1224 最大子矩阵

题目描述

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵，你的任务是找到最大的非空(大小至少是1×1)子矩阵。

比如，如下4×4的矩阵

```
 0  -2  -7   0
 9   2  -6   2
-4   1  -4   1
-1   8   0  -2
```

的最大子矩阵是

```
 9  2
-4  1
-1  8
```

这个子矩阵的大小是15。

输入

输入是一个N×N的矩阵。输入的第一行给出N(0<N<=100)。再后面的若干行中，依次(首先从左到右给出第一行的N个整数，再从左到右给出第二行的N个整数……)给出矩阵中的N2个整数，整数之间由空白字符分隔(空格或者空行)。已知矩阵中整数的范围都在[-127,127]。

输出

输出最大子矩阵的大小。

输入样例

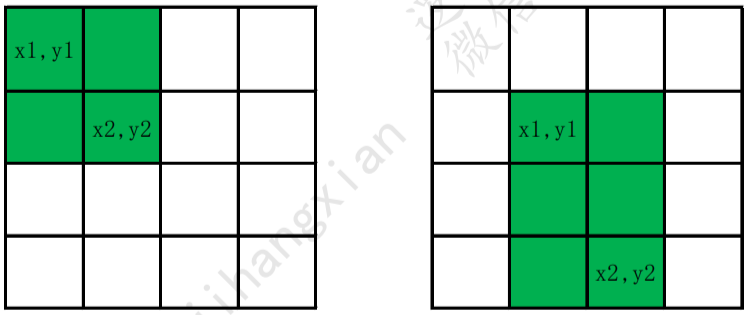
```
4
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

输出样例

```
15
```

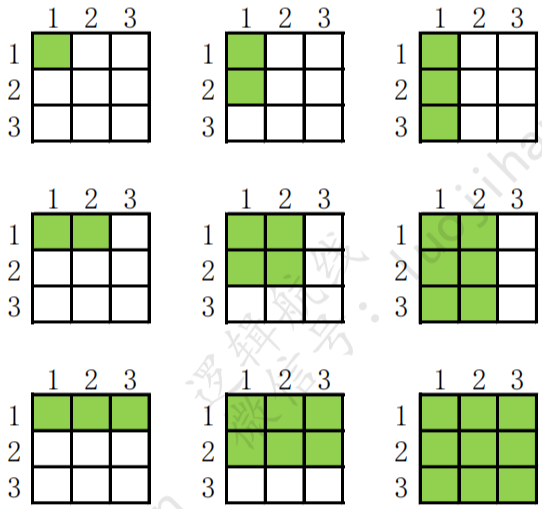
解析

最大子矩阵的一般思路就是枚举全部的组合，即找到所有(x1,y1)，(x2,y2)之间的组合，然后比较其中最大的一个。例如下图所示：

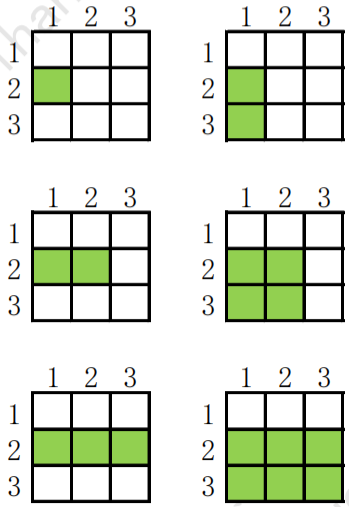


对于一般的矩阵，我们应按照如下方法进行遍历：

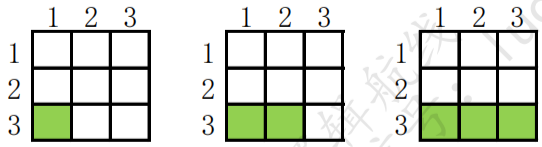
首先，我们将起点定在1,1处，然后不断的枚举终止点，y值最先增长。



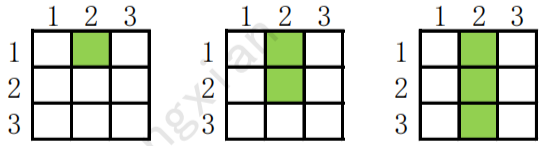
起始点在1,2处，枚举终点。

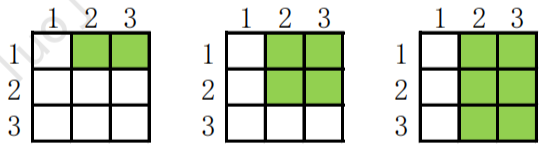


起始点在1,3处，枚举终点。

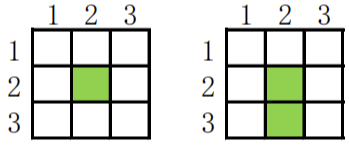


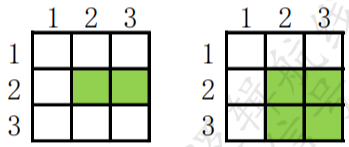
接下来，我们起点的x应该发生变化了，具体表现如下：





接下来，我们的起始y应该发生变化。





就这样，我们不断的枚举起点和终点的位置，计算由这个范围组成的矩形和，最终就能够选择出一个最大的值。

基本的暴力法代码如下：

```
#include <bits/stdc++.h>
using namespace std;
//矩阵数组
int matrix[101][101];

int main(int argc, char **argv) {
    //读入数据
    int n, max = 0;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &matrix[i][j]);
        }
```

```
        }
        //枚举左上顶点
        for (int x1 = 1; x1 <= n; x1++) {
            for (int y1 = 1; y1 <= n; y1++) {
                //枚举右下顶点
                for (int x2 = x1; x2 <= n; x2++) {
                    for (int y2 = y1; y2 <= n; y2++) {
                        //开始计算这个范围内的和
                        int ans = 0;
                        for (int i = x1; i <= x2; i++) {
                            for (int j = y1; j <= y2; j++) {
                                ans += matrix[i][j];//累加
                            }
                        }
                        //和全局最大值进行比较，并更新最大值
                        if (max < ans) {
                            max = ans;
                        }
                    }
                }
            }
        }
        cout << max;
        return 0;
    }
```

很明显，以上计算的代码效率及其低下，在这里我们需要对其进行优化。

前缀和优化

前缀和

我们先来学习一个知识点：前缀和。

前缀和是指从第1个元素到第i个元素的总和，则题目中的数据就可以表示如下：

原始数组

索引	0	1	2	3	4
1	1	1	1	1	1

接下来，我们按照行来计算前缀和，则得到了新的数组pre，如下所示，即每一行从左往右的值，均是前面的所有数字之和。

例如：pre[3](图中蓝色部分) 便是原数组0-3项的总和。

我们若想得到任意两个索引i,j之间的和，只需要使用pre[j]-pre[i-1]即可，非常方便。

举例，我们想求索引pre[2]与pre[3]之和（图中紫色部分）：

我们只需使用pre[3](图中蓝色部分)-pre[1](图中绿色部分) = 3-1 = 2

前缀和数组Pre

索引	0	1	2	3	4
1	0	1	2	3	4

接下来，我们看一下如何使用前缀和来进行优化矩阵和部分。首先，我们先来求解第一个格子的值。

绿色部分是我们想要求的数值，它等于前缀和pre[1][1] - pre[1][0](第一维为y轴)

	0	1	2	3	4
1	0	0	-2	-9	-9
2	0	9	11	5	7
3	0	-4	-3	-7	-6
4	0	-1	7	7	5

然后，我们该计算第1列第2行的总和了，它等于：

(pre[1][1] - pre[1][0]) + (pre[2][1] - pre[2][0)

如下所示：

	0	1	2	3	4
1	0	0	-2	-9	-9
2	0	9	11	5	7
3	0	-4	-3	-7	-6
4	0	-1	7	7	5

在这里，我们将原本所需的两重循环变更为使用一重循环，即只遍历y方向的数值，而减少了x方向的遍历，这样便提高了性能。

注意：矩阵的索引必须从1开始，否则当索引-1的时候会出现异常。

一维前缀和编码

```
#include <bits/stdc++.h>

using namespace std;
```

```
//矩阵数组
int matrix[501][501];
//前缀和数组
int pre[501][501];

int main(int argc, char **argv) {
    //读入数据
    int n;
    //求全局最大值，因为结果可能为负，所以需要赋值int的最小值
    int max = -INT_MAX;
    scanf("%d", &n);
    for (int y = 1; y <= n; y++) {
        for (int x = 1; x <= n; x++) {
            scanf("%d", &matrix[y][x]);
            pre[y][x] = pre[y][x - 1] + matrix[y][x];
        }
    }

    //枚举左上顶点
    for (int x1 = 1; x1 <= n; x1++) {
        for (int y1 = 1; y1 <= n; y1++) {
            //枚举右下顶点
          for (int x2 = x1; x2 <= n; x2++) {
                for (int y2 = y1; y2 <= n; y2++) {
                    //开始计算这个范围内的和
                 int ans = 0;
                    for (int y = y1; y <= y2; y++) {
                        ans += pre[y][x2] - pre[y][x1 - 1];
                    }
                    //和全局最大值进行比较，并更新最大值
                    if (max < ans) {
                        max = ans;
                    }
                }
            }
        }
    }
    cout << max;
    return 0;
}
```

二维前缀和优化

在理解了一维前缀和的优化之后，有没有更好的方法呢？答案是一定的，那就是二维前缀和。即将某一区块的数据直接进行存储。在这里我们用区块范围内的右下角的坐标的数组索引来存储当前区块的整体和。

现有如下图1数组，我们用点(1,1)(第一维是纵向)来存储当前绿色区块的数值，则有图2结果。以此类推，我们可以得到如下图3

图1

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图2

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图3

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	1	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

接下来，我们开始处理第二行，还是将目标点周围的四个数字进行相加，此时点(2,1)的值为2，一切看起来都很正常，我们继续后移，得到图5，此时得到的结果为6！我们回到图1口算，这个值实际上应该为4啊，哪里错了？

图4

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图5

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	6	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图1

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

原因如下：

点(1,2)所代表的值是图6中绿色部分，点(2,1)的值是图7中绿色部分，计算点(2,2)的值时如果直接让两个绿色部分相加，则必然重复计算了一次红色1，所以，我们在这里需要将其减掉。

图6

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图7

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

图8

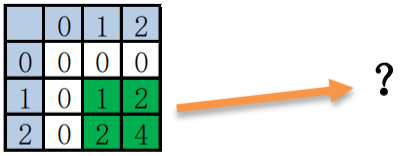
	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	1	1	1
3	0	1	1	1	1
4	0	1	1	1	1

因此，最终的二维前缀和公式如下：

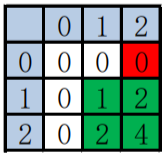
某点所代表的前缀和 = 正上方前缀和值 + 正左方前缀和值 - 左上方前缀和值 + 当前点值

pre[i][j] = pre[i-1][j] + pre[i][j-1] - pre[i][j] + matrix[i][j]。

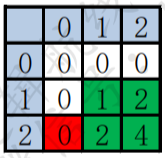
现在我们有了二维前缀和数组，该如何快速的求解某一区块的前缀和呢？如下图所示，这是一个前缀和数组，我怎么才能快速的计算出区块点(1,1)到点(2,2)的前缀和呢？



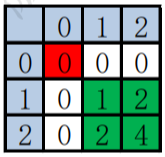
既然要求绿色区块，所以，我们需要将不在这个区块内的数据删除，很明显应该删除上边缘和左边缘的数据，因为它们已经包含在点(2,2)中了。



减去点(0,2)即为删除上边缘以上的全部数据。



减去点(2,0)即为删除左边缘以上的全部数据。



此时，我们会发现左上角的点被减了两次，应该被加回来

最终则有：

某点所代表的区块前缀和 = 当前点的实际前缀和 - 正上方前缀和值 - 正左方前缀和值 + 左上方前缀和值

```
ans = pre[i][j] - pre[i-1][j] - pre[i][j-1] + pre[i][j];
```

二维前缀和比一维在循环上又少了一层，即不用再对y值进行遍历，所以速度更快。

二维前缀和编码

```
#include <bits/stdc++.h>

using namespace std;

//矩阵数组
int matrix[501][501];
//前缀和数组
int pre[501][501];

int main(int argc, char **argv) {
    //读入数据
    int n;
    //求全局最大值，因为结果可能为负，所以需要赋值int的最小值
```

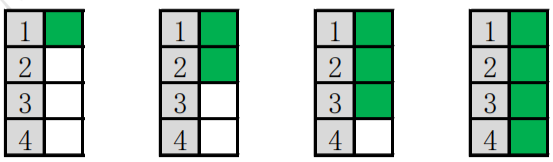
```
    int max = -INT_MAX;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &matrix[i][j]);
            pre[i][j] = pre[i - 1][j] + pre[i][j - 1]
                    - pre[i - 1][j - 1] + matrix[i][j];
        }
    }


    //枚举左上顶点
    for (int x1 = 1; x1 <= n; x1++) {
        for (int y1 = 1; y1 <= n; y1++) {
            //枚举右下顶点
          for (int x2 = x1; x2 <= n; x2++) {
                for (int y2 = y1; y2 <= n; y2++) {
                    //开始计算这个范围内的和
                int ans = pre[x2][y2] - pre[x1 - 1][y2]
                            - pre[x2][y1 - 1] + pre[x1 - 1][y1 - 1];
                    //和全局最大值进行比较，并更新最大值
                if (max < ans) {
                        max = ans;
                    }
                }
            }
        }
    }
    cout << max;
    return 0;
}
```

使用贪心算法对前缀和进行优化

如果你此刻认真的思考过了，你会发现不断的枚举y的起始点和终止点，意义似乎不大。事实上，我们其实只需要枚举一次y的数值就可以了。

如下图所示：

y从1开始遍历，绿色为当前的和。



在这里你可能会提出疑问，那以第2行作为起点的区块呢？我们不计算了吗？

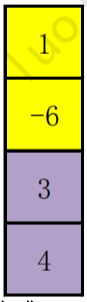
没错，我们确实不计算了。而这正式贪心算法的奥妙所在。

观察下图，试想如果绿色区域的值大于0，那么蓝色加上绿色就一定比蓝色大，这时，我们就可以用蓝绿之和与前次的计算作比较（前次计算的结果就是绿色小块）取二者中较大的一个。

但是，如果绿色区域的值小于0，那么我们直接就将其抛弃。因为蓝色方块加上一个小于0的值必然小于自己本身，我们不必计算。

抛弃之后，我们重新以蓝色区块的数值作为基础，继续向下求和。

这种方式使得我们不用逐一的去遍历Y值，大大的提高了效率。如下所示：黄色区域的和是-5，舍弃，我们直接从紫色开始计算，最终得到最大值7。



一维前缀和贪心优化

```
#include <bits/stdc++.h>
using namespace std;
//矩阵数组
int matrix[501][501];
//前缀和数组
int pre[501][501];
int main(int argc, char **argv) {
    //读入数据
    int n;
    //求全局最大值，因为结果可能为负，所以需要赋值int的最小值
    int max = -INT_MAX;
    scanf("%d", &n);
    for (int y = 1; y <= n; y++) {
        for (int x = 1; x <= n; x++) {
            scanf("%d", &matrix[y][x]);
            pre[y][x] = pre[y][x - 1] + matrix[y][x];
        }
    }
    //枚举起始列数
    for (int x1 = 1; x1 <= n; ++x1) {
        //枚举终止列数
        for (int x2 = x1; x2 <= n; x2++) {
            int ans = 0;
            //枚举起始的行数
            for (int y = 1; y <= n; y++) {
```

```
                ans += pre[y][x2] - pre[y][x1 - 1];
                //更新最大值
                if (ans > max) {
                    max = ans;
                }
                //贪心的要点，当前的答案如果小于0，对后面的值肯定有影响，所以
                if (ans < 0) {
                    ans = 0;
                }
            }
        }
    }
    cout << max;
    return 0;
}
```

二维前缀和贪心优化

```
#include <bits/stdc++.h>
using namespace std;
//矩阵数组
int matrix[501][501];
//前缀和数组
int pre[501][501];
int main(int argc, char **argv) {
    //读入数据
    int n;
    //求全局最大值，因为结果可能为负，所以需要赋值int的最小值
    int max = -INT_MAX;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &matrix[i][j]);
            pre[i][j] = pre[i - 1][j] + pre[i][j - 1] -
                    pre[i - 1][j - 1] + matrix[i][j];
        }
    }
    int ans;
    //枚举起始x坐标
    for (int x1 = 1; x1 <= n; x1++) {
        //枚举终止x坐标
        for (int x2 = x1; x2 <= n; x2++) {
            ans = 0;
            for (int y = 1; y <= n; y++) {
                //开始计算这个范围内的和
                ans += pre[x2][y] - pre[x1 - 1][y] -
                        pre[x2][y - 1] + pre[x1 - 1][y - 1];
                //和全局最大值进行比较，并更新最大值
```

```
                if (max < ans) {
                    max = ans;
                }
                if (ans < 0) {
                    ans = 0;
                }
            }
        }
    }
    cout << max;
    return 0;
}
```