華中科技大學

《算法设计与分析实践》

实 验 报 告

成　　绩：
评　　语：
教师签字：
评阅日期：

专业班级：_____

学　　号：_____

姓　　名：_____

指导教师：_____

报告日期：_____

计算机科学与技术学院

2024 年11 月

目 录

1. 完成情况

本次实验中，我共完成并通过了以下 22 道题目，覆盖了各种算法。

序号	题目编号	题目名	算法归类
1	01-P2678	跳石头	二分法
2	04-P1220	关路灯	深度优先搜索
3	08-P1437	敲砖块	动态规划
4	09-P1434	滑雪	深度优先搜索
5	10-P4017	最大食物链	拓扑排序
6	12-P1106	删数游戏	贪心算法
7	16-P1658	购物	贪心算法
8	18-P1433	马的遍历	广度优先搜索
9	20-P2895	Meteor shower	宽度优先搜索
10	21-P1825	玉米田迷宫	广度优先搜索
11	24-P1141	01 迷宫	并查集
12	25-P1019	单词接龙	深度优先搜索
13	30-P3809	后缀数组	后缀数组
14	33-P3374	树状数组 1	树状数组
15	34-P3368	树状数组 2	差分数组
16	37-P3373	线段树模板 2	线段树
17	38-P1904	天际线	线段树
18	39-P1993	小 K 的农场	差分约束
19	41-P1250	种树	贪心算法
20	42-P1111	修建公路	并查集
21	44-P2024	食物链	并查集
22	49-P3379	最近公共祖先	DFS 倍增法

以下为我的洛谷个人主页截图

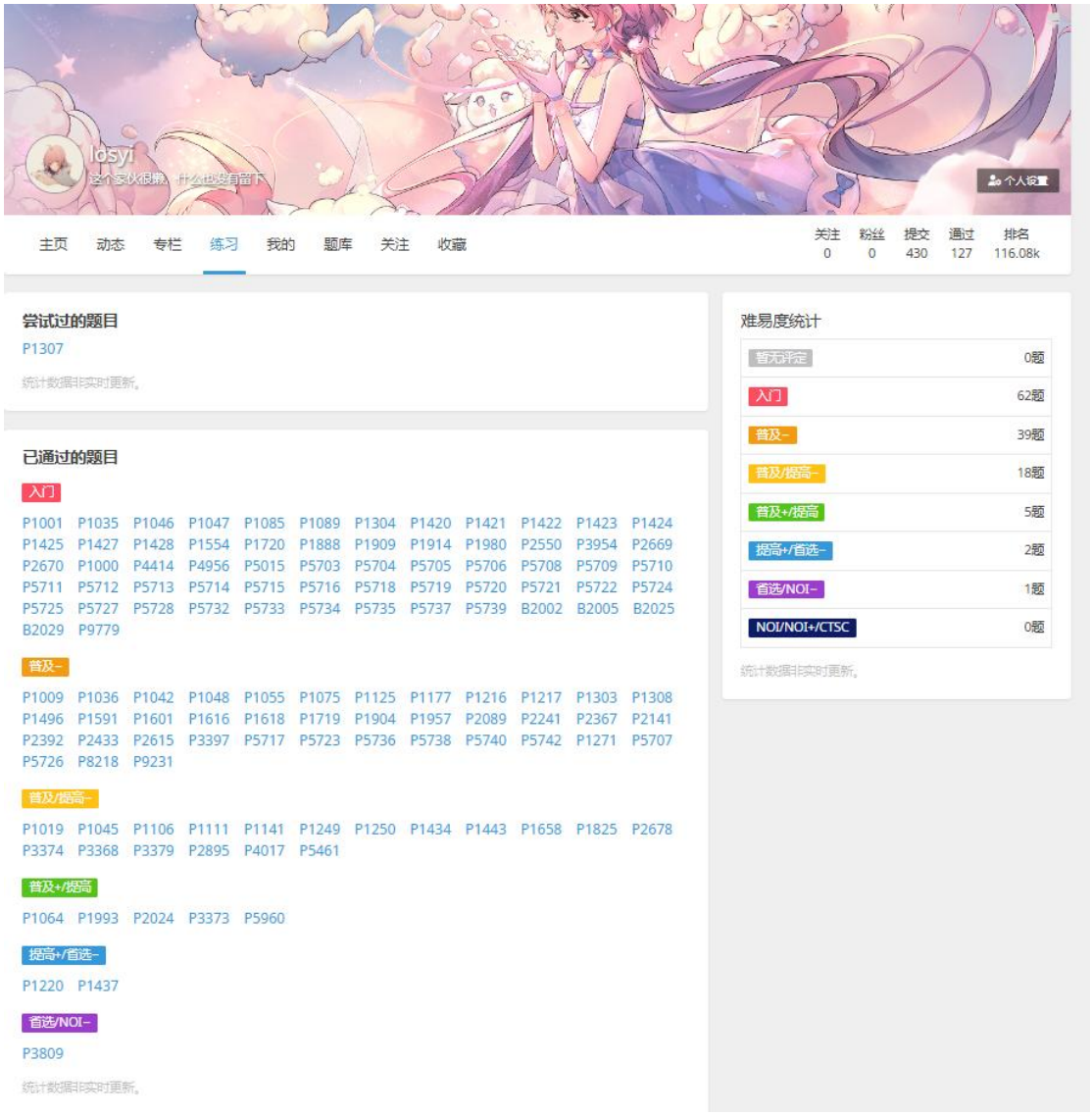

图 1-1 洛谷个人主页截图

2. P1220 解题报告

2.1 题目分析

P1220：关路灯 提高+/省选−

一条路上安装了 n 盏路灯，每盏灯的功率不同。老张住在这条路中间某一路灯旁，他的工作是每天早上天亮时一盏一盏地关掉这些路灯。

为节省电费，老张记录下了每盏路灯的位置和功率，他每次关灯时也都是尽快地去关，但是老张不知道怎样去关灯才能够最节省电。他每天都是在天亮时首先关掉自己所处位置的路灯，然后可以向左也可以向右去关灯。现在已知老张走的速度为 1m/s，每个路灯的位置(是一个整数，即距路线起点的距离，单位：m)，功率(W)，老张关灯所用的时间很短而可以忽略不计。现在需要安排关灯顺序，使从老张开始关灯时刻算起所有灯消耗电最少。

分析题目可知，老张每次可以选择向左或向右移动，并关闭一盏路灯。路灯在被关之前一直消耗能量，而总能量时间有关，移动时的时间会导致灯消耗更多的功率，因此老张的移动路径和关灯顺序直接影响了最终的能耗。

第一眼看上去很容易想到贪心方法，因为我们想要尽可能减少能量消耗，那应该每一步都选择关闭功率较大的那一侧的灯，这显然是不可取的，贪心算法只关注当前增量能耗最小的方向，而不考虑后续路径的能耗情况。如果当前灯功率较低，但选择关闭该灯，让老张走很远，这可能会让高功率的灯持续消耗更长的时间。老张关灯的顺序对未关灯的能耗有积累效应。例如，优先关闭远处的低功率灯可能会让近处的高功率灯一直保持开启，导致总能耗增加。贪心算法无法全局评估这些复杂的积累效应。

如果我们考虑的是当前状态的总功率，我们是应该关闭距离近但功率小的还是距离远但功率大的？这是无法确认的，局部最优性并不保证全局最优性，尤其在灯的功率和位置差异较大时，其局部最优选择可能导致次优解。这是因为本题涉及到累积效应和整体规划问题，而贪心算法只关注当前增量能耗的最小值。

最优解应使用动态规划(区间 DP)来求解，通过逐步计算所有子区间的最优值并合并，才能找到全局最优解。通过分析问题，我们可以确定该问题具有最优子结构性质，即在解决小规模区间问题的基础上，可以递推出更大的区间问题的解。因此，我们可以采用区间动态规划(Interval DP)来解决本问题。

另一方面，通过贪心方法的思考，即使我们无法确认当前的最优选择，由于题目的性能要求较低(路灯数少于 50)，我们也可以利用 DFS 搜索求解，在关灯问题中，老张需要在每一步选择向左或向右关灯，这形成了一棵搜索树。DFS 可以通过递归枚举老张关灯的所有可能顺序，并计算每一种顺序的总能耗，从中选择最优解。相比 DP 方法，DFS 不必考虑复杂的状态转移方程，适合快速解决这种小型问题。

2.2 算法设计

我们要找到一种最优关灯顺序，使得老张关闭所有路灯时的总能耗达到最小。这是一类典型的优化问题，既需要在递归中探索可能的路径，也需要设计合理的剪枝规则来避免无效的计算。问题的规模限制在 50 以内，这意味着我们可以使用深度优先搜索(DFS)结合剪枝的思路，逐步探索所有可能的关灯顺序，并通过动态维护状态找到全局最优解。

首先，我们需要明确核心状态和转移方式。在本题中，老张可以选择向左或向右关闭未关闭的灯，每次移动都会增加总能耗，能耗的大小取决于移动的距离和当前剩余功率的总和。因此，问题的关键在于：如何选择下一盏灯，使得最终关灯的总能耗最小。这种决策过程会受到前一步选择的影响，形成了一个依赖性很强的搜索树。

在搜索的过程中，每个状态可以通过三个变量描述：

(1) 当前老张所处的灯的位置(索引 now)。

(2) 剩余功率的总和(total_power)，影响未来关灯时的功耗。

(3) 当前累计的能耗(current_energy)，表示当前路径的能耗总值。

为了简化计算和提高效率，我们采用递归的方式进行深度优先搜索，同时动态更新这些状态变量。搜索树的每一层都对应老张关闭一盏灯的操作。在每次操作中，我们需要选择关灯的方向(向左或向右)，并计算移动带来的能耗增加，然后递归地探索下一盏灯的关闭情况。由此得代码框架：

```
// DFS 搜索函数
void dfs(int now) {
    bool hasNext = false; // 标记是否还有灯未关闭
    // 剪枝：当前能量消耗超过最优解时直接返回
    if (current_energy >= ans) return;
    // 向右探索
    for (int i = now + 1; i <= n; i++) {
        if (!visited[i]) { // 找到未关闭的灯
            visited[i] = true;
            // 更新当前能量和剩余功率
            dfs(i); // 递归探索
            // 回溯：恢复状态
            hasNext = true;
            break;//注意break!
        }
```

算法在每次递归调用中，动态更新当前能耗和剩余功率。当老张向某一方向移动并关闭灯时，总能耗会增加一个值，该值是移动距离与剩余功率的乘积。同时，关闭灯的功率会从剩余功率中减去。完成该分支的递归后，需要回溯到上一层状态，这要求恢复能耗和剩余功率的值，并将刚才关闭的灯重新标记为未关闭。这样的状态回溯操作保证了每条搜索路径的独立性，使得算法能够准确地遍历所有可能的关灯顺序。

```
if (!visited[i]) { // 找到未关闭的灯
            visited[i] = true;
            // 更新当前能量和剩余功率
            int distance = lamps[i].position - lamps[now].posi
tion;
```

```
        current_energy += distance * total_power;//distance 即时间,相乘结果为功耗
   total_power -= lamps[i].power;//当前剩余功率
            dfs(i); // 递归探索
            // 回溯：恢复状态
            total_power += lamps[i].power;
            current_energy -= distance * total_power;
            visited[i] = false;
            hasNext = true;
            break;//注意 break!
        }
```

此外，剪枝也是必要的，否则 2^50 同样会造成 TLE，具体而言，如果当前路径上的累计能耗已经超过当前的最优解，就没有必要继续探索该路径，因为它不可能产生更优的结果。这样的剪枝规则能够显著减少搜索空间，使算法能够在合理时间内求解规模较大的问题。此外，只有当所有灯都关闭时，才能更新最优解。这种终止条件保证了搜索树的完整性，同时避免了不必要的回溯操作。

```
bool hasNext = false; // 标记是否还有灯未关闭
   // 剪枝：当前能量消耗超过最优解时直接返回
   if (current_energy >= ans) return;
```

总结得算法处理流程如下：

(1) 初始化数据：从输入中读取灯的数量 n 和老张的初始位置 pl，同时记录每盏灯的位置和功率信息，并计算所有灯功率的总和 total_power。将老张所在的灯标记为已关闭，并从 total_power 中扣除其功率值，作为初始状态。

(2) 定义搜索状态与递归函数：构造一个递归函数 dfs(now)，表示从当前老张所在的位置 now 开始搜索所有可能的关灯路径。函数内部会动态更新三个关键状态。

(3) 剪枝优化：在每次递归进入新状态前，判断当前累计能耗是否已经超过目前已知的最优解 ans。如果是，则直接终止当前分支的搜索，这一规则避免了不必要的冗余计算。

(4) 向右探索分支：在递归函数中，从当前灯向右查找第一个未关闭的灯，计算老张从当前位置移动到该灯的距离。根据移动距离和当前的剩余功率，更新

(5) 能耗，并将该灯标记为已关闭。然后递归调用 dfs 函数以继续探索。在完成右侧分支的递归后，恢复搜索前的状态。将总能耗减去移动增加的部分，恢复剩余功率，并将刚才关闭的灯重新标记为未关闭，以便探索其他分支。

(6) 向左探索分支：类似右侧分支，从当前灯向左查找第一个未关闭的灯，并重复上述操作。计算移动距离并更新能耗和状态，然后递归调用 dfs 以继续探索。在完成左侧分支的递归后，同样恢复能耗、剩余功率和灯的状态，为继续探索其他路径做好准备。

(7) 检查是否到达叶子节点：如果在当前递归中左右两侧均没有可以继续探索的灯，说明所有灯已关闭。此时更新当前路径的最优能耗值 ans，记录是否找到比之前更优的解。

(8) 输出结果：在完成整个搜索后，ans 中存储的就是全局最优解，表示最小的关灯总能耗。将其输出作为结果。

2.3 性能分析

每次 DFS 递归时，老张最多只有两个可选方向(向左或向右)。理论时间复杂度为 $O(2^n)$，但每个分支一旦被访问，其状态就被更新，后续分支在部分情况下会因剪枝被跳过。因此，实际的搜索树深度为 n，分支因剪枝减少，进一步降低了实际复杂度。

2.4 运行测试

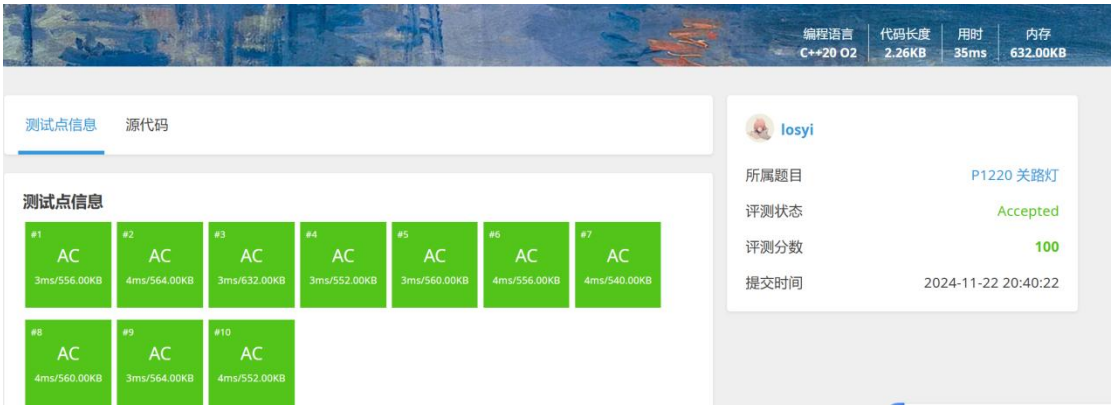

图 2-1 P1220 运行结果图

代码长度 2.26KB,用时 35ms ,内存 632.00KB

3. P3809 解题报告

3.1 题目分析

P3809：后缀排序 省选/NOI-

读入一个长度为 n 的由大小写英文字母或数字组成的字符串，请把这个字符串的所有非空后缀按字典序(用 ASCII 数值比较)从小到大排序，然后按顺序输出后缀的第一个字符在原串中的位置。位置编号为 1 到 n。n 小于 10^6。

题目要求实现一个后缀排序，意思不难理解，例如，对于字符串 ababa，有以下后缀：ababa，baba，aba，ba，a。

排序后为：a，aba，ababa，ba，baba

对应在原串的位置为：5，3，1，4，2

如果我们采用暴力方法直接生成所有后缀及其初始位置，再进行排序，时间复杂度将达到 n^2，空间复杂度同样达到 n^2,以下面的代码为例，具体而言，在

buildSuffixArrayBruteForce 函数中，有一个循环用于生成所有后缀及其起始位置，循环从 i = 0 到 i = n - 1(n 是输入字符串 s 的长度)，每次循环执行的操作是构造一个长度最长为 n 的子字符串(通过调用 substr 函数)并将其与起始位置组成 pair 后插入到 vector 中。构造子字符串 substr 的时间复杂度与提取的子串长度有关，平均来看每次操作时间复杂度大致为 O(n) ，这个循环总共执行 n 次，所以这部分的时间复杂度是 O(n^2) 。

```
// 生成所有后缀及其起始位置
    for (int i = 0; i < n; ++i) {
        suffixes.emplace_back(s.substr(i), i);
    }
    // 按字典序对后缀排序
    sort(suffixes.begin(), suffixes.end());
    // 提取排序后后缀的起始位置
    vector<int> suffixArray;
    for (const auto& suffix : suffixes) {
        suffixArray.push_back(suffix.second);
    }
```

经测试，该算法只能通过五个测试用例，6 个 MLE，用时 900ms，内存占用超过了上限。

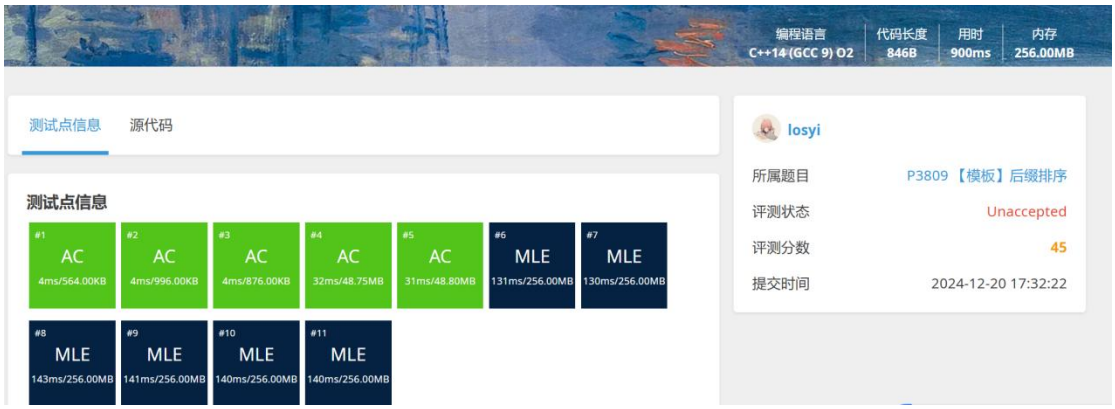

图 3-1 P3908 暴力算法运行结果

为了更高效的解决问题，我们需要使用基于倍增的后缀数组算法，那么什么是后缀数组呢？

后缀数组(Suffix Array)是一种高效的字符串数据结构，用于处理字符串相关的各种问题，如字符串的排序、匹配、查找重复子串等。后缀数组的核心思想是通过对字符串的后缀进行排序来构建一个数组，其中每个元素表示该后缀在原字符串中的起始位置。

为了加速构建后缀数组的构建，我们需要使用**倍增**的思想，倍增在整个后缀排序过程中起到了关键作用。开始时，我们只对长度为 1 的后缀进行排序，这很简单，因为只需要比较单个字符即可得到一个初步的顺序。然后，将考虑的后缀长度倍增，比如变为 2。这时，对于两个后缀的比较，不能仅仅看开头的一个字符了，而是要综合考虑前半部分(已经在长度为 1 时排好序的部分，也就是基于 rank)以及新增的后半部分(通过 rank + len 来获取后半部分对应的排名情况)。每次倍增后，都利用上一轮的排序结果(存储在 rank 数组中)来进一步细化排序，随着倍增次数的增加，最终能涵盖整个后缀的长度，使得所有后缀都能按照完整的字典序准确排序。这样逐步扩大比较范围的方式，避免了一开始就对很长后缀进行复杂比较的难题，把大问题拆解成多个逐步递进的小问题来解决。

那么什么是 rank 呢，rank 数组的作用又是什么？

在倍增法中，我们通过每一轮增加后缀的比较长度(即倍增)，逐步扩展排序范围。每一轮的排序需要对后缀进行比较，在比较时，我们不仅需要考虑后缀的第一个字符(或者说是当前已排序部分的内容)，还需要考虑后续部分的内容。

rank 数组的作用是为每个后缀提供一个"秩"(字典序排名)，使得我们能够高效地比较后缀。第一次排序时，rank[i] 是后缀的第一个字符的字典序(ASCII 值或其对应的排序排名)。第二次排序时，rank[i] 表示后缀的前两个字符的字典序。以此类推，每增加一倍的后缀长度，rank[i] 就存储该后缀的前 2^k 个字符的字典序信息。通过 rank 数组，我们可以直接使用当前后缀的排名来判断它们在字典序中的位置，而不必每次都重新对整个后缀进行字符串比较。这大大提高了排序的效率。

在后缀排序的过程中，如果不使用 rank 数组，每次比较两个后缀时都需要逐个字符地比较这两个后缀的所有字符，这样的操作非常低效，尤其是当后缀长度较大时。rank 数组使我们能够通过较短的字符串(即当前后缀的前半部分)来决定整个后缀的排序。比如，当我们对两个后缀进行排序时，我们不仅看它们当前的排名(rank[a] 和 rank[b])，如果相同，再比较它们后续部分的排序(rank[a + len] 和 rank[b + len])。这样一来，我们避免了每次都进行全量的字符比较，只需比较部分信息。在倍增法每轮排序后，rank 数组会被更新。新的 rank 数组基于当前排序后的后缀数组，通过比较后缀的字典序关系来为每个后缀赋予新的秩。如果两个后缀在排序时的字典序相同，它们会被赋予相同的秩，否则秩会递增。

这个过程利用了 rank 数组来高效更新后缀的秩，从而确保每一轮排序能反映出后缀的真实字典序。

我们如此麻烦地解决后缀排序问题当然不仅仅是为了排序，它有着更多方面地应用，比如解决字符串匹配问题、最长公共子串问题、最长公共前缀问题，当然，这与本题无关了。下面我可以开始算法设计了。

3.2 算法设计

基于上述分析，整个解题算法设计如下：

(1) 初始化后缀数组和秩数组：将字符串 s 的所有后缀存入 suffixArray，并记录每个后缀的起始位置。使用 rank 数组为每个后缀赋予初始排名，根据后缀的首字符的 ASCII 值直接确定其排名。初始排名反映了长度为 1 的后缀的字典序。

```
int n = s.size();
    vector<int> suffixArray(n), rank(n), tmpRank(n);
    // 初始化后缀数组和秩数组
    for (int i = 0; i < n; ++i) {
        suffixArray[i] = i;
        rank[i] = s[i];
    }
```

(2) 倍增排序：倍增的核心是每次扩展后缀的比较范围，将排序依据从单字符扩展为长度为 2^k 的范围。排序依据于比较当前后缀的前半部分(即当前的排名 rank[a])。如果前半部分排名相同，则比较后缀的后半部分(即 rank[a+len])。若后半部分越界，排名设为 -1。通过倍增逐步扩大排序的范围(从长度为 1 到 2，再到 4、8……)。自定义排序函数 cmp 保证了排序的稳定性，即能够根据当前排名准确比较两个后缀的字典序。

```
// 自定义比较函数：首先按前半段(rank)排序，如果相等则按后半段(rank+len)排序
        auto cmp = [&](int a, int b) {
            if (rank[a] != rank[b]) return rank[a] < rank[b];
            int ra = (a + len < n) ? rank[a + len] : -1;
            int rb = (b + len < n) ? rank[b + len] : -1;
            return ra < rb;
            };
```

(3) 更新排名：在排序完成后，重新为每个后缀分配新的排名。

如果当前后缀与前一个后缀不同(即 cmp 比较返回 true)，分配一个新的秩；否则，沿用前一个后缀的秩。通过更新 rank 数组，为下一轮排序提供新的比较依据。避免了直接比较字符串内容，提高了排序效率。

```
  tmpRank[suffixArray[0]] = 0;//第一个后缀的秩为0
        for (int i = 1; i < n; ++i) {
   tmpRank[suffixArray[i]] = tmpRank[suffixArray[i - 1]];//先令秩相同
            if (cmp(suffixArray[i - 1], suffixArray[i])) {
   tmpRank[suffixArray[i]]++;//若前一个后缀小于后一个后缀，则秩加1
```

```
        }
         }
         rank = tmpRank;
```

(4) 输出后缀数组：最终的 suffixArray 存储了所有后缀按字典序排序后的起始位置。 按题目要求将位置编号从 0 基改为 1 基。

下面我们举一个稍微复杂的例子来说明算法流程，假设给定的字符串为 **i love luotianyi**

(1) 初始化：后缀数组表示每个后缀的起始位置，初始时，后缀数组是从 0 到 n-1 的索引序列：

```
suffixArray = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

rank[i] 表示字符串中每个位置的字符的 ASCII 值：

```
 rank = [105, 108, 111, 118, 101, 108, 117, 111, 116, 105, 97,
 110, 121, 105]
```

(2) 第一次排序：长度 len = 1：算法首先对每个后缀的第一个字符进行排序。我们需要按照 rank[] 排序后缀数组。这里使用 rank[] 来定义比较规则：首先比较后缀的首字符(即 rank[i])。我们通过自定义比较函数进行排序：比较两个后缀的位置 a 和 b：首先看它们的第一个字符的 rank 值。如果相等，再比较它们的第二个部分(长度为 1 的后缀比较只有一个字符)。

排序后，suffixArray 更新为：

```
suffixArray = [10, 4, 13, 9, 0, 1, 5, 11, 7, 2, 8, 6, 3, 12]
```

对应的后缀为：

```
10 anyi
 4 eluotianyi
 13 i
 9 ianyi
```

```
0 iloveluotianyi
1 loveluotianyi
5 luotianyi
11 nyi
7 otianyi
2 oveluotianyi
8 tianyi
6 uotianyi
3 veluotianyi
12 yi
```

排序后，秩数组 rank[] 根据新的顺序进行更新。我们逐个比较后缀，给它们赋予新的秩值。初始时，第一个后缀的秩是 0，其他后缀的秩由前一个后缀的秩来决定。如果两个后缀的前缀部分相同，那么它们的秩相同；否则，后缀的秩加 1。排序后，rank[] 更新为：

```
rank = [4, 5, 9, 12, 1, 6, 11, 8, 10, 3, 0, 7, 13, 2]
```

(3) 第二次排序：此时我们将排序长度增大到 2，排序依据不仅是后缀的前半部分，还包括后半部分。对于每个后缀，比较其前 len 个字符和后 len 个字符的排名，len 在本轮排序时是 2。具体步骤：比较后缀的前 len 个字符(由 rank[a] 和 rank[b] 表示)。如果前 len 个字符相同，再比较后 len 个字符的位置，比较的是 rank[a + len] 和 rank[b + len]。

排序后的 suffixArray：

```
suffixArray = [10, 4, 13, 9, 0, 1, 5, 11, 7, 2, 8, 6, 3, 12]
```

排序结果和第一次排序完全一样，因为前 len = 1 的字符已经完全区分了所有后缀，后半部分的比较并没有改变顺序：

```
rank = [4, 5, 9, 12, 1, 6, 11, 8, 10, 3, 0, 7, 13, 2]
```

(4) 第……次排序：同样的，我们将排序的长度再增加到 4、8，继续按相同的规则排序，直到得到最终结果：

```
10 anyi
4 eluotianyi
13 i
9 ianyi
0 iloveluotianyi
1 loveluotianyi
5 luotianyi
11 nyi
7 otianyi
2 oveluotianyi
8 tianyi
6 uotianyi
3 veluotianyi
12 yi
```

总之，本算法以倍增法为核心通过逐步增大排序的后缀长度，从长度 1 开始，逐步扩展到更长的部分，利用 rank[] 数组来对后缀进行有效排序。

3.3 性能分析

在算法中，我们的排序会在每一次迭代中将比较的范围加倍，即长度从 1、2、4、8、... 依次增长，由于每次排序时 len 的值翻倍，所以最多需要进行 logn 次排序。每次排序的复杂度是 $O(n\log n)$，而总的排序次数是 $\log n * n\log n$ 次。综上所述，整个算法的时间复杂度就是：$O(n\log^2 n)$

3.4 运行测试

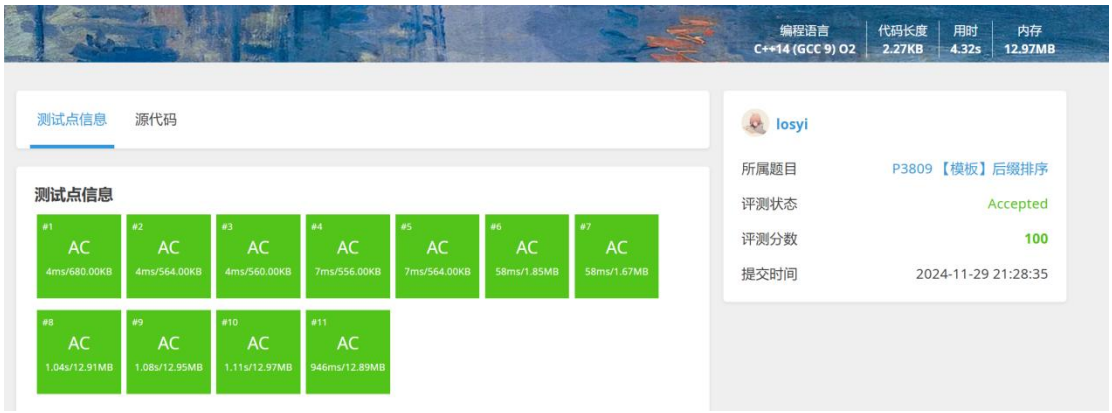

图 3-2 P3908 倍增快排算法运行结果

用时 4.32s,内存 12.97MB

虽然 AC 了题目但是发现用时却比暴力方法要长，可见在某些输入条件下算法的时间复杂度表现得不是很好，实际上，可以考虑用基数排序对排序部分进行优化，原始代码通过快排排序后缀数组，并在每次排序后重新计算秩。这个过程本质上是分阶段地按后缀的前缀长度进行排序。可以通过基数排序来替代快排进行优化，基数排序的稳定性可以保证每次排序时不会破坏相同秩的后缀的相对顺序，从而减少排序中的递归复杂度。

4. P1993 解题报告

4.1 题目分析

P1993：小 K 的农场 普及+/提高

小 K 在 MC 里面建立很多很多的农场，总共 n 个，以至于他自己都忘记了每个农场中种植作物的具体数量了，他只记得一些含糊的信息(共 m 个)，以下列三种形式描述：

农场 a 比农场 b 至少多种植了 c 个单位的作物；

农场 a 比农场 b 至多种植了 c 个单位的作物；

农场 a 与农场 b 种植的作物数一样多。

但是，由于小 K 的记忆有些偏差，所以他想要知道存不存在一种情况，使得农场的种植作物数量与他记忆中的所有信息吻合。

如果存在某种情况与小 K 的记忆吻合，输出 Yes，否则输出 No。

本题题干较短，核心在于判断是否存在一种特定的作物种植数量分布，以满足小 K 所记忆的所有信息。题目中提供的信息可以归纳为三类不等式约束，具体包括：

农场 a 比农场 b 至少多种植了 c 个单位的作物；

$$x_a - x_b \geq c$$

农场 a 比农场 b 至多种植了 c 个单位的作物；

$$x_a - x_b \leq c$$

农场 a 与农场 b 种植的作物数一样多。

$$x_a - x_b = c$$

显然，题目中的约束条件可以转化为差分约束系统的形式，差分约束问题的核心是利用加权图的路径松弛思想，通过检查是否存在负环来判断约束系统是否有解：如果不存在负权环，则说明约束系统有解。如果存在负权环，则说明约束系统无解。

对给定的差分约束系统 Ax≤b，其对应的约束图是一个带权重的有向图 G=(V,E)，这里，)约束图中引入一个额外的结点 v0，从其出发可以达到其 他所有结点。因此结点集合 V 由代表每个变量 xi 的结点 vi，额外的结点 v0 组成。边集合 E 包含代表每个差分约束的边，同时包含 v0 到其他所有结点的边(v0,vi)，i=1,2，…，n。边的权重：如果 xj-xi≤bk 是一个差分约束条件，则边(vi,vj)的权重为 ω(vi,vj) = bk，而从 v0 出发到其他结点的边的权重为 0。

如果图 G 不包含权重为负值的回路，则

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n))$$

是该系统的一个可行解。

如果图 G 包含权重为负值的回路，则该系统没有可行解。考虑任意一条边(vi,vj)∈E，根据三角不等式有：

$$\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$$

$$\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$$

因此，令 $x_i = \delta(v_0, v_i)$ 和 $x_j = \delta(v_0, v_j)$ ，则 xi 和 xj 满足对应边(vi,vj)的差分约束条件，$x_j - x_i \leq w(v_i, v_j)$

而如果约束图包含权重为负值的环路，不是一般性，设权重为负值的环路为 $c = \langle v_1, v_2, \ldots, v_k \rangle$，这里 v1=vk。环路 c 对应下面的差分约束条件组：

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2), \\ x_3 - x_2 &\leq w(v_2, v_3), \\ &\vdots \\ x_{k-1} - x_{k-2} &\leq w(v_{k-2}, v_{k-1}), \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k). \end{aligned}$$

将不等式左侧求和，其和等于 0：vi 相互抵消，注： v1=vk。

将不等式右侧求和，其和等于ω(c)，即环路 c 的权重，有：0 ≤ ω(c)，

显然与 c 是权重为负值的环路相矛盾。故该组不等式不可能有解。

由上述分析可知，可以使用 Bellman-Ford 算法来求解差分约束系统。

Bellman - Ford 算法是一种用于解决单源最短路径问题的算法，尤其适用于处理边权值可能为负的有向图，Bellman-Ford 算法的核心思想是通过松弛操作迭代地更新每个节点到源节点的最短距离。具体而言，如果有一条边从节点 u 到

节点 v，且 distance[u] + weight(u, v) < distance[v]，则更新 distance[v] = distance[u] + weight(u, v)，直到所有节点的最短路径都得到更新。

```
if (dist[edge.from] != INT_MAX && dist[edge.to] > dist[edge.fr
om] + edge.weight)
            {
                dist[edge.to] = dist[edge.from] + edge.weight;
            }
```

Bellman-Ford 算法的一个重要特点是，算法通过最多 |V| - 1 次迭代就能够找出图中所有的最短路径，其中 |V| 是图中节点的数量。原因在于图中最短路径的最长边数不会超过 |V| - 1 条。

在完成 |V| - 1 次松弛后，再对每一条边进行一次松弛操作。如果有节点的距离发生变化，说明图中存在负权环。

```
 if (dist[edge.from] != INT_MAX && dist[edge.to] > dist[edge.f
rom] + edge.weight)
        {
            return false; // 存在负环，表示无解
        }
```

4.2 算法设计

那么如何将本题需求转化为图呢？

对于每种信息类型，我们将其转化为图中的一条边。本题的关键在于对第二种和第三种约束进行一定转换以满足差分约束系统的形式。

对于第一种信息，即农场 a 比农场 b 至少多种植了 c 个单位的作物，我们可以得到 $x_b - x_a \le -c$，即农场 b 到农场 a 的边权重为-c。

对于第二种信息，即农场 a 比农场 b 至多种植了 c 个单位的作物，我们可以得到 $x_a - x_b \le c$，即农场 a 到农场 b 的边权重为 c。

对于第三种信息，即两个农场种植的作物数一样多，为了保持形式的一致性，我们可以得到 $x_a - x_b \le 0$ 和 $x_b - x_a \le 0$ 我们可以得到即两个农场之间有两条边，权重都为 0。

```
if (type == 1)
        {
            cin >> a >> b >> c;
            edges.push_back({b, a, -c}); // 农场 a 比农场 b 至少多 c -> x_b - x_a <= -c
        }
        else if (type == 2)
        {
            cin >> a >> b >> c;
            edges.push_back({a, b, c}); // 农场 a 比农场 b 至多多 c -> x_a - x_b <= c
        }
        else if (type == 3)
        {
            cin >> a >> b;
            edges.push_back({a, b, 0}); // 农场 a = 农场 b -> x_a - x_b <= 0
            edges.push_back({b, a, 0}); // 农场 b = 农场 a -> x_b - x_a <= 0
        }
```

综上所述，可得算法流程如下

(1) 初始化：读取农场数量 n 和约束数量 m。遍历所有的约束，根据约束的类型，添加对应的有向边到边集中，再为虚拟源点 0 添加 n 条边，指向所有结点。

(2) 实现 Bellman 算法：

① 初始化距离数组 dist，将除虚拟源点(设为 0)外的所有顶点的距离值设为正无穷大(INT_MAX)，即 dist [i] = INT_MAX(i = 1, 2, …, n)，并将 dist [0] 设为 0。

② 进行 n 轮迭代(因为最多经过 n - 1 次迭代就能得到单源最短路径，这里多进行一次迭代用于检测负环)：对于边集中的每一条边 (u, v, w)(其中 u 是起点，v 是终点，w 是边权)：如果 dist [u] 不等于正无穷大且 dist [v]> dist [u]+w：更新 dist [v] = dist [u] + w。

③ 检查负环：再次遍历边集中的每一条边 (u, v, w)：如果 dist [u] 不等于正无穷大且 dist [v]> dist [u]+w：则存在负环，返回 false，表示不存在满足条件的作物数量分配方案。如果没有发现这样的边，则返回 true，表示存在满足条件的作物数量分配方案。

(3) 输出：如果 Bellman - Ford 算法返回 true，则输出 “Yes”，表明存在一种情况，使得农场的种植作物数量与小 K 记忆中的所有信息吻合。如果返回 false，则输出 “No”，表明不存在这样的情况。

4.3 性能分析

处理 m 条约束，每条约束都会被转换为一条或两条边，边数最多为 2m，我们从虚拟源点 0 连向每个节点 i，共添加 n 条边，因此这部分的时间复杂度为 $O(m + n)$

Bellman-Ford 算法中，我们需要进行 n+1 次松弛操作，其中 n 是农场的数量，额外的一次用于检测负权环。每次松弛操作要处理所有边(2m+1)，因此，总松弛复杂度为 $O(n*(\mathrm{m}+n))$

检测负权环时，再遍历所有边一次，尝试松弛，时间复杂度为 $O(m+n)$

综上，算法总时间复杂度为 $O(n*(m+n))$

空间复杂度取决于存储边的数组 Edges 和距离数字 dist，复杂度为 $O(m+n)$

4.4 运行测试

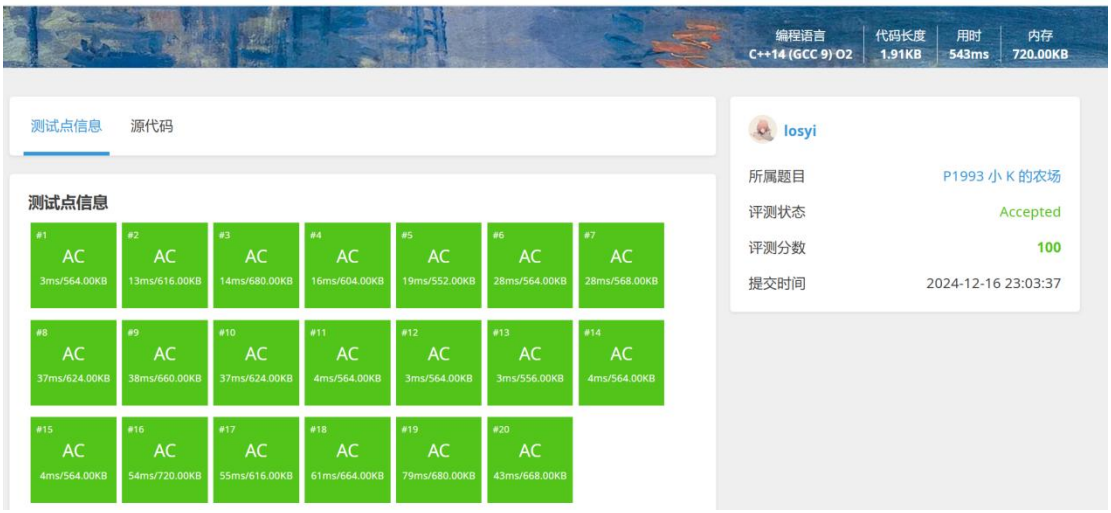

图 4-1 P1993 运行结果图

用时 543ms，内存 720MB，时间和空间表现均良好，符合算法预期。

5. P1437 解题报告

5.1 题目分析

P1437:敲砖块 提高+/省选−

在一个凹槽中放置了 n 层砖块，最上面的一层有 n 块砖，从上到下每层依次减少一块砖。每块砖都有一个分值，敲掉这块砖就能得到相应的分值，如下所示：

|14 15 4 3 23|

|33 33 76 2|

|2 13 11|

|22 23|

|31|

如果你想敲掉第 i 层的第 j 块砖的话，若 i = 1，你可以直接敲掉它；若 i > 1，则你必须先敲掉第 i - 1 层的第 j 和第 j + 1 块砖。

你现在可以敲掉最多 m 块砖，求得分最多能有多少。

首先暴力法是完全不可取的，总砖块数为 n 的平方级别，再加上 m 导致的组合数，时间复杂度将相当高。

分析题意可知敲掉某层的砖块需要满足上一层砖块已被敲掉的条件，也就是依赖于前面的状态，具有子结构性质。具体来说，敲掉第 i 层第 j 块砖时，得分的最大值取决于之前层(第 i−1 层)的状态。这种计算具有重复性，适合用动态规划进行求解。个状态的最优解可以通过子状态的最优解推导出来。例如，第 i 层的最大得分可以通过第 i−1 层的最大得分计算得到。

现在我们考虑设计状态转移方程，用动态规划解决问题。

5.2 算法设计

为方便处理，符合我们平常列方程的习惯，我们先对输入进行转置处理，定义 dp[i][j][k] 表示在第 i 层，选了 j 列，移除了 k 个砖块的最大得分，同时，可以观察到，若我们选择敲掉(i,j),则第 i 行（原来的第 n-j+1 列）的砖块都会被敲掉，且还需要敲掉 i-1 行的 j-1 列，因此我们可以维护一个前缀和数组便于后续的计算。

```
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            transformedValue[i][j] = brickValue[j][n - i + 1];
// 转置砖块的分值
```

```
            prefixSum[i][j] = prefixSum[i][j - 1] + transformedV
alue[i][j];  // 计算每列的前缀和
        }
    }
```

现在考虑如何设计 DP 函数，根据前面我们对 dp 数组的定义,第 0 层没有砖块，是动态规划的边界条件。因此，无论累积敲掉多少砖块，得分均为 0。这一条件为后续的层次递推提供了基础。动态规划的核心是逐层递推计算每个状态的值。在代码中，需要用到三重循环依次完成层、列、砖块数量的遍历。核心部分是动态规划的状态转移公式：

```
if (buff[i - 1][j - 1>=0?j-1:0][k - j] >= 0) {  // 如果前一个状
态有效
dp[i][j][k] =  buff[i - 1][max(0, j - 1)][k - j] + prefixSum[i]
[j]; // 更新当前状态的最大得分
  //可以观察到，若我们选择敲掉(i,j),则第 i 行（原来的第 n-j+1 列）的砖
块都会被敲掉，且还需要敲掉 i-1 行的 j-1 列
  buff[i][j][k] = max(buff[i][j + 1][k], dp[i][j][k]);
 //正向依赖关系,不能直接在 dp 数组上进行更新，dp 数组的值是在计算过程
中会被覆盖的
```

首先进行状态有效性检查:buff[i-1][j-1][k-j] >= 0 表示上一层状态有效，保证当前层计算的合法性。k - j 表示当前累积敲砖数 k 减去敲掉当前层砖块所需的砖块数量。j - 1 >= 0 检查上一层列的范围是否合理。

状态转移逻辑:dp[i][j][k] = buff[i-1][...][...] + prefixSum[i][j]表示从上一层的有效状态 buff[i-1] 转移到当前层 dp[i][j][k]，加上当前列砖块的得分。prefixSum[i][j] 是预处理得到的当前列砖块得分总和，避免逐块累加。

那么这里为什么需要一个 buff 数组呢?

在动态规划问题中，我们通常希望当前的状态转移只依赖于之前已经计算完成的状态，而不会对后续状态产生影响。然而，在本题中，由于状态转移过程中存在 “前向依赖”，即当前状态可能依赖后续状态的值（违背了无后效性原则），

因此需要引入一个额外的辅助数组 buff，以避免在计算过程中覆盖掉尚未使用的值。

什么是**无后效性原则**?

在动态规划中，无后效性意味着：当前状态的值只依赖于之前已经计算完成的状态（通常是上一次迭代的结果）。当前状态的更新不会影响同一轮迭代中尚未计算的状态。例如，经典的背包问题中，状态 dp[i][w] 只依赖于前一轮的 dp[i−1][w] 的结果，而不会影响同一轮的其他状态。

本题的状态转移关系如下 dp[i][j][k] = max(依赖的上一层状态, 当前得分)在计算过程存在前向依赖关系，即 dp[i][j][k] 的值不仅依赖于 dp[i-1][j'][k'] 的状态，还需要从同一层尚未更新的状态 dp[i][j+1][k] 中获取值。如果直接在 dp 数组上进行更新，由于 dp 是原地更新的，会导致已经更新的状态覆盖了尚未使用的值，从而使后续的状态计算错误。

从代码中可以看到，在计算 dp[i][j][k] 时，需要取当前层 j+1 列的最大值：

```
buff[i][j][k] = max(buff[i][j+1][k], dp[i][j][k])
```

这里的 buff[i][j+1][k] 是第 i 层的 j+1 列尚未完全更新的值。如果我们直接在 dp 数组上更新状态，那么 dp[i][j+1][k] 的值可能已经被修改，导致当前层状态计算出错。这种**前向依赖关系**导致了无法直接使用 dp 数组进行原地更新。

为了解决上述问题，防止状态覆盖，代码中引入了一个辅助数组 buff，用于存储当前层的中间计算结果。buff 的作用是缓冲当前层的计算结果在计算当前层 dp[i][j][k] 时，buff[i][j][k] 用于存储第 j 列以及后续列（即 j, j+1, ...）的最大值。buff[i][j][k] 是从右往左累积的，因此可以保证 buff[i][j+1][k] 在更新 buff[i][j][k] 之前不会被覆盖。

换言之，buff 将当前层的状态计算与更新隔离，确保不会因为直接修改 dp 数组而导致前向依赖问题。借助 buff，我们可以先计算右侧列的值，再根据右侧列的结果更新左侧列，满足状态转移的需求。具体工作流程如下。

首先根据 dp[i-1] 的状态计算当前层 dp[i][j][k] 的基础值：

```
dp[i][j][k] = dp[i-1][相关列][k-砖块数量] + 当前层得分
```

再使用 buff 从右往左累积当前层的最大值：

```
buff[i][j][k] = max(buff[i][j+1][k], dp[i][j][k])
```

最终 buff[i][j][k] 保存了当前层的最大值，供下一层使用。

就此我们已经完成了核心逻辑的设计，最终得出算法流程如下：

(1) 数据处理：

读取砖块的层数 n 和最多敲掉的砖块数量 m，输入每一层砖块的分值矩阵 a[i][j],为了方便按列计算，我们对砖块分值矩阵进行转置，将原来的行列关系转化为列优先的表示，结果存储在 transformedValue 中对每列的砖块分值计算前缀和，存储在 prefixSum[i][j] 中，用于快速获取当前列的得分总和。

(2) 动态规划：

① 初始化：初始化 dp[0][0][k] = 0（第 0 层的边界条件）。初始化 dp[i][j][k] = -1 表示无效状态。

② 状态转移：从第 1 层开始逐层递推，每层的计算依赖于上一层的状态，具体过程如下：遍历第 i 层的所有列 j。遍历砖块数量 k，尝试敲掉第 j 列的砖块。为了确保转移的合法性，需检查上一层的状态 dp[i-1][相关列][k-j] 是否有效。当前砖块数是否在限制范围内 k ≤ m。若有效，则根据状态转移公式更新 dp[i][j][k]。

③ 更新 buff：由于本题存在“前向依赖”的问题（即当前状态的计算依赖于同一层尚未更新的状态），直接在 dp 数组上更新会导致状态覆盖，从而影响计算结果。因此，我们引入一个辅助数组 buff。在每次状态更新后，通过以下公式计算辅助数组：`buff[i][j][k] = max(buff[i][j + 1][k], dp[i][j][k])`

(3) 输出结果：遍历 DP 数组，找到最大得分并输出。

```
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            maxScore = max(maxScore, dp[i][j][m]);  // 选取最大得分
     }
    }
```

5.3 性能分析

读取输入并进行数据处理的时间复杂度$O(\mathrm{n}^2)$，动态规划循环遍历的时间复杂度$O(n^2 * m)$

故总时间复杂度为$O(n^2 * m)$

空间占用来源于 dp 数组 buff 数组，两者大小相同，均为$O(n^2 * m)$

5.4 运行测试

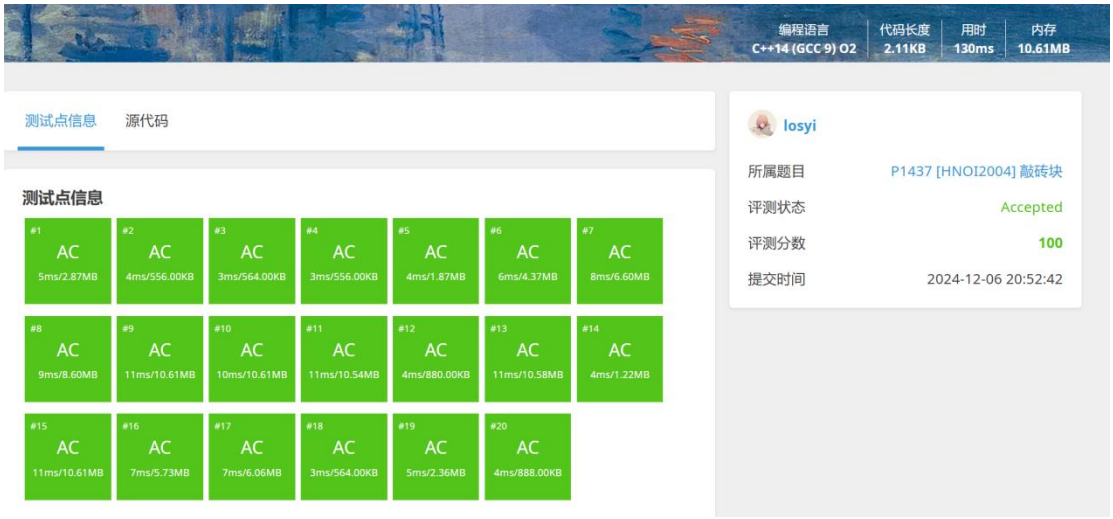

图 5-1 P1437 运行结果图

用时 130ms,内存占用 10.61MB，顺利通过了测试。

6. 总结

6.1 实验总结

本次实验中，通过完成洛谷平台上的各类题目，我进一步巩固了课内所学知识，对于差分约束，动态规划等新接触的算法更为熟悉，此外，实验也拓展了不少课外知识，例如并查集，线段树，不仅扩展了知识面，也提高了我的算法设计能力和编程实践能力，掌握算法优化的技术与方法，为今后从事算法设计、软件开发打下坚实的基础。

通过做题，交流，我也学到了不少程序技术和算法思想，具体来讲，我有以下一些收获和经验。

1. **简化代码**：在过往的语言课学习中，我们习惯于使用动态内存分配，而在做算法题时，我们通常习惯于开全局数组和变量，可以减少不必要的参数传递和初始化操作。通过实验我也学习了一些新的 STL 知识，比如通过 for(auto it:a) 对容器进行遍历，避免了传统 for 循环中冗长的输入，通过 sort 和自定义的 cmp 函数快速实现排序功能(尽管快排有时并不是最佳排序手段),pair 和 map 有时同样具有妙用，比如在本实验的迷宫问题中可以用于标记传送门位置，避免了用数组表示，更加直观地实现了想要的功能。

2. **问题转换**：有时题目想要考察的东西复杂，只需要我们分析得出正确的方向，转化为我们所熟悉的算法；也有时题目存在多种解，比如在关路灯问题中，状态转移方程难以得出，使用 DFS 爆搜却反而能简单地解决问题；还有的问题需要我们对数据进行一定地处理，以更便于操作，比如敲砖块问题。通过本次算法实验，我的思维变得更加开阔。

3. **学以致用**：在实验之前和实验之中我们学了不少经典的模板，比如二分，前缀和，差分，线段树等等算法，在做模板题中我们可能只是觉得这些算法很妙，但不知具体有什么用，随着做题数的增加，我发现其实在很多情景下都可以用这些算法对代码进行优化(虽然在很多情况下不做额外的优化也能通过题目)，这让我改进算法的技术能力得到了进一步提高。

6.2 心得体会和建议

在实验完成后，我有以下几点心得体会。

1.重视算法原理：在本次实验中，我解决了多个经典算法问题，包括动态规划、分治、搜索、差分约束系统等。在这些问题中，我发现算法的核心并不仅仅是完成一段代码，而是如何找到高效、清晰且具普适性的解法。有时候，我们离正确答案往往只差一两行代码，然而却会造成巨大的差别，需要我们通过调试和分析理清背后的原理和思想，就如同敲砖块问题一样，看似是典型的动态规划问题，但若是直接套模板而不使用 buff 数组，输出结果会大相径庭，关键是理解算法的本质和条件，在有限的时间内理清复杂约束关系并加以实现，这极大地锻炼了我的建模能力和抽象思维。

2.注意细节：本实验中也有一些简单的问题，比如运用贪心思想的删数问题，但是这道题目我却 WA 了好几次，一次是没有考虑存在删除前缀 0 的问题，一次是没有考虑输出 0 的问题，在竞赛过程中我们无法查看测试数据，这种细节性的失误往往造成不必要的损失解题时，最容易出错的地方往往是对边界条件的处理，以及特殊情况的覆盖。包括后缀数组问题中，也要考虑空字符串等特殊情形，这些细节的处理是让程序运行正确的关键。

3.从模版到灵活运用：很多基础算法都有固定的模版，比如并查集、树状数组、线段树、动态规划等。然而在实际题目中，模版往往需要根据题意进行修改和扩展，甚至重新设计部分代码逻辑。这让我体会到，模版只是工具，真正重要的是理解算法的原理，只有这样才能在需要时灵活运用。例如，在 P3368 树状数组的区间查询题目中，我需要对经典单点修改和区间查询的操作进行扩展，并结合前缀和的思想，实现了更高效的查询过程。

另一方面，我也发现自己仍存在不足，比如对新算法的学习深度不足，可能仅仅是了解了模板，知道算法流程和作用，但缺乏具体的应用，对算法的实现细节和优化原理理解还不够深入，对它们的变种和扩展思考较少。在树状数组 2，中，没有想到通过差分的方式来实现区间更新。此外，对于本课程的形式以及题目，我也有一些建议：

1. 希望能以题目组的形式来进行学习，有些题目虽然已经给出了所考察的算法，但是我们可能缺乏前置基础，若是能提供一系列从简到难的题目(比如从模板题到简单的运用题到需要进行一定转换的题)，对我们的算法学习和做题节奏会更有帮助。

2. 可以在洛谷团队中建立题单，类似于寒假刷题计划，可以随时看到大家的完成情况，有利于进行随堂检查完成情况，目前的随堂检查形式过于单一，无法考察，督促大家的算法学习。

3. 在本届实验完成后，可以考虑将大家的一些优秀代码或题解报告进行汇总展示，供彼此学习，也可以供以后的学生进行参考。虽然目前洛谷也有题解(相信大家实验过程中或多或少也有进行阅览和参考),但大多个人风格明显，偏向竞赛风格，且缺少注释和详细的说明，说实话对于基础差的人来说有点难理解，可读性也不是很好。如果明年题目会更新的话，将本年的这些题目纳入学弟学妹的寒假刷题单中也是个不错的主意。这样后来者会有更多优秀的代码和题解可以参考。

4. 关于课程的考勤情况，说实话也有所欠缺，一方面实验课缺乏监督，难以确保所有人都在场，当然就像 2 中提到的，通过团队题单可以看到大家的提交时间，可以依次作为考勤的一部分；另一部分考勤分数占比较少，可能大家觉得迟到或缺课也不所谓，或许课程的分数配比可以再进行一些改善。

总之呢，通过实验中遇到的挑战，我明确了未来学习的方向。在巩固基础的同时，我需要进一步探索高阶算法的细节与应用，如网络流、启发式搜索等。同时，深入研究算法在实际场景中的扩展应用也是一个重要课题。我继续努力，不断提升自己的问题解决能力，为更复杂的算法挑战做好准备。