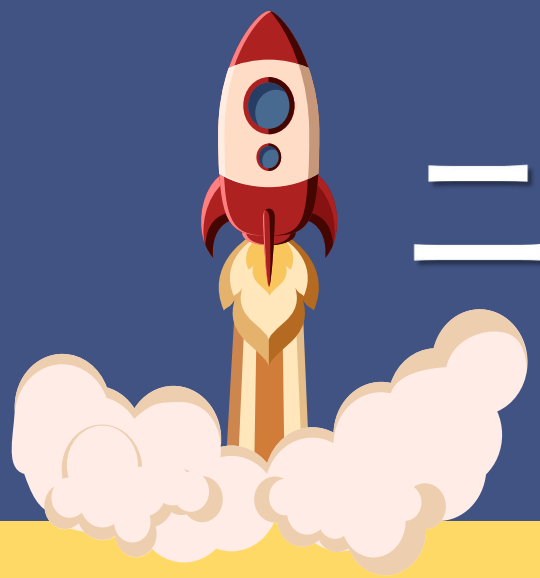


二维前缀和 && 二阶差分

NKOJ4299，4300，2143

1.二维前缀和

子矩阵之和1 NKOJ4299

NKOJ4299 子矩阵之和1

给出一个n*m的矩阵，该矩阵全由整数构成。

有q个询问，格式是x1 y1 x2 y2 表示询问以(x1,y1)和(x2,y2)为对角线的子矩阵中的数字总和。

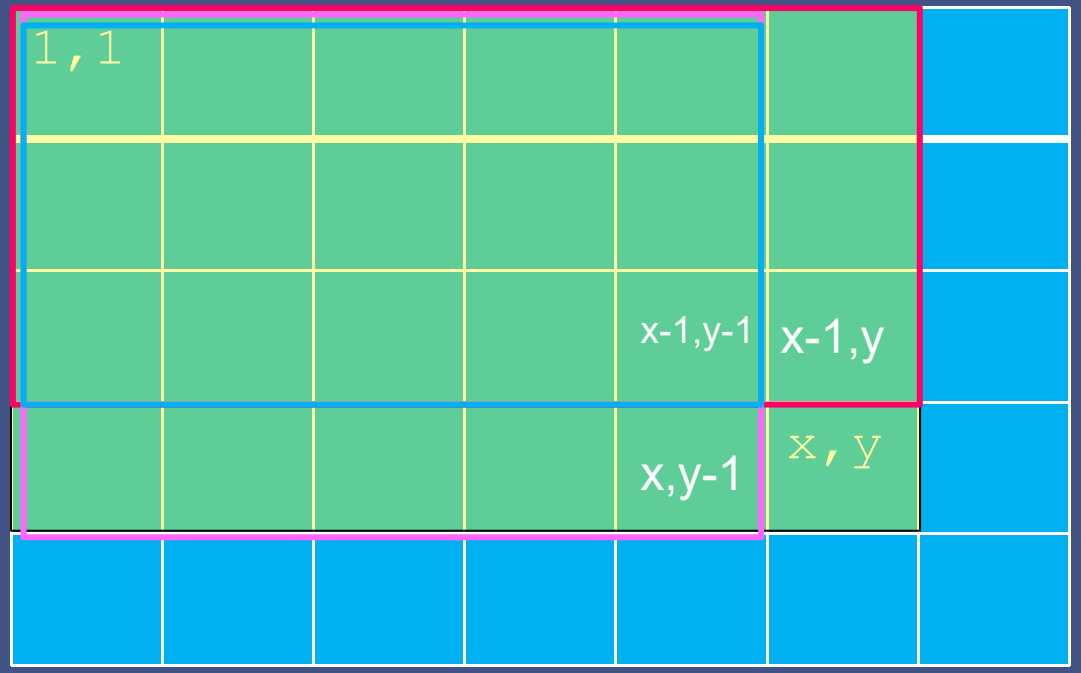
x1<=x2，y1<=y2

1<=n,m<=1000

1<=q<=10000

NKOJ4299 子矩阵之和1

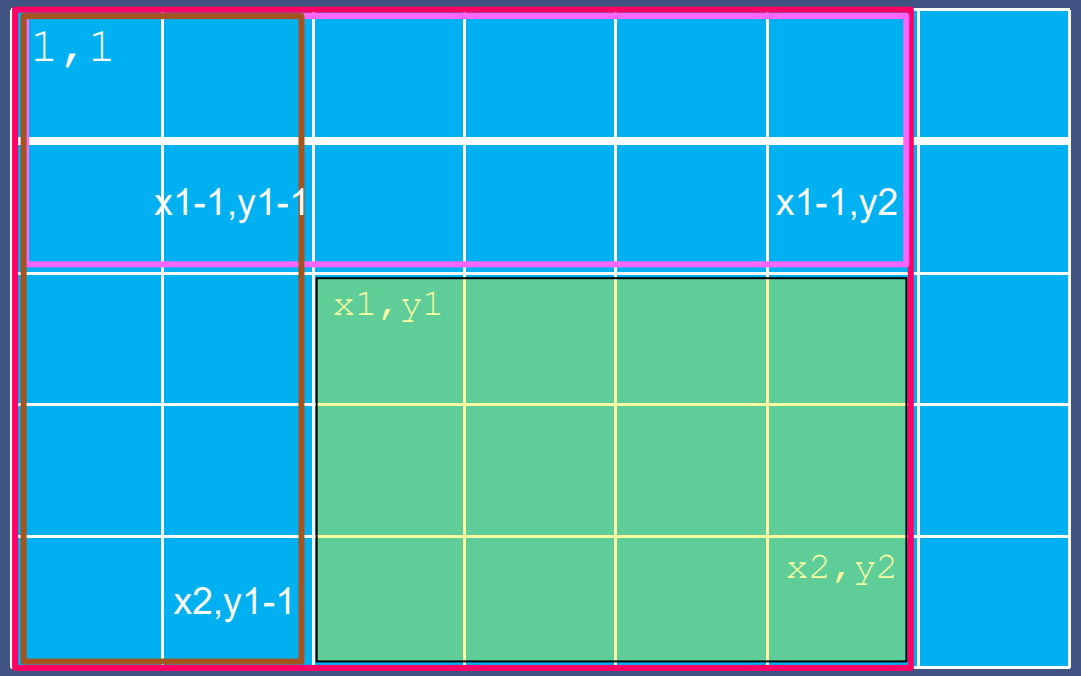
二维前缀和
设Sum[x][y]表示以(1,1) (x,y)为对角线的子矩阵的数字总和。




```
Sum[x][y]=Sum[x-1][y]+Sum[x][y-1]-Sum[x-1][y-1]+a[x][y]
```

NKOJ4299 子矩阵之和1

二维前缀和
设Sum[x][y]表示以(1,1) (x,y)为对角线的子矩阵的数字总和。
询问子矩阵(x1,y1) (x2,y2)的数字总和




```
ans=Sum[x2][y2]-Sum[x1-1][y2]-Sum[x2][y1-1]+Sum[x1-1][y1-1]
```

NKOJ4299 子矩阵之和1

```
for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
        Sum[i][j]=1LL*(Sum[i-1][j]+Sum[i][j-1]-Sum[i-1][j-1]+a[i][j]);

for(i=1;i<=q;i++)
{
      ans=0;
      scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
      ans=Sum[x2][y2]-Sum[x1-1][y2]-Sum[x2][y1-1]+Sum[x1-1][y1-1];
      printf("%lld\n",ans);
}
```

2.二维差分

子矩阵之和2 NKOJ4300

NKOJ4300 子矩阵之和2

给出一个n*m的矩阵，该矩阵全由整数构成。
先有p个1号操作：
操作1.格式是x1 y1 x2 y2 k 表示以(x1,y1)和(x2,y2)为对角线的子矩阵中的每个数字都增加k

接着有q个2号操作：
操作2.格式是x1 y1 x2 y2 表示询问以(x1,y1)和(x2,y2)为对角线的子矩阵中的数字总和。对于每个询问，输出计算结果。

x1<=x2，y1<=y2
1<=n,m<=1000
1<=q<=50000

NKOJ4300 子矩阵之和2

回忆一维差分数组的情形：

申明一差分数组d[ ]，d的作用用来记录某一个位置上的总改变量。**d[i]表示的是i~n这些元素都要加上d[i]这个数。**

我们对[L,R]区间整体进行加k的操作，只需在d[L]处加一个k，然后在d[R+1]处就减去一个k。最后求序列的每个位置变成了多少，只需要求一下d数组的前缀和，然后和原数组按位相加就好。

现在考虑二维的情况：

一个n*m的矩阵，要求支持操作add(x1,y1,x2,y2,k)，表示对于以(x1,y1)为左上角，(x2,y2)为右下角的矩形区域，每个元素都加上k。

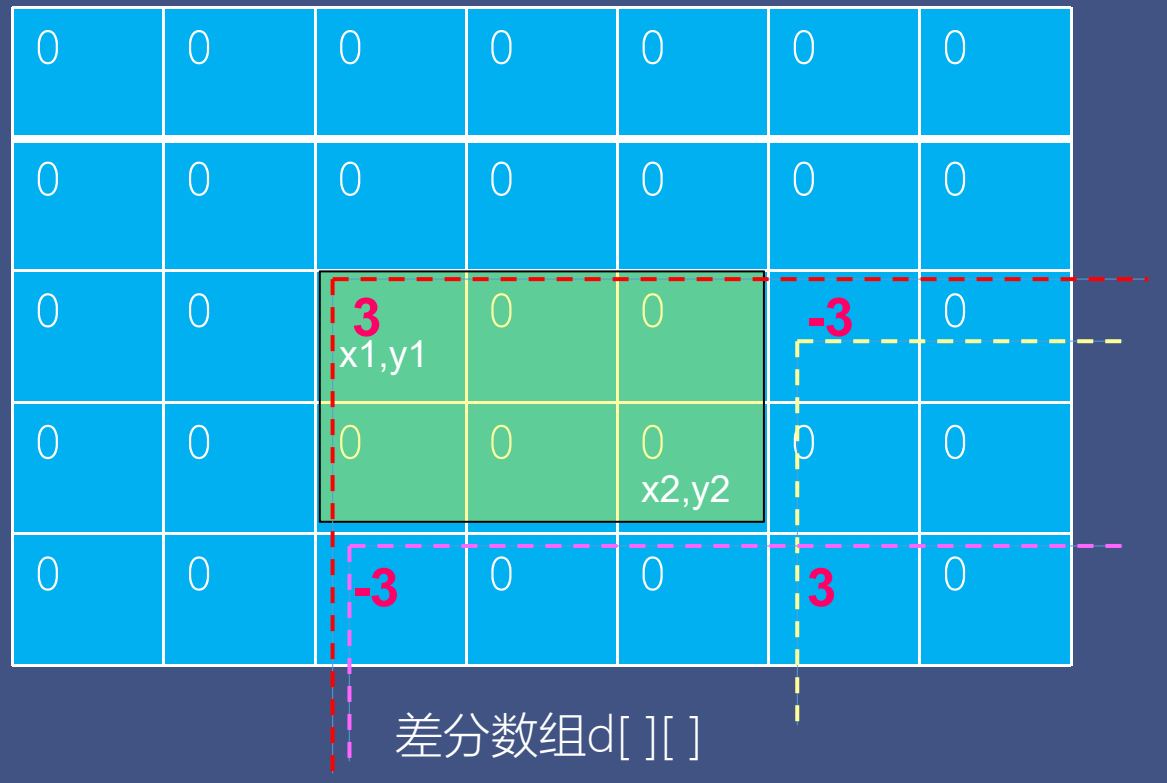
我们的做法和一维类似。用数组d[ ]存储每个位置上总改变量。

在d[x1][y1]处加上k，在d[x2+1][y1]和d[x1][y2+1]处减k，在d[x2+1][y2+1]再加上k。

最后(x,y)位置上的修改的总数值就是d数组在(x,y)位置的二维前缀和。

NKOJ4300 子矩阵之和2

现在考虑二维的情况：
在d[x1][y1]处加上k，在d[x2+1][y1]和d[x1][y2+1]处减k，在d[x2+1][y2+1]再加上k。
最后(x,y)位置上的修改的总数值就是d数组在(x,y)位置的二维前缀和。



差分数组d[ ][ ]

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	3	3	3	0	0
0	0	3	3	3	0	0
0	0	0	0	0	0	0

差分数组的二维前缀和SumD[ ][ ]

NKOJ4300 子矩阵之和2

```
//处理差分数组D[ ],将矩阵x1,y1,x2,y2整体增加k
for(i=1;i<=p;i++)
{
        cin>>x1>>y1>>x2>>y2>>k;
        D[x1][y1]+=k;
        D[x2+1][y1]-=k;
        D[x1][y2+1]-=k;
        D[x2+1][y2+1]+=k;
}
```

```
//求差分数组的二维前缀和SumD[i][j]表示坐标(i,j)处总的修改量
for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
    {
        SumD[i][j]=SumD[i-1][j]+SumD[i][j-1]-SumD[i-1][j-1]+D[i][j];   //差分数组二维前缀和
        A[i][j]+=SumD[i][j];                                            //计算i,j位置最后的值
        S[i][j]=S[i-1][j]+S[i][j-1]-S[i-1][j-1]+A[i][j];                //计算修改后的矩阵的二维前缀和
     }
```

NKOJ4300 子矩阵之和2

```
//处理询问，按前例的二维前缀和处理即可
   for(k=1;k<=q;k++)
   {
         cin>>x1>y1>>x2>>y2;
         ans=S[x2][y2]-S[x1-1][y2]-S[x2][y1-1]+S[x1-1][y1-1];
         cout<<ans;
   }
```

3.应用例子

打地鼠 NKOJ2143

打地鼠 NKOJ2143

给定一个m*n的洞穴矩阵，每个洞穴里面有若干地鼠，我们需要自行选定一个r*c的锤子进行击打，每次击打必须保证r*c的范围内所有洞穴均有地鼠，且每次击打只会打掉每个洞穴恰好一只地鼠，求最小击打次数。
m,n<=100

考虑一个1*8的洞穴，当我们把锤子设作1*4时可以完成击打，而1*3不能 故不满足单调性，二分不正确

打地鼠 NKOJ2143

既然这题没有二分的性质，那我们就由大到小暴力枚举锤子的长i和宽j，然后去判断是否合适。

```
int ans=1;
for(i=m; i>=1; i--)
    for(j=n; j>=1; j--)
        if(i*j>ans && tot%(i*j)==0 && Check(i,j)) ans=i*j;
        //锤子尺寸越大，需要挥锤的次数就越少

cout<<tot/Area;     //tot为总的地鼠数量
```

//关键是Check(i,j)怎么写?

打地鼠 NKOJ2143

挥锤子实际上，就是一个二维区间修改+单点查询。O（n^2）。

```
bool Check(int x,int y)
{
    int i,j;
    memset(s,0,sizeof(s));
    for (i=1; i<=m; i++)          //以(i,j)位置为左上角对角线起点，锤x*y的区域
        for (j=1; j<=n; j++)
        {
            s[i][j]+=s[i-1][j]+s[i][j-1]-s[i-1][j-1];    //差分前缀和
            if (s[i][j]>a[i][j]) return 0;          //s[i][j]为(i,j)位置已有的修改值。即之前被砸总次数。
            int t=a[i][j]-s[i][j];
            if ((i>m-x+1 || j>n-y+1) && t) return 0;   //(i,j)需要砸，但又砸不下
            if (t){ s[i][j]+=t; s[i+x][j]-=t; s[i][j+y]-=t; s[i+x][j+y]+=t; }
        }    //(i,j)出发的大小为x*y的子矩阵至少要砸t次，即砸完(i,j)位置的所有地鼠
    return true;
}
```