CSP-S 提高组

常见优化技巧

2_前缀和与差分

前缀和与差分概述

基本概念

前缀和：数组前 i 项的和，用于快速计算区间和

差分：数组相邻元素的差值，用于快速进行区间修改

关系：前缀和与差分是互逆运算

应用场景：

- 快速查询区间和
- 快速进行区间加减操作
- 二维矩阵操作优化

一维前缀和

基本定义

对于数组 $a[1 \ldots n]$，定义前缀和数组 s：

$$s[i] = \sum_{j=1}^{i} a[j] = a[1] + a[2] + \cdots + a[i]$$

区间和计算：区间 $[l, r]$ 的和为：

$$\sum_{i=l}^{r} a[i] = s[r] - s[l-1]$$

一维前缀和实现

```cpp
const int N = 100010;

int a[N];     // 原数组
int s[N];     // 前缀和数组

// 预处理前缀和
void initPrefix(int n) {
    s[0] = 0;
    for (int i = 1; i <= n; i++)
        s[i] = s[i - 1] + a[i];
}

// 查询区间 [l, r] 的和
int querySum(int l, int r) {
    return s[r] - s[l - 1];
}
```

```cpp
// 使用示例
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    initPrefix(n);

    while (m--) {
        int l, r;
        cin >> l >> r;
        cout << querySum(l, r) << endl;
    }
}
```

一维前缀和应用示例

问题：给定长度为 n 的数组，进行 m 次区间和查询

暴力解法：每次查询 $O(n)$，总复杂度 $O(mn)$

前缀和解法：预处理 $O(n)$，查询 $O(1)$，总复杂度 $O(n+m)$

```cpp
// 计算所有长度为 k 的连续子数组的和
void slidingWindowSum(int a[], int n, int k) {
    int s[N] = {0};

    // 计算前缀和
    for (int i = 1; i <= n; i++) {
        s[i] = s[i - 1] + a[i];
    }

    // 计算所有长度为 k 的窗口和
    for (int i = k; i <= n; i++) {
        int window_sum = s[i] - s[i - k];
        cout << window_sum << " ";
    }
}
```

二维前缀和

基本定义

对于二维数组 $a[1 \ldots n][1 \ldots m]$，定义前缀和数组 s：

$$s[i][j] = \sum_{x=1}^{i} \sum_{y=1}^{j} a[x][y]$$

子矩阵和计算：以 (x_1, y_1) 为左上角，(x_2, y_2) 为右下角的子矩阵和为：

$$\text{sum} = s[x_2][y_2] - s[x_1 - 1][y_2] - s[x_2][y_1 - 1] + s[x_1 - 1][y_1 - 1]$$

二维前缀和实现

```
const int N = 1010;

int a[N][N];     // 原矩阵
int s[N][N];     // 前缀和矩阵

// 预处理二维前缀和
void init2DPrefix(int n, int m) {
    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] - s[i-1][j-1] + a[i][j];
        }
    }
}

// 查询子矩阵和 [(x1,y1) 到 (x2,y2)]
int queryMatrixSum(int x1, int y1, int x2, int y2) {
    return s[x2][y2] - s[x1-1][y2] - s[x2][y1-1] + s[x1-1][y1-1];
}
```

```cpp
// 使用示例
int main() {
    int n, m, q;
    cin >> n >> m >> q;

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

    init2DPrefix(n, m);

    while (q--) {
        int x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        cout << queryMatrixSum(x1, y1, x2, y2) << endl;
    }

    return 0;
}
```

二维前缀和推导过程

前缀和计算：

```
s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j]
```

子矩阵和计算：

```
sum = s[x2][y2] - s[x1-1][y2] - s[x2][y1-1] + s[x1-1][y1-1]
```

记忆技巧：

- 预处理：**上加左减左上加当前**
- 查询：**大减小左加上左减左上**

一维差分

基本概念

对于原数组 $a[1 \dots n]$，定义差分数组 d：

$$d[i] = \begin{cases} a[1] & i = 1 \\ a[i] - a[i-1] & i > 1 \end{cases}$$

性质：

- 对差分数组求前缀和得到原数组
- 区间 $[l, r]$ 加 c：$d[l] += c, d[r+1] -= c$

一维差分实现

```
const int N = 100010;
int a[N];    // 原数组
int d[N];    // 差分数组

// 构建差分数组
void buildDiff(int n) {
    d[1] = a[1];
    for (int i = 2; i <= n; i++) {
        d[i] = a[i] - a[i - 1];
    }
}

// 区间 [l, r] 加 c
void rangeAdd(int l, int r, int c) {
    d[l] += c;
    if (r + 1 <= n) {
        d[r + 1] -= c;
    }
}

// 从差分数组恢复原数组
void restoreArray(int n) {
    a[1] = d[1];
    for (int i = 2; i <= n; i++) {
        a[i] = a[i - 1] + d[i];
    }
}
```

```cpp
// 使用示例
int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    buildDiff(n);

    while (m--) {
        int l, r, c;
        cin >> l >> r >> c;
        rangeAdd(l, r, c);
    }

    restoreArray(n);

    // 输出修改后的数组
    for (int i = 1; i <= n; i++) {
        cout << a[i] << " ";
    }

    return 0;
}
```

一维差分应用示例

问题：进行 m 次区间加操作，最后查询整个数组

暴力解法：每次操作 $O(n)$，总复杂度 $O(mn)$

差分解法：每次操作 $O(1)$，恢复 $O(n)$，总复杂度 $O(n+m)$

```cpp
// 直接构建差分数组（不需要原数组）
void buildDiffDirect(int d[], int n) {
    int last = 0;
    for (int i = 1; i <= n; i++) {
        int x;
        cin >> x;
        d[i] = x - last;
        last = x;
    }
}

// 批量区间操作
void batchOperations(int d[], int n, int m) {
    for (int i = 0; i < m; i++) {
        int l, r, c;
        cin >> l >> r >> c;
        d[l] += c;
        if (r + 1 <= n) d[r + 1] -= c;
    }
}
```

二维差分

基本概念

对于二维数组 $a[1\dots n][1\dots m]$，定义差分数组 d：

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

子矩阵加操作：以 (x_1, y_1) 为左上角，(x_2, y_2) 为右下角的子矩阵加 c：

```
d[x1][y1] += c
d[x2+1][y1] -= c
d[x1][y2+1] -= c
d[x2+1][y2+1] += c
```

二维差分实现

```
const int N = 1010;

int a[N][N];    // 原矩阵
int d[N][N];    // 差分矩阵

// 构建二维差分数组
void build2DDiff(int n, int m) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            d[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1];
}

// 子矩阵加操作 [(x1,y1) 到 (x2,y2)] 加 c
void matrixRangeAdd(int x1, int y1, int x2, int y2, int c) {
    d[x1][y1] += c;
    if (x2 + 1 <= n) d[x2 + 1][y1] -= c;
    if (y2 + 1 <= m) d[x1][y2 + 1] -= c;
    if (x2 + 1 <= n && y2 + 1 <= m) d[x2 + 1][y2 + 1] += c;
}

// 从差分矩阵恢复原矩阵
void restore2DArray(int n, int m) {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            a[i][j] = d[i][j] + a[i-1][j] + a[i][j-1] - a[i-1][j-1];
        }
}
```

二维差分简化实现

```cpp
// 直接操作差分矩阵（推荐）
void matrixRangeAddSimple(int d[][N], int x1, int y1, int x2, int y2, int c) {
    d[x1][y1] += c;
    d[x2 + 1][y1] -= c;
    d[x1][y2 + 1] -= c;
    d[x2 + 1][y2 + 1] += c;
}

// 一次性构建和恢复
void process2DOperations(int n, int m, int q) {
    int d[N][N] = {0};

    // 输入原矩阵并构建差分
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int x;
            cin >> x;
            d[i][j] += x;
            d[i + 1][j] -= x;
            d[i][j + 1] -= x;
            d[i + 1][j + 1] += x;
        }
    }

    // 执行操作
    while (q--) {
        int x1, y1, x2, y2, c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        matrixRangeAddSimple(d, x1, y1, x2, y2, c);
    }

    // 计算前缀和得到结果
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            d[i][j] += d[i - 1][j] + d[i][j - 1] - d[i - 1][j - 1];
            cout << d[i][j] << " ";
        }
        cout << endl;
    }
}
```

综合应用实例

问题：矩阵区间修改与查询

要求：

- 支持子矩阵加减操作
- 支持子矩阵和查询
- 高效处理大规模数据

```
const int N = 1010;

int a[N][N];    // 原矩阵
int s[N][N];    // 前缀和矩阵（用于查询）
int d[N][N];    // 差分矩阵（用于修改）

// 初始化
void init(int n, int m) {
    // 构建前缀和
    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] - s[i-1][j-1] + a[i][j];
        }
    }

    // 构建差分
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            d[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1];
        }
    }
}
```

```cpp
// 子矩阵修改
void modify(int x1, int y1, int x2, int y2, int c) {
    d[x1][y1] += c;
    d[x2+1][y1] -= c;
    d[x1][y2+1] -= c;
    d[x2+1][y2+1] += c;
}

// 子矩阵查询
int query(int x1, int y1, int x2, int y2) {
    return s[x2][y2] - s[x1-1][y2] - s[x2][y1-1] + s[x1-1][y1-1];
}

// 更新前缀和（修改后需要调用）
void updatePrefix(int n, int m) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            a[i][j] = d[i][j] + a[i-1][j] + a[i][j-1] - a[i-1][j-1];
            s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j];
        }
    }
}
```

算法复杂度总结

操作类型	暴力复杂度	优化后复杂度	优化倍数
一维区间和查询	$O(n)$	$O(1)$	n 倍
一维区间修改	$O(n)$	$O(1)$	n 倍
二维子矩阵和查询	$O(n^2)$	$O(1)$	n^2 倍
二维子矩阵修改	$O(n^2)$	$O(1)$	n^2 倍

使用技巧与注意事项

1. **数组下标**：通常从 1 开始，避免边界判断
2. **空间分配**：数组大小开 $N + 10$ 防止越界
3. **初始化**：确保前缀和数组 $s[0] = 0$
4. **差分恢复**：修改后需要恢复原数组才能正确查询
5. **数据类型**：根据数据范围选择 `int` 或 `long long`

```cpp
// 安全的前缀和实现
const int N = 100010;
long long s[N];  // 使用 long long 防止溢出

void safeInit(int n) {
    s[0] = 0;
    for (int i = 1; i <= n; i++) {
        s[i] = s[i - 1] + a[i];
    }
}
```

复习要点

1. 理解前缀和与差分的互逆关系
2. 掌握二维前缀和的容斥原理
3. 熟练运用差分进行区间修改
4. 注意边界条件的处理
5. 根据问题选择合适的优化方法

掌握前缀和与差分，轻松应对区间操作问题！