**深 圳 大 学 实 验 报 告**

**课程名称： 算法设计与分析**

**实验名称： 实验四 动态规划—鸡蛋掉落问题**

**学院： 计算机与软件学院 专业：计算机科学与技术（创新班）**

**报告人： XXX 学号： XXXXXXXXXX 班级： 高性能班**

**同组人：**

**指导教师： 杨烜**

**实验时间： 2024.5.7——2024.5.19**

**实验报告提交时间： 2024.5.19**

**教务处制**

**一、实验目的**

* + 1. 掌握动态规划算法设计思想。
    2. 掌握鸡蛋坠落问题的动态规划解法。

**二、问题描述**

有一栋N 层的楼，K 个鸡蛋（K 至少为 1）。现在确定这栋楼存在楼层 0 <= F <= N，在这层楼将鸡蛋扔下去，鸡蛋恰好不摔碎（高于 F 的楼层都会碎，低于 F 的楼层都不会碎），这样的楼层F被称为门槛层。问最坏情况下，至少要扔几次鸡蛋，才能确定这个门槛层 F。

问题约束条件：

* 从跌落中幸存下来的鸡蛋可以再次使用。
* 破蛋必须丢弃。
* 摔碎对所有鸡蛋的影响都是一样的。
* 如果一个鸡蛋掉在地上摔碎了，那么它从高处掉下来也会摔碎。
* 如果一个鸡蛋在跌落中幸存下来，那么它在较短的跌落中也能完整保留下来。

任务要求：

* 给出解决问题的动态规划方程；
* 随机产生f，e的值，对小数据模型利用蛮力法测试算法的正确性；
* 随机产生f ，e的值，对不同数据规模测试算法效率，并与理论效率进行比对，请提供能处理的数据最大规模，注意要在有限时间内处理完；
* 该算法是否有效率提高的空间？包括空间效率和时间效率。

**三、求解问题的算法原理描述**

**1、问题分析**

**1）最坏情况**

在不考虑鸡蛋限制、楼层数N=8的条件下，若我们采取从下到上扔鸡蛋的方法来确认门槛层，则最坏情况需要扔8次鸡蛋，即门槛层为8层。假设在第3层扔时鸡蛋便碎了，说明第2层为门槛层，此时只扔了4次鸡蛋便确定了门槛层，显然不是我们想要的最坏情况。

又或者，采取二分查找的思路来扔鸡蛋，先去（1+8）/2=4层扔鸡蛋，碎了说明F小于4，则去（1+3）/2=2层扔鸡蛋……；没碎则说明F大于等于4，则取（4+8）/2=6层扔鸡蛋……显然对于该策略来说，最坏情况显然是第8层或第0层为门槛层，搜索到门槛层扔鸡蛋的次数最高。

图示

描述已自动生成

**2）最少扔鸡蛋次数**

我们的另一个目标是求扔鸡蛋的最少次数。

还是以刚刚的条件为例，可以看到线性扫描和二分查找两个方法最坏情况下门槛层都在第8层，但显然用二分查找的策略去找门槛层扔鸡蛋的次数要比线性扫描少得多。因此，我们会更倾向于选择二分查找的策略来扔鸡蛋，由此达到最少扔鸡蛋次数的目标。

**2、问题的复杂性**

**1）鸡蛋数K的限制**

由于有鸡蛋数K的限制条件，我们无法一直使用二分查找的策略，假设现在只有两个鸡蛋，如果在尝试过程中一个鸡蛋碎了，另一个鸡蛋为了确保能在碎之前能确定门槛层，因此不能再进行二分查找的策略（如下图，第一个鸡蛋在第8层碎了，说明F小于8，如果继续使用二分查找的策略去尝试第4层，倘若门槛层是小于4的，那第二个鸡蛋也破碎，就没有剩余鸡蛋可以用于确认小于4的门槛层），只能改为使用线性扫描的策略。

图表

描述已自动生成

**2）使用策略引起最坏情况的变化**

在前面分析最坏情况时，有提到对于线性扫描的策略，最坏情况是门槛层在未测楼层的最高层，而对于二分查找的策略，最坏情况为楼层顶或楼层底。如果在求解问题时，同时使用了这两种方法，必然会导致最坏情况发生变化。

例如，当楼层数N=16、鸡蛋数K=2时，一开始使用二分查找的策略，此时最坏情况是门槛层为第0或16层，但但第一次尝试时，鸡蛋碎了，说明门槛层F小于8，此时必须改用线性扫描的策略，同时最坏情况也改变为门槛层是第7层。

图表

描述已自动生成

**3）求解问题的策略多**

虽然我们前面提到了线性扫描和二分查找这两种策略，但求解该问题可并不局限于该两种方法，我们还可以采取四分、五分、十分等各种策略。

如下图，采用四分的策略，即当鸡蛋数大于1时，隔四层楼扔一次鸡蛋，若鸡蛋碎了，就可以从上一次扔鸡蛋的位置开始向上线性扫描或采取其他策略进行求解。如果在N=16、K=2的条件下使用该方法，就能在最坏情况下最少扔7次鸡蛋就能找到门槛层；而如果使用二分查找的策略，则在最坏情况下最少要扔9次鸡蛋才能找到门槛层。因此，在该条件下四分查找是优于二分查找的。当然，也可能还存在其他策略能在该条件下得到更优的解。

因此，最优策略可能非常多，并且没有什么规律可寻。

图表

描述已自动生成

根据前面分析的三点，我们足以见到该问题的复杂性。这不经令我们想，有什么方法可以不考虑之前的决策，只需考虑当前决策以及某些限定条件，去达到问题的最优呢？

**3、动态规划**

动态规划（Dynamic Programming）是一种将问题分解成更小子问题，并逐步解决子问题以构建最终解的方法。

假设我们要在鸡蛋数为K、楼层数为N的条件下求解问题，若已经知道了鸡蛋数小于K或楼层数小于N的条件下问题的最优解，我们就有机会基于某种限定条件，利用已知的最优解来求解目前条件的最优解，而在求解过程中，我们不需要关心这些已知最优解是怎么得来的、它们之前是采取了何种策略，只需要关系当前做何种决策能使得问题达到最优即可。可见，动态规划确实有可能帮助我们避免了之前提到的各种复杂细节和限制。

使用动态规划的关键概念：

**1）状态定义**

dp[k][n]表示使用k个鸡蛋和有n层楼时，找到门槛层在最坏情况下的最少扔鸡蛋次数。

**2）状态转移方程**

以求解dp[2][8]为例，当鸡蛋数K=2、楼层数N=8时，假设我们在第4层仍一次鸡蛋。

如果鸡蛋碎了，那么鸡蛋的个数减一，变为2-1=1；同时，因为门槛层必然小于4，所以搜索的楼层区间应该从1到8改为1到4-1=3。因此，我们需要接着求解鸡蛋数K=1、楼层数N=3条件下的问题，即求解dp[1][3]。

如果鸡蛋没碎，那么鸡蛋的个数不变，仍为2-0=2；同时，因为门槛层必然大于等于4，所以搜索的楼层区间应该从1到8改为1到8-4=4。因此，我们需要接着求解鸡蛋数K=2、楼层数N=4条件下的问题，即求解dp[2][4]。

以上两种不同的情况均会划分出两个新的子问题，我们则需要接着子问题按照上面的分析方法往下求解，把求解得到的结果提供给当前问题，帮助当前问题做出最优决策。

电脑屏幕的照片

低可信度描述已自动生成一些文字和图案

描述已自动生成

当求解出两个子问题后，假设dp[2][4]=2，dp[1][3]=3，因为我们考虑的是最坏情况下的最优解，所以此时鸡蛋碎了的情况为最坏情况，因为它的最少扔鸡蛋次数更大。因此，加上一开始时在第4层扔鸡蛋的一次，对于dp[2][8]来说，其答案便为max(2,3)+1=4。

图表, 图示

描述已自动生成

当然，以上是基于在第4层扔鸡蛋得到的dp[2][8]，那其实可能在其他楼层进行扔鸡蛋，或许还能得到比当前更小的答案，即为了满足目标求解最少扔鸡蛋次数，我们考虑对当前所有楼层都扔一次鸡蛋，哪个需要的扔鸡蛋次数最少，则为dp[2][8]的最终答案。

综上，我们可以写出状态转移方程。

* 如果我们从第i层楼扔鸡蛋，有两种情况：

鸡蛋碎了：此时我们需要在第i-1层楼的剩余k-1个鸡蛋中继续尝试。

鸡蛋没碎：此时我们需要在剩下的n-i层楼和k个鸡蛋中继续尝试。

* 转移方程为

**3）初始条件**

* dp[1][N] = N：只有一个鸡蛋时，需要从第1层到第N层线性扫描。
* dp[K][0]= 0：楼层数为0，则无需尝试。
* dp[K][1]= 1：当楼层数为1，只需要在第1层扔一次鸡蛋，若鸡蛋碎了，则第0层位门槛层，若鸡蛋没碎，则第1层位门槛层。
* 其余dp[K][N]=INF：问题还未求解，即处于无解状态。

**4、小数据集验证算法的正确性**

假设拥有的鸡蛋数为3 、需要测试的楼层数为10，给出最坏情况下的最少测试次数。

这里我们用动态规划进行求解，首先按要求初始化全部dp。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 楼层数N | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 鸡蛋数K | 1 | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| 2 | **0** | **1** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** |
| 3 | **0** | **1** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** | **INF** |

接下来，根据动态转移方程，我们可以计算出N=2时dp[K][2]的全部值。

求dp[2][2]：

n=1时，dp[2][2] = min(INF, max(dp[1][0], dp[2][1]) + 1) = min(INF, max(0, 1) + 1) = 2

n=2时，dp[2][2] = min(2, max(dp[1][1], dp[2][0]) + 1) = min(2, max(1, 0) + 1) = 2

求dp[3][2]：

n=1时，dp[3][2] = min(INF, max(dp[2][0], dp[3][1]) + 1) = min(INF, max(0, 1) + 1) = 2

n=2时，dp[3][2] = min(2, max(dp[2][1], dp[3][0]) + 1) = min(2, max(1, 0) + 1) = 2

根据上面的计算结果更新表格。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 楼层数N | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 鸡蛋数K | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 0 | 1 | **2** | INF | INF | INF | INF | INF | INF | INF | INF |
| 3 | 0 | 1 | **2** | INF | INF | INF | INF | INF | INF | INF | INF |

同样，我们还可以根据动态规划转移方程计算出N=3时dp[K][3]的全部值。

求dp[2][3]：

n=1时，dp[2][3] = min(INF, max(dp[1][0], dp[2][2]) + 1) = min(INF, max(0, 2) + 1) =3

n=2时，dp[2][3] = min(3, max(dp[1][1], dp[2][1]) + 1) = min(3, max(1, 1) + 1) = 2

n=3时，dp[2][3] = min(2, max(dp[1][2], dp[2][0]) + 1) = min(2, max(2, 0) + 1) = 2

求dp[3][3]：

n=1时，dp[3][3] = min(INF, max(dp[2][0], dp[3][2]) + 1) = min(INF, max(0, 2) + 1) =3

n=2时，dp[3][3] = min(3, max(dp[2][1], dp[3][1]) + 1) = min(3, max(1, 1) + 1) = 2

n=3时，dp[3][3] = min(2, max(dp[2][2], dp[3][0]) + 1) = min(2, max(2, 0) + 1) = 2

根据上面的计算结果更新表格。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 楼层数N | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 鸡蛋数K | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 0 | 1 | 2 | **2** | INF | INF | INF | INF | INF | INF | INF |
| 3 | 0 | 1 | 2 | **2** | INF | INF | INF | INF | INF | INF | INF |

如此，我们可以按上面的步骤，通过状态转移方程计算出全部dp值。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 楼层数N | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 鸡蛋数K | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 |
| 3 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 |

接下来，我们利用代码验证以上结果是否正确。

图形用户界面, Word

中度可信度描述已自动生成

文本

描述已自动生成

可以看到，代码的输出结果和我们的上面推导得到的表格完全一致。

最后，我们在LeetCode平台在K=3、N=10的条件下对代码进行测试，答案正确，动态规划算法的正确性得以验证。

图片包含 应用程序

描述已自动生成

**5、动态规划算法的代码实现**

**1）递归法**

通过递归函数的方式来写动态转移方程。

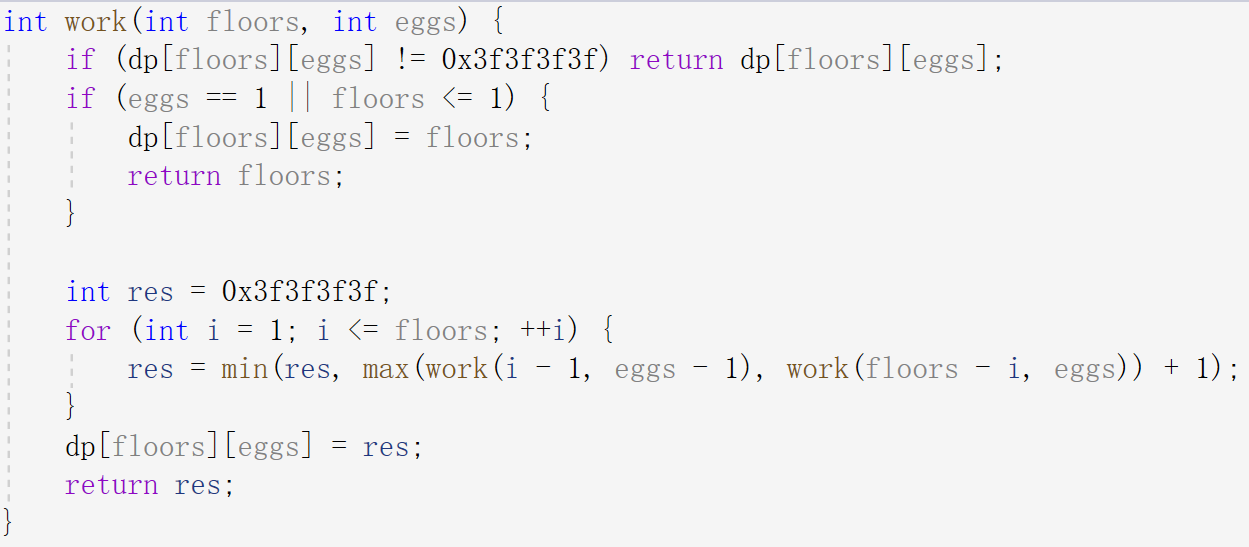
图形用户界面, Word

中度可信度描述已自动生成

由于递归过程会产生很多分支，并且当K和N的规模较大时，递归的深度将比较大，整个算法的时间效率便会非常低。所以该算法的时间复杂度大致为，K表示递归的深度，N表示递归时产生的分支；在空间上，由于递归需要占用大量的栈空间，空间复杂度应该也会很高。

**2）递归法+记忆化**

在上面的递归法中，由于每条分支都需要递归至底才能返回答案，因此同一节点产生的不同分支很可能产生了大量的重复计算，这将拉低的算法的效率。因此，我们可以采用一个二维数组dp记录下每个节点计算出的最优解，当再次递归到该节点时，直接返回之前求解出的最优解，从而达到剪枝效果，避免了大量不必要的计算。



由于使用记忆化后，递归法的效率大大提高，其时间复杂度大致为，当然由于大量的函数调用，也会产生一些额外的时间开销；由于使用了二维数组dp，该算法的空间复杂度大致为，且由于递归过程需要占用大量的栈空间，因此还会产生额外的空间开销。

**3）动态规划**

上面的两种方法均采用递归法的方式来实现动态规划，这里将采用for循环结构来实现动态规划。在上面小规模数据集验证算法的过程中，我们能注意到，动态规划的过程实际上便是填一个二维表格的过程，因此我们完全可以用两层for循环来实现不同K和N下的dp[K][N]的转移，同时再加上一层for循环，用于在决策过程中尝试每一个楼层来得到最优解。

文本

中度可信度描述已自动生成

由于该算法使用了三层for循环，且每层循环的代价为、、，所以该算法的时间复杂度为；同时，由于使用了二维数组，该算法的空间复杂度为。

**4）动态规划+二分查找**

前面的动态规划算法需要暴力穷举尝试在所有楼层1 <= i <= N扔鸡蛋，每次选择扔鸡蛋次数最少的那一层，这将会导致大量的时间开销。有没有什么方法可以帮助我们快速找到扔鸡蛋次数最少的那一层呢？

首先，根据前面的小规模数据集验证的过程中，我们不难发现当K固定时，dp[N][K]随着N的增加一定是单调递增的。这也符合我们日常逻辑，因为楼层数越高，在拥有的鸡蛋数一定的条件下，不管策略再怎么优秀，都不可避免测试次数的增加。

因此注意到动态转移方程中的dp[K-1][i-1]和dp[K][N- i]，其中i是从1到N单增的，如果我们固定K和N，把这两个状态看作是关于i的函数，前者随着i的增加应该也是单调递增的，而后者随着i的增加应该是单调递减的。

在下面的图中，我们不难看出，红色曲线代表了dp[K][N- i]，蓝色曲线代表了dp[K-1][i-1]，而绿色曲线则代表了我们期望得到的max(dp[K][N- i]，dp[K-1][i-1])。显然，绿色曲线必然是先单调递减再单调递增，其必然存在一个最低点，而这个最低点即表示最坏情况下最少扔鸡蛋的次数。但由于dp[K][N- i]和dp[K-1][i-1]为离散函数，所以只要当dp[K-1][i-1]最接近dp[K][N- i]时，则表示当前尝试的楼层就能得到最坏情况下最少扔鸡蛋的次数。

*图表, 折线图

描述已自动生成*

采用二分法查找的方式便能很快的找到这个最低点。

初始时，令l=1，r=n，mid=(l+r)/2。

若dp[k][n- mid]>dp[k-1][mid -1]，则表示最低点N>=mid，更新答案ans= dp[k][n- mid]+1，更新l=mid+1；

若dp[k][n- mid]<dp[k-1][mid -1]，则表示最低点N<=mid，更新答案ans= dp[k-1][mid -1]+1，更新r=mid-1；

若dp[k][n- mid]=dp[k-1][mid -1]，则表示最低点N=mid，更新答案ans= dp[k-1][mid -1]+1，退出循环。

以上过程循环至找到最低点或l>r。

文本

描述已自动生成

由于采用了二分查找的方式来优化找最少扔鸡蛋次数楼层的过程，所以该过程的时间复杂度从优化至，结合外层的两个for循环，该算法的时间复杂度为；同时，由于使用了二维数组，该算法的空间复杂度为。

**5）动态规划+重写状态转移方程**

前面的全部算法都是基于dp[K][N]表示有K个鸡蛋、测N层楼时最坏情况下最少扔鸡蛋次数来进行求解的。按照这个定义，当确定当前的鸡蛋个数和面对的楼层数，就知道最少扔鸡蛋次数为dp[K][N]。

现在，我们稍微修dp数组的定义，确定当前的鸡蛋个数和最多允许的扔鸡蛋次数，就知道能够确定门槛层F的最高楼层数，即dp[K][M]表示有K个鸡蛋，可以尝试扔M次鸡蛋，在最坏情况下能测出一栋楼门槛层的这栋楼的最多层数。

举例来说，如dp[1][5] = 5表示有1个鸡蛋，可以尝试扔5次鸡蛋，在最坏情况下能测出最多有5层的楼的门槛层，相当于一层一层进行线性扫描。我们能发现，这个表示方法，与我们之前的表示方法是等价的。

对于问题的解，显然当M逐渐递增，当出现dp(K,M) >= N 时，则表明K个鸡蛋，测试M次，最坏情况下最多能找到楼层数大于等于N层的楼的门槛层，即等价于有K个鸡蛋、N层楼最坏情况下最少扔鸡蛋次数为M，这便是本问题的最优解。

以求解dp[2][4]，dp[2][4]得到F最多能测试的楼层数取决于下面两种状态。

当鸡蛋碎了，就需要往楼下进行扔鸡蛋，此时鸡蛋数K减一，即为2-1=1，测试次数M也要减一，即4-1=3，这时候便需要去求解dp[1][3]得到F最多能测试的楼层数。

当鸡蛋没碎，就需要往楼上进行扔鸡蛋，此时鸡蛋数K不变，即为2-0=2，测试次数M要减一，即4-1=3，这时候便需要去求解dp[2][3]得到F最多能测试的楼层数。

无论是往楼上测还是往楼下测，最终dp[2][4]得到F最多能测试的总楼层数=楼上最多能测试的楼层数+楼下最多能测试的楼层数+当前测试的楼层数，即dp[2][4]= dp[2][3]+ dp[1][3]+1。

图示

描述已自动生成

根据上面描述的思路，我们可以得到新的状态转移方程：

dp[k][m - 1]表示楼上最多能测试的楼层数，鸡蛋个数k不变，也就是鸡蛋没碎，扔鸡蛋次数m减一；

dp[k - 1][m - 1]表示楼下最多能测试的楼层数，鸡蛋个数k减一，也就是鸡蛋碎了，同时扔鸡蛋次数m减一。

初始化：

* dp(0,M) = 0：当鸡蛋数为0时，只能测楼层数为0的楼。
* dp(K,0) = 0：当扔鸡蛋次数为0时，只能测楼层数为0的楼。
* dp(K,1) = 1：当扔鸡蛋次数为1时，最多能测的楼层数为1，即在第1层扔鸡蛋，若鸡蛋碎了，则表明第0层为门槛层；当鸡蛋没碎，则表明第1层为门槛层。

文本

描述已自动生成

根据上面的分析，我们便能通过两层for循环实现所有状态的求解，由于扔鸡蛋的次数M必然小于等于楼层数N，所以该算法的时间复杂度为，K表示鸡蛋数，N表示扔鸡蛋的次数；但显然根据测试结果，扔鸡蛋的次数远小于楼层数的，故经过数学证明，能得到该算法的时间复杂度应该为；同时，由于使用了二维数组，该算法的空间复杂度为。

**6）动态规划+重写状态转移方程+空间压缩**

重新观察上面新的到的状态转移方程，不难发现，扔鸡蛋的次数M必定只和前一个状态M-1相关，即每次状态转移必然从dp[K][M-1]转移到dp[K][M]，因此我们能将状态转移方程化简为：

文本, 白板

描述已自动生成

需要注意的是，在进行状态转移时，必须从后往前进行转换转移，即从K到1进行状态转移。这是因为如果从前往后进行状态转移，则会导致旧状态的解被覆盖，然而后面新状态的解还需要基于该旧状态的解进行更新，此时便会导致错误。

下图便展示了不同转移顺序过程导致的结果，能看到，从前往后进行状态转移中，新状态的dp[k]的解由新状态下的dp[k-1]和旧状态下的dp[k]转移而来，但正确的结果应该是新状态的dp[k]的解由旧状态下的dp[k-1]和dp[k]转移而来，因此从前往后进行状态转移必然会导致结果出错。

图示

描述已自动生成

该算法的时间复杂度为；由于此时只需使用一维数组便能完成状态转移，因此空间复杂度为。

**四、算法测试结果及效率分析**

**1、递归法**

在LeetCode上仅通过了32个测试点，其余测试点超出了时间限制。

**图形用户界面, 文本, 应用程序

描述已自动生成**

**2、递归法+记忆化**

在LeetCode上仅通过了73个测试点，其余测试点超出了时间限制。

**图形用户界面, 文本, 应用程序

描述已自动生成**

**3、动态规划**

在LeetCode上仅通过了73个测试点，其余测试点超出了时间限制。

**图形用户界面, 文本, 应用程序

描述已自动生成**

在本地测试，固定鸡蛋个数，楼层数N的规模不断增大，检验算法能处理数据的最大规模，并且以楼层数N=200000为基准点，计算出理论测试时间。

最终结果如下。

|  |  |  |
| --- | --- | --- |
| 动态规划法在鸡蛋数K=2、不同规模楼层数N、最坏情况下找到最少测试次数所用时间 | | |
| 楼层数N | 实际测试时间/s | 理论测试时间/s |
| 100 | 0.000000 | 0.000031 |
| 1000 | 0.008600 | 0.003090 |
| 10000 | 1.470800 | 0.308989 |
| 100000 | 77.692000 | 30.898900 |
| 200000 | 123.595600 | 123.595600 |
| 400000 | 318.439200 | 494.382400 |
| 600000 | 988.140600 | 1112.360400 |
| 800000 | 1922.595200 | 1977.529600 |
| 1000000 | 3127.942000 | 3089.890000 |

可以看到，对于该算法来说，数据规模的瓶颈为N=1000000，当数据量再增大时，耗时将会变得非常长。并且能发现，理论与实测误差比较小，曲线基本呈的趋势增长。

下面随机生成K和N，查看算法效率的变化。

最终结果如下（横轴为N的规模，纵轴为K的规模）。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 | 100000 |
| 2 | 0.0104 | 0.1592 | 0.7968 | 2.1343 | 3.8969 | 4.6857 | 3.0471 | 4.1346 | 5.0034 | 5.5629 |
| 3 | 0.2144 | 1.2127 | 3.0408 | 6.6606 | 12.0635 | 22.0351 | 20.1802 | 29.2452 | 36.0140 | 49.0176 |
| 4 | 0.2058 | 2.0188 | 6.3277 | 12.4379 | 19.5651 | 26.9242 | 37.2179 | 57.5228 | 72.0214 | 97.0236 |
| 5 | 0.3445 | 2.8037 | 7.9789 | 19.5365 | 25.5142 | 38.2980 | 54.6716 | 76.1006 | 90.0278 | 108.0319 |
| 6 | 0.5819 | 3.1302 | 10.9333 | 23.5081 | 37.8184 | 49.7566 | 69.2549 | 97.6524 | 123.0307 | 135.0371 |
| 7 | 0.3888 | 5.3890 | 13.5078 | 25.4007 | 42.9083 | 63.7648 | 79.7604 | 116.9746 | 139.0349 | 158.0361 |
| 8 | 0.4766 | 6.7507 | 15.4884 | 32.0132 | 57.3156 | 68.8723 | 99.6529 | 130.4690 | 157.0391 | 179.0416 |
| 9 | 0.5496 | 6.7996 | 18.8910 | 35.6446 | 59.2025 | 78.4742 | 112.5900 | 143.1916 | 174.0406 | 196.0439 |
| 10 | 0.6697 | 8.3437 | 23.1088 | 43.8728 | 65.8148 | 92.0833 | 124.8380 | 162.8624 | 189.0418 | 204.0456 |

图表, 折线图

描述已自动生成

不难发现，随着N的增大，算法的时间花销也增大；随着K的增大，算法的时间花销也同样在增大，这是因为算法的时间复杂度为，显然时间的开销会随着K和N的变化而产生变化。

**4、动态规划+二分查找**

在LeetCode上通过了全部测试点，用时88ms，消耗内存13.54MB。

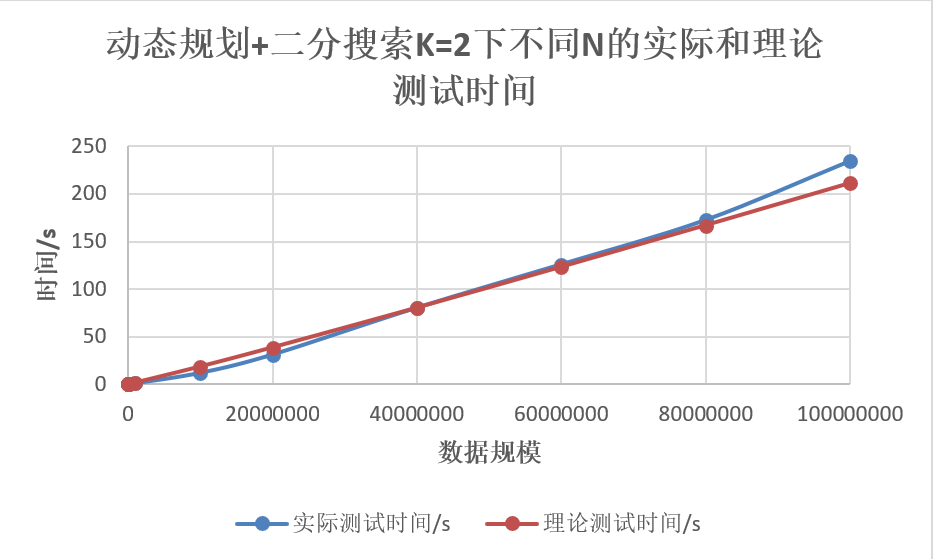
**图形用户界面, 文本, 应用程序

描述已自动生成**

在本地测试，固定鸡蛋个数，楼层数N的规模不断增大，检验算法能处理数据的最大规模，并且以楼层数N=40000000为基准点，计算出理论测试时间。

最终结果如下。

|  |  |  |
| --- | --- | --- |
| 动态规划+二分查找法在鸡蛋数K=2、不同规模楼层数N、最坏情况下找到最少测试次数所用时间 | | |
| 楼层数N | 实际测试时间/s | 理论测试时间/s |
| 100 | 0.000000 | 0.000053 |
| 1000 | 0.000000 | 0.000792 |
| 10000 | 0.001800 | 0.010554 |
| 100000 | 0.059200 | 0.131931 |
| 1000000 | 0.833200 | 1.583166 |
| 10000000 | 11.746200 | 18.470273 |
| 20000000 | 30.922800 | 38.529148 |
| 40000000 | 80.235500 | 80.235500 |
| 60000000 | 125.801500 | 123.141067 |
| 80000000 | 172.458200 | 166.825407 |
| 100000000 | 234.412900 | 211.088837 |



可以看到，对于该算法来说，数据规模的瓶颈为N=100000000，虽然时间上还没达到上千秒，但如果规模再增大，其运行消耗的内存就会超过本地运行内存，本地计算机无法支持该程序的运行而导致崩溃。并且能发现，相较于未使用二分查找的动态规划算法，其运行效率有了大幅度的提升，且理论与实测误差比较小，曲线基本呈的趋势增长。

下面随机生成K和N，查看算法效率的变化。

最终结果如下（横轴为N的规模，纵轴为K的规模）。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 | 100000 |
| 2 | 0.0000 | 0.0011 | 0.0018 | 0.0009 | 0.0006 | 0.0028 | 0.0041 | 0.0059 | 0.0034 | 0.0029 |
| 3 | 0.0008 | 0.0023 | 0.0043 | 0.0051 | 0.0069 | 0.0096 | 0.0118 | 0.0146 | 0.0140 | 0.0176 |
| 4 | 0.0008 | 0.0034 | 0.0071 | 0.0076 | 0.0109 | 0.0134 | 0.0188 | 0.0176 | 0.0214 | 0.0236 |
| 5 | 0.0007 | 0.0042 | 0.0076 | 0.0092 | 0.0124 | 0.0164 | 0.0209 | 0.0223 | 0.0278 | 0.0319 |
| 6 | 0.0015 | 0.0048 | 0.0082 | 0.0105 | 0.0146 | 0.0180 | 0.0242 | 0.0258 | 0.0307 | 0.0371 |
| 7 | 0.0014 | 0.0062 | 0.0095 | 0.0120 | 0.0167 | 0.0203 | 0.0255 | 0.0279 | 0.0349 | 0.0361 |
| 8 | 0.0017 | 0.0063 | 0.0105 | 0.0129 | 0.0187 | 0.0202 | 0.0287 | 0.0326 | 0.0391 | 0.0416 |
| 9 | 0.0012 | 0.0066 | 0.0091 | 0.0144 | 0.0185 | 0.0245 | 0.0299 | 0.0346 | 0.0406 | 0.0439 |
| 10 | 0.0018 | 0.0067 | 0.0095 | 0.0137 | 0.0197 | 0.0254 | 0.0286 | 0.0353 | 0.0418 | 0.0456 |

图表, 折线图

描述已自动生成

不难发现，随着N的增大，算法的时间花销也增大；随着K的增大，算法的时间花销也同样在增大，这是因为算法的时间复杂度为，显然时间的开销会随着K和N的变化而产生变化。

**5、动态规划+重写状态转移方程**

在LeetCode上通过了全部测试点，用时36ms，消耗内存13.37MB。

图形用户界面, 文本, 应用程序, 电子邮件, 网站

描述已自动生成

在本地测试，固定鸡蛋个数，楼层数N的规模不断增大，检验算法能处理数据的最大规模，并且以楼层数N=600000000为基准点，计算出理论测试时间。

最终结果如下。

|  |  |  |
| --- | --- | --- |
| 动态规划+重写状态方程在鸡蛋数K=2、不同规模楼层数N、最坏情况下找到最少测试次数所用时间 | | |
| 楼层数N | 实际测试时间/s | 理论测试时间/s |
| 100000000 | 0.000064 | 0.000061 |
| 200000000 | 0.000079 | 0.000086 |
| 400000000 | 0.000113 | 0.000122 |
| 600000000 | 0.000149 | 0.000149 |
| 800000000 | 0.000174 | 0.000172 |
| 1000000000 | 0.000199 | 0.000192 |
| 2000000000 | 0.000255 | 0.000272 |

可以看到，对于该算法来说，数据规模的瓶颈为N=2000000000，虽然运行时间非常低，但如果规模再增大，其运行消耗的内存就会超过本地运行内存，本地计算机无法支持该程序的运行而导致崩溃。并且能发现，理论与实测误差比较小，曲线基本呈的趋势增长。

**6、动态规划+重写状态转移方程+空间压缩**

在LeetCode上通过了全部测试点，用时0ms，消耗内存6.99MB。

图形用户界面

描述已自动生成

在本地测试，固定鸡蛋个数，楼层数N的规模不断增大，检验算法能处理数据的最大规模，并且以楼层数N=600000000为基准点，计算出理论测试时间。

最终结果如下。

|  |  |  |
| --- | --- | --- |
| 动态规划+重写状态方程+空间压缩在鸡蛋数K=2、不同规模楼层数N、最坏情况下找到最少测试次数所用时间 | | |
| 楼层数N | 实际测试时间/s | 理论测试时间/s |
| 1e12 | 0.004900 | 0.003263 |
| 1e13 | 0.012500 | 0.010320 |
| 1e14 | 0.038900 | 0.032635 |
| 1e15 | 0.103200 | 0.103200 |
| 1e16 | 0.416400 | 0.326347 |
| 1e17 | 1.137800 | 1.032000 |
| 1e18 | 3.516500 | 3.263471 |

可以看到，对于该算法来说，数据规模的瓶颈为N=1e18，虽然运行时间非常低，但如果规模再增大，N超过了long long的表示范围，会导致程序出错。并且能发现，其效率相较于空间压缩前有所提升，且理论与实测误差比较小，曲线基本呈的趋势增长。

**五、经验总结**

这次实验进一步加深了我对动态规划的理解。如何能快速理解一个动态规划的算法过程，使用表格来记录变化是一个非常好的选择。通过表格来展示动态转移的过程，便能直观地看到整个变化过程，理解算法的正确性。

同时，用算法解决问题不能将思维定死，最典型的例子便是重写状态转移方程的动态规划算法，显然只是换了一种新的状态表示方法，但在效率上却取得了大幅度飞跃，而这仅仅是从一个逆向的角度去考虑问题，才能得到如此的结果。所以看待问题不能一上来便将思维定死，要从多角度考虑问题，想想有没有更好地状态表示来优化算法的效率。

|  |
| --- |
| 指导教师批阅意见：    成绩评定：  指导教师签字：  年 月 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。