華中科技大學

课 程 实 验 报 告

课程名称：算法设计与分析实践

专业班级：计算机本硕博 2301 班

学　　号：U202315763

姓　　名：王家乐

指导教师：王多强

报告日期：2024 年 12 月 18 日

计算机科学与技术学

目 录

1 题目完成情况

本次实验共做 34 道题，通过 26 道题：

- 分治：P2678、P1228
- 动态规划：P1220、P1854、P4999、P1434、P4017
- 贪心：P1106、P2512、P2127、P1658、P1090
- 搜索：P1443、P1135、P2895、P1825、P1784、P1141、P1019、P3067
- 后缀数组：P3809
- 树状数组：P3374、P3368、P3605
- 线段树：P1816、P3373、P1904
- 差分约束系统：P1250
- 并查集：P1111、P1196、P2024、P1197
- 前缀和：P1314、P3865

完成截图如下：

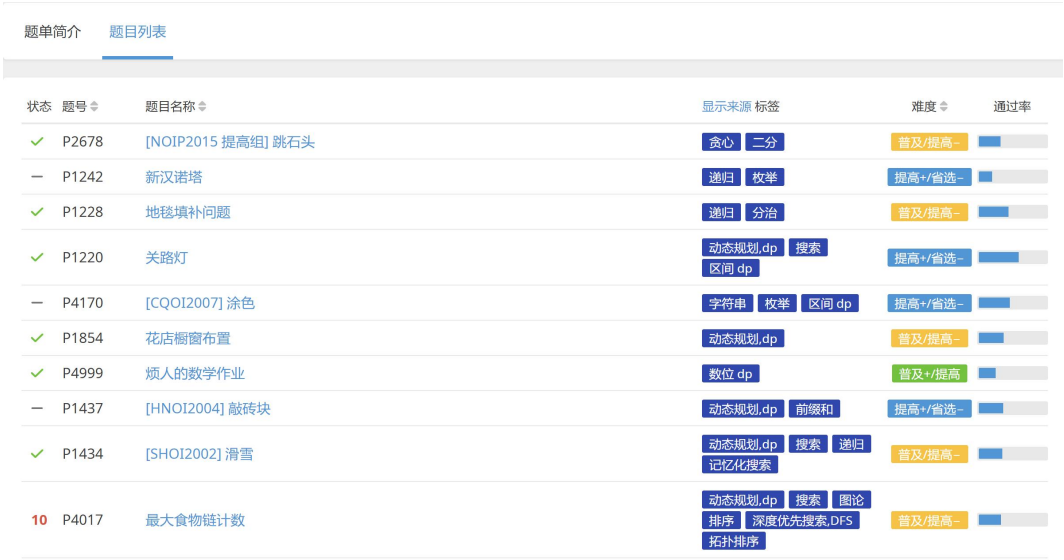

图 1-1 完成截图 1

–	P2018	消息传递	动态规划,dp 贪心 树形 dp	普及+/提高
✓	P1106	删数问题	字符串 贪心	普及/提高-
–	P1080	[NOIP2012 提高组] 国王游戏	贪心 高精度 排序	普及+/提高
0	P2512	[HAOI2008] 糖果传递	贪心	提高+/省选-
✓	P2127	序列排序	模拟 贪心 离散化	提高+/省选-
✓	P1658	购物	贪心	普及/提高-
✓	P1090	[NOIP2004 提高组] 合并果子 / [USACO06NOV] Fence Repair G	贪心 堆 优先队列	普及/提高-
✓	P1443	马的遍历	搜索 广度优先搜索,BFS 队列	普及/提高-
✓	P1135	奇怪的电梯	模拟 广度优先搜索,BFS 深度优先搜索,DFS	普及/提高-
✓	P2895	[USACO08FEB] Meteor Shower S	搜索 广度优先搜索,BFS	普及/提高-
✓	P1825	[USACO11OPEN] Corn Maze S	模拟 搜索 广度优先搜索,BFS 最短路	普及/提高-
–	P1433	吃奶酪	动态规划,dp 状态压缩	普及+/提高
✓	P1784	数独	搜索 Dancing Links	普及/提高-
✓	P1141	01迷宫	搜索 广度优先搜索,BFS 队列	普及/提高-

图 1-2 完成截图 2

51	P1019	[NOIP2000 提高组] 单词接龙	字符串 搜索	普及/提高-
–	P2349	金字塔	图论 最短路	普及+/提高
–	P2324	[SCOI2005] 骑士精神	搜索 启发式搜索 启发式迭代加深搜索,IDA* A*算法	提高+/省选-
–	P5691	[NOI2001] 方程的解数	数学 折半搜索, meet in the middle	普及+/提高
34	P3067	[USACO12OPEN] Balanced Cow Subsets G	搜索 折半搜索, meet in the middle	提高+/省选-
✓	P3809	【模板】后缀排序	字符串 排序 后缀数组,SA	省选/NOI-
–	P2852	[USACO06DEC] Milk Patterns G	二分 哈希, hash 后缀数组,SA	普及+/提高
–	P2870	[USACO07DEC] Best Cow Line G	字符串 贪心 队列	提高+/省选-
✓	P3374	【模板】树状数组 1	树状数组	普及/提高-
✓	P3368	【模板】树状数组 2	树状数组	普及/提高-
10	P3605	[USACO17JAN] Promotion Counting P	线段树 树状数组 离散化 深度优先搜索,DFS	提高+/省选-
✓	P1816	忠诚	线段树 倍增 RMQ st表	普及/提高-

图 1-3 完成截图 3

30	P3373	【模板】线段树 2	线段树	普及+/提高
✓	P1904	天际线	模拟 线段树 离散化	普及–
–	P1993	小 K 的农场	图论 广度优先搜索,BFS 差分约束	普及+/提高
–	P3275	[SCOI2011] 糖果	最短路 差分约束	提高+/省选-
✓	P1250	种树	贪心 排序 差分约束	普及/提高-
✓	P1111	修复公路	二分 并查集 排序	普及/提高-
✓	P1196	[NOI2002] 银河英雄传说	并查集	普及+/提高
✓	P2024	[NOI2001] 食物链	并查集	普及+/提高
20	P1197	[JSOI2008] 星球大战	并查集 连通块	普及+/提高
–	P2756	飞行员配对方案问题	网络流 二分图 最大流	普及+/提高
✓	P1314	[NOIP2011 提高组] 聪明的质监员	数学 二分 前缀和	普及+/提高
–	P1726	上白泽慧音	图论	普及+/提高
–	P3379	【模板】最近公共祖先（LCA）	最近公共祖先,LCA	普及/提高-
70	P3865	【模板】ST 表 && RMQ 问题	st表	普及/提高-

图 1-4 完成截图 4

2 分治：P2678 跳石头

2.1 题目描述

一年一度的“跳石头”比赛又要开始了！

这项比赛将在一条笔直的河道中进行，河道中分布着一些巨大岩石。组委会已经选择好了两块岩石作为比赛起点和终点。在起点和终点之间有 N 块岩石（不含起点和终点的岩石）。在比赛过程中，选手们将从起点出发，每一步跳向相邻的岩石，直至到达终点。

为了提高比赛难度，组委会计划移走一些岩石，使得选手们在比赛过程中的最短跳跃距离尽可能长。由于预算限制，组委会至多从起点和终点之间移走 M 块岩石（不能移走起点和终点的岩石）。

2.2 输入格式

第一行包含三个整数 L，N，M，分别表示起点到终点的距离，起点和终点之间的岩石数，以及组委会至多移走的岩石数。保证 L≥1 且 N≥M≥0。

接下来 N 行，每行一个整数，第 i 行的整数 Di (0<Di<L)， 表示第 i 块岩石与起点的距离。这些岩石按与起点距离从小到大的顺序给出，且不会有两个岩石出现在同一个位置。

2.3 输出格式

一个整数，即最短跳跃距离的最大值。

2.4 算法思想

本题的核心是通过移除岩石来最大化跳跃过程中最短的跳跃距离。我们可以利用二分法来不断调整最短跳跃距离的值。具体地，首先将岩石按位置排序，然后通过二分法确定最小跳跃距离的最大值。在每次二分时，利用贪心策略检查当前的跳跃距离是否可行，即判断是否能通过移除至多 M 块岩石来保证所有跳跃的距离大于等于当前最短跳跃距离。检查的过程是遍历岩石位置，若相邻岩石的跳跃距离小于当前最短距离，则需要移除该岩石；若移除的岩石数量超过 M，则该跳跃距离不可行。通过二分查找，最终能够确定出能够最大化最短跳跃距离的值。

2.5 伪代码

```
Input L, N, M  // 输入起点到终点的距离、岩石的数量和最多可以移除的岩石数
```

```
Input rock_positions[1..N]  // 输入岩石的位置
Sort rock_positions in ascending order  // 将岩石位置按从小到大的顺序排序
low = 1  // 最小可能的最短跳跃距离
high = L  // 最大可能的最短跳跃距离

// 使用二分法查找最短跳跃距离的最大值
while low <= high:
    mid = (low + high) / 2  // 计算中间值（候选的最短跳跃距离）
    if canAchieve(mid):
        low = mid + 1  // 如果当前距离可行，尝试更大的跳跃距离
    else:
        high = mid - 1  // 如果当前距离不可行，尝试更小的跳跃距离
Output high  // 输出最大可行的最短跳跃距离

// 检查是否可以保证最短跳跃距离为 dist
function canAchieve(dist):
    cnt = 0  // 记录需要移除的岩石数量
    last_position = 0  // 上一个被保留的岩石位置，从起点（位置 0）开始
    for i = 1 to N:
        // 如果当前岩石和上一个岩石之间的距离小于 dist
        if rock_positions[i] - last_position < dist:
            cnt++  // 需要移除当前岩石
        // 否则，保留当前岩石并更新上一个位置
        else:
            last_position = rock_positions[i]
    if L - last_position < dist:  // 检查最后一个岩石到终点的距离
        cnt++  // 如果这个距离小于 dist，意味着需要移除终点的岩石
    return cnt <= M  // 移除的岩石数量小于等于 M，返回 True，否则返回 False
```

表 2-1 P2678 伪代码

2.6 性能分析

2.6.1 时间复杂度

由于二分查找进行 O(log L) 次，每次调用 canAchieve 的时间复杂度为 O(N)，因此整体的时间复杂度为：O(N· logL)，其中：N 是岩石的数量，L 是起点到终点的距离。

2.6.2 空间复杂度

主要的空间消耗是用于存储岩石位置的数组 rock_positions[1..N]，因此空间

复杂度是 O(N)，其中 N 是岩石的数量。

2.7 优化方案

2.7.1 加速输入输出

使用 ios::sync_with_stdio(false)和 cin.tie(nullptr) 来加速输入输出，适合大数据量时使用。

2.7.2 优化 canAchieve

当 cnt 超过 M 时，已经不需要继续检查后面的岩石。可以在 cnt > M 时提前退出函数，避免不必要的计算。

2.7.3 用位运算代替除法

mid = (low + high)>>1 相当于除以 2，并且在某些情况下可以加速计算。这是因为位运算的执行速度通常比除法操作要快。

2.8 测试结果

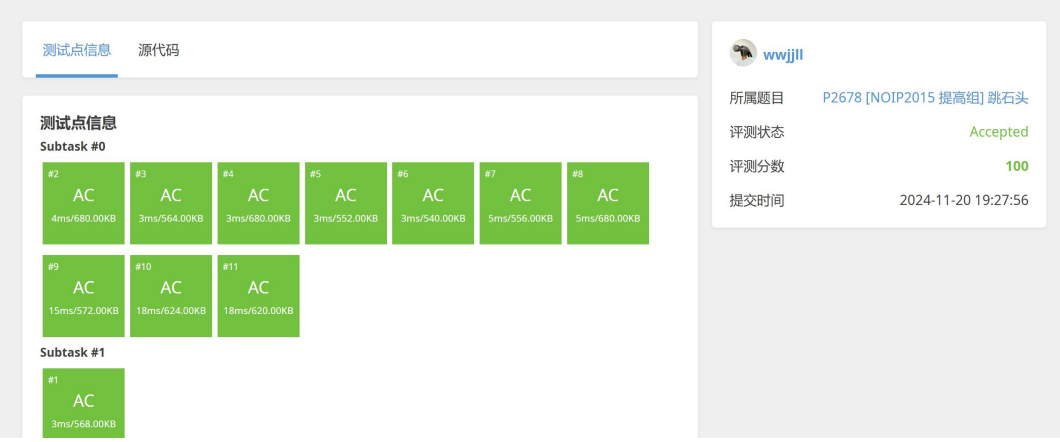

图 2-1 P2678 测试结果

3 贪心：P2127 序列排序

3.1 题目描述

小 C 有一个 N 个数的整数序列，这个序列的中的数两两不同。小 C 每次可以交换序列中的任意两个数，代价为这两个数之和。小 C 希望将整个序列升序排序，问小 C 需要的最小代价是多少？

3.2 输入格式

第一行，一个整数 N。第二行，N 个整数，表示小 C 的序列。

3.3 输出格式

一行，一个整数，表示小 C 需要的最小代价。

3.4 算法思想

该问题本质上是一个“最小代价排序”问题，可以通过考虑元素交换的代价来求解。每次交换的代价是交换的两个数的和，因此需要找到一种方法来最小化交换所需的总代价。

问题中的关键点是，序列可以看作是由多个“置换环”组成。在每个置换环内，元素之间的排序通过交换来完成。对于每个置换环，我们可以有两种方式来计算交换的代价：

1. 环内最小元素法：将环内的最小元素作为中心来交换，代价为环内最小元素乘以环的大小减去 1，再加上环内所有元素的和减去环内最小元素。
2. 全局最小元素法：使用全局最小元素与环内最小元素进行交换，代价为全局最小元素和环内最小元素的和乘以 2，再加上全局最小元素乘以环的大小减去 1，再加上环内所有元素的和减去环内最小元素。

通过比较这两种方式的代价，选择较小的方式来处理每个置换环。最终的总代价就是所有置换环的代价之和。

3.5 伪代码

```
function minSwapCost(N, arr):
    sorted_arr = sorted(arr)  // 排序后的数组
    index_map = {}  // 存储每个数的最终位置
    for i from 1 to N:
        index_map[sorted_arr[i]] = i
```

```
    global_min = sorted_arr[1]  // 全局最小元素
    total_cost = 0  // 初始化总代价
    visited = [false] * (N + 1)  // 记录元素是否已被处理

    for i from 1 to N:
        if visited[i] or arr[i] == sorted_arr[i]:
            continue  // 跳过已排序的元素或已访问的元素

        cycle_size = 0  // 置换环的大小
        cycle_sum = 0  // 置换环的元素之和
        cycle_min = infinity  // 置换环的最小值
        current_index = i  // 当前索引

        // 处理置换环
        while not visited[current_index]:
            visited[current_index] = true
            cycle_size += 1
            cycle_sum += arr[current_index]
            cycle_min = min(cycle_min, arr[current_index])
            current_index = index_map[arr[current_index]]

        // 计算环内最小元素法的代价
        cycle_cost = cycle_min * (cycle_size - 1) + cycle_sum - cycle_min

        // 计算全局最小元素法的代价
        global_cost = (global_min + cycle_min) * 2 + global_min *
(cycle_size - 1) + cycle_sum - cycle_min

        // 选择较小的代价
        total_cost += min(cycle_cost, global_cost)

    return total_cost
```

表 3-1 P2127 伪代码

3.6 样例分析

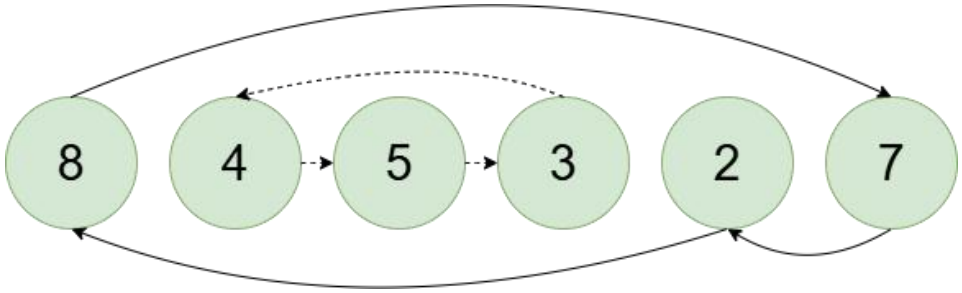

图 3-1 样例：8 4 5 3 2 7

该样例有两个置换环。通过环内最小元素交换时，逆着箭头换一圈即可；通过全局最小元素交换时，先将全局最小元素与环内最小元素交换，再逆着箭头换一圈, 最后再将全局最小元素与环内最小元素交换回来即可。依次处理每个置换换，因此对该样例而言，total_cost=min(19,27)+min(15,23)=34。

3.7 性能分析

3.7.1 时间复杂度

对原始数组进行排序，时间复杂度为 O(N log N)，其中 N 是数组的长度。遍历数组一次，通过记录访问状态来处理每个置换环。每个元素最多被访问一次，因此时间复杂度为 O(N)。主要时间开销来自于排序操作，因此总的时间复杂度为：O(N log N)。

3.7.2 空间复杂度

arr 和 sorted 数组分别存储原始数组和排序后的数组，每个数组大小为 N，因此空间复杂度为 O(N)。index_map 是一个哈希表，用于存储元素及其排序位置，大小也是 N，空间复杂度为 O(N)。visited 数组用于标记元素是否已处理，大小为 N，空间复杂度为 O(N)。所有额外的数据结构（arr, sorted, index_map, visited）的空间复杂度都是 O(N)。因此，空间复杂度为：O(N)。

3.8 优化方案

3.8.1 优化排序和索引映射的存储方式

当前代码通过两个数组（arr 、sorted）来存储原始数据和排序后的数据，这

导致了重复的存储开销。可以考虑直接在原数组上进行排序和标记，从而减少额外空间的使用。index_map 使用了 unordered_map，它是基于哈希表实现的，可能会带来一定的空间和性能开销。如果元素范围比较小，使用数组代替哈希表可能会更高效。

3.8.2 优化环的处理过程

在处理置换环时，当前代码通过 visited 数组来标记是否已经访问过每个元素，基本是合理的，但如果能避免不必要的重复访问可能会提升效率，尤其是在某些情况下有大量已经排序好的元素。

3.9 测试结果

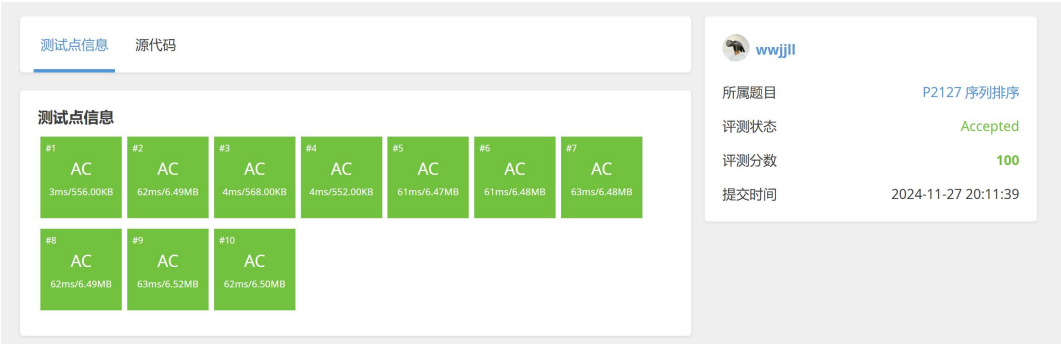

图 3-2 P2127 测试结果

4 搜索：P2895 Meteor Shower S

4.1 题目描述

贝茜听说一场特别的流星雨即将到来：这些流星会撞向地球，并摧毁它们所撞击的任何东西。她为自己的安全感到焦虑，发誓要找到一个安全的地方（一个永远不会被流星摧毁的地方）。

如果将牧场放入一个直角坐标系中，贝茜现在的位置是原点，并且，贝茜不能踏上一块被流星砸过的土地。

根据预报，一共有 M 颗流星 (1≤M≤50000) 会坠落在农场上，其中第 i 颗流星会在时刻 Ti（0≤Ti≤1000）砸在坐标为 (Xi,Yi)(0≤Xi≤300，0≤Yi≤300) 的格子里。流星的力量会将它所在的格子，以及周围 4 个相邻的格子都化为焦土，当然贝茜也无法再在这些格子上行走。

贝茜在时刻 0 开始行动，她只能在会在横纵坐标 X,Y≥0 的区域中，平行于坐标轴行动，每 1 个时刻中，她能移动到相邻的（一般是 4 个）格子中的任意一个，当然目标格子要没有被烧焦才行。如果一个格子在时刻 t 被流星撞击或烧焦，那么贝茜只能在 t 之前的时刻在这个格子里出现。贝茜一开始在 (0,0)。

请你计算一下，贝茜最少需要多少时间才能到达一个安全的格子。如果不可能到达输出 -1。

4.2 输入格式

共 M+1 行，第 1 行输入一个整数 M ，接下来的 M 行每行输入三个整数分别为 Xi,Yi,Ti 。

4.3 输出格式

贝茜到达安全地点所需的最短时间，如果不可能，则为 -1。

4.4 算法思想

该问题的核心思想是通过广度优先搜索（BFS）来找到从原点 (0, 0) 到达一个安全格子的最短时间。首先，我们需要记录每个格子及其相邻格子被流星摧毁的最早时间。对于每颗流星，我们更新它所在格子及周围四个相邻格子的烧焦时间。然后，使用 BFS 从原点 (0, 0) 开始探索，在探索过程中，只有当前时刻 t 小于某格子被流星摧毁的时间时，贝茜才能进入该格子。如果 BFS 遍历过程中找到一个未被烧焦的格子，立即返回到达该格子的最短时间。如果 BFS 完成后依然没有找到安全的格子，则返回 -1，表示无法到达安全地点。

4.5 伪代码

```
Initialize a[310][310] to INT_MAX  // 将 a 初始化为 INT_MAX
Initialize visited[310][310] to false  // 将 visited 初始化为 false

Read m  // 读取障碍点的数量

For each obstacle point (x, y, t):
    Set a[x][y] to min(a[x][y], t)  // 记录障碍点的时间
    For each direction (dx[j], dy[j]):
        Compute the new coordinates (nx, ny) = (x + dx[j], y + dy[j])
        If nx >= 0 and ny >= 0:
            Set a[nx][ny] to min(a[nx][ny], t)  // 更新相邻位置的时间

Initialize queue q, push (0, 0, 0)  // 推入起点 (0, 0) 和时间 0
Set visited[0][0] to true  // 标记起点为已访问

While queue q is not empty:
    Pop the front element p = (current coordinates, current time)
    If a[p.x][p.y] == INT_MAX  // 如果当前位置不可达：
        Set res to p.time  // 更新结果为当前时间
        Exit the loop  // 结束搜索

    For each direction (dx[i], dy[i]):
        Compute the new coordinates (nx, ny) = (p.x + dx[i], p.y + dy[i])
        If nx >= 0 and ny >= 0, not visited, and p.time + 1 < a[nx][ny]:
            Push (nx, ny, p.time + 1) into the queue q  // 入队并更新时间
            Set visited[nx][ny] to true  // 标记该位置为已访问

Output res, if res == INT_MAX, output -1  // 输出 res
```

表 4-1 P2895 伪代码

4.6 性能分析

4.6.1 时间复杂度

每个位置出队和入队的操作是 $O(1)$。最多有 n^2 个位置，所以总的队列操作时间复杂度是 $O(n^2)$。每次从队列中取出一个位置后，检查其四个相邻的方向，每个方向的操作是 $O(1)$。因此总的时间复杂度是 $O(n^2)$。

4.6.2 空间复杂度

主要是由于存储二维数组 a[][] 和 visited[][] 以及队列的空间需求。整体的空间复杂度为 $O(n^2)$。

4.7 优化方案

4.7.1 优先队列（A*或 Dijkstra 算法的变体）

目前的 BFS 使用普通队列，但可以使用优先队列（如 std::priority_queue 或 heapq）来优化访问最短路径的效率。每次从队列中弹出的节点都是当前路径最短的格子，避免了无效路径的扩展。这是基于“贪心”策略的优化，可以让每次访问到的格子都是最接近目标的格子。

4.7.2 提前终止

当 BFS 发现某个格子已经是安全的时，可以立刻终止搜索，而不是继续搜索其他不必要的格子。通过提前退出搜索，我们可以尽早找到答案，减少不必要的计算。

4.7.3 移动判断的优化

在 BFS 中，判断一个格子是否可以访问的条件是 $t + 1 < a[nx][ny]$，这保证了贝茜只能在流星撞击之前到达格子。如果我们将流星的撞击时间存储在一个单独的二维数组中（而不是每次查询时进行比较），可以更快地判断格子的状态。

4.8 测试结果

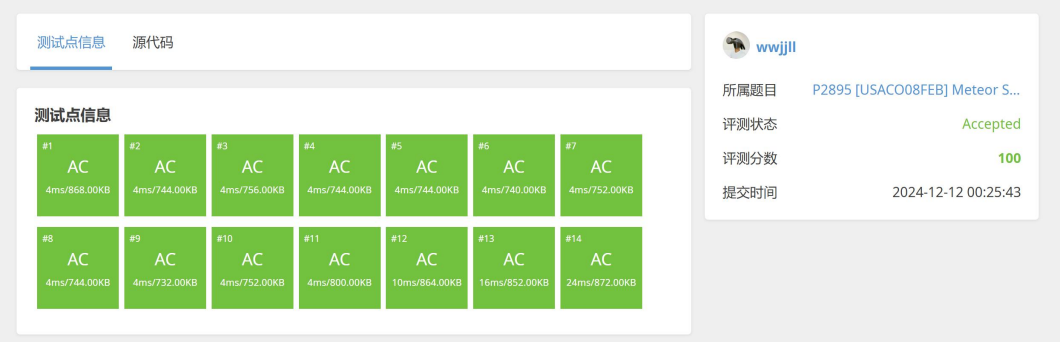

图 4-1 P2895 测试结果

5 并查集：P1196 银河英雄传说

5.1 题目描述

宇宙历 799 年，银河系的两大军事集团在巴米利恩星域爆发战争。泰山压顶集团派宇宙舰队司令莱因哈特率领十万余艘战舰出征，气吞山河集团点名将杨威利组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 1,2,…,30000。之后，他把自己的战舰也依次编号为 1,2,…,30000，让第 i 号战舰处于第 i 列，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 M i j，含义为第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令: C i j。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

5.2 输入格式

第一行有一个整数 T（1≤T≤500000），表示总共有 T 条指令。

以下有 TT 行，每行有一条指令。指令有两种格式：

1.M i j：i 和 j 是两个整数（1≤i,j≤30000），表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令，并且保证第 i 号战舰与第 j 号战舰不在同一列。

2. C i j：i 和 j 是两个整数（1≤i,j≤30000），表示指令涉及的战舰编。该指令是莱因哈特发布的询问指令。

5.3 输出格式

依次对输入的每一条指令进行分析和处理：如果是杨威利发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息。如果是莱因哈特发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i 号战舰与第 j 号战舰之间布置的战舰数目。如果第 i 号战舰与第 j 号战舰当前不在同一列上，则输出 -1。

5.4 算法思想

该问题可以通过带权的并查集（Union-Find）算法来高效地处理战舰的合并和查询操作。初始时，每个战舰都在独立的列中，且每个战舰的父节点是它自身。并查集主要通过两个操作来管理集合：查找（find）和合并（union）。查找操作通过递归查找某个战舰的根节点，同时在查找过程中使用路径压缩技术来优化后续查询，即更新路径上的所有节点的父节点指向根节点，这样可以降低树的高度，提高查询效率。每个战舰到其根节点的距离通过 dist 数组来维护。合并操作则是将两个集合合并成一个集合，具体做法是查找两个战舰所在的集合的根节点，然后通过按秩合并（将较小的树合并到较大的树下）来保持树的平衡，合并时还需要更新合并后的根节点与新子节点之间的距离。在查询操作中，当询问两个战舰是否在同一列时，首先通过查找操作得到两个战舰的根节点，如果它们属于同一个集合，则计算它们之间的战舰数目，即它们的距离差（减去 1）；如果它们不属于同一个集合，则返回 -1。整个算法通过路径压缩和按秩合并，保证了查找和合并操作的时间复杂度接近常数，从而能够在高效处理大量指令时保持良好的性能。

5.5 伪代码

```
function find(x):
    if parent[x] != x:
        original_parent = parent[x]
        parent[x] = find(parent[x])  // 递归查找父节点
        dist[x] += dist[original_parent]  // 更新到根节点的距离
    return parent[x]

function union_sets(x, y):
    rootX = find(x)
    rootY = find(y)
    if rootX != rootY:
        parent[rootX] = rootY  // 将 rootX 合并到 rootY
        dist[rootX] = setSize[rootY]  // 更新距离
        setSize[rootY] += setSize[rootX]  // 更新集合大小
        setSize[rootX] = 0  // rootX 不再是一个独立的集合

initialize:
    for i from 1 to MAX_N:
        parent[i] = i
        setSize[i] = 1
        dist[i] = 0
```

```
for each instruction:
    if instruction is 'M i j':
        union_sets(i, j)  // 合并战舰 i 和战舰 j 所在的队列
    else if instruction is 'C i j':
        if find(i) == find(j):
            print(abs(dist[i] - dist[j]) - 1)  // 输出战舰间的数目
        else:
            print(-1)  // 不在同一列
```

表 5-1 P1196 伪代码

5.6 流程图

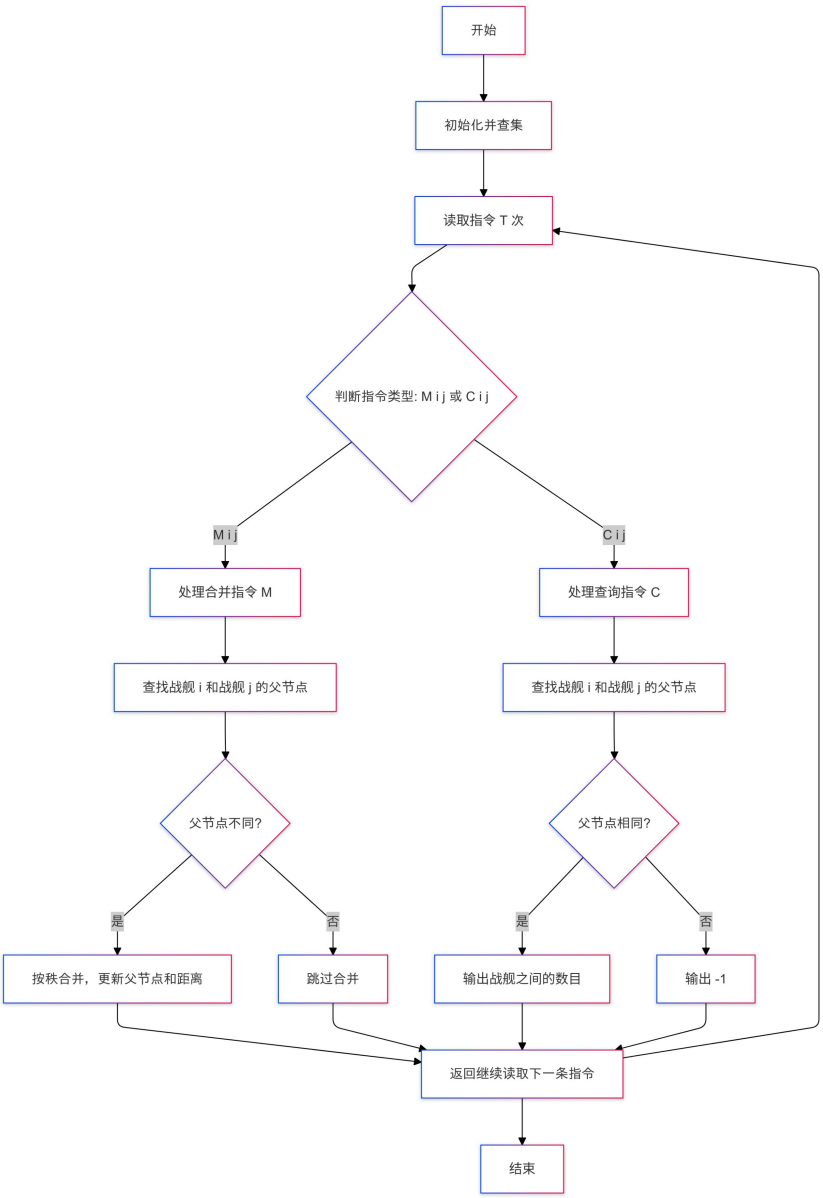

图 5-1 P1196 流程图

5.7 性能分析

5.7.1 时间复杂度

在有路径压缩和按秩合并的情况下，查找操作的时间复杂度接近 常数。理论上，查找操作的时间复杂度为 $O(\alpha(n))$，其中 $\alpha(n)$ 是 阿克曼函数的反函数，它的增长非常慢，对于实际应用来说几乎可以认为是常数时间。合并操作本质上是对两个集合的根进行查找，找到它们的根节点后将其中一个集合合并到另一个集合。由于查找操作的时间复杂度是 $O(\alpha(n))$，合并操作也会进行两次查找，因此其时间复杂度是 $O(\alpha(n))$。查询操作需要先进行两次查找，检查两个战舰是否属于同一集合。如果属于同一集合，则输出它们之间的战舰数目；否则输出 -1。因此，查询操作的时间复杂度是 $O(\alpha(n))$。执行了 T 次操作时，总的时间复杂度为 $O(T * \alpha(n))$。

5.7.2 空间复杂度

空间复杂度是由三个大小为 30000 的数组决定的，因此空间复杂度是 $O(n)$，其中 $n = 30000$。

5.8 优化方案

5.8.1 优化合并操作的策略

目前的算法已经采用了按秩合并，但按秩的标准只是比较树的高度。可以进一步优化合并策略，通过结合按大小合并和按高度合并，使用树的 ″平衡因子″（如集合的元素数量）来决定合并策略。这样可以进一步减少树的高度，使得查找和合并操作更加高效。

5.8.2 引入并行化处理(多线程)

对于极大量的查询和合并操作，可以考虑使用多线程来并行处理查询和合并。虽然并查集的查找和合并操作本身是顺序的（即依赖于父节点的路径），但是通过一些方法，可以将某些部分操作并行化，特别是在查询和合并的数量非常大的情况下。

5.8.3 更精细的路径压缩

虽然当前路径压缩已经是一种较为高效的方式，但我们可以进一步改进路径

压缩策略。例如，在路径压缩过程中，除了更新节点的父节点为根节点，还可以增加一些跳跃式压缩，不仅将当前节点直接指向根节点，还可以通过分析树的结构，适当跳跃更多的节点，从而更有效地减少路径的深度。

5.9 测试结果

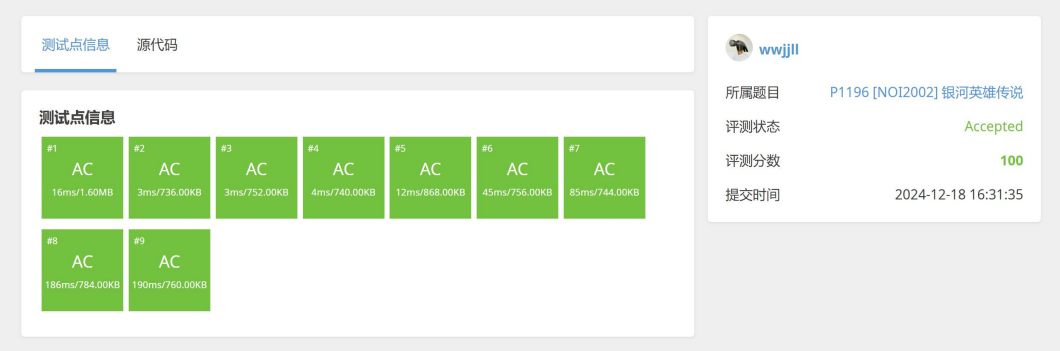

图 5-2 P1196 测试结果

6 总结

6.1 个人收获

在学习和掌握分治法、动态规划、贪心策略、图算法、搜索算法以及其他高级数据结构与算法的过程中，我逐步提升了自己的算法思维和编程能力。分治法的学习让我深刻理解了如何将一个复杂的问题分解为多个小的子问题，通过递归的方式独立解决每个子问题，并将结果合并以得到最终解。这种方法不仅提高了问题解决的效率，也帮助我在处理大规模数据时能够保持较低的时间复杂度。

动态规划带来了全新的思考方式，尤其是在处理最优解问题时。通过记录子问题的解决过程，避免了重复计算，极大地提升了算法效率。我学会了如何通过状态转移方程进行建模，以及如何通过备忘录法（记忆化递归）或迭代法来实现动态规划。这一过程让我更加深刻地理解了递推关系和最优子结构的思想。

贪心策略则让我体会到问题解决中的“局部最优”思路。通过在每一步选择最优解，贪心算法能够快速找到某些问题的解，尤其是在某些问题满足贪心选择性质时，它的效果尤为突出。但我也认识到，贪心算法并非在所有问题中都能保证最优解，因此在面对一些复杂问题时，需要对比贪心与动态规划的异同，选择合适的算法进行求解。

高级数据结构如并查集、树状数组的学习，使我能够高效地处理一些复杂的查询和更新操作。通过路径压缩技术和优化查询，不断提高并查集操作效率，而树状数组在处理区间查询和区间修改等问题时显示了巨大的优势。

这一系列算法和数据结构的学习，不仅提升了我的编程能力，更增强了我在面对复杂问题时的解决方案设计能力。我深刻体会到，算法不仅仅是理论知识的积累，更是解决实际问题的工具，通过不断学习和实践，我能够更加自信地运用不同的算法和数据结构来应对各种挑战。

6.2 体会和建议

这门课程让我对算法的设计与优化有了更深入的理解。课程涵盖各类算法的设计思想与应用，帮助我掌握了如何在面对复杂问题时选择合适的算法，并对算法的时间和空间复杂度进行分析和优化。这不仅提升了我的编程能力，也锻炼了我解决问题的思维方式，特别是在处理实际问题时，能够更加注重算法的效率与可扩展性。通过编程实践，我将理论知识与实际应用相结合，取得了较好的进展。

不过，课程的部分内容相对较为抽象，建议增加更多具体的应用案例，尤其是对常见算法的实际优化方法进行详细讲解。同时，可以针对安排编程练习和小项目，帮助学生更好地掌握算法分析与设计的技巧，并提高解决问题的能力。

附录

P2678 跳石头(源码)

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#define OK 1
#define NO 0

using namespace std;

int L, N, M, d[50001];
bool check(int dist);

int main()
{
    cin >> L >> N >> M;
    for(int i = 1; i <= N; i++)
        cin >> d[i];
    sort(d + 1, d + N + 1);
    int low = 1, high = L, mid;
    while(low <= high)
    {
        mid = (low + high) / 2;
        if(check(mid) == OK)
            low = mid + 1;
        else
            high = mid - 1;
    }
    cout << high << endl;  // 输出结果应该是high
}

bool check(int dist)  // 判断能否最多移走M块石头使得每块石头之间的距离都大于等于
dist
    int cnt = 0, last = 0;
    for(int i = 1; i <= N; i++)
    {
        if(d[i] - last < dist)
            cnt++;
        else
            last = d[i];
    }
    if(L - last < dist) // 判断最后一块石头到终点的距离
        cnt++;
    return cnt <= M ? OK : NO;
}
```

P2127 序列排序(源码)

```cpp
#include <iostream>
#include <algorithm>
#include <unordered_map>
#include <vector>
#include <climits> // 必须包含此头文件才能使用 INT_MAX

using namespace std;

int main()
{
    int N;
    cin >> N;
    vector<long long> arr(N + 1), sorted(N + 1);
    unordered_map<long long, int> index_map;
    vector<bool> visited(N + 1, false);

    for (int i = 1; i <= N; i++)
    {
        cin >> arr[i];
        sorted[i] = arr[i];
    }

    sort(sorted.begin() + 1, sorted.end());

    // 构建索引映射
    for (int i = 1; i <= N; i++)
        index_map[sorted[i]] = i;

    long long global_min = sorted[1]; // 全局最小值
    long long total_cost = 0;

    for (int i = 1; i <= N; i++)
    {
        if (visited[i] || arr[i] == sorted[i])
            // 跳过已访问或已就位的元素
            continue;

        int cycle_size = 0; // 置换环大小
        long long cycle_sum = 0; // 置换环总和
        long long cycle_min = 2147483647; // 置换环最小值
        int current_index = i; // 当前索引

        // 处理一个环
        while (!visited[current_index]) {
            visited[current_index] = true;
            cycle_size++;
            cycle_sum += arr[current_index];
            cycle_min = min(cycle_min, static_cast<long long>(arr[current_index]));
            current_index = index_map[arr[current_index]];
        }

        // 通过环内最小元素计算代价
        long long cycle_cost=cycle_min*(cycle_size-1)+cycle_sum-cycle_min;

        // 通过全局最小元素计算代价
        long long global_cost=(global_min+cycle_min)*2+global_min*(cycle_size-1)+cycle_sum-cycle_min;

        total_cost += min(cycle_cost, global_cost);
    }

    cout << total_cost;
    return 0;
}
```

P2895 Meteor Shower S（源码）

```
#include <iostream>
#include <queue>
#include <climits>
#include <utility>
#include <vector>

using namespace std;

int m;
vector<vector<int>> a(310, vector<int>(310, INT_MAX));
int dx[4] = {0, 0, 1, -1};
int dy[4] = {1, -1, 0, 0};
queue<pair<pair<int, int>, int>> q;
int res = INT_MAX;
vector<vector<bool>> visited(310, vector<bool>(310, false)); // 访问数组

//横纵坐标不能小于0，但可以大于300
int main()
{
    cin >> m;
    for (int i = 0; i < m; i++)
    {
        int x, y, t;
        cin >> x >> y >> t;
        a[x][y] = min(a[x][y], t);
        for (int j = 0; j < 4; j++)
        {
            int nx = x + dx[j];
            int ny = y + dy[j];
            if (nx >= 0 && ny >= 0)
                a[nx][ny] = min(a[nx][ny], t);
        }
    }
    q.push(make_pair(make_pair(0, 0), 0));
    visited[0][0] = true; // 标记起始点为已访问
    while (!q.empty())
    {
        pair<pair<int, int>, int> p = q.front();
        q.pop();
        if (a[p.first.first][p.first.second] == INT_MAX)
        {
            res = p.second;
            break;
        }
        for (int i = 0; i < 4; i++)
        {
            int nx = p.first.first + dx[i];
            int ny = p.first.second + dy[i];
            if (nx >= 0 && ny >= 0 && !visited[nx][ny] && p.second + 1 < a[nx]
[ny])        {
                q.push(make_pair(make_pair(nx, ny), p.second + 1));
                visited[nx][ny] = true; // 标记为已访问
            }
        }
    }
    cout << (res == INT_MAX ? -1 : res);
    return 0;
}
```

P1196 银河英雄传说(源码)

```cpp
#include <iostream>
using namespace std;
#define MAX_N 30005

// parent[i] 表示 i 的父节点
// setSize[i] 表示 i 所在集合的大小
// distance[i] 表示 i 到其父节点的距离
int parent[MAX_N], setSize[MAX_N], dist[MAX_N];

// 查找操作，带路径压缩
int find(int x) {
    if (parent[x] != x) {
        int original_parent = parent[x];
        parent[x] = find(parent[x]);
        dist[x] += dist[original_parent]; // 更新距离
    }
    return parent[x];
}

// 合并操作
void union_sets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        // 按秩合并，保持树的平衡
        parent[rootX] = rootY;
        dist[rootX] = setSize[rootY]; // 更新rootX到rootY的距离
        setSize[rootY] += setSize[rootX]; // 更新rootY的集合大小
        setSize[rootX] = 0;
    }
}

int main() {
    int T;
    cin >> T;
    // 初始化并查集
    for (int i = 1; i <= MAX_N; ++i) {
        parent[i] = i;
        setSize[i] = 1;
        dist[i] = 0;
    }
    // 处理每一条指令
    while (T--) {
        char op;
        int i, j;
        cin >> op >> i >> j;
        if (op == 'M') // 合并指令 M i j
            union_sets(i, j);
        else if (op == 'C') { // 查询指令 C i j
            if (find(i) == find(j)) // 此时更新距离
                // 如果在同一列，输出战舰之间的数目
                cout << abs(dist[i] - dist[j]) - 1 << endl;
            else
                // 如果不在同一列，输出 -1
                cout << -1 << endl;
        }
    }
    return 0;
}
```