浙江大學

本科生课程报告

学年、学期：2021 — 2022 学年 秋冬 学期

课程名称：数据分析与算法设计

任课教师：李东晓

题　　目：自选编程作业 3

学生姓名：黄嘉欣

学　　号：3190102060

数据分析与算法设计：自选编程作业 3

3190102060 黄嘉欣 信工 1903 班

一、题目：887.鸡蛋掉落(难度：困难)

给你 k(1 <= k <= 100)枚相同的鸡蛋，并可以使用一栋从第 1 层到第 n(1 <= n <= 10^4)层共有 n 层楼的建筑。已知存在楼层 f，满足 0 <= f <= n，任何从高于 f 的楼层落下的鸡蛋都会碎，从 f 楼层或比它低的楼层落下的鸡蛋都不会破。每次操作，你可以取一枚没有碎的鸡蛋并把它从任一楼层 x 扔下(满足 1 <= x <= n)。如果鸡蛋碎了，你就不能再次使用它；如果某枚鸡蛋扔下后没有摔碎，则可以在之后的操作中重复使用这枚鸡蛋。请你计算并返回要确定 f 确切的值的最小操作次数是多少？

二、算法设计：动态规划

由题意，我们需要求出在 n 层楼中使用 k 枚鸡蛋确定楼层 f 所需的最小操作次数，则必须将所有的可能情况一一列举，从中找出最小值。面对此穷举问题，可以考虑动态规划算法，根据状态转移方程推导状态，利用空间换取时间，巧妙地解决问题。其具体思路分析如下：

根据题目要求，当我们身处第 x 层楼时扔鸡蛋时，鸡蛋可能会被摔碎，说明楼层 f 比当前楼层低，需要下楼；若鸡蛋未被摔碎，则楼层 f 比当前楼层高，需要上楼。显然，在下楼或上楼的同时，我们可以将问题的规模进行改变，故从此处入手寻找递推关系式。设 F(i,j)表示在拥有 i 枚鸡蛋，操作 j 次情况下所能验证的最大楼层数，根据前述逻辑，当鸡蛋在某层楼扔下后，若其被摔碎，则需下楼继续测试，此时我们还拥有 i-1 枚鸡蛋，可操作 j-1 次(因为已操作 1 次)，故可以验证楼下的最大楼层数为 F(i-1,j-1)；若鸡蛋没碎，则需要上楼，此时这枚鸡蛋可以在之后的操作中重复使用，即我们拥有 i 枚鸡蛋，可操作 j-1 次，故可以验证楼上的最大楼层数为 F(i,j-1)，加上此次实验可以验证当前楼层是否为 f 层(注意：鸡蛋是否被摔碎是不确定的)，故最终能够验证的最大楼层数有：

$$F(i,j)=F(i-1,j-1)+F(i,j-1)+1$$

考虑初始情况，若只有 0 枚鸡蛋，则无论操作多少次，能够验证的最大楼层数始终为 0，即 F(0,j)=0；若只能操作 0 次，则无论有多少枚鸡蛋，能够确定的也始终是 0 层，即 F(i,0)=0。综上，完整的递推关系为：

$$\begin{cases} F(i,j) = F(i-1,j-1) + F(i,j-1) + 1, 0 \le i \le k, 0 \le j \le n \\ F(i,0) = 0, 0 \le i \le k \\ F(0,j) = 0, 0 \le j \le n \end{cases}$$

由于我们需要求解的是给定 k 枚鸡蛋条件下，在 n 层楼中能够确定 f 的最小操作次数，即满足 F(k,j)≥n 的最小 j，故在算法中，我们可将 j 进行累加，用 F(k,j)≥n 作为循环终止的条件。除此之外，为了在计算 F(i,j)之前确保 F(i-1,j-1)和 F(i,j-1)的值已被算出，我们需要通过按行从上往下或者按列从左往右的遍历方式逐行填充二维数组。考虑到对 j 的限制，我们采用按列从左往右的填充方法，从而只需要填充表的一部分，提高时间效率。此算法的伪代码为：

算法：superEggDrop(int k, int n)

```
// 求解确定 f 所需的最小扔鸡蛋次数
// 输入：鸡蛋数 k，楼层数 n
// 输出：最小操作次数 j
// 注意：F(,)为(k+1)*(n+1)的全 0 数组，不再额外定义
for i <- 0 to k do
    F(i,0) <- 0
for j <- 0 to n do
    F(0,j) <- 0 // 初始条件
j <- 0 // 操作次数
// 找到最小的 j 即可，不需要将整张表填完
while F(k,j) < n do
    j <- j+1 // 第 0 列已全部初始化为 0
    for i <- 1 to k do // 注意不要下标越界
        F(i,j) <- F(i-1,j-1)+F(i,j-1)+1
return j // 最小操作次数
```

三、代码实现

根据(二)中伪代码，具体化各步骤，可得 C 语言代码实现为：

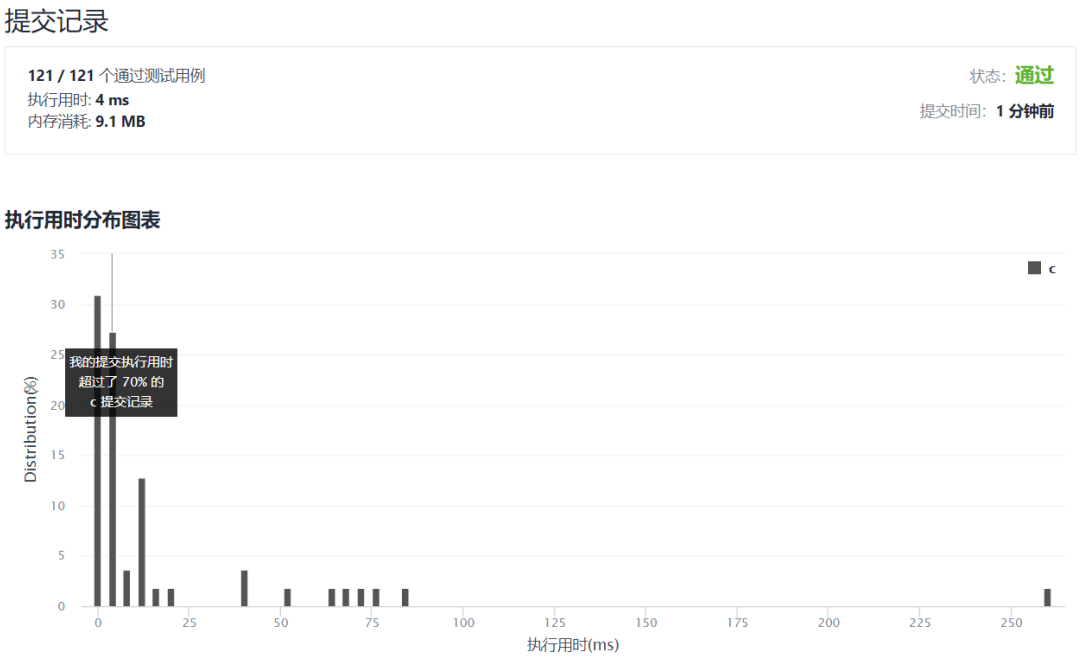
代码：int superEggDrop(int k, int n)

```
{
    int F[k+1][n+1]; // 最大楼层数矩阵
    int i, j;
    for (i=0;i<k+1;i++){
        for (j=0;j<n+1;j++){
            F[i][j] = 0; // 二维数组初始化
        }
    }
    for (i=0;i<k+1;i++){ // 初始条件
        F[i][0] = 0;
    }
```

```
        for (j=0;j<n+1;j++){
            F[0][j] = 0;
        }
        j = 0; // 操作次数
        while (F[k][j]<n){
            j++; // 注意不要下标越界
            for (i=1;i<k+1;i++){
                F[i][j] = F[i-1][j-1]+F[i][j-1]+1;
            }
        }
        return j;
    }
```

四、运行结果

如图 4.1，将动态规划算法代码提交至 LeetCode，得其执行用时为 4ms，内存消耗 9.1MB，通过全部 121 个测试用例。当测试输入为{2　6}时，程序输出为 3，与正确结果一致，如图 4.2 所示。采用自测试实例，若输入的数据为{1　2}，则输出为 2，因为此时：如果从 1 楼扔下鸡蛋，若它碎了，则 f=0；否则，使鸡蛋从 2 楼掉落，若它碎了，则 f=1，若没碎，则 f=2。因此，我们最少需要操作 2 次才能得到确切的楼层 f 值，与程序输出一致；若输入数据为{3　14}，得输出为 4，与理论分析吻合，如图 4.3 所示。对于更多的输入可能，无法一一列举，但由 LeetCode 测试结果可知，算法设计正确。



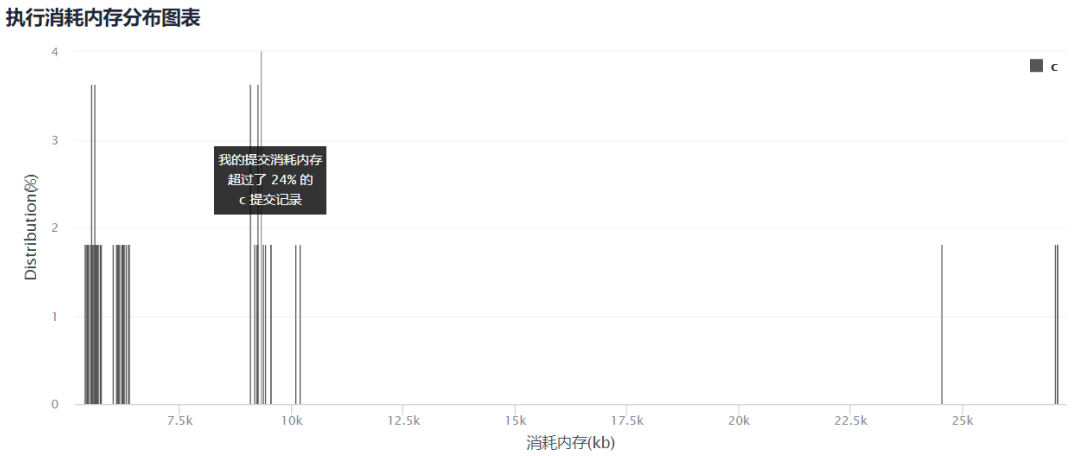


图 4.1 LeetCode 算法运行结果

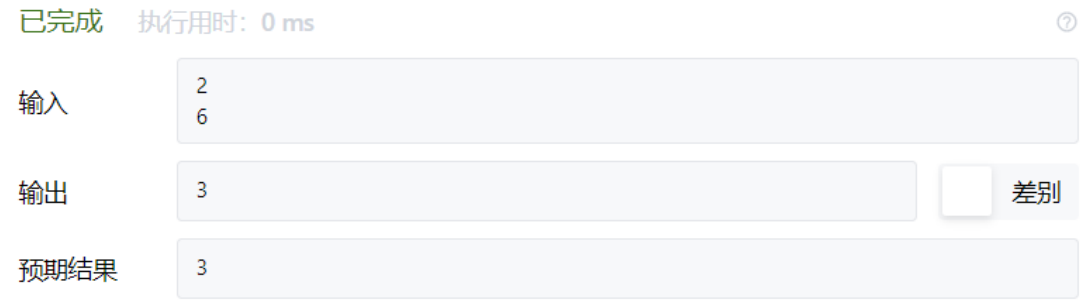


图 4.2 LeetCode 测试实例

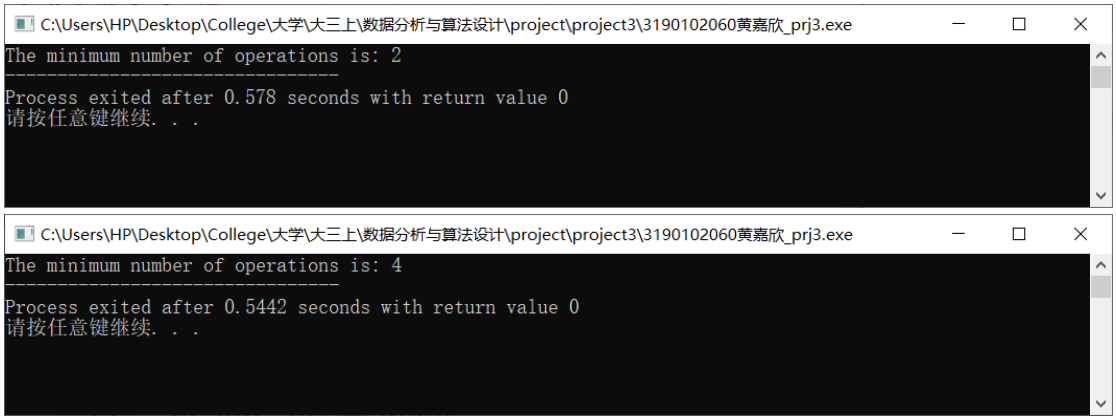


图 4.3 自测试实例

五、效率分析

由代码，设对输入 k，n，算法的基本操作(赋值)执行次数为 C(n)，则在最差情况，即当 j=n 时才有 F[k][j]==n 的条件下，有：

$$C(n)=\sum_{j=0}^{n}\sum_{i=1}^{k+1}1=\sum_{i=0}^{n}(k+1)=(n+1)(k+1)\in O(kn)$$

即动态规划算法的最差时间效率为 O(kn)。

显然，由于在动态规划算法中，我们只需要对最大楼层数矩阵 F[k+1][n+1]进行填充，故所需的额外空间为：V(n)=(k+1)(n+1)∈O(kn)，即算法的空间效率为 O(kn)。

六、总结

通过此题我们可以发现，对于一些比较复杂的问题，我们往往不能第一时间看出其解法，此时要切忌从常理的角度去看待问题，否则很容易陷入思维定式，不能得到很好的解题思路。以此题为例，若我们在求解递推关系时将操作次数作为动态规划的对象，即设 F(i,j)为拥有 i 枚鸡蛋，楼层数为 j 的条件下确定 f 所需的最小操作次数，则在遍历填表时所需时间会急剧增加，算法效率严重不足。因此，学会放宽思路，尝试从别的角度看待问题，可能会让我们在解题时“茅塞顿开”，而这也是我们使用动态规划算法时一个比较关键的重点。总的来说，动态规划算法能够将大规模的问题分解成互独立、相对简单的小规模问题，用空间换取时间，具有较好的时间效率和空间效率。对一些特定题型，如穷举求最值问题，其往往具有“奇效”，需要我们能够灵活掌握。