Q1. 用 dp[k][i] 表示长度为 k 的所有序列中，序列中最大值为 i 的序列数。

$$dp[k][i]=\begin{cases} 1 & i=1 \text{ 或 } i=k+1 \\ dp[k-1][i]*i+dp[k-1][i-1] & i\geq 2 \\ 0 & i>k+1 \end{cases}$$

```
count_sequence (n)
let dp be a two dimension array with size of n×(n+1)
for k= 1 to n
    for i = 2 to (k+1)
        dp[k][i]=0.
    dp[k][1]=1
for k=2 to n
    for i = 2 to (k+1)
        dp[k][i] =dp[k-1][i]×i
                 +dp[k-1][i-1]

return sum(dp[n][i])
       1≤i≤n+1
```

算法过程：dp 中所有元素初始化为 0，所有 dp[k][1] 初始化为 1，因为任意长为 k 且序列中最大值为 1 的序列只能为全 1 序列。对于其它情况：dp[k][i] 只能由 dp[k-1][i-1] ~~且末~~的序列末尾添 i 或 dp[k-1][i] 的序列末尾添一个不大于 i 的数构成，即为 dp[k-1][i-1] + dp[k-1][i]×i

最后对所有 dp[n][i] 求和即得所有满足条件序列数量。

复杂度分析：k 从 2 到 n，i 从 2 到 (k+1)

复杂度为 $O(n^2)$

Q2.

Q2. 设 m[i,j] 保存了从 A_{i+1} 到 A_j 这一段字符串的最佳q评分

则 $m[i,j]=\begin{cases}0 & \text{if } i=j\\ \max\{m[i,k]+m[k+1,j]\} & \text{if } i<j\end{cases}$

$i\le k\le j-1$

```
count_maxq(A)
n=len(A)   // n表示A中字符数.
let m be a two-dimension array with size of n×n
for i=0 to n
    m[i][i]=0   // m[i][i]无意义,初始化为0.
    for l=2 to n
        j=i+l-1
        m[i][j]=q(A_{i+1}~Aj)   //初始化为整串q值
        for k=i to j-1
            max=m[i][k]+q(A_{k+1}~Aj)
            if max>m[i][j]
                m[i,j]=max.
return m.
```

main函数: return count_maxq(A)[0][n]

算法的解释:

一串字符串的最佳q值等于其任意划分的最佳q值之和中的最大值，因为其子串也可以用相同的方法求最佳q，所以就表示为递归式中的

$m[i,j]=\begin{cases}0 & \text{if } i=j\\ \max\{m[i,k]+m[k+1,j]\} & \text{if } i<j\end{cases}$

$i\le k\le j-1$

count_maxq函数的过程为：

首先初始化二维数组 m，大小为 n×n

然后初始化 m[i][i]为0，将正常的m[i,j]都初始化为对整个字符串 $(A_{i+1}\sim A_j)$ 直接求q的值。最后开始分段比较，找到最佳点。由于对m[i][j]，使用 m[i][k]时已经被计算过了，所

以降低了复杂度。

分析复杂度：i从1~

复杂度为 $O(n^3)$

Q3. 取数组dp[i

第i年结束后最佳

那么dp[i][j]=

```
max_save
for i=1 to n
    dp[0][n]
    dp[0][i]=
    for i=1 to
```

{

```
for j=
    dp
    p
    f
```

```
max_sav
final=1
for j=2
    if
```

retur

以降低了复杂度。

分析复杂度：i从1—n，j从2到n，k从j到(i+1-2)

复杂度为 $O(n^3)$

Q3. 取数组dp[i][j]表示第i年采取副业j，在第i年结束后最佳收益。

那么 $dp[i][j] = \begin{cases} dp[i-1][j] + P[j] & \text{保持j更大} \\ \max(dp[i-1][k] + P[j] - P[k] - C[j]) & 1 \le k \le n \\ dp[i-1][k] > C[j] & \text{从副业k变过来更大。} \end{cases}$

```
max_save (n.m, C.P)
for i=1 to n
    dp[0][n]=-∞
 dp[0][1]=0
 for i=1 to m
maxq(A)[0][n] ;
      for j=1 to n
          dp[i][j]=dp[i-1][j]+P[j]
          path[i][j]=j
          for k=1 to n
              if dp[i-1][k]-p[k]>=C[j]
                AND dp[i-1][k]+P[j]-P[k]-C[j]
                                    >dp[i][j]
                    dp[i][j]=dp[i-1][k]+P[j]
                                -P[k]-C[j]
                    path[i][j]=k.
max_saving=dp[m][1]
final=1
for j=2 to n
    if dp[m][j]>max_saving
        max_saving=dp[m][j]
        final=j
return max_saving, path, j
```

```
find_path (path, j, m)
plan = [], year = m, strategy = j
while year > 0
    plan.insert(0, (year, strategy))
    strategy = path[year][strategy]
    year -= 1
return plan

main ( )
    max_saving, path, j = max_save(n, m, c, p)
    plan = findpath (path, j)
```

算法解释：max_save 分别计算第 i 年以每种副业结尾的最大收益，计算中初始默认上一年也是此副业时收益最大，后比较从另一副业切进来收益是否更大，更大则替换，并把上一年的最佳副业存入 path。

j从1到n，K从1到n

算法复杂度为 $O(n^2)$

Q4. 首先假设 distances 数组、profits 数组分别保存了各夜宵摊到路口的距离、各夜宵摊每晚的期望收益

即 distances[1]=m1 … distances[n]=mn

profits[1]=p1 … profits[n]=pn.

```
find_path (path, i)
target = i
answer = [ ] // 初始化 answer 存储最佳策略中选择的摊位号.
answer=answer∪{i}
while (path[target]!=-1)
    answer = answer ∪{path[target]}
    target = path[target]
return answer.
```

```
max_profit (n, k, distances, profits)
dp=[ ]
path=[ ]  //初始化长度为n的数组dp和path
          dp存利润, path存上一摊位位置.
for i=1 to n
    dp[i] = profits[i], path[i]=-1
    for j=1 to i
        if distances[i]-distances[j]>=k
            dp[i]=max(dp[i], dp[j]+profits[i])
            path[i]=j
max=dp[1], temp=1
for i=2 to n
    if dp[i]>max
        max=dp[i]
        temp=i
answer=find_path(path, temp)
return max, answer.
```

这个算法包含2个函数，其中 max_profit 函数计算当策略中最远摊位为第i个摊位时，其对应的最佳收益。dp[i]初始化为profit[i]

递归式为：$dp[i]=\begin{cases} profit[i] & \text{不存在摊位号比i小且距离不小于k的摊位} \\ \max\limits_{j}(dp[j]+profit[i]) & \text{从所有与i距离大于等于k且摊位号比i小的摊位中找最大的dp[j]，加profit[i]即为dp[i]} \end{cases}$

这也即算法中的

dp[i]=max(dp[i], dp[j]+profits[i])

若i能找到对应的最优j，则path[i]=j，

否则 path[i]=-1

find_path 函数就起到找到路径上所有最优摊位号的作用，每次找 path[i]、path[path[i]]……以此类推，直至找到 path[target]=-1 则结束。

分析复杂度：max_profit 函数中的依次求解dp[i]过程应为复杂度最高的部分，因为i从(1~n)中，j从(1~i)，复杂度为O(n²)