已知某加密芯片通过密钥对数据进行相关运算，假设无密钥、密钥为1和密钥为0的出现服从马尔科夫特性，当前密钥为1下一密钥为1、0的或者无密钥的概率分别是0.5，0.2，0.3，当前密钥为0下一密钥为1、0的或者无密钥的概率分别是0、0.7、0.3，当前无密钥下一密钥为1、0的或者无密钥的概率分别是0.4、0.2、0.4。通过能耗收集工具观测该芯片，发现采用密钥1进行运算时，以0.9的概率观测到下图标记为1的宽波形，0.1的概率观测到下图标记为0的窄波形，当采用密钥0进行运算时，以0.8的概率观测到窄波形，以0.2的概率观测到宽波形，当无密钥时，以概率0.6观测到宽波形，以概率0.4观测到窄波形。

1.请分析该HMM的三要素

图画错了，应该是 1 0 无

$$
\text{一步转移概率矩阵} A = \begin{array}{c} \begin{array}{ccc} 0 & 1 & \text{无} \end{array} \\ \begin{bmatrix} 0.5 & 0 & 0.4 \\ 0.2 & 0.7 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{bmatrix} \end{array}
$$

$$
\text{发射矩阵} B = \begin{array}{c} \begin{array}{ccc} 1 & 0 & \text{无} \end{array} \\ \begin{bmatrix} \text{宽} & 0.9 & 0.8 & 0.6 \\ \text{窄} & 0.1 & 0.2 & 0.4 \end{bmatrix} \end{array}
$$

初始状态概率向量π不知道，上面这段话还没说

Python代码

一个HMM类，基本属性包括HMM的三元组，后面的题目依赖其成员函数实现

```
import numpy as np
import copy

class HMM:
    def __init__(self, A, B, P=None):
        self.A = np.array(A)
        self.B = np.array(B)
        self.P = np.array(P)
    #A:  一步转移矩阵 B: 发射矩阵  P: 概率向量，可以暂时不给出
```

2.若初始状态为 (1,0,0)，请给出一条可能的观测序列

使用穷举法

Python的列表支持列表套列表。直接用列表搞一个简易的二叉树

这里用到两个函数

一个类外的函数

这个函数旨在对列表进行数乘，例如multiply(2, [[1,2], [3,4]]),希望返回一个[[2,4], [6,8]]

运用了递归来完成这一目的

```
def multiply(number, item):
    spam = copy.deepcopy(item)#进行这种操作的时候不希望是原地修改的，否则会引起意想不到的错误
    if isinstance(spam, list):
        for index in range(0, len(spam)):
            spam[index] = multiply(number, spam[index])
    else:
        return number * spam
    return spam
```

一个HMM类的成员函数

```
class HMM:


      def exhaustion_observation_sequence(self, length ,p='default' ):#给出一个初始的概率向量p，穷举其可能的观测序列及其概率,length为长度
```

```
        if p == 'default':
            p = self.P
        p = np.array(p).T
        rst = []
        probability_wide, probability_narrow = self.B.dot(p)  # 初始时刻观测到宽、窄的概率
        rst.append(probability_wide)
        rst.append(probability_narrow)

        for cnt in range(1, length):
            p = self.A.dot(p) #更新概率向量
            probability_wide, probability_narrow = self.B.dot(p)  # 这一时刻观测到宽、窄的概率
            buf1 = multiply(probability_wide, rst)
            buf2 = multiply(probability_narrow, rst)
            rst = [buf1, buf2]

        return rst
```

main函数直接实例化一个HMM类的对象，调用此函数

调用函数，生成长度为3的序列，结果如下:

```
[[[0.548181, 0.060909000000000005], [0.14571900000000002, 0.016191]], [[0.16281900000000002, 0.018091000000000003], [0.04328100000000001, 0.004809000000000002]]]
```

下标0代表宽，下标1代表窄。此列表嵌套了三层，举个例子

第一个元素0.548181的索引是000,代表的就是宽宽宽。 意味着宽宽宽的概率是0.548181

而0.060909000000000005索引是001，代表的就是宽宽窄

3.若观测序列为*O*=(宽，窄，宽)，初始状态为(0.2,0.4,0.4)，求该观测序列出现的概率

方法1：使用2中的方法

这个序列长度仅是3，穷举法是可以算的

```
    A = [[0.5, 0, 0.4], [0.2, 0.7, 0.2], [0.3, 0.3, 0.4]]
    B = [[0.9, 0.8, 0.6], [0.1, 0.2, 0.4]]
    hmm = HMM(A, B, P=[0.2, 0.4, 0.4])
    hmm.exhaustion(length=3)
```

结果:

```
[[[0.4261870160000001, 0.14974138400000006], [0.13606498400000003, 0.04780661600000002]], [[0.13473298400000003, 0.04733861600000002], [0.04301501600000002, 0.015113384000000006]]]
```

宽对应0，窄对应1 下标010的就是其概率，为0.13606498400000003 ≈ 0.136

方法2：新做一个HMM类的成员函数，接受一个“宽窄宽”这样的字符串，返回其概率。这样可以满足很长序列的概率计算

结果已经给定，设string = [宽窄宽]

只需遍历string 完成如下循环

1.
当前p向量下，观测到$string[i]$的概率p_i。设$spam = B * p^T$，如果$string[i]$是宽，$p_i = spam[0]$，如果$string[i]$是窄，$p_i = spam[1]$
2.
更新p向量，也就是$p \ = \ Ap$
3.
返回第一步
最后把所有的pi乘起来。当然也可以使用$P \ = \ P * pi$这种形式

```
def probability_of_certain_observation_sequence(self, string, p='default'):
        if p == 'default':
            p = self.P
        p = np.array(p).T
        probability = 1
        for i in string:
            spam = self.B.dot(p)
            buf = spam[0] if i == '宽' else spam[1]
            probability *= buf
            p = self.A.dot(p)
        return probability


if __name__ == '__main__':
    A = [[0.5, 0, 0.4], [0.2, 0.7, 0.2], [0.3, 0.3, 0.4]]
    B = [[0.9, 0.8, 0.6], [0.1, 0.2, 0.4]]
    hmm = HMM(A, B, P=[0.2, 0.4, 0.4])
    string = "宽窄宽"
    print(hmm.probability_of_certain_observation_sequence(string=string))

————————————————————————————————————————————————————————————————————————————————

0.13606498400000003

进程已结束,退出代码0
```

0.13606498400000003,与上面的完全一样。

但是它可以算很长的序列,速度还非常快

```
string = "宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽宽"
print(hmm.probability_of_certain_array(string=string))
______________________________________________________________________________________________________________

0.0023176388866588344

进程已结束,退出代码0
```

4.若观测到序列*O*=(宽，窄，宽)，求最大概率出现的隐藏状态序列I

> *在发布通知前，这部分内容已经完成了，所以当时我发出了如下疑问:*
>
> *为啥Viterbi会与穷举法结果不一样?*
>
> *检查后，代码算的概率都是没问题的*
>
> *是我的程序哪里错了还是本来就有可能出现这种情况呢?*
>
> 那么现在看来可能本来就是这样的

方法1：穷举、贝叶斯公式

这一问需要三元组的三个元素。初始状态采用了稳定分布下的P=[0.2667,0.4,0.3333]

$$P(I\,|\,O)=\frac{P(I)P(O\,|\,I)}{P(O)}$$

现在只需写一个新成员函数，算出每一个I对应的$P(I)$

再算出每一个I下产生此O的概率$P(O\,|\,I)$。相乘找一个最大的

```
    def exhaustion_hidden_sequence(self, length, p='default' ):#这个函数在HMM三要素给定的情况下，穷举长度为length的隐藏序列的概率
        if p == 'default':
            p = self.P
        p = np.array(p).T
        zero, one, nothing = self.A.dot(p)
        rst = [zero, one, nothing]
```

```
        for cnt in range(1, length):
            p = self.A.dot(p) #更新概率向量
            zero, one, nothing = self.A.dot(p)  # 这一时刻1\0\无的概率
            buf1 = multiply(zero, rst)
            buf2 = multiply(one, rst)
            buf3 = multiply(nothing, rst)
            rst = [buf1, buf2, buf3]
        return rst
```

返回值是一个三叉树列表，下标**0/1/2**分别代表 **密钥1/密钥0/无密钥**

```
[[[0.018962962989274076, 0.028444444448355553, 0.023703703683259258],
[0.028444444480355557, 0.042666666667199996, 0.035555555552044444],
[0.023703703731259258, 0.035555555555244443, 0.029629629597407406]],
[[0.028444444483555556, 0.042666666671999996, 0.035555555552444444],
[0.042666666719999996, 0.063999999999999999, 0.053333333327999999],
[0.035555555559644444, 0.053333333327999985, 0.044444444439555555]],
[[0.023703703736059260, 0.035555555555964444, 0.029629629603407405],
[0.035555555559964444, 0.053333333332799986, 0.044444444439955555],
[0.029629629663407406, 0.044444444439555544, 0.037037036995925920]]]
```

例如第一个元素0.018962962989274076，索引000，代表I=(密钥1，密钥1，密钥1)的概率，

稳态分布下，p不变，这一概率就等于0.2667^3 ≈ 0.01897 看来计算的是正确的

上面的**probability_of_certain_observation_sequence**函数是给定一个概率向量p，计算特定观测序列的概率

这一问需求是给定一个特定的隐藏序列I，计算特定观测序列的概率，即 $P(O \mid I)$

如果算[1，0，无]产生(宽，窄，宽)的概率，就等于（1产生宽）×（0产生窄）×（无产生宽）

```
    def HiddenSequence_Cause_ObservationSequence(self, HS, OS):#HS、OS为形如[0, 1, 2]或“宽窄宽”这样的任何可下标索引的数据结构
        probability = 1
        for cnt in range(0, len(HS)):
            i = 0 if OS[cnt] == '宽' else 1
            j = HS[cnt] #B矩阵的行、列下标
            probability *= self.B[i][j]
        return probability

print(hmm.HiddenSequence_Cause_ObservationSequence(HS=[0, 1, 2], OS=['宽', '窄', '宽']))
__________________________________________________________________________________________________________________________________________

0.10800000000000001
```

验证一下。这一值应该是0.9乘0.2乘0.6=0.108，程序返回的是正确的

最后再来处理数据,这里要用到递归来遍历穷举出来的 树， 因此需要额外一个函数完成递归任务

```
    def most_probable_hidden_sequence(self, sequence):#sequence为观测序列
        I_list = self.exhaustion_hidden_sequence(length=len(sequence))#穷举I的概率
        result = []
        self.func(I_list=I_list, HS=[], OS=sequence, result=result)#递归计算每一个P(O|I)
        result.sort(key=lambda item: item[1], reverse=True)#按概率排序,概率大的排在前面
        return result


    def func(self, I_list, HS, OS, result):
        for i in range(0, len(I_list)):
            HS.append(i)#记录下当前进入的下标
            if type(I_list[i]) == type(I_list):
                self.func(I_list[i], HS=HS, OS=OS, result=result)
                HS.pop()#HS就像一个栈，记录了下标，完成访问操作之后弹出
            else:
                probability = self.HiddenSequence_Cause_ObservationSequence(HS=HS, OS=OS)
                #已经访问到嵌套的最底层了，开始计算当前 隐藏序列 产生给定 观测序列 的概率
                result.append([HS[:], probability * I_list[i]])  #probability = P(O|I) ; I_list[i] = P(I)
                HS.pop()
        return result


    if __name__ == '__main__':
    A = [[0.5, 0, 0.4], [0.2, 0.7, 0.2], [0.3, 0.3, 0.4]]
    B = [[0.9, 0.8, 0.6], [0.1, 0.2, 0.4]]
    hmm = HMM(A, B, P=[0.26666667, 0.4, 0.33333333])
    sequence = ('宽', '窄', '宽')
    print(hmm.most_probable_hidden_sequence(sequence=sequence))
```

```
____________________________________________________________________________________
________________________________________________
[[[1, 2, 1], 0.013653333331968], [[1, 2, 0], 0.010240000011776001], [[0, 2, 1],
0.010239999991103997], [[2, 2, 1], 0.008533333332394664], [[1, 2, 2],
0.008533333323946667], [[1, 1, 1], 0.008192], [[0, 2, 0], 0.007680000008928001],
[[2, 2, 0], 0.006400000007296], [[0, 2, 2], 0.00639999999304], [[1, 1, 0],
0.006144000007680002], [[0, 1, 1], 0.0061440000000768], [[2, 2, 2],
0.005333333274133326], [[2, 1, 1], 0.005119999999948798], [[1, 1, 2],
0.005119999948800005], [[0, 1, 0], 0.004608000005817601], [[2, 1, 0],
0.003840000047615996], [[0, 1, 2], 0.00383999996208], [[2, 1, 2],
0.003199999996767999], [[1, 0, 1], 0.0027306666670080006], [[1, 0, 0],
0.002048000002816001], [[0, 0, 1], 0.0020480000002816], [[2, 0, 1],
0.0017066666666862933], [[1, 0, 2], 0.0017066666651733334], [[0, 0, 0],
0.0015360000021312004], [[2, 0, 0], 0.0012800000017472001], [[0, 0, 2],
0.0012799999998896], [[2, 0, 2], 0.0010666666657226664]]
```

所以结果就是:**(密钥0，无密钥，密钥0)**是最有可能的序列，要算它的概率具体是多少，只需要把上面这个列表中的概率全部归一化就可以了。

方法2：维特比算法（不是近似的）

> 有点懒，不编程了，公式也不摆了，用手算一下吧

概率向量仍然是稳态分布时的，$P=(0.2667,0.4,0.3333)^T$

首先确定各状态的δ_1

$$\delta_1(\text{密钥1})=0.2667\times0.9=0.24003=0.24$$

$$\delta_1(\text{密钥0})=0.4\times0.8=0.32$$

$$\delta_1(\text{无密钥})=0.3333\times0.6=0.1998=0.20$$

然后确定各状态的δ_2

$$\delta_2(\text{密钥1}) = \max\begin{Bmatrix} \delta_1(\text{密钥1})\times 0.5 \\ \delta_1(\text{密钥0})\times 0 \\ \delta_1(\text{无密钥})\times 0.4 \end{Bmatrix}\times 0.1 = \max\begin{Bmatrix} 0.24\times 0.5 \\ 0.32\times 0 \\ 0.20\times 0.4 \end{Bmatrix}\times 0.1 = 0.012$$

$$\delta_2(\text{密钥0}) = \max\begin{Bmatrix} \delta_1(\text{密钥1})\times 0.2 \\ \delta_1(\text{密钥0})\times 0.7 \\ \delta_1(\text{无密钥})\times 0.2 \end{Bmatrix}\times 0.2 = \max\begin{Bmatrix} 0.24\times 0.2 \\ 0.32\times 0.7 \\ 0.20\times 0.2 \end{Bmatrix}\times 0.2 = 0.0448$$

$$\delta_2(\text{无密钥}) = \max\begin{Bmatrix} \delta_1(\text{密钥1})\times 0.3 \\ \delta_1(\text{密钥0})\times 0.3 \\ \delta_1(\text{无密钥})\times 0.4 \end{Bmatrix}\times 0.4 = \max\begin{Bmatrix} 0.24\times 0.3 \\ 0.32\times 0.3 \\ 0.20\times 0.4 \end{Bmatrix}\times 0.4 = 0.0384$$

同理来确定各状态的δ_3

$$\delta_3(\text{密钥1}) = \max\begin{Bmatrix} \delta_2(\text{密钥1})\times 0.5 \\ \delta_2(\text{密钥0})\times 0 \\ \delta_2(\text{无密钥})\times 0.4 \end{Bmatrix}\times 0.9 = \max\begin{Bmatrix} 0.012\times 0.5 \\ 0.0448\times 0 \\ 0.0384\times 0.4 \end{Bmatrix}\times 0.9 = 0.013824$$

$$\delta_3(\text{密钥0}) = \max\begin{Bmatrix} \delta_2(\text{密钥1})\times 0.2 \\ \delta_2(\text{密钥0})\times 0.7 \\ \delta_2(\text{无密钥})\times 0.2 \end{Bmatrix}\times 0.8 = \max\begin{Bmatrix} 0.012\times 0.2 \\ 0.0448\times 0.7 \\ 0.0384\times 0.2 \end{Bmatrix}\times 0.8 = 0.025088$$

$$\delta_3(\text{无密钥}) = \max\begin{Bmatrix} \delta_2(\text{密钥1})\times 0.3 \\ \delta_2(\text{密钥0})\times 0.3 \\ \delta_2(\text{无密钥})\times 0.4 \end{Bmatrix}\times 0.6 = \max\begin{Bmatrix} 0.012\times 0.3 \\ 0.0448\times 0.3 \\ 0.0384\times 0.4 \end{Bmatrix}\times 0.6 = 0.009216$$

现在达到终点，可以开始逆推了

δ_3中最大的是$\delta_3(\text{密钥0}) = 0.025088$，故第三个状态取“密钥0”

$\delta_3(\text{密钥0})$是由$\delta_2(\text{密钥0})\times 0.7\times 0.8$算出来的，故第二个状态取“密钥0”

$\delta_2(\text{密钥0})$是由$\delta_1(\text{密钥0})\times 0.7\times 0.2$算出来的，故第一个状态取“密钥0”

所以，$I =$（密钥0，密钥0，密钥0）

但是个又跟穷举算出来的不一样。

到底相信谁呢？？