CSP-S 提高组

常见优化技巧

单调队列

单调队列概述

基本概念

单调队列：队列内元素保持单调性的数据结构

单调性：

- 单调递增队列：队首到队尾元素递增
- 单调递减队列：队首到队尾元素递减

核心思想：维护一个可能成为最值的候选集合，及时排除不可能选项

时间复杂度：每个元素入队出队各一次，$O(n)$

单调队列解决的问题

典型应用场景

1. **滑动窗口最值**（Sliding Window Maximum/Minimum）
2. **区间最值查询**
3. **优化动态规划**
4. **维护单调性的相关问题**

原理

基本操作

- **入队操作**：维护队列单调性，删除破坏单调性的元素
- **出队操作**：删除超出窗口范围(长度)的元素
- **查询操作**：队首元素即为当前窗口的最值

关键点：队列中存储的是**下标**，便于判断元素是否在窗口内

单调递减队列模板

求滑动窗口最大值

```
int a[N];         // 原数组
int q[N];         // 单调队列（存储下标）
int hh = 0, tt = -1;  // 队首、队尾指针

// 求每个长度为k的窗口的最大值
void slidingWindowMax(int n, int k) {
    hh = 0, tt = -1;
    for (int i = 0; i < n; i++) {
        // 删除超出窗口范围的元素
        if (hh <= tt && i - k + 1 > q[hh])
            hh++;
        // 维护队列单调递减性
        while (hh <= tt && a[q[tt]] <= a[i])
            tt--;
        // 当前元素入队
        q[++tt] = i;
        // 输出窗口最大值（窗口形成后）
        if (i >= k - 1)
            cout << a[q[hh]] << " ";
    }
}
```

单调递增队列模板

求滑动窗口最小值

```
void slidingWindowMin(int n, int k) {
    hh = 0, tt = -1;
    for (int i = 0; i < n; i++) {
        // 删除超出窗口范围的元素
        if (hh <= tt && i - k + 1 > q[hh])
            hh++;
        // 维护队列单调递增性
        while (hh <= tt && a[q[tt]] >= a[i])
            tt--;
        // 当前元素入队
        q[++tt] = i;
        // 输出窗口最小值
        if (i >= k - 1)
            cout << a[q[hh]] << " ";
    }
}
```

单调队列推导过程

问题分析

给定数组 $a[0 \ldots n-1]$ 和窗口大小 k，求每个长度为 k 的窗口的最大值

暴力解法：对每个窗口遍历求最大值，时间复杂度 $O(nk)$

优化思路：

- 维护一个候选集合，包含可能成为窗口最大值的元素
- 及时排除不可能成为最大值的元素
- 利用队列的先进先出特性维护窗口范围

单调性维护推导

设当前遍历到位置 i，值为 $a[i]$

关键观察：

- 如果 $a[i] \geq a[j]$ 且 $i > j$，那么 $a[j]$ 永远不可能成为后面窗口的最大值
- 因为 $a[j]$ 会比 $a[i]$ 先离开窗口，且值更小

维护策略：

- 在 $a[i]$ 入队前，删除队尾所有 $\leq a[i]$ 的元素
- 保证队列单调递减，队首即为当前窗口最大值

P1886 滑动窗口 /【模板】单调队列

题目描述

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个窗口从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

数据范围：

- $1 \leq n \leq 10^6$
- $1 \leq k \leq n$
- $-10^9 \leq a[i] \leq 10^9$

解题思路

问题分析:

- 需要维护一个固定大小的滑动窗口
- 对每个窗口求最大值和最小值
- 数据规模要求 $O(n)$ 算法

单调队列应用:

- 最小值：使用单调递增队列
- 最大值：使用单调递减队列

时间复杂度：$O(n)$

参考代码

```
const int N = 1e6 + 10;
int a[N], q[N], ansMin[N], ansMax[N];
int n, k;

// 滑动窗口求最小
void func(int res[])
{
    int hh = 0, tt = -1;
    for (int i = 1; i <= n; i++)
    {
        int l = i - k + 1;
        while (hh <= tt && l > q[hh])
            q[hh] = 0, hh++;
        while (hh <= tt && a[i] < a[q[tt]])
            tt--;
        q[++tt] = i;
        if (i >= k)
            res[l] = q[hh];
    }
}
```

```cpp
int main()
{
    cin >> n >> k;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    func(ansMin);
    for (int i = 1; i <= n - k + 1; i++)
        printf("%d ", a[ansMin[i]]);
    puts("");

    // 巧妙技巧（减少代码量）：取相反数
    for (int i = 1; i <= n; i++)
        a[i] = -a[i];
    func(ansMax);
    for (int i = 1; i <= n - k + 1; i++)
        printf("%d ", -a[ansMax[i]]); // 输出时，再取相反数
    puts("");

    return 0;
}
```

P2032 扫描

题意

有一个 $1 \times n$ 的矩阵，包含 n 个整数。给定一个长度为 k 的木板，初始时覆盖第 $1 \sim k$ 个数，每次将木板向右移动一个单位，直到右端与第 n 个数重合。在每次移动前，输出被覆盖数字中的最大值。

数据范围：$1 \leq k \leq n \leq 2 \times 10^6$，矩阵中的元素大小不超过 10^4 且均为正整数。

分析

本题要求滑动窗口中的最大值，如果直接遍历每个窗口求最大值，时间复杂度为 $O(n \times k)$，在 n 和 k 较大时会超时。因此需要一种高效的方法。

单调队列 可以在 $O(n)$ 时间内解决这个问题。维护一个单调递减队列，队列中存储元素的索引，且对应的值单调递减。这样队列头部始终是当前窗口的最大值。

具体如下：

1. 初始化队列为空。
2. 遍历前 $k-1$ 个元素，对于每个元素，从队列尾部开始移除所有值小于等于当前元素的索引（因为这些索引对应的值不可能成为后续窗口的最大值），然后将当前索引加入队列尾部。

3. 从第 k 个元素开始，处理每个元素：
    i. 同样从队列尾部移除所有值小于等于当前元素的索引。
    ii. 将当前索引加入队列尾部。
    iii. 检查队列头部索引是否超出当前窗口范围（即索引小于等于 $i-k$），如果超出则从头部移除。
    iv. 此时队列头部索引对应的值即为当前窗口的最大值，输出该值。

每个元素最多入队和出队一次，因此时间复杂度为 $O(n)$。

参考代码

```
const int N = 2e6 + 10; // 定义数组大小
int n, k, v[N], q[N];                // 存储原数组，数组模拟单调队列，存储索引
int head = 0, tail = -1; // 队列头尾指针

void solve() {
    // 处理前k-1个元素，构建初始单调队列
    for (int i = 1; i < k; i++) {
        // 从队尾移除所有值小于等于当前元素的索引
        while (head <= tail && v[q[tail]] <= v[i]) tail--;
        q[++tail] = i; // 当前索引入队
    }
    // 从第k个元素开始，处理每个窗口
    for (int i = k; i <= n; i++) {
        // 从队尾移除所有值小于等于当前元素的索引
        while (head <= tail && v[q[tail]] <= v[i]) tail--;
        q[++tail] = i; // 当前索引入队
        // 如果队首索引不在当前窗口内，则从队首移除
        while (q[head] <= i - k) head++;
        // 输出当前窗口最大值
        cout << v[q[head]] << endl;
    }
}

int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++) cin >> v[i];
    solve();
    return 0;
}
```

P2698 [USACO12MAR] Flowerpot S

题意

有 N 滴水滴，每滴水滴的初始坐标为 (x, y)，以每秒 1 个单位长度的速度下落。需要在 x 轴上放置一个花盆，使得花盆接到第 1 滴水与最后 1 滴水之间的时间差至少为 D。求满足条件的最小花盆宽度 W。

数据范围：$1 \leq N \leq 10^5$，$1 \leq D \leq 10^6$，$0 \leq x, y \leq 10^6$。

分析

水滴下落的时间就是其 y 坐标值，因为水滴以每秒 1 个单位长度的速度下落。问题转化为：在 x 轴上找到一个区间 $[L, R]$，使得区间内水滴的 y 坐标最大值与最小值之差至少为 D，并且 $R - L$ 尽可能小。

思路：

1. 将所有水滴按 x 坐标排序，这样我们可以使用滑动窗口的方法来处理。
2. 维护两个单调队列：
    - 一个单调递减队列（$qMax$）用于获取窗口内的最大 y 值
    - 一个单调递增队列（$qMin$）用于获取窗口内的最小 y 值

3. 使用双指针（L 和 i）维护滑动窗口：
    - 右指针 i 从 1 到 n 遍历
    - 对于每个 i，将当前水滴加入两个单调队列
    - 当窗口满足条件（最大 y - 最小 $y \geq D$）时，移动左指针 L 来缩小窗口，并更新答案

关键观察：

- 水滴按 x 排序后，时间差条件转化为 y 坐标的极差条件
- 使用两个单调队列可以在 $O(1)$ 时间内获取当前窗口的最大值和最小值
- 滑动窗口确保我们找到的是满足条件的最小宽度

时间复杂度：每个元素最多入队和出队一次，因此为 $O(n)$。

参考代码

```cpp
const int N = 1e5 + 10, INF = 1e6 + 10;

struct Node {
    int x, y;
    // 按照x坐标排序，方便滑动窗口处理
    bool operator<(const Node &b) const {
        return x < b.x;
    }
} d[N];

int n, D, ans;
int qMax[N], qMin[N]; // 单调队列：qMax维护最大值，qMin维护最小值
// ...

int main() {
    cin >> n >> D;
    for (int i = 1; i <= n; i++) cin >> d[i].x >> d[i].y;

    sort(d + 1, d + 1 + n); // 按x坐标排序

    solve();

    cout << (ans < INF ? ans : -1) << endl;
    return 0;
}
```

```
void solve() {
    ans = INF;
    int h1 = 0, h2 = 0, t1 = -1, t2 = -1; // 两个队列的头尾指针
    int L = 1; // 滑动窗口左边界

    for (int i = 1; i <= n; i++) {
        // 维护单调递减队列（队首为最大值）
        while (h1 <= t1 && d[i].y > d[qMax[t1]].y) t1--;
        qMax[++t1] = i;

        // 维护单调递增队列（队首为最小值）
        while (h2 <= t2 && d[i].y < d[qMin[t2]].y) t2--;
        qMin[++t2] = i;

        // 当窗口满足条件（时间差>=D）时，尝试缩小窗口
        while (L <= i && d[qMax[h1]].y - d[qMin[h2]].y >= D) {
            ans = min(ans, d[i].x - d[L].x); // 更新最小宽度
            L++; // 移动左边界

            // 更新队列，移除超出窗口范围的元素
            while (h1 <= t1 && qMax[h1] < L) h1++;
            while (h2 <= t2 && qMin[h2] < L) h2++;
        }
    }
}
```

P2216 [HAOI2007] 理想的正方形

题意

有一个 $a \times b$ 的整数矩阵，需要找出一个 $n \times n$ 的正方形区域，使得该区域内最大值和最小值的差最小。

数据范围：$2 \leq a, b \leq 1000$，$n \leq a$，$n \leq b$，$n \leq 100$，矩阵中的数不超过 10^9。

分析

1. **问题转化**：需要在 $a \times b$ 的矩阵中，对所有可能的 $n \times n$ 正方形，计算其最大值与最小值的差，并找出最小值。
2. **直接枚举的复杂度**：若直接枚举每个正方形并计算最值，时间复杂度为 $O(a \cdot b \cdot n^2)$，在极限数据下会超时。

3. **单调队列优化**：
   - **行处理**：对每一行，使用单调队列分别计算该行中每个长度为 n 的滑动窗口的最大值（`rowMax`）和最小值（`rowMin`）。
     时间复杂度：$O(a \cdot b)$。
   - **列处理**：对 `rowMax` 和 `rowMin` 数组的每一列，使用单调队列计算每个长度为 n 的滑动窗口的最大值（`colMax`）和最小值（`colMin`），此时 `colMax[i][j]` 和 `colMin[i][j]` 分别表示以 (i, j) 为左上角的 $n \times n$ 正方形的最大值和最小值。
     时间复杂度：$O(a \cdot b)$。
4. **总体复杂度**：两次单调队列处理的总时间复杂度为 $O(a \cdot b)$，能够高效处理最大数据范围。

参考代码

```
const int N = 1005, INF = 0x3f3f3f3f;
int a, b, n;
int g[N][N];            // 原始矩阵
int rowMax[N][N];       // 行滑动窗口最大值
int rowMin[N][N];       // 行滑动窗口最小值
int colMax[N][N];       // 列滑动窗口最大值
int colMin[N][N];       // 列滑动窗口最小值
int q[N];               // 单调队列数组

int main() {
    cin >> a >> b >> n;
    for (int i = 1; i <= a; i++)
        for (int j = 1; j <= b; j++)
            cin >> g[i][j];
    // ...
```

先对每一行进行滑动窗口处理（自左向右）

```
// 对每一行进行滑动窗口处理
for (int i = 1; i <= a; i++) {
    int hh = 0, tt = -1; // 队列头尾指针
    // 计算行最大值
    for (int j = 1; j <= b; j++) {
        // 维护窗口范围：移除超出窗口左侧的元素
        while (hh <= tt && q[hh] <= j - n) hh++;
        // 维护单调递减队列：队尾元素小于当前元素时出队
        while (hh <= tt && g[i][q[tt]] <= g[i][j]) tt--;
        q[++tt] = j; // 当前元素入队
        // 记录窗口最大值（从第n个位置开始记录）
        if (j >= n) rowMax[i][j - n + 1] = g[i][q[hh]];
    }

    hh = 0, tt = -1; // 重置队列
    // 计算行最小值
    for (int j = 1; j <= b; j++) {
        while (hh <= tt && q[hh] <= j - n) hh++;
        // 维护单调递增队列：队尾元素大于当前元素时出队
        while (hh <= tt && g[i][q[tt]] >= g[i][j]) tt--;
        q[++tt] = j;
        if (j >= n) rowMin[i][j - n + 1] = g[i][q[hh]];
    }
}
```

再对每一列进行滑动窗口处理（自上向下）

```
    // 对列进行滑动窗口处理
    for (int j = 1; j <= b - n + 1; j++) {
        int hh = 0, tt = -1;
        // 计算列最大值 (基于rowMax)
        for (int i = 1; i <= a; i++) {
            while (hh <= tt && q[hh] <= i - n) hh++;
            while (hh <= tt && rowMax[q[tt]][j] <= rowMax[i][j]) tt--;
            q[++tt] = i;
            if (i >= n) colMax[i - n + 1][j] = rowMax[q[hh]][j];
        }

        hh = 0, tt = -1;
        // 计算列最小值 (基于rowMin)
        for (int i = 1; i <= a; i++) {
            while (hh <= tt && q[hh] <= i - n) hh++;
            while (hh <= tt && rowMin[q[tt]][j] >= rowMin[i][j]) tt--;
            q[++tt] = i;
            if (i >= n) colMin[i - n + 1][j] = rowMin[q[hh]][j];
        }
    }
```

```
    // 遍历所有n×n正方形，寻找最小差值
    int ans = INF;
    for (int i = 1; i <= a - n + 1; i++)
        for (int j = 1; j <= b - n + 1; j++)
            ans = min(ans, colMax[i][j] - colMin[i][j]);

    cout << ans << endl;
    return 0;
}
```

单调队列复杂度分析

时间复杂度

- **每个元素**：入队一次，出队最多一次
- **总操作次数**：$2n$ 次
- **时间复杂度**：$O(n)$

空间复杂度

- 队列最多存储 k 个元素
- **空间复杂度**：$O(k)$

相比暴力解法的 $O(nk)$，优化效果明显

使用技巧与注意事项

技巧总结

1. **存储下标**：队列中存储 `下标` 而非 `值`，便于判断窗口范围
2. **哨兵技巧**：可在数组前后添加哨兵元素简化边界判断
3. **双端操作**：单调队列需要同时在队首和队尾进行操作
4. **单调性选择**：根据问题需求选择递增或递减队列

常见错误

1. **窗口范围判断错误**：注意下标计算
2. **队列初始化错误**：空队列或初始元素处理
3. **单调性维护错误**：比较符号方向搞反
4. **边界条件遗漏**：第一个窗口的特殊处理

算法复杂度对比

方法	时间复杂度	空间复杂度	适用场景
暴力求解	$O(nk)$	$O(1)$	k 很小
单调队列	$O(n)$	$O(k)$	通用
线段树	$O(n \log n)$	$O(n)$	动态查询
ST表	$O(n \log n)$	$O(n \log n)$	静态查询

复习要点

1. **理解单调性原理**：为什么可以删除某些元素
2. **掌握模板代码**：熟练写出单调队列的三种操作
3. **注意下标处理**：队列存储下标的重要性
4. **分析问题特征**：识别适合单调队列优化的问题

单调队列核心：维护一个**可能成为答案**的候选集合，及时排除**不可能**的选项。

练习题推荐

1. **P1886 滑动窗口**：基础单调队列应用
2. **P2032 扫描**：二维单调队列
3. **P1714 切蛋糕**：单调队列优化前缀和
4. **P2216 [HAOI2007]理想的正方形**：二维滑动窗口最值
5. **P2698 [USACO12MAR] Flowerpot S**：双指针+单调队列
6. **P2569 [SCOI2010]股票交易**：复杂单调队列优化DP

掌握单调队列，高效解决滑动窗口问题！