Informatique — MPSI/PCSI, 2024–2025

Correction du TP n° 5

1 Rendu de monnaie

Exercice 5.1

1. La stratégie gloutone consiste à maximiser le rendu des grandes valeurs de la liste `P`. Ainsi, il est nécessaire de partir de la fin et d'épuiser la somme à rendre tant que c'est possible avant de passer à la valeur inférieure. Une première idée peut être d'utiliser une boucle `while` dont la condition d'arrêt est l'annulation de la somme à rendre, à l'intérieure de laquelle on trouve la valeur à rendre en partant de la fin, mais sans jamais revenir en arrière (de `k=len(P)-1` à $k \geqslant 0$). Ainsi, sortant de la boucle `while` imbriquée, on a forcément $s \geqslant P_k$ et donc on peut ajouter P_k à la liste des valeurs à rendre `M` et retrancher cette valeur à s pour mettre à jour la somme restant à rendre. Il vient alors :

```
def rendu_monnaie_0(s,P):
    M=[]
    k=len(P)-1
    while s!=0:
        while k>0 and s<P[k]:
            k-=1
        s-=P[k]
        M.append(P[k])
    return(M)
```

Ainsi, avec $[c_0, c_1, \dots, c_{n-1}]$ la séquences de n valeurs classées par ordre croissant et notant S_0 la somme à rendre, il vient la relation de récurrence :

$$\text{tant que } k < n \text{ et } S_k \neq 0, \quad S_{k+1} = S_k - \left\lfloor \frac{S_k}{c_{n-1-k}} \right\rfloor \times c_{n-1-k}$$

où $\left\lfloor \dfrac{S_k}{c_{n-1-k}} \right\rfloor \in \mathbb{N}$ donne le nombre d'instances de la valeurs c_{n-1-k} à rendre. Dans tous les cas, on doit avoir $k < n$ car la dernière somme ne peut être que celle amputée des c_0, c'est-à-dire avec $n-1-k=0 \iff k+1=n$. Notant alors $t = n-1-k$, partant de $t = n-1$, il vient le code suivant :

```
def rendu_monnaie_1(s,P):
    M=[]
    t=len(P)-1
    while k>=0 and s!=0:
        s=(s%P[t])
        for j in range(s//P[t]):
            M.append(P[t])
        t-=1
    return(M)
```

2. Pour tester la fonction avec les exemples du sujet, on définit le code :

```
EUR = [i*10**e for e in range(3) for i in [1,2,5]]
print(rendu_monnaie_1(49,EUR))
print(rendu_monnaie_1(76,EUR))
```

qui renvoie

```
[20, 20, 5, 2, 2]
[50, 20, 5, 1]
```

3. Pour prendre en compte les centimes, il faut faire attention à la représentation des nombres. Sachant que les flottants ne permettent pas de faire du calcul exact, on décide de procéder aux calculs en centimes pour conserver des nombres entiers. Pour ce faire, on commence par multiplier par 100 la somme à rendre et définir une nouvelle séquence `PI` avec les valeurs de la séquence `P` multipliées par 100, forçant la conversion en entier avec la fonction `int` (qui, pour des nombres positifs renvoie la partie entière). Aussi, pour éviter toute erreur d'arrondi, on définit une fonction :

```
def arrondi(f):
    i = int(f)
    if f-i>.5:
        i+=1
    return(i)
```

qui renvoie l'entier `i` le plus proche du flottant `f`. De même, pour éviter toute erreur d'arrondi avec une division par 100, on ajoute les valeurs de la séquence initiale des `P` (de même indice) et non de la séquence `PI`. Le code suivant convient.

```
def rendu_monnaie_3(s,P):
    s = arrondi(100*s)
    PI = [arrondi(100*e) for e in P]
    M=[]
    t=len(P)-1
    while k>=0 and s!=0:
        s=(s%PI[t])
        for j in range(s//PI[t]):
            M.append(P[t])
        t-=1
    return(M)
```

4. Pour tester la fonction avec les exemples du sujet, on définit le code :

```
EUROC = [i*10**e for e in range(-2,3) for i in [1,2,5]]
print(rendu_monnaie_3(13.45,EURO))
print(rendu_monnaie_3(25.58,EURO))
```

qui renvoie

```
[10.0, 2.0, 1.0, 0.2, 0.2, 0.05]
[20.0, 5.0, 0.5, 0.05, 0.02, 0.01]
```

5. Avec l'exemple

```
>>> rendu_monnaie_1(49,[1, 3, 6, 12, 24, 30])
[30, 12, 6, 1]
```

il est clair que la somme des optima locaux ne correspond pas à un optimum global qui minimiserait le nombre de valeurs à rendre, ici `[24, 24, 1]`.

6. Avec l'exemple

```
>>> rendu_monnaie_1(7,[2,3])
[3, 3]
```

on voit même qu'il est possible d'escroquer un client en ne rendant pas correctement la monnaie : ici $3+3=6 \ll 7$! On notera donc que le choix des valeurs du système monétaire est d'une importance cruciale ; au minimum avoir les valeurs 1 et 0,01 pour éviter les escroqueries !

2 Le problème du sac à dos

Exercice 5.2 (Force brute)

1. Soit `S` une séquence de longueur $n > 1$. Pour générer tous les n-uplets possibles contenant 0 (absence) ou 1 (présence), il est nécessaire d'avoir une représentation sur n-bits. Or `000....01` se représente sur 1 bit. Par exemple :

```
>>> a=[print(bin(k)) for k in range(2**4)]
0b0
```

```
0b1
0b10
0b11
0b100
0b101
0b110
0b111
0b1000
0b1001
0b1010
0b1011
0b1100
0b1101
0b1110
0b1111
```

Pour avoir systématiquement une représente sur n bits, une astuce peut être de rajouter 2^n puis de tronquer aux n derniers bits. Par exemple :

```
>>> a=[print(bin(2**4+k)[3:]) for k in range(2**4)]
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

où nous en avons profité pour couper la chaîne `'0b'` en amont. Sachant enfin que la fonction `list` permet de convertir toute séquence (dont de caractères) en liste et qu'il faut convertir chaque caractère `'0'` ou `'1'` en nombre entier avec la fonction `int`, il vient finalement la fonction :

```
def ens_des_parties(objets):
    return([[int(e) for e in list(bin(k)[3:])]
        for k in range(2**len(objets),2**(1+len(objets)))])
```

2. On commence par définir une fonction qui fait la somme d'une colonne (valeur ou masse) en fonction de la liste.

```
def somme(objets, liste, indice):
    s=0
    for k in range(len(objets)):
        s+=objets[k][indice]*liste[k]
    return(s)
```

On définit ensuite comme proposé dans le sujet deux fonctions qui calculent respectivement la valeur totale et la masse totale d'une configuration donnée par une liste

```
def valeur_totale(objets, liste):
    return(somme(objets, liste, 1))
def masse_totale(objets, liste):
    return(somme(objets, liste, 2))
```

la différence se faisant seulement sur l'indice. Ensuite, pour déterminer par force brute la liste d'objets qui permet de maximiser la valeur tout en respectant l'exigence de masse maximale, on utilise la

méthode du candidat que l'on initialise à la première configuration, correspondant volontairement à un sac à dos vide. Le code suivant convient.

```
def force_brute(objets, masse_max):
    C = ens_des_parties(objets)
    c,v = C[0],0 # 1er candidat, sac vide
    for liste in C[1:]:
        if masse_totale(objets, liste) <= masse_max \
            and valeur_totale(objets, liste)>v:
            c = liste
            v = valeur_totale(objets, c)
    return([e[0] for e,f in zip(objets,c) if f==1],v)
```

On notera l'évaluation de la valeur que si la masse est inférieure à la masse maximale (évaluation paresseuse des booléens). Enfin, pour élaborer la liste d'objets, nous avons utilisé la fonction `zip` qui permet de parcourir simultanément plusieurs séquences de même longueur et l'ajout d'un élément si `f==1`. Ce que l'on aurait pu écrire de façon beaucoup plus laborieuse :

```
    S=[]
    for k in range(len(objets)):
        if c[k]==1:
            S.append(objets[k][0])
    return(S,v)
```

qui est équivalente à

```
    return([objets[k][0] for k in range(len(objets)) if c[k]==1],v)
```

et donc à

```
    return([e[0] for e,f in zip(objets,c) if f==1],v)
```

3. Pour les deux exemples proposés, il vient :

```
>>> force_brute(Sac1, 20)
[1, 3, 4]
>>> force_brute(Sac2, 40)
[3, 5, 6, 7]
```

Exercice 5.3 (Stratégie gloutonne)

1. Pour appliquer la stratégie glouton selon le critère de masse, on commence par trier la liste d'objets par masse décroissante. On remplit ensuite le sac à dos avec les objets du plus au moins lourd tant qu'il reste des objets assez légers et que la masse totale ne dépasse pas la masse maximale spécifiée. Quand un objet est trop lourd pour être ajouté, on passe au suivant. Avec cette description, on a donc un parcours des objets avec une boucle `for` et une condition sur la masse pour l'ajout. En définissant deux fonctions :

```
def masse(objet):
    return(objet[2])
def valeur(objet):
    return(objet[1])
```

selon le crière choisi, il vient le code :

```
def glouton(objets, masse_max, choix):
    M=[]
    mr = masse_max # masse restante à minimiser mais à garder positive
    F = sorted(objets, key=lambda x:choix(x), reverse=True)
    k = 0
    for objet in F:
        mo = masse(objet)
        if mo<=mr:
            M.append(objet[0])
            mr -= mo
    return(M)
```

2. Avec ce code, on obtient :

```
>>> glouton(Sac1, 20, masse)
[2, 4]
>>> glouton(Sac1, 20, valeur)
[4, 2]
>>> glouton(Sac2, 40, masse)
[6, 2, 1]
>>> glouton(Sac2, 40, valeur)
[7, 6, 3, 5]
```

On voit clairement que la stratégie gloutonne ne permet pas de trouver un optimum global, contrairement à la force brute. Elle a par contre une durée de calcul bien plus faible.

* *
*