在此处键入公式。

深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验项目名称：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

指导教师：刘刚

报告人：林宪亮 学号：2022150130

实验时间：2024 年 5 月 10 日—2024 年 5 月 14 日

实验报告提交时间：2024 年 5 月 日

教务部制

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、**实验内容：**

鸡蛋掉落问题是理解动态规划如何实现最佳解决方案的一个很好的例子。问题描述如下：

我们需要用鸡蛋确认在多高的楼层鸡蛋落下来会破碎，这个刚刚使鸡蛋破碎的楼层叫门槛层，门槛楼层是鸡蛋开始破碎的楼层，上面所有楼层的鸡蛋也都破了。另外，如果鸡蛋从门槛楼层以下的任何楼层掉落，它都不会破碎。如上图所示，如果有 5 层，我们只有 1 个鸡蛋，要找到门槛层，则必须尝试从每一层一层一层地放下鸡蛋，从第一层到最后一层，如果门槛层是第 k 层，那么鸡蛋就会在第 k 层抛下时破裂，应该做了 k 次试验。

注意：我们不能随机选择任何楼层，例如，如果我们选择 4 楼并放下鸡蛋并且它打破了，那么它不确定它是否也从 3 楼打破。 因此，我们无法找到门槛层，因为鸡蛋一旦破碎，就无法再次使用。

给定建筑物的一定数量的楼层（比如 f 层）和一定数量的鸡蛋（比如 e 鸡蛋），找出阈值地板必须执行的最少的鸡蛋掉落试验的次数，注意，这里需要求的是试验的测试，不是鸡蛋的个数。还要记住的一件事是，我们寻找的是找到门槛层所需的最少鸡蛋掉落试验次数，而不是门槛层下限本身。

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。

1、给出解决问题的动态规划方程；

2、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；

3、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；

4、该算法是否有效率提高的空间？包括空间效率和时间效率。

四、实验内容及过程：

1. 题目分析：

• 题目大意：已知有 e 个鸡蛋，f 层楼，求至少需要仍几次鸡蛋，才能保证无论临界楼层是多少都能把它找出来。

• 解题思路：如果我尝试从 1-f 层楼间的任意一层楼 x 扔鸡蛋，那么会有两种情况：

（1）鸡蛋没碎，那么说明 1-x 层楼就不需要考虑了，因为比 x 层楼低的楼层扔鸡蛋都不会碎，那么我们可以用 e 个鸡蛋继续从剩下的 f-x 层楼找出临界楼层。

（2）鸡蛋碎了，那么说明 x-f 层楼是不需要考虑的，因为比 x 层楼高的楼层扔鸡蛋一定会碎，那么我们可以用 e-1 个鸡蛋继续从剩下的 x-1 层楼找出临界楼层。

由于临界楼层是随意的，我们需要保证一定能找出这个临界值，所以我们要考虑的是最坏的情况，但是我们要求解所需要的最少的扔鸡蛋次数，所以我们需要找出最好的扔鸡蛋策略。

所以解决问题的动态规划方程为：
公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1))+1)1 <= x <= f$。

其中，F(i,j)表示已知有 i 个鸡蛋，j 层楼时，至少需要扔的鸡蛋数。F(i,j-x)表示鸡蛋没碎的情况，F(i-1,x-1)表示的是鸡蛋碎了的情况。由于考虑最坏的情况，所以我们需要取两种情况的最大值，由于要找出最优解，我们又需要在不同的扔法中找出一个最小值。

设置初始条件 F(1,1)-F(1,f)为 f，F(1,1)-F(e,1)为 1 因为当只有一个鸡蛋时只能一层层尝试而当层数为 1 时，不管有几个鸡蛋都只需要扔一次鸡蛋。

2. 蛮力法

蛮力法即将动态规划方程：
公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1))+1)1 <= x <= f$。
使用递归的方式实现，这样会产生很多重复性的计算，时间复杂度很高。

伪代码：

Algorithm 1： Drop_Egg_Brute

Input： num_of_egg(e), num_of_floor(f)
Output: drop_num
1. If(e == 1 | | f == 0 | | f == 1 | | f == 2)
2. Return f;

```
3. For i in range(f):
4.      drop = max(Drop_Egg_Brute(e,f-i),Drop_Egg_Brute(e-1,i-1))+1
5.      Drop_min=min(drop,Drop_min)
6. Return Drop_min
```

说明：
当只有一个鸡蛋或者楼层数为 0，1，2 时，直接返回当前的楼层数，因为这些情况下最少的扔鸡蛋次数就是当前的楼层数。

```
7. For i in range(f):
8.      drop = max(Drop_Egg_Brute(e,f-i),Drop_Egg_Brute(e-1,i-1)+1
9.      Drop_min=min(drop,Drop_min)
```

这三条语句即为对动态规划方程：
公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1)) + 1)1 <= x <= f$。
的实现。

复杂度分析：
由于使用了循环加上递归实现，那么蛮力法的时间复杂度一定是指数级别的。
空间复杂度为 O(E×F)。

3. 动态规划

一样的是实现动态规划方程：
公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1)) + 1)1 <= x <= f$。

可以使用二维数组对子问题的结果进行保存，这样可以除去很多的重复性工作，提高算法的时间效率。

伪代码：

Algorithm 2： Drop_Egg_DTGH

Input： num_of_egg(e), num_of_floor(f)
Output: drop_num

```
1. If(e == 1 || f == 0 || f == 1 || f == 2)
2. Return f;
3. EF[e+1][f+1]={0};初始化二维数组为全 0
4. For i in range(1-e):
5.      For j in range(1-f):
6.          EF[i][j] = j;
7. For i in range(2-e):开始一步步从子问题开始求解
8.     For j in range(2-f):
```

```
9.        Drop_min=EF[i][j]
10.       For k in range(1-j):
11.           Drop=max(EF[i-1][k-1],EF[i][j-k])+1;
12.           Drop_min= min(Drop_min,drop);
13.       EF[i][j]=Drop_min
14. Return EF[e][f]
```

说明：
以上代码即使根据递推式：
公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1))+1)1<=x<=f$。
一步步填充下面的二维数组，直到填满，返回 EF[e][f],即当有 e 个鸡蛋，f 层楼需要扔几次鸡蛋。

表一： 二维矩阵 1

I \ J	1	2	3	4	5	…
1	1	2	3	4	5	…
2	1	2	3	3	3	…
3	1	2	3	3	4	…
4	1	2	3	3	4	…
5	1	2	3	3	4	…
…	…	…	…	…	…	…

复杂度分析：

- 时间复杂度：时间复杂都由算法中的三层循环决定，为 O(F^2×E)。
- 空间复杂度：空间复杂度为存储二维数组需要的额外空间，为 O（F×E)。

4. 算法优化

（1）优化 1：动态规划+二分查找

公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1))+1)1<=x<=f$。

对于上述动态规划方程中，随着 x 的增大：
- F(i,j-x)中，j-x 的值不断减小，也就是楼层数不断减少，那么也就是说明 F(i,j-x)的值也是非增的。
- F(i-1,x-1)中，x-1 的值是不断增大的，也就是说楼层数是增大，那么也就是说 F(i-1,x-1)的值是非减的。
- 根据动态规划方程，不难得出，当 x 加一或者减一时，F(i,j-x)和 F(i-1,x-1)的变化

范围不会超过 1。

所以，可以画出下图：

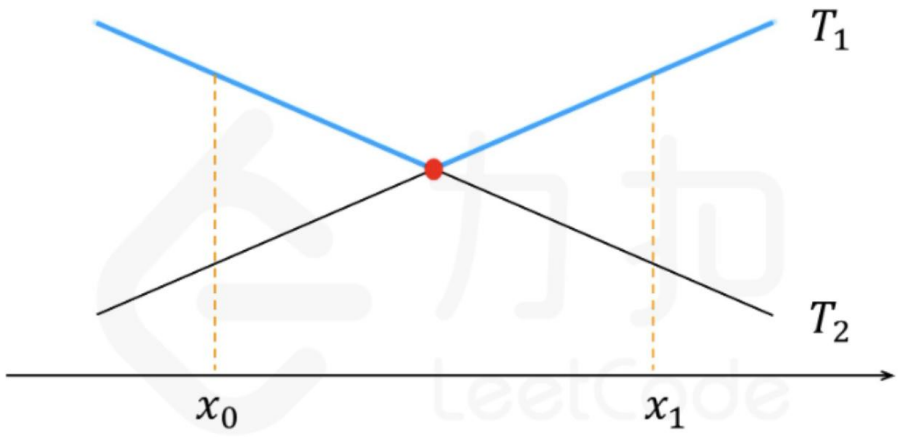


图 1 二分查找

横坐标代表 x，上升的直线 T1 代表 F(i-1,x-1)，下降的直线代表 F(i,j-x)。
由于 F(i,j)=Min(Max(F(i,j-x),F(i-1,x-1))+1) 1<=x<=f，那么其实 F(i,j)得值就是蓝色折线的最小值，也就是 F(i,j-x)==F(i-1,x-1)的时候（当然也可能不存在 F(i,j-x)==F(i-1,x-1)的情况，但由于根据动态规划方程，不难得出，当 x 加一或者减一时,F(i,j-x)和 F(i-1,x-1)的变化范围不会超过 1,这不会影响我们二分查找的结果）。所以我们可以使用二分查找找出 F(i,j)的值。

Algorithm 3： Drop_Egg_DTGH_improve1

Input： num_of_egg(e), num_of_floor(f)
Output: drop_num

```
1. If(e == 1 || f == 0 || f == 1 || f == 2)
2. Return f;
3. EF[e+1][f+1]={0};初始化二维数组为全 0
4. For i in range(1-e):
5.       For j in range(1-f):
6.             EF[i][j] = j;
7. For i in range(2-e):开始一步步从子问题开始求解
8.   For j in range(2-f):
9.        Drop_min=EF[i][j]
10.        While(left<right)
11.           if(EF[i-1][mid-1]<EF[i][j-mid]
12.                 left=mid+1;
13.            else right = mid
14.        EF[i][j] = max (EF[i-1][left-1],EF[i][j-left])+1
11. Return EF[e][f]
```

说明：
“当然也可能不存在 F(i,j-x)==F(i-1,x-1)的情况，但由于根据动态规划方程，不难得出，当 x 加一或者减一时，F(i,j-x)和 F(i-1,x-1)的变化范围不会超过 1，这不会影响我们二分查找的结果。”
为什么说这样并不影响我们二分查找结果呢，比如达到 left>right 的前一步，F(i,j-x)=a，F(i-1,x-1)=b，此时 F(i,j-x)>F(i-1,x-1),当 x 加一，因为假设不存在 F(i,j-x)==F(i-1,x-1)的情况，并且 F(i,j-x)，F(i-1,x-1)的变化量都只能是一，那么变化后 F(i,j-x)=a-1，F(i-1,x-1)=b+1，此时 F(i,j-x)<F(i-1,x-1),就只存在一种情况，那么就是 a-1=b，b+1=a，所以不管是在 left 求最大值还是在 right 求最大值都是一样的。

复杂度分析：
• 时间复杂度为 O(F×E×logF)，优化后的改变就是降低了求 EF[i][j]时的复杂度。
• 空间复杂度依旧是存储二维数组所花销的空间，O（E×F）。

（2）优化 2
我们可以改变一下解题思路，使用 EF[i][j]表示 i 个鸡蛋，j 次扔鸡蛋尝试最多可以测试的楼层。
那么即可以得出全新的状态转移方程：
公式 2：$EF[i][j] = EF[i][j-1] + EF[i-1][j-1] + 1$
其中 EF[i][j-1]表示如果鸡蛋没碎，那么剩下的 i 个鸡蛋和 j-1 次尝试可以检测的楼层数目，EF[i-1][j-1]表示如果鸡蛋碎了，那么剩下的 i-1 个鸡蛋，j-1 次尝试可以检测的楼层数。所以如果有 EF[i][j-1] + EF[i-1][j-1] + 1 层楼，i 个鸡蛋和 j 次尝试机会，那么在第 EF[i][j-1]+1 层扔一次鸡蛋，如果鸡蛋碎了，那么剩下的低 EF[i-1][j-1]层可以使用 i-1 个鸡蛋，j-1 尝试检测出，如果鸡蛋没碎，那么剩下的高 EF[i][j-1]层可以使用 i 个鸡蛋，j-1 次尝试检测出来，所以 i 个鸡蛋和 j 次尝试机会最多可以检测出 EF[i][j-1] + EF[i-1][j-1] + 1 层楼。

新的动态规划方程：
公式 2：$EF[i][j] = EF[i][j-1] + EF[i-1][j-1] + 1$

伪代码：

Algorithm 4： Drop_Egg_DTGH_improve2

Input： num_of_egg(e), num_of_floor(f)
Output: drop_num

```
1. If(e == 1 || f == 0 || f == 1 || f == 2)
2. Return f
3. EF[egg+1][floor+1],其中 EF[1][1]-EF[1][floor]=floor,EF[1][1]-EF[egg][1]=1
4. For i in range(2-egg):
5.     For j in range(2-floor):
6.         EF[i][j]=EF[i][j-1]+EF[i-1][j-1]+1
7.         If( i==e and EF[i][j]>f)
```

8. Return j
9. Return f

说明：
上述代码的过程即为补齐下面二维矩阵的过程，直到补到 i 为题目给出的 egg 数目，并且 EF[i][j]>floor 时返回当前的 j，也就是尝试次数。

表二： 二维矩阵 2

I \ J	1	2	3	4	5	...
1	1	2	3	4	5	...
2	1	3	6	9	11	...
3	1	3	7	14	24	...
4	1	3	7	15	30	...
5	1	3	7	15	31	...
...	...	...	...	...	...	...

复杂度分析：
时间复杂度：O（E×F）
空间复杂度：O（E×F）

（3）优化 3
仔细观察表二中的二维矩阵，结合递推式：公式 2：$EF[i][j] = EF[i][j-1] + EF[i-1][j-1] + 1$，我们可以发现 EF[i][j]的变化只与 j-1 状态有关，我可以把二维数组的存储方式改进成不断更新的一维数组，而更新的次数就是尝试的次数 j。

伪代码：

Algorithm 5： Drop_Egg_DTGH_improve3

Input： num_of_egg(e), num_of_floor(f)
Output: drop_num
1. If(e == 1 || f == 0 || f == 1 || f == 2) return f
2. EF[e+1]并初始化为 EF[0]=0，其余为 1。
3. Count =1
4. While（EF[egg]<floor)
5. Count++
6. For i in range(egg -1)//注意是逆序的
7. EF[i]=EF[i-1] + EF[i] +1
8. Return count

复杂度分析：
时间复杂度：O（K×logF）。
空间复杂度：O（K）。

5. 小规模测试算法正确性

表三：小规模测试

规模 / 算法	2/8	15/1	10/5	19/9	3/5	6/6	2/8	2/2
Algorithm 1	2	15	4	5	2	3	2	2
Algorithm 2	2	15	4	5	2	3	2	2
Algorithm 3	2	15	4	5	2	3	2	2
Algorithm 4	2	15	4	5	2	3	2	2
Algorithm 5	2	15	4	5	2	3	2	2

解释：规模中，第一个数表示层数，第二个数表示鸡蛋数，比如 10/5 表示 10 层 1 个鸡蛋。

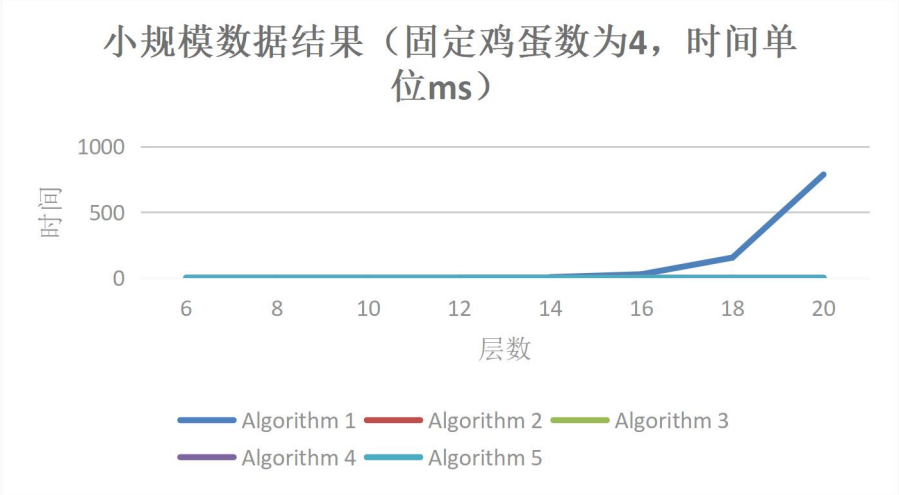
根据上表的测试，可以基本得出我的无法算法都是正确的。

6. 算法效率测试

• 小规模数据测试：

表四：小规模数据结果（固定鸡蛋数为 4，时间单位 ms）

规模 / 算法	6	8	10	12	14	16	18	20
Algorithm 1	0	0	0	1	5	27	154	787
Algorithm 2	0	0	0	0	1	0	0	0
Algorithm 3	0	0	0	0	0	0	0	0
Algorithm 4	0	0	0	0	0	0	0	0
Algorithm 5	0	0	0	0	0	0	0	0



图二：小规模数据结果

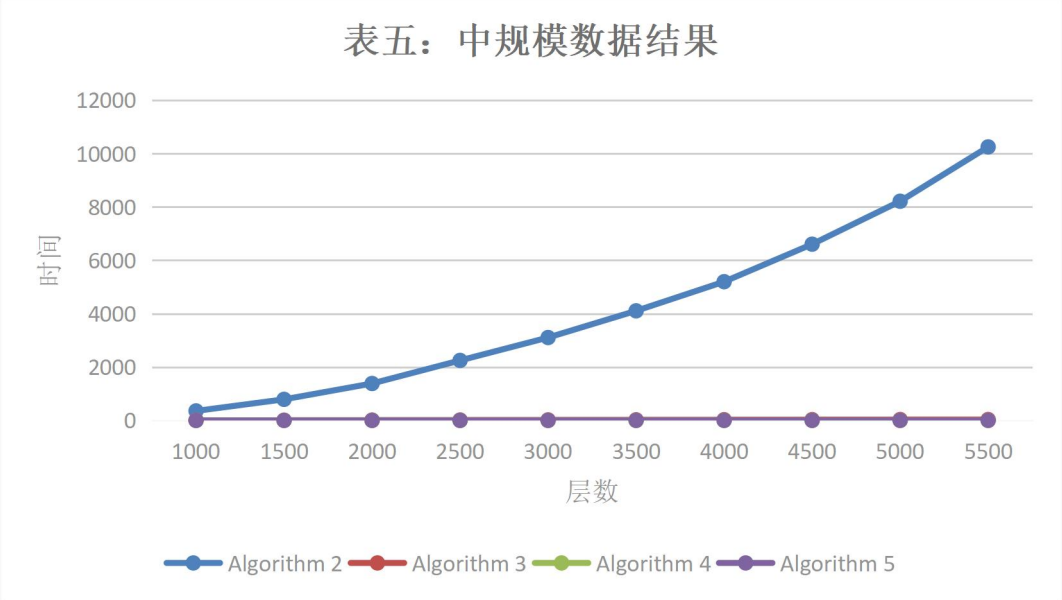
可以看出除了蛮力法的时间有明显变化外，其它算法的运行时间都低于毫秒级别。

• 中规模数据测试：

表五：中规模数据结果（固定鸡蛋数为 40，时间单位 ms）

规模 / 算法	1000	1500	2000	2500	3000
Algorithm 2	357	790	1384	2248	3106
Algorithm 3	5	9	13	16	20
Algorithm 4	1	1	2	2	2
Algorithm 5	0	0	0	0	0

规模 / 算法	3500	4000	4500	5000	5000
Algorithm 2	4106	5198	6603	8212	10247
Algorithm 3	23	27	30	33	33
Algorithm 4	3	3	4	4	4
Algorithm 5	0	0	0	0	0



图三：中规模数据结果

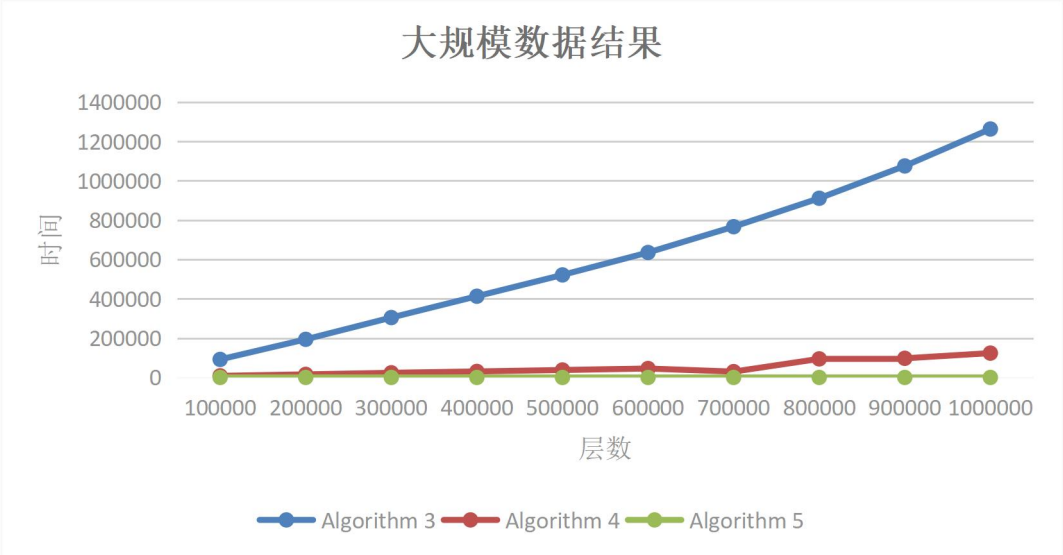
可以看出对比任意的优化后的动态规划算法，普通的动态规划算法时间效率都低了特别多。

• 大规模数据测试：

表六：大规模数据结果（固定鸡蛋数为 4000，时间单位 ms）

规模 / 算法	100000	200000	300000	400000	500000
Algorithm 3	91618	193763	304485	412906	521270
Algorithm 4	8156	15498	23675	30514	37832
Algorithm 5	0	0	0	0	0

规模 / 算法	600000	700000	800000	900000	1000000
Algorithm 3	635187	766406	910953	1075260	1263070
Algorithm 4	45314	59601	94390	97302	123596
Algorithm 5	0	0	0	0	0



图四：大规模数据结果

可以看出对比第一次优化，我的第二次和第三次优化都明显有了更好的时间效率。

7. 探索能测试的最大数据规模:

• 超大规模数据测试

表七：超大规模数据结果（固定鸡蛋数为 40000，时间单位 ms）

规模 / 算法	1×10^8	2×10^8	3×10^8	4×10^8	5×10^8
Algorithm 5	2	2	3	2	3

规模 / 算法	6×10^8	7×10^8	8×10^8	9×10^8	1×10^9
Algorithm 5	2	2	2	2	3

经过测试方向，我的算法 5，也就是第三次优化后的算法，在确定鸡蛋数是 40000 时，即使层数的规模很大，达到 10 亿层，也可以在 3ms 跑出结果。

表八：超大规模数据结果（固定鸡蛋数为 4×10^8，时间单位 s）

规模 / 算法	1×10^8	2×10^8	3×10^8	4×10^8	5×10^8
Algorithm 5	21376	21555	21905	22250	23106

规模 / 算法	6×10^8	7×10^8	8×10^8	9×10^8	1×10^9
Algorithm 5	23709	23060	22933	23318	23811

表九：超大规模数据结果（固定鸡蛋数为 1×10^9，时间单位 ms）

规模 / 算法	1×10^8	2×10^8	3×10^8	4×10^8	5×10^8

Algorithm 5	54242	55766	58892	57793	58882

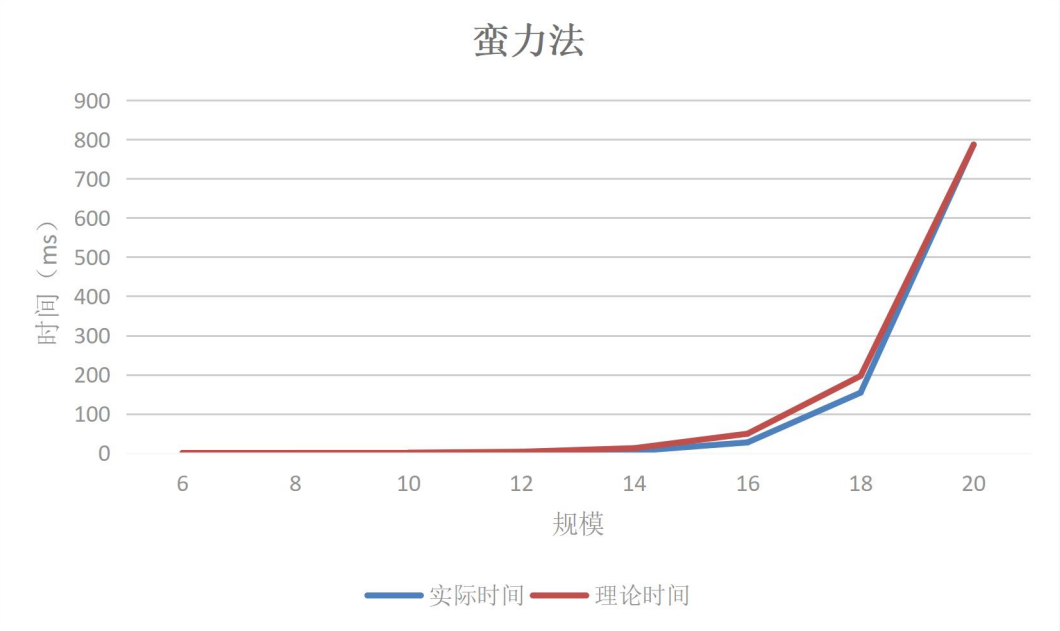
规模 / 算法	6×10^8	7×10^8	8×10^8	9×10^8	1×10^9
Algorithm 5	59955	60434	60075	59754	60689

最大规模的尝试：2.1×10^9个鸡蛋，2×10^9层楼。程序运行时间：127095ms
再大就超出了内存的限制。

从上面记录的测试结果也可以看出，当鸡蛋数一定时，楼层的增大对运行时间的影响并不会很大。
只有当鸡蛋数目增大了，运行时间才会有很大的改变。

8. 理论值实际值分析

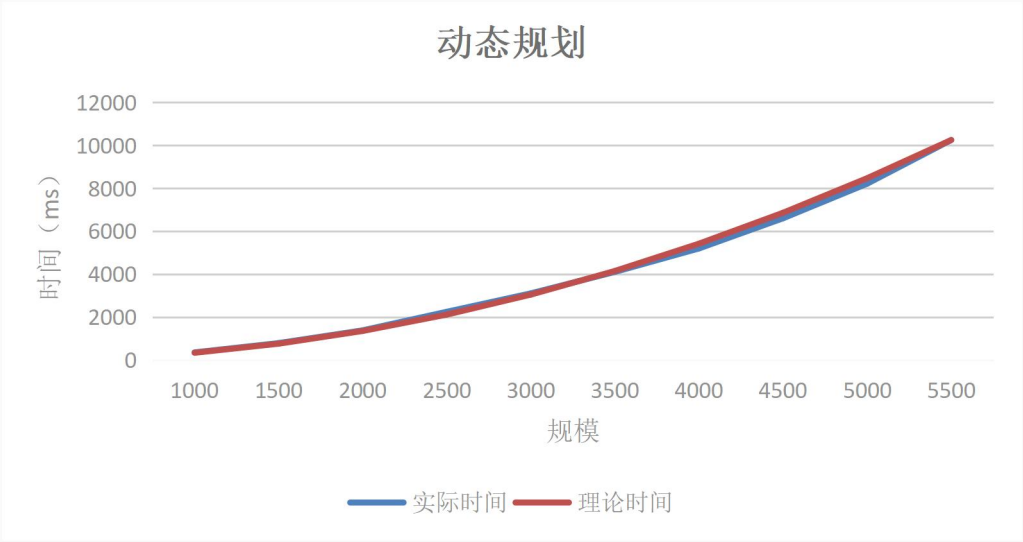
（1）蛮力法：



图五：蛮力法实际时间与理论时间对比（鸡蛋数为 4）

图五中，蓝色曲线是程序运行的实际时间，红色曲线是程序运行的理论时间，可以看出，程序运行的实际时间和理论时间基本拟合，并且大部分情况下，实际运行时间要低于理论运行时间。

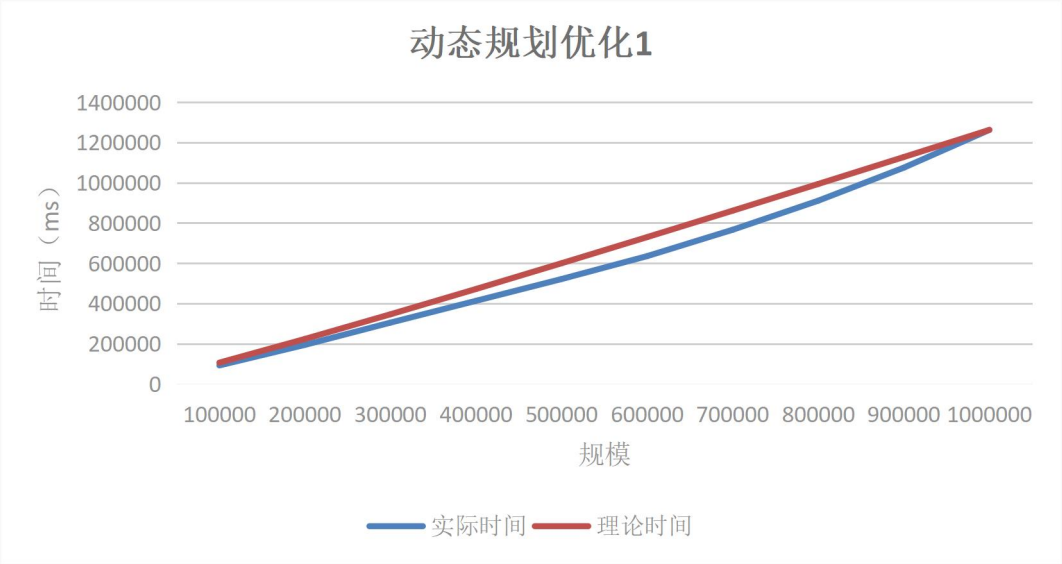
（2）动态规划



图六：动态规划实际时间与理论时间对比（鸡蛋数为 40）

图六中，蓝色曲线是程序运行的实际时间，红色曲线是程序运行的理论时间，可以看出，程序运行的实际时间和理论时间基本拟合，并且大部分情况下，实际运行时间要低于理论运行时间。

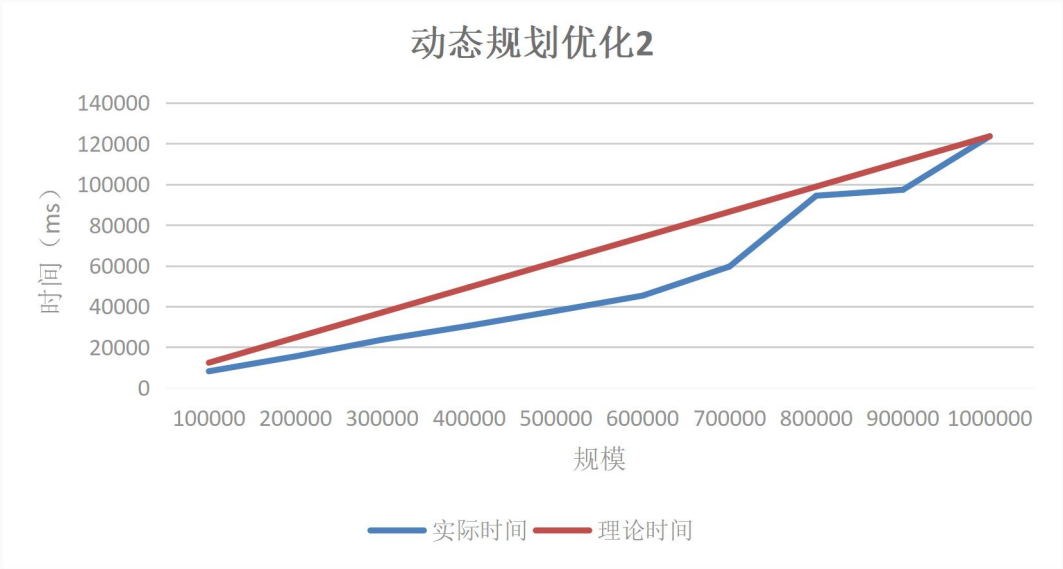
（3）动态规划优化 1



图七：动态规划优化 1 的实际时间与理论时间对比（鸡蛋数为 4000）

图七中，蓝色曲线是程序运行的实际时间，红色曲线是程序运行的理论时间，可以看出，程序运行的实际时间和理论时间有一定的差距，大部分情况下，实际运行时间要低于理论运行时间。

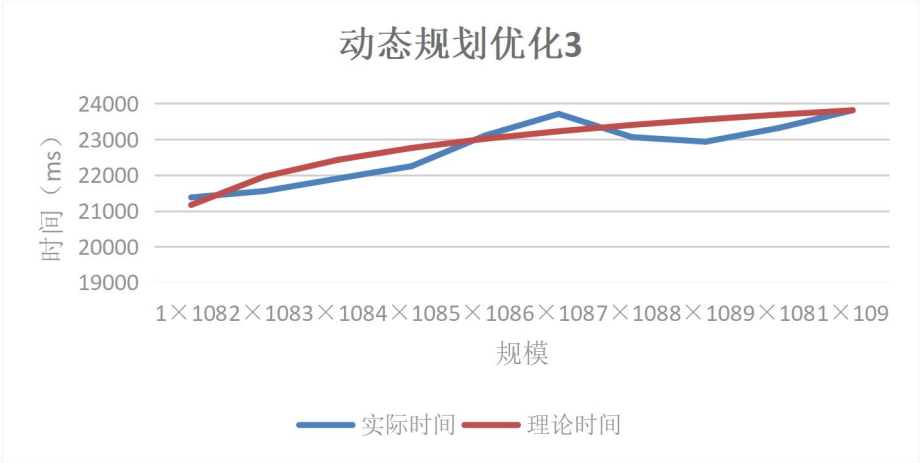
（4）动态规划优化 2



图八：动态规划优化 2 的实际时间与理论时间对比（鸡蛋数为 4000）

图八中，蓝色曲线是程序运行的实际时间，红色曲线是程序运行的理论时间，可以看出，程序运行的实际时间和理论时间有一定的差距，大部分情况下，实际运行时间要低于理论运行时间。

（5）动态规划优化 3



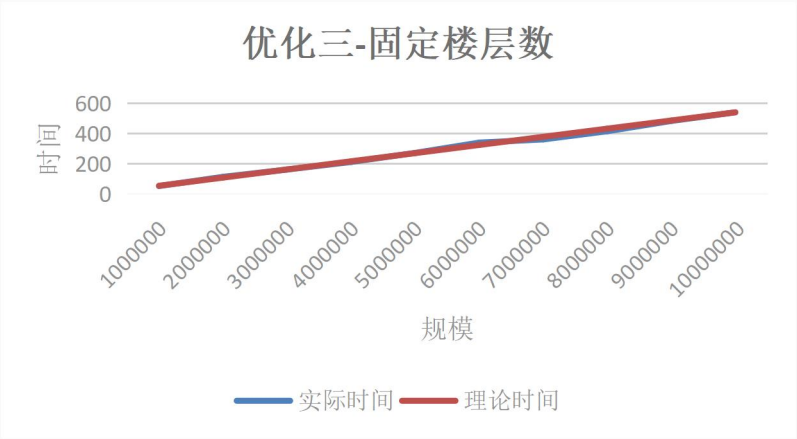
图九：动态规划优化 3 的实际时间与理论时间对比（鸡蛋数为 4×10^8）

图九中，蓝色曲线是程序运行的实际时间，红色曲线是程序运行的理论时间，可以看出，程序运行的实际时间和理论时间有差距，但是其实差距就是零点几秒的时间，是可以接受的。

如果确定层数为 2×10^8，当鸡蛋数改变时：

表十：超大规模数据结果（固定层数为 2×10^8，时间单位 ms）

规模 / 算法	1×10^6	2×10^6	3×10^6	4×10^6	5×10^6
Algorithm 5	52	113	161	211	271
规模 / 算法	6×10^6	7×10^6	8×10^6	9×10^6	10×10^6
Algorithm 5	339	360	414	482	539



图十：动态规划优化 3 的实际时间与理论时间对比（楼层数为 2×10^8）

可以看出实际时间和理论时间的拟合效果很好。

五、实验结果及分析：

本次实验，我共使用了蛮力法，动态规划法以及动态规划的三种改进方法对鸡蛋掉落问题进行求解。蛮力法的时间复杂度为指数级别的，对于稍微大一点规模的数据就很难在可以接受的时间内跑出结果了。之后使用了未优化的动态规划算法，使用动态规划方程:公式 1：$F(i,j) = Min(Max(F(i,j-x),F(i-1,x-1))+1)1 <= x <= f$。进行求解，可以把时间复杂度降低到 O（E×$F^2$),相比蛮力法，动态规划算法可以运行更大规模的数据了，但对于十万级别的数据，依旧很难在可接受的时间内运行出结果，之后采用了动态规划＋二分查找算法进行优化，把时间复杂都优化至 O（E×F×logF)，这样再一次减少了时间的开销，但是依旧没有办法在十万级别的数据集很快跑出结果，于是改变思路，使用新的动态规划方程：公式 2：$EF[i][j] = EF[i][j-1] + EF[i-1][j-1] + 1$，这样就把时间复杂度再次降低到了 O（E×F），在实际运行时也快了很多。最后，我又对空间进行了优化，把**空间复杂度优化到 O（E）**的同时，也把时间复杂度优化到了 O（E×logF），此次优化后，程序的运行速度有了质的飞跃，在大部分规模下运行时间甚至不需要 1ms，最后测试的极限为：**2.1×10^9个鸡蛋，2×10^9层楼，程序运行时间：127095ms，**如果不是受限与计算机的内容，在更大规模的数据下也是能跑出结果的。

之后我又对算法进行了效率对比以及理论值分析，每一次改进，算法对比前一版本的算法都有着非常明显的提升。之后的理论值分析中，大部分情况下，理论值和实际值都是可以拟合的，误差也在可接受的范围之内。大部分情况下，实际运行时间是要比理论运行时间要短的。

六、实验结论：

1.蛮力法的时间复杂度为指数级别，当楼层超过 27 层，就很难跑出结果。
2.动态规划（算法 2）的时间复杂度为 O（E×F×F），当楼层数目达到十万级别的时候很难跑出结果。
3.动态规划＋二分查找（算法 3）的时间复杂度为 O（E×F×logF），当楼层数达到十万级别时，可以跑出答案，但并不那么快速。
4.动态规划（算法 4）的时间复杂度为 O（E×F）相比与算法 3 有了很大的速度提升。
5.动态规划（算法 5）的时间复杂度为 O（ElogF），空间复杂度为 O（E），不过是时间复杂度还是空间复杂度都是最优的算法，它可以在超大规模的数据下运行出结果。
6.大部分情况下，实际运行时间都比理论值小。

指导教师批阅意见： **成绩评定：** **指导教师签字：** **2024 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。