```
            } else if (cn > 0) {
                cn = next[cn];
            } else {
                next[pos++] = 0;
            }
        }
        return next;
    }
```

getNextArray 方法中的 while 循环就是求解 nextArr 数组的过程，现在证明这个循环发生的次数不会超过 $2M$ 这个数量。先来看两个量，一个为 pos 量，一个为（pos-cn）的量。对 pos 量来说，从 2 开始又必然不会大于 match 的长度，即 pos<M。对（pos-cn）量来说，pos 最大为 M-1，cn 最小为 0，所以(pos-cn)<=M。

循环的第一个逻辑分支会让 pos 的值增加，(pos-cn)的值不变。循环的第二个逻辑分支为 cn 向左跳的过程，所以会让 cn 减小，pos 值在这个分支中不变，所以（pos-cn）的值会增加。循环的第三个逻辑分支会让 pos 的值增加，(pos-cn)的值也增加。如下表所示：

	Pos	pos-cn
循环的第一个逻辑分支	增加	不变
循环的第二个逻辑分支	不变	增加
循环的第三个逻辑分支	增加	增加

因为 pos+(pos-cn)<$2M$，又有上表的关系，所以循环发生的总体次数小于 pos 量和（pos-cn）量的增加次数，也必然小于 $2M$，证明完毕。

所以整个 KMP 算法的复杂度为 $O(M)$(求解 nextArr 数组的过程)+$O(N)$（匹配的过程），因为有 $N \geqslant M$，所以时间复杂度为 $O(N)$。

丢棋子问题

【题目】

一座大楼有 0~N 层，地面算作第 0 层，最高的一层为第 N 层。已知棋子从第 0 层掉落肯定不会摔碎，从第 i 层掉落可能会摔碎，也可能不会摔碎($1 \leqslant i \leqslant N$)。给定整数 N 作为楼层数，再给定整数 K 作为棋子数，返回如果想找到棋子不会摔碎的最高层数，即使在最差的情况下扔的最少次数。一次只能扔一个棋子。

【举例】

N=10，K=1。

返回 10。因为只有 1 棵棋子，所以不得不从第 1 层开始一直试到第 10 层，在最差的情况下，即第 10 层是不会摔坏的最高层，最少也要扔 10 次。

N=3，K=2。

返回 2。先在 2 层扔 1 棵棋子，如果碎了，试第 1 层，如果没碎，试第 3 层。

N=105，K=2

返回 14。

第一个棋子先在 14 层扔，碎了则用仅存的一个棋子试 1～13。

若没碎，第一个棋子继续在 27 层扔，碎了则用仅存的一个棋子试 15～26。

若没碎，第一个棋子继续在 39 层扔，碎了则用仅存的一个棋子试 28～38。

若没碎，第一个棋子继续在 50 层扔，碎了则用仅存的一个棋子试 40～49。

若没碎，第一个棋子继续在 60 层扔，碎了则用仅存的一个棋子试 51～59。

若没碎，第一个棋子继续在 69 层扔，碎了则用仅存的一个棋子试 61～68。

若没碎，第一个棋子继续在 77 层扔，碎了则用仅存的一个棋子试 70～76。

若没碎，第一个棋子继续在 84 层扔，碎了则用仅存的一个棋子试 78～83。

若没碎，第一个棋子继续在 90 层扔，碎了则用仅存的一个棋子试 85～89。

若没碎，第一个棋子继续在 95 层扔，碎了则用仅存的一个棋子试 91～94。

若没碎，第一个棋子继续在 99 层扔，碎了则用仅存的一个棋子试 96～98。

若没碎，第一个棋子继续在 102 层扔，碎了则用仅存的一个棋子试 100、101。

若没碎，第一个棋子继续在 104 层扔，碎了则用仅存的一个棋子试 103。

若没碎，第一个棋子继续在 105 层扔，若到这一步还没碎，那么 105 便是结果。

【难度】

校 ★★★☆

【解答】

方法一。假设 $P(N,K)$的返回值是 N 层楼有 K 个棋子在最差情况下扔的最少次数。

1. 如果 N==0，也就是楼层只有第 0 层，那不用试，肯定不碎，即 $P(0,K)=0$。

2. 如果 K==1，也就是楼层有 N 层，但只有 1 个棋子了，这时只能从第 1 层开始试，

一直试到第 N 层，即 $P(N,1)=N$。

3．以上两种情况较为特殊，对一般情况($N>0$，$K>1$)，我们需要考虑第 1 个棋子从哪层楼开始扔一次，如果第 1 个棋子从第 i 层开始扔，有以下两种情况：

1）碎了。那么可以知道，没有必要去试第 i 层以上的楼层，接下来的问题就变成了还剩下 i-1 层楼，还剩下 K-1 个棋子，所以总步数为 1+P(i-1,K-1)。

2）没碎。那么可以知道，没有必要去试第 i 层以下的楼层，接下来的问题就变成了还剩下 N-i 层楼，仍有 K 个棋子，所以总步数为 1+P(N-i,K)。

根据题意，在 1)和 2)中哪个是最差的情况，最后的取值就应该来自哪个，所以最后取值为 max{ P(i-1,K-1), P(N-i,K) } + 1。那么 i 可以选择哪些值呢？从 1 到 N 都可以选择，这就是说，第 1 个棋子丢在哪里呢？从第 1 层到第 N 层都可以试试，那么在这么多尝试中，我们应该选择哪个尝试呢？应该选择最终步数最少的那种情况。所以，P(N,K)=min{max{P(i-1,K-1),P(N-i,K)}(1<=i<=N)}+1。具体请参看如下代码中的 solution1 方法。

```
public int solution1(int nLevel, int kChess) {
        if (nLevel < 1 || kChess < 1) {
                return 0;
        }
        return Process1(nLevel, kChess);
}

public int Process1(int nLevel, int kChess) {
        if (nLevel == 0) {
                return 0;
        }
        if (kChess == 1) {
                return nLevel;
        }
        int min = Integer.MAX_VALUE;
        for (int i = 1; i != nLevel + 1; i++) {
                if (i == nLevel) {

                }
                min = Math.min(min,
                                Math.max(Process1(i - 1, kChess - 1),
                                                Process1(nLevel - i, kChess)));
        }
        return min + 1;
}
```

方法一为暴力递归的方法，如果楼数为 N，将尝试 N 种可能。在下一步的递归中，楼数最多为 N-1，将尝试 N-1 种可能，所以时间复杂度为 $O(N!)$，这个时间复杂度非常高。

方法二，动态规划方法。通过研究如上递归函数我们发现，$P(N,K)$过程依赖 $P(0..N-1,K-1)$

和 $P(0..N\text{-}1,K)$。所以，若把所有递归过程的返回值看作是一个二维数组，可以用动态规划的方式优化整个递归过程，从而减少递归重复计算，如下所示：

dp[0][K] = 0，dp[N][1] = N，dp[N][K] = min{max{dp[i-1][K-1], dp[N-i][K]} (1<= i<=N) } +1。

动态规划的具体过程参看如下代码中的 solution2 方法。

```
public int solution2(int nLevel, int kChess) {
        if (nLevel < 1 || kChess < 1) {
                return 0;
        }
        if (kChess == 1) {
                return nLevel;
        }
        int[][] dp = new int[nLevel + 1][kChess + 1];
        for (int i = 1; i != dp.length; i++) {
                dp[i][1] = i;
        }
        for (int i = 1; i != dp.length; i++) {
                for (int j = 2; j != dp[0].length; j++) {
                        int min = Integer.MAX_VALUE;
                        for (int k = 1; k != i + 1; k++) {
                                min = Math.min(min,
                                        Math.max(dp[k - 1][j - 1], dp[i - k][j]));
                        }
                        dp[i][j] = min + 1;
                }
        }
        return dp[nLevel][kChess];
}
```

求每个位置(a,b)（即 $P(a,b)$）的过程中，需要枚举 $P(0..a\text{-}1,b)$和 $P(0..a\text{-}1,b\text{-}1)$，所以每个位置枚举过程的时间复杂度为 $O(N)$。递归过程，即 $P(i,j)$，i 从 0 到 N，j 从 0 到 K，所以用一张 $N\times K$ 的二维表可以表示所有递归过程的返回值，即一共有 $O(N\times K)$个位置。所以方法二整体的时间复杂度为 $O(N^2\times K)$。

方法三，把方法二的额外空间复杂度从使用 $N\times K$ 的矩阵，减少为 2 个长度为 N 的数组。分析动态规划的过程我们发现，dp[N][K]只需要它左边的数据 dp[0..N-1][K-1]，和它上面一排的数据 dp[0..N-1][K]。那么在动态规划计算时，就可以用两个数组不停地复用的方式实现，而并不真的需要申请整个二维数组的空间。具体请参看如下代码中的 solution3 方法。

```
public int solution3(int nLevel, int kChess) {
    if (nLevel < 1 || kChess < 1) {
        return 0;
    }
    if (kChess == 1) {
        return nLevel;
```

```
        }
        int[] preArr = new int[nLevel + 1];
        int[] curArr = new int[nLevel + 1];
        for (int i = 1; i != curArr.length; i++) {
            curArr[i] = i;
        }
        for (int i = 1; i != kChess; i++) {
            int[] tmp = preArr;
            preArr = curArr;
            curArr = tmp;
            for (int j = 1; j != curArr.length; j++) {
                int min = Integer.MAX_VALUE;
                for (int k = 1; k != j + 1; k++) {
                    min = Math.min(min, Math.max(preArr[k - 1], curArr[j - k]));
                }
                curArr[j] = min + 1;
            }
        }
        return curArr[curArr.length - 1];
    }
```

方法二和方法三的时间复杂度为 $O(N^2 \times K)$，还是很高。但我们注意到，求解动态规划表中的值时，有枚举过程，此时往往可以用“四边形不等式”及其相关猜想来进行优化。

优化的方式——四边形不等式及其相关猜想：

1．如果已经求出了 k+1 个棋子在解决 n 层楼时的最少步骤(dp[n][k+1])，那么如果在这个尝试的过程中发现，第 1 个棋子扔在 m 层楼的这种尝试最终导致了最优解。则在求 k 个棋子在解决 n 层楼时(dp[n][k])，第 1 个棋子不需要去尝试 m 层以上的楼。

举一个例子，3 个棋子在解决 100 层楼时，第 1 个棋子扔在 37 层楼时最终导致了最优解。那么 2 个棋子在解决 100 层楼时，第 1 个棋子不需要去试 37 层楼以上的楼层。

2．如果已经求出了 k 个棋子在解决 n 层楼时的最少步骤(dp[n][k])，那么如果在这个尝试的过程中发现，第 1 个棋子扔在 m 层楼的这种尝试最终导致了最优解。则在求 k 个棋子在解决 n+1 层楼时(dp[n+1][k])，不需要去尝试 m 层以下的楼。

举一个例子，2 个棋子在解决 10 层楼时，第 1 个棋子扔在 4 层楼时最终导致了最优解。那么 2 个棋子在解决 11 层楼或更多的层楼时（想象一下 100 层），第 1 个棋子也不需要去试 1、2、3 层楼，只用从 4 层及其以上的楼层试起。

也就是说，动态规划表中的两个参数分别为棋子数和楼数，楼数变多之后，第 1 个棋子的尝试楼层的下限是可以确定的。棋子数变少之后，第 1 个棋子的尝试楼层的上限也是可以确定的。这样就省去了很多无效的枚举过程。证明略。注：“四边形不等式”的相关内容及其证明是相当复杂而烦琐的，本书由于篇幅所限，不再进行进一步的展开，有兴趣的

读者可以搜集相关资料进行深入学习。本书是想用本题给面试者提一个醒，如果在面试时发现某一道面试题解法是动态规划，但在计算动态规划二维表的过程中，发现计算每一个值时有类似本题和本书的“画匠问题”、“邮局选址问题”这样的枚举过程，则往往可以通过“四边形不等式”的优化把时间复杂度降一个维度，可以从 $O(N^2 \times k)$或 $O(N^3)$降到 $O(N^2)$。具体过程请参看如下代码中的 solution4 方法。

```
public int solution4(int nLevel, int kChess) {
        if (nLevel < 1 || kChess < 1) {
                return 0;
        }
        if (kChess == 1) {
                return nLevel;
        }
        int[][] dp = new int[nLevel + 1][kChess + 1];
        for (int i = 1; i != dp.length; i++) {
                dp[i][1] = i;
        }
        int[] cands = new int[kChess + 1];
        for (int i = 1; i != dp[0].length; i++) {
                dp[1][i] = 1;
                cands[i] = 1;
        }
        for (int i = 2; i < nLevel + 1; i++) {
                for (int j = kChess; j > 1; j--) {
                        int min = Integer.MAX_VALUE;
                        int minEnum = cands[j];
                        int maxEnum = j == kChess ? i / 2 + 1 : cands[j + 1];
                        for (int k = minEnum; k < maxEnum + 1; k++) {
                            int cur = Math.max(dp[k - 1][j - 1], dp[i - k][j]);
                            if (cur <= min) {
                                min = cur;
                                cands[j] = k;
                            }
                        }
                        dp[i][j] = min + 1;
                }
        }
        return dp[nLevel][kChess];
}
```

最优解。最优解比以上各种方法都要快。首先我们换个角度来看这个问题，以上各种方法解决的问题是 N 层楼有 K 个棋子最少扔多少次。现在反过来看 K 个棋子如果可以扔 M 次，最多可以解决多少层楼这个问题。根据上文实现的函数可以生成下表。在这个表中记为 map，map[i][j]的意义为 i 个棋子扔 j 次最多搞定的楼数。

0　1　2　3　4　5　6　7　8　9　10　->　次数

1 0 1 2 3 4 5 6 7 8 9 10

2 0 1 3 6 10 15 21 28 36 45 55

3 0 1 3 7 14 25 41 63 92 129 175

4 0 1 3 7 15 30 56 98 162 255 385

5 0 1 3 7 15 31 62 119 218 381 637

|

V

棋子数

通过研究 map 表我们发现，第一横排的值从左到右依次为 1，2，3，...，第一纵列都为 0，除此之外的其他位置(i,j)，都有 map[i][j]==map[i][j-1]+map[i-1][j-1]+1。

如何理解这个公式呢？假设 i 个棋子扔 j 次最多搞定 m 层楼，“搞定最多”说明每次扔的位置都是最优的且棋子肯定够用的情况，假设第 1 个棋子扔在 a 层楼是最优的尝试。

1．如果第 1 个棋子已碎，那就向下，看 i-1 个棋子扔 j-1 次最多搞定多少层楼。

2．如果第 1 个棋子没碎，那就向上，看 i 个棋子扔 j-1 次最多搞定多少层楼。

3．a 层楼本身也是被搞定的 1 层。

1、2、3 的总楼数就是 i 个棋子扔 j 次最多搞定的楼数，map 表的生成过程极为简单，同时数值增长极快。原始问题可以用 map 表得到很好的解决，比如，想求 5 个棋子搞定 200 层楼最少扔多少次的问题。注意到第 5 行（表示 5 个棋子的情况）第 8 列（表示扔 8 次的情况）对应的值为 218，是第 5 行的所有值中第一次超过 200 的值，则可以知道 5 个棋子搞定 200 层楼最少扔 8 次。同时在 map 表中其实 9 列 10 列的值也完全可以不需要计算，因为算到第 8 列（即扔 8 次）就已经搞定，那么时间复杂度也可以进一步得到优化。另外还有一个特别重要的优化，我们知道 N 层楼完全用二分的方式扔 log$N$+1 次就可以确定哪层楼是会碎的最低层楼，所以当棋子数（k）大于 log$N$+1 时，我们就可以直接返回 log$N$+1。

如果棋子数为 K、楼数为 N，最终的结果为 M 次，那么最优解的时间复杂度为 $O(K\times M)$，在棋子数大于 log$N$+1 时，时间复杂度为 $O(\log N)$。在只有一个棋子的时候，$K\times M$ 等于 N，在其他情况下，$K\times M$ 比 N 要小得多。最优解求解过程参看如下代码中的 solution5 方法。

```
public int solution5(int nLevel, int kChess) {
        if (nLevel < 1 || kChess < 1) {
                return 0;
        }
        int bsTimes = log2N(nLevel) + 1;
        if (kChess >= bsTimes) {
                return bsTimes;
```

```
        }
        int[] dp = new int[kChess];
        int res = 0;
        while (true) {
                res++;
                int previous = 0;
                for (int i = 0; i < dp.length; i++) {
                        int tmp = dp[i];
                        dp[i] = dp[i] + previous + 1;
                        previous = tmp;
                        if (dp[i] >= nLevel) {
                                return res;
                        }
                }
        }
}

public int log2N(int n) {
        int res = -1;
        while (n != 0) {
                res++;
                n >>>= 1;
        }
        return res;
}
```

画匠问题

【题目】

给定一个整型数组 arr，数组中的每个值都为正数，表示完成一幅画作需要的时间，再给定一个整数 num 表示画匠的数量，每个画匠只能画连在一起的画作。所有的画家并行工作，请返回完成所有的画作需要的最少时间。

【举例】

arr=[3,1,4]，num=2。

最好的分配方式为第一个画匠画 3 和 1，所需时间为 4。第二个画匠画 4，所需时间为 4。因为并行工作，所以最少时间为 4。如果分配方式为第一个画匠画 3，所需时间为 3。第二个画匠画 1 和 4，所需的时间为 5。那么最少时间为 5，显然没有第一种分配方式好。所以返回 4。

arr=[1,1,1,4,3]，num=3。

最好的分配方式为第一个画匠画前三个 1，所需时间为 3。第二个画匠画 4，所需时间