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： 本硕博 2301

学　　号： U202315752

姓　　名： 陈宇航

指导教师： 王多强

完成日期： 2024.12.2

计算机科学与技术学院

2024 年 11 月

目　　录

1 题目完成情况

本次实验一共完成 50 道题目，具体如下：

- **分治** P2678, P1242, P1228
- **动态规划** P1220, P4170, P1854, P4999, P1437, P1434, P4017, P2018
- **贪心** P1106, P1080, P2512, P2127, P1658, P1090
- **搜索** P1443, P1135, P2895, P1825, P1433, P1784, P1141, P1019, P2349, P2324
- **后缀数组** P3809, P2852, P2870
- **树状数组** P3374, P3368, P3605
- **线段树** P1816, P3373, P1904
- **差分约束系统** P1993, P3275, P1250
- **并查集** P1111, P1196, P2024, P1197
- **网络流** P2756
- **前缀和** P1314
- **Tarjan** P1726, P3379
- **LCA/RMQ** P3379, P3865

完成的截图如下：

图 1-1 实验通过截图 1

✓	P2678	[NOIP2015 提高组] 跳石头
✓	P1242	新汉诺塔
✓	P1228	地毯填补问题
✓	P1220	关路灯
✓	P4170	[CQOI2007] 涂色
✓	P1854	花店橱窗布置
✓	P4999	烦人的数学作业
✓	P1437	[HNOI2004] 敲砖块
✓	P1434	[SHOI2002] 滑雪
✓	P4017	最大食物链计数
✓	P2018	消息传递

图 1-2　实验通过截图 2

✓	P1106	删数问题
✓	P1080	[NOIP2012 提高组] 国王游戏
✓	P2512	[HAOI2008] 糖果传递
✓	P2127	序列排序
✓	P1658	购物
✓	P1090	[NOIP2004 提高组] 合并果子 / [USACO06NOV] Fence Repair G
✓	P1443	马的遍历
✓	P1135	奇怪的电梯
✓	P2895	[USACO08FEB] Meteor Shower S
✓	P1825	[USACO11OPEN] Corn Maze S
✓	P1433	吃奶酪
✓	P1784	数独
✓	P1141	01迷宫

图 1-3　实验通过截图 3

100	P3275	[SCOI2011] 糖果
✓	P1250	种树
✓	P1111	修复公路
✓	P1196	[NOI2002] 银河英雄传说
✓	P2024	[NOI2001] 食物链
✓	P1197	[JSOI2008] 星球大战
✓	P2756	飞行员配对方案问题
✓	P1314	[NOIP2011 提高组] 聪明的质监员
✓	P1726	上白泽慧音
✓	P3379	【模板】最近公共祖先（LCA）

图 1-4　实验通过截图 4

✓	P1141	01迷宫
✓	P1019	[NOIP2000 提高组] 单词接龙
✓	P2349	金字塔
✓	P2324	[SCOI2005] 骑士精神
✓	P5691	[NOI2001] 方程的解数
✓	P3067	[USACO12OPEN] Balanced Cow Subsets G
✓	P3809	【模板】后缀排序
✓	P2852	[USACO06DEC] Milk Patterns G
✓	P2870	[USACO07DEC] Best Cow Line G
✓	P3374	【模板】树状数组 1
✓	P3368	【模板】树状数组 2
✓	P3605	[USACO17JAN] Promotion Counting P
✓	P1816	忠诚
✓	P3373	【模板】线段树 2
✓	P1904	天际线
✓	P1993	小 K 的农场

图 1-5　实验通过截图 5

2 分 治：P2678 跳石头

2.1 问题描述

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 $L \geq 1$ 且 $N \geq M \geq 0$。

接下来 N 行，每行一个整数，第 i 行的整数 D_i $(0 < D_i < L)$，表示第 i 块岩石与起点的距离。这些岩石按与起点的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 输入输出样例

输入	25 5 2 2 11 14 17 21
输出	4

表 2-1 输入输出示例

2.5 题目算法分析

如果我们遍历移走的石头，从 0 到 M 的话，我们会发现时间复杂度会达到 $O(M^n)$ 难以承受，所以我们采取其他方式思考。我们发现，题目要求选手跳的最短距离尽可能长，立刻想到二分答案的算法。那么什么是二分的区分区间呢？由于我们的结果是问的区间的长度，而且我们知道，最远距离是可控的（从起始到末尾的距离）。那么，我们可以以区间长度作为答案的二分区间。具体来说，我们每一次折半寻找答案的时候，是假定了选手能跳的最远距离为 mid，然后我们假设移走选手的下一块石头，如果移走之后还可以下雨这个距离，那么就让答案自增。每一次二分答案结束后，我们将其和传入的“假想答案”进行对比，再进行处理

2.6 代码分析

这一部分的完整代码请看附页6.2

2.6.1 二分答案 check 函数

```
bool check(int x)
{
    int num=0;
    int i =0;
    int now=0;
    while(i<=n)
    {
        i++;
        if(stone[i]-stone[now]<x)
        {
            num++;
        }
        else{
            now=i;
```

```
15              }
16          }
17          if (num>m)
18          {
19              return  false ;
20          }
21          else
22          {
23              return  true ;
24          }
25      }
```

如上代码所示，传入参数 x 表示目前尝试的答案，i 表示下一个石头的位置，now 表示现在的位置。如果下一个石头和现在的距离差小于尝试答案 x，这块石头可以移走，于是 num 加 1；如果大于答案 x，我们不能移走石头，我们只能走到 i，所以 now 变成了 i。在判断部分，如果最后的移走的石头多于上限，说明答案不可行；反之，可行

2.6.2 二分答案判断部分

```
1   while( left1 <=right1 )
2       {
3           mid=(left1 +right1 ) /2;
4           if (check(mid))
5           {
6               ans=mid;
7               left1 =mid+1;
8           }
9           else
10          {
11              right1 =mid−1;
12          }
```

```
13      }
```

如上代码所示，ans 记录最短跳跃的最远距离，它的范围上限就是第一块石头到最后一块的石头的距离。每次进入 check 函数后，如果目前的尝试可行，那么，目前能跳的距离的最短距离是可以达到的，那么我们更新这个答案，而且，比这个答案小的值没有意义了，哪怕可行也不会更新 ans，所以让处理的下界变成 mid+1，再次进入循环。循环的结束标志是 left 超过 right1 的时候，在相等的时候还可以做一次更新，因为我们采用的更新方式没有注意最后一次两者刚好相遇的情况

2.7 运行结果

代码在洛谷平台上的 OJ 评测如下图2-1, 成功通过

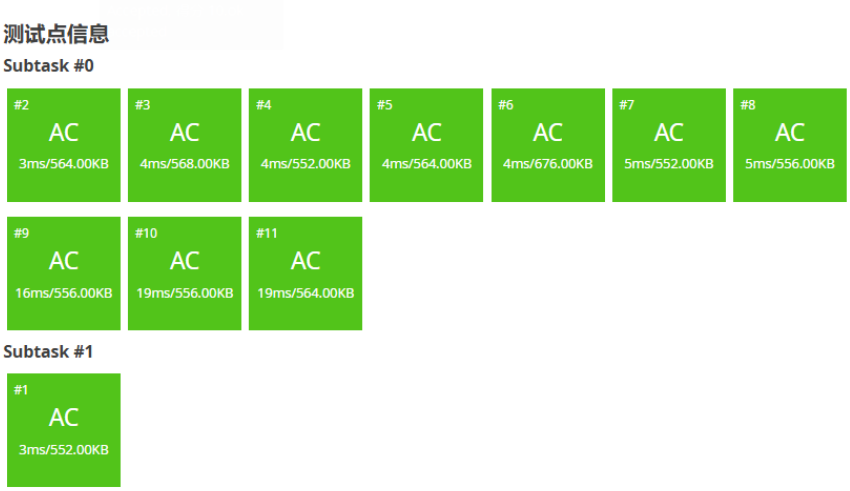

图 2-1 跳石头测试通过

2.8 性能分析

2.8.1 时间复杂度分析

主循环 **二分查找：** 二分查找的范围是 $[0, l]$，所以进行二分查找的次数是 $O(\log l)$，其中 l 是区间的长度。

check 函数 对于每次二分查找中的某个 x，需要调用 `check(x)` 来判断给定最小距离 x 是否可行。`check(x)` 函数中的主循环遍历石头的位置，最多需要 $O(n)$ 的时间（因为最多遍历 n 块石头）。

2.8.2 总时间复杂度

因此，总的时间复杂度是：

$$O(\log l \times n)$$

其中，$\log l$ 是二分查找的次数，n 是石头的数量。

2.8.3 空间复杂度分析

除了输入数据外，程序只使用了一个大小为 $n+2$ 的数组 `stone` 来存储石头的位置，以及几个常量和变量。因此，空间复杂度为：

$$O(n)$$

其中，n 是石头的数量。

2.9 优化思路

1. 优化 check 函数

`check` 函数的核心是一个贪心算法，通过遍历石头位置来判断某个最小距离 x 是否可行。尽管该函数已经是线性的，但可以考虑以下几个方面的改进：

- **提前停止：**如果在 `check` 函数中，`num` 已经超过 m，说明当前尝试的最小距离 x 已经不满足条件，可以立即返回 `false`，而不是继续遍历后续的石头。这样可以减少不必要的计算。
- **更高效的 `stone` 遍历：**如果 `stone` 数组已经排序，可以进一步优化遍历顺序，确保贪心策略的效率。例如，如果 `stone` 数组有序，我们可以在遍历时更高效地判断是否需要移除某些石头。

2. 调整二分查找的范围

在某些情况下，二分查找的范围可以做一定的优化。如果已经知道答案的上下界（例如，最小距离肯定在某个范围内），可以提前确定 `right1` 和 `left1` 的范围，从而减少不必要的查找范围。特别是在某些特定问题中，最小距离的上下界是已知的，提前设置合理的范围会减少计算量。

3. 使用更加高效的排序算法

当前代码假设输入数据已经排好序。如果石头的位置没有排序，可以在 `main` 函数中添加排序步骤。排序的时间复杂度为 $O(n \log n)$，这是一次性的开销。排序后的数据有助于快速判断相邻石头的最小距离。排序可以通过以下代码实现：

```
sort(stone + 1, stone + n + 1);
```

这一步骤会将 `stone` 数组中的位置从小到大排序，从而使得贪心策略更加高效。

4. 考虑用位运算代替除法

在二分查找过程中，`mid = (left1 + right1) / 2` 可能会频繁计算。如果代码特别关注性能，可以考虑使用位运算代替除法。具体而言，可以使用位运算 `mid = (left1 + right1)>>1` 相当于除以 2，并且在某些情况下可以略微加速计算。这是因为位运算的执行速度通常比除法操作要快。

3 贪 心：p2512 合并果子

3.1 问题描述

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n-1$ 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如，有 3 种果子，数目依次为 1、2、9。可以先将 1 和 2 这两堆果子合并，新堆数目为 3，耗费体力为 3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12，耗费体力为 12。所以多多总共耗费体力为：

$$3+12=15$$

可以证明 15 为最小的体力耗费值。

3.2 输入格式

输入格式：

第一行是一个整数 n $(1 \leq n \leq 10000)$，表示果子的种类数。

第二行包含 n 个整数，用空格分隔，第 i 个整数 a_i $(1 \leq a_i \leq 20000)$ 是第 i 种果子的数目。

3.3 输出格式

输出格式：

一个整数，表示最小的体力耗费值。输入数据保证这个值小于 2^{31}。

3.4 输入输出样例

输入	NO.1	NO.2	NO.3
3	1	2	9
输出	15		

表 3-1 输入输出示例

3.5 题目算法分析

由于每次合并的消耗体力等于合并的果子重量和，为了最省力，我们自然可以想到，每次我们合并最轻的两对果子变成一堆新的果子，再在着一些果子里面判断，循环这个过程，直到只有一堆。我们希望通过贪心算法最小化多多在合并果子时的体力消耗。每次合并时，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。为了最小化体力消耗，我们使用优先队列（最大堆）来存储果子堆，每次选择最小的两堆进行合并。下面我们将证明该贪心算法的正确性。

贪心策略的核心是每次选择体力消耗最小的两堆果子进行合并。我们选择最小的两堆进行合并是局部最优选择，因为如果选择较大的堆合并，会导致后续合并时体力消耗增加，进而增加整体体力消耗。通过不断选择局部最优的合并顺序，我们最终能够得到全局最优的解。此外，每次合并的决策是独立的，和之前的合并步骤无关，这使得该算法具有无后效性。因此，贪心算法能够通过每次局部最优选择来保证最小化整体体力消耗，从而得到全局最优解。

3.6 代码分析

这一部分的完整代码请看附页6.2

3.6.1 优先队列处理

```
1
2       int  fin=0;
3       int  n;
```

```
4       cin>>n;
5       for( int  i  =  1;i<=n;i++)
6       {
7           cin>> fruit [ i ];
8           pp.push(− fruit [ i ]) ;
9       }
10      while(pp. size () !=1)  {
11          int  x1  =  pp.top () ;
12          pp.pop() ;
13          int  x2  =  pp.top () ;
14          pp.pop() ;
15          int  sum = (−x1)  + (−x2);
16          fin  += sum;
17          pp.push(−sum);
18      }
```

如上代码所示，pp 为定义的优先队列，确保优先队列每一次的顶部是一个最值，方便我们的 pop 的实现。注意，优先队列的顶部是最大值，但是我们要最小值，所以我们选择 push 当前索要的体力的相反数入队，在 pop 的时候还原处理，每一次合并，都要把新生成的果子入队。处理结束的标志是队列里面只有一堆果子，这个时候，fin 每次都会加上合并的体力，它就是最终答案

3.7 运行结果

代码在洛谷平台上的 OJ 评测如下图3-1, 成功通过

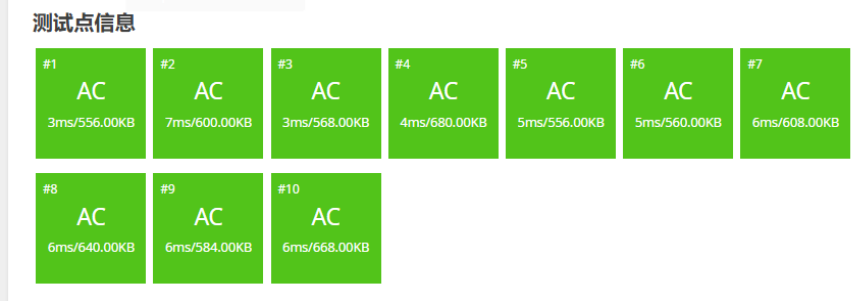

图 3-1　合并果子测试通过

3.8 性能分析

时间复杂度分析

假设输入中有 n 种果子。

1. 优先队列的初始化：- 首先，我们需要将每种果子的数量插入到优先队列中。由于每次插入操作的时间复杂度为 $O(\log k)$，其中 k 是当前队列中元素的个数，因此对于 n 个果子，插入操作的总时间复杂度为 $O(n \log n)$。

2. 合并过程：- 在合并的过程中，每次我们从堆中取出两个最小的堆，进行合并操作并将新的堆放回优先队列。每次合并的操作是 $O(\log n)$，因为每次都需要插入和删除堆元素。- 由于总共进行 $n-1$ 次合并（每次合并两个堆，直到最后剩下一个堆），因此合并过程的总时间复杂度为 $O(n \log n)$。

综上所述，整个算法的时间复杂度为：

$$O(n \log n)$$

这包括了初始化优先队列和进行合并操作的时间。

空间复杂度分析

1. 优先队列：- 我们使用一个优先队列来存储每个堆的果子数量，最多需要存储 n 个元素。因此，优先队列的空间复杂度为 $O(n)$。

2. 果子数量的存储：- 我们还使用一个大小为 n 的数组来存储每种果子的数量，空间复杂度为 $O(n)$。

因此，总的空间复杂度为：

$$O(n)$$

总结：- 时间复杂度：$O(n \log n)$ - 空间复杂度：$O(n)$

3.9 优化思路

1. 考虑带权并查集（Union-Find）

在某些情况下（虽然这个问题本身不涉及），带权并查集（Union-Find）也能优化某些类型的合并操作，尤其是当我们需要动态处理合并的情况。但对于当前问题，这种方法的引入反而可能使得代码复杂化，不一定带来明显的优化。

2. 不使用优先队列

另一种可能的优化方向是，避免使用优先队列，转而使用一个简单的排序过程来合并堆。

首先，排序所有的果子数量，然后每次选择最小的两个堆进行合并。这种方法的时间复杂度是 $O(n \log n)$，虽然看起来和使用优先队列的复杂度一样，但实现上可以有所简化。具体来说，我们可以通过在每次合并后将新堆插入已排序数组来模拟合并过程。

3. 减少不必要的操作

优先队列的负数取反：目前，代码通过将堆中的每个数取负来模拟最小堆的行为，这虽然没有问题，但使用了额外的负数取反操作。如果能够直接使用最小堆，可能会稍微提高可读性和效率。不过在 C++ 中，标准库没有提供最小堆的直接实现，通常通过将数值取负来模拟最小堆。

空间优化：每次读取果子的数量并插入优先队列时，空间复杂度是 $O(n)$，这里没有问题，因为空间使用已经较为高效。然而，如果输入数据非常大，可以考虑如何避免过多的临时存储（例如，在一些特殊情况下直接使用流式输入/输出）。

4. 优化数据结构

在极端情况下，可能会考虑使用其他数据结构来替代优先队列。例如，平衡二叉搜索树（如 std::set 或 std::multiset）也能实现合并操作的快速插入和删除，但优先队列的操作（尤其是 push 和 pop）通常在时间复杂度上更为优化，尤其是当我们不关心中间元素的顺序时。

4 搜 索：p1443 马的遍历

4.1 问题描述

有一个 $n \times m$ 的棋盘，在某个点 (x, y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。

输入格式

输入只有一行四个整数，分别为 n, m, x, y。

输出格式

一个 $n \times m$ 的矩阵，代表马到达某个点最少要走几步（不能到达则输出 -1）。

4.2 输入输出示例

输入示例

$$\begin{bmatrix} 3 & 3 \\ 1 & 1 \end{bmatrix}$$

输出示例

$$\begin{bmatrix} 0 & 3 & 2 \\ 3 & -1 & 1 \\ 2 & 1 & 4 \end{bmatrix}$$

4.3 题目算法分析

由于题目问的是到达某一个点的最少步数，我们从一个点出发，如果用 dfs，还需要去更新每个点的深度。但是如果我们用 bfs，我们每一次到达的点被一次性更新完全，下一次更新的点的数值一定会比上一次的大从而被舍弃，所以我们

选取 bfs。为了让二维图上的两个点始终要统一，用 pair 实现两者同时传递，用 queue 实现 bfs

4.4 代码分析

这一部分的完整代码请看附页6.2

4.4.1 初始化

```
int dx [8]={−1,−2,−2,−1,1,2,2,1};
int dy [8]={2,1,−1,−2,2,1,−1,−2};
queue<pair<int , int >>myque;
```

代码如上，我们定义 dx 和 dy 数组对应每一次移动 x 坐标和 y 坐标的同时变化，queue 队列元素为 pair，为坐标对。

4.4.2 bfs 实现

```
step [x_now][y_now]=0;
vis [x_now][y_now]=1;
myque.push(make_pair(x_now,y_now));
while (! myque.empty())
{
    int x1=myque.front(). first ;
    int y1 = myque.front() .second;
    myque.pop();
    for ( int i = 0; i<8;i++)
    {
        int x2 = x1+dx[i ];
        int y2 = y1 + dy[i ];
        if (x2<1||x2>x||y2<1||y2>y|| vis [x2][y2])
        {
            continue ;
```

```
16                  }
17                  vis[x2][y2]=1;
18                  step[x2][y2]=step[x1][y1]+1;
19                  myque.push(make_pair(x2,y2));
20              }
21          }
22          for( int  i=1;i<=x;i++)
23          {
24              for( int  j=1;j<=y;j++){
25                  printf("%-5d",step[i][j]);
26              }
27              printf("\n");
28          }
```

代码如上所示，我们用 step 数组记录步数，用 vis 数组记录是否已经到达过。因为，如果没有 vis 数组，可能会进入死循环。而且，由于第一次到达的时候就是最优解，即使后面有很多种方法到达某一个块我们都不能取。所以，加入 vis 数组判断是否到达。把目前到达，且 vis 还是 0 的块入队，同时更新 step 和 vis 变为 1，不断入队出队直到没有

4.5 运行结果

代码在洛谷平台上的 OJ 评测如下图4-1, 成功通过

图 4-1　马的遍历测试通过

4.6 性能分析

4.6.1 时间复杂度

广度优先搜索（BFS）算法的时间复杂度主要取决于棋盘的大小和每个位置的处理情况。在最坏的情况下，算法需要遍历棋盘上的所有位置，且每个位置最多会被访问一次。对于每个位置，最多需要尝试 8 个方向的移动。因此，时间复杂度为：

$$O(n \times m)$$

其中，n 和 m 分别是棋盘的行数和列数。最坏情况下，所有位置都需要被访问，因此时间复杂度为 $O(n \times m)$。

4.6.2 空间复杂度

空间复杂度主要来自于存储棋盘的状态以及 BFS 使用的队列。我们需要一个大小为 $n \times m$ 的 ‘step‘ 数组来记录每个位置的最短步数，另一个大小为 $n \times m$ 的 ‘vis‘ 数组来标记位置是否已访问。队列最多存储 $n \times m$ 个元素。因此，空间复杂度为：

$$O(n \times m)$$

其中，n 和 m 是棋盘的行数和列数。由于在任何时刻队列最多保存所有位置，空间复杂度为 $O(n \times m)$。

4.7 优化思路

1. 使用更高效的队列实现：当前算法使用的是 ‘queue<pair<int, int»‘，它在每次访问时只处理了一个位置。如果可以考虑使用双端队列（‘deque‘）替代普通队列，可能会在一些场景下提升队列操作的效率，但在这个问题中，使用标准队列已经足够高效。

2. 优化空间利用：当前算法使用了两个大小为 $n \times m$ 的数组 ‘step‘ 和 ‘vis‘ 来存储每个位置的步数和访问状态。考虑到很多位置的步数可能已经被计算出来，可以尝试通过修改 ‘step‘ 数组，减少额外的 ‘vis‘ 数组的使用。例如，可以直

接将 ‘step‘ 数组中的负值作为未访问的标志，进一步节省空间。

3. 剪枝优化：虽然广度优先搜索本身已经是最短路径的理想选择，但在某些情况下，如果问题输入允许，可以通过提前计算一些条件，减少无意义的扩展。例如，如果棋盘的边界或者某些特殊位置不再影响结果，可以提前做一些判断，避免无效的搜索。

5 动态规划：p1220 关路灯

5.1 问题描述

某一村庄在一条路线上安装了 n 盏路灯，每盏灯的功率有大有小（即同一段时间内消耗的电量有多有少）。老张就住在这条路中间某一路灯旁，他有一项工作就是每天早上天亮时一盏一盏地关掉这些路灯。

为了给村里节省电费，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。开始他以为先算一下左边路灯的总功率再算一下右边路灯的总功率，然后选择先关掉功率大的一边，再回过头来关掉另一边的路灯，而事实并非如此，因为在关的过程中适当地调头有可能会更省一些。

现在已知老张走的速度为 1 m/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（单位：W），老张关灯所用的时间很短而可以忽略不计。

请你为老张编一程序来安排关灯的顺序，使从老张开始关灯时刻算起所有灯消耗电最少（灯关掉后便不再消耗电了）。

5.2 输入输出示例

5.2.1 输入格式

输入的第一行包含一个整数 n（$1 \leq n \leq 10^5$），表示路灯的数量。

接下来的 n 行，每行包含两个整数：p_i 和 w_i（$1 \leq p_i \leq 10^5, 1 \leq w_i \leq 1000$），表示第 i 盏路灯的位置 p_i 和功率 w_i。

5.2.2 输出格式

输出一个整数，表示最节省电的总电量消耗。

5.3 题目算法分析

这个题目是经典区间 dp 题目，这个题目我一开始的思路是把状态定为在 i 处结束的用量但是这个状态转移无法撰写，因为它可以从 2 个方向到达 i。所以应该是区间的 dp，这样子的话，就可以定义结束的状态了。如果以单一的某一点结束，你怎么确定在那个点结束呢？所以状态 i 和 j 就是 i 到 j 区间上结束的用值那么又有一个问题怎么确定站在哪呢？我们看到，为了节约时间，实际上每一个经过的点都会灭掉，所以最后要么在 i 要么在 j，左端点定义为 0 右端点是 1 就行。

区间 dp 的一个要点就是怎么写状态，怎么循环。由于一个大区间的最后状态实际上依托于每一个小区间的状态，所以我们可以按照区间长度步长去循环，这事实上也是区间 dp 的经典方法。还有一个细节，由于我们所有的灯都是在刚刚开始一起还在运作，而在结束的时候消耗的是还在运作的所有的灯消耗的能力，所以我们要获得所有的没关的灯在跑的时间内的 w 的和。这里我们可以用 sum 前缀和去表示。

5.4 代码分析

这一部分的完整代码请看附页6.2

5.4.1 前缀和实现和逻辑

```
1        for( int  i  =  1; i<=n;i++)
2    {
3            cin>>pos[i]>>w[i];
4            sum[i]=sum[i−1]+w[i]; // 利用前缀和可以快速得到其他部分功
                 率消耗总和
5    }
```

前缀和的优势在于，他可以快速获得某一个区间的区间和，这样做的话，题目中有关某一个区间的数量和就可以用一次减法获得。那么，就可以在这个基础上获得总值

5.4.2 端点动态规划

```
1        for(int l = 2;l<=n;l++)
2    {
3        for(int i = 1;i+l-1<=n;i++)
4        {
5            //初始化起点循环
6            int j = i+l-1;
7            dp[i][j][0]=min(dp[i+1][j][0]+(pos[i+1]-pos[i])*(sum[i]+
                 sum[n]-sum[j]),dp[i+1][j][1]+(pos[j]-pos[i])*(sum[i]+
                 sum[n]-sum[j]));
8            dp[i][j][1]=min(dp[i][j-1][0]+(pos[j]-pos[i])*(sum[i-1]+
                 sum[n]-sum[j-1]),dp[i][j-1][1]+(pos[j]-pos[j-1])*(sum[
                 i-1]+sum[n]-sum[j-1]));
9        }
10   }
```

这里循环的逻辑在于，第一层循环是步长，从 2 开始，第二层循环是左端点。第三维表明是在左端点还是右端点完成一个区间的关灯的所有任务。0 表示左端点，1 表示右端点。在 i 到 j 区间内，如果完成 i 到 j 的关灯是在 i，那么他一定完成了 i+1 到 j 的任务，就看是从哪个端点过来的可以达到最小。同理，如果完成 i 到 j 的关灯是在 j，那么他一定完成了 i 到 j-1 的任务，两者中结合前缀和取最小值即可。

5.5 运行结果

代码在洛谷平台上的 OJ 评测如下图5-1, 成功通过

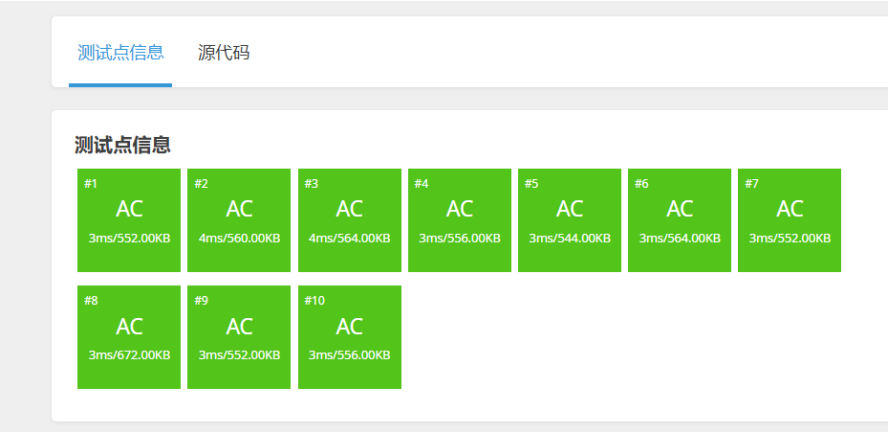

图 5-1　关路灯测试通过

5.6　性能分析

5.6.1　时间复杂度

该程序的时间复杂度主要由动态规划的过程决定。动态规划的核心是填充 dp 数组。外层循环控制区间长度，从 $l = 2$ 到 n，共进行 $O(n)$ 次迭代。对于每个区间 $[i, j]$，内层循环遍历所有可能的区间，最多进行 $O(n)$ 次操作。每次状态转移需要常数时间计算。

因此，时间复杂度为：

$$O(n^2)$$

其中，n 是问题规模，即位置的数量。最坏情况下，动态规划算法需要填充一个大小为 $O(n \times n)$ 的三维状态数组，因此总体时间复杂度为 $O(n^2)$。

5.6.2　空间复杂度

空间复杂度主要由以下几个部分组成：

- 数组 `pos` 和 `w`：分别存储位置和功率消耗，大小为 $O(n)$。
- 数组 `dp`：这是一个三维数组，维度为 $O(n \times n \times 2)$，用于存储区间的状态。其空间复杂度为 $O(n^2)$。
- 数组 `sum`：用于存储前缀和，大小为 $O(n)$。

因此，总体空间复杂度为：

$$O(n^2)$$

这是因为 dp 数组的大小为 $O(n^2)$，其他数组的空间复杂度均为 $O(n)$。总体空间复杂度由 dp 数组主导，为 $O(n^2)$。

5.7 算法优化思路

1. 状态压缩优化：当前程序使用三维数组 `dp[i][j][0/1]` 来表示区间 $[i,j]$ 的最小功率消耗，且每次状态转移时需要访问大量的历史状态。实际上，对于每个区间，状态转移只依赖于其邻近的子区间，因此可以考虑将二维动态规划转化为一维滚动数组。即仅保留当前区间的状态，避免存储完整的二维数组，从而大幅降低空间复杂度。

2. 提前计算和记忆化：在当前算法中，前缀和数组 sum 每次更新时会重新计算区间和，导致冗余的计算。如果能够通过记忆化（Memoization）技术，在求解过程中保存已经计算过的前缀和，避免重复计算，可以有效减少计算时间，提升效率。具体而言，可以使用哈希表或一维数组缓存每个区间的前缀和，减少每次计算的复杂度。

3. 优化状态转移顺序：目前的动态规划算法以固定的顺序处理所有区间，但可能存在某些区间在早期阶段就能得出最优解。因此，可以考虑使用贪心策略或者启发式方法，在某些情况下调整状态转移的顺序。例如，优先处理那些可能更早达到最优解的区间，或者通过某种规则（如区间长度）来安排转移顺序，从而减少不必要的状态计算。

4. 基于动态规划的分治优化：当前的动态规划方法直接计算每个区间的最优解，可以考虑通过分治法（Divide and Conquer）优化。具体来说，可以将问题分解为多个子问题，并通过递归或分治的方式，合并子问题的解。例如，在解决 $dp[i][j]$ 的时候，考虑将区间 $[i,j]$ 分为两个部分，通过合并这些部分的最优解来减少计算量。

5. 减少区间选择中的冗余计算：由于当前算法每次都计算所有区间的最优解，导致大量的冗余计算。如果可以通过某种规则提前排除一些明显不可能的区间，或者通过动态剪枝减少不必要的计算，可能会进一步提高效率。例如，若某个区间的功率消耗已经大于当前最优解，则可以提前结束该区间的状态计算，从而减少不必要的遍历。

6. 结合启发式搜索优化：对于动态规划问题，如果启发式搜索能够提供有效的界限，可以考虑结合启发式算法（如 A*）来加速搜索过程。通过引入启发

式函数（例如估计当前区间的最小功率消耗），可以在搜索过程中优先扩展最有可能达到最优解的区间，从而减少不必要的状态转移。

6 总结

6.1 实验总结

在做题中我有以下的收获：

- **动态规划的关键在于确定状态和状态转移方程**。状态是问题的关键属性，它的变化会影响问题的解。通过定义合适的状态和状态转移方程，我们可以将问题转化为子问题的求解，并利用已知的子问题解来推导出更大规模问题的解。这种自底向上的求解方法能够有效地降低时间复杂度。
- **状态压缩是一种将问题中的状态信息用更小的空间表示的技巧**。通过位运算和位操作，我们可以将原始数据的状态信息压缩成更紧凑的形式，从而减少内存占用和提高算法效率。状态压缩特别适用于地图类问题，其中每个坐标点的可能状态数较少。在状态压缩中，我们使用二进制位来表示状态。对于每个状态，我们可以使用一个比特位来表示它的取值，例如只有黑或者白两种状态的单元位置，我们可以使用一个比特位来表示它的状态。如果状态的取值更多，我们可以使用更多的比特位来表示。
- **深入理解了 A* 算法原理**。 A* 算法结合了广度优先搜索和启发式搜索的优点，通过合理的估计函数和搜索策略来找到最佳路径。通过深入理解算法原理，我能够更好地应用和调整算法，以适应不同问题的求解。
- **在求解问题时遵循问题分析到结合理论贴合实际情况设计算法**。在设计算法的时候，依据题目分析的内容，在草稿纸上列出一个简单的数据测试过程可以更快地找到算法的设计要点以及思路。

6.2 心得体会和建议

通过算法课程的学习，我深刻认识到算法对于实际问题求解的重要性。算法不仅决定了问题的机械求解步骤，还直接影响到时间和空间的开销。在时间和空间之间总会有一方牺牲，这是一种博弈关系。我们需要根据具体应用场景，选择适当的算法来实现我们的目标。有时我们会选择时间换取更高的空间效率，有时则会选择空间换取更快的时间执行。这种权衡取舍是为了达到最佳的问题解决效果。

然而，我也认识到学习算法的道路还很长。目前的算法课程内容虽然涵盖了

一些经典基础的算法，但课时有限，无法深入探讨各种算法的细节和应用。我希望未来的算法教学能够增加更多的内容，降低考核难度，让学生有更多的机会深入理解和应用算法。

总之，学习新的算法让我开拓了逻辑思维，它要求我们敏锐地把握问题的本质，进行数学建模，并选择最合适的算法来解决问题。同时，我也认识到算法对于问题求解的重要性，以及时间和空间之间的权衡取舍。希望未来的算法教学能够增加算法的内容，让学生有更多机会深入学习和应用算法。

分 治：P2678 跳石头（完整代码）

```
#include <bits/stdc++.h>
using namespace std;

const int maxn = 50005;  // 定义石头的最大数量
int stone[maxn];         // 存储石头的位置
int ans;                 // 存储最终的答案（最大最小距离）
int l, n, m;             // l: 起点到终点的距离，n: 石头数量，m: 最多可以移走的石头数
int right1 ;             // 二分查找的右边界
int left1 ;              // 二分查找的左边界
int mid;                 // 当前二分查找的中点

// check函数用来验证当前尝试的最大最小距离x是否满足条件
// 如果能够以x为最小距离放置石头，并且移除的石头数不超过m，返回true，否则返回false
bool check(int x)
{
    int num = 0;         // 记录移除的石头数量
    int i = 0;           // 用来遍历石头数组的索引
    int now = 0;         // 当前放置石头的位置（初始化为0）

    // 遍历所有石头位置，判断能否满足当前的最小距离x
    while (i <= n)
    {
        i++;  // 遍历石头数组
        if (stone[i] - stone[now] < x)  // 如果当前石头与上一个放置的石头距离小于x
        {
            num++; // 需要移除当前的石头
```

```
27              }
28              else
29              {
30                  now = i;   // 当前石头可以放置，更新“上一个放置的石
                        头”的位置
31              }
32          }
33
34          // 如果移除的石头数超过了m，返回false，否则返回true
35          if (num > m)
36          {
37              return false;
38          }
39          else
40          {
41              return true;
42          }
43      }
44
45      int main()
46      {
47          // 输入起点到终点的距离，石头数量，最多可移走的石头数
48          cin >> l >> n >> m;
49
50          // 输入每个石头的位置
51          for (int i = 1; i <= n; i++)
52          {
53              cin >> stone[i];
54          }
55
56          // 添加起点和终点的石头位置
```

```
57          stone[n + 1] = 1;    // 终点的位置
58          stone[0] = 0;        // 起点的位置
59
60          right1 = 1;          // 初始化二分查找的右边界为终点的距离
61          left1 = 0;           // 初始化二分查找的左边界为起点的位置
62
63          // 二分查找，寻找能够满足的最大最小距离
64          while ( left1 <= right1 )
65          {
66              mid = ( left1 + right1 ) / 2;   // 计算中点
67
68              // 如果当前最小距离mid满足条件，则尝试增大最小距离
69              if (check(mid))
70              {
71                  ans = mid;   // 更新答案为当前的最小距离
72                  left1 = mid + 1;   // 尝试更大的最小距离
73              }
74              else
75              {
76                  right1 = mid - 1;   // 否则尝试更小的最小距离
77              }
78          }
79
80          // 输出最终的最大最小距离
81          cout << ans << endl;
82
83          return 0;
84      }
```

贪 心：P1090 合并果子（完整代码）

```cpp
#include<bits/stdc++.h>  // 引入标准库，包含所有常用的头文件
using namespace std;  // 使用标准命名空间

int fruit[10005];  // 存储每种果子的数量
priority_queue<int> pp;  // 优先队列（最大堆），用来存储果子的数量，最大堆的性质是根节点是最大的数

int main() {
    int fin = 0;  // 用于记录总的体力消耗
    int n;  // 存储果子的种类数
    cin >> n;  // 输入果子的种类数

    // 读取每种果子的数量，并将其放入最大堆（priority_queue）
    for (int i = 1; i <= n; i++) {
        cin >> fruit[i];  // 输入第 i 种果子的数量
        pp.push(-fruit[i]);  // 将果子的数量加入最大堆，为了使用最小堆，我们将数字取负数
    }

    // 当堆中只剩一个元素时，停止合并
    while (pp.size() != 1) {
        // 取出堆中最大的两个数（由于是最大堆，取出的是最小的负数，所以取负数得到最小的两个数）
        int x1 = pp.top();  // 取出最大值
        pp.pop();  // 移除堆顶元素
        int x2 = pp.top();  // 取出第二大的值
        pp.pop();  // 移除堆顶元素

        // 合并这两个堆，新的堆的大小为它们的和，计算体力消耗
```

```
27              int  sum = (−x1) + (−x2);   // 体力消耗是这两个堆的总重量
28              fin  += sum;   // 累加体力消耗
29
30              // 将新的堆重新加入堆中，堆的大小是它们的和
31              pp.push(−sum);   // 再次将其取负，放入最大堆中
32          }
33
34          cout << fin;   // 输出总的体力消耗
35      }
```

搜 索：P1443 马的遍历（完整代码）

```cpp
#include<bits/stdc++.h>
using namespace std;

// 定义马的八个可能的移动方向
int dx[8] = {-1, -2, -2, -1, 1, 2, 2, 1};   // 水平方向上的偏移量
int dy[8] = {2, 1, -1, -2, 2, 1, -1, -2};   // 竖直方向上的偏移量

// 创建一个队列来进行广度优先搜索（BFS）
queue<pair<int, int>> myque;

// 定义棋盘的最大大小
const int maxn = 505;

// 用来记录每个位置的最短步数
int step[maxn][maxn];

// 用来标记某个位置是否已被访问
bool vis[maxn][maxn];

int main()
{
    // 初始化step数组，-1表示该位置尚未访问
    memset(step, -1, sizeof(step));

    int x, y;   // 棋盘的大小
    int x_now, y_now; // 马的起始位置

    // 读取棋盘大小和马的起始位置
    cin >> x >> y >> x_now >> y_now;
```

```
30
31      // 设置起始位置的步数为0，并标记为已访问
32      step[x_now][y_now] = 0;
33      vis[x_now][y_now] = 1;
34
35      // 将起始位置加入队列
36      myque.push(make_pair(x_now, y_now));
37
38      // 广度优先搜索（BFS）
39      while(!myque.empty())
40      {
41          // 取出队列中的第一个位置
42          int x1 = myque.front().first;
43          int y1 = myque.front().second;
44          myque.pop();
45
46          // 尝试8个方向的移动
47          for(int i = 0; i < 8; i++)
48          {
49              // 计算新的位置
50              int x2 = x1 + dx[i];
51              int y2 = y1 + dy[i];
52
53              // 检查新的位置是否越界或已被访问
54              if(x2 < 1 || x2 > x || y2 < 1 || y2 > y || vis[x2][y2])
55              {
56                  continue;  // 如果无效则跳过
57              }
58
59              // 标记该位置为已访问，并更新步数
60              vis[x2][y2] = 1;
```

```
61                  step[x2][y2] = step[x1][y1] + 1;
62
63                  // 将新的位置加入队列
64                  myque.push(make_pair(x2, y2));
65              }
66          }
67
68          // 输出最终的步数矩阵
69          for(int i = 1; i <= x; i++)
70          {
71              for(int j = 1; j <= y; j++)
72              {
73                  // 每个数字输出宽度为5，右对齐
74                  printf("%-5d", step[i][j]);
75              }
76              // 每行输出完毕后换行
77              printf("\n");
78          }
79      }
```

动态规划：P1220 关路灯（完整代码）

```cpp
#include<bits/stdc++.h>
using namespace std;

const int maxn = 60;  // 最大问题规模
int pos[maxn];        // 存储位置
int w[maxn];          // 存储每个位置的功率消耗
int dp[maxn][maxn][2]; // 动态规划状态数组，dp[i][j][0/1]表示处理区间[i, j]时的最小功率消耗
int sum[maxn];        // 前缀和数组，用于快速计算功率消耗

int main()
{
    int n, c;
    cin >> n >> c;  // 输入n（区域数）和c（当前起始位置）

    // 初始化dp数组，dp数组的值初始化为一个大的数，代表未计算状态
    memset(dp, 20000, sizeof(dp));

    // 输入位置和功率，并计算前缀和
    for (int i = 1; i <= n; i++)
    {
        cin >> pos[i] >> w[i];  // 输入每个位置和功率
        sum[i] = sum[i - 1] + w[i]; // 计算前缀和
    }

    // 初始化起始位置的dp值，c表示起始位置
    dp[c][c][0] = 0;
    dp[c][c][1] = 0;
```

```
28
29      // 动态规划填表过程
30      for (int l = 2; l <= n; l++)  // l表示区间长度，范围从2到n
31      {
32          for (int i = 1; i + l - 1 <= n; i++)  // 遍历所有的区间
33          {
34              int j = i + l - 1;  // 计算区间的终点位置
35
36              // 状态转移方程：选择区间[i, j]时的最小功率消耗
37              // dp[i][j][0]表示从i开始到j结束时，最后一次操作是在位置i
38              // dp[i][j][1]表示从i开始到j结束时，最后一次操作是在位置j
39
40              dp[i][j][0] = min(
41                  dp[i + 1][j][0] + (pos[i + 1] - pos[i]) * (sum[i] + sum[n] - sum[j]),  // 从i+1到j的最小功率消耗
42                  dp[i + 1][j][1] + (pos[j] - pos[i]) * (sum[i] + sum[n] - sum[j])  // 从j到i+1的最小功率消耗
43              );
44
45              dp[i][j][1] = min(
46                  dp[i][j - 1][0] + (pos[j] - pos[i]) * (sum[i - 1] + sum[n] - sum[j - 1]),  // 从i到j-1的最小功率消耗
47                  dp[i][j - 1][1] + (pos[j] - pos[j - 1]) * (sum[i - 1] + sum[n] - sum[j - 1])  // 从j-1到j的最小功率消耗
48              );
49          }
50      }
51
```

```
52        // 输出最终的最小功率消耗，选择从1到n区间内的最优结果
53        cout << min(dp[1][n ][0], dp[1][n ][1]) ;
54        return 0;
55    }
```