动态规划的各种模型

谢秋锋

长沙市长郡中学

目录

01 背包

有 n 个物品，每个物品只有一个，第 i 个物品的体积为 $w[i]$，价值为 $c[i]$。你有一个容量为 V 的背包，你希望在背包能够装下的情况下，装走价值之和尽可能大的物品。

01 背包

设 $f[i][v]$ 表示考虑到第 i 个物品时，已经装了容积 v 的最大价值。
初始时 $f[0][0] = 0$，其他值为负无穷，每次转移的时候枚举前面的容积和当前物品选或者不选。
即状态转移方程为：

$$f[i][v] = max\{f[i-1][v], f[i-1][v-w[i]] + c[i]\}$$

其中 $f[i-1][v]$ 表示的是当前第 i 个物品不选，那么已经装进背包里的体积就不会变。
$f[i-1][v-w[i]] + c[i]$ 则是表示当前第 i 个物品被装进背包，既然其被装进了背包，且装完后背包中的体积为 v，那么前 $i-1$ 个物品的体积和就是 $v - w[i]$。
复杂度为 $O(nV)$。

如果直接按照状态转移方程，我们可以写出一个时间和空间复杂度均为 $O(nV)$ 的方法。
事实上，空间复杂度可以优化到 $O(V)$。我们可以将第一维省略，设 $f[v]$ 表示当前已经装了体积为 v 的物品时的最大价值。
我们依然要按照一定次序枚举所有物品，注意这里的 $f[v]$ 表示的应该是使用我们枚举的这些物品装了 v 体积的最大价值。
看看之前的状态转移方程：

$$f[i][v] = max\{f[i-1][v], f[i-1][v-w[i]] + c[i]\}$$

如果我们忽略掉第一维重写这个转移方程则是：

$$f[v] = max\{f[v], f[v-w[i]] + c[i]\}$$

01 背包

注意如果这里从小往大枚举 v 进行转移的话，由于较小的 v 已经在之前就进行了修改，此时按照这个方法转移的话出现了后效性。解决的方法是我们强制较小的 v 只能在较大的 v 转移完成后才能转移。我们只需要从大往小枚举体积就行了。
于是核心代码为：

```
1 for(int i=1;i<=n;++i)
2     for(int j=V;j>=w[i];--j)
3         f[j]=max(f[j],f[j-w[i]]+c[i]);
```

完全背包

有 n 种物品，每种物品有无限个，每种物品有一个容积 $w[i]$ 和一个价值 $c[i]$。你希望在不超过背包体积的情况下放入价值尽可能大的物品。

完全背包

状态与 01 背包一致，设 $f[i][v]$ 表示考虑了前 i 个物品，它们装了体积为 v 时的最大价值。
想一想我们之前 01 背包说的转移出现的后效性，这个后效性出现的原因是因为较小体积如果被当前这个物品更新之后，继续用较小体积转移，则意味着这个物品就会被用多次。
而在完全背包中，我们恰好希望的就是每个物品能够被用到多次，即在完全背包的情况下，原本对于 01 背包的后效性，恰好方便了完全背包的转移。
那么核心代码为：

```
1 for(int i=1;i<=n;++i)
2     for(int j=w[i];j<=V;++j)
3         f[j]=max(f[j],f[j-w[i]]+c[i]);
```

多重背包

有 n 种物品，第 i 个物品有 a_i 个，每个物品都有它自己的价值 $c[i]$ 和体积 $w[i]$。你有一个体积为 V 的背包。你希望在背包装的下的情况下，让放入背包的物品价值和最大。

多重背包

如果我们把 a_i 个相同的物品看成 a_i 个价值为 $c[i]$，体积为 $w[i]$ 的物品的话，那么就转变成了 01 背包。此时复杂度变成了 $O(V * \sum a_i)$。

有一种更加好的做法——二进制拆分。

我们把一个 a_i 拆成 $1 + 2 + 4 + ... + 2^k + x_i$ 个物品的集合。不难证明用这些二进制与最终的余项 x 一定能够表示出 $0 - a_i$ 中的任意一个整数。于是拆分之后转化成 01 背包问题就解决了。

这样的复杂度为 $O(V * log(\sum a_i))$

混合背包

有 n 种物品，有些物品有个数限制，其中第 i 个物品有 a_i；另外一些物品没有个数限制。每个物品都有它自己的价值 $c[i]$ 和体积 $w[i]$。你有一个体积为 V 的背包。你希望在背包装的下的情况下，让放入背包的物品价值和最大。

混合背包

首先将多重背包的部分全部转化为 01 背包，于是问题变成了 01 背包和完全背包的混合。
那么只需要先处理完 01 背包的部分，再用已经得到的 dp 数组继续做完全背包就行了。

最长上升子序列 (LIS)

有一个长度为 n 的数列，第 i 个数为 a_i，求这个数列的最长上升子序列长度。

最长上升子序列 (LIS)

设 $f[i]$ 表示从左往右考虑到了 i，且 a_i 这个数必定被选入子序列时的最长上升子序列长度。
考虑如何转移，如果 $j < i$ 且 $a_j < a_i$，那么必定可以直接把 a_i 放在 a_j 后面形成一个更长的上升子序列。
于是得到了转移：

$$f[i] = \max_{j<i \& a_j<a_i} f[j] + 1$$

时间复杂度为 $O(n^2)$。初值为 $f[0] = 0$。

最长上升子序列 (LIS)

那么我们可不可以继续优化我们的过程呢?
先考虑我们的 $O(n^2)$ 做法：设 $f[i]$ 表示从左往右考虑到了 i，且 a_i 这个数必定被选入子序列时的最长上升子序列长度。
转移是：

$$f[i] = \max_{j<i \& a_j<a_i} f[j] + 1$$

现在的问题就是如何优化找 j 的过程。

方法一：单调性 + 二分

我们额外考虑一个 b 数组，假设当前转移的是位置 i，那么 $b_l = \min_{j<i\&f[j]=l} \{a_j\}$。也即，到目前的 i 位置为止，所有长度为 l 的最长上升子序列的末尾元素值的最小值。

现在证明 b 数组是严格单调的。考虑反证，如果存在 $x < y$，且 $b[x] >= b[y]$，那么我们在长度为 y 的 LIS 里面删掉最后的 $y - x$ 个数，就成为了一个长度为 x 的 LIS，且这个长度为 x 的 LIS 的末尾元素值一定严格小于长度为 y 的 LIS 的末尾元素值，也就是我们找到了一个长度为 x 的 LIS，它的末尾元素值 $< b[y]$，矛盾，故假设不成立。所以 b 数组必定是严格单调的。

接下来考虑怎么算答案: 首先在当前的 b 数组中二分答案，根据前面所说，数组内必定是严格单调的，所以我们二分到满足 $b[k] < a_i \leq b[k+1]$ 的 k 进行转移，此时 $f[i] = k+1$，之后用 $a[i]$ 更新 $b[k+1]$。如果 $b[k+1]$ 之前已经存在，只需要取 min 即可(实际上就是直接令 $b[k+1] = a[i]$)，否则令 $b[k+1] = a[i]$。也就是说，二分完成之后，我们只需要令 $b[k+1] = a[i]$ 即可。
这样便做到了时间复杂度 $O(nlogn)$

我们来模拟一下这个过程：
比如：2 5 3 4 1 7 6。假设我们考虑到了第 3 位，即 $i=3$
前面的 f 值是 1 2，b 数组的值是 2 5。
我们二分找到的 $k=1$，所以 $f[3]=k+1=2$，然后
$b[k+1]=a[3]$，之后 b 数组变成了 2 3。
然后 $i=4$，二分找到的 $k=2$，那么 $f[4]=3$，$b[3]$ 原先不存在，
于是把它直接赋值为 $a[4]=4$。也就是 b 数组变成了 2 3 4。

方法二：数据结构

还是考虑 $O(n^2)$ 的做法，我们将原来的所有数据全部离散化 (缩小值域，方便使用数据结构)。
由转移 $f[i] = \max_{j<i\&a_j<a_i} f[j] + 1$，我们从左往右，只将每一个处理完的数据丢进数据结构，那么 $j < i$ 的限制便不用考虑。
我们对于值域开数据结构，对于每个值，我们存下，到当前为止，以这个值结尾的 LIS 的最大长度。即考虑一个数组 c，其中 $c[x] = \max_{j<i\&a[j]=x} \{f[j]\}$，也就是以 x 结尾的 LIS 中最长的长度。

方法二：数据结构

对于当前的 i，我们需要找到以比 $a[i]$ 小的数字结尾的 LIS 中最长的那个，然后把 $a[i]$ 接在后面，也就是，我们只需要求出 $c[1..a[i]-1]$ 的最大值，然后 $f[i]$ 就是这个最大值 $+1$。
算出 $f[i]$ 之后，又多了一个长度为 $f[i]$ 的、以 $a[i]$ 结尾的 LIS，因此，$c[a[i]] = \max\{c[a[i]], f[i]\}$。
我们要做的事情可以总结为：区间求 $\max$，单点修改，可以用线段树维护。时间复杂度为 $O(nlogn)$

最长公共子序列

给定两个数列，一个长度为 n 的 $\{a_n\}$ 和一个长度为 m 的 $\{b_m\}$。求这两个数列的最长公共子序列。

最长公共子序列

设 $f[i][j]$ 表示在 i 数列上考虑到了第 i 个位置，在 j 数列上考虑到了第 j 个位置时的最长公共子序列。
考虑转移：

$$f[i][j] = \max \begin{cases} f[i][j-1] \\ f[i-1][j] \\ f[i-1][j-1]+1 & a_i = b_j \end{cases}$$

时间复杂度为 $O(nm)$。初值为 $f[0][0] = 0$

[洛谷 P2758] 编辑距离

设 A 和 B 是两个字符串。我们要用最少的字符操作次数，将字符串 A 转换为字符串 B。这里所说的字符操作共有三种：

1、删除一个字符；

2、插入一个字符；

3、将一个字符改为另一个字符

[洛谷 P2758] 编辑距离

设 $f[i][j]$ 表示通过操作将 A 的前 i 个字符和 B 的前 j 个字符变成一样的最小操作次数。或者说，表示的是只考虑 A 的前 i 个字符和 B 的前 j 个字符的答案。
考虑三种操作，得到转移：

$$f[i][j] = \min \begin{cases} f[i-1][j]+1 \\ f[i][j-1]+1 \\ f[i-1][j-1] & a_i = b_j \\ f[i-1][j-1]+1 & a_i \neq b_j \end{cases}$$

区间 dp

区间 dp 是在区间上进行动态规划。题目一般有很明显的特征：数据范围较小 (一般不会超过 500)，且整个区间上的答案一定可以通过两个子区间的答案合并得到。
状态一般都是设 $f[l][r]$ 表示考虑子区间 $[l, r]$ 的答案。
既然大区间可以通过小区间合并答案得到，那么我们只需要从小往大枚举区间长度 (用来消除后效性)，每次转移枚举小区间的断点 k，$f[l][r]$ 便可以用 $f[l][k]$ 与 $f[k+1][r]$ 来进行更新。

[LOJ#10147] 石子合并

将 n 堆石子绕圆形操场排放，现要将石子有序地合并成一堆。规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记做该次合并的得分。

请编写一个程序，读入堆数 n 及每堆的石子数，并进行如下计算：

1. 选择一种合并石子的方案，使得做 $n-1$ 次合并得分总和最大。
2. 选择一种合并石子的方案，使得做 $n-1$ 次合并得分总和最小。

[LOJ#10147] 石子合并

先不考虑排成一个环带来的影响，只考虑排成一列的情况。
设 $f[l][r]$ 表示合并区间 $[l,r]$ 的最大得分。如果考虑最后一个合并操作，那么必定存在一个断点 k，使得先会合并完 $[l,k]$ 与 $[k+1,r]$，然后再将两段子区间所合并出来的两堆石头合并成一个大堆。于是我们得到转移方程：

$$f[l][r]=\max_{k=l}^{r}\{f[l][k]+f[k+1][r]+sum[l,r]\}$$

其中 $sum[l,r]$ 表示区间 $[l,r]$ 的所有石子的总个数。
而最小值的转移是类似的，只需要将 $\max$ 改成 $\min$ 就行了。

[LOJ#10147] 石子合并

现在把环的情况考虑回来。我们有一种很通用的方法来处理环上的连续区间问题，即破环成链。
首先还是当成一列读进来，然后将这个数组在自己后面复制一次。这个长度为 $2n$ 的数组的某一个长度为 n 的子列必定不会出现跨越断点的情况，此时等价于一个链。这样子我们对于所有的 $[i, i+n]$ 的答案取一个最值就可以了。
关于上面的证明，因为是一个环，所以存在 n 个断点，但又只会合并 $n-1$ 次。所以必定存在一个断点不会被合并，意味着不会进行任何跨越这个断点的操作。

树型 dp

树型 dp 一定要确定好树的结构，一定是有根树 (无根树随便选择一个节点当做有根树)。
确定好结构之后，一般的 dp 的状态设计为：设 $f[i]$ 表示 i 节点的子树的答案，根据实际情况会附加额外的维度来维护更多的情况。转移一般考虑清楚如何将两棵子树的答案合并成一个。只要想清楚了两个的合并，那么每次新增一个子树合并就可以了。
所以，在树型 dp 中，如何将不同的子树的结果合并到当前的根节点上来就是思考的关键。而由于要消除后效性，所以一定是按照 dfs 序来逆序进行处理。一般实现时，通过 dfs，在回朔时更新根节点答案即可。

[洛谷 P1352] 没有上司的舞会

某大学有 n 个职员，编号为 $1 \ldots n$。
他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。
现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 r_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。
所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

[洛谷 P1352] 没有上司的舞会

题目已经告诉我们了，所有的关系可以看成一棵树。题目可以归结成：有一棵有根树，每个点有一个权值，你需要选择一些点，使得这些点之间不存在父子关系，且这些点点的权值和最大。
设 $f[i][0/1]$ 表示考虑 i 这棵子树，0 表示 i 这个点不选,1 表示这个点选时的子树内的答案。
考虑如何转移，如果我们的 i 不选，那么儿子的子树的根节点无论选不选都无所谓，设 v 是 i 的子节点，于是：

$$f[i][0] = \sum \max\{f[v][0], f[v][1]\}$$

如果 i 要选择，那么儿子的子树的根节点都必定不能选，于是：

$$f[i][1] = \sum f[v][0]$$

位运算

$i\&j$：i 和 j 的二进制按位取和 (and)
$i|j$：i 和 j 的二进制按位取或 (or)
$i\hat{}j$：i 和 j 的二进制按位取异或 (xor)，后面会写成 $i \oplus j$ 便于辨认
$i << k$：$i * 2^k$
$i >> k$：$i/2^k$，这个是向下取整
以上操作只适用于整数。

状态压缩 dp

如果要强行设状态会出现多个维度 (一般维度不超过 25)，但是每个维度的取值很小时 (一般不会超过 4)，假设每一维的取值只有 p 个，我们可以用 $[0, p)$ 来表示，然后所有维度连在一起，则可以看成一个 p 进制数，这样子我们就可以用一个十进制数来表示这个状态了。

大部分题目下每一个维度都如：选/不选两个取值。于是所有的状态都可以对应到一个二进制数，进而可以用一个十进制数来表示。比如说我们不得不要在状态中记录 n 个物品是否被选择的情况时，由于 n 是变量，同时还比较大，直接开 n 维数组是不方便的。但我们把这 n 维变成一维，而这一位对应的二进制又可以与 n 维每一维的 0 或 1 一一对应起来，这样子极大的方便了我们状态的设计和代码的构建。

这类题目的难点之一在于二进制的处理和应用，一些恰当而合适的二进制写法可以让代码又好看又简洁。

[洛谷 P1896] 互不侵犯

在 $N \times N$ 的棋盘里面放 K 个国王，使他们互不攻击，共有多少种摆放方案。
国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

[洛谷 P1896] 互不侵犯

如果我们按行从上往下来放国王，那么我们需要直到的只有上一行的哪些位置放了国王，以及这一行的哪些位置放了国王。
设 $f[i][S][k]$ 表示当前考虑到了第 i 行，第 i 行放国王的状态为 S，当前总共放了 k 个国王的方案数。其中，S 的二进制的第 k 位 (即 2^k 位)，表示从左往右第 $k+1$ 个格子是否放了国王。
我们枚举上一行哪些位置放了国王，假设枚举出来的状态是 $S1$。
然后枚举这一行哪些位置放国王，枚举出来的是 $S2$。
由放置国王的限制，如果 $S2\&S1 \neq 0$ 或者 $S2\&(S1 << 1) \neq 0$ 或者 $S2\&(S1 >> 1) \neq 0$ 或者 $S2\&(S2 >> 1) \neq 0$ 或者 $S2\&(S2 << 1) \neq 0$ 则意味着这个状态不合法。
否则则可以进行状态的转移：

$$f[i][S2][j + pop\_count(S2)] += f[i-1][S1][j]$$

其中 $pop\_count(S)$ 表示 S 二进制下 1 的个数。初值 $f[0][0][0] = 0$。

数位 dp

数位 dp 一般是让你求一个区间内具有某种特性的数的个数。而这个特性与每一位数字分别是什么有关系。
而求区间内的个数，我们会进行以下转化。假设要求 $[l, r]$ 之间符合要求的数的个数，等价于求 $[0, r]$ 和 $[0, l-1]$ 符合要求的数的个数然后做差。同时，这样子做差也让我们的前导零变为了可行，方便了计算。
数位 dp 一般思路是：设 $f[i][0/1]$ 表示当前考虑到了第 i 个数位，是否恰好卡在上界上。如果当前数继续卡在上界上，那么我们的填数方式依然受到上界的限制，否则已经严格小于上界，可以随意填数。如果有更多的限制条件，则可能需要增加额外的维度来加以限制。

[LOJ#10164] 数字游戏

定义一个数是不降数，当前仅当这个数字从左往右的每个数位都依次满足小于等于的关系。如 $111, 112, 123$。
求 $[a, b]$ 中不降数的个数。
$1 \leq a \leq b \leq 2^{31} - 1$

[LOJ#10164] 数字游戏

这题很明显符合前面所说的数位 dp 的性质。所以我们只需要考虑 $[0, n]$ 如何计算答案。

设 $f[i][j][0/1]$ 表示从最高位到第 i 位，第 i 位填的数是 j，是否严格卡在上界上的方案数。

其中，0 表示的是在前面已经存在某一位小于上界，即之后不再受到上界的限制

1 表示到当前为止，仍然在上界上，受到上界的限制

如果之前已经严格小于上界值了，那么当前第 i 位填的数只需要大于等于第 $i-1$ 位填的数就行了。

否则，第 i 位能够填的数不仅要大于等于第 $i+1$ 位填的数，还需要小于等于上界在这一位上的值，上界的这一位的值就是 n 的第 i 位上的数字。

在转移过程中注意更新上界条件。

初值 $f[l+1][0][1]=1$，其中 l 表示 n 的位数。

[LOJ#10164] 数字游戏

例如：
要求 $[0, n]$ 的方案数，$n = 123456$。
假如到了第 2 位，你卡在上界上，意味着你前面两位填的是 12。
如果到了第 4 位，仍然卡在上界上，意味着你前面 4 位填的是 1234。
而如果到了第 5 位，你不卡在上界上了，那就可能是 $12340, 12341, 12342, 12343, 12344$ 这样子的情况。
而一旦不再卡在上界了，那么意味着之后的每一位填数时都不再受到限制，可以任意填数，同时也意味着后面不再存在上界了。

[POJ3208]Apocalypse Someday

定义一个数是魔鬼数，当且仅当它存在连续的三个数位都是 6。
求第 x 个魔鬼数。

[POJ3208]Apocalypse Someday

所有求第几个某种数的题目，如果这种数的性质满足数位 dp 的性质的话，基本都可以二分答案 + 数位 dp，即二分一个答案 mid，求 $[0, mid]$ 中这种数的个数。
那么我们先二分答案，问题就变成了一般的数位 dp，即求 $[0, n]$ 中的魔鬼数的个数，在本题中 $n = mid$。
设 $f[i][0/1/2][0/1]$ 表示当前从高位往低位考虑到了第 i 位，已经连续填了 $0/1/2$ 个 6，且是否严格卡在上界上的，非魔鬼数的个数。最终算答案的时候就是 $n + 1-$ 非魔鬼数的个数。
转移的时候枚举当前数，更新一下连续 6 的个数，然后注意上界限制就好了。

期望 dp

这类 dp 题准确的说是在 dp 的外面套了一个期望的壳子。而期望 $=\sum$ 可能取值 $\times$ 对应概率。换而言之，期望就是在概率下的平均值。我们可以直接当它是一个具体的、确定的、可以用于替代概率带来的不确定的具体值。

原本的转移是转移值乘上系数，在期望的壳子下，转移变成了 (转移值 $+$ 改变量)$\times$ 对应概率。注意系数和是 1(因为概率和是 1)。

这里的状态有两种设计方法：一种是设当前状态下要达到目标状态的期望，另外一种是从初始状态达到当前状态的期望。

[HDU4336]Card Collector

N 个物品，每次得到第 i 个物品的概率为 pi，而且有可能什么也得不到，问期望多少次能收集到全部 N 个物品

$n \leq 20$

[HDU4336]Card Collector

不难发现我们需要记录下我们当前已经收集到了哪些物品，也就是我们需要记录每一个物品是否被收集到。因此考虑状压 dp。
设 $f[S]$ 表示当前已经拿到了 S 所对应的二进制数所表示的那些物品，收集完所有物品的期望次数。我们要的最终答案就是 $f[0]$。
显然初值 $f[(1<<n)-1]=0$。
考虑后效性的影响，我们从大往小枚举 S，考虑枚举 S 状态中最后一个收集到的物品，设 $(1<<j)\&S=0$，即最后一个收集的物品是 j。
于是有转移：

$$f[S]=(\sum p[j]*f[S|(1<<j)]+1)/(\sum p[j])$$

[HDU4336]Card Collector

证明这个转移：
如果 $(1 << j)\&S \neq 0$，意味着这个是从 S 转移到 S。设 $P = \sum_{(1<<j)\&S=0} P[j]$，也就是：

$$f[S] = \sum_{(1<<j)\&S=0} p[j] * f[S|(1 << j)] + (1 - P) * f[S] + 1$$

化简后就得到了：

$$f[S] = (\sum_{(1<<j)\&S=0} p[j] * f[S|(1 << j)] + 1)/(\sum_{(1<<j)\&S=0} p[j])$$

[洛谷 P1291] 百事世界杯之旅

“……在 2002 年 6 月之前购买的百事任何饮料的瓶盖上都会有一个百事球星的名字。只要凑齐所有百事球星的名字，就可参加百事世界杯之旅的抽奖活动，获得球星背包，随声听，更可赴日韩观看世界杯。还不赶快行动！”

你关上电视，心想：假设有 n 个不同的球星名字，每个名字出现的概率相同，平均需要买几瓶饮料才能凑齐所有的名字呢？

[洛谷 P1291] 百事世界杯之旅

设 $f[i]$ 表示收集到了 i 个名字的期望的瓶数。设 $p = \frac{i-1}{n}$，即在已有 $i-1$ 个名字的情况下拿到已有名字的概率。

$$f[i] = f[i-1] + \sum_{j=0}^{\infty}(1-p) * p^j * (j+1)$$

即枚举拿到了多少次已有的名字之后，才拿到了一个新的名字。假设重复拿到已有名字的次数是 j，那么拿到了 j 次已有的名字的概率是 p^j，而拿完这 j 次之后拿到新名字的概率是 $1-p$，一共拿了 $j+1$ 次，所以对于期望的贡献就是 $(1-p) * p^j * (j+1)$。化简后得到

$$f[i] = f[i-1] + \frac{1}{1-p} = f[i-1] + \frac{n}{n-i+1}$$

[洛谷 P1291] 百事世界杯之旅

化简过程：考虑怎么求式子中的 $\sum_{j=0}^{\infty} p^j * (j+1)$。

设 $S = \sum_{j=0}^{\infty} p^j * (j+1)$。

则 $pS = \sum_{j=0}^{\infty} p^{j+1} * (j+1) = \sum_{j=1}^{\infty} p^j * j$

做差后可以得到：

$$
\begin{aligned}
(1-p)S &= 1 + \sum_{j=1}^{\infty} p^j \\
&= \sum_{j=0}^{\infty} p^j \\
&= \frac{1-p^{\infty}}{1-p} \\
&= \frac{1}{1-p}
\end{aligned}
$$

于是 $S = \frac{1}{(1-p)^2}$，带回上面的式子即可求得结果。

[洛谷 P1291] 百事世界杯之旅

这题同样可以逆推，设 $f[i]$ 表示已经得到了 i 个，需要收集到 n 个的期望。
于是有：

$$f[i] = \frac{i}{n}(f[i]+1) + \frac{n-i}{n}(f[i+1]+1)$$

化简后得到：

$$f[i] = f[i+1] + \frac{n}{n-i}$$

初值 $f[n] = 0$

有关状态的设计问题

在动态规划题目里面，状态的设计是最重要的一个环节，状态设计出来之后转移顺应思路是很容易得到的。
在 APIO2019 的现场，讲师提出了“从集合角度看待 dp”，即我们的状态是一个集合，包括了所有满足状态条件的东西，这些集合之间必定无交集，因此在设计状态时要做到不重不漏，这就需要我们的状态包含了足够多的特征，能够让相同的、可以放在一起的东西能够被归于一个集合，而存在区别的就要在不同的集合。
而通过上面的题目，我们也能够得到一个设计状态的启示：题目问什么我们就设计什么。这也是绝大多数 dp 题状态设计的方法。

Ending

动态规划是一类难度不定的题目，它可以很容易，也可以非常难。同时，动态规划也是使用率最高的方法，如果要熟练掌握，不仅仅要有恰当的方式来设计状态，同时还需要我们进行大量练习，才能够对这类题目更加熟稔。

资料推荐

《算法导论》第十五章

谢谢大家