Exercice 1 : Rendu de monnaie

On doit rendre la monnaie avec une variété de pièces limitée. Pour quelle première valeur de r l'algorithme glouton n'est-il pas optimal ?

1) Avec uniquement des pièces (1,4,5,10),
2) Avec uniquement des pièces (1,3,4,10),
3) Avec uniquement des pièces (1,3,4,5),

1) Non, mis en défaut avec $r = 8$
2) Non, mis en défaut avec $r = 6$
3) Non, mis en défaut avec $r = 7$

Exercice 2 : Rendue de monaie

La fonction *monnaie* ci-dessous utilise une liste x pour représenter les pièces rendues.

```
def monnaie(p,r):
    n=len(p)
    reste=r
    x=n*[0]
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        x[i]=x[i]+1
        reste=reste-p[i]
    return x
```

Ecrire une fonction qui utilise un dictionnaire dont les clés sont les valeurs des pièces rendues et les valeurs associées aux clés sont le nombre de pièces correspondantes.

```
def monnaie2(p,r):
    n=len(p)
    reste=r
    x={1:0,2:0,5:0,10:0,20:0,50:0,100:0,200:0,500:0}
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        x[p[i]]=x[p[i]]+1
        reste=reste-p[i]
    return x

def monnaie3(p,r):
    n=len(p)
    reste=r
    x={}
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        if p[i] in x:
            x[p[i]]=x[p[i]]+1
        else:
            x[p[i]]=1
        reste=reste-p[i]
    return x
```

Exercice 3 : Rendu de monnaie

La fonction *monnaie* ci-dessous utilise une liste x pour représenter les pièces rendues.

```
def monnaie(p,r):
    n=len(p)
    reste=r
    x=n*[0]
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        x[i]=x[i]+1
        reste=reste-p[i]
    return x
```

Reprendre le programme ci-dessus afin d'obtenir dans la variable résultat les valeurs des pièces rendues avec leur nombre sous la forme d'une liste de liste.

```
def monnaie2(p,r):
    n=len(p)
    reste=r
    x=[]
    for i in range(n):
        x.append([0,0])
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        x[i][0]=p[i]
        x[i][1]=x[i][1]+1
        reste=reste-p[i]
    return x
```

Exercice 4 : Rendu de monnaie

La fonction *monnaie* ci-dessous utilise un dictionnaire x pour représenter les pièces rendues.

```
def monnaie(p,r):
    n=len(p)
    reste=r
    x={}
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        if p[i] in x:
            x[p[i]]=x[p[i]]+1
        else:
            x[p[i]]=1
        reste=reste-p[i]
    return x
```

Modifier ce code afin que le résultat soit un dictionnaire dont chaque élément est un dictionnaire donnant la solution pour une valeur à rendre comprise entre 1 et r_{max} où r_{max} est un paramètre de la fonction qui remplace le paramètre r.

```
def monnaie2(p,r_max):
    n=len(p)
    x_final={}
    for j in range(1,r_max):
        x={}
        reste=j
        i=n-1
        while reste>0:
            while reste-p[i]<0:
                i=i-1
            if p[i] in x:
                x[p[i]]=x[p[i]]+1
            else:
                x[p[i]]=1
            reste=reste-p[i]
        x_final[j]=x
    return x_final
```

Exercice 5 : Rendu de monnaie

La fonction *monnaie* ci-dessous utilise une liste x pour représenter les pièces rendues.

```
def monnaie(p,r):
    n=len(p)
    reste=r
    x=n*[0]
    i=n-1
    while reste>0:
        while reste-p[i]<0:
            i=i-1
        x[i]=x[i]+1
        reste=reste-p[i]
    return x
```

Ecrire une version récursive prenant pour argument la liste p, le reste r ainsi que la liste x initialisée telle que $x = [0] * len(p)$.

```
def monnaie2(p,r,L):
    if r==0 or len(p)==0:
        return L
    else :
        if r-p[-1]<0:
            return monnaie2(p[:-1],r,L)
        else :
            r=r-p[-1]
            n=len(p[:-1])
            L[n]=L[n]+1
            return monnaie2(p,r,L)

def monnaie3(p,r,i):
    if r==0 :
        return []
    else :
        if r-p[i]<0:
            i=i-1
            return monnaie3(p,r,i)+[0]
        j=0
        while r-p[i]>=0 :
            r=r-p[i]
            j=j+1
            i=i-1
            return monnaie3(p,r,i)+[j]
```

Exercice 6 : Problème du sac à dos

On considère un ensemble de n objets i de masse m_i et de valeur v_i.

On travaille sur 3 listes :

- La liste des objets, ici $n = 4$ et $O = [1,2,3,4]$
- La liste des valeurs associées à ces objets $v = [4,3,1,9]$
- La liste des masses associés à ces objets $m = [3,2,1,4]$

Si, pour remplir le sac, on utilise un algorithme glouton cherchant à optimiser le rapport $\frac{v_i}{m_i}$ avec la contrainte $\sum m_i \leq M$ (où M est la masse maximale), alors le résultat global n'est pas le meilleur résultat.

On peut obtenir le résultat optimal à l'aide d'une version récursive (l'analyse n'est alors plus locale, car tous les cas seront étudiés).

- Pour chacun des objets, deux choix possibles : il est pris ou pas dans le sac,
- La récursivité s'effectuera sur la liste des indices O. Le 1e appel de la fonction se fera en utilisant l'indice n, puis l'indice $n-1$ et ainsi de suite jusqu'à l'indice 0 (correspondant au cas où il n'y a plus de produits)
- Pour $i \in [0,1,2,\dots,n]$ et $\omega \in [0,1,2,..M]$, on note $S(i,\omega)$ la valeur maximale cumulée des valeurs

$$S(i,\omega) = \begin{cases} 0 \text{ si } i = 0 \text{ correspond au cas trivial d'absence de produit} \\ S(i-1,\omega) \text{si } i > 0 \text{ et } p_i > \omega \text{ correspond au cas du dépassement de } P_{max} \Rightarrow \text{nvl appel sans ce produit} \\ \max\left(S(i-1,\omega), r_i + S(i-1,\omega-p_i)\right) \text{ si } i > 0 \text{ et } p_i \leq \omega \text{ correspond au cas du remplissage optimisé} \end{cases}$$

1) Définir une fonction *maximum* ayant pour arguments deux réels et renvoyant le maximum parmi ces réels.
2) Proposer un algorithme récursif prenant comme argument les listes m, v, un indice i (initialisé à 4), une masse ω (initialisée à $M = 8$) et renvoyant $S(i,\omega)$.
3) Calculer la valeur maximale transportable si $M = 8$.

```
def maximum(elt1,elt2):
    if elt1<elt2:
        return elt2
    else:
        return elt1
def recursive(m,v,i,w):
    if i==0:
        return 0
    else :
        if w<m[i-1]:
            return recursive(m[:-1],v[:-1],i-1,w)
        else :
            return maximum(recursive(m[:-1],v[:-1],i-
1,w),v[i-1]+recursive(m[:-1],v[:-1],i-1,w-m[i-1]))
print(recursive(m,v,4,8))
```

On a fait les appels suivants :

<table>
<tr><td colspan="14">Cas d'arrêt =>0</td></tr>
<tr><td>0</td><td>4</td><td>3</td><td>7</td><td>1</td><td>5</td><td>4</td><td>8</td><td>9</td><td>13</td><td>12+ recur([3], [4], 0,2))</td><td>10</td><td>14</td><td>13+ recur([3], [4], 0,1))</td></tr>
<tr><td colspan="2">recur([3], [4],1,8))</td><td colspan="2">3+ recur([3], [4],1,6))</td><td colspan="2">1+ recur([3], [4],1,7))</td><td colspan="2">1+3 recur([3], [4],1,5))</td><td colspan="2">9+ recur([3], [4],1,4))</td><td>9+3 recur([3], [4],1,2))</td><td colspan="2">10+ recur([3], [4],1,3))</td><td>10+3 recur([3], [4],1,1))</td></tr>
<tr><td colspan="4">recur([3,2], [4,3],2,8))</td><td colspan="4">1+ recur([3,2], [4,3],2,7))</td><td colspan="4">9+ recur([3,2], [4,3],2,4))</td><td colspan="3">9+1+ recur([3,2], [4,3],2,3))</td></tr>
<tr><td colspan="8">recur([3,2,1], [4,3,1],3,8))</td><td colspan="6">9+ recur([3,2,1], [4,3,1],3,4))</td></tr>
<tr><td colspan="14">recur([3,2,1,4], [4,3,1,9],4,8))</td></tr>
</table>

Exercice 7 : Station essence(**)

Un véhicule peut parcourir une distance d avec un réservoir plein. La route empruntée comporte n stations services s_i $(s_1, s_2, \ldots s_{n-1})$ rangée dans l'ordre rencontrée, la première est à une distance d_1 du départ et la 2e à une distance d_2 de la première et ainsi de suite. Nous supposerons pour toutes les distances que $d_i < d$.

On propose la liste suivante donnant la distance entre chaque station et la station précédente :

```
liste=[["départ",0],
       ["station1",500],
       ["station2",200],
       ["station3",300],
       ["station4",400],
       ["station5",800],
       ["station6",500]]
```

On donne l'algorithme itératif permettant d'obtenir la liste des stations qu'il est nécessaire de visiter et la distance totale parcourue (liste est la liste des stations avec le distance et d est la distance maximale pouvant parcourir avec un plain)

```
def essence(liste,d):
    distance_possible=d
    distance_tot=0
    liste_stations_visitées=[]
    i=1
    while i < len(liste):
        while i<len(liste) and distance_possible -
liste[i][1]>=0:
            distance_possible=distance_possible-liste[i][1]
            distance_tot=distance_tot+liste[i][1]
            i=i+1
        liste_stations_visitées.append(liste[i-1][0])#on
s'arrête à la station d'avant
        distance_possible=d#on remet les compteurs à zero
    return liste_stations_visitées,distance_tot
```

1) Prévoir ce que retourne la fonction précédente avec l'appel *essence*(*liste*, 800)
2) Ecrire la version récursive du programme ci-dessus (on pourra introduire des arguments supplémentaires)

Réponse :
(['station2', 'station4', 'station5', 'station6'], 2700)

```
def essence2(liste,distance_max,i,d):#initialisation à i=0
    n=len(liste)
    if i==n:
        return []
    else :
        distance_possible=distance_max
        while i<n and distance_possible-liste[i][1]>=0:
            distance_possible=distance_possible-liste[i][1]
            d=d+liste[i][1]
            i=i+1
        return [i-1,d]+essence2(liste,distance_max,i,d)
```