Assignment2

T1

Given a set of distinct positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies:$S_i\%S_j = 0$ or $S_j\%S_i = 0$.

Solution1：

最优子结构及 DP 方程：

最优子结构记为 OPT[i],其中 i 表示该集合由小到大排序后前 i 个元素中符合整除性质且包含第 i 个元素的最大子集的长度。记原集合为 S，由小到大排序后数组为 SS，有如下方程：

$$OPT[i]=\begin{cases}\max(OPT[j])+1 & \text{if SS[i]\%SS[j]=0;}\\ 1 & \text{other.}\end{cases}$$

上式中，SS[i]表示 SS 中第 i 个数，i=0,1,2,……,n，j=0,1,2,……,i。

算法描述及伪代码：

首先观察整除性质，满足该性质的子集中，最大的数可以整除其他任意一个元素，所以若一个数能整除其中最大的元素，则能整除其中所有元素，因此，我们若要向该集合中添加元素，只要对最大的元素进行判断即可。所以，为了方便操作，我们首先要对该集合进行排序，变成一个由小到大的数组 SS。对于任意一个数，它本身可以整除自己，所以初始化 OPT[i]=1;对于第 i 个元素，它要找到前 i 个元素中最大的且它可以插入的子集，若有，则原最大加一，若没有，则自身设为 1。这样对所有 i 操作完后，我们就得到了所有以第 i 个元素为最大元素的最大子集合。再到其中遍历找最大的子集合便找到整体最大子集的大小，为了能得到该集合中的元素，可以再设置一个指针数组 P[],P[i]中存储第 i 元素的前一个元素的索引，借此可以找到所有元素。

```
IntPut: S
(1) SS=Sort(S)
(2) 初始化 OPT=[1]*n,P=[null]*n, LagestS={},
(3) for i=0 to n:
        for j=0 to i:
           if SS[i]%SS[j]==0&&OPT[i]<OPT[j]+1:
              OPT[i]=OPT[j]+1
              P[i]=j
           end if
        end for
    end for
(4) maxLen=max(OPT[]), imax=Indexof(max(OPT[]))
(5) while (imax!=null):
        LagestS+={OPT[imax]}
        imax=P[imax]
    end while
(6) return (maxLen,LagestS)
OutPut: LagestS
```

算法正确性：

见上算法描述，正确性保证有两点：第一，按上述算法能确保所找到的子集

是满足整除性质的，因为只有能整除了才会添加；第二，保证将所有满足条件的子集全都考虑了，且选择的子集是其中最长的。

算法复杂度分析：

对集合进行排序 O(nlogn), 下面对每个元素进行操作：1+2+3+……+n,是$O(n^2)$,下面从 OPT 中找最大，O(n),最后找到所有元素小于 O(n)，因此，整体的时间复杂度为$O(n^2)$。

Solution2：

给定一组不重复的正整数集合，找一个最大的子集，使得里面的每一个元素满足 Si % Sj = 0 或者 Sj %Si = 0。首先我们将集合 s 里的元素按照从小到大的顺序排序，这样我们每次就只要看后面的数字能否整除前面的数字。定义一个动态数组 OPT，其中 OPT[i]表示到元素 s[i]位置最大可整除的子集合的长度，另外还定义一个一维数组 pre，来保存上一个能整除的元素的位置.在这个过程中，如果 s[i]能整除 s[j], 且 OPT[i] < OPT[j] + 1 的话，更新 OPT[i]和 pre[i]，最后回溯，根据 pre 数组来找到每一个元素。

所以最优子结构性质 OPT=max(OPT(i),OPT(j)+1) (j in range(i))

(b) pseudo-code:

```
def LargestSubset(s)
s = sorted(s)
    n = len(s)
    OPT = [0] *n
    pre = [None] *n
    for i in range(n):
        for j in range(i):
            if s[i] % s[j] == 0 and OPT[j] + 1 > OPT[i]:
                OPT[i] = OPT[j] + 1
                  pre[i] = j
    l = OPT.index(max(OPT)
    while l is not None:
    subset.append(s[l])
    l = pre[l]
    return subset
```

(c) the correctness of algorithm:

用动态规划来解决.使得问题转化为子问题。通过将集合元素按照升序排序,当 a%b==0，b%c==0 则有 a%c==0 这就保证了最优子结构性质的正确性. 当 s[i]% s[j]==0 的时候 OPT[i] = OPT[j]+1，所以有 OPT[i] =max(OPT[i], OPT[j]+1)。

(d) the complexity of algorithm:

对于包含 n 个元素的集合，对于每一个元素 s[i]都要查找其是否能整除 s[j](j in range (i)),所以时间复杂度为 O(n^2)

T2：

A robber is planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

1. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.
2. What if all houses are arranged in a circle?

Solution1：

最优子结构及 DP 方程：

对于第一问：OPT[i]表示前 i 个房子所能抢的钱的最大值，设第 i 个房子的现金数记为 M[i],则其 DP 方程为：

$$OPT[i]=\begin{cases}\max(OPT[i-2]+M[i],OPT[i-1]) & \text{i>2;}\\ \max(M[i-1],M[i]) & \text{i=2;}\\ M[i] & \text{i=1;}\end{cases}$$

对于第二问：OPT[i] 表示前 i 个房子所能抢的钱的最大值，设第 i 个房子的现金数记为 M[i],则其 DP 方程为：

$$OPT1[i]=\begin{cases}\max(OPT1[i-2]+M[i],OPT1[i-1]) & \text{2<i<n;}\\ \max(M[i-1],M[i]) & \text{i=2;}\\ M[i] & \text{i=1;}\\ OPT1[i\text{-}1] & \text{i=n;}\end{cases}$$

$$OPT2[i]=\begin{cases}\max(OPT2[i-2]+M[i],OPT2[i-1]) & \text{i>2;}\\ M[i] & \text{i=2;}\\ 0 & \text{i=1;}\end{cases}$$

$$OPT[i]=\max(OPT1[i],OPT2[i])$$

其中，OPT1[i]是指选第 1 个不选最后一个的前提下的最优解，OPT2[i]是指不选第一个的前提下的最优解。

算法描述及伪代码：

对于第一问：算法比较简单，假设已有前 i-1 个房子的最优解，在考虑当前房子是否抢的时候，只要第 i-2 个房子时的最优解加上当前房子的钱大于第 i-1 个房子的最优解时，就抢，否则，不抢该房子，将第 i-1 个房子的最优解当做自己的最优解，因为如果抢了该房子，则第 i-1 个房子就不能抢了。

对于第二问：若房子按圆排列，则第 1 个房子与最后 1 个房子也相邻，其实与第一问变化不大，只有在第 1，第 n 个房子时要特殊考虑下而已，我们可以分类讨论，第一种情况，第 1 个房子不抢，那么只要用第一问的算法，无论第 n 个房子抢不抢，都不会开启警报；第二种情况，第 1 个房子抢，此时第 n 个房子便不能抢了，则前 n-1 个房子用第一问的算法也可以求出最优解，我们只要从这两

种情况中选择更好的，便是整体的最优解。

第一问伪代码：注：为与描述一致，数组下标从 1 开始

```
InPut：M[]
(1) OPT[]=[0]*n
(2) OPT[1]=M[1] OPT[2]=max(M[1],M[2])
(3) for i=3 to n:
        if OPT[i-2]+M[i]>OPT[i-1]:
            OPT[i]=OPT[i-2]+M[i]
        else:
            OPT[i]=OPT[i-1]
        end if
    end for
(4) return OPT[n]
OutPut: OPT[n]
```

第二问伪代码：

```
IntPut:M[]
(1) OPT1[]=[0]*n,OPT2[]=[0]*n
(2) OPT1[1]=M[1],OPT1[2]=max(M[1],M[2])
(3) for i=3 to n-1:
        if OPT1[i-2]+M[i]>OPT1[i-1]:
            OPT1[i]=OPT1[i-2]+M[i]
        else:
            OPT1[i]=OPT1[i-1]
        end if
    end for
(4) OPT1[n]=OPT1[n-1]
(5) OPT2[1]=0,OPT2[2]=M[2]
(6) for i=3 to n:
        if OPT2[i-2]+M[i]>OPT2[i-1]:
            OPT2[i]=OPT2[i-2]+M[i]
        else:
            OPT2[i]=OPT2[i-1]
        end if
    end for
(7) MaxOPT=max(OPT1[n],OPT2[n])
(8) return MaxOPT
OutPut: MaxOPT
```

算法正确性：

根据上述算法描述，

当 i=1 时，OPT[1]=M[1]，显然正确；

当 i=2 时，OPT[2]=max(M[1],M[2]),也正确；

假设前 i-1 个最优解都正确，那么当第 i 个时，如果抢第 i 个，那第 i-1 个就不能抢，所以 只有当 OPT[i-2]+M[i]>OPT[i-1]时，才会选择抢第 i 个，OPT[i]=OPT[i-2]+M[i],否则，不抢第 i 个，则 OPT[i]=OPT[i-1].因此，第 i 个最优

解也正确。

综上，该算法是正确的。

对于第二问，由于第一问的算法是正确的，可以直接得到第二问算法也是正确的。

算法复杂度分析：

对于第一问，很明显，只需要遍历一次数组即可，复杂度为 O(n).

对于第二问，需要遍历两次数组，复杂度为 O(2n)=O(n).

Solution2：

（1）Solution:假设有 n 个要抢劫的房子，其中 value(i)表示第 i 个房子被抢劫的价值 OPT(i)表示对第 i 个房子做完决策后抢到的总钱数。对每一个房子我们都有两个选择，选择抢劫的话，那么我们需要考虑对 i-2 个房子抢还是不抢，如果决定不抢劫第 i 个房子，需要我们对第 i-1 个房子决定强还是不抢。

Sub-problem：对每一个 i 都要抢到最大利益

DP equation:

$$\mathrm{OPT(i)} = \max\begin{cases}\mathrm{value(i)} + \mathrm{OPT(i-2)} & when\ rob\ \ i \\ OPT(i-1) & not\ rob\ i\end{cases}$$

pseudo-code：

```
MoneyRobbing(i)//value 是存有房子价值的数组
1: if i==-1 then
2:   return 0
3: end if
4: if i==0 then
5:   return value[0]
6:  end if
7: return max{value[i]+MoneyRobbing(i-2),MoneyRobbing(i-1)}
```

Prove the correctness:

无论我们对第 i 个房子抢或者不抢，我们都会比较每一步，都抢到了最多的钱，所以第 i 个房子是在前面的基础上做出的最好的选择。

Complexity:

将算法分成了两个子问题：T(n)=T(n-1)+T(n-2)+c 常数项的时间复杂度，所以最后的时间复杂度为 O(2n)

（2）solution：此时的抢劫策略仍应该保持不变，但是房子首位相连，所以第一个房子和第二个房子一定不能同时抢劫。这个时候的抢劫策略有两种:一种是从抢劫（1，n-1）。第二种是抢劫（2，n）个房子。比较这两个哪一个价值更多。

Sub-problem:对第 i 个房子抢到最大值

DP equation:

$$\mathrm{OPT(i)} = \max\begin{cases}\mathrm{value(i)} + \mathrm{OPT(i-2)} & when\ rob\ \ i \\ OPT(i-1) & not\ rob\ i\end{cases}$$

Circle OPT=max{$OPT_{1,n-1}$,$OPT_{2,n}$}

pseudo-code：

```
MoneyRobbing1toN_1(i)//value 是存有房子价值的数组
1: if i==-1 then
2:   return 0
```

```
3: end if
4: if i==0 then
5:   Return value[0]
6:  end if
7: return max{value[i]+MoneyRobbing(i-2),MoneyRobbing(i-1)}
MoneyRobbing2toN(i)
8: if i==0 then
9:   return 0
10:end if
11:if i==1 then
12:  return value[1]
13:end if
14:return max{value[i]+MoneyRobbing(i-2),MoneyRobbing(i-1)}
CircleMonneyRobbing
15:return max(MoneyRobbing1toN_1(n-1), MoneyRobbing2toN(n))
```

Prove the correctness:

这个问题和上述问题的区别就在于，1 和 n 不可以同时抢，所以在 $OPT_{1,n-1}$ 中包含了抢 1 的情况。$OPT_{2,n}$ 中包括了第 n 个房子抢的情况，综上考虑到了所有的情况。

Complexity:

该算法的复杂度应该为 O(4n)即 O(n)

T3：

Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.

For example, given $s =$ "aab", return 1 since the palindrome partitioning ["aa", "b"] could be produced using 1 cut.

Solution1：

(a) Describe the optimal substructure and DP equation:

分割字符串 s，用最少的分割次数使分割 s 后产生的每个子串都是回文串，用 c[i]表示第 i 个字符后的子串的最小分割数，当 p[i][j]=TRUE 时，表示第 i 个字符到第 j 个字符是回文串，对于每一个 i(从 n-1 到 1),j 从 i 开始往后遍历，如果当前 s[j]=s[i]并且 p[i+1][j-1]=TRUE 则更新 OPT(i)=min[OPT(i), OPT(j+1)+1]，最后返回 c[0]即为所求。

所以最优子结构性质：OPT(i)=min[OPT(i), OPT(j+1)+1]

(b) pseudo-code:

```
Partition(s):
   n=len(s)
for i in range(1,n):
       c[i]=n-i
for i in range(n-1,1):
         for j in range(i,n-1):
             if(s[i]==s[j]&&j-i<2)||(s[i]==s[j]&& p[i+1][j-1])
```

```
            p[i][j]=True
            c[i]=min(c[i], c[j+1]+1)
 return c[0]
```

(c) correctness of algorithm:
假设有一个存在一个 OPT'(i)优于 OPT(i),那就意味着 OPT'(i)存在一个不同解，使得 c[i to j']小于 c[i to j],但是这与我们最优子结构性质 OPT 方程的定义相冲突。因此可验证算法的正确性。

(d) the complexity of algorithm:
对于长度为 n 的字符串，对于分割循环中的每一个步骤，复杂度为 O(n)，另外需要（n-1）的比较，所以时间复杂度为 O(n^2).

Solution2：

对于字符串 S，若最后一刀切在 j 处，则此时最小切为字符串 S[0:j] 的最小切再加一刀，即满足最优子结构。

对于长度为 N 的字符串 S，令 c[i] 为分割 S[0:i] 的最小切，则从 i 往前遍历判断 s[j:i] 是否为回文，对于所有满足 s[j:i] 为回文的 j，若 s[0:i] 是回文，则取 c[j] 为 0，否则则取 j 使得 c[j] 最小，可得 c[i]=c[j-1]+1。

最差情况下时间复杂度为 $O(N^2)$。

ALGORITHM 3: Partition

```
for i = 1; i ≤ n; i++ do
    c[i] =∞
    if S[0:i] is a palindrome then
        c[i]=0
    end
    else
        for j = 1; j ≤ i; j++ do
            if S[j:i] is a palindrome then
                c[i]=min(c[i],c[j-1]+1)
            end
        end
    end
end
```

T4：

A message containing letters from A-Z is being encoded to numbers using the following mapping:

A : 1
B : 2
...
Z : 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12). The number of ways decoding "12" is 2.

Solution1：

最优子结构及 DP 方程：

OPT[i]表示从第 i 个数字到第 n 个数字的所有方式，设第 i 个数字为 N[i]，其 DP 方程如下：

$$OPT[i]=\begin{cases} OPT[i+1]+OPT[i+2] & \text{i<n-1,N[i]=2 and N[i+1]=1~6 or N[i]=1 and N[i+1]}\neq 0; \\ OPT[i+2] & \text{i<n-1,N[i]=1,2 and N[i+1]=0;} \\ OPT[i+1] & \text{i<n-1,N[i]}\neq 1,2 \text{ or N[i]=2 and N[i+1]=7,8,9.} \end{cases}$$

由此可得，按上述方程来解决问题，需要从后往前遍历该数组。注意，上方程没写 OPT[n]和 OPT[n-1],这两个值可以通过枚举所有情况很容易得到。OPT[n]为 0 或 1，OPT[n-1]为 1 或 2.

算法描述及伪代码：

算法本身比较简单，我们只需要考虑当前元素以及当前元素的后一元素。比较复杂的是其边界判定条件，只有当第 i 个元素是 1 或 2 且第 i+1 个元素是 1~6 时，这里才有两种编码方式，当认为这两个元素对应两个字母时，OPT[i]=OPT[i+1]，当认为这两个元素对应同一个字母时，OPT[i]=OPT[i+2],因此总的 OPT[i]为两者相加。若当前元素为 0，则说明必定是 i-1 与此元素共同对应一个字母，所以此时 OPT[i]=OPT[i+1],如果第 i 个元素大于 2 或者 i+1 个元素大于 6，则对于该两个元素，只有一种固定的编码方式，OPT[i]=OPT[i+1].综上，可以总结出，只要不满足第一个条件的其他所以情况都可以归结为后一类。

伪代码：

```
InPut:N[]
(1) OPT[]=[0]*n
(2) if N[n]!=0:
        OPT[n]=1
    end if
(3) if N[n-1]==2 and N[n]>0 and N[n]<7 or N[n-1]==1 and N[n]!=0:
        OPT[n-1]=2
    else:
        OPT[n-1]=1
    end if
(4) for i=n-2 to 1 :
        if N[i-1]==2 and N[i]>0 and N[i]<7 or N[i-1]==1 and N[i]!=0:
            OPT[i]=OPT[i+1]+OPT[i+2]
        else:
            OPT[i]=OPT[i+1]
        end if
    end for
(5) return OPT[1]
OutPut:OPT[1]
```

算法正确性：

当 i=n 时，根据判断条件 OPT[n]=0 或 1，正确；

当 i=n-1 时，根据判断条件 OPT[n-1]=1 或 2，正确；

假设当 i=j+1,j+2 时，算法是正确的，及有 OPT[j+1],OPT[j+2]正确，那么当 i=j 时，根据上述判断条件，OPT[j]=OPT[j+1]+OPT[j+2]或者 OPT[j]=OPT[j+1].此时算法也正确，综上所述，该算法是正确的。

算法复杂度分析：

根据 DP 方程，容易看出 T[i]=T[i+1]+T[i+2],注意，此处由于数组遍历是倒序遍历，其相当于 T[n]=T[n-1]+T[n-2],只要遍历一次数组就可以得到所有 OPT[i],所以算法复杂度是 O(n).

Solution2：

Solution：分析整个情况可以得知，我们不妨从后向前讨论，如果L_i和L_{i+1}组成的数字<=26，说明，L_i选择和L_{i+1}组合变成一个两位数时，$\text{num}(L_i) = num(L_{i+2})$;当$L_i$选择作为一个独立的数字时，$\text{num}(L_i) = num(L_{i+1})$;所以$\text{num}(L_i) = num(L_{i+1}) + num(L_{i+2})$。再次我们假设 OPT(i)代表当前的$\text{num}(L_i)$

Sub-problem：计算当前$\text{num}(L_i)$

DP equation:

$$\text{OPT(i)} = \begin{cases} 1 & when\ i = n \\ OPT(i+1) + OPT(i+2) & when\ 'L_i' + 'L_{i+1}' \le 26 \\ OPT(i+1) & when\ L_i' + 'L_{i+1}' > 26 \\ 0 & when\ L_i = '0' \end{cases}$$

pseudo-code:

```
Decoding(L)
1: if len(L)!=0 and (len(L)!=1 and len(1)==’0’) then//判断是否至少有 1 个不为 0 的元素
2:   return numI=0
3: end if
4: num//创建一个长度为 n+1 的数组
5: for i=0 to len(L)-1://初始化该数组
6:   num[i]=0
7: end for
8: Num[len(L)]=1
9: if L[n-1] !=’0’ then
10:  num[n-1]=1
11:else
12:  num[n-1]=0
13:end if
14:for i=n-2 to 0 ://取 L 中的前 n-1 个元素
15:  if L[i]==’0’ then
16:    continue
17:  else if ‘L[i]’+’L[i+1]’ <=26 then
18:    num[i]+=num[i+2]
19:  end if
20:  num[i]+=num[i+1]
21:end for
```

```
22:return num[0]
```

Prove the correctness:

考虑到当L_i = '0'的情况，此时不仅不能增加当前的num(L_i)，反而要设置为 0，因为'0'不能作为单一的情况出现，前面必须有'1'或者'2'否则，最后返回的结果为 0；

考虑L_i和L_{i+1}组成的数字<=26，当的L_i选择和L_{i+1}组合变成一个两位数时，包含了全部的L_i后数两位的情况：num(L_i) = $num(L_{i+2})$；

当L_i选择作为一个独立的数字时，L_i包含了其后数 1 位的全部情况：num(L_i) = $num(L_{i+1})$；

综上所述，本算法正确。

Complexity:

由于从后往前计算且仅计算了一次，该算法的复杂度应该为 O(n)

但是开辟了一个数组 num，所以空间复杂度为 O(1)

T5：

You are given a sequence L and an integer k, your task is to find the longest consecutive subsequence the sum of which is the multiple of k.

目前没有同学发这一题，我简单写一下思路：

以 L=[1,2,3,4,5,6],K=3 为例

第一步先在序列前加 0（目的是为了保证边界计算正确）变为 L=[0,1,2,3,4,5,6]

第二步，根据 L 构造一个新的加和数列，S=[0,1,3,6,10,15,21],（第 i 个元素为前 i 个元素的和）。

第三步，根据加和数列 S 构造一个关于 k 的余数数列 R=[0,1,0,0,1,0,0]（第 i 个元素为 S[i]%k 的余数）k 的余数只有{0,1,2}三个值。

第四步，根据 k 的余数构造对应的数组，p0,p1,p2:

这里的 pi 代表余数是 i 的数组，pi 里面只存两个值，R 中第一次出现余数为 i 时的位置（从第 0 个位置开始算）和最后一次出现余数为 i 时的位置：

所以，本例中 p0=[0,6],p1=[1,4],p2=[](只要遍历一次 R 便能找出)

之后如果 pi 中元素个数为 2，计算 pi[1]-pi[0],最大的那个值就是所求的最长子串长度。

p0:6-0=6,　　p1:4-1=3,　　p2:无，　因此最长子串长度为 6.