前缀和、差分、离散化

dbywsc

2025/7

目录

引入

P8218 【深进 1. 例 1】求区间和
对于求 $[a_l, a_r]$ 的和，如果单纯的通过遍历的方式累加区间和时间复杂度为 $O(nm)$，无法通过本题。

定义

对于这种计算区间和的问题，我们一般使用前缀和来解决。
前缀和即对于数组 $\{a_n\}$，额外维护一个前缀和数组 $\{s_n\}$，使得

$$s_i = \sum_{j=1}^{i} a_j$$

容易想到，此时 $[a_l, a_r]$ 的和可以表示为 $s_r - s_{l-1}$。
因此，我们只需要在输入 $\{a_n\}$ 之后计算一次前缀和，就可以在接下来的 m 次讯问中 $O(1)$ 地查询结果，对于这种操作，我们统称为 **预处理**。

实现

前缀和数组的维护本质上是一个递推，即 $s_i = s_{i-1} + a_i$ 。

```
int main(void) {
    std::cin >> n >> m;
    for(int i = 1; i <= n; i++) {
        std::cin >> a[i];
        s[i] = s[i - 1] + a[i];
    }
    while(m--) {
        int l, r;
        std::cin >> l >> r;
        std::cout << s[r] - s[l - 1] << std::endl;
    }
    return 0;
}
```

引入

P1719 最大加权矩形

本质上，本题需要求出任意子矩阵的和的最大值。由于需要枚举起点和终点，再算上计算矩阵和需要的时间，总时间复杂度达到了 $O(n^6)$ 。

对于本题这种求“矩阵和”的问题，实际上是将一维的求“区间和”扩充到了二维，因此，我们也可以将一维前缀和扩充到二维前缀和。

思路

因此，我们可以维护一个二维数组 $\{s_{n,n}\}$，使得

$$s_{i,j} = \sum_{x=1}^{i} \sum_{y=1}^{j} a_{x,y}$$

因为涉及二维，因此 $\{s_n\}$ 的维护方式和子矩阵和的计算方式并不容易想到，需要画图。在 PPT 中我们先给出公式：

$$s_{i,j} = s_{i-1,j} + s_{i,j-1} - s_{i-1,j-1} + a_{i,j}$$

$$\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} a_{i,j} = s_{x_2,y_2} - s_{x_1-1,y_2} - s_{x_2,y_1-1} + s_{x_1-1,y_1-1}$$

实现

```
int main(void) {
    std::cin >> n;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            std::cin >> a[i][j];
            s[i][j] = s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1] + a[i][j];
        }
    }
    for(int x1 = 1; x1 <= n; x1++)
        for(int y1 = 1; y1 <= n; y1++)
            for(int x2 = x1; x2 <= n; x2++)
                for(int y2 = y1; y2 <= n; y2++)
                    ans = std::max(ans, s[x2][y2] - s[x1 - 1][y2] -
                        s[x2][y1 - 1] + s[x1 - 1][y1 - 1]);
    std::cout << ans << std::endl;
    return 0;
}
```

由于计算矩阵和是 $O(1)$ 的，因此我们将总时间复杂度优化到了 $O(n^4)$ ，可以通过本题。

引入

P2367 语文成绩

本质上来说，本题需要多次对区间内的每个数同时加上或者减去一个数，直接遍历的话显然无法通过。

对于“区间内加上一个数”这种问题，我们一般使用差分来解决。

差分即对于数组 $\{a_n\}$，额外维护一个差分数组 $\{b_n\}$，使得 $b_i = a_i - a_{i-1}$。

容易发现，差分本质上是前缀和的 逆运算 。

思路

因为差分和前缀和是互逆运算，因此我们可以反过来，对 $\{b_n\}$ 求一次前缀和，使得 $a_i = \sum_{j=1}^{i} b_j$ 。

这时我们发现，如果令 $b_l += x$ ，此时从求一遍前缀和，从 a_l 开始的每一项由于都让 b_l 参与了运算，所以它们的值都加上了 x ，为了保证 a_r 之后都每一项不受影响，因此我们从 b_{r+1} 开始，每一项减去 x ，后面的值就恢复原状了。

所以，使用差分维护区间加减的方式即为，先对 $b_l += x$，再对 $b_{r+1} -= x$，之后求一遍前缀和就可以了。

在本题中，我们只需要得到经过所有修改后的 $\{a_n\}$ ，因此，我们可以先求一次差分数组，再对 $\{b_n\}$ 进行 m 次操作，最后求一次前缀和并输出就可以了。

由于我们只是分别进行了长度为 n 和 m 的循环，因此总时间复杂度为 $O(n+m)$ 。

实现

```cpp
int main(void) {
    std::cin >> n >> m;
    for(int i = 1; i <= n; i++) {
        std::cin >> a[i];
        b[i] = b[i - 1] + a[i];
    }
    for(int i = 1; i <= m; i++) {
        int l, r, x; std::cin >> l >> r >> x;
        b[l] += x;
        b[r + 1] -= x;
    }

    for(int i = 1; i <= n; i++) {
        a[i] = a[i - 1] + b[i];
        minn = std::min(minn, a[i]);
    }
    std::cout << minn << std::endl;
    return 0;
}
```

引入

P3397 地毯

这道题的形式化题意即给定一个 $n \times n$ 的，初始值全为 0 的二维矩阵 $\{a_{n,n}\}$，每次任选一个子矩阵，为子矩阵内的每一个点增加 1 ，最后输出修改后的矩阵。

由于涉及到二维的矩阵修改，因此我们可以像前缀和那样，维护一个二维的差分数组。

思路

差分是前缀和的逆运算，这句话在二维的前缀和、差分中仍然适用，因此，关于维护二维差分数组的操作一般是由前缀和公式移项得来的。
由于推导过程很简单，因此我们直接放出最后的公式：

$$b_{i,j} = a_{i,j} - a_{i,j-1} - a_{i-1,j} + a_{i-1,j-1}$$

事实上，由于本题一开始的数组是全零的，所以不需要维护差分数组，让 $\{a_{n,n}\}$ 和 $\{b_{n,n}\}$ 一开始全零即可。
修改操作由于和前缀和互逆，因此二者也比较类似：

$$b_{x_1,y_1} += x,\ b_{x_1,y_2+1} -= x$$

$$b_{x_2+1,y_1} -= x,\ b_{x_2+1,y_2+1} += x$$

所有的修改操作结束后，反过来对 $\{b_{n,n}\}$ 求一次前缀和就能得到 $\{a_{n,n}\}$。

实现

```
int main(void) {
    std::cin >> n >> m;
    for(int i = 1; i <= m; i++) {
        int x1, x2, y1, y2; std::cin >> x1 >> y1 >> x2 >> y2;
        b[x1][y1] += 1;
        b[x1][y2 + 1] -= 1;
        b[x2 + 1][y1] -= 1;
        b[x2 + 1][y2 + 1] += 1;
    }
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            a[i][j] = a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1] + b[i][j];
            std::cout << a[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

类似得，由于 m 次操作和计算 $n \times n$ 的矩阵是分开的，因此总时间复杂度为 $O(n^2 + m)$ 。

引入

P1496 火烧赤壁

可以发现，这道题中 a, b 的取值非常大，并且还有负数。如果我们进行将数组作为数轴，当前位置起火就将其在数组中对应的位置标记为一的操作，会发现数组是存不下这么多范围的。

注意到虽然数字的范围非常大，但是个数却很少 $(n \leq 4 \times 10^4)$，因此我们可以把出现过的输入收集起来，整理后为它们建立一个映射关系，将出现过的 4×10^4 个数字映射到 $1 - n$ 中，之后我们就可以直接操作映射后的数字了，最后计算结果时再反过来使用原本映射的值进行计算就可以了。

上面的操作就叫做 **离散化**

实现 – I

离散化的过程，总结起来就是：排序、去重、映射、查找映射值原本的值。在本题中，为了让大家深刻的理解离散化的过程，我们使用手写的方式进行。

首先，为了为每个不同的数组建立一一对应的映射关系，我们要避免对重复的数字进行映射，因此我们要将重复的数字进行去重，一种比较容易理解的方式就是先对原数组排序，然后进行遍历，如果当前数字和上一个数字重复了就舍去它。

```
for(int i = 1; i <= n; i++) {
    std::cin >> a[i] >> b[i];
    d[++dtop] = a[i];    //使用 d 数组存所有出现的数
    d[++dtop] = b[i];
}
sort(d + 1, d + 1 + dtop);
for(int i = 1; i <= dtop; i++)
    if(d[i] != d[i - 1] || i == 1) c[++ctop] = d[i];
//使用 c 数组存储去重后的数组
```

实现 – II

之后我们做一个 $1-n$ 的映射，也就是说，做一个 $1-n$ 的遍历，为了找到当前位置原本映射的值，我们可以使用二分操作分别找到去重后的 c 数组中第一个等于 a_i, b_i 的元素的位置，之后以这个位置建立起一个新的坐标轴 f，再 f 中对这一段区间进行标记。上述的二分操作我们可以使用 *STL* 中的 *lower_bound()* 实现。

```
for(int i = 1; i <= n; i++) {
    int x = std::lower_bound(c + 1, c + ctop + 1, a[i]) - c;
    int y = std::lower_boudn(c + 1, c + ctop + 1, b[i]) - c;
    for(int j = x; j < y; j++) f[j] = 1;
}
```

实现 – III

最后，按照题意，做一遍对数轴 f 的遍历，如果当前 f 被标记了，我们就要把长度加上原本映射的值。

```
for(int i = 1; i < ctop; i++) {
    if(f[i]) ans += c[i + 1] - c[i];
}
std::cout << ans << std::endl;
```

其他写法

事实上，这种离散化写法比较麻烦，在了解离散化的基本原理后，我们可以尝试使用别的更方便的写法，下面提供一种离散化模版：

```
void Init_hash(void) {
    int tmp[N];
    for(int i = 1; i <= n; i++) tmp[i] = a[i];
    sort(tmp + 1, tmp + 1 + n);
    int *ed = unique(tmp + 1, tmp + 1 + n);
    for(int i = 1; i <= n; i++) a[i] = lower_bound(tmp + 1, ed, a[i]) - tmp;
}
```

我们今天学习到的内容属于基本的优化时间复杂度的方式，在算法竞赛中往往它们会作为一道题目的一部分。因此，今天的部分习题在考察了刚刚所学的情况下还和之前我们学习的内容结合了起来，有些或许会有些难度，希望大家能够凭借自己的力量完成它们！

P2004 领地选择

P1083 [NOIP 2012 提高组] 借教室

P1314 [NOIP 2011 提高组] 聪明的质监员

AT_abc405_c [ABC405C] Sum of Product

CF2113C Smilo and Minecraft