奇思妙学 信息学竞赛

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

离散后相同元素相同位置

```cpp
const int N = 1e5 + 10;
int arr[N], tmp[N],n;
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> arr[i];
    // arr[i] 为初始数组,下标范围为 [1, n]
    for (int i = 1; i <= n; ++i) // step 1
        tmp[i] = arr[i];
    sort(tmp + 1, tmp + n + 1);                          // step 2
    int len = unique(tmp + 1, tmp + n + 1) - (tmp + 1); // step 3
    for (int i = 1; i <= n; ++i)                         // step 4
        arr[i] = lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;
    for (int i = 1; i <= n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

离散后相同元素不同位置

```cpp
const int MAXN = 1e5 + 10;
int n;
struct node {
    int idx, val; // 位置、原值
    bool operator<(const node &a) const {
        if (val == a.val) // 值相同，顺序小的优先
            return idx < a.idx;
        return val < a.val;
    }
} tmp[MAXN], arr[MAXN]; // 副本、原数组
int main() {
    cin >> n;
    // 1.初始化
    for (int i = 1; i <= n; i++)
        cin >> arr[i].val, arr[i].idx = i;
    for (int i = 1; i <= n; i++) // 2.创建副本
        tmp[i] = arr[i];
    sort(tmp + 1, tmp + 1 + n); // 3.排序
    for (int i = 1; i <= n; i++)
        arr[tmp[i].idx].val = i; // 4.将原数组的值，重新映射为元素间相对大小
    for (int i = 1; i <= n; i++)
        cout << arr[i].val << " ";
    return 0;
}
```

重复元素处理

```cpp
// 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：$O(n)$

题目训练

P1496 火烧赤壁

题意

给定 n 个区间 $[a_i, b_i)$（左闭右开），求这些区间覆盖的总长度。

数据范围：$1 \le n \le 2 \times 10^4$，$-2^{31} \le a < b < 2^{31}$，答案小于 2^{31}。

> 样例解释：区间 $[-1, 1)$、$[5, 11)$、$[2, 9)$ 覆盖的总长度为 $2 + 6 + 7 = 15$，但注意区间重叠部分不能重复计算。

分析

核心思路：使用离散化处理大范围坐标，通过**标记法**统计覆盖区间。

1. **离散化处理**：
    - 将所有区间的端点坐标收集起来
    - 排序并去重，建立坐标到索引的映射
    - 将原始的大范围坐标映射到小范围的索引上

2. **区间标记**：
    - 对于每个区间，在离散化后的坐标索引上进行标记
    - 注意区间是左闭右开，所以右端点不包含在内
3. **长度计算**：
    - 遍历离散化后的坐标段
    - 如果该段被标记，则将其实际长度加入答案

时间复杂度：$O(n \log n)$，主要来自排序操作。

参考代码

```
#define ll long long
const int N = 4e4 + 10;
unordered_map<int, int> mp;
int n, pos[N], cnt, _cnt, a[N], b[N];
bool vis[N];
ll ans, _pos[N];
int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i] >> b[i], pos[++cnt] = a[i], pos[++cnt] = b[i];
    // 排序+ 去重
    sort(pos + 1, pos + 1 + cnt);
    for (int i = 1; i <= cnt; i++)
        if (pos[i] != pos[i - 1] || i == 1)
            mp[pos[i]] = ++_cnt, _pos[_cnt] = pos[i];
    for (int i = 1; i <= n; i++) {
        int j = mp[a[i]], k = mp[b[i]];
        for (int j = mp[a[i]]; j < k; j++)
            vis[j] = true; // 左闭右开 < mp[b[i]]
    }
    for (int i = 1; i < _cnt; i++)
        if (vis[i])
            ans += _pos[i + 1] - _pos[i];
    cout << ans << endl;
    return 0;
}
```

小结：

1. **离散化处理：**
   - `pos` 数组存储所有区间的端点
   - 排序去重后得到 `_pos` 数组
   - `mp` 建立原始坐标到离散化索引的映射
2. **区间标记：**
   - 对于每个区间 $[a, b)$，在离散化后的索引范围 $[a[i], b[i])$ 进行标记
   - 注意右端点是开区间，所以标记到 `b[i]-1`

3. **长度计算**：
    - 遍历离散化后的相邻坐标点
    - 如果该段被标记，则计算实际坐标差并累加
4. **复杂度分析**：
    - 排序：$O(n \log n)$
    - 标记：最坏情况 $O(n^2)$，但实际数据中表现良好
    - 对于更大数据可以使用差分数组优化标记过程

这种方法通过离散化将大范围坐标映射到小范围索引，有效解决数组 `RE` 问题。

P1884 [USACO12FEB] Overplanting S

题意

给定 N 个矩形在笛卡尔坐标系中的位置（左上角和右下角坐标），求这些矩形覆盖的总面积。重复覆盖的区域只计算一次。

数据范围：$1 \leq N \leq 1000$，$-10^8 \leq x_1, y_1, x_2, y_2 \leq 10^8$。

分析

由于坐标范围很大但矩形数量较少，需要使用**离散化**和**扫描线**技术：

- **离散化**：将坐标值映射到连续的整数索引，减少处理范围
- **差分数组**：在离散化后的网格上标记矩形覆盖情况
- **面积计算**：统计所有被覆盖的小矩形面积之和

具体步骤：

1. 收集所有坐标值并排序去重
2. 建立原坐标与离散化索引的映射
3. 使用差分数组标记矩形覆盖区域
4. 前缀和还原覆盖情况
5. 计算每个离散化网格的实际面积并累加

时间复杂度：$O(N^2)$，主要来自离散化后的网格处理。

参考代码

```
#define ll long long

const int N = 4005;
int ctop, top, n;
ll a[N][5], b[N], c[N], f[N][N], ans;
std::map<int, int> Map;

int main()
{
    std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
    std::cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= 4; j++)
            std::cin >> a[i][j], b[++top] = a[i][j];
    std::sort(b + 1, b + 1 + top); // 排序
    // ...
```

```
    // 0 位置初始化无穷小
    b[0] = -1e8 - 1;
    for (int i = 1; i <= top; i++) // 去重
        if (b[i] != b[i - 1])
            c[++ctop] = b[i], Map[b[i]] = ctop; // c「下标」映射「值」，map 「值」映射「下标」
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= 4; j++)
            a[i][j] = Map[a[i][j]]; // 给原坐标，映射到离散后的坐标位置，通过 c[] 来取
    for (int i = 1; i <= n; i++)
        for (int j = a[i][1]; j < a[i][3]; j++) // 枚举行
            f[j][a[i][2]]++, f[j][a[i][4]]--;   // 每行进行差分
    // 前缀和，还原序列
    for (int i = 1; i < ctop; i++)
        for (int j = 1; j < ctop; j++)
            f[i][j] += f[i][j - 1];
    for (int i = 1; i < ctop; i++)
        for (int j = 1; j < ctop; j++)
            if (f[i][j])
                ans += (c[i + 1] - c[i]) * (c[j + 1] - c[j]);
    std::cout << ans << "\n";
    return 0;
}
```

启发与总结

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

离散后相同元素相同位置

```
const int N = 1e5 + 10;
int arr[N], tmp[N],n;
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> arr[i];
    // arr[i] 为初始数组,下标范围为 [1, n]
    for (int i = 1; i <= n; ++i) // step 1
        tmp[i] = arr[i];
    sort(tmp + 1, tmp + n + 1);                          // step 2
    int len = unique(tmp + 1, tmp + n + 1) - (tmp + 1); // step 3
    for (int i = 1; i <= n; ++i)                         // step 4
        arr[i] = lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;
    for (int i = 1; i <= n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

离散后相同元素不同位置

```cpp
const int MAXN = 1e5 + 10;
int n;
struct node {
    int idx, val; // 位置、原值
    bool operator<(const node &a) const {
        if (val == a.val) // 值相同，顺序小的优先
            return idx < a.idx;
        return val < a.val;
    }
} tmp[MAXN], arr[MAXN]; // 副本、原数组
int main() {
    cin >> n;
    // 1.初始化
    for (int i = 1; i <= n; i++)
        cin >> arr[i].val, arr[i].idx = i;
    for (int i = 1; i <= n; i++) // 2.创建副本
        tmp[i] = arr[i];
    sort(tmp + 1, tmp + 1 + n); // 3.排序
    for (int i = 1; i <= n; i++)
        arr[tmp[i].idx].val = i; // 4.将原数组的值，重新映射为元素间相对大小
    for (int i = 1; i <= n; i++)
        cout << arr[i].val << " ";
    return 0;
}
```