 枚举上边界
        for (int i = 0; i < n; i++) {
            int[] rowSum = new int[m]; // 存储压缩后的行和

```

```

// 枚举下边界
for (int j = i; j < n; j++) {
    // 更新压缩后的行和
    for (int l = 0; l < m; l++) {
        rowSum[l] += matrix[j][l];
    }

    // 在压缩后的一维数组中找不超过 k 的最大子数组和
    max = Math.max(max, findMaxSubarrayNoLargerThanK(rowSum, k));

    // 如果已经找到等于 k 的解，直接返回
    if (max == k) {
        return k;
    }
}

return max;
}

// 辅助函数：在一维数组中找不超过 k 的最大子数组和
private int findMaxSubarrayNoLargerThanK(int[] arr, int k) {
    int maxSum = Integer.MIN_VALUE;
    int prefixSum = 0;

    // 使用 TreeSet 存储前缀和，支持快速查找
    java.util.TreeSet<Integer> prefixSums = new java.util.TreeSet<>();
    prefixSums.add(0); // 初始前缀和为 0

    for (int num : arr) {
        prefixSum += num;

        // 寻找 prefixSum - x <= k → x >= prefixSum - k
        // 寻找最小的 x >= (prefixSum - k)
        Integer ceiling = prefixSums.ceiling(prefixSum - k);
        if (ceiling != null) {
            maxSum = Math.max(maxSum, prefixSum - ceiling);
        }

        // 将当前前缀和加入集合
        prefixSums.add(prefixSum);
    }
}

```

```
    return maxSum;
}

}

// =====
// 题目 3: LeetCode 1074. 元素和为目标值的子矩阵数量
// =====
// 题目来源: LeetCode (力扣)
// 题目链接: https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/
// 难度等级: 困难
//
// =====
// 题目描述
// =====
// 给出矩阵 matrix 和目标值 target，返回元素总和等于目标值的非空子矩阵的数量。
// 子矩阵 x1, y1, x2, y2 是满足  $x_1 \leq x \leq x_2$  且  $y_1 \leq y \leq y_2$  的所有单元 (x, y) 的集合。
// 如果  $(x_1, y_1, x_2, y_2)$  和  $(x'_1, y'_1, x'_2, y'_2)$  两个子矩阵中部分坐标不同 (如:  $x_1 \neq x'_1$ )，那么它们被视为不同的子矩阵。
//
// 输入:
// - matrix: 二维整数数组
// - target: 整数
//
// 输出:
// - 元素和等于 target 的子矩阵数量
//
// 示例:
// 输入: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
// 输出: 4
// 解释: 四个只包含 0 的 1x1 子矩阵和两个 1x3 子矩阵，以及两个 3x1 子矩阵。
//
// =====
// 核心算法: 二维压缩 + 前缀和 + 哈希表
// =====
// 【算法思想】
// 1. 基础思想仍然是二维压缩，枚举上下边界
// 2. 对于压缩后的一维数组，计算前缀和
// 3. 使用哈希表统计前缀和出现的次数，快速查找符合条件的子数组数量
//
// 【时间复杂度】 $O(n^2 \times m)$ 
// - 枚举上下边界:  $O(n^2)$ 
// - 每次压缩数组和哈希表查询:  $O(m)$ 
```

```

// 【空间复杂度】O(m)
// - 压缩数组: O(m)
// - 哈希表: O(m)
//
// =====

class LeetCode1074_SubmatrixSumEqualsTarget {
    // 主函数: 计算和为 target 的子矩阵数量
    public int numSubmatrixSumTarget(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int n = matrix.length;      // 行数
        int m = matrix[0].length; // 列数
        int count = 0;

        // 枚举上边界
        for (int i = 0; i < n; i++) {
            int[] rowSum = new int[m]; // 存储压缩后的行和

            // 枚举下边界
            for (int j = i; j < n; j++) {
                // 更新压缩后的行和
                for (int l = 0; l < m; l++) {
                    rowSum[l] += matrix[j][l];
                }
            }

            // 在压缩后的一维数组中计算和为 target 的子数组数量
            count += findSubarraySum(rowSum, target);
        }
    }

    return count;
}

// 辅助函数: 计算一维数组中和为 target 的子数组数量
private int findSubarraySum(int[] arr, int target) {
    java.util.HashMap<Integer, Integer> prefixSumCount = new java.util.HashMap<>();
    prefixSumCount.put(0, 1); // 初始前缀和为 0, 出现一次

    int prefixSum = 0;
    int count = 0;

```

```
for (int num : arr) {
    prefixSum += num;

    // 查找是否存在前缀和为 (prefixSum - target)
    if (prefixSumCount.containsKey(prefixSum - target)) {
        count += prefixSumCount.get(prefixSum - target);
    }

    // 更新当前前缀和的出现次数
    prefixSumCount.put(prefixSum, prefixSumCount.getOrDefault(prefixSum, 0) + 1);
}

return count;
}

}

// =====
// 题目 4: 洛谷 P1719 最大加权矩阵
// =====
// 题目来源: 洛谷 (Luogu)
// 题目链接: https://www.luogu.com.cn/problem/P1719
// 难度等级: 普及+/提高
//
// =====
// 题目描述
// =====
// 为了更好地备战 NOIP, 电脑组的几个女孩子 Yjq 和 Hgq 正在努力训练。他们各自编写了一个程序, 现在需要测试哪一个程序的效率更高。
// 测试内容是, 给定一个包含  $N \times N$  个元素的矩阵, 每个元素有一个权值 (可能为负数), 找出一个子矩阵, 使得这个子矩阵内的所有元素的和最大。
//
// 输入:
// - 第一行包含一个整数  $N$  ( $1 \leq N \leq 120$ )
// - 接下来  $N$  行, 每行包含  $N$  个整数, 表示矩阵中的元素
//
// 输出:
// - 一个整数, 表示最大子矩阵和
//
// =====
// 核心算法: 二维压缩 + Kadane 算法
// =====
// 【算法思想】
```

```

// 与题目 1 完全相同，标准的最大子矩阵和问题
// 适用于 ACM 静态空间模式
//
// 【时间复杂度】O(n3)
// 【空间复杂度】O(n)
//
// -----
class LuoguP1719_MaxWeightRectangle {
    private static final int MAXN = 125; // 题目限制 N≤120
    private static int[][] mat = new int[MAXN][MAXN];
    private static int[] arr = new int[MAXN];
    private static int n;

    public static int solve() {
        int max = Integer.MIN_VALUE;

        // 枚举上边界
        for (int i = 0; i < n; i++) {
            // 重置辅助数组
            java.util.Arrays.fill(arr, 0, n, 0);

            // 枚举下边界
            for (int j = i; j < n; j++) {
                // 压缩列
                for (int k = 0; k < n; k++) {
                    arr[k] += mat[j][k];
                }
            }

            // 应用 Kadane 算法
            max = Math.max(max, kadane());
        }
    }

    return max;
}

private static int kadane() {
    int max = Integer.MIN_VALUE;
    int cur = 0;

    for (int i = 0; i < n; i++) {
        cur += arr[i];
        max = Math.max(max, cur);
    }
}

```

```
        cur = cur < 0 ? 0 : cur;
    }

    return max;
}

// =====
// 题目 5: 牛客网 BM97 子矩阵最大和
// =====
// 题目来源: 牛客网 (NowCoder)
// 题目链接: https://www.nowcoder.com/practice/840eee05dccd4ffd8f9433ce8085946b
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个 n*m 的矩阵, 求其最大子矩阵和
//
// 输入:
// - 输入为一个 n*m 的二维数组
//
// 输出:
// - 输出最大子矩阵和
//
// =====
// 核心算法: 二维压缩 + Kadane 算法
// =====
// 与题目 1 完全相同, 是最大子矩阵和的标准题目
//
// =====
class NowCoder_BM97_MaxSumSubmatrix {
    public int maxsumofSubmatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int n = matrix.length;
        int m = matrix[0].length;
        int max = Integer.MIN_VALUE;

        for (int i = 0; i < n; i++) {
            int[] arr = new int[m];

```

```

        for (int j = i; j < n; j++) {
            for (int k = 0; k < m; k++) {
                arr[k] += matrix[j][k];
            }
            max = Math.max(max, maxSubarray(arr, m));
        }
    }

    return max;
}

private int maxSubarray(int[] arr, int m) {
    int max = Integer.MIN_VALUE;
    int cur = 0;

    for (int i = 0; i < m; i++) {
        cur += arr[i];
        max = Math.max(max, cur);
        cur = cur < 0 ? 0 : cur;
    }

    return max;
}

}

// =====
// 题目 6: CodeChef - MAXREC
// =====
// 题目来源: CodeChef
// 题目链接: https://www.codechef.com/problems/MAXREC
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个  $N \times M$  的整数矩阵, 找出一个子矩阵, 使得其中元素的总和最大。
//
// 输入格式:
// - 第一行包含两个整数 N 和 M
// - 接下来 N 行, 每行包含 M 个整数
//
// 输出格式:
// - 最大子矩阵和

```

```

// =====
// =====
// 核心算法：二维压缩 + Kadane 算法
// =====
// 与前面的题目相同，标准的最大子矩阵和问题
// =====
// =====

class CodeChef_MAXREC {

    public static int solve(int[][] matrix, int n, int m) {
        int max = Integer.MIN_VALUE;

        for (int i = 0; i < n; i++) {
            int[] arr = new int[m];
            for (int j = i; j < n; j++) {
                for (int k = 0; k < m; k++) {
                    arr[k] += matrix[j][k];
                }
                max = Math.max(max, kadane(arr, m));
            }
        }

        return max;
    }

    private static int kadane(int[] arr, int m) {
        int max = Integer.MIN_VALUE;
        int cur = 0;

        for (int i = 0; i < m; i++) {
            cur += arr[i];
            max = Math.max(max, cur);
            cur = cur < 0 ? 0 : cur;
        }

        return max;
    }
}

// =====
// 题目 7: SPOJ - MAXSUBM
// =====
// 题目来源: SPOJ
// 题目链接: https://www.spoj.com/problems/MAXSUBM/

```

```
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个二维数组，求其子矩阵的最大和。
//
// 输入格式:
// - 第一行包含两个整数 R 和 C (行数和列数)
// - 接下来 R 行，每行包含 C 个整数
//
// 输出格式:
// - 最大子矩阵和
//
// =====
// 核心算法: 二维压缩 + Kadane 算法
// =====
// 与前面的题目相同，标准的最大子矩阵和问题
//
// =====
class SPOJ_MAXSUBM {
    public static int solve(int[][] matrix, int r, int c) {
        int max = Integer.MIN_VALUE;

        for (int i = 0; i < r; i++) {
            int[] arr = new int[c];
            for (int j = i; j < r; j++) {
                for (int k = 0; k < c; k++) {
                    arr[k] += matrix[j][k];
                }
                max = Math.max(max, kadane(arr, c));
            }
        }
    }

    return max;
}

private static int kadane(int[] arr, int c) {
    int max = Integer.MIN_VALUE;
    int cur = 0;

    for (int i = 0; i < c; i++) {
        cur += arr[i];
        if (cur < max) {
            max = cur;
        }
    }
    return max;
}
```

```
    max = Math.max(max, cur);
    cur = cur < 0 ? 0 : cur;
}

return max;
}

}

// =====
// 题目 8: LeetCode 152. 乘积最大子数组
// =====
// 题目来源: LeetCode (力扣)
// 题目链接: https://leetcode.com/problems/maximum-product-subarray/
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给你一个整数数组 nums，请你找出数组中乘积最大的非空连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。
//
// 测试用例的答案是一个 32-位 整数。
//
// 示例 1:
// 输入: nums = [2, 3, -2, 4]
// 输出: 6
// 解释: 子数组 [2, 3] 有最大乘积 6。
//
// 示例 2:
// 输入: nums = [-2, 0, -1]
// 输出: 0
// 解释: 结果不能为 2, 因为 [-2, -1] 不是子数组。
//
// =====
// 核心算法: 动态规划 (Kadane 算法变种)
// =====
// 【算法思想】
// 由于乘积可能由负数×负数得到正数，因此需要同时维护最大值和最小值
// 1. 维护三个变量: maxProd (全局最大值)、curMax (当前最大值)、curMin (当前最小值)
// 2. 对于每个元素，计算三种可能: 当前元素本身、当前元素×curMax、当前元素×curMin
// 3. 更新 curMax 和 curMin，然后更新全局最大值
//
// 【时间复杂度】O(n)
```

```
// - 只需一次遍历数组
//
// 【空间复杂度】O(1)
// - 只需要常数空间存储变量
//
// 【是否为最优解】是的！
// - 这是该问题的最优解法，无法继续优化
// =====
class LeetCode152_MaximumProductSubarray {
    // 主函数：求乘积最大子数组
    public int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxProd = nums[0];      // 全局最大值
        int curMax = nums[0];      // 当前最大值
        int curMin = nums[0];      // 当前最小值

        for (int i = 1; i < nums.length; i++) {
            // 由于负数×负数可能得到正数，需要同时考虑三种情况
            int tempMax = curMax;
            curMax = Math.max(Math.max(nums[i], nums[i] * curMax), nums[i] * curMin);
            curMin = Math.min(Math.min(nums[i], nums[i] * tempMax), nums[i] * curMin);

            maxProd = Math.max(maxProd, curMax);
        }

        return maxProd;
    }
}

// =====
// 题目 9: LeetCode 918. 环形子数组的最大和
// =====
// 题目来源: LeetCode (力扣)
// 题目链接: https://leetcode.com/problems/maximum-sum-circular-subarray/
// 难度等级: 中等
//
// =====
// 题目描述
//
// =====
// 给定一个由整数数组组成的环形数组 nums，请返回该环形数组中的最大子数组和。
```

```
//  
// 环形数组意味着数组的末端将会与开头相连呈环状。  
// 形式上，nums[i] 的下一个元素是 nums[(i + 1) % n]，nums[i] 的前一个元素是 nums[(i - 1 + n) % n]。  
  
//  
// 子数组最多只能包含固定缓冲区 nums 中的每个元素一次。  
// 形式上，对于子数组 nums[i], nums[i+1], ..., nums[j]，不存在 i <= k1, k2 <= j 其中 k1 % n == k2 % n。  
  
//  
// 示例 1:  
// 输入: nums = [1,-2,3,-2]  
// 输出: 3  
// 解释: 从子数组 [3] 得到最大和 3  
  
//  
// 示例 2:  
// 输入: nums = [5,-3,5]  
// 输出: 10  
// 解释: 从子数组 [5,5] 得到最大和 5 + 5 = 10  
  
//  
// ======  
// 核心算法: Kadane 算法 + 环形数组处理技巧  
// ======  
  
// 【算法思想】  
// 环形数组的最大子数组和有两种情况:  
// 1. 情况一: 最大子数组在数组中间 (非环形) → 标准 Kadane 算法  
// 2. 情况二: 最大子数组跨越数组首尾 (环形) → 总和 - 最小子数组和  
  
//  
// 特殊情况: 如果数组全为负数, 则直接返回最大元素  
  
//  
// 【时间复杂度】O(n)  
// - 需要两次遍历数组  
  
//  
// 【空间复杂度】O(1)  
// - 只需要常数空间  
  
//  
// 【是否为最优解】是的!  
// - 这是环形数组最大子数组和的最优解法  
// ======  
  
class LeetCode918_MaximumSumCircularSubarray {  
    // 主函数: 求环形数组的最大子数组和  
    public int maxSubarraySumCircular(int[] nums) {  
        if (nums == null || nums.length == 0) {  
            return 0;  
        }  
        int n = nums.length;  
        int[] dp = new int[n];  
        int[] minDp = new int[n];  
        int maxSum = Integer.MIN_VALUE;  
        int minSum = 0;  
        int totalSum = 0;  
        for (int i = 0; i < n; i++) {  
            totalSum += nums[i];  
            if (i == 0) {  
                dp[i] = nums[0];  
                minDp[i] = nums[0];  
            } else {  
                dp[i] = Math.max(dp[i - 1] + nums[i], nums[i]);  
                minDp[i] = Math.min(minDp[i - 1] + nums[i], nums[i]);  
            }  
            maxSum = Math.max(maxSum, dp[i]);  
        }  
        int result = maxSum;  
        for (int i = n - 1; i >= 0; i--) {  
            dp[i] -= nums[i];  
            minDp[i] -= nums[i];  
            if (dp[i] < minDp[i]) {  
                dp[i] = minDp[i];  
            }  
            result = Math.max(result, dp[i]);  
        }  
        if (result == 0) {  
            return maxSum;  
        }  
        return result;  
    }  
}
```

```

}

int n = nums.length;
int maxKadane = kadane(nums); // 情况一：标准最大子数组和

// 如果最大子数组和为负数，说明整个数组都是负数
if (maxKadane < 0) {
    return maxKadane;
}

// 计算数组总和
int totalSum = 0;
for (int num : nums) {
    totalSum += num;
}

// 情况二：环形情况的最大和 = 总和 - 最小子数组和
int minKadane = minKadane(nums);
int maxCircular = totalSum - minKadane;

// 返回两种情况的最大值
return Math.max(maxKadane, maxCircular);
}

// 标准 Kadane 算法：求最大子数组和
private int kadane(int[] nums) {
    int max = Integer.MIN_VALUE;
    int cur = 0;

    for (int num : nums) {
        cur += num;
        max = Math.max(max, cur);
        cur = cur < 0 ? 0 : cur;
    }

    return max;
}

// 求最小子数组和（Kadane 算法变种）
private int minKadane(int[] nums) {
    int min = Integer.MAX_VALUE;
    int cur = 0;
}

```

```

        for (int num : nums) {
            cur += num;
            min = Math.min(min, cur);
            cur = cur > 0 ? 0 : cur;
        }

        return min;
    }
}

// =====
// 题目 10: HDU 1559 - 最大子矩阵 (二维前缀和版本)
// =====
// 题目来源: 杭电 OJ (HDU)
// 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1559
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个大小为 N×M 的矩阵, 以及两个整数 x 和 y, 要求找出一个大小为 x×y 的子矩阵, 使得该子矩阵的元素和最大。
//
// 输入格式:
// - 第一行: 测试用例数 T
// - 每个测试用例:
//   * 第一行: 四个整数 N, M, x, y
//   * 接下来 N 行: 每行 M 个整数
//
// 输出格式:
// - 对于每个测试用例, 输出最大子矩阵和
//
// =====
// 核心算法: 二维前缀和 + 滑动窗口
//
// 【算法思想】
// 1. 计算二维前缀和数组 prefixSum[i][j], 表示从(0, 0)到(i, j)的矩形区域和
// 2. 对于每个可能的子矩阵起始位置(i, j), 计算大小为 x×y 的子矩阵和
// 3. 使用前缀和公式: sum = prefixSum[i+x][j+y] - prefixSum[i][j+y] - prefixSum[i+x][j] + prefixSum[i][j]
// 4. 记录所有子矩阵中的最大值
//
// 【时间复杂度】O(N×M)

```

```

// - 需要计算前缀和: O(N×M)
// - 需要遍历所有可能的子矩阵: O((N-x+1) × (M-y+1)) ≈ O(N×M)
//
// 【空间复杂度】O(N×M)
// - 需要存储前缀和数组
//
// 【是否为最优解】是的!
// - 对于固定大小的子矩阵问题, 这是最优解法
// =====

class HDU1559_MaximumSubmatrixFixedSize {
    // 主函数: 求固定大小的最大子矩阵和
    public int maxSumSubmatrixFixedSize(int[][] matrix, int x, int y) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int n = matrix.length;
        int m = matrix[0].length;

        // 计算二维前缀和
        int[][] prefixSum = new int[n + 1][m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                prefixSum[i][j] = matrix[i - 1][j - 1]
                    + prefixSum[i - 1][j]
                    + prefixSum[i][j - 1]
                    - prefixSum[i - 1][j - 1];
            }
        }
    }

    int maxSum = Integer.MIN_VALUE;

    // 遍历所有可能的 x×y 子矩阵
    for (int i = 0; i <= n - x; i++) {
        for (int j = 0; j <= m - y; j++) {
            // 计算子矩阵和: 使用前缀和公式
            int sum = prefixSum[i + x][j + y]
                - prefixSum[i][j + y]
                - prefixSum[i + x][j]
                + prefixSum[i][j];

            maxSum = Math.max(maxSum, sum);
        }
    }
}

```

```
    }

    return maxSum;
}

}

// =====
// 题目 11: POJ 2479 - Maximum sum (最大两段子段和)
// =====
// 题目来源: POJ (北大 OJ)
// 题目链接: http://poj.org/problem?id=2479
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个整数数组，将其分成两个不相交的连续子数组，使得两个子数组的和最大。
// 注意: 两个子数组不能重叠，但可以相邻。
//
// 输入格式:
// - 第一行: 测试用例数 T
// - 每个测试用例:
//   * 第一行: 整数 n
//   * 第二行: n 个整数
//
// 输出格式:
// - 对于每个测试用例，输出最大两段子段和
//
// =====
// 核心算法: 前后缀分解 + Kadane 算法
// =====
// 【算法思想】
// 1. 从左到右计算每个位置的最大前缀和 (Kadane 算法)
// 2. 从右到左计算每个位置的最大后缀和 (Kadane 算法)
// 3. 对于每个分割点 i，计算 maxPrefix[i] + maxSuffix[i+1] 的最大值
//
// 【时间复杂度】O(n)
// - 需要三次遍历数组
//
// 【空间复杂度】O(n)
// - 需要存储前缀最大值和后缀最大值数组
//
// 【是否为最优解】是的!
```

```
// - 这是最大两段子段和问题的最优解法
// =====

class POJ2479_MaximumTwoSegmentSum {
    // 主函数: 求最大两段子段和
    public int maxTwoSegmentSum(int[] nums) {
        if (nums == null || nums.length < 2) {
            return 0;
        }

        int n = nums.length;

        // 从左到右计算最大前缀和
        int[] maxPrefix = new int[n];
        int cur = 0;
        int max = Integer.MIN_VALUE;

        for (int i = 0; i < n; i++) {
            cur += nums[i];
            max = Math.max(max, cur);
            maxPrefix[i] = max;
            cur = cur < 0 ? 0 : cur;
        }

        // 从右到左计算最大后缀和
        int[] maxSuffix = new int[n];
        cur = 0;
        max = Integer.MIN_VALUE;

        for (int i = n - 1; i >= 0; i--) {
            cur += nums[i];
            max = Math.max(max, cur);
            maxSuffix[i] = max;
            cur = cur < 0 ? 0 : cur;
        }

        // 计算最大两段和
        int result = Integer.MIN_VALUE;
        for (int i = 0; i < n - 1; i++) {
            result = Math.max(result, maxPrefix[i] + maxSuffix[i + 1]);
        }

        return result;
    }
}
```

```
}

// =====
// 题目 12: Codeforces 977F - Consecutive Subsequence (最长连续递增子序列)
// =====
// 题目来源: Codeforces
// 题目链接: https://codeforces.com/problemset/problem/977/F
// 难度等级: 中等
//
// =====
// 题目描述
// =====
// 给定一个整数数组，找出最长的连续递增子序列（每个元素比前一个元素大 1）。
// 输出该子序列的长度和起始位置。
//
// 示例:
// 输入: [3, 3, 4, 7, 5, 6, 8]
// 输出: 4 (子序列[3, 4, 5, 6]或[5, 6, 7, 8])
//
// =====
// 核心算法: 动态规划 + 哈希表
// =====
// 【算法思想】
// 1. 使用哈希表记录每个数字对应的最长连续序列长度
// 2. 对于每个数字 num，检查 num-1 是否在哈希表中
// 3. 如果存在，则 dp[num] = dp[num-1] + 1
// 4. 否则，dp[num] = 1
// 5. 记录最大长度和对应的数字
//
// 【时间复杂度】O(n)
// - 只需一次遍历数组
//
// 【空间复杂度】O(n)
// - 需要哈希表存储状态
//
// 【是否为最优解】是的！
// - 这是最长连续递增子序列的最优解法
//
class Codeforces977F_ConsecutiveSubsequence {
    // 主函数: 求最长连续递增子序列
    public int[] findLongestConsecutiveSubsequence(int[] nums) {
        if (nums == null || nums.length == 0) {
            return new int[0];
        }
        Map<Integer, Integer> dp = new HashMap<>();
        int maxLen = 0;
        int start = 0;
        for (int i = 0; i < nums.length; i++) {
            if (!dp.containsKey(nums[i])) {
                dp.put(nums[i], 1);
            } else {
                dp.put(nums[i], dp.get(nums[i]) + 1);
            }
            if (dp.get(nums[i]) > maxLen) {
                maxLen = dp.get(nums[i]);
                start = i - maxLen + 1;
            }
        }
        int[] result = new int[maxLen];
        for (int j = start; j < start + maxLen; j++) {
            result[j - start] = nums[j];
        }
        return result;
    }
}
```

```

}

Map<Integer, Integer> dp = new HashMap<>();
int maxLength = 0;
int endValue = 0;

for (int num : nums) {
    // 如果 num-1 存在，则当前序列长度 = dp[num-1] + 1
    // 否则当前序列长度 = 1
    dp.put(num, dp.getOrDefault(num - 1, 0) + 1);

    // 更新最大长度
    if (dp.get(num) > maxLength) {
        maxLength = dp.get(num);
        endValue = num;
    }
}

// 重构序列
int[] result = new int[maxLength];
int current = endValue;
for (int i = maxLength - 1; i >= 0; i--) {
    result[i] = current;
    current--;
}

return result;
}

// 简化版本：只返回长度
public int findLongestConsecutiveSubsequenceLength(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    Map<Integer, Integer> dp = new HashMap<>();
    int maxLength = 0;

    for (int num : nums) {
        dp.put(num, dp.getOrDefault(num - 1, 0) + 1);
        maxLength = Math.max(maxLength, dp.get(num));
    }
}

```

```
        return maxLength;
    }
}
```

文件: Code01_FillFunction.py

```
# =====
# 题目 1: 子矩阵的最大累加和 (填函数风格 - Python 版本)
# =====
# 题目来源: 牛客网 (NowCoder)
# 题目链接: https://www.nowcoder.com/practice/840eee05dccd4ffd8f9433ce8085946b
# 难度等级: 中等
#
# =====
# Python 实现特点
# =====
# 1. 使用列表推导式简化代码
# 2. 使用 float('inf') 代替 Java 的 Integer.MIN_VALUE
# 3. 使用内置 max() 函数
# 4. Python 没有类型声明, 代码更简洁但需注意类型
#
# =====
# 算法核心: 二维压缩 + Kadane 算法
# =====
#
# 时间复杂度: O(n2 × m)
# 空间复杂度: O(m)
# 是否最优解: 是
#
# =====
```

```
class Solution:
    """子矩阵最大累加和求解器"""

    def sumOfSubMatrix(self, mat, n):
        """
        主函数: 求子矩阵的最大累加和
```

参数:

```
mat: List[List[int]] - 输入矩阵  
n: int - 矩阵维度（方阵）
```

返回:

```
int - 最大子矩阵和
```

时间复杂度: $O(n^3)$ 当 $n=m$ 时

空间复杂度: $O(n)$

"""

```
return self.maxSumSubmatrix(mat, n, n)
```

```
def maxSumSubmatrix(self, mat, n, m):
```

"""

核心函数: 求最大子矩阵和

算法思路:

1. 枚举所有可能的上边界(i)
2. 对于每个上边界 i, 枚举下边界(j), $j \geq i$
3. 将第 i 行到第 j 行的每一列相加, 形成一个一维数组
4. 对这个一维数组求最大子数组和 (Kadane 算法)
5. 记录所有情况下的最大值

参数:

```
mat: List[List[int]] - 输入矩阵  
n: int - 行数  
m: int - 列数
```

返回:

```
int - 最大子矩阵和
```

时间复杂度: $O(n^2 \times m)$

- 外层循环: 枚举上边界 i, 共 n 次
- 中层循环: 枚举下边界 j, 平均 n 次
- 内层循环 1: 更新压缩数组, m 次
- 内层循环 2: Kadane 算法, m 次
- 总计: $n \times n \times m = O(n^2 m)$

空间复杂度: $O(m)$

- 辅助数组 arr: $O(m)$
- 其他变量: $O(1)$

是否为最优解：是的！

- 这是经典的最优解，无法在一般情况下继续优化
- 已有理论证明：基于比较模型，该问题的下界为 $\Omega(n^2 m)$

"""

初始化最大值为负无穷，以应对全负数矩阵

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

外层循环：枚举上边界（起始行）

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

辅助数组，用于存储列压缩结果

arr[k] 表示第 k 列在 [i, j] 行范围内的和

```
arr = [0] * m
```

中层循环：枚举下边界（结束行）

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

将第 j 行的每一列加入压缩数组

这样 arr[k] 就代表 [i, j] 行范围内第 k 列的总和

```
for k in range(m):
```

```
    arr[k] += mat[j][k]
```

对压缩后的一维数组应用 Kadane 算法

找到当前行范围 [i, j] 内的最大子矩阵和

```
max_sum = max(max_sum, self.maxSumSubarray(arr, m))
```

```
return max_sum
```

```
def maxSumSubarray(self, arr, m):
```

"""

Kadane 算法：求一维数组的最大子数组和

这是经典的 Kadane 算法（卡德内算法），用于解决一维数组的最大子数组和问题

算法原理：

动态规划思想：

- 状态定义：cur = 以当前位置结尾的最大子数组和
- 状态转移：
 - 如果 cur > 0，那么将当前元素加入子数组
 - 如果 cur <= 0，那么从当前元素重新开始
- 简化形式：cur = cur + arr[i] if cur > 0 else arr[i]
- 更简化：cur += arr[i]; if cur < 0: cur = 0

算法步骤：

1. 初始化：max_sum = float('-inf'), cur = 0

2. 遍历数组:
 - a. $cur += arr[i]$ (将当前元素加入)
 - b. $max_sum = \max(max_sum, cur)$ (更新全局最大值)
 - c. if $cur < 0$: $cur = 0$ (若变负, 重新开始)
3. 返回 max_sum

为什么这样做是对的?

- 如果当前 $cur > 0$, 说明之前的子数组对后续有贡献, 应该保留
- 如果 $cur \leq 0$, 之前的子数组只会拖累, 应该从下一个元素重新开始
- 这是贪心+动态规划的完美结合

参数:

$arr: List[int]$ - 一维数组
 $m: int$ - 数组长度

返回:

int - 最大子数组和

时间复杂度: $O(m)$ - 只需一次遍历

空间复杂度: $O(1)$ - 只需要两个变量

是否为最优解: 是的!

- $O(m)$ 时间复杂度已经是最优 (必须遍历每个元素)
- $O(1)$ 空间复杂度已经是最优 (只需常数空间)

注意事项:

1. 必须将 max_sum 初始化为 $\text{float}('-\infty')$, 而不是 0
 - 原因: 如果所有元素都是负数, $max_sum=0$ 会导致错误
 - 例如: $[-3, -1, -2]$, 正确答案是 -1, 但如果 $max_sum=0$ 会返回 0
2. cur 初始化为 0 是正确的
 - 因为第一个元素加入后 $cur=arr[0]$, 然后判断是否重置
3. 必须先更新 max_sum , 再判断 cur 是否重置
 - 如果顺序颠倒, 会漏掉某些情况

"""

```
# 初始化最大值为负无穷, 以处理全负数数组
```

```
max_sum = float('-\infty')
```

```
# cur 表示以当前位置结尾的最大子数组和
```

```
cur = 0
```

```
# 遍历数组
```

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

```
    # 将当前元素加入子数组
```

```

    cur += arr[i]

    # 更新全局最大值（注意：必须在重置 cur 之前更新）
    max_sum = max(max_sum, cur)

    # 如果当前 cur 变为负数，从下一个元素重新开始
    # 因为负数只会拖累后续的累加
    if cur < 0:
        cur = 0

    return max_sum

```

```

# =====
# Python 语言特性说明
# =====

# 1. 类型提示 (Type Hints)
#   - Python 3.5+支持类型提示，增强代码可读性
#   - from typing import List
#   - def sumOfSubMatrix(self, mat: List[List[int]], n: int) -> int:
#
# 2. 列表 vs 数组
#   - Python 列表：灵活，自动扩容，但效率略低
#   - NumPy 数组：固定大小，效率高，适合大规模数值计算
#
# 3. float('-inf') vs Integer.MIN_VALUE
#   - Python: float('-inf') 负无穷
#   - Java: Integer.MIN_VALUE
#
# 4. 内置函数
#   - Python: max(), min() 内置函数
#   - Java: Math.max(), Math.min()
#
# =====
# 性能优化建议
# =====

# 1. 使用 NumPy 加速（适合大规模数据）
#   import numpy as np
#   mat = np.array(mat)
#
# 2. 使用列表推导式代替循环（Python 风格）
#   arr = [sum(mat[r][c] for r in range(i, j+1)) for c in range(m)]
#

```

```
# 3. 使用 PyPy 解释器 (比 CPython 快 2-5 倍)
#
# 4. 使用 Cython 编译为 C 代码
#
# =====
# 测试用例
# =====
# 测试用例 1 (全正数):
# 输入: [[1, 2], [3, 4]]
# 输出: 10
#
# 测试用例 2 (全负数):
# 输入: [[-1, -2], [-3, -4]]
# 输出: -1
#
# 测试用例 3 (混合):
# 输入: [[1, 2, 3], [-4, 5, -6], [7, 8, 9]]
# 输出: 27
#
# =====

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

    # 测试用例 1: 全正数
    mat1 = [[1, 2], [3, 4]]
    result1 = solution.sumOfSubMatrix(mat1, 2)
    print(f"Test 1 (全正数): {result1}") # 期望: 10 (整个矩阵)
    assert result1 == 10

    # 测试用例 2: 全负数
    mat2 = [[-1, -2], [-3, -4]]
    result2 = solution.sumOfSubMatrix(mat2, 2)
    print(f"Test 2 (全负数): {result2}") # 期望: -1 (最大单元素)
    assert result2 == -1

    # 测试用例 3: 混合
    mat3 = [[1, 2, 3], [-4, 5, -6], [7, 8, 9]]
    result3 = solution.sumOfSubMatrix(mat3, 3)
    # 最优解: 选择所有行[0-2], 所有列[0-2]
    # 压缩数组: [1-4+7, 2+5+8, 3-6+9] = [4, 15, 6]
    # Kadane: 4+15+6 = 25
```

```
print(f"Test 3 (混合正负): {result3}") # 期望: 25
assert result3 == 25
```

```
# 测试用例 4: 包含零
mat4 = [[0, -2, -7], [9, 2, -6], [2, 4, 5]]
result4 = solution.sumOfSubMatrix(mat4, 3)
print(f"Test 4 (包含零): {result4}")
```

```
print("\n✓ 所有测试通过!")
```

```
# =====
# 题目 2: LeetCode 363. 矩形区域不超过 K 的最大数值和
# =====
# 题目来源: LeetCode (力扣)
# 题目链接: https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/
# 难度等级: 困难
#
# 【算法思路】
# 1. 二维压缩: 枚举所有可能的上下边界, 将矩阵压缩为一维数组
# 2. 前缀和 + 有序集合: 在压缩后的一维数组中, 使用前缀和和有序集合查找不超过 k 的最大值
#
# 【时间复杂度】O(n2 × m × log m)
# - 枚举上下边界: O(n2)
# - 前缀和计算: O(m)
# - 有序集合查找: O(m × log m)
#
# 【空间复杂度】O(m)
# - 压缩数组: O(m)
# - 有序集合: O(m)
#
# 【是否为最优解】是的, 对于这个问题的一般情况, 这是最优解。
# =====
```

```
class LeetCode363:
    """求矩形区域不超过 K 的最大数值和"""

    def maxSumSubmatrix(self, matrix, k):
```

主函数: 求不超过 k 的最大矩形和

参数:

matrix: List[List[int]] - 输入矩阵
k: int - 目标值

返回：

```
int - 不超过 k 的最大矩形和
"""
if not matrix or not matrix[0]:
    return 0

n = len(matrix)      # 行数
m = len(matrix[0])   # 列数
max_val = float('-inf')

# 枚举上边界
for i in range(n):
    row_sum = [0] * m  # 存储压缩后的行和

    # 枚举下边界
    for j in range(i, n):
        # 更新压缩后的行和
        for l in range(m):
            row_sum[l] += matrix[j][l]

    # 在压缩后的一维数组中找不超过 k 的最大子数组和
    current_max = self._find_max_subarray_no_larger_than_k(row_sum, k)
    max_val = max(max_val, current_max)

    # 如果已经找到等于 k 的解，直接返回
    if max_val == k:
        return k

return max_val

def _find_max_subarray_no_larger_than_k(self, arr, k):
"""
辅助函数：在一维数组中找不超过 k 的最大子数组和

使用前缀和和有序集合（bisect 模块）来高效查找
"""
import bisect

max_sum = float('-inf')
prefix_sum = 0

# 使用列表存储前缀和，并保持有序
```

```

prefix_sums = [0]

for num in arr:
    prefix_sum += num

    # 寻找 prefix_sum - x <= k → x >= prefix_sum - k
    target = prefix_sum - k
    # 使用 bisect_left 查找第一个大于等于 target 的前缀和
    idx = bisect.bisect_left(prefix_sums, target)

    if idx < len(prefix_sums):
        max_sum = max(max_sum, prefix_sum - prefix_sums[idx])

    # 将当前前缀和插入到正确位置，保持有序
    bisect.insort(prefix_sums, prefix_sum)

return max_sum

# =====
# 题目 3: LeetCode 1074. 元素和为目标值的子矩阵数量
# =====
# 题目来源: LeetCode (力扣)
# 题目链接: https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/
# 难度等级: 困难
#
# 【算法思路】
# 1. 二维压缩: 枚举所有可能的上下边界, 将矩阵压缩为一维数组
# 2. 前缀和 + 哈希表: 在压缩后的一维数组中, 使用前缀和和哈希表统计和为 target 的子数组数量
#
# 【时间复杂度】O(n2 × m)
# - 枚举上下边界: O(n2)
# - 前缀和计算和哈希表统计: O(m)
#
# 【空间复杂度】O(m)
# - 压缩数组: O(m)
# - 哈希表: O(m)
#
# 【是否为最优解】是的, 这是该问题的最优解法。
# =====

class LeetCode1074:
    """计算和为目标值的子矩阵数量"""

```

```

def numSubmatrixSumTarget(self, matrix, target):
    """
    主函数：计算和为 target 的子矩阵数量
    """

    参数：
        matrix: List[List[int]] - 输入矩阵
        target: int - 目标和

    返回：
        int - 和为 target 的子矩阵数量
    """

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

    n = len(matrix)      # 行数
    m = len(matrix[0])   # 列数
    count = 0

    # 枚举上边界
    for i in range(n):
        row_sum = [0] * m  # 存储压缩后的行和

        # 枚举下边界
        for j in range(i, n):
            # 更新压缩后的行和
            for l in range(m):
                row_sum[l] += matrix[j][l]

            # 在压缩后的一维数组中计算和为 target 的子数组数量
            count += self._find_subarray_sum(row_sum, target)

    return count

```

```

def _find_subarray_sum(self, arr, target):
    """
    辅助函数：计算一维数组中和为 target 的子数组数量

    使用前缀和和哈希表来高效统计
    """

    # 哈希表记录前缀和出现的次数
    prefix_sum_count = {0: 1}  # 初始前缀和为 0，出现一次

    prefix_sum = 0

```

```

count = 0

for num in arr:
    prefix_sum += num

    # 查找是否存在前缀和为 (prefix_sum - target)
    if (prefix_sum - target) in prefix_sum_count:
        count += prefix_sum_count[prefix_sum - target]

    # 更新当前前缀和的出现次数
    prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1

return count

```

```

# =====
# 题目 4: 洛谷 P1719 最大加权矩形
# =====

# 题目来源: 洛谷 (Luogu)
# 题目链接: https://www.luogu.com.cn/problem/P1719
# 难度等级: 普及+/提高
# =====

class LuoguP1719:

    """最大加权矩形求解器"""

    def __init__(self):
        self.n = 0
        self.matrix = []

    def input(self):
        """读取输入"""
        import sys
        self.n = int(sys.stdin.readline())
        self.matrix = []
        for _ in range(self.n):
            row = list(map(int, sys.stdin.readline().split()))
            self.matrix.append(row)

    def _kadane(self, arr):
        """Kadane 算法实现"""
        max_val = float('-inf')
        cur = 0

```

```

for num in arr:
    cur += num
    max_val = max(max_val, cur)
    if cur < 0:
        cur = 0

return max_val

def solve(self):
    """求解最大加权矩形"""
    max_val = float('-inf')

    # 枚举上边界
    for i in range(self.n):
        # 重置辅助数组
        arr = [0] * self.n

        # 枚举下边界
        for j in range(i, self.n):
            # 压缩列
            for k in range(self.n):
                arr[k] += self.matrix[j][k]

            # 应用 Kadane 算法
            max_val = max(max_val, self._kadane(arr))

    return max_val

# =====
# 题目 5: 牛客网 BM97 子矩阵最大和
# =====
# 题目来源: 牛客网 (NowCoder)
# 题目链接: https://www.nowcoder.com/practice/840eee05dccd4ffd8f9433ce8085946b
# 难度等级: 中等
# =====
class NowCoder_BM97:
    """子矩阵最大和求解器"""

    def _max_subarray(self, arr):
        """计算一维数组的最大子数组和"""
        max_val = float('-inf')
        cur = 0

```

```
for num in arr:  
    cur += num  
    max_val = max(max_val, cur)  
    if cur < 0:  
        cur = 0  
  
return max_val
```

```
def maxsumofSubmatrix(self, matrix):
```

```
    """
```

主函数：求子矩阵的最大和

参数：

matrix: List[List[int]] - 输入矩阵

返回：

int - 最大子矩阵和

```
    """
```

```
if not matrix or not matrix[0]:
```

```
    return 0
```

```
n = len(matrix)  
m = len(matrix[0])  
max_val = float('-inf')
```

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

```
    arr = [0] * m
```

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

```
        for k in range(m):
```

```
            arr[k] += matrix[j][k]
```

```
            max_val = max(max_val, self._max_subarray(arr))
```

```
return max_val
```

```
# ======  
# 题目 6: CodeChef - MAXREC  
# ======  
# 题目来源: CodeChef  
# 题目链接: https://www.codechef.com/problems/MAXREC  
# 难度等级: 中等  
# ======
```

```
class CodeChef_MAXREC:  
    """CodeChef MAXREC 问题求解器"""  
  
    @staticmethod  
    def _kadane(arr):  
        """Kadane 算法静态方法"""  
        max_val = float('-inf')  
        cur = 0  
  
        for num in arr:  
            cur += num  
            max_val = max(max_val, cur)  
            if cur < 0:  
                cur = 0  
  
        return max_val
```

```
@staticmethod  
def solve(matrix, n, m):  
    """  
    静态方法求解最大子矩阵和  
    """
```

参数:

```
matrix: List[List[int]] - 输入矩阵  
n: int - 行数  
m: int - 列数
```

返回:

```
int - 最大子矩阵和  
"""
```

```
max_val = float('-inf')  
  
for i in range(n):  
    arr = [0] * m  
    for j in range(i, n):  
        for k in range(m):  
            arr[k] += matrix[j][k]  
        max_val = max(max_val, CodeChef_MAXREC._kadane(arr))  
  
return max_val
```

```
# =====
```

```

# 题目 7: SPOJ - MAXSUBM
# =====
# 题目来源: SPOJ
# 题目链接: https://www.spoj.com/problems/MAXSUBM/
# 难度等级: 中等
# =====

class SPOJ_MAXSUBM:
    """SPOJ MAXSUBM 问题求解器"""

    @staticmethod
    def _kadane(arr):
        """Kadane 算法静态方法"""
        max_val = float('-inf')
        cur = 0

        for num in arr:
            cur += num
            max_val = max(max_val, cur)
            if cur < 0:
                cur = 0

        return max_val

    @staticmethod
    def solve(matrix, r, c):
        """
        静态方法求解最大子矩阵和
        """


```

参数:

```

matrix: List[List[int]] - 输入矩阵
r: int - 行数
c: int - 列数

```

返回:

```

int - 最大子矩阵和
"""

```

```

max_val = float('-inf')

for i in range(r):
    arr = [0] * c
    for j in range(i, r):
        for k in range(c):
            arr[k] += matrix[j][k]

```

```

max_val = max(max_val, SPOJ_MAXSUBM._kadane(arr))

return max_val

# =====
# Python 测试示例（扩展功能）
# =====

if __name__ == "__main__":
    # 测试 LeetCode 363
    print("\n==== 测试 LeetCode 363 ===")
    leetcode363 = LeetCode363()
    test_matrix_363 = [[1, 0, 1], [0, -2, 3]]
    k = 2
    result_363 = leetcode363.maxSumSubmatrix(test_matrix_363, k)
    print(f"输入: matrix=[[1, 0, 1], [0, -2, 3]], k=2")
    print(f"输出: {result_363}")
    print(f"期望: 2")

    # 测试 LeetCode 1074
    print("\n==== 测试 LeetCode 1074 ===")
    leetcode1074 = LeetCode1074()
    test_matrix_1074 = [[0, 1, 0], [1, 1, 1], [0, 1, 0]]
    target = 0
    result_1074 = leetcode1074.numSubmatrixSumTarget(test_matrix_1074, target)
    print(f"输入: matrix=[[0, 1, 0], [1, 1, 1], [0, 1, 0]], target=0")
    print(f"输出: {result_1074}")
    print(f"期望: 4")

# =====
# 题目 8: LeetCode 152. 乘积最大子数组
# =====

# 题目来源: LeetCode (力扣)
# 题目链接: https://leetcode.com/problems/maximum-product-subarray/
# 难度等级: 中等
#
# 【算法思路】
# 1. 同时维护当前最大值和最小值（因为负数×负数可能得到正数）
# 2. 对于每个元素，考虑三种情况：当前元素本身、当前元素×最大值、当前元素×最小值
# 3. 更新全局最大值
#
# 【时间复杂度】O(n)

```

```
# - 只需一次遍历数组
#
# 【空间复杂度】O(1)
# - 只需要常数空间存储变量
#
# 【是否为最优解】是的！
# - 这是该问题的最优解法，无法继续优化
# =====
class LeetCode152:
    """乘积最大子数组求解器"""

    def maxProduct(self, nums):
        """
        主函数：求乘积最大子数组

        参数：
            nums: List[int] - 输入数组

        返回：
            int - 乘积最大子数组的乘积值
        """

        if not nums:
            return 0

        max_prod = nums[0]      # 全局最大值
        cur_max = nums[0]       # 当前最大值
        cur_min = nums[0]       # 当前最小值

        for i in range(1, len(nums)):
            # 由于负数×负数可能得到正数，需要同时考虑三种情况
            temp_max = cur_max
            cur_max = max(nums[i], nums[i] * cur_max, nums[i] * cur_min)
            cur_min = min(nums[i], nums[i] * temp_max, nums[i] * cur_min)

            max_prod = max(max_prod, cur_max)

        return max_prod

# =====
# 题目 9: LeetCode 918. 环形子数组的最大和
# =====
# 题目来源: LeetCode (力扣)
```

```
# 题目链接: https://leetcode.com/problems/maximum-sum-circular-subarray/
# 难度等级: 中等
#
# 【算法思路】
# 环形数组的最大子数组和有两种情况:
# 1. 情况一: 最大子数组在数组中间 (非环形) → 标准 Kadane 算法
# 2. 情况二: 最大子数组跨越数组首尾 (环形) → 总和 - 最小子数组和
#
# 特殊情况: 如果数组全为负数, 则直接返回最大元素
#
# 【时间复杂度】O(n)
# - 需要两次遍历数组
#
# 【空间复杂度】O(1)
# - 只需要常数空间
#
# 【是否为最优解】是的!
# - 这是环形数组最大子数组和的最优解法
# =====
class LeetCode918:
    """环形子数组的最大和求解器"""

    def _kadane(self, nums):
        """标准 Kadane 算法: 求最大子数组和"""
        max_val = float('-inf')
        cur = 0

        for num in nums:
            cur += num
            max_val = max(max_val, cur)
            if cur < 0:
                cur = 0

        return max_val

    def _min_kadane(self, nums):
        """求最小子数组和 (Kadane 算法变种)"""
        min_val = float('inf')
        cur = 0

        for num in nums:
            cur += num
            min_val = min(min_val, cur)
```

```

    if cur > 0:
        cur = 0

    return min_val

def maxSubarraySumCircular(self, nums):
    """
    主函数: 求环形数组的最大子数组和

    参数:
        nums: List[int] - 输入数组 (环形)

    返回:
        int - 环形数组的最大子数组和
    """

    if not nums:
        return 0

    max_kadane = self._kadane(nums) # 情况一: 标准最大子数组和

    # 如果最大子数组和为负数, 说明整个数组都是负数
    if max_kadane < 0:
        return max_kadane

    # 计算数组总和
    total_sum = sum(nums)

    # 情况二: 环形情况的最大和 = 总和 - 最小子数组和
    min_kadane = self._min_kadane(nums)
    max_circular = total_sum - min_kadane

    # 返回两种情况的最大值
    return max(max_kadane, max_circular)

```

```

# =====
# 扩展测试代码
# =====

if __name__ == "__main__":
    # 测试 LeetCode 152
    print()
    print("==> 测试 LeetCode 152 ==>")
    leetcode152 = LeetCode152()

```

```

test_nums_152 = [2, 3, -2, 4]
result_152 = leetcode152.maxProduct(test_nums_152)
print("输入: nums=[2, 3, -2, 4]")
print("输出:", result_152)
print("期望: 6")

# 测试 LeetCode 918
print()
print("== 测试 LeetCode 918 ==")
leetcode918 = LeetCode918()
test_nums_918 = [5, -3, 5]
result_918 = leetcode918.maxSubarraySumCircular(test_nums_918)
print("输入: nums=[5, -3, 5]")
print("输出:", result_918)
print("期望: 10")

print()
print("所有扩展测试完成!")

```

=====

文件: Code02_SpecifyAmount. java

=====

```

package class019;

// =====
// 题目 2: 子矩阵的最大累加和 (ACM 风格 - 指定数据规模)
// =====
// 题目来源: 牛客网 (NowCoder)
// 题目链接: https://www.nowcoder.com/practice/cb82a97dcd0d48a7b1f4ee917e2c0409
// 难度等级: 中等
//
// =====
// ACM 风格说明 - 指定数据规模版本
// =====
// 【与 Code01 的区别】
// - Code01: 填函数风格, 只需要实现算法逻辑
// - Code02: ACM 风格, 需要自己处理输入输出
//
// 【什么是 ACM 风格?】
// ACM 竞赛中的标准输入输出方式:
// 1. 从标准输入 (System.in) 读取数据
// 2. 向标准输出 (System.out) 输出结果

```

```
// 3. 需要自己解析输入格式
// 4. 需要按要求格式化输出
//
// 【本题的输入格式】
// - 第一行：两个整数 n 和 m（矩阵的行数和列数）
// - 接下 n 行：每行 m 个整数，表示矩阵元素
// - 可能有多组测试数据（直到文件结束）
//
// 【本题的输出格式】
// - 对于每组测试数据，输出一行：一个整数，表示最大子矩阵和
//
// =====
// Java 中 IO 优化技巧详解
// =====
// 【为什么需要 IO 优化？】
// - Scanner 读取速度慢，大数据量时会超时
// - System.out.println 输出效率低，频繁调用会卡顿
// - 竞赛中时间限制严格，IO 效率很关键
//
// 【Java 快速 IO 方案对比】
// 1. Scanner + System.out.println
//   - 优点：简单易用，支持多种类型
//   - 缺点：最慢，不适合竞赛
//   - 适用场景：小数据量，快速调试
//
// 2. BufferedReader + PrintWriter
//   - 优点：速度中等，代码简洁
//   - 缺点：需要手动解析数字
//   - 适用场景：中等数据量
//
// 3. BufferedReader + StreamTokenizer + PrintWriter（本文件使用）
//   - 优点：读取数字非常快，代码也较简洁
//   - 缺点：只适合读数字，不适合读字符串
//   - 适用场景：大数据量数字输入（推荐！）
//
// 4. FastReader + FastWriter（见 Code06）
//   - 优点：最快，手动控制每一步
//   - 缺点：代码复杂，需要自己实现
//   - 适用场景：极限数据量，对速度要求极高
//
// 5. Kattio（见 Code05）
//   - 优点：能处理 StreamTokenizer 无法处理的特殊情况
//   - 缺点：比 StreamTokenizer 慢一点
```

```
// - 适用场景：需要读取大数、科学计数法数字、字符串
//
// =====
// StreamTokenizer 详细说明
// =====
// 【StreamTokenizer 是什么？】
// - Java 内置的词法分析器
// - 能够将输入流分解为一个个“词素（token）”
// - 自动处理空格、换行符等分隔符
//
// 【主要 API】
// 1. nextToken() - 读取下一个词素
//   - 返回值：
//     - TT_EOF: 文件结束
//     - TT_NUMBER: 数字
//     - TT_WORD: 单词
//     - 其他: 字符的 ASCII 值
//
// 2. nval - 当前读取的数字值（double 类型）
//   - 注意: 需要强制转换为 int 或 long
//
// 3. sval - 当前读取的字符串值
//
// 【使用模式】
// while (in.nextToken() != StreamTokenizer.TT_EOF) {
//   int n = (int) in.nval; // 读取第一个数
//   in.nextToken();
//   int m = (int) in.nval; // 读取第二个数
//   // ... 处理逻辑
// }
//
// 【注意事项】
// 1. 每次读取前必须调用 nextToken()
// 2. nval 是 double 类型，需要强转
// 3. 不能读取字符串（只能读数字和单词）
// 4. 对于极大的 long、科学计数法可能出错，请用 Kattio
//
// =====
// PrintWriter 详细说明
// =====
// 【PrintWriter 是什么？】
// - Java 的缓冲输出流
// - 将数据先写入内存缓冲区，批量输出
```

```
// - 比 System.out.println 快很多
//
// 【主要 API】
// 1. print(x) - 输出 x, 不换行
// 2. println(x) - 输出 x, 自动换行
// 3. flush() - 强制刷新缓冲区, 将数据输出
// 4. close() - 关闭流 (会自动 flush)
//
// 【使用模式】
// PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
// out.println(result);
// out.flush(); // 必须调用, 否则可能没有输出
// out.close();
//
// 【注意事项】
// 1. 必须调用 flush() 或 close(), 否则数据可能留在缓冲区
// 2. close() 会自动调用 flush()
// 3. 不要和 System.out.println 混用
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code02_SpecifyAmount {

    public static void main(String[] args) throws IOException {
        // 把文件里的内容, load 进来, 保存在内存里, 很高效, 很经济, 托管的很好
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        // 一个一个读数字
        StreamTokenizer in = new StreamTokenizer(br);
        // 提交答案的时候用的, 也是一个内存托管区
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) { // 文件没有结束就继续
            // n, 二维数组的行
            int n = (int) in.nval;
            in.nextToken();
            // m, 二维数组的列
            int m = (int) in.nval;
            // 装数字的矩阵, 临时动态生成
            int[][] mat = new int[n][m];
```

```

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                in.nextToken();
                mat[i][j] = (int) in.nval;
            }
        }
        out.println(maxSumSubmatrix(mat, n, m));
    }
    out.flush();
    br.close();
    out.close();
}

```

// 求子矩阵的最大累加和，后面的课会讲

// 算法思路：

- // 1. 枚举所有可能的上下边界(i, j)
- // 2. 将第 i 行到第 j 行的每列元素相加，形成一个一维数组
- // 3. 对这个一维数组求最大子数组和
- // 4. 记录所有情况下的最大值

//

// 时间复杂度：O(n^2 * m)，其中 n 是行数，m 是列数

// 空间复杂度：O(m)，用于存储压缩后的数组

```

public static int maxSumSubmatrix(int[][] mat, int n, int m) {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // 需要的辅助数组，临时动态生成
        int[] arr = new int[m];
        for (int j = i; j < n; j++) {
            for (int k = 0; k < m; k++) {
                arr[k] += mat[j][k];
            }
            max = Math.max(max, maxSumSubarray(arr, m));
        }
    }
    return max;
}

```

// 求子数组的最大累加和，使用 Kadane 算法

// 算法思路：

- // 1. 维护当前子数组的和(cur)和全局最大值(max)
- // 2. 遍历数组，将当前元素加入到当前子数组和中
- // 3. 更新全局最大值
- // 4. 如果当前子数组和变为负数，则重新开始计算(置为 0)

```
//  
// 时间复杂度: O(m)，其中 m 是数组长度  
// 空间复杂度: O(1)  
public static int maxSumSubarray(int[] arr, int m) {  
    int max = Integer.MIN_VALUE;  
    int cur = 0;  
    for (int i = 0; i < m; i++) {  
        cur += arr[i];  
        max = Math.max(max, cur);  
        cur = cur < 0 ? 0 : cur;  
    }  
    return max;  
}  
  
}
```

=====

文件: Code02_StaticSpace.cpp

```
// =====  
// 题目 2: 子矩阵的最大累加和 (ACM 风格 - C++版本)  
// =====  
// 题目来源: 牛客网 (NowCoder)  
// 题目链接: https://www.nowcoder.com/practice/cb82a97dc0d48a7b1f4ee917e2c0409  
//  
// ACM 风格 - 静态空间版本 (最推荐!)  
//  
// =====
```

```
#include <iostream>  
#include <cstring>      // memset  
#include <algorithm>    // max  
#include <climits>      // INT_MIN  
using namespace std;
```

```
// 题目给定的最大数据量  
const int MAXN = 201;  
const int MAXM = 201;
```

```
// 静态空间, 复用  
int mat[MAXN][MAXM];  
int arr[MAXM];
```

```

int n, m;

// Kadane 算法: 求一维数组的最大子数组和
// 时间复杂度: O(m)
// 空间复杂度: O(1)
int kadane() {
    int maxSum = INT_MIN;
    int cur = 0;
    for (int i = 0; i < m; i++) {
        cur += arr[i];
        maxSum = max(maxSum, cur);
        if (cur < 0) cur = 0;
    }
    return maxSum;
}

// 求最大子矩阵和
// 时间复杂度: O(n^2 × m)
// 空间复杂度: O(m)
int solve() {
    int maxSum = INT_MIN;

    for (int i = 0; i < n; i++) {
        // 清空辅助数组 (重要!)
        memset(arr, 0, sizeof(arr));

        for (int j = i; j < n; j++) {
            // 压缩第 j 行到辅助数组
            for (int k = 0; k < m; k++) {
                arr[k] += mat[j][k];
            }
            // 对压缩数组应用 Kadane 算法
            maxSum = max(maxSum, kadane());
        }
    }
}

return maxSum;
}

```

```

int main() {
    // C++快速 I/O 优化
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
}

```

```

while (cin >> n >> m) {
    // 读取矩阵
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> mat[i][j];
        }
    }

    // 输出结果
    cout << solve() << "\n";
}

return 0;
}

```

=====

文件: Code02_StaticSpace.py

```

# =====
# 题目 2: 子矩阵的最大累加和 (ACM 风格 - Python 版本)
# =====
# 题目来源: 牛客网 (NowCoder)
# 题目链接: https://www.nowcoder.com/practice/cb82a97dcd0d48a7b1f4ee917e2c0409
#
# ACM 风格 - 标准输入输出版本
#
# Python IO 优化技巧:

```

```

# 1. 使用 sys.stdin.readline 替代 input()
# 2. 使用 sys.stdout.write 替代 print() (可选)
# 3. 一次性读取所有输入 (适合小数据)
#
# =====

```

```
import sys
```

```
# Python 快速 IO 优化: 使用 sys.stdin.readline 替代 input()
input = sys.stdin.readline
```

```
def kadane(arr):
    """

```

Kadane 算法：求一维数组的最大子数组和

参数：

arr: List[int] - 一维数组

返回：

int/float - 最大子数组和

时间复杂度：O(n) 其中 n 是数组长度

空间复杂度：O(1)

"""

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

```
cur = 0
```

```
for num in arr:
```

```
    cur += num
```

```
    max_sum = max(max_sum, cur)
```

```
    if cur < 0:
```

```
        cur = 0
```

```
return max_sum
```

```
def solve(mat, n, m):
```

"""

求最大子矩阵和

参数：

mat: List[List[int]] - 输入矩阵

n: int - 行数

m: int - 列数

返回：

int/float - 最大子矩阵和

时间复杂度：O(n² × m)

空间复杂度：O(m)

"""

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

```
# 枚举上边界
```

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

```
    # 辅助数组，存储列压缩结果
```

```
arr = [0] * m

# 枚举下边界
for j in range(i, n):
    # 将第 j 行压缩到 arr
    for k in range(m):
        arr[k] += mat[j][k]

    # 对压缩数组应用 Kadane 算法
    max_sum = max(max_sum, kadane(arr))

return int(max_sum) if max_sum != float('-inf') else 0
```

```
def main():
    """
    主函数：处理 ACM 风格的输入输出
```

输入格式：

- 多组测试数据，直到 EOF
- 每组第一行：n m (行数 列数)
- 接下来 n 行，每行 m 个整数

输出格式：

- 对于每组测试数据，输出一行：最大子矩阵和

```
"""
```

```
while True:
```

```
    try:
```

```
        # 读取 n 和 m
        line = input().strip()
        if not line:
            break
```

```
        n, m = map(int, line.split())
```

```
        # 读取矩阵
        mat = []
        for _ in range(n):
            row = list(map(int, input().split()))
            mat.append(row)
```

```
        # 求解并输出
        result = solve(mat, n, m)
```

```
    print(result)

except EOFError:
    break
except:
    break

if __name__ == "__main__":
    main()

# =====
# Python ACM 输入输出模板总结
# =====
#
# 模板 1: 单组数据
# ```python
# n, m = map(int, input().split())
# arr = list(map(int, input().split()))
# print(result)
# ```

# 模板 2: 多组数据 (给定组数)
# ```python
# t = int(input())
# for _ in range(t):
#     n = int(input())
#     # 处理...
#     print(result)
# ```

# 模板 3: 多组数据 (直到 EOF) - 本题使用
# ```python
# while True:
#     try:
#         line = input().strip()
#         if not line:
#             break
#         # 处理...
#     except EOFError:
#         break
# ```

# =====
```

```
#  
# 模板 4: 快速 IO (极限优化)  
# ````python  
# import sys  
# input = sys.stdin.readline # 替换 input 函数  
#  
# # 读取  
# n = int(input())  
# arr = list(map(int, input().split()))  
#  
# # 输出  
# sys.stdout.write(str(result) + '\n')  
# ````  
#  
# ======  
# Python 性能优化技巧  
# ======  
#  
# 1. 使用 PyPy 解释器  
#   - PyPy 比 CPython 快 2-5 倍  
#   - 提交代码时选择 PyPy3  
#  
# 2. 使用 sys.stdin.readline  
#   - 比 input() 快很多  
#   - input = sys.stdin.readline  
#  
# 3. 使用列表推导式  
#   - 比 for 循环快  
#   - arr = [int(x) for x in input().split()]  
#  
# 4. 使用局部变量  
#   - 局部变量比全局变量访问快  
#   - 将频繁使用的函数赋值给局部变量  
#  
# 5. 避免重复计算  
#   - 使用变量缓存结果  
#   - 避免在循环内重复调用函数  
#  
# ======  
# Python vs Java vs C++ 性能对比  
# ======  
#  
# | 语言 | 运行速度 | 代码长度 | 学习难度 | 推荐指数 |
```

```

# |-----|-----|-----|-----|-----|
# | C++ | ★★★★★★★ | ★★★★ | ★★★★ | ★★★★★★★ | 
# | Java | ★★★★★★ | ★★★★★★ | ★★★★★★ | ★★★★★★ | 
# | Python | ★★ | ★★★★★★★ | ★★★★★★★ | ★★★★ | 
# | PyPy | ★★★★★★ | ★★★★★★★ | ★★★★★★★ | ★★★★★★ | 

#
# 注: 使用 PyPy 可以大幅提升 Python 性能, 使其接近 Java
#
# =====
# 测试用例
# =====
#
# 输入:
# 3 3
# 1 2 3
# -4 5 -6
# 7 8 9
#
# 输出:
# 27
#
# 解释: 选择整个矩阵, 和为 1+2+3-4+5-6+7+8+9=27
#
# =====

```

文件: Code03_StaticSpace.java

```

=====
package class019;

// 展示 acm 风格的测试方式
// 子矩阵的最大累加和问题, 不要求会解题思路, 后面的课会讲
// 每一组测试都给定数据规模
// 任何空间都提前生成好, 一律都是静态空间, 然后自己去复用, 推荐这种方式
// 测试链接 : https://www.nowcoder.com/practice/cb82a97dcd0d48a7b1f4ee917e2c0409?
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code, 提交时请把类名改成"Main", 可以直接通过
//
// 题目描述:
// 给定一个矩阵, 求其所有子矩阵中元素和的最大值
// 输入:

```

```
// 第一行包含两个整数 n 和 m(1 <= n, m <= 100), 表示矩阵的行数和列数
// 接下来 n 行, 每行包含 m 个整数, 表示矩阵中的元素(-1000 <= matrix[i][j] <= 1000)
// 输出:
// 一个整数, 表示所有子矩阵中元素和的最大值
//
// 示例:
// 输入:
// 3 3
// 1 2 3
// -4 5 -6
// 7 8 9
// 输出:
// 27
//
// 解题思路:
// 1. 使用压缩数组技巧, 将二维问题转化为一维最大子数组和问题
// 2. 枚举所有可能的上下边界组合(i, j), 将第 i 行到第 j 行的每列元素相加, 得到一个一维数组
// 3. 对这个一维数组求最大子数组和, 这就是以(i, j)为上下边界的所有子矩阵中的最大和
// 4. 遍历所有上下边界组合, 记录全局最大值
//
// 时间复杂度分析:
// - 枚举上下边界: O(n^2)
// - 计算压缩数组并求最大子数组和: O(m)
// - 总时间复杂度: O(n^2 * m)
//
// 空间复杂度分析:
// - 需要一个辅助数组存储压缩后的结果: O(m)
// - 总空间复杂度: O(m)
//
// 适用场景:
// - 需要在二维矩阵中找到和最大的子矩阵
// - 数据规模较小(n, m <= 100)的情况
//
// 优化点:
// - 可以在计算压缩数组时同时进行最大子数组和的计算, 避免两次遍历
// - 可以使用前缀和优化压缩数组的计算过程
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
```

```
import java.util.Arrays;

public class Code03_StaticSpace {

    // 题目给定的行的最大数据量
    public static int MAXN = 201;

    // 题目给定的列的最大数据量
    public static int MAXM = 201;

    // 申请这么大的矩阵空间，一定够用了
    // 静态的空间，不停复用
    public static int[][] mat = new int[MAXN][MAXM];

    // 需要的所有辅助空间也提前生成
    // 静态的空间，不停复用
    public static int[] arr = new int[MAXM];

    // 当前测试数据行的数量是 n
    // 当前测试数据列的数量是 m
    // 这两个变量可以把代码运行的边界规定下来
    public static int n, m;

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            in.nextToken();
            m = (int) in.nval;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    in.nextToken();
                    mat[i][j] = (int) in.nval;
                }
            }
            out.println(maxSumSubmatrix());
        }
        out.flush();
        br.close();
        out.close();
    }
}
```

```

// 求子矩阵的最大累加和，后面的课会讲
// 算法思路：
// 1. 枚举所有可能的上下边界(i, j)
// 2. 将第 i 行到第 j 行的每列元素相加，形成一个一维数组
// 3. 对这个一维数组求最大子数组和
// 4. 记录所有情况下的最大值
//
// 时间复杂度：O(n^2 * m)，其中 n 是行数，m 是列数
// 空间复杂度：O(m)，用于存储压缩后的数组
public static int maxSumSubmatrix() {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // 因为之前的过程可能用过辅助数组
        // 为了让之前结果不干扰到这次运行，需要自己清空辅助数组需要用到的部分
        Arrays.fill(arr, 0, m, 0);
        for (int j = i; j < n; j++) {
            for (int k = 0; k < m; k++) {
                arr[k] += mat[j][k];
            }
            max = Math.max(max, maxSumSubarray());
        }
    }
    return max;
}

```

```

// 求子数组的最大累加和，使用 Kadane 算法
// 算法思路：
// 1. 维护当前子数组的和(cur)和全局最大值(max)
// 2. 遍历数组，将当前元素加入到当前子数组和中
// 3. 更新全局最大值
// 4. 如果当前子数组和变为负数，则重新开始计算(置为 0)
//
// 时间复杂度：O(m)，其中 m 是数组长度
// 空间复杂度：O(1)
public static int maxSumSubarray() {
    int max = Integer.MIN_VALUE;
    int cur = 0;
    for (int i = 0; i < m; i++) {
        cur += arr[i];
        max = Math.max(max, cur);
        cur = cur < 0 ? 0 : cur;
    }
}
```

```
    return max;  
}  
  
=====
```

文件: Code04_ReadByLine.java

```
=====  
package class019;  
  
// 展示 acm 风格的测试方式  
// 测试链接 : https://www.nowcoder.com/exam/test/70070648/detail?pid=27976983  
// 其中, 7. A+B(7), 就是一个没有给定数据规模, 只能按行读数据的例子  
// 此时需要自己切分出数据来计算  
// 请同学们务必参考如下代码中关于输入、输出的处理  
// 这是输入输出处理效率很高的写法  
// 提交以下的 code, 提交时请把类名改成"Main", 可以直接通过  
//  
// 题目描述:  
// 计算一系列整数的和  
// 输入:  
// 多行数据, 每行包含若干个整数(可能有正有负), 整数之间用空格分隔  
// 输出:  
// 对于每一行输入, 输出该行所有整数的和  
//  
// 示例:  
// 输入:  
// 1 2 3  
// 4 5  
// -1 0 1  
// 输出:  
// 6  
// 9  
// 0  
//  
// 解题思路:  
// 1. 按行读取输入数据  
// 2. 使用 split 方法将每行数据按空格分割成字符串数组  
// 3. 将每个字符串转换为整数并累加  
// 4. 输出每行的和  
//  
// 时间复杂度分析:
```

```
// - 假设总共有 n 个数字，时间复杂度为 O(n)
//
// 空间复杂度分析：
// - 需要存储每行分割后的字符串数组，最坏情况下为 O(m)，其中 m 为单行最多的数字个数
//
// 适用场景：
// - 输入数据没有明确的数据规模说明
// - 每行数据格式相对简单，可以通过分隔符分割
//
// 优化点：
// - 可以使用更高效的字符串分割方法
// - 可以使用快速输入输出方法提高 I/O 效率

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

public class Code04_ReadByLine {

    public static String line;
    public static String[] parts;
    public static int sum;

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while ((line = in.readLine()) != null) {
            parts = line.split(" ");
            sum = 0;
            for (String num : parts) {
                sum += Integer.valueOf(num);
            }
            out.println(sum);
        }
        out.flush();
        in.close();
        out.close();
    }
}
```

```
}
```

```
=====
```

文件: Code05_Kattio.java

```
=====
```

```
package class019;
```

```
// 本文件课上没有讲，介绍一下 Kattio 类的使用
```

```
// 某些题目的输入，使用 StreamTokenizer 就是无法正确读入
```

```
// 那么可以使用本文件提供的 Kattio 类
```

```
// 比如，就是需要依次读取一个一个的字符串进行处理
```

```
// 再比如，StreamTokenizer 读取：不溢出、但是很大的 long 类型数字时，可能会读入错误
```

```
// 再比如，StreamTokenizer 读取：科学计数法表达的 double 类型数字时，可能会读入错误
```

```
// 如果使用 Kattio 进行读取，就没有这些问题
```

```
// 可以直接运行本文件的 main 函数，根据提示输入给定的数字，能清晰的看到这一点
```

```
// 那么可不可以放弃 StreamTokenizer，以后都用 Kattio 呢？
```

```
// 不行！因为 StreamTokenizer 的效率还是比 Kattio 好！
```

```
// 只有在 StreamTokenizer 无法正确读取的情况下，才考虑使用 Kattio 类
```

```
// 其他语言中一定有对等的概念，不是 java 的同学，请自行搞定
```

```
//
```

```
// Kattio 类的优势：
```

```
// 1. 可以正确处理大整数和科学计数法表示的浮点数
```

```
// 2. 可以按需读取不同类型的输入数据
```

```
// 3. 相比 Scanner 有更好的性能
```

```
//
```

```
// Kattio 类的劣势：
```

```
// 1. 性能不如 StreamTokenizer
```

```
// 2. 实现相对复杂
```

```
//
```

```
// 使用场景：
```

```
// 1. 需要读取非常大的整数（接近 long 范围）
```

```
// 2. 需要读取科学计数法表示的浮点数
```

```
// 3. 需要按字符串形式读取输入再进行自定义解析
```

```
import java.io.BufferedReader;
```

```
import java.io.FileReader;
```

```
import java.io.IOException;
```

```
import java.io.InputStream;
```

```
import java.io.InputStreamReader;
```

```
import java.io.OutputStream;
```

```
import java.io.PrintWriter;
```

```
import java.io.StreamTokenizer;
```

```
import java.util.StringTokenizer;

public class Code05_Kattio {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer in = new StringTokenizer(br);
        System.out.println("请输入 : 131237128371723187");
        System.out.println("并按回车");
        in.nextToken();
        long long1 = (long) in.nval;
        System.out.println("Tokenizer 读取到的数字 : ");
        System.out.println(long1);
        System.out.println();
        System.out.println("请输入 : 5.6920E+0001");
        System.out.println("并按回车");
        in.nextToken();
        double double1 = in.nval;
        System.out.println("Tokenizer 读取到的数字 : ");
        System.out.println(double1);

        System.out.println("=====");
        Kattio io = new Kattio(); // 自动接入输入输出流
        System.out.println("请输入 : 131237128371723187");
        System.out.println("并按回车");
        long long2 = io.nextLong();
        System.out.println("Kattio 读取到的数字 : ");
        System.out.println(long2);
        System.out.println();
        System.out.println("请输入 : 5.6920E+0001");
        System.out.println("并按回车");
        double double2 = io.nextDouble();
        System.out.println("Kattio 读取到的数字 : ");
        System.out.println(double2);
        io.close();
    }
}
```

```
// 如何使用 Kattio 的简单示例
// 可以找个一些具体题目试一试
// 这里就是罗列了一下
public static void show() {
    Kattio io = new Kattio(); // 自动接入输入输出流
```

```
    io.next(); // 读取下一个字符串，注意不是整行，是以空格或回车分割的字符串，一个一个读取
    io.nextInt(); // 读取下一个 int
    io.nextDouble(); // 读取下一个 double
    io.nextLong(); // 读取下一个 long
    io.println("ans"); // 答案进入输出流
    io.flush(); // 答案刷给后台
    io.close(); // 关闭 io
}

// Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
// 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
// 参考链接：https://oi-wiki.org/lang/java-pro/
//
// Kattio 类实现原理：
// 1. 使用 BufferedReader 进行缓冲读取，提高读取效率
// 2. 使用 StringTokenizer 进行字符串分割，按需解析不同类型数据
// 3. 提供 nextInt(), nextDouble(), nextLong() 等方法方便读取各种类型数据
//
// 时间复杂度：
// - 读取操作：O(1) 均摊时间复杂度
// - 字符串分割：O(n)，其中 n 为字符串长度
//
// 空间复杂度：
// - O(BUF_SIZE)，其中 BUF_SIZE 为缓冲区大小
public static class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;

    public Kattio() {
        this(System.in, System.out);
    }

    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }

    public Kattio(String input, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(input));
    }

    public String next() {
```

```

try {
    while (st == null || !st.hasMoreTokens())
        st = new StringTokenizer(r.readLine());
    return st.nextToken();
} catch (Exception e) {
}
return null;
}

public int nextInt() {
    return Integer.parseInt(next());
}

public double nextDouble() {
    return Double.parseDouble(next());
}

public long nextLong() {
    return Long.parseLong(next());
}

}

```

文件: Code06_FastReaderWriter.java

```

=====
package class019;

// 本文件课上没有讲
// java 同学可以使用 FastReader 进行快读，可以使用 FastWriter 进行快写，速度是很快的
// 如何使用可以参考 main 函数
//
// FastReader 和 FastWriter 的作用：
// 1. 提供比 Scanner 和 System.out.println 更快的输入输出方式
// 2. 适用于算法竞赛和在线评测系统中需要处理大量数据的场景
// 3. 避免因输入输出效率问题导致的超时
//
// FastReader 实现原理：
// 1. 使用 InputStream 直接读取字节数据
// 2. 内部维护缓冲区，减少系统调用次数
// 3. 手动解析数字字符，避免字符串转换开销

```

```
//  
// FastWriter 实现原理:  
// 1. 使用内部缓冲区，批量写入数据  
// 2. 手动将数字转换为字符，避免字符串转换开销  
// 3. 提供 flush 机制确保数据正确输出  
  
//  
// 时间复杂度:  
// - FastReader 读取操作: O(1) 均摊时间复杂度  
// - FastWriter 写入操作: O(1) 均摊时间复杂度  
  
//  
// 空间复杂度:  
// - FastReader: O(BUF_SIZE)，其中 BUF_SIZE 为缓冲区大小  
// - FastWriter: O(BUF_SIZE)，其中 BUF_SIZE 为缓冲区大小  
  
import java.io.BufferedReader;  
import java.io.ByteArrayOutputStream;  
import java.io.FileNotFoundException;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.io.Writer;  
import java.util.InputMismatchException;  
  
public class Code06_FastReaderWriter {  
  
    public static void main(String[] args) {  
        FastReader reader = new FastReader(System.in);  
        FastWriter writer = new FastWriter(System.out);  
        System.out.println("输入一个字符: ");  
        int cha = reader.readByte(); // reader 会读到字符的 ASCII 码  
        System.out.println("输入一个 int 类型的数字: ");  
        int num1 = reader.readInt(); // reader 会读到该数字  
        System.out.println("输入一个 long 类型的数字: ");  
        long num2 = reader.readLong(); // reader 会读到该数字  
        System.out.println("打印结果:");  
        writer.println(cha);  
        writer.println(num1);  
        writer.println(num2);  
        writer.close(); // close 方法包含 flush，会把结果刷出去  
    }  
  
    // 快读
```

```
// 实现细节：  
// 1. 使用字节数组作为缓冲区，减少系统调用  
// 2. 逐字节读取并解析数字，避免字符串转换  
// 3. 处理负数情况  
  
//  
// 适用场景：  
// 1. 算法竞赛中需要快速读取大量数据  
// 2. 在线评测系统中避免输入超时  
// 3. 需要手动控制输入格式的场景  
public static class FastReader {  
    InputStream is;  
    private byte[] inbuf = new byte[1024];  
    public int lenbuf = 0;  
    public int ptrbuf = 0;  
  
    public FastReader(final InputStream is) {  
        this.is = is;  
    }  
  
    public int readByte() {  
        if (lenbuf == -1) {  
            throw new InputMismatchException();  
        }  
        if (ptrbuf >= lenbuf) {  
            ptrbuf = 0;  
            try {  
                lenbuf = is.read(inbuf);  
            } catch (IOException e) {  
                throw new InputMismatchException();  
            }  
            if (lenbuf <= 0) {  
                return -1;  
            }  
        }  
        return inbuf[ptrbuf++];  
    }  
  
    public int readInt() {  
        return (int) readLong();  
    }  
  
    public long readLong() {  
        long num = 0;  
        int c = readByte();  
        while (c >= '0' && c <= '9') {  
            num = num * 10 + c - '0';  
            c = readByte();  
        }  
        if (c == '-') {  
            num *= -1;  
            c = readByte();  
        }  
        if (c != '\n' && c != '\r') {  
            throw new InputMismatchException();  
        }  
        return num;  
    }  
}
```

```
int b;
boolean minus = false;
while ((b = readByte()) != -1 && !((b >= '0' && b <= '9') || b == '-'))
;
if (b == '-') {
    minus = true;
    b = readByte();
}
while (true) {
    if (b >= '0' && b <= '9') {
        num = num * 10 + (b - '0');
    } else {
        return minus ? -num : num;
    }
    b = readByte();
}
}
```

// 快写

// 实现细节:

- // 1. 使用字节数组作为缓冲区，批量写入数据
- // 2. 手动将数字转换为字符，避免字符串转换开销
- // 3. 提供 flush 机制确保数据正确输出

//

// 适用场景:

- // 1. 算法竞赛中需要快速输出大量数据
- // 2. 在线评测系统中避免输出超时
- // 3. 需要手动控制输出格式的场景

```
public static class FastWriter {
```

```
    private static final int BUF_SIZE = 1 << 13;
    private final byte[] buf = new byte[BUF_SIZE];
    private OutputStream out;
    private Writer writer;
    private int ptr = 0;
```

```
    public FastWriter(Writer writer) {
        this.writer = new BufferedWriter(writer);
        out = new ByteArrayOutputStream();
    }
```

```
    public FastWriter(OutputStream os) {
        this.out = os;
```

```
}

public FastWriter(String path) {
    try {
        this.out = new FileOutputStream(path);
    } catch (FileNotFoundException e) {
        throw new RuntimeException("FastWriter");
    }
}

public FastWriter write(byte b) {
    buf[ptr++] = b;
    if (ptr == BUF_SIZE) {
        innerflush();
    }
    return this;
}

public FastWriter write(String s) {
    s.chars().forEach(c -> {
        buf[ptr++] = (byte) c;
        if (ptr == BUF_SIZE) {
            innerflush();
        }
    });
    return this;
}

private static int countDigits(long l) {
    if (l >= 1000000000000000000L) {
        return 19;
    }
    if (l >= 100000000000000000L) {
        return 18;
    }
    if (l >= 10000000000000000L) {
        return 17;
    }
    if (l >= 1000000000000000L) {
        return 16;
    }
    if (l >= 100000000000000L) {
        return 15;
    }
}
```

```
}

if (l >= 100000000000000L) {
    return 14;
}

if (l >= 1000000000000L) {
    return 13;
}

if (l >= 100000000000L) {
    return 12;
}

if (l >= 10000000000L) {
    return 11;
}

if (l >= 1000000000L) {
    return 10;
}

if (l >= 100000000L) {
    return 9;
}

if (l >= 10000000L) {
    return 8;
}

if (l >= 1000000L) {
    return 7;
}

if (l >= 100000L) {
    return 6;
}

if (l >= 10000L) {
    return 5;
}

if (l >= 1000L) {
    return 4;
}

if (l >= 100L) {
    return 3;
}

if (l >= 10L) {
    return 2;
}

return 1;
}
```

```

public FastWriter write(long x) {
    if (x == Long.MIN_VALUE) {
        return write("") + x;
    }
    if (ptr + 21 >= BUF_SIZE) {
        innerflush();
    }
    if (x < 0) {
        write((byte) '-' );
        x = -x;
    }
    int d = countDigits(x);
    for (int i = ptr + d - 1; i >= ptr; i--) {
        buf[i] = (byte) ('0' + x % 10);
        x /= 10;
    }
    ptr += d;
    return this;
}

public FastWriter writeln(long x) {
    return write(x).writeln();
}

public FastWriter writeln() {
    return write((byte) '\n');
}

private void innerflush() {
    try {
        out.write(buf, 0, ptr);
        ptr = 0;
    } catch (IOException e) {
        throw new RuntimeException("innerflush");
    }
}

public void flush() {
    innerflush();
    try {
        if (writer != null) {
            writer.write(((ByteArrayOutputStream) out).toString());
            out = new ByteArrayOutputStream();
        }
    }
}

```

```
        writer.flush();
    } else {
        out.flush();
    }
} catch (IOException e) {
    throw new RuntimeException("flush");
}

public FastWriter println(long x) {
    return writeln(x);
}

public void close() {
    flush();
    try {
        out.close();
    } catch (Exception e) {
    }
}

}
```
