2022 香农先修班第三次课

排序

排序理论

排序算法 (Sorting algorithm)，是一种将一组特定的数据按某种顺序进行排列的算法。

> 例如，下面的操作是排序：
>
> - 给定若干个实数，从小到大排列输出
> - 给定若干个字符串，按字典序排序输出
> - 给定若干个学生各科分数，先按总分，同分依次按每科分数高低排序直到可以区分为止 `OJ 1072 排名`

稳定性 指作为比较依据的值相同的多个元素的相对位置在排序前后是否不变。不变的是**稳定排序**，变的是**不稳定排序**。

通俗地讲就是能保证排序前两个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。在简单形式化一下，如果$A_i = A_j$，Ai原来在位置前，排序后Ai还是要在Aj位置前。

> 例如，有五个二元数据 (x, y)：$(1, 10), (2, 5), (5, 10), (4, 12), (3, 10)$，以 y 为依据排序，若排序后有：
>
> $$(2, 5), (1, 10), (5, 10), (3, 10), (4, 12)$$
>
> 则，这样的排序是稳定排序；若中间三个元素不是按这个顺序的，是不稳定排序。

常见的稳定和不稳定排序有：

- 稳定排序：冒泡排序、插入排序、计数排序、归并排序……
- 不稳定排序：选择排序、快速排序、堆排序……

> 判断排序是否是稳定排序的方法：若无不作为比较依据的值，设辅助标记(如下标)形成结构体(若有不作为比较依据的值就直接用就行了)，进行结构体排序；然后用任意稳定排序算法再排序一次，逐项比较这两种排序的结果，若完全一致就是稳定，不一致就是不稳定。复杂度为两次排序复杂度 + 逐项比较复杂度。

常见排序算法表：

方法	平均	最坏	最好	空间	方式	稳定
冒泡	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	in	Y
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	in	N
插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	in	Y
希尔	$O(n^{1.25})$	$O(n^2)$	$O(n)$	$O(1)$	in	N
归并	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	out	Y
快速	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(\log n)$	in	N
堆	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	in	N
计数	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	out	Y
桶	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	out	Y
基数	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	out	Y

> 各经典排序动图过程演示可参考 这里

逆序对：对数组 a ，满足 $i<j, a_i>a_j$ 的两个数。逆序对数目(**逆序数**)能体现数组错乱程度。

一种求法：暴力枚举所有满足 $i<j$ 的两数之对逐个判定即可。时间复杂度为 $O(n^2)$

> 实际上常用的求法在下文

> 性质：冒泡排序的交换次数为逆序数。

常用排序

归并排序

merge sort

实现思路：

- 将数组划分为平衡的两个子数组 (即长度相差不大于 1)
- 对两个子数组实施归并排序，使两个子数组分别有序
- 合并两个有序子数组为单个有序数组

使用**滑动窗口法**合并两个有序子数组：

- 左右子数组、新数组当前下标为起始值
- 若当前下标左数组元素小于等于右数组元素，把当前左数组元素放到新数组，左数组、新数组下标自增
- 否则，把当前右数组元素放到新数组，右数组、新数组下标自增
- 当一边子数组访问完毕后，将另一边子数组所有元素先后放到新数组后面即可

一种参考手写代码：(洛谷 P1177)

```
#include <bits/stdc++.h>
using namespace std;
#define mn 100010
typedef long long ll;
ll n, a[mn], b[mn];
void mergesort(ll lf, ll rf)
{
    if (lf < rf)
    {
        ll cf = (lf + rf) >> 1; //left-face; center-face; right-face
        mergesort(lf, cf);
        mergesort(cf + 1, rf);
        ll i = lf, j = cf + 1, je = rf, ie = cf, k = 0; //左[i,ie],右[j,je]
        while (i <= ie && j <= je)
        {
            if (a[i] <= a[j])
            {
                b[k++] = a[i++];
            }
            else
            {
                b[k++] = a[j++];
            }
        }
        while (i <= ie)
        {
            b[k++] = a[i++];
        }
        while (j <= je)
        {
            b[k++] = a[j++];
        }
        for (ll h = 0; h < k; ++h)
        {
            a[lf + h] = b[h];
        }
    }
}
signed main()
{
    scanf("%lld", &n);
    for (ll i = 1; i <= n; ++i)
    {
        scanf("%lld", &a[i]);
    }
    mergesort(1, n);
```

```
    for (ll i = 1; i <= n; ++i)
    {
        printf("%lld ", a[i]);
    }
    return 0;
}
```

时间复杂度分析：根据主定理有 $a = b = 2, k = 1$ ，$a = b^k$ ，所以 $\mathrm{O}(n^k log_b n) = \mathrm{O}(n \log n)$

空间复杂度： $\mathrm{O}(n)$

性质：是稳定排序

用途：以 $\mathrm{O}(n \log n)$ 求逆序对数目

- 滑动窗口合并时，若发现左数组元素大于右数组元素，说明形成逆序，由于左数组有序，即左数组接下来的元素必然也都大于该有数组元素。

  当前左下标到左结尾共有 $ie - i + 1$ 个元素，它们都大于当前右元素，贡献这么多个逆序对

即：

```
while (i <= ie && j <= je)
{
    if (a[i] <= a[j])
    {
        b[k++] = a[i++];
    }
    else
    {
        ans += ie - i + 1;
        b[k++] = a[j++];
    }
}
```

> 逆序对也可以用线段树、树状数组求，大佬可以自行尝试

快速排序

Quick sort

快速排序抽象定义：

- 将数组划分为两个子数组，使得任意左子数组元素小于所有右子数组元素

- 对左右子数组依次执行快速排序，使得两个子数组分别有序

  由于上列性质，当两个子数组分别有序时，直接拼接起来就是有序的数组

对所有基于比较的排序，快速排序是理论上平均复杂度最优的排序。

实现思路：(以**三路快速排序**为例)

- 对长为 n 的待排序数组 a ，随机选择一个数组元素作基准值 p
- 所有小于基准值 p 的元素移动到数组的前面

  所有大于基准值 p 的元素移动到数组的后面
- 对所有小于基准值元素组成的子数组和所有大于基准值元素组成的子数组作为待排序数组，返回第一步

一种参考手写代码：(洛谷 P1177)

```
#include <bits/stdc++.h>
using namespace std;
#define mn 100010
typedef long long ll;
void quicksort(ll *a, ll n)
{
    if (n <= 1)
    {
        return;
    }
    ll lf = 0, rf = n, i = 0, p = a[rand() % n];
    while (i < rf)
    {
        if (a[i] < p)
        {
            swap(a[i++], a[lf++]);
        }
        else if (a[i] > p)
        {
            swap(a[i], a[--rf]);
        }
        else
        {
            ++i;
        }
    }
    quicksort(a, lf);
    quicksort(a + rf, n - rf);
}
ll n, a[mn];
signed main()
{

    scanf("%lld", &n);
```

```
    for (ll i = 1; i <= n; ++i)
    {
        scanf("%lld", &a[i]);
    }
    quicksort(a + 1, n);
    for (ll i = 1; i <= n; ++i)
    {
        printf("%lld ", a[i]);
    }
    return 0;
}
```

> 这个实现的空间复杂度是 $\mathrm{O}(1)$；根据具体实现不同，空间复杂度会有不同

性质：不稳定排序，平均复杂度 $\mathrm{O}(n \log n)$，最差复杂度 $\mathrm{O}(n^2)$

> 复杂度分析：
>
> 1. 最容易分析的情况是每次左右都分到一半(相差不大于 1)，可以近似看成两边相等，此时，根据主定理，$a = b = 2, k = 1, a = b^k$，复杂度为 $\mathrm{O}(n^k \log_b n) = \mathrm{O}(n \log n)$
>
>   事实上，这种情况是最优情况
>
> 2. 最坏的情况是每次某一边都是空，那么每次长度只减少 1，需要递归 $\mathrm{O}(n)$ 次，每次跑一层循环，复杂度为 $\mathrm{O}(n^2)$
>
>   如果不使用随机基准，可以构造特殊数组使得每次基准都是最值；使用随机基准时，每次都取最值的概率约为 $\frac{2}{n!} \approx 0$ (注：计算不严谨，实际概率表达式远比这个复杂)
>
> 3. 平均情况的严格证明相对复杂 (具体参见文章 这里，前置知识：高等数学下-调和级数，各种高中数列知识)
>
>   (p.s. 这已经是算法理论证明里相当易懂的证明步骤了，多数算法的证明远比这个复杂)
>
>   直观上判断，可以看出平均复杂度介于最优和最坏之间
>
> 4. 实践表明，绝大多数情况下平均约等于最优
>
>   (还记得一开始的那句话吗：快速排序是理论上平均复杂度最优的排序)

std::sort

复杂度

库 `algorithm` ；当然万能头即可。

实现方法：**内省排序** (intro sort)，是快速排序和堆排序、插入排序的结合。当递归深度超过 $[\log_2 n]$ 时使用堆排序，当元素数量少于阈值(如 16) 时，用插入排序；否则使用快速排序。由于堆排序最差复杂度是 $\mathrm{O}(n\log n)$ ，所以内省排序的最差复杂度是 $\mathrm{O}(n\log n)$ 。

> 基本数据类型比较的复杂度是 $\mathrm{O}(1)$ ，所以复杂度如上；若每次比较复杂度是 f ，时间复杂度是 $\mathrm{O}(fn\log n)$ 。

由于最大递归层数是 $\log n$ ，每层递归起码需要一个变量参数，所以空间复杂度起码是 $\mathrm{O}(\log n)$

> 小数据时用插入排序，是因为插入排序在几乎有序时的复杂度约为 $\mathrm{O}(n)$

> 具体原理参见 这里

> 对于大的、乱数列表一般相信是最快的已知排序

使用方法

```
sort(头迭代器，尾迭代器[，比较依据函数]);
```

将会对迭代器范围 [头, 尾) 进行排序，依据是比较函数或已定义的比较运算符。**默认升序排序**。

1. 对于数组，头迭代器是首个元素指针地址；尾迭代器是最后一个元素指针地址的下一个地址
2. 对于迭代器，通常使用 `.begin()` , `.end()` 代表上述两个含义
3. 对于结构体等没有定义大小关系运算符的数据类型，可以增加比较函数，或者重载运算符

> 例：
>
> 1. 给一维整型数组 a 下标范围 $[0,n)$ 的元素升序排序
>
> ```
> sort(a, a + n);
> ```
>
> 2. 给一维双精度浮点型数组 a 下标范围 $[1,n]$ 的元素升序排序

```
sort(a + 1, a + 1 + n)
```

3. 给二维长整数数组 b 第 $[1,n]$ 行每行下标范围 $[1,m]$ 的元素升序排序

```
for (ll i = 1; i <= n; ++i)
{
    sort(b[i] + 1, b[i] + 1 + m);
}
```

4. 给一维整型向量 v 所有元素升序排序

```
sort(v.begin(), v.end());
```

5. 给一维整型数组 a 下标范围 $[1,n]$ 的元素逆序排序

```
sort(a + 1, a + 1 + n);
reverse(a + 1, a + 1 + n);
```

```
sort(a + 1, a + 1 + n, [](const ll &x, const ll &y)
     { return x > y; }); //重定义 x < y 关系
```

```
bool cmp(const ll &x, const ll &y) // ll x, ll y 也行，上同
{
    return x > y; //不要写成 x-y因为负数也是true
}
//...
sort(a + 1, a + 1 + n, cmp);
```

6. 给二元整数对 (x,y) 向量升序 d 排序，先按 x 大小，x 相同时按 y 大小

```
// vector<pair<ll, ll>> d = {{2, 2}, {1, 2}, {-1, -2}, {1, 1}, {2, 3}};
vector<pair<ll, ll>> d; //pair本身的比较运算符就是如此定义的
sort(d.begin(), d.end());
// for (auto i : d)
// {
//     printf("%lld %lld\n", i.first, i.second);
// }
```

```cpp
struct node
{
    ll x, y;
    //bool operator<(node r) 但是不可以仅定义>不定义<
    bool operator<(const node &r) const
    {
        if (x != r.x)
        {
            return x < r.x;
        }
        return y < r.y;
    }
};
// vector<node> d = {{2, 2}, {1, 2}, {-1, -2}, {1, 1}, {2, 3}};
vector<node> d;
sort(d.begin(), d.end());
// for (auto i : d)
// {
//     printf("%lld %lld\n", i.x, i.y);
// }
```

```cpp
struct node
{
    ll x, y;
};
bool cmp(const node &l, const node &r)
{ //即 l.x != r.x ? l.x < r.x : l.y < r.y;
    if (l.x != r.x)
    {
        return l.x < r.x;
    }
    return l.y < r.y;
}
vector<node> d = {{2, 2}, {1, 2}, {-1, -2}, {1, 1}, {2, 3}};
signed main()
{
    sort(d.begin(), d.end(), cmp);
    for (auto i : d)
    {
        printf("%lld %lld\n", i.x, i.y);
    }
    return 0;
}
```

```cpp
struct node
{
    ll x, y;
};
vector<node> d = {{2, 2}, {1, 2}, {-1, -2}, {1, 1}, {2, 3}};
signed main()
{
    sort(d.begin(), d.end(), [](const node &l, const node &r)
```

```
            { return l.x != r.x ? l.x < r.x : l.y < r.y; });
        for (auto i : d)
        {
            printf("%lld %lld\n", i.x, i.y);
        }
        return 0;
    }
```

7. OJ1072 排名

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll n, m;
struct student
{
    string name;
    ll sum;
    vector<ll> subj;
    bool operator<(student r)
    {
        return sum != r.sum ? sum > r.sum : subj > r.subj;
    }
} s[1010];
signed main()
{
    cin >> n >> m;
    for (ll i = 0; i < n; ++i)
    {
        cin >> s[i].name;
        for (ll j = 0, v; j < m; ++j)
        {
            cin >> v;
            s[i].sum += v;
            s[i].subj.emplace_back(v);
        }
    }
    sort(s, s + n);
    for (ll i = 0; i < n; ++i)
    {
        cout << s[i].name << ' ';
    }
    return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll n, m;
struct student
{
    string name;
    ll sum;
```

```
    vector<ll> subj;
    bool operator<(student r)
    {
        if (sum != r.sum)
        {
            return sum > r.sum;
        }
        for (ll i = 0; i < m; ++i)
        {
            if (subj[i] != r.subj[i])
            {
                return subj[i] > r.subj[i];
            }
        }
    }
} s[1010];
signed main()
{
    cin >> n >> m;
    for (ll i = 0; i < n; ++i)
    {
        cin >> s[i].name;
        for (ll j = 0, v; j < m; ++j)
        {
            cin >> v;
            s[i].sum += v;
            s[i].subj.emplace_back(v);
        }
    }
    sort(s, s + n);
    for (ll i = 0; i < n; ++i)
    {
        cout << s[i].name << ' ';
    }
    return 0;
}
```

时间复杂度分析：每次比较最差复杂度 $\mathrm{O}(m)$，故时间复杂度为 $\mathrm{O}(nm\log n)$

其他函数

nth_element

`std::nth_element(first, nth, last[, compare])`

使得迭代器 `nth` 指向的第 $nth-first+1$ 大元素恰在该位置

时间复杂度 $\mathrm{O}(n)$

如：洛谷 P1923 求第 k 小的数

```cpp
#include<bits/stdc++.h>
using namespace std;
long long n,k,a[5000010];
int main()
{
    scanf("%d%d",&n,&k);
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    nth_element(a,a+k,a+n);//使第k小整数就位
    printf("%d",a[k]);//调用第k小整数
}
```

> 想看具体实现思路找洛谷题解区别的题解

STL

xx_bound

对升序序列：

`lower_bound(begin, end, v)` 在 $[begin, end)$ 找第一个大于等于 v 的值，返回该值的迭代器

`upper_bound(begin, end, v)` 在 $[begin, end)$ 找第一个大于 v 的值，返回该值的迭代器

若找不到返回 end

对降序序列：

`lower_bound(begin, end, v, greater<type>())` 在 $[begin, end)$ 找第一个小于等于 v 的值，返回该值的迭代器

`upper_bound(begin, end, v, greater<type>())` 在 $[begin, end)$ 找第一个小于 v 的值，返回该值的迭代器

若找不到返回 end

时间复杂度： $O(\log n)$

> 例：SCNUOJ 1216 二分查找(区间计数)

请尝试手写实现二分查找。

给定一个长度为 n 的序列 a，你需要处理 m 个询问。

对于每一个询问，给定两个整数 l 和 r，你需要回答序列中有多少项满足 $l \leq a_i \leq r$。

输入

第一行包含两个整数 n, m $(1 \leq n, m \leq 10^5)$，含义如题目所示。

第二行包含 n 个用空格间隔的整数 $a_1, a_2, ..., a_n$ $(-10^9 \leq a_i \leq 10^9)$，表示序列 a。

接下来 m 行每行包含两个用空格间隔的整数 l, r $(-10^9 \leq l \leq r \leq 10^9)$，含义如题目描述所示。

输出

对于每个询问，输出一行，包含一个整数，表示答案。

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll n, m, a[100010], l, r;
signed main()
{
    scanf("%lld%lld", &n, &m);
    for (ll i = 1; i <= n; ++i)
    {
        scanf("%lld", &a[i]);
    }
    sort(a + 1, a + 1 + n);
    while (m--)
    {
        scanf("%lld%lld", &l, &r);
        auto pos1 = lower_bound(a + 1, a + 1 + n, l);
        auto pos2 = upper_bound(a + 1, a + 1 + n, r);
        printf("%lld\n", pos2 - pos1);
        // printf("%lld %lld\n", pos1 - a , pos2 - a );
    }
    return 0;
}
```

前缀和与差分

一维前缀和

定义：

前缀和可以简单理解为「数列的前 项的和」，是一种重要的预处理方式，能大大降低查询的时间复杂度。

例题：

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 $B[i]$ 是原数组 A 第 0 到第 i 个数的和。

输入：

```
5
1 2 3 4 5
```

输出：

```
1 3 6 10 15
```

```cpp
#include <iostream>
using namespace std;

int N, A[10000], B[10000];

int main() {
    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }

    // 前缀和数组的第一项和原数组的第一项是相等的。
    B[0] = A[0];

    for (int i = 1; i < N; i++) {
        // 前缀和数组的第 i 项 = 原数组的 0 到 i-1 项的和 + 原数组的第 i 项。
        B[i] = B[i - 1] + A[i];
    }

    for (int i = 0; i < N; i++) {
        cout << B[i] << " ";
    }

    return 0;

}
```

知识点：

- 多次询问求区间和，对其进行预处理

- ```
  sum[i]=sum[i-1]+a[i]
  ```

- 计算平均值与方差

  设前缀和 $a_i = \sum_{j=1}^{i} x_j$ ， 平方前缀和数组 $b_i = \sum_{j=1}^{i} x_j^2$ ，由前缀和公式，有：

  $$\overline{x} = \frac{\sum_{i=l}^{r} x_i}{r-l+1} = \frac{\sum_{i=1}^{r} x_i - \sum_{i=1}^{l-1} x_i}{r-l+1} = \frac{a_r - a_{l-1}}{r-l+1}$$

  同理根据前缀和公式，原式有：

  $$s^2 = \frac{b_r - b_{l-1} - \overline{x}(a_r - a_{l-1})}{r-l+1}$$

  $$s = \sqrt{\frac{b_r - b_{l-1} - \overline{x}(a_r - a_{l-1})}{r-l+1}}$$

-

例：蓝桥杯热身赛 H异世界的稳定测试

```
#include<bits/stdc++.h>
using namespace std;
#define mn 200010
typedef long long ll;
ll n,m,s1[mn],s2[mn],x[mn],p,k,l,r,c;
double s;
int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++)
    {
        scanf("%lld",&x[i]);
        x[i+n]=x[i];
    }
    for(int i=1;i<=2*n;i++)
    {
        s1[i]=s1[i-1]+x[i]*x[i];
        s2[i]=s2[i-1]+x[i];
    }
    while(m--)
    {
        scanf("%lld",&c);
        if(c==1)
        {
            scanf("%lld",&k);
            p=(p+k)%n;
        }
        else
        {
            scanf("%lld%lld",&l,&r);
            double ave=(s2[r+p]-s2[l-1+p])*1.0/(r-l+1);
```
```

```
            s=sqrt((s1[r+p]-s1[l-1+p]-ave*(s2[r+p]-s2[l-1+p]))*1.0/(r-l+1));
            printf("%.6f\n",s);
        }
    }
    return 0;
}
```

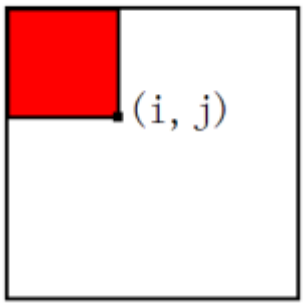
二维前缀和

定义：

二维前缀和顾名思义就是二维的前缀和，二维很显然了，有x轴和y轴也就是一个面，这很显然

那二维前缀和中一个$f[i][j]$表示的意思就是，以$(1,1)$为左上角以(i,j)为右下角这个矩阵里面数的和

如图：

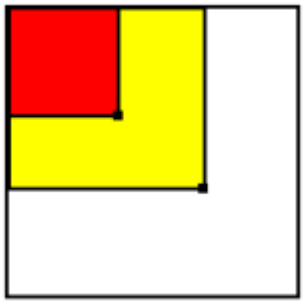


$f[i][j]$表示的就是图中红色的部分

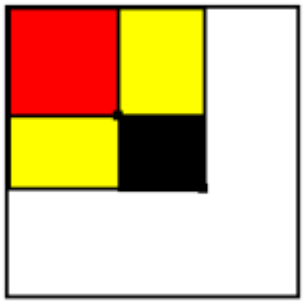
二维前缀和求矩阵元素和

二维前缀和可以用来干什么呢？一维前缀和你可以用来$O(1)$求某个点的值。那么类比一下，二维前缀和也是可以用来求某个矩阵的值的

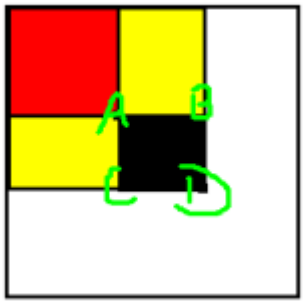
但是怎么来求呢？



就如图中，知道了两个点的位置和他们的二维前缀和，图中红色是左上角的那个点的二维前缀和，红色+黄色部分是右下角的那个点的二维前缀和，是不是可以用这个来求出他们之间的矩阵的和呢？也就是这一部分：



图中黑色的部分就是我们要求的那个矩阵和



D点表示的二维前缀和值是红色部分+两个黄色部分+黑色部分
A点表示的是红色部分
B点表示的是上面的黄色部分+红色部分
C点表示的是下面的黄色部分+红色部分

这里面只有D的前缀和里面包括黑色部分，只要减去D里面的那两个黄色部分和红色部分是不是就剩下了我们要求的黑色部分了？那怎么减去呢？可以这样：$D-B-C+A$
这就是二维前缀和最重要的部分了，化成二维数组的形式就是这样的：

$$f[i][j]-f[i-1][j]-f[i][j-1]+f[i-1][j-1]$$

具体求法：

这个可以类比上面求矩阵的思想 只是这个矩阵的右下角是$(i,\ j)$，左上角也是(i,j)，就是一个1×1的矩阵，所以也是很好求的，但是上面是已知D，A，B，C求黑色部分，这里你只知道A，B，C和黑色部分。因为是一个1×1的矩阵，所以黑色部分就只有一个元素也就是$(i,\ j)$坐标上的那个元素值，所以就可以个加法变减法，减法变加法一个性质的。通过A，B，C和黑色部分来求出D

D点表示的二维前缀和值是红色部分+两个黄色部分+黑色部分
A点表示的是红色部分
B点表示的是上面的黄色部分+红色部分
C点表示的是下面的黄色部分+红色部分

所以 D 就可以等于 $B+C-D+$ 黑色部分： 上面的黄色部分+红色部分+下面的黄色部分+红色部分-红色部分+黑色部分 =上面的黄色部分+红色部分+下面的黄色部分+黑色部分 刚好等于 D，方程式为：

$$f[i][j]=f[i-1][j]+f[i][j-1]-f[i-1][j-1]+a[i][j]$$

知识点：

- 二维前缀和中一个 $f[i][j]$ 表示的意思就是以 $(1,1)$ 为左上角以 (i,j) 为右下角这个矩阵里面数的和
- $f[i][j]$ 的预处理 $f[i][j]=f[i-1][j]+f[i][j-1]-f[i-1][j-1]+a[i][j]$
- ```
  for(int i=1;i<=n;i++)
      {
          for(int j=1;j<=m;j++)
          {
              cin>>a[i][j];
              f[i][j]=f[i-1][j]+f[i][j-1]-f[i-1][j-1]+a[i][j];
          }
      }
  ```
- ```
  f[i][j]-f[i-c][j]-f[i][j-c]+f[i-c][j-c];//求边长为c的矩形
  ```

模板：

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int Max = 1010;
int a[Max][Max];
int f[Max][Max];
signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int n,m,c;
    cin >> n >> m >> c;
    for(int i = 1; i <= n; ++ i)
```

```
    {
        for(int j = 1; j <= m; ++ j)
        {
            cin >> a[i][j];
            f[i][j] = f[i - 1][j] + f[i][j - 1] - f[i - 1][j - 1] + a[i][j];
        }
    }

    int q;
    cin >> q;
    while(q--)
    {
        int x1,x2,y1,y2;//x1,y1是左上角的坐标，另一对是右下角的坐标
        cin >> x1 >> y1 >> x2 >> y2;
        cout << f[x2][y2] - f[x1 - 1][y2] - f[x2][y1 - 1] + f[x1 - 1][y1 - 1];
    }

    return 0;

}
```

例：洛谷**P2004** 领地选择

```
#include<bits/stdc++.h>
using namespace std;
#define N 1010
typedef long long ll;
ll f[N][N],a[N][N],n,m,c,maxx,x,y;
int main()
{
    cin>>n>>m>>c;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            cin>>a[i][j];
            f[i][j]=f[i-1][j]+f[i][j-1]-f[i-1][j-1]+a[i][j];//预处理
        }
    }
    maxx=INT_MIN;
    for(int i=c;i<=n;i++)
    {
        for(int j=c;j<=m;j++)
        {
            if(f[i][j]-f[i-c][j]-f[i][j-c]+f[i-c][j-c]>maxx)
            {
                maxx=f[i][j]-f[i-c][j]-f[i][j-c]+f[i-c][j-c];
                x=i-c+1,y=j-c+1;
            }
        }
    }
    cout<<x<<" "<<y;
    return 0;
}
```

三维前缀和

PS : 与二维前缀和思想差不多

维护:

$$s[i][j][k] = s[i-1][j][k] + s[i][j-1][k] + s[i][j][k-1] - s[i-1][j-1][k] - s[i-1][j][k-1] - s[i][j-1][k-1] + s[i-1][j-1][k-1] + a[i][j][k]$$

总的来说，其实就是一个容斥。

求**[x1, y1, z1]** 到 **[x2, y2, z2]**的和：

$$s[x2][y2][z2] - s[x1-1][y2][z2] - s[x2][y1-1][z2] - s[x2][y2][z1-1] + s[x1-1][y1-1][z2] + s[x1-1][y2][z1-1] + s[x2][y1-1][z1-1] - s[x1-1][y1-1][z1-1]$$

一维差分

知识点：

- 区间数值多次修改一次询问
- 若区间 $[l, r]$ 里面修改数值，只需 $a[l] += c, a[r+1] -= c$

例：牛客寒假训练**4 C**题蓝彗星

小红是一个天文学爱好者。她最喜欢的业余活动就是天体观测。
这天夜里，星汉灿烂。小红拉上了小梦前往了教学楼的天台，组装好了天文望远镜。
根据预测，有两种不同颜色的彗星：红彗星和蓝彗星。每颗彗星会在某一时间段出现 t 秒然后消失。
小红想知道，自己总共有多少秒，能看到蓝彗星且看不到红彗星？

ps：不用考虑昼夜更替等真实场景。我们假设小红所在的为架空世界，夜晚有无限长。

输入描述:

> 第一行输入两个正整数 n 和 t ，用空格隔开。分别代表彗星的数量、每个彗星的持续时间。
> 第二行输入一个长度为 n 的，只有两种字符'B'和'R'组成的字符串。用来表示每颗彗星的颜色。字符'B'代表蓝色，字符'R'代表红色。
> 第三行输入 n 个正整数 a_i ，代表每颗彗星的开始时刻。
> 数据范围：
> $1 \leq n, t, a_i \leq 100000$

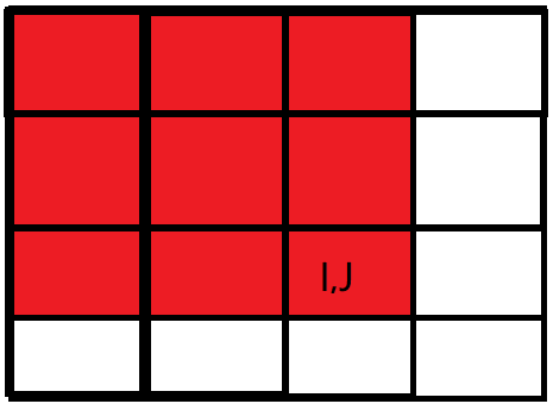
输出描述:

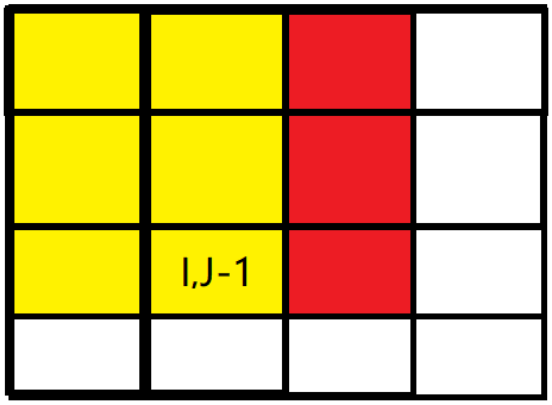
> 能看到蓝彗星且看不到红彗星的总秒数。

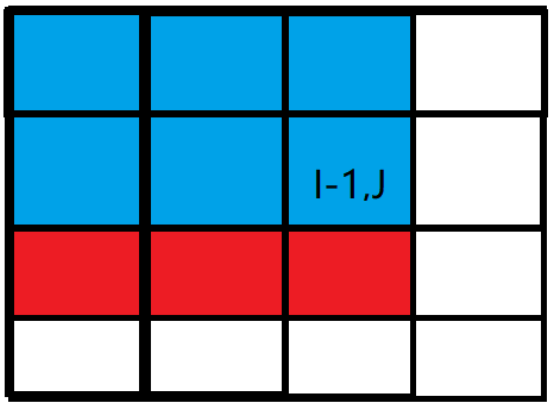
```cpp
#include<bits/stdc++.h>
using namespace std;
#define N 100010
#define mn 200010
typedef long long ll;
ll n,t,a,ans,b[mn],r[mn];
char s[N];
int main()
{
    cin>>n>>t;
    scanf("%s",s+1);
    for(int i=1;i<=n;i++)
    {
        scanf("%lld",&a);
        if(s[i]=='B')
        {
            b[a]++;
            b[a+t]--;
        }
        else
        {
            r[a]++;
            r[a+t]--;
        }
    }
    for(int i=1;i<=mn;i++)
    {
        b[i]+=b[i-1];
        r[i]+=r[i-1];
        if(b[i]>0&&r[i]==0) ans++;
    }
    cout<<ans;
    return 0;
}
```

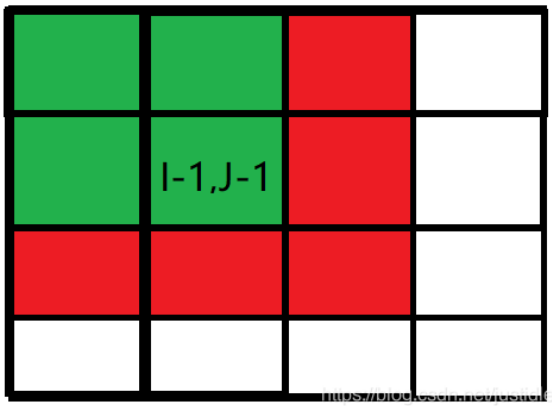
二维差分

- $p = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1]$

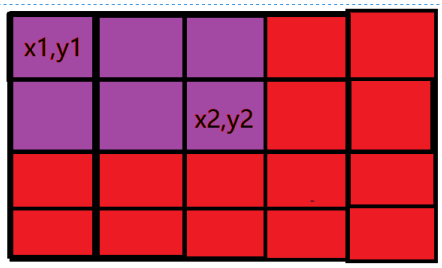


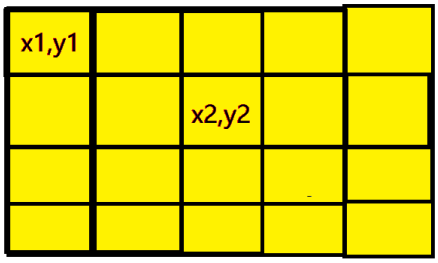


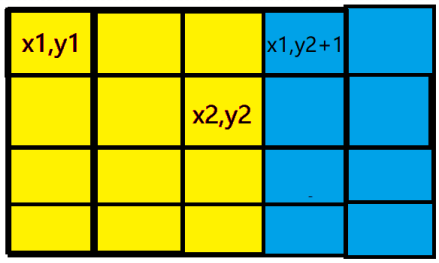


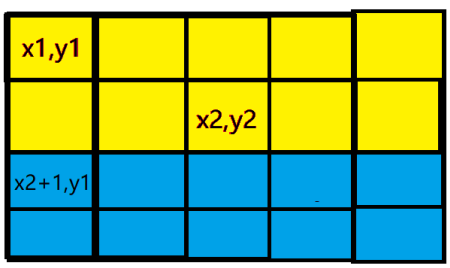


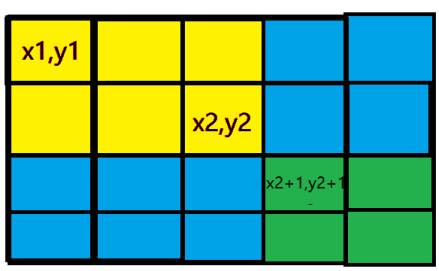
- 在$[x1, y1]$ 到 $[x2, y2] + c$
- $p[x1][y1] += c;$
- $p[x1][y2 + 1] -= c;$
- $p[x2 + 1][y1] -= c;$
- $p[x2 + 1][y2 + 1] += c;$











三维差分

PS：与二维差分思路差不多

将 $[x1,y1,z1]$ 到 $[x2,y2,z2]$ 上的数字 $+c$:

```
b[x1][y1][z1]+=c , b[x2+1][y1][z1] -=c , b[x1][y2+1][z1] -= c , b[x1][y1][z2+1]-=c , b[x2+1][y2+1][z1] +=c , b[x2+1][y1][z2+1] +=c , b[x1][y2+1][z2+1] +=c , b[x2+1][y2+1][z2+1] -=c
```

求$a[i][j][k]$的值：其实就是求**b**数组的三维前缀和

提升练习

完成小组第三次课题单