算法基础（二）:前缀和与差分，双指针

前缀和

基本思想

原数组：$a_1, a_2, a_3.....a_n,$

称$S_i = a_1, a_2, a_3.....a_i$ 为前缀和

1. 求前缀和的公式：
    1. `s[i] = s[i -1] + a[i]`，从前递推即可，`s[0] = 0`
2. 作用：
    1. 可以快速求原数组的一段的和，当需要计算原数组的`[l, r]`的和的时候，不用从`l`加到`r`，只需要用$S_r - S_{l-1}$即可，将时间复杂度降低
3. 为什么从`1`开始？
    1. 当计算`[1, 10]`的和的时候可以用`s[10] - s[0]`得出，保持求所有的区间的一段和的时候形式一致
    2. 若从`0`开始，则求`[0, 10]`的时候需要特判，而从`1`开始不会出现下标为`0`的区间和

代码实现

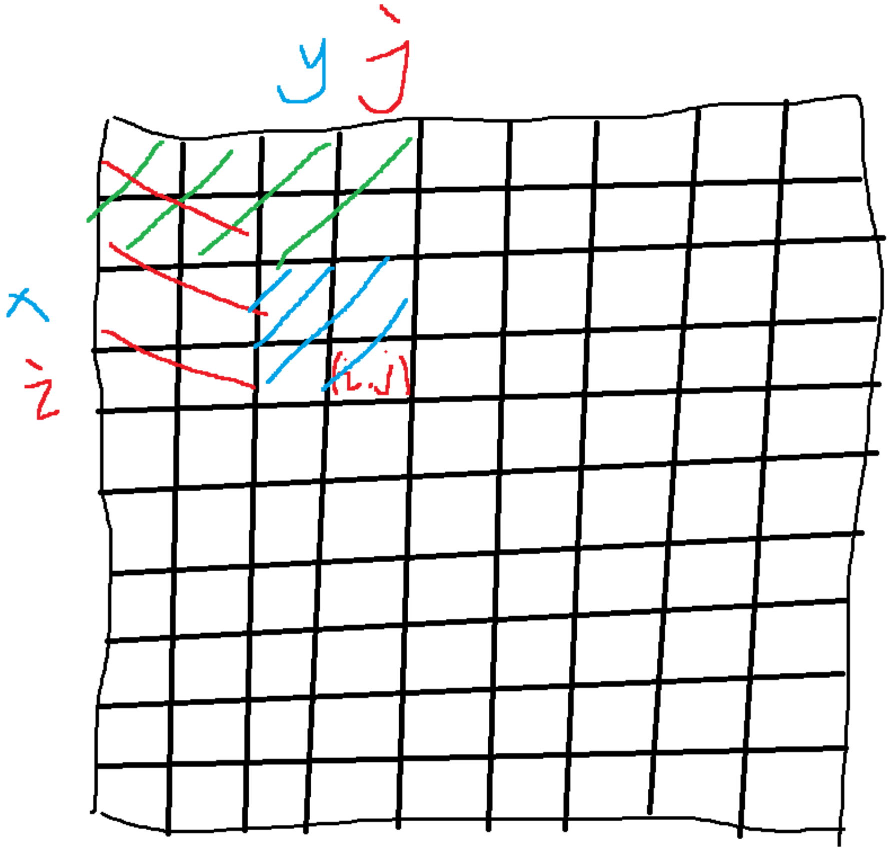
[795. 前缀和 - AcWing题库]

```
#include<iostream>
```

```cpp
using namespace std;
const int N 100010;
int n, m;
//a表示数组，s表示前缀和
int a[N], s[N];
int main(){
    s[0] = 0;
    scanf("%d%d", &n, &m);
    //n表示数组中的n个数
    for(int i = 1; i <= n; i++){
        scanf("%d",&a[i]);
        s[i] = s[i-1] + a[i];
    }
    //输入m个查询
    while(m--){
        int l, r;
        scanf("%d%d",&l,&r);
        printf("%d\n", s[r] - s[l-1]);
    }
    return 0;

}
```

二维前缀和

基本思想



1. 应用：
    1. 求部分和，一个小方块代表一个数组元素，`s[i][j]`表示下标为`i,j`的左上角的矩形数组的所有的和，我们要求`x,y到i,j`的小矩形的和，也就是蓝色矩形的和，计算的公式为：`s = s[i][j] - s[x-1][j] - s[i][y-1] + s[x-1][y-1]`
2. 根据矩阵`a`递推求其前缀和`s`的公式：
    1. `s[i][j] = a[i][j] + s[i-1][j] + s[i][j-1] - s[i-1][j-1]`

代码实现

[796. 子矩阵的和 - AcWing题库]

```
#include<iostream>
using namespace std;
const int N = 1010;

int n, m, q;
int a[N][N], s[N][N];

int main(){

    scanf("%d%d%d", &n, &m, &q);

    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            scanf("%d",&a[i][j]);
```

```
            //cout << a[i][j] <<" ";
        }
        //cout << endl;
    }
    //这种递推求和方式可以保证以i，j为下标的矩形除了整块矩形之外，其内部的其他所有矩形的和都被求了出来
    for(int i = 1; i <= n; i ++){
        for(int j = 1; j <= m; j++){
            s[i][j] = s[i-1][j] + s[i][j-1] + a[i][j] - s[i-1][j-1];
        }
    }

    while(q--){
        int x1, y1, x2, y2;
        scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
        printf("%d\n", s[x2][y2] - s[x2][y1-1] - s[x1-1][y2] + s[x1-1][y1-1]);
    }
    return 0;
}
```

差分

基本思想

数组`a[1], a[2], a[3] ...a[n]`，数组`b[1], b[2], b[3]...b[n]`，其中，`a[i] = b[1] + b[2] + b[3]....b[i]`，称`b`是`a`的差分，相当于是前缀和的逆运算

在前缀和数组`a`中，若要求：区间`[l, r]`中的数全部加上`c`即：`a[l]+c, ....a[r]+c`，得到一个新的数组`a1`，其对应的新差分数组`b1` 是原来的差分数组`b`的`b[l]+c`以及`b[r+1] -c` 后的结果

证明：

1. `b[l] + c`这样会导致他的前缀和相当于`a`来说是从`a[l]`到`a[n]`的所有元素都会加上`c`，此时再`b[r+1] -c`，其前缀和相当于`a`是让`a[r+1]`以及以后的元素都再减去`c`从而保持不变得到`a1`

在构造`a`的差分数组的时候，若`a1[1....n] = 0`则显然其差分数组为`b1[1......n]`为0， 那么数组`a`可以看作在数组`a1`的每个区间`[i, i]`插入`a[i]`后的结果，所以`a`对应的差分数组就是对原来的差分数组`b1`进行`b1[i] + a[i], b1[i+1] - a[i]`这样我们知道了`a`就可以直接求出其差分数组`b`

应用：

1. 可以快速的将一个数组中的`[l, r]`中的元素都加上`c`，只需要对其差分数组进行上述操作
    1. 先求其差分数组
    2. 再对差分数组进行上述操作
    3. 再根据求一维前缀和的公式，求操作后的差分数组的前缀和，即得

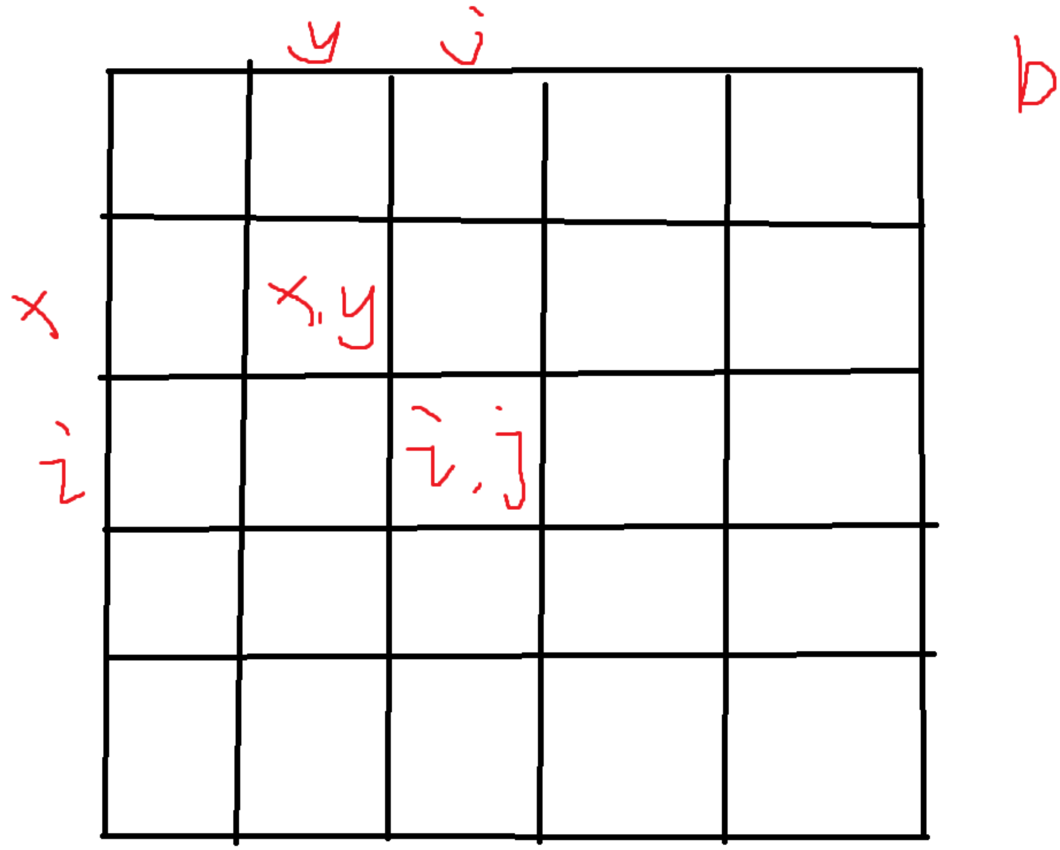
代码实现

[797. 差分 - AcWing题库]()

```cpp
#include<iostream>

using namespace std;

const int N = 100010;

int m, n;
int a[N], b[N];

void insert(int l, int r, int c){
    b[l] += c;
    b[r + 1] -= c;
}

int main(){

    scanf("%d%d", &n, &m);

    for(int i = 1; i <= n; i++){
        scanf("%d", &a[i]);
    }

    //构造a[i]的差分数组b[i],看作是从全0的数组中每个区间[i, i]插入a[i]
    for(int i = 1; i <= n; i++){
        insert(i, i, a[i]);
    }

    //m个插入操作
    while(m--){
        int l, r, c;
        scanf("%d%d%d",&l, &r, &c);
        insert(l, r, c);
    }

    //将b[i]变成自己的前缀和
    for(int i = 1; i <= n; i++){
        b[i] = b[i] + b[i-1];
        cout << b[i] <<" ";
    }
    return 0;

}
```

二维差分

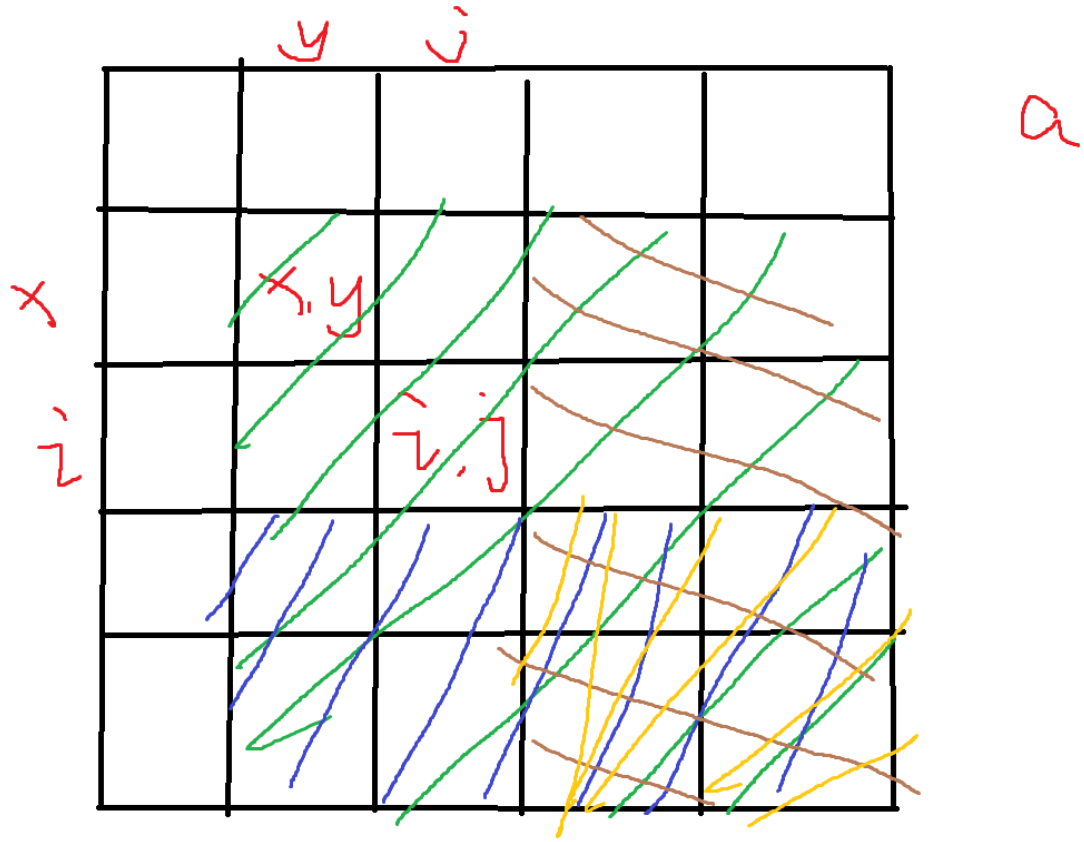
基本思想



此时`b[i][j]`是一个矩阵，`a[i][j]`是`b`的前缀和，表示以`b[i][j]`为右下角的左上方小矩阵的和

我们此时的要求是：将`a`矩阵的一个小方块内的所有的数加上一个值得到`a1`

做法是：将`a`的差分矩阵`b`进行以下四个操作得到`b1`，此时`b1`就是`a1`的差分矩阵，再对`b1`求前缀和就得到了`a1`



证明：

1. `b[x][y] + c` 后其前缀和相当于 `a` 来说是整个绿色部分的 `a` 加上 `c`
2. `b[x][j+1]-c` 后其前缀和相当于 `a` 来说是整个棕色的 `a` 恢复到原来的值
3. `b[i+1][y]-x` 后其前缀和相当于 `a` 来说是整个蓝色的 `a` 恢复到原来的值，且黄色部分多减了一次 `c`
4. 此时 `b` 再进行一次操作 `b[i+1][j+1]+c`，其前缀和相当于 `a` 来说是以 `a[x][y],a[i][j]` 为对角元素的矩阵中的每一个元素都加上 `c`，于是得到的矩阵 `a1`

所以当知道了前缀和矩阵 `a` 来求差分矩阵 `b` 的时候可以看作是原来有两个矩阵 `a1 = {0},b1 = {0}`,然后，对 `a1` 的每个小矩阵 `[i][i]` 插入 `a[i][i]`,接着对 `b1` 进行以上操作就得到了 `b` 矩阵

应用：

1. 对一个矩阵的某个子矩阵的所有的元素加上 `c`
   1. 先求出这个矩阵的差分矩阵
   2. 然后对差分矩阵进行上述操作
   3. 再对操作后的差分矩阵求其前缀和矩阵即得

代码实现

[AcWing 798. 差分矩阵 - AcWing]

```
#include<iostream>
#include<vector>
```

```

int n, m, q;
const int N = 1010;
int a[N][N], b[N][N];

using namespace std;

void insert(int x1, int y1, int x2, int y2, int c){
    b[x1][y1] += c;
    b[x2+1][y1] -= c;
    b[x1][y2+1] -= c;
    b[x2+1][y2+1] += c;
}

int main(){
    scanf("%d%d%d", &n, &m, &q);


    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            scanf("%d",&a[i][j]);
        }
    }

    //构造差分矩阵b
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            insert(i, j, i, j, a[i][j]);
        }
    }

    //输入q个插入
    while(q--){
        int x1, y1, x2, y2, c;
        scanf("%d%d%d%d%d", &x1, &y1, &x2, &y2,&c);
        insert(x1, y1, x2, y2, c);
    }

    //根据处理后的差分矩阵得到最后的前缀和矩阵，注意求前缀和的公式
    for(int i = 1; i<= n; i++){
        for(int j = 1; j <= m; j++){
            b[i][j] = b[i][j] + b[i-1][j] + b[i][j-1] - b[i-1][j-1];
            printf("%d ", b[i][j]);
        }
        printf("\n");
    }


    return 0;
}
```

双指针

基本思想

快排和归并排序得时候用到

核心得性质：用来优化时间复杂度，一般得话就从暴力做法的`O(n^2)`复杂度降低到`O(n)`

代码实现

```
for(int i = 0, j = 0; i < n; i++){
    while(j < i && check(i, j)){
        j++;
    }
    //剩下是每道题的具体逻辑
}
```

[799. 最长连续不重复子序列 - AcWing题库]()

很经典的双指针题目

基本思想是，`i`往后移动，让`j`总是指向以`i`为末尾的最长不重复序列的第一个数，`i`不断移动，不断比较，从而找到最长的序列长度。

一个巧妙的思想是另开一个数组使用哈希策略来记录重复。

可以以`1 2 2 3 5`这个例子来帮助理解

```
#include<iostream>
using namespace std;
const int N = 100010;

int a[N], s[N];

int main(){

    int n ;
    cin >> n;

    for(int i = 0; i < n; i++){
        cin >> a[i];
    }

    int res = 0;
    for(int i = 0, j = 0; i < n; i++){
        //a[i]对应下标的s[a[i]]++
        s[a[i]]++;
        //若s[a[i]] > 1说明i移动到了重复元素,比如1 2 2 3，i移动到了第二个2

        while(s[a[i]] > 1){
            //此时移动j，让s[]中的那些下标归零，重新计数
```

```
                //并且跳出循环的时候j必指向i的位置
                s[a[j]]--;
                j++;
            }

            //每次移动i都进行一次比较
            res = max(res, i - j + 1);
        }
        cout << res <<endl;
        return 0;
}
```

[2816. 判断子序列 - AcWing题库]()

判断子序列也是一个经典的双指针问题

基本思想就是：

1. 从前往后遍历 `b` 数组，每次用最前面的一个数与 `a` 数组进行匹配，若能将 `a` 数组里的数匹配完，就说明 `a` 的确是 `b` 的一个子序列，称找到了一个匹配
2. 若 `b` 遍历完了但是还是没有匹配完，就说明 `a` 不是 `b` 的子序列，称这个匹配不存在

下面来证明这个算法的正确性，需要注意的是：

1. 我们若能用上述算法的确找到一个匹配，当然可以说明 `a` 是 `b` 的子序列即：

$$\text{若用这个算法能找到一个匹配} => \text{这个匹配存在（}a\text{是}b\text{的子序列）}$$

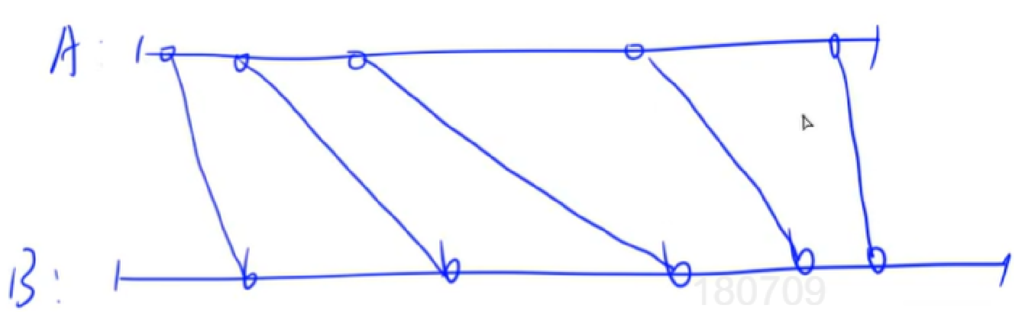
2. 但能找到并不代表一定可以找到，即：

$$\text{存在一个匹配} \neq> \text{这个算法一定可以找到该匹配}$$

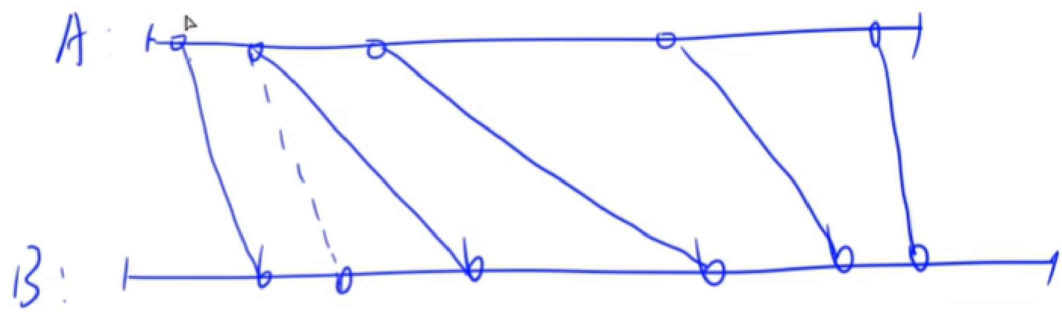
也就是说，当这个算法失效的时候，有可能匹配仍然存在，所以我们证明该算法的正确性的意思就是，证明存在一个匹配的时候，这个算法一定可以找到

证明：

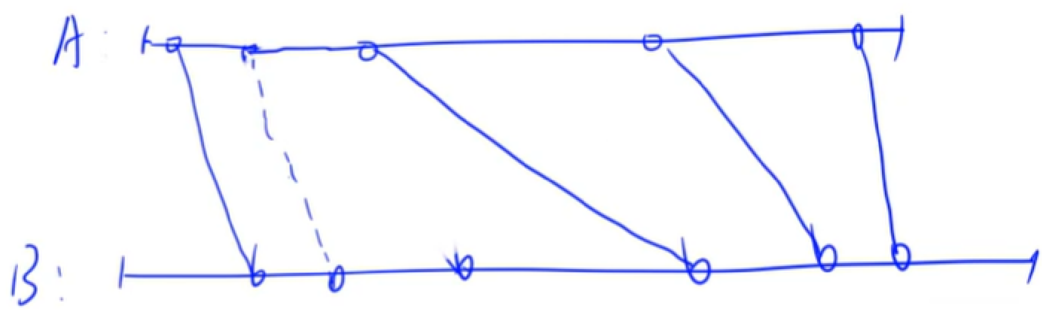
假设存在一个匹配，即，`a` 是 `b` 的子序列，关系如下，圆圈代表匹配的元素：



我们用算法进行匹配，假设第一个元素匹配符合该算法，当到第二个元素时，由于原匹配存在，所以用该算法必然可以匹配一个元素，要么是原匹配的元素要么匹配的元素是`b`中的别的元素，由于元素的有序性，该元素一定在原匹配元素的前面，这里用虚线表示：



这时，我们可以调整`b`中的匹配元素，将`b`中匹配`a`第二个元素的实线删掉而只保留虚线，这样的修改不会改变后面的匹配，而前面的匹配仍然是合法的



于是，只要`b`中存在一个原匹配，我们就一定可以通过这样的修改，得到一个算法匹配，并且两者等价，所以，只要`b`存在一个原匹配，我们使用算法就一定可以找到一个与原匹配等价的匹配，也就证明了算法与原匹配的等价性。

代码实现：

```
#include<iostream>
using namespace std;
const int N = 100010;
int a[N], b[N];

int main(){
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < n; i++) cin >> a[i];
    for(int i = 0; i < m; i++) cin >> b[i];
    int i = 0, j = 0;
    while(i < n && j < m){
        if(a[i] == b[j]) i++;
```

```
        j++;
    }
    if(i == n){//i遍历完了，说明匹配成功
        cout << "Yes";
    }else{//i没有遍历完，j遍历完了，说明匹配失败
        cout << "No";
    }
    return 0;
}
```