[方格取数]

```
方格中的左上角出发。往下走取数。
走两次，求两次的数字和的最大值。
```

$20mins$

- 首先想到两次取最大。
- 简单验证之后，
- 发现并不成立。

$solve()$

- 四维dp:设计状态，
- 两个人一起走，当前状态：$f_{i,j.l.r}$表示当前第一个人在(i,j)的位置上第二个人的位置在(l,r)的位置上。
- 状态转移符方程：
  $f_{i,j,l,r} = max(f_{i-1,j,l-1,k}, f_{i,j-1,k-1,l}, f_{i,j-1,k,l-1});$
  注意如果最终两个人在同一个点上看，要注意一个数字不能重复拿。
- $amazing$!!!!!

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

void MAIN();
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);
    MAIN();
}
typedef long long ll;
const int maxn = 12;
//------code-----٩(ˊωˋ*)و ------靓仔代码-----٩(ˊωˋ*)و ----talk is cheap , show me the code--------

int g[maxn][maxn];
int f[maxn][maxn][maxn][maxn]; //状态记录

void MAIN()
{
    int n;
    cin >> n;
    //状态转义符方程的顺序性问题。
    while (true)
    {
        int x, y, z;
        cin >> x >> y >> z;
        if (x == 0)
```

```
                break;
            g[x][y] = z;
        }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            for (int l = 1; l <= n; l++)
                for (int r = 1; r <= n; r++)
                {
                    f[i][j][l][r] = max({f[i - 1][j][l - 1][r],
                                         f[i][j - 1][l - 1][r], f[i - 1][j][l][r
- 1],                                            f[i][j - 1][l][r - 1]}) + g[i]
[j] + g[l][r];
                    if (i == l && j == r)
                        f[i][j][l][r] -= g[i][j];
                }
    cout << f[n][n][n][n] << '\n';
}
```

$growing$

- 生长思考:
  - 四维dp的这一个状态怎么想出来的?
  - 探究解空间。
    - 本质上就是条路径。
      - 所有的子问题，路径的终点分别来到一个点的最优状态，向上转移。
      - 然后感受到状态转移方程的的正确定，可以正确的解决每一个子问题。、、
- 可以进行类比比较的问题：传纸条问题：

传纸条问题：$three$

[传纸条]

```
两人在教室上互相传纸条，
除了终点两条路径不重叠。
每一个电视行有一个对应的数字。
求取满足前提的两条路径上的最大和。
```

$20mins$

```
类似的思路： 状态转移方法。
如果保证了当前子更小规模的解的子问题，不会重叠，那么新构造出来的解时，只要两个点不一致，新解决的问题中，它的解也不会重叠。

另外的问题，初始化的问题，关于转移情况的问题
            转移顺序的问题：
            滚动数组优化的问题：
况且这一个问题的数据范围高达50.4次方的dp直接爆炸。
还要考虑关于状态转移的优化问题。
在这一个过程当中是否有一些非必要的计算？
```

solve

- 第一种朴素想法：
    - 定义$f_{i,j,k,r}$表示当前两人在什么位置的情况。
    - 关于重复的问题，由于两个人的到达同一个点，如果不能重复贡献，肯定不是最优的，最终每一个状态只要终点不相同，计算出来的结果的方案肯定是会是重叠的。
    - 最终的复杂度为$O(n^4)$不知道。大概会完蛋。
    - 但是事实上由于数据太水，最终并没有完蛋。
- 另外一种优化的角度。
    - 优化子问题的定义：
        - $f_{sum,i,j}$表示当前两个落点的横坐标，纵坐标之和。
        - 发现这种迁移之下的一个重要的性质是：两者的横纵坐标之和加1.所以只要这样逐步迁移直到最终解决问题$f_{n+m,m,m}即可。
    - 方法的复杂度为$O(n^2 * (n + m))$
- 实现细节
    - 注意迁移过程中红防止越界越界。防止盘外的点也被计算，导致该状态的解不为0进一步影响了其它状态的解决。

---

code

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

void MAIN();
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);
    MAIN();
}
typedef long long ll;
const int maxn = 51;
//------code------٩(ˊωˋ*)و ------靓仔代码------٩(ˊωˋ*)و ----talk is cheap , show me the code--------

int g[51][51];
int f[maxn << 1][maxn][maxn];

void MAIN()
{
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> g[i][j];
    for (int sum = 3; sum <= n + m; sum++)
        for (int i = 1; i <= min(sum - 1, m); i++) //表示纵坐标情况，两个人都要另外讨论。
            for (int j = max(sum - n, 1); j <= min(sum - 1, m); j++)
            {
                f[sum][i][j] = max({f[sum - 1][i][j], f[sum - 1][i - 1][j],
```

```
                                    f[sum - 1][i][j - 1], f[sum - 1][i - 1][j -
1]}) +
                               g[sum - i][i] + g[sum - j][j];
                if (i == j)
                    f[sum][i][j] -= g[sum - i][i];
            }
    cout << f[n + m][m][m] << '\n';
}
```