单掉落装备套装收集加强问题

一、提出问题：

假想击杀游戏中某BOSS，每次可以掉落装备套装集合$\{I_i\}=\{I_1,\dots,I_i,\dots,I_s\}$中的一件装备，并且各个掉落事件是互斥的。装备掉落的对应概率为$\{p_i\}=\{p_1,\dots,p_i,\dots,p_s\},p_i>0$，如果一整套装备所包含单件装备的对应数目为$\{n_i\}=\{n_1,\dots,n_i,\dots,n_s\},n_i\geq 1$。求：收集一整套装备，需要击杀此BOSS次数的数学期望。

二、思路分析：

设收集一整套装备所需的次数为：

$$F(n_1,\dots,n_i,\dots,n_s)$$

下面以第一次的掉落结果情况分类：

（1）第一次没有掉落任何装备，则仍需击杀$F(n_1,\dots,n_i,\dots,n_s)$次；

（2）第一次掉落一件第$i(i=1\dots s)$件装备，则仍需击杀$p_i\cdot F(n_1,\dots,n_i-1,\dots,n_s)$次；

用数学表达式写出来就是：

$$F(n_1,\dots,n_i,\dots,n_s)=1+\sum_{i=1}^{s}p_i\cdot F(n_1,\dots,n_i-1,\dots,n_s)+\left(1-\sum_{i=1}^{s}p_i\right)\cdot F(n_1,\dots,n_i,\dots,n_s)$$

化简得：

$$F(n_1,\dots,n_i,\dots,n_s)=\frac{1+\sum_{i=1}^{s}p_i\cdot F(n_1,\dots,n_i-1,\dots,n_s)}{\sum_{i=1}^{s}p_i}$$

三、算法实现：

下面用递归思路来实现这个算法：

```
function R=CollectRecursion(M)
% M(1,:)为事件概率；M(2,:)为对应次数。
% R 为所有事件刚好发生不小于对应的次数时，发生次数之和。
M(:,M(2,:)<=0)=[];
[~,n]=size(M);
if n==1
    R=M(2,1)/M(1,1);
else
    R=1/sum(M(1,:));
    for k=1:n
        R=R+M(1,k)*CollectRecursion(Left(M,k))/sum(M(1,:));
    end
end

function T=Left(P,k)
% 辅助函数：P 第二行第 k 个元素自减 1.
P(2,k)=P(2,k)-1;
T=P;
```

为了验证上面算法的正确性，特写一个模拟函数，如下：

```
function R=CollectCount(M,N)
% 模拟函数（默认模拟 10 万次）。
```

```
if nargin<2
    N=100000;
end
P=[0 cumsum(M(1,:))];
R=0;
for k=1:N
    r=zeros(1,size(M,2));
    while sum(r<M(2,:))
        x=rand();
        R=R+1;
        for ii=1:size(M,2)
            if x>P(ii) && x<=P(ii+1)
                r(ii)=r(ii)+1;
            end
        end
    end
end
R=R/N;
```

举例计算：

例 1：

```
clc;clear;format long g;
M=[1/8 1/4 1/2;1 2 3];
fprintf('1、模拟结果 = %g\n',CollectCount(M));
fprintf('2、递归结果 = %g\n',CollectRecursion(M));
```

结果：

```
1、模拟结果 = 12.3932
2、递归结果 = 12.3958
```

例 2：

```
clc;clear;format long g;
M=[1/16 1/8 1/4 1/2;1 2 3 4];
fprintf('1、模拟结果 = %g\n',CollectCount(M));
fprintf('2、递归结果 = %g\n',CollectRecursion(M));
```

结果：

```
1、模拟结果 = 25.0396
2、递归结果 = 25.0108
```

四、算法优化：

1、初步优化

当装备件数和对应的收集数目比较小的时候，递归算法相对模拟方法，不论是在耗时还是在精度上，都有很高的优越性；当装备件数或对应的收集数目比较大的时候，递归算法的缺点暴露无遗，巨大的资源占用和超长的时间耗费使其比之模拟方法相形见绌。

鉴于上面的分析，可知寻找一个高效的算法势在必行。

实际上，普通递归（尤其是多元递归）方法之所耗时耗资源，是因为每次递归都要达到底层结果，几乎所有中间结果都要进行多次重复计算。如果设法将各个中间数据记录下来，之后每次

递归的时候直接调用，效率势必大幅度提高。但是，即便把中间数据记录下来，由于递归流程是自上而下的，在第一次递归过程中，其对系统资源的占用仍旧较大，如果反其道而行之，即采用自下而上的流程，将进一步提高效率。递推公式：

$$F(n_1,\dots,n_i,\dots,n_s)=\frac{1+\sum_{i=1}^{s}p_i\cdot F(n_1,\dots,n_i-1,\dots,n_s)}{\sum_{i=1}^{s}p_i}$$

所体现的就是这样的思路。接下来的实现方法，实际上是在计算一个$(n_1+1)\times(n_2+1)\dots\times(n_i+1)\times\dots(n_s+1)$的$s$维数组$F$，而且数组中的元素满足如上的关系。即根据$F(n_1,\dots,n_i-1,\dots,n_s),(i=1,2,\dots,s)$这$s$个结果来唯一确定$F(n_1,\dots,n_i,\dots,n_s)$，也就是说要计算出这个最终结果，必须事先确定$(n_1+1)\times(n_2+1)\dots\times(n_i+1)\times\dots(n_s+1)-1$个元素，每个元素只需计算一次并进行存储，从而实现对递归算法的优化，很显然，此算法的运算量与数组F的元素个数大致成线性正相关。下面给出具体的Matlab实现此递推方法的函数。

```
function [R,M]=Collect(P,N)
%% 函数主体
% 一、输入参数：
%   1、P：各个物品掉落几率；
%   2、N：各个物品收集数目。
% 二、输出结果：
%   1、R：要求结果；
%   2、M：所有中间结果数组。
if numel(N)==1
    R=N/P;
    M=arrayfun(@(n) n/P,1:N);
else
    Mat=zeros(N+2);
    GetIndex(N+1);
    R=eval(['Mat(end',cell2mat(arrayfun(@(x) ',end',1:numel(N)-1,'UniformOutput',false)),')']);
    M=squeeze(eval(['Mat(3:end',cell2mat(arrayfun(@(x) ',3:end',1:numel(N)-1,'UniformOutput',false)),')']));
end
    %% 内嵌函数：依次获取数组的完整索引
    function GetIndex(Index_Remain,Index_Store)
        if nargin<2
            Index_Store=[];
        end
        switch numel(Index_Remain)
            case 1
                for i=1:Index_Remain(1)
                    Count([Index_Store i]+1);
                end
            otherwise
                for i=1:Index_Remain(1);
                    index_store=[Index_Store i];
                    index_remain=Index_Remain(2:end);
                    GetIndex(index_remain,index_store);
                end
```

```
        end
    end
    %% 内嵌函数：根据获取的数组索引操作原始数组元素
    function Count(Index)
        Temp=0;
        for i=1:numel(Index)
            index=Index;
            index(i)=Index(i)-1;
            Temp=Temp+P(i)*eval(['Mat',Index2Str(index)]);
        end
        if sum(P(Index>=3))==0
            Temp=0;
        else
            Temp=(1+Temp)/sum(P(Index>2));
        end
        eval(['Mat',Index2Str(Index),'=',num2str(Temp),';']);
    end
    %% 内嵌函数：一维索引数组转化成索引字符串
    function Str=Index2Str(Index)
        Str=mat2str(Index(1));
        if numel(Index)>1
            for i=2:numel(Index)
                Str=[Str,',',mat2str(Index(i))];
            end
        end
        Str=['(',Str,')'];
    end
end
```

下面仍旧使用上面方法中的几个例子。

例 1：

```
clc;clear;format short;
P=[1/4,1/2];
N=[1,2 ];
tic;R2=CollectRecursion([P;N]);toc;
tic;[R3,M]=Collect(P,N);toc;
[R2;R3]
```

结果：

```
Elapsed time is 0.001641 seconds.
Elapsed time is 0.028377 seconds.
ans =
    5.7778
    5.7778
```

例 2：

```
clc;clear;format short;
```

```
P=[1/8,1/4,1/2];
N=[1,2,3 ];
tic;R2=CollectRecursion([P;N]);toc;
tic;[R3,M]=Collect(P,N);toc;
[R2;R3]
```

结果：

```
Elapsed time is 0.004808 seconds.
Elapsed time is 0.066477 seconds.
ans =
    12.3958
    12.3958
```

从上面两个例子可以看出，当计算规模比较小的时候，递归方法优于递推方法。但这不代表前者在任何时候都优于后者，且看下面两个计算规模稍大一点的例子。

例 3：

```
clc;clear;format short;
P=[1/16 ,1/8,1/4,1/2];
N=[1,2,3,4];
tic;R2=CollectRecursion([P;N]);toc;
tic;[R3,M]=Collect(P,N);toc;
[R2;R3]
```

结果：

```
Elapsed time is 0.678237 seconds.
Elapsed time is 0.459696 seconds.
ans =
    25.0108
    25.0108
```

例 4：

```
clc;clear;format short;
P=[1/32,1/16,1/8,1/4,1/2];
N=[1,2,3,4,2];
tic;R2=CollectRecursion([P;N]);toc;
tic;[R3,M]=Collect(P,N);toc;
[R2;R3]
```

结果：

```
Elapsed time is 41.706194 seconds.
Elapsed time is 2.021279 seconds.
ans =
    50.0223
    50.0222
```

随着计算规模的增大，递推方法越发明显地优于递归方法。

2、进一步优化

上面的算法的核心是按照一定的顺序依次对一个数组的各个元素进行操作，其中获取数组元素的部分占了不小的比例。而且由于在获取数组元素索引的时候使用了递归，从而为解决大规模

数组情况埋下了低效隐患。

实际上，不管数组有多大，维数有多高，编译器总是在内存中为其分配一串连续的存储空间，即总是以一维数组的形式存储的，具体为依次嵌套循环各个维度，然后将获取的数组元素与顺序排列的内存地址对应。不同的编译器在确定数组循环嵌套顺序的时候略有不同，在Matlab中按照从左至右的顺序对各个维度索引进行循环，而在C中则是按照从右至左的顺序对各个维度索引进行循环的。根据这个事实，用一维数组代替动态多维数组的存储方式，利用多维数组的虚拟索引和一维数组索引的对应关系及相互转化，重新编写函数实现前面的递推方法。具体的Matlab代码如下所示。

```
%% 主函数
% 一、参数：P 各个物品掉落几率，N 各个物品收集数目；
% 二、结果：R 要求结果。
function R=Collect(P,N)
    n=numel(N);Mat=zeros(1,prod(N+1));
    Pow=cumprod(N+1);Pow=[1 Pow(1:end-1)];
    for index=2:numel(Mat)
        vector=Ind2Vec(index,Pow);
        SubInd=ones(n,1)*vector-eye(n);
        Uper=zeros(1,n);Down=zeros(1,n);
        for i=1:n
            if SubInd(i,i)>0
                Uper(i)=P(i)*Mat(Vec2Ind(SubInd(i,:),Pow));
                Down(i)=P(i);
            end
        end
        Mat(index)=(1+sum(Uper))/sum(Down);
    end
    R=Mat(end);
end
%% 子函数
% 1、内存索引转数组索引
function Vector=Ind2Vec(Index,Pow)
    Vector=zeros(size(Pow));
    for i=numel(Pow):-1:1
        Vector(i)=ceil(Index/Pow(i));
        Index=Index-(Vector(i)-1)*Pow(i);
    end
end
% 2、数组索引转内存索引
function Index=Vec2Ind(Vector,Pow)
    Index=dot(Vector-1,Pow)+1;
end
```

下面比较新旧两个优化方法的运行效率（笔者将旧方法函数名改为 CollectOld）：

```
P=[0.25 0.25 0.25 .25];
N=[3 4 5 6]+10;
```

```
tic;R=CollectOld(P,N);toc;
tic;R=Collect(P,N);toc;
```

结果：

```
Elapsed time is 98.468995 seconds.
Elapsed time is 4.602368 seconds.
```

旧方法98.5秒，新方法4.6秒，运行效率得到了大幅提升。另外，由于没有使用一些Matlab的特殊函数（比如 eval 函数），新方法的可移植性明显高于旧方法。

PS：补充 Python 代码，以供查询调用。

```
# -*- coding: utf-8 -*-
import numpy as np
#   子函数：
#   1、内存索引转数组索引
def Ind2Vec(Index,Pow):
    Vector=np.zeros(len(Pow))
    for i in range(len(Pow),0,-1):
        Vector[i-1]=int(np.ceil(Index/float(Pow[i-1])))
        Index-=(Vector[i-1]-1)*Pow[i-1]
    return Vector
#   2、数组索引转内存索引
def Vec2Ind(Vector,Pow):
    Index=np.dot(Vector-np.ones(len(Vector)),Pow)+1
    return int(Index)

#   主函数：
#   参数：P 各个物品掉落几率，N 各个物品收集数目；
def Collect(P,N):
    n=len(P)
    Mat=np.zeros(np.prod(N+np.ones(n)))
    Pow=np.cumprod(N+np.ones(len(N)))
    Pow=np.append(1,Pow[0:-1])
    for index in range(1,len(Mat)):
        vector=Ind2Vec(index+1,Pow)
        SubInd=np.ones([n,1])*vector-np.eye(n)
        Uper=np.zeros(n)
        Down=np.zeros(n)
        for i in range(n):
            if SubInd[i,i]>0:
                Uper[i]=P[i]*Mat[Vec2Ind(SubInd[i,:],Pow)-1]
                Down[i]=P[i]
        Mat[index]=(1+np.sum(Uper))/np.sum(Down)
    return Mat[-1]
```

五、实例延伸：

前面的递推方法已经将之前的弱套装收集问题包含在内，实际上只要令n_i全为 1 即可。另外作为对此问题的深入理解，利用此方法解决如下的问题。

问题：击杀某BOSS，可等概率随机掉落A、B、C三件装备中的一件，掉落事件互斥。三件各取其一组合成一套。那么，要组成，1 套、2 套……10 套装备分别需要击杀此BOSS的平均次数为多少？

代码：

```
clc;clear;format short;
P=[1/3,1/3,1/3];
N=[1,1,1];
arrayfun(@(n) Collect(P,N*n),1:10)'
```

结果（左栏）：

```
ans =
    5.5000
    9.6389
   13.4869
   17.1920
   20.8084
   24.3628
   27.8709
   31.3426
   34.7848
   38.2023
```

```
    4.1389
    3.8480
    3.7051
    3.6164
    3.5544
    3.5081
    3.4717
    3.4422
    3.4175
```

很显然，平均击杀次数并不是套数的线性函数，因为相邻两项的差值不为常数（上右栏）。

虽然在理论上可以得到此规律的准确形式，但考虑到冗繁的过程，笔者并不打算这么做，也不建议这么做。这个例子的目的就是指出此规律的非线性特征，以纠正一些主观上想当然的错误结论。

QQ:707509279