CSP-S 提高组

常见优化技巧

离散化

离散化概述

基本概念

离散化：将无限空间中的有限个体映射到有限空间中的方法

核心思想：只关注数据的相对大小关系，不关注具体数值

适用场景：

- 数据范围很大但数据量较小
- 需要建立数组但值域过大
- 坐标压缩、数据归一化

为什么需要离散化

问题示例：有 10^5 个整数，数值范围 $[-10^9, 10^9]$，需要建立索引

方法	空间复杂度	可行性
直接数组	$O(2 \times 10^9)$	❌ 不可行
离散化	$O(10^5)$	✅ 可行

优势：将稀疏的大范围数据压缩为稠密的小范围数据

离散化基本步骤

三步法

1. **收集**：收集所有需要离散化的值
2. **排序去重**：对收集的值排序并去重
3. **映射**：建立原值到新下标的映射关系

```cpp
const int N = 100010;

int n;                          // 数据个数
int a[N];                       // 原始数据
int temp[N];                    // 临时数组用于离散化
int discrete[N];                // 离散化后的值
int cnt;                        // 离散化后不同值的个数

// 离散化函数
void discretization() {
    // 1. 复制数据到临时数组
    for (int i = 0; i < n; i++) {
        temp[i] = a[i];
    }

    // 2. 排序
    sort(temp, temp + n);

    // 3. 去重并计数
    cnt = unique(temp, temp + n) - temp;

    // 4. 建立映射（可选，根据需求）
    for (int i = 0; i < n; i++) {
        discrete[i] = lower_bound(temp, temp + cnt, a[i]) - temp;
    }
}
```

离散化映射函数

```
// 查询原值 x 的离散化下标
int get_discrete_index(int x) {
    return lower_bound(temp, temp + cnt, x) - temp;
}

// 查询离散化下标 i 对应的原值
int get_original_value(int i) {
    return temp[i];
}

// 使用示例
int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    discretization();

    // 使用离散化后的下标
    for (int i = 0; i < n; i++) {
        cout << "原值:" << a[i] << " -> 离散下标:" << discrete[i] << endl;
    }

    return 0;
}
```

一维离散化应用

区间和问题

问题：在数轴上进行区间加操作和区间和查询，坐标范围很大

```cpp
const int N = 100010;

struct Operation {
    int type;  // 0: 加操作, 1: 查询
    int x, y, c;
} op[N];

int discrete[2 * N];       // 离散化数组
int cnt;                   // 离散化后坐标个数

// 离散化所有出现的坐标
void collect_coordinates(int m) {
    int idx = 0;
    for (int i = 0; i < m; i++) {
        discrete[idx++] = op[i].x;
        discrete[idx++] = op[i].y;
    }

    sort(discrete, discrete + idx);
    cnt = unique(discrete, discrete + idx) - discrete;
}

// 获取离散化下标
int get_idx(int x) {
    return lower_bound(discrete, discrete + cnt, x) - discrete;
}
```

区间和问题实现

```
int sum[2 * N];  // 离散化后的前缀和数组
void solve_interval_sum(int m) {
    collect_coordinates(m);

    // 初始化前缀和数组
    for (int i = 0; i < m; i++) {
        if (op[i].type == 0) {  // 加操作
            int l = get_idx(op[i].x);
            int r = get_idx(op[i].y);
            // 在离散化后的数组上进行操作
            sum[l] += op[i].c;
            if (r + 1 < cnt) {
                sum[r + 1] -= op[i].c;
            }
        }
    }

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

    // 处理查询
    for (int i = 0; i < m; i++) {
        if (op[i].type == 1) {  // 查询操作
            int l = get_idx(op[i].x);
            int r = get_idx(op[i].y);
            int result = sum[r] - (l > 0 ? sum[l - 1] : 0);
            cout << result << endl;
        }
    }
}
```

二维离散化

矩阵坐标压缩

问题：在二维平面上进行操作，坐标范围很大但操作点很少

```cpp
const int N = 10010;

struct Point {
    int x, y, val;
} points[N];

int discrete_x[N], discrete_y[N];
int cnt_x, cnt_y;

// 收集所有 x 和 y 坐标
void collect_2d_coordinates(int n) {
    for (int i = 0; i < n; i++) {
        discrete_x[i] = points[i].x;
        discrete_y[i] = points[i].y;
    }

    // 分别对 x 和 y 坐标离散化
    sort(discrete_x, discrete_x + n);
    cnt_x = unique(discrete_x, discrete_x + n) - discrete_x;

    sort(discrete_y, discrete_y + n);
    cnt_y = unique(discrete_y, discrete_y + n) - discrete_y;
}

// 获取二维离散化坐标
void get_2d_index(int x, int y, int &idx_x, int &idx_y) {
    idx_x = lower_bound(discrete_x, discrete_x + cnt_x, x) - discrete_x;
    idx_y = lower_bound(discrete_y, discrete_y + cnt_y, y) - discrete_y;
}
```

二维离散化应用

```
int matrix[N][N];  // 离散化后的矩阵

void process_2d_operations(int n) {
    collect_2d_coordinates(n);

    // 初始化离散化后的矩阵
    for (int i = 0; i < n; i++) {
        int idx_x, idx_y;
        get_2d_index(points[i].x, points[i].y, idx_x, idx_y);
        matrix[idx_x][idx_y] += points[i].val;
    }

    // 计算二维前缀和
    for (int i = 0; i < cnt_x; i++) {
        for (int j = 0; j < cnt_y; j++) {
            if (i > 0) matrix[i][j] += matrix[i - 1][j];
            if (j > 0) matrix[i][j] += matrix[i][j - 1];
            if (i > 0 && j > 0) matrix[i][j] -= matrix[i - 1][j - 1];
        }
    }

    // 现在可以在离散化后的矩阵上进行查询
    // 查询 [x1, y1] 到 [x2, y2] 的子矩阵和
}
```

离散化注意事项

边界处理

问题：离散化可能丢失区间信息

解决方案：将区间端点都加入离散化，必要时插入中间点

```cpp
// 处理区间 [l, r] 的离散化
void process_interval(int l, int r) {
    discrete[cnt++] = l;
    discrete[cnt++] = r;
    // 如果需要保留区间信息，可以插入 r+1
    discrete[cnt++] = r + 1;
}

// 处理开区间和闭区间
void process_different_intervals() {
    // 对于 [l, r] 闭区间，离散化 l 和 r
    // 对于 [l, r) 左闭右开，离散化 l 和 r-1
    // 根据具体问题调整
}
```

重复元素处理

```
// 手动实现去重（理解原理）
int manual_unique(int arr[], int n) {
    if (n == 0) return 0;

    int idx = 1;
    for (int i = 1; i < n; i++) {
        if (arr[i] != arr[i - 1]) {
            arr[idx++] = arr[i];
        }
    }
    return idx;
}

// 使用 STL 去重（推荐）
int stl_unique(int arr[], int n) {
    sort(arr, arr + n);
    return unique(arr, arr + n) - arr;
}
```

离散化性能分析

时间复杂度

操作	时间复杂度	说明
排序	$O(n \log n)$	主要开销
去重	$O(n)$	线性扫描
二分查找	$O(\log n)$	每次映射
总体	$O(n \log n)$	可接受

空间复杂度

- 原始数据：$O(n)$
- 离散化数组：$O(n)$
- 映射结构：$O(n)$

实际应用案例

案例：矩形面积并

问题：计算多个矩形的并集面积

```cpp
struct Rectangle {
    int x1, y1, x2, y2;  // 左下角和右上角坐标
} rect[N];

// 离散化所有 x 和 y 坐标
void discretize_rectangles(int n) {
    // 收集所有 x 坐标
    for (int i = 0; i < n; i++) {
        discrete_x[i * 2] = rect[i].x1;
        discrete_x[i * 2 + 1] = rect[i].x2;
        discrete_y[i * 2] = rect[i].y1;
        discrete_y[i * 2 + 1] = rect[i].y2;
    }

    // 排序去重
    sort(discrete_x, discrete_x + 2 * n);
    cnt_x = unique(discrete_x, discrete_x + 2 * n) - discrete_x;

    sort(discrete_y, discrete_y + 2 * n);
    cnt_y = unique(discrete_y, discrete_y + 2 * n) - discrete_y;
}
```