问题描述:

https://leetcode-cn.com/problems/super-egg-drop/comments/

参阅：https://leetcode-cn.com/problems/super-egg-drop/solution/shuang-bai-shu-xue-fa-dai-ma-by-xiao-ming-199/

887. 鸡蛋掉落

难度 困难 342

你将获得 `K` 个鸡蛋，并可以使用一栋从 `1` 到 `N` 共有 `N` 层楼的建筑。

每个蛋的功能都是一样的，如果一个蛋碎了，你就不能再把它掉下去。

你知道存在楼层 `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*移动*，你可以取一个鸡蛋（如果你有完整的鸡蛋）并把它从任一楼层 `X` 扔下（满足 `1 <= X <= N`）。

你的目标是**确切地**知道 `F` 的值是多少。

无论 `F` 的初始值如何，你确定 `F` 的值的最小移动次数是多少？

示例 1：

```
输入：K = 1, N = 2
输出：2
解释：
鸡蛋从 1 楼掉落。如果它碎了，我们肯定知道 F = 0 。
否则，鸡蛋从 2 楼掉落。如果它碎了，我们肯定知道 F = 1 。
如果它没碎，那么我们肯定知道 F = 2 。
因此，在最坏的情况下我们需要移动 2 次以确定 F 是多少。
```

示例 2：

```
输入：K = 2, N = 6
输出：3
```

解题思路

动态规划算法；可惜第一种方法是超时的，因为时间复杂度为 $O(KN^2)$ 太大了。读了很多份讲解后明白了，可以利用函数单调性实现二分法来降低复杂度。

代码实现

```
#include<iostream>
#include<stdio.h>
#include<vector>
#include<queue>
#include<string>
using namespace std;


class Solution {
public:
    int superEggDrop(int K, int N) {
        vector<vector<int>> dp(N + 1, vector<int>(K + 1, 0));
        for (int i = 0; i <= N; i++)
            dp[i][1] = i;             //只有一个鸡蛋时需要遍历
        for (int i = 1; i <= N; i++)         //想求出dp[N][K] 需要自底向上,外层循环为
楼层数
        {
```

```
            for (int k = 2; k <= K; k++)    //内层循环为鸡蛋数。同时，只针对鸡蛋数求移
动次数的循环也需要自底向上
            {
                int res = INT_MAX;
                for (int j = 1; j <= i; j++)    //对固定楼层，固定鸡蛋数的次数求取；
                {
                    res = min(res, max(dp[j - 1][k - 1], dp[i - j][k]) + 1);
                }
                dp[i][k] = res;                 //将值存入数组
            }
        }
        return dp[N][K];
    }
};

int main()
{

    Solution S;
    cout << S.superEggDrop(2, 4);

    return 0;
}
```

二分法取代最内层循环的O(n)复杂度搜索：

```
int superEggDrop(int K, int N) {
        vector<vector<int>> dp(N + 1, vector<int>(K + 1, 0));
        for (int i = 0; i <= N; i++)
            dp[i][1] = i;           //只有一个鸡蛋时需要遍历
        for (int i = 1; i <= N; i++)        //想求出dp[N][K] 需要自底向上,外层循环为
楼层数
        {
            for (int k = 2; k <= K; k++)    //内层循环为鸡蛋数。同时，只针对鸡蛋数求移
动次数的循环也需要自底向上
            {
                int res = INT_MAX;
                int lo = 1, hi = i,mid;
                while (lo <= hi)
                {
                    mid = (lo + hi) / 2;

                    int broken = dp[mid - 1][k - 1];
                    int not_broken = dp[i - mid][k];
                    if (broken > not_broken)
                    {
                        hi = mid - 1;
                        res = min(res, broken + 1);
                    }
                    else
                    {
                        lo = mid + 1;
                        res = min(res, not_broken + 1);
                    }
                }
                dp[i][k] = res;
            }
```

```
        }
        return dp[N][K];
}
```

PS:

最强的：逆向思维法，也被称为数学法。逆向思考本题，在T次操作,K个鸡蛋下，所能确定的最多的层数为N。即：函数$F(T,K)=N$。那么本题中，给出N,K值后只需要去遍历T值，取整即可。

我们这里定义的这个$F(T,K)=N$表示的是, T次操作，K个鸡蛋，可以确定$0-N$层中无论哪一层是题中的临界层：$F(N\geq F\geq 0)$，T次操作和K个鸡蛋都可以找出这个F。即$F(T,K)_{可以确定}N_{层楼的情况}$

这样思考的原理是：只去思考在某一层扔出一个鸡蛋的后果是什么：

- 当只有一个鸡蛋的时候,操作次数就代表着所能确定的最高层数；
- 当只有一步操作时，无论多少个鸡蛋也只能确定一层；

除去前面的两种极端情况，我们的目标是用动态规划的方法确定$F(T,K)$的动态方程。

通常扔出一个鸡蛋会发生什么呢：在$F(T,K)$已经将N确定的情况下，在0-N的任意一层扔出鸡蛋：

- 假如鸡蛋碎了，那么这层之下的所有层数为$F(T-1,K-1)$；T-1次操作和K-1个鸡蛋可以完全将其确定
- 假如鸡蛋没碎，那么这层之上的所有层数为$F(T-1,K)$;同上；

即有动态方程：

- $F(T,K)=F(T-1,K-1)+F(T-1,K)+1$
- 之所以是相加的关系：我们上面是一种假设，类似分类讨论，不管是碎还是不碎，不管在哪一层抛出鸡蛋，都会符合上面的等式，才可以确定$F(T,K)$是真的将所有0-N层楼里不论哪一层楼是临界值。都可以确定出来。这就是这个想法的重点，为了想通这一步我花了很久的时间。

有了上面的基础我们可以写如下代码：

```
class Solution {
public:
    int superEggDrop(int K, int N) {
        //vector<vector<int>> dp(N + 1, vector<int>(K + 1, 0));
        if (N == 1)
            return 1;
        vector<vector<int>>f(N + 1, vector<int>(K + 1,0));
        for (int i = 1; i <= K; i++)
            f[1][i] = 1;            //即只有一个操作时，只能确定一层楼
        int i = 2;
        for(;i<=N;i++)
        {
            for (int j = 1; j <= K; j++)
                f[i][j] = 1 + f[i - 1][j - 1] + f[i - 1][j];
            if(f[i][K]>=N)
                return i;
        }
        return i;
    }
};
```

参阅别人更加优秀的代码：0ms 下图

```
class Solution {
public:
    int superEggDrop(int K, int N) {
        vector<vector<int>>dp(K + 1, vector<int>(N + 1, 0));
        for (int i = 0; i <= N; i++) dp[0][i] = 0;  //没鸡蛋时：0层
        for (int i = 0; i <= K; i++) dp[i][0] = 0;  //无操作次数时：0层
        int ans = 0;
        while (dp[K][ans] < N)
        {
            ans++;
            for (int i = 1; i <= K; i++)
                dp[i][ans] = 1 + dp[i - 1][ans - 1] + dp[i][ans - 1];
        }
        return ans;
    }
};
// 取外层循环为：次数，内层循环为鸡蛋数：道理相同，但是相比上法更清晰简洁
// 在我看来二者时间复杂度相同，也不知道为什么差别甚远
//即使我简化为如下函数：仍然差别甚远：最好116ms
class Solution {
public:
    int superEggDrop(int K, int N) {
        vector<vector<int>>f(N + 1, vector<int>(K + 1,0));
        int i = 0;
        while(f[i][K]<N)
        {
            i++;
            for (int j = 1; j <= K; j++)
                f[i][j] = 1 + f[i - 1][j - 1] + f[i - 1][j];
        }
        return i;
    }
};

//分析两函数差，问题出在vector容器的初始化上，建立vector容器比建立正常数组要快速的多
//故应该将较大的N去容纳入vector 使较小的K 去用去做一维索引，在调换二者位置后，第二种方法速度可达12ms

//再进行优化：当添加两行初始化时，速度更快
class Solution {
public:
    int superEggDrop(int K, int N) {
        vector<vector<int>>f(K + 1, vector<int>(N + 1,0));
        for (int i = 0; i <= N; i++) f[0][i] = 0;   //没鸡蛋时：0层
        for (int i = 0; i <= K; i++) f[i][0] = 0;   //无操作次数时：0层
        int i = 0;
        while(f[K][i]<N)
        {
            i++;
            for (int j = 1; j <= K; j++)
                f[j][i] = 1 + f[j][i - 1] + f[j - 1][i-1];
        }
        return i;
    }
};
```

```
//我认为是先用O(N+K)的操作使得内存中先存入了很多的变量，在运行时速度更快，可至0ms
```