矩阵压缩算法（固定大小子矩阵最大和）

一、问题背景

给定一个大小为 `n × m` 的矩阵 `mp`，其中 `n`、`m` 的规模可能达到 `10^6` 级别。 现要求在该矩阵中，找出一个 **`c × c` 的子矩阵**，使其 **元素之和最大**，并输出该子矩阵的值或其左上角坐标。

直接暴力的缺陷

如果直接枚举每一个 `c × c` 子矩阵，并逐个计算其内部元素和：

- 每个子矩阵求和：`O(c^2)`
- 子矩阵数量：`(n-c+1)(m-c+1)`
- 总时间复杂度：`O(nmc^2)`

当数据规模较大时，该方法显然不可行。

二、核心思想：矩阵压缩

> **将二维问题拆解为两次一维问题**

矩阵压缩的基本思路是：

1. 先在 **行方向** 进行压缩（横向滑动窗口）
2. 再在 **列方向** 进行压缩（纵向滑动窗口）

通过这种方式，可以在 `O(nm)` 的时间复杂度内，求出所有 `c × c` 子矩阵的元素和。

三、第一步：行方向压缩（横向滑动窗口）

思想说明

对矩阵的每一行，计算长度为 `c` 的连续区间和。

设原矩阵为：

```
mp[n][m]
```

构造一个中间矩阵：

```
rowSum[n][m - c + 1]
```

其中：

```
rowSum[i][j] = mp[i][j] + mp[i][j+1] + ... + mp[i][j+c-1]
```

表示第 `i` 行中，从第 `j` 列开始、长度为 `c` 的连续区间之和。

实现方式（滑动窗口）

```
for (int i = 0; i < n; i++) {
    long long sum = 0;
    for (int j = 0; j < m; j++) {
        sum += mp[i][j];
        if (j >= c) sum -= mp[i][j - c];
        if (j >= c - 1) {
            rowSum[i][j - c + 1] = sum;
        }
    }
}
```

特点

- 单行时间复杂度：`O(m)`
- 所有行总复杂度：`O(nm)`
- 利用"加新元素、减旧元素"的方式避免重复计算

四、第二步：列方向压缩（纵向滑动窗口）

思想说明

在完成行方向压缩后，每个 `rowSum[i][j]` 表示一段横向宽度为 `c` 的区间和。

接下来，对 `rowSum` 的每一列，再做一次纵向滑动窗口，计算连续 `c` 行的区间和。

目标值为：

```
blockSum[i][j] = rowSum[i][j] + rowSum[i+1][j] + ... + rowSum[i+c-1][j]
```

这正好对应原矩阵中一个 **`c × c` 子矩阵的元素和。**

实现方式（滑动窗口）

```
for (int j = 0; j <= m - c; j++) {
    long long sum = 0;
    for (int i = 0; i < n; i++) {
        sum += rowSum[i][j];
```

```
            if (i >= c) sum -= rowSum[i - c][j];
            if (i >= c - 1) {
                // 此时 sum 即为一个 c × c 子矩阵的和
            }
        }
    }
```

五、时间与空间复杂度分析

时间复杂度

- 行压缩：`O(nm)`
- 列压缩：`O(nm)`
- 总时间复杂度：**`O(nm)`**

空间复杂度

- 中间矩阵 `rowSum`：`O(n(m-c+1))`

相较于暴力解法，有数量级上的优化。

六、空间优化：滚动数组（进阶）

实际上，`rowSum` 只是一个中间结果，并非必须完整存储。

可以采用 **滚动数组 + 实时更新** 的方式：

- 使用一维数组 `col[m]`
- 表示当前连续 `c` 行在每一列上的累计和
- 每向下移动一行：
    - 加入新行
    - 移除最旧的一行
- 再在 `col` 上进行横向滑动窗口

最终：

- 时间复杂度仍为 `O(nm)`
- 空间复杂度可优化至：**`O(m)`**

这是竞赛中最常见、最优的写法。

七、适用场景总结

场景	推荐算法

场景	推荐算法
固定大小子矩阵最大/最小值	矩阵压缩（滑动窗口）
多次任意子矩阵查询	二维前缀和
数据规模极大、内存受限	矩阵压缩 + 滚动数组

八、思维总结

> **矩阵压缩的本质在于：**
>
> - 用滑动窗口代替重复计算
> - 用增量维护代替整体重算
> - 将二维问题拆解为一维问题逐步解决

这是从「暴力枚举」走向「高效算法设计」的一个重要典型模型。

示例代码(完整版)

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long

int main()
{
    int n, m, c;
    cin >> n >> m >> c;

    vector<vector<int>> mp(n, vector<int>(m));
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++)
            cin >> mp[i][j];

    // 横向滑动
    vector<vector<ll>> prex(n, vector<ll>(m - c + 1));

    for(int i = 0; i < n; i++)
    {
        ll sum = 0;
        for(int j = 0; j < m; j++)
        {
            sum += mp[i][j];
            if(j >= c) sum -= mp[i][j - c];
            if(j >= c - 1)
                prex[i][j - c + 1] = sum;
        }
    }

    // 纵向滑动
    vector<vector<ll>> prey(n - c + 1, vector<ll>(m - c + 1));
```

```
    for(int j = 0; j <= m - c; j++)
    {
        ll sum = 0;
        for(int i = 0; i < n; i++)
        {
            sum += prex[i][j];
            if(i >= c) sum -= prex[i - c][j];
            if(i >= c - 1)
                prey[i - c + 1][j] = sum;
        }
    }

    ll ans = LLONG_MIN;
    int x = 0, y = 0;

    for(int i = 0; i <= n - c; i++)
        for(int j = 0; j <= m - c; j++)
            if(prey[i][j] > ans)
            {
                ans = prey[i][j];
                x = i;
                y = j;
            }

    cout << x+1 << " " << y+1 << "\n";
}
```

代码优化版

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long

int main() {
    int n, m, c;
    cin >> n >> m >> c;

    vector<vector<int>> mp(n, vector<int>(m));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> mp[i][j];

    vector<ll> col(m, 0);
    ll ans = LLONG_MIN;
    int x = 0, y = 0;

    for (int i = 0; i < n; i++) {
        // 加当前行
        for (int j = 0; j < m; j++)
            col[j] += mp[i][j];
```

```
        // 去掉超出的行
        if (i >= c) {
            for (int j = 0; j < m; j++)
                col[j] -= mp[i - c][j];
        }

        // 行数不足 c，不能形成矩阵
        if (i < c - 1) continue;

        // 对 col 做横向滑窗
        ll sum = 0;
        for (int j = 0; j < m; j++) {
            sum += col[j];
            if (j >= c) sum -= col[j - c];
            if (j >= c - 1) {
                if (sum > ans) {
                    ans = sum;
                    x = i - c + 1;
                    y = j - c + 1;
                }
            }
        }
    }

    cout << x << " " << y << "\n";
}
```