2A：  
标准程序在各个测试数据上最大用时为.

时间限制为1秒，是上式的两倍。

所以解得 , 即 n最大为2236

2B

标准程序在各个测试数据上最大用时为 .

因此

所以解得n最大为145746.

3A：

Solution1的bug有两个：

首先，sum没有在每一次查询之后归零，这会导致第二次及以后的查询结果出错

其次，输入数据时角标从1开始，这会导致当n或者m被指定为2000时数组越界

Solution2的bug除上述两个之外，还有在计算sum的时候角标错误。计算从第y个开始的b个数的和应该用第y+b-1个数的rolsum减去第y-1个数的rolsum。

均通过阅读代码以及使用题中给出的测例找出。

大语言模型可以找出其中所有的bug，提供修复之后的版本，以及进一步优化时间复杂度的建议。

3B：

在单步调试程序时，首先需要生成可执行文件，并在命令行里用gdb调试器打开。

之后使用next命令可以单步执行，step可以单步进入函数，break可以设置断点，run可以执行到断点处，print命令可以查看变量的值，quit命令可以退出。

在vscode的图形化调试界面里面，点击逐过程可以单步执行，点击单步调试可以进入当前语句的函数，点击单步跳出可以跳出当前函数。点击继续可以执行至断点，点击每一行最左侧可以设置断点，在IDE左侧边栏可以直接查看变量或者查找。

为了使用调试器在编译程序时需要加入-g，以在编译时加入调试信息。

3C：

Srand(time（0）)的作用是用系统时钟初始化随机数种子，以便生成随机数。

3D:

System函数实际上是将函数参数当做命令输入命令行。

前四行的含义是编译rand\_input.cpp、check\_input.cpp、solution\_1.cpp、solution\_2.cpp这四个文件生成可执行文件。

system("./rand\_input > rand.in")执行 rand\_input ，把它的输出重定向到 rand.in 文件里。

system("./check\_input < rand.in")把 rand.in 作为输入执行 check\_input。

system("./solution\_1 < rand.in > 1.out");

system("./solution\_2 < rand.in > 2.out")分别用 rand.in 作为输入运行两个程序，把各自输出写到 1.out 和 2.out。

system("diff 1.out 2.out")用 diff 命令比较 1.out 和 2.out 是否一样，不同输出1.

3E：

根据注释，n、m最大为100，矩阵元素最大为1000，因此矩阵元素和最大为,远小于int类型的最大值，因此使用int是没问题的。

4A：

根据sol2在算法上相对sol1的改进，我们发现sol2通过预处理减少了一层循环，但是由于仍需要循环因此复杂度不够低。

因此我们在预处理时存储当前元素的二维信息（即以矩阵左上角元素和其本身为对角的矩阵的元素和）从而使得在每次查询时只需要对预处理数据进行加减运算，实现了O（1）的复杂度*。*

4B:

Sol3显著优于1和2：

1. n=m=1000, q=100000;
   1. sol1时间：11.21s, 11.50s, 14.75s, average: 12.49s
   2. sol2 时间：0.40s, 0.44s, 0.36s, average: 0.40s
   3. sol3 时间：0.23s, 0.29s, 0.24s, average: 0.25s
2. n=1000, m=100, q=100000;
   1. sol1时间：1.42s, 4.98s, 1.41s, average: 2.60s
   2. sol2 时间：0.17s, 0.16s, 0.16s, average: 0.16s
   3. sol3 时间：0.10s, 0.10s, 0.11s, average: 0.10s

sol3 不显著优于1和2：

1. n=100, m=1000, q=100000;
   1. sol1时间：1.31s, 1.28s, 1.26s, average: 1.28s
   2. sol2 时间：0.08s, 0.09s, 0.08s, average: 0.08s
   3. sol3 时间：0.10s, 0.10s, 0.10s, average: 0.10s
2. n=100, m=1000, q=10000;
   1. sol1时间：0.14s, 0.14s, 0.13s, average: 0.14s
   2. sol2 时间：0.02s, 0.02s, 0.01s, average: 0.02s
   3. sol3 时间：0.02s, 0.02s, 0.02s, average: 0.02s
3. n=100, m=100, q=100000;
   1. sol1时间：0.20s, 0.20s, 0.20s, average: 0.20s
   2. sol2 时间：0.07s, 0.06s, 0.07s, average: 0.07s
   3. sol3 时间：0.07s, 0.07s, 0.08s, average: 0.08s
4. n=1000, m=1000, q=100;
   1. sol1时间：0.13s, 0.14s, 0.13s, average: 0.13s
   2. sol2 时间：0.14s, 0.13s, 0.13s, average: 0.13s
   3. sol3 时间：0.13s, 0.13s, 0.14s, average: 0.13s

4C:

首先我们发现当n和m较小的时候三种算法差别较小。

其次当m较大而n不大时sol2与sol3无明显差异。

最后mn均较大时其差异显著度与q正相关。

总之，当n、m、q均较大时sol3明显优于其他两个算法，而当查询次数较小或者矩阵的行数很少的时候则优势不明显。

5B：

以下为ChatGPT的回答：

你描述的是**“二维数组，既要修改，又要矩形求和”**，这正是典型的 **二维动态区间和问题**。

**1️⃣ 朴素思路**

* **二维前缀和**：  
  先求一个 prefix[i][j] 表示 (1,1) 到 (i,j) 的和。  
  查询矩形和 = 四个前缀和相减 → **O(1)**。  
  但缺点：**不能快速修改**。修改一个元素后，前缀和需要整片重算 → **O(nm)**。

如果修改次数很少、查询很多，可以接受。

**2️⃣ 高效支持“修改 + 查询”**

常见两种数据结构：

**(a) 二维树状数组（Fenwick Tree / BIT）**

* 思想：把一维 Fenwick Tree 推广到二维。
* 支持：
  + 单点更新：O(log n \* log m)
  + 任意矩形求和：四次前缀求和 → O(log n \* log m)
* 实现：bit[i][j] 保存某一块区域的和；更新和查询都是双重循环累加低位。

**(b) 二维线段树（Segment Tree）**

* 思想：把线段树在二维展开。
* 支持：
  + 单点更新或整行/整列区间更新
  + 矩形求和
* 时间复杂度：O(log n \* log m)（建树 O(nm)）。
* 但代码量大、常数大，不如二维 BIT 简洁。

**3️⃣ 如果更新是整块区域、查询是单点**

可以用**二维差分数组**配合前缀和；或者用线段树做懒标记。  
（但你说的是“修改某个数字”，所以更像单点更新。）

**4️⃣ 总结对比表**

| **数据结构** | **支持的操作** | **查询复杂度** | **更新复杂度** | **适用场景** |
| --- | --- | --- | --- | --- |
| 二维前缀和 | 矩形求和 | O(1) | O(nm)（全重算） | 查询多更新少 |
| 二维 BIT | 矩形求和+单点更新 | O(log n log m) | O(log n log m) | 常用、实现简单 |
| 二维线段树 | 矩形求和+单点或区间更新 | O(log n log m) | O(log n log m) | 功能最强大 |

**推荐**

如果只是**单点更新 + 矩形求和**，最实用的是 **二维树状数组**。  
如果还要支持**矩形更新 + 矩形求和**，可以用 **二维树状数组的差分技巧** 或 **二维线段树带懒标记**。

疑问：

1. 什么是二维树状数组和二维线段树？
2. 他们是怎么解决修改之后整片区域的前缀和需要重算这个问题的？
3. 什么是差分技巧和懒标记？