# Chapter 4

## 4.3 Money exchange 问题分析

dp[i][j]

*Use only 0…i-1 to consist of aim ‘j’, 🡪 dp[i-1][j]*

*Use 0…i-1 and 1 sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-array[i]] + 1*

*Use 0…i-1 and 2 sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-2\*array[i]] + 2*

*Use 0…i-1 and 3 sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-3\*array[i]] + 3*

*…*

*Use 0…i-1 and k sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-k\*array[i]] + k*

*(j-k\*array[i] >= 0, k >=0)*

…

So, min\_sheets = min{ dp[i-1][j], dp[i-1][j-k\*array[i]] + k}, (j-k\*array[i] >= 0, k >=1)

Use 0…i-1 and (t+1) sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-(t+1)\*array[i]] + (t+1)

(j-(t+1)\*array[i] >= 0, (t+1) >=1),

Which also is,

Use 0…i-1 and (t+1) sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j-t\*array[i] - array[i]] + t+1

(j-(t+1)\*array[i] >= 0, t >= 0)

i.e.

*Use only 0…i-1 to consist of aim ‘j-array[i]’, 🡪 dp[i-1][j-array[i]]*

*Use 0…i-1 and 1 sheet of array[i] to consist of aim ‘j-array[i]’ 🡪 dp[i-1][j-array[i] - array[i]] + 1*

*Use 0…i-1 and 2 sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j- array[i]-2\*array[i]] + 2*

*Use 0…i-1 and 3 sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j- array[i]-3\*array[i]] + 3*

*…*

*Use 0…i-1 and k sheet of array[i] to consist of aim ‘j’ 🡪 dp[i-1][j- array[i]-t\*array[i]] + t*

*(j-t\*array[i] >= 0, t >=0)*

So indeed this is dp[i][j-array[i]], which is to say,

dp[i-1][j-t\*array[i] - array[i]] + t+1 (j-(t+1)\*array[i] >= 0, t >= 0), equals,

dp[i][j-array[i]].

Finally,

min\_sheets = min{ dp[i-1][j], dp[i][j-array[i]]}

## 4.6 汉诺塔问题

进阶问题分析：

根据汉诺塔的程序解法

**void** **hanoi**(**int** n, **const** ***std***::string& from, **const** ***std***::string& help, **const** ***std***::string& to) {

**if**(n <= 0) {

**return**;

}

**if**(n == 1) {

***std***::cout << "Move " << n << " from " << from << " to " << to << ***std***::**endl**;

**return**;

}

// Move top 'n-1' plate from 'from' to 'help' with the help of 'to'

hanoi(n-1, from, to, help);

// Move the last one, which is 'n' from 'from' to 'to' directly with help of 'help'

hanoi(1, from, help, to);

// Move the 'n-1' plates from 'help' to 'to' with the help of 'from'

hanoi(n-1, help, to, from);

}

根据解法，总的移动次数为

S(N) = S(N-1) + 1 + S(N-1)

即，第一个步骤（N-1个圆盘从from移动到to，辅助为help）的移动次数，第二个步骤移动次数（就是1次），第三个步骤的移动次数（N-1个圆盘从help移动到to，辅助为from）

故有，S(N)+1=2\*[S(N-1)+1]，即为等比数列（通项公式为）。

所以，又为1（即只有一个圆盘的情况），所以.

下面分析1~i个圆盘已经移动次数，

1. 第i个圆盘在from上，那么有可能第一个步骤已完成，或者未完成，要看剩下的  
   1~i-1个圆盘的状态
2. 第i个圆盘在to上，那么第一个步骤肯定已完成，而且第二个步骤也已经完成，那么已经移动的次数是，剩下的移动次数要看剩下的1~i-1个圆盘的状态
3. 第i个圆盘在help上，这在最优的轨迹当中是不可能出现的，如果出现了，那么就不说话最优的轨迹。

# Chapter 2

## 2.6 Josephus Problem约瑟夫环问题分析

分析：

转化为0~n-1个人，从0开始报数，报到m-1时，此人（即第m个人）退出。m可以小于、大于或者等于n。

所以，0, 1, 2, …, n-2, n-1中，第一个退出的人是(m-1)%n，其之后第一个人是m%n，设  
k=(m-1)%n为其之后的第一个人。

第一个人出列之后，队伍编号为

**0, 1, 2, 3, … , k-3, k-2, k, k+1, k+2, …, n-1**

注意因为退出的人是k-1，所以上面没有k-1。

因为自k开始，要从0重新报数，所以调整队列如下：

**k, k+1, k+2, …, n-1, 0, 1, 2, 3, … , k-3, k-2**

所以，原来的k报数为0，原来的k+1报数为1，k+2报数为2，以此类推，直到n-1报数应为n-k-1（X – 0 + 1 = (n-1) – k + 1, X = n-k-1）。

那么自然，如果还没有人退出队列，原来的0报数为n-k，1报数为n-k+1，直到最后k-2报数为n-2。如下

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Q1** | **k** | **k+1** | **k+2** | **…** | **n-1** | **0** | **1** | **2** | **3** | **…** | **k-3** | **k-2** |
| **Q2** | **0** | **1** | **2** | **…** | **n-k-2** | **n-k** | **n-k+1** | **n-k+2** | **n-k+3** | **…** | **n-3** | **n-2** |

明显Q2就是0~n-2从0开始报数，并且报到m个数（从0开始，报到m-1）的情况。

那么就采用递推的办法解决问题。

假设，Nn是0~n-1个人报m个数的最后剩余者的下标（**在最初数组中的原始下标**），

Nn-1是0~n-2个人报m个数的最后剩余者的下标（**在已退出一个人的情况下的数组中的下标**）。

需要注意的是，Nn和Nn-1实际上是同一个人，只是在每次有人退出队列后重新分配下标的情况下，下标有所不同而已。

分情况，

1. 如果最后剩下的那个人，在k,k+1,k+2,…,n-1这个序列中，那么Nn = Nn-1 + k
2. 如果最后剩下的那个人，在0,1,2,3,…,k-3,k-2这个序列中，那么Nn = (Nn-1 + k)%n，其中，该式可以使用归纳法证明。

又（1）中的任意Nn-1满足Nn = (Nn-1 + k)%n（亦可有归纳法推出）。

所以由（1）（2）得出Nn = (Nn-1 + k)%n

而k=m%n，故有k<n，所以有

再将k=m%n代入上式，得

由此退出了递推式。

# Chapter 9

## 9.12 调整[0,x)区间上数出现的概率

当要生成x2概率时，想要返回[0, x)区间上的数，两次调用Math.rand()的返回值必须落在[0, x)区间上，否则会返回大于x的值。

当调用k次Math.rand()之后，若最大值是0 <= max <=x，你们说明k次调用每次返回值都是v，并且0 <= v <=x。而对于任意一次调用，生成v的概率是p，并且0 <= p <=x。所以k次调用产生的概率就是xk。