优化，再优化！

——从《鹰蛋》一题浅析对动态规划算法的优化

安徽省芜湖市第一中学 朱晨光

目录

关键字

优化　动态规划　模型

摘要

本文就 Ural 1223《鹰蛋》这道题目介绍了五种性能各异的算法，并在此基础上总结了优化动态规划算法的本质思想及其一般方法。全文可以分为四个部分。
第一部分引言，阐明优化动态规划算法的重要性；
第二部分给出本文讨论的题目；
第三部分详细讨论这道题目五种不同的动态规划算法，并阐述其中的优化思想；
第四部分总结全文，再次说明对于动态规划进行优化的重要性，并分析优化动态规划的本质思想与一般方法。

正文

引言

在当今的信息学竞赛中，动态规划可以说是一种十分常用的算法。它以其高效性受到大家的青睐。然而，动态规划算法有时也会遇到时间复杂度过高的问题。因此，要想真正用好用活动态规划，对于它的优化方法也是一定要掌握的。

优化动态规划的方法有许多，例如四边形不等式、斜率优化等。但是这些方法只能对某些特定的动态规划算法进行优化，尚不具有普遍的意义。本文将就《鹰蛋》这道题目做较为深入的分析，并从中探讨优化动态规划的本质思想与一般方法。

问题

有一堆共 M 个鹰蛋，一位教授想研究这些鹰蛋的坚硬度 E。他是通过不断从一幢 N 层的楼上向下扔鹰蛋来确定 E 的。当鹰蛋从第 E 层楼及以下楼层落下时是不会碎的，但从第（E+1）层楼及以上楼层向下落时会摔碎。如果鹰蛋未摔碎，

还可以继续使用；但如果鹰蛋全碎了却仍未确定 E，这显然是一个失败的实验。教授希望实验是成功的。

例如：若鹰蛋从第 1 层楼落下即摔碎，E=0；若鹰蛋从第 N 层楼落下仍未碎，E=N。

这里假设所有的鹰蛋都具有相同的坚硬度。给定鹰蛋个数 M 与楼层数 N。要求最坏情况下确定 E 所需要的最少次数。

样例：

M=1，N=10

ANS=10

样例解释：为了不使实验失败，只能将这个鹰蛋按照从一楼到十楼的顺序依次扔下。一旦在第（E+1）层楼摔碎，E 便确定了。（假设在第（N+1）层摔鹰蛋会碎）

分析

算法一

乍一看这道题，算法并不十分明晰，因为这并不是简单的二分查找，还有对鹰蛋个数的限制。但由于这题是求最优值，我们便自然想到了动态规划。

状态定义即为用 i 个蛋在 j 层楼上最坏情况下确定 E 所需要的最少次数，记为 f(i,j)。

很显然，当层数为 0 时 f(i,j)=0，即 f(i,0)=0 (i>=0); 当鹰蛋个数为 1 时，为了不使实验失败，只能从下往上依次扔这个鹰蛋以确定 E，即 f(1,j)=j (j>=0).

下面是状态转移：

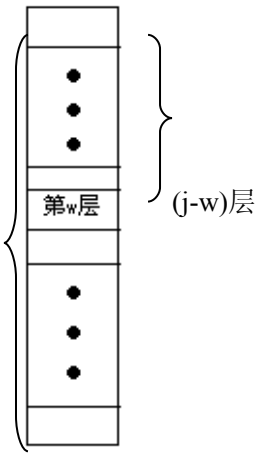

图 1

假设我们在第 w 层扔下鹰蛋，无外乎有两种结果：

①鹰蛋摔碎了，此时必有 E<w,我们便只能用（i-1）个蛋在下面的（w-1)层确定 E，并且要求最坏情况下次数最少，这是一个子问题，答案为 f(i-1,w-1),总次数便为 f(i-1,w-1)+1；

(w-1)层

②鹰蛋没摔碎，此时必有 E>=w .我们还能用这 i 个蛋在上面的（j-w)层确定 E。注意，这里的实验与在第 1~(j-w)层确定 E 所需次数是一样的，因为它们的实验方法与步骤都是相同的，只不过这(j-w)层在上面罢了。完全可以把它看成是对第 1~(j-w)层进行的操作。因此答案为 f(i,j-w),总次数便为 f(i,j-w)+1。（如图 1）

题目要求最坏情况下的最小值，所以这两种情况的答案须取较大值，且又要在所有决策中取最小值，所以有

f(i,j)=min{max{f(i-1,w-1),f(i,j-w)}+1|1<=w<=j}　①

很显然，所需鹰蛋个数必不会大于 N，因此当 M>N 时可令 M=N，且并不影响结果。所以这个算法的时间复杂度是 $O(MN^2)=O(N^3)$,空间复杂度是 O(N)（可用滚动数组优化）。

算法二

这个算法的时间复杂度太高，有没有可以优化的余地呢？答案是肯定的。首先，这题很类似于二分查找，即每次根据扔鹰蛋所得信息来决定下一步操作的区间，只不过对鹰蛋碎的次数有限制罢了。假设我们对于鹰蛋的个数不加限制，那么根据判定树的理论[1]，叶子结点个数共有(n+1)个，（E 的取值只可能是 0,1,2,…,n 共（n+1）种情况），则树的高度至少为$\lceil \log_2(n+1) \rceil$+1,即比较次数在最坏情况下需要$\lceil \log_2(n+1) \rceil$次。而我们又知道，在 n 个排好序的数里进行二分查找最坏情况下需要比较$\lceil \log_2(n+1) \rceil$次（在这个问题中，若未查找到可视为 E=0）。这两点便决定了$\lceil \log_2(n+1) \rceil$是下限而且是可以达到的下限。换句话说，

[1] 有关判定树的理论详情请见参考文献[1]第 292~293 页。

对于一个确定的 n，任意 M 所得到的结果均不会小于$\lceil \log_2(n+1) \rceil$。一旦 M>=$\lceil \log_2(n+1) \rceil$,该题就成了求二分查找在最坏情况下的比较次数，可以直接输出$\lceil \log_2(n+1) \rceil$。因此时间复杂度立即降为 $O(N^2\log_2 N)$.

由此可见，对于动态规划的优化是十分必要的。算法二仅通过考察问题自身的性质便成功地减少了状态总数，从而降低了算法一的时间复杂度，大大提高了算法效率。

算法三

然而优化还远未结束。经实践证明，M 的大小已经不能再加限制了，因此我们不妨从动态规划方程本身入手，探寻新的优化手段。

在实际操作中，我们发现 f(i,j)>=f(i,j-1) (j>=1)总是成立的。那么是否可以对该性质进行证明呢？是的。这里用数学归纳法加以证明。

当 i=1 时，由于 f(1,j)=j (j>=0),显然有 f(i,j)>=f(i,j-1)(j>=1)成立。

当 i>=2 时[2],f(i,0)=0,f(i,1)=max{f(i-1,0),f(i,0)}+1=1,即当 j=1 时，f(i,j)>=f(i,j-1)

现在假设当 j=k-1 时,f(i,j)>=-f(i,j-1)成立(k>1) ,则当 j=k 时,

f(i,j)=f(i,k)=min{max{f(i-1,w-1),f(i,k-w)}+1|1<=w<=k}
f(i,j-1)=f(i,k-1)=min{max{f(i-1,w-1),f(i,k-1-w)}+1|1<=w<=k-1}

当 1<=w<=k-1 时,∵k-1-w<k-w<=k-1,根据归纳假设，有 f(i,k-w)>=f(i,k-1-w),

∴max{f(i-1,w-1),f(i,k-w)}+1>=max{f(i-1,w-1),f(i,k-1-w)}+1

令 t=min{ max{f(i-1,w-1),f(i,k-w)}+1|1<=w<=k-1},

则有 t>=min{max{f(i-1,w-1),f(i,k-1-w)}+1|1<=w<=k-1}=f(i,k-1)　(1)

又 f(i,k)=min{t,max{f(i-1,k-1),f(i,0)}+1}=min{t,max{f(i-1,k-1),0}+1}

[2] 这里，我们按照 i 依次增大的顺序进行证明，即先证明 i =2 的情况，再证明 i=3 的情况，…… ,依此类推。

∵对于 1<=i<=n,0<=j<=m,f(i,j)>=0

∴f(i,k)=min{t,f(i-1,k-1)+1}

在(1)式中,令 w=k-1, =>f(i,k-1)<=max{f(i-1,k-2),f(i,0)}

+1=f(i-1,k-2)+1<=f(i-1,k-1)+1

即 f(i-1,k-1)+1>=f(i,k-1)，又 t>=f(i,k-1) ,

∴f(i,k)=max{t,f(i-1,k-1)+1}>=f(i,k-1)

即 f(i,j)>=f(i,j-1),命题得证。

所以有 f(i,j)>=f(i,j-1) (j>=1)　②

由此结论，可以在状态转移中使用二分法：

i) 若 f(i-1,w-1)<f(i,j-w),则对于 w'<w,必有 f(i,j-w')>=f(i,j-w)

∴max{f(i-1,w'-1),f(i,j-w')}+1>=f(i,j-w')+1>=f(i,j-w)+1=max{f(i-1,w-1),f(i,j-w)}+1

∴决策为 w'必无决策为 w 好；

ii) 若 f(i-1,w-1)>=f(i,j-w),则对于 w'>w,必有 f(i-1,w'-1)>=f(i-1,w-1),

∴max{f(i-1,w'-1),f(j,j-w')}+1>=f(i-1,w'-1)+1>=f(i-1,w-1)+1=max{f(i-1,w-1),f(i,j-w)}+1

∴决策为 w'必无决策为 w 好；

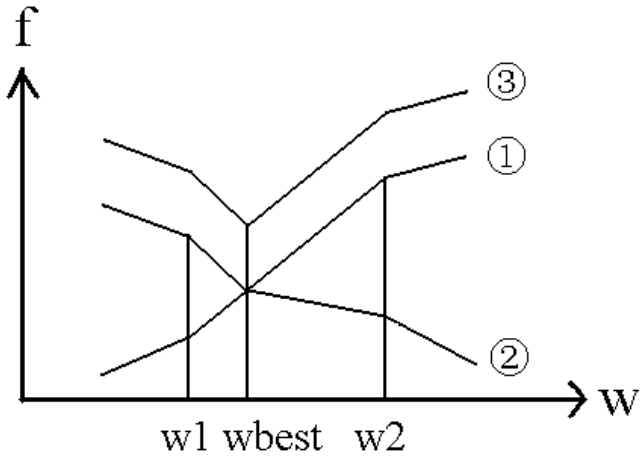

图 2

（如图 2，令①为 f(i-1,w-1)图象，②为 f(i,j-w)图象（皆用线段连结相邻两点）,③即为 max{f(i-1,w-1),f(i,j-w)}+1 的图象）

可见 w1 对应情况 i),w2 对应情况 ii)，最优取值即为 w=wbest 时的取值.

因此，在状态转移中，可以像二分查找那样，每次将 w 的取值范围缩小一半，这样就能在$\lceil \log_2(n+1) \rceil$次之内找到最佳的决策 wbest。

这样，根据 f(i,j)的单调性，我们又成功地将状态转移降为 O($\log_2$N)级，从而使算法的时间复杂度降至 O(N$\log_{\frac{2}{2}}$ N)级，已经可以解决 N 较大时的情况了。

从算法二到算法三的优化利用了动态规划函数 f(i,j)的单调性，成功地降低了状态转移部分的时间复杂度，使算法的效率又提高了一步。这说明研究算法本身同样可以找到优化的空间，为我们找到更低阶的算法创造了条件。

算法四

在对算法三进行研究之后，我们会萌生一个想法：既然现在 f(i,j)都需要求出，要想找到更高效的算法就只能从状态转移入手，因为这一步是 O($\log_2$N)，仍然不够理想。

这一步优化需要我们进一步挖掘状态转移方程：

f(i,j)=min{max{f(i-1,w-1),f(i,j-w)}+1|1<=w<=j}

很显然，f(i,j)<=max{f(i-1,w-1),f(i,j-w)}+1 (1<=w<=j)

令 w=1，则 f(i,j)<=max{f(i-1,0),f(i,j-1)}+1=f(i,j-1)+1

即 f(i,j)<=f(i,j-1)+1(j>=1) ③

又据②式，有 f(i,j-1)<=f(i,j)<=f(i,j-1)+1 (j>=1)④

这是一个相当重要的结论，由此可以得到一个推理:

若某个决策 w 可使 f(i,j)=f(i,j-1),则 f(i,j)=f(i,j-1)

若所有决策 w 都不能使 f(i,j)=f(i,j-1),则 f(i,j)=f(i,j-1)+1（且必存在这样的 w 使 f(i,j)=f(i,j-1)+1）

由此，我们设一指针 p,使 p 始终满足：

f(i,p)<f(i,j-1)且 f(i,p+1)=f(i,j-1)

很显然，f(i,p)=f(i,j-1)-1,

f(i,p+1)=f(i,p+2)=……=f(i,j-1)　(2)

在计算 f(i,j)时,我们令 p=j-w,则 w=j-p

令 tmp=max{f(i-1,w-1),f(i,j-w)}+1

则 tmp=max{f(i-1,j-p-1),f(i,p)}+1

若 f(i,p)>=f(i-1,j-p-1),则 tmp=f(i,p)+1=f(i,j-1)-1+1=f(i,j-1)

这说明当前决策 w 可以使 f(i,j)=f(i,j-1), ∴f(i,j)=f(i,j-1)

若 f(i,p)<f(i-1,j-p-1),则：

i) 当 p'<p 时,必有 f(i-1,j-p'-1)>=f(i-1,j-p-1)>f(i,p),

∴max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i-1,j-p'-1)+1>=f(i-1,j-p-1)+1>f(i,p)+1=f(i,j-1)

即 max{f(i,p'),f(i-1,j-p'-1)}+1>f(i,j-1)

max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i,j-1)+1,无法使 f(i,j)=f(i,j-1)

ii) 当 p'=p 时,

max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i-1,j-p'-1)+1>f(i,p')+1=f(i,p)+1=f(i,j-1)

同样无法使 f(i,j)=f(i,j-1)

iii) 当 p'>p 时,必有 f(i,p')>f(i,p),此时 f(i,p')=f(i,j-1)　(据(2)式)

所以 max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i,p')+1=f(i,j-1)+1,还是无法使 f(i,j)=f(i,j-1)

综上所述，当 f(i,p)<f(i-1,j-p-1)时，无论任何决策都不能使 f(i,j)=f(i,j-1),所以此时 f(i,j)=f(i,j-1)+1。

因此，我们只需根据 f(i,p)与 f(i-1,j-p-1)的大小关系便可直接确定 f(i,j)的取值[3],使状态转移成功地降为 O(1),算法的时间复杂度随之降至 $O(N\log_2 N)$。

从算法三到算法四，我们是从尚未完全优化的部分（即状态转移）入手，通过进一步挖掘动态规划方程中的特性，恰当地找到了一个可以用来求 f(i,j)的剖分点 p，使得状态转移部分的时间复杂度降为 O(1)，最终将算法的效率又提高了一步。

小结

从算法一到算法四，我们一共进行了三步优化，让我们先来小结一下：

算法一建立动态规划模型，这也是后面几个算法进行优化的基础；

算法二将动态规划中 M 的取值限定在$\lceil \log_2(n+1) \rceil$以内，这样就从状态总数方面优化了这个动态规划算法；

算法三利用 f(i,j)的单调性，改进了动态规划中的状态转移部分，提高了算法效率；

算法四挖掘出 f(i,j)所具备的另一个特殊性质，让状态转移部分的时间复杂度变为 O(1),把原来算法中不尽人意的地方进行了进一步的优化与改进。

这时我们会发现，经过了数次优化的动态规划模型已经不可能再有所改进了，对这题的讨论似乎可以到此为止了。但是，经过进一步思考，我们又找到了另一种动态规划模型，在这种模型下的算法五，可以将时间复杂度降为 $O(\sqrt{N}$

[3] 有两点注意事项：
① f(i,1)需特殊处理，即令 f(i,1)=1,因为 p 初值为 0，并不满足 f(i,p)=f(i,j-1)-1
②若 f(i,j)=f(i,j-1)+1,则将 p 赋值为 j-1

)。让我们来看一看算法五的精彩表现吧！

算法五

这里，我们需要定义一个新的动态规划函数 g(i,j)，它表示用 j 个蛋尝试 i 次在最坏情况下能确定 E 的最高楼层数。下面具体讨论 g(i,j).

很显然，无论有多少鹰蛋，若只试 1 次就只能确定一层楼， 即 g(1,j)=1 (j>=1)

而且只用 1 个鹰蛋试 i 次在最坏情况下可在 i 层楼中确定 E，即 g(i,1)=i (i>=1)

状态转移也十分简单，假设第一次在某一层楼扔下一只鹰蛋，且碎了，则在后面的(i-1)次里，我们要用(j-1)个蛋在下面的楼层中确定 E。为了使 g(i,j)达到最大，我们当然希望下面的楼层数达到最多，这便是一个子问题，答案为 g(i-1,j-1);假设第一次摔鹰蛋没碎，则在后面(i-1)次里，我们要用 j 个蛋在上面的楼层中确定 E，这同样需要楼层数达到最多，便为 g(i-1,j) (见图 3)。

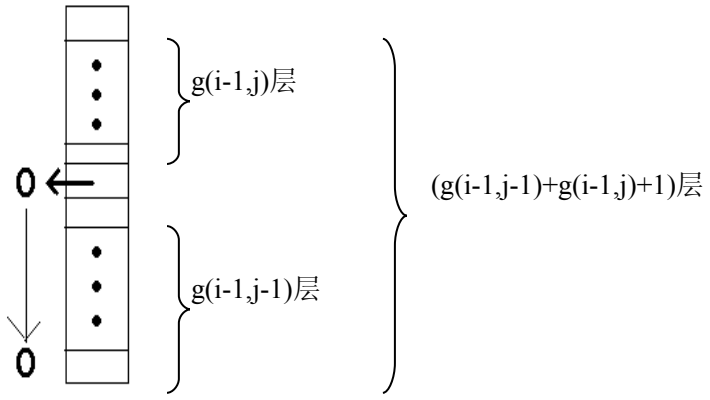

图 3

因此，有如下等式成立：

g(i,j)=g(i-1,j-1)+g(i-1,j)+1 ⑤

我们的目标便是找到一个 x，使 x 满足 g(x-1,M)<N 且 g(x,M)>=N ⑥，答案即为 x.[4]

这个算法乍一看是 O(N$\log_2$N)的，因为 i 是次数，最大为 N；j 为鹰蛋数，最大为 M, 即 $\log_2$N，状态转移为 O(1)，所以时间复杂度与状态数同阶，为 i*j，即 O(N$\log_2$N).但实际情况却并非如此，下面予以证明。

经过观察，我们很快会发现，函数 g(i,j)与组合函数 C_i^j 有着惊人的相似之处，让我们来比较一下（见下表）：

g(i,j)	C_i^j
g(1,j)=1(j>=1)	C_1^1 =1, C_1^j =0 (j>=2)
g(i,1)=i(i>=1)	C_i^1 =i (i>=1)
g(i,j)=g(i-1,j-1)+g(i-1,j)+1	$C_i^j = C_{i-1}^{j-1} + C_{i-1}^j$ (j<=i) , C_i^j =0 (j>i)

根据边界条件与递推公式，我们可以很容易用数学归纳法证明对于任意 i,j (i>=1,j>=1) 总有 g(i,j)>= C_i^j

又据⑥式，可以得到

C_{x-1}^M <=g(x-1,M)<N

即 C_{x-1}^M <N, $\frac{(x-1)(x-2)...(x-M)}{M!} < N$ (3)

这里介绍一个引理：当 1<=N<=3 或 N>=17 时，$(\log_2 N)^2$<N （用函数图象可以证明）

注：当 4<=n<=16 时, $(\log_2 N)^2$>=N，但由于相差很小，并不影响对于渐近复杂度[5]的分析

若 x<M，则 $xM<M^2<=(\log_2 N)^2<N$

[4] 若 x=1,须特殊判断

[5] 关于渐近复杂度的理论，详情请见参考文献[3]第 41~49 页。

若 x>=M，则根据(3)式,有$(x-M)^M$<=(x-1)(x-2)...(x-M)<N∑M!

x-M<=$\sqrt[M]{NM!}$ <=$\sqrt[M]{NM^M}$ =M$\sqrt[M]{N}$

∴x<=M+M$\sqrt[M]{N}$ =M(1+$\sqrt[M]{N}$)

∴xM<=M^2(1+$\sqrt[M]{N}$)　(4)

又∵N$^{\frac{M-1}{M}}$ ≈N,且 N>$(\log_2 N)^2$>=M^2 (引理)

∴N$^{\frac{M-1}{M}}$ >=M^2 (此性质对于 M=1 仍成立)

$\log_2$ N$^{\frac{M-1}{M}}$ >=$\log_2 M^2$

$\frac{M-1}{M}\log_2 N$>= $\log_2 M^2$

$\log_2 N - \frac{1}{M}\log_2 N$>= $\log_2 M^2$

$\frac{1}{M}\log_2 N$+ $\log_2 M^2$<=$\log_2 N$

$\log_2$ N$^{\frac{1}{M}}$ +$\log_2 M^2$<=$\log_2 N$

又 $\log_2$ N$^{\frac{1}{M}}$ ≈ $\log_2$ (N$^{\frac{1}{M}}$ +1)

∴$\log_2$ (N$^{\frac{1}{M}}$ +1)+$\log_2 M^2$<=$\log_2 N$

$\log_2$[(N$^{\frac{1}{M}}$ +1)M^2]<= $\log_2 N$

∴(N$^{\frac{1}{M}}$ +1) ∑M^2<= N,即 M^2 (1+$\sqrt[M]{N}$) <=N

又据(4)式，得 xM<= M^2 (1+$\sqrt[M]{N}$)<=N

综上所述，xM<=N

这就说明了 g(i,j)函数的实际运算量是 O(N)级的，那么又是怎样变为 O($\sqrt{N}$)的呢？

观察$\frac{(x-1)(x-2)...(x-M)}{M!}<N$，可得 (x-1)(x-2)…(x-M)<NM!

这可以大致得出当 M 不太大时，x 与$\sqrt[M]{N}$是同阶的。在实际情况中，可以发现只有当 M=1 时，x=N, xM=N ;当 M>1 时，xM 立即降至($\sqrt{N}$)的级别。因此，

只需要在 M=1 时特殊判断一下就可以使算法的时间复杂度降为 O($\sqrt{N}$)了，空间复杂度可用滚动数组降为 O(M),即 O($\log_2 N$)。

在新的动态规划模型之下，我们找到了一个比前几种算法都优秀得多的方法。这就提醒我们不要总是拘泥于旧的思路。换个角度来审视问题，往往能收到奇效。倘若我们仅满足于算法四，就不能打开思路，找到更高效的解题方法。可见多角度地看问题对于动态规划的优化也是十分重要的。

总结

本文就《鹰蛋》一题谈了五种性能各异的算法，这里做一比较：（见下表）

算法编号	时间复杂度	空间复杂度	优化方法
算法一	$O(N^3)$	O(N)	
算法二	$O(N^2\log_2 N)$	O(N)	考察问题自身的性质，减少状态总数
算法三	$O(N\log_{\frac{2}{2}} N)$	O(N)	研究动态规划方程，找出其中的特性，优化状态转移部分
算法四	$O(N\log_2 N)$	O(N)	进一步挖掘动态规划方程的特性，从而再次降低状态转移部分的时间复杂度
算法五	O($\sqrt{N}$)	$O(\log_2 N)$	建立新的动态规划模型，从另一个角度重新审视问题

从这张表格中，我们可以很明显地看出优化能显著提高动态规划算法的效率。并且，优化动态规划的方法也是多种多样的。这就要求我们在研究问题时必须深入探讨，大胆创新，永不满足，不断改进，只有这样才能真正将优化落到实处。在实际问题中，尽管优化手段千变万化，但万变不离其宗，其本质思想都是找到动态规划算法中仍然不够完美的部分，进行进一步的改进；或是另辟蹊径，建立新的模型，从而得到更高效的算法。而在具体的优化过程中，需要我们从减少状态总数、挖掘动态规划方程的特性、降低状态转移部分的时间复杂度以及建立新模型等几方面入手，不断完善已知算法。这便是优化动态规划算法的一般方法。

当然，世上的任何事物都是既有普遍性，又有特殊性的。当我们用一般方法难以解决的时候，使用特殊方法（如四边形不等式、斜率优化等）也是不错的选择。因此，只有将一般方法与特殊方法都灵活掌握，才能真正高效地解决动态规划的优化问题。

结束语

本文仅是讨论了对于动态规划算法进行优化的本质思想及一般方法，实际上，优化思想极其重要，也无处不在。优化可以使原本效率低下的算法变为一个非常优秀的算法，可以使能够解决的问题规模扩大几十倍，几百倍，乃至成千上万倍。而更重要的是，优化思想的应用是极为广泛的。无论是在信息学竞赛中，还是在日常的生产生活中，优化思想都发挥着十分重要的作用。具备了优化思想，我们就不会满足于现有的方法，而会不断地开拓、创新，去创造出更好、更优秀的方法。

优化，再优化，就是为了让算法得到进一步的完善，同时也开阔了我们的思维，锻炼了我们深入分析研究问题的能力，培养了我们不断进取的精神，对今后进一步的科学研究也是大有益处的。

本文仅对一道信息学竞赛的题目做了粗浅的分析，希望能够起到抛砖引玉的作用，让我们进一步了解优化思想的重要作用，用好用活优化思想，以便更好、更高效地解决问题。

参考文献

[1] 严蔚敏 吴伟民 ，1992，《数据结构（第二版）》。 北京：清华大学出版社 。

[2] 吴文虎 王建德，1997，《信息学奥林匹克竞赛指导——组合数学的算法与程序设计(PASCAL 版)》。 北京：清华大学出版社 。

[3] Thomas H.Cormen Charles E.Leiserson Ronald L.Rivest Clifford Stein, 2001 ，《Introduction to Algorithms, Second Edition》. The MIT Press.

[4] Ural Online Judge acm.timus.ru

原题网页 http://acm.timus.ru/problem.aspx?space=1&num=1223

（本文为了讨论方便，对题目某些部分进行了改动）

附录

1、本文讨论原题：

Chernobyl' Eagle on a Roof

Time Limit: 1.0 second

Memory Limit: 1 000 КБ

Once upon a time an Eagle made a nest on the roof of a very large building. Time went by and some eggs appeared in the nest. There was a sunny day, and Niels Bohr was walking on the roof He suddenly said: "Oops! All eggs surely have the same solidity, thus there is such non-negative number *E* that if one drops an egg from the floor number *E*, it will not be broken (and so for all the floors below the *E*-th), but if one drops it from the floor number *E*+1, the egg will be broken (and the same for every floor higher, than the *E*-th). Now Professor Bohr is going to organize a series of experiments (i.e. drops). The goal of the experiments is to determine the constant *E*. It is evident that number *E* may be found by dropping eggs sequentially floor by floor from the lowest one. But there are other strategies to find *E* for sure with much less amount of experiments. You are to find the least number of eggs droppings, which is sufficient to find number *E* for sure, even in the worst case. Note that dropped eggs that are not broken can be used again in following experiments.

The number of floors is a positive integer and a number *E* is a non-negative one. They both do not exceed 1000. The floors are numbered with positive integers starting from 1. If an egg hasn't been broken even being dropped from the highest floor, one can suppose, that *E* is also determined and equal to the total number of floors.

Input

Input contains multiple test cases. Each line is a test case. Each test case consists of two numbers separated with a space: first the number of eggs, and then the number of floors. Tests will end with the line containing a single zero.

Output

For each test case output in a separate line the minimal number of experiments, which Niels Bohr will have to make even in the worst case.

Sample Input

1 10

2 5

0

Sample Output

10

3

2、相关程序

算法一　eagle_1.cpp　(由于时间复杂度过高，在 Ural Online Judge 上超时)

算法二　eagle_2.cpp

算法三　eagle_3.cpp　} 均在 Ural Online Judge 上测试通过

算法四　eagle_4.cpp

算法五　eagle_5.cpp

1) eagle_1.cpp

```
#include<iostream.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
  else return(b);
}

void work()
{
 long i,j,w,temp;
 for (i=2; i<=eggnum; i++)
  {
```

```
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    {
     f[now][j]=maxnum;
     for (w=1; w<=j; w++)
      {
       temp=max(f[old][w-1],f[now][j-w])+1;
       if (temp<f[now][j])
        f[now][j]=temp;
      }
    }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
{
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   init();
   work();
   output();
  }
 return 0;
}
```

2）eggle_2.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000
```

```
long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
  else return(b);
}

void work()
{
 long i,j,w,temp;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    {
     f[now][j]=maxnum;
     for (w=1; w<=j; w++)
      {
       temp=max(f[old][w-1],f[now][j-w])+1;
       if (temp<f[now][j])
        f[now][j]=temp;
      }
    }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}
```

```
int main()
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
          init();
          work();
          output();
         }
  }
 return 0;
}
```

3) eggle_3.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
```

```
  else return(b);
}

void calc(long i,long j)
{
 long w,temp,start,stop,mid;
 f[now][j]=maxnum;
 start=1; stop=j;
 while (start<=stop)
  {
   mid=(start+stop)/2;
   if (f[old][mid-1]>f[now][j-mid])
    {
     if (f[old][mid-1]+1<f[now][j])
      f[now][j]=f[old][mid-1]+1;
     stop=mid-1;
    }
    else if (f[old][mid-1]<f[now][j-mid])
         {
          if (f[now][j-mid]+1<f[now][j])
           f[now][j]=f[now][j-mid]+1;
          start=mid+1;
         }
         else {
              f[now][j]=f[now][j-mid]+1;
              return;
             }
  }
}

void work()
{
 long i,j;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    calc(i,j);
  }
}

void output()
```

```
{
 cout<<f[now][n]<<endl;
}

int main()
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         work();
         output();
        }
  }
 return 0;
}
```

4) eggle_4.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

void work()
```

```
{
 long i,j,p;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   p=f[now][0]=0;
   f[now][1]=1; //special!!! In case of mistake
   for (j=2; j<=n; j++)
    if (f[now][p]>=f[old][j-p-1])
     f[now][j]=f[now][j-1];
     else {
          f[now][j]=f[now][j-1]+1;
          p=j-1;
         }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
          init();
          work();
          output();
         }
  }
 return 0;
}
```

5) eggle_5.cpp

```
#include<iostream.h>
#include<math.h>
#define maxlogn 20
#define maxnum 1000000000

long n,eggnum,now,old,g[maxlogn+1];

void init()
{
 long i;
 now=1;
 for (i=1; i<=eggnum; i++)
  g[i]=1;
}

void work()
{
 long i,j,p;
 for (i=2; i<=n; i++)
  {
   for (j=eggnum; j>=2; j--)
    {
     g[j]=g[j-1]+g[j]+1;
     if ((j==eggnum)&&(g[j]>=n))
      {
       cout<<i<<endl;
       return;
      }
    }
   g[1]=i;
   if ((eggnum==1)&&(g[1]>=n))
    {
     cout<<i<<endl;
     return;
    }
  }
}

int main()
{
 long temp;
 while (1)
```

```
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         if(g[eggnum]>=n)
          cout<<1<<endl;
          else if (eggnum==1)
               cout<<n<<endl;
               else work();
        }
  }
 return 0;
}
```