Devoir n°1

Problème du sac à dos

Étant donné un sac à dos de volume total V, n types d'objets, numérotés de 1 à n, avec n_i objets de type i disponibles, de volume v_i et d'une utilité u_i pour l'objet de type i.

On souhaite remplir le sac en maximisant l'utilité des objets que l'on y met.

On définit x_i comme le nombre d'exemplaires de l'objets i mis dans le cas.
Les applications numériques se feront avec l'exemple suivant : $n = 4$, $V = 15$:

objet	1	2	3	4
v_i	3	5	2	2
n_i	2	2	3	4
u_i	10	15	9	7

Attention, dans les algorithmes, l'indice i représentant le numéro d'objet variera entre 0 et $n-1$, alors que les objets sont indexés entre 1 et n.

1. Méthode gloutonne extrême.

   Le glouton extrême ne voit que le gain immédiat, il choisit le type d'objet le plus utile, et en met le plus possible, et il recommence avec ce qui reste avec le même principe.

   Sur l'exemple, quel est le sac et l'utilité alors obtenue ? Est-elle optimisée ?

   Pouvez-vous imaginer un autre exemple, où cette méthode gloutonne donne tout de même une solution optimale ?

2. Méthode gloutonne avec retenue.

   Il voit le gain, mais veut aussi tenir compte du volume, et donc il considère les rapports u_i/v_i, c'est à dire l'utilité relative au volume, et son attention gloutonne vise cette quantité. Il applique alors avec ces gros yeux le même principe que dans la question précédente.

   Sur l'exemple, quel est le sac et l'utilité alors obtenue ? Est-elle optimisée ?

3. On souhaite éviter d'utiliser la fonction partie entière qui opère sur un flottant. On commence par écrire une fonction qui détermine le nombre maximum d'objets de même volume vo que l'on peut mettre dans un volume v donné. Compléter le code suivant :

```
def n_max(vo, v):
    n = 0  # initialisation nombre d'objets
    sv = 0 # somme des volumes des n objets
    while sv <= ?:
        n += 1
        sv += ?
    return ?
```

Tester bien votre code pour que cette fonction appliquée par exemple à (2,15) renvoie 7.

4. On procède par récursivité, en mettant l'utilité à -1 pour marquer un type d'objet déjà choisit, avant de poursuivre par appel récursif.
On donne le squelette de la fonction gloutonne extrême à compléter :

```
def glouton_extreme(ln, lv, lu, V):
    # recherche d'utilité maximum
    n = len(ln)
    u_max = lu[0]  # initialisation
    i_max = 0      # initialisation

    for i in range(1,n):
        if lu[i] > u_max :
            i_max= ?
            u_max = ?
    if u_max == -1: # ?
        return
    nb = n_max(lv[i_max], V) # nombre possible en volume
    # on renvoie le numéro d'objet et le nombre
    k = min(nb, ?)
    if k != 0 : #
        print("type "+str(i_max+1)+" : "+str(k)+" objet(s)")
    # on met à jour les données pour la récursivité
    V -= ? # volume restant
    lu[i_max] = -1 # type déjà utilisé
    glouton_extreme(ln, lv, lu, V)
```

Exécuter le code sur l'exemple (faire un tableau de suivi des variables et de l'affichage).
Quels sont les inconvénients d'une telle programmation ?

5. On donne le squelette de la fonction gloutonne avec retenue à compléter. Ici, on ne procède pas par récursivité, afin de résoudre les problématiques vues dans le code précédent :

```
def glouton_retenue(ln, lv, lu, V):
    n = len(ln)
    # tableau des [u_i/v_i,i]
    lr = [[?,i] for i in range(n)]
    # on trie par décroissance selon la première coordonnée
    lr.sort(reverse = True) # tri en place
```

```
u = ? # initialisation utilité totale
v = ? # initialisation volume restant
liste = [] # initialisation liste des [objets, nombre]
for i in range(n):
    indice = ?
    vo = ?
    nb = min(n_max(vo, v), ln[indice])
    if nb != 0 and nb*lv[indice] <= ? : # on les r(m)ange
        liste.append([indice, nb])
        v -= ? # mise à jour volume
        u += ? # mise à jour utilité
# on renvoie la liste, le volume pris et l'utilité
return liste, ?, u
```

6. Exprimer l'utilité totale du sac noté $U(x_1, \ldots, x_n)$ si on a donc choisit x_i objets de type i, $1 \leqslant i \leqslant n$.

7. Quel le volume $V(x_1, \ldots, x_n)$ utilisé avec la configuration $(x_1, \ldots, x_n)$? Exprimer alors la contrainte de volume.

On définit alors la quantité qui va permettre la programmation dynamique. On note $u(i, v)$ l'utilité maximale d'un sac de volume v en utilisant les types d'objets de 1 à i, c'est à dire la valeur maximale de $U(x_1, \ldots, x_i)$ avec la contrainte v.

8. Quelle est la quantité qui permet de résoudre le problème du sac à dos de volume V avec les types d'objets de 1 à n ?

9. Montrer que le nombre maximum d'objets de type 1 que l'on peut mettre dans le sac de volume v est l'entier (notation partie entière)
$$\left\lfloor \frac{v}{v_1} \right\rfloor$$
calculé par l'instruction n_max(lv[0], v) où lv sera le tableau des volumes.

   Sachant qu'il y a n_1 objets de type 1 au maximum, en déduire que
$$f(1, v) = \min\left(\left\lfloor \frac{v}{v_1} \right\rfloor, n_1 \right) u_1$$

10. Soit i, avec $2 \leqslant i \leqslant n$. Soit un choix optimisé $(x_1, \ldots, x_i)$ associé aux objets de type de 1 à i, pour un volume maximum v, montrer que le choix des objets $(x_1, \ldots, x_{i-1})$ de 1 à $i-1$, pour le volume $v - x_i v_i$ est optimisé.

11. En déduire que
$$f(i, v) = \max_{k=0 \cdots \min\left(\left\lfloor \frac{v}{v_i} \right\rfloor, n_i\right)} f(i-1, v - k v_i) + k u_i$$
   On expliquera soigneusement le rôle de chaque terme et on citera précisément la notion utilisée.

12. La formule de récurrence ci-dessus incite à écrire une fonction récursive. On donne le squellette à compléter et commenter :

```
def sac_a_dos(ln, lv, lu, V):
    n = len(ln)
    return sac_a_dos_r(ln, lv, lu, n, V)

def sac_a_dos_r(ln, lv, lu, i, v):
    if i == 1:
        return min(n_max(lv[0], v),ln[0])*lu[0]
    u_max = -1
    K = min(n_max(lv[i-1], v), ln[i-1])
    for k in range(?):
        u = sac_a_dos_r(ln,lv,lu,?,v-k*lv[?]) + k*lu[?]
        if u > u_max:
            u_max = ?
    return u_max
```

Pourquoi la récursivité n'est pas efficace ? On ne demande pas de faire un calcul de la complexité (pourquoi ?).

13. On se propose donc d'utiliser la formule de récurrence par une méthode dynamique. Compléter le code suivant :

```
def sac_a_dos_dyn(ln, lv, lu, V):
    n = len(ln)
    f = [[0]*(V+1) for _ in range(n)] # tableau valeurs de f
    for v in range(1, ?): # v correspond au volume
        f[0][v] = ?
    for i in range(1,n):
        for v in range(1, ?):
            u_max = -1
            K = min(n_max(lv[i], v), ?)
            for k in range(?):
                u = f[?][v-k*lv[?]] + k*lu[i]
                if u > u_max:
                    u_max = ?
            f[i][v] = ?
    return f[?][?]
```

Quelle remarque peut-on faire sur la longueur du code de la méthode dynamique par rapport aux autres méthodes ?

14. Appliquer la méthode dynamique sur l'exemple. Le tableau de f sera

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1																
2																
3																58

Remarques : les deux premières lignes à remplir se remplissent assez facilement sans forcément utiliser la formule de récurrence.
Les premières colonnes sont aussi assez évidentes.
Il y a croissance des valeurs lorsque i croît et aussi lorsque v croît.

15. Il s'agit maintenant de déterminer une composition optimale du sac à dos. Le tableau de f ne permet pas d'obtenir une telle composition. Nous devons construire en parallèle le tableau noté tk des valeurs de k obtenues pour effectuer le calcul de $f(i, v)$. Compléter alors le code suivant :

```
def sac_a_dos_dyn(ln, lv, lu, V):
    n = len(ln)
    f = [[0]*(V+1) for _ in range(n)]
    tk = ?
    for v in range(V+1):
        f[0][v] = ?
        tk[0][v] = ?
    for i in range(1,n):
        for v in range(V+1):
            u_max = -1
            K = min(n_max(lv[i], v), ln[i])
            for k in range(?):
                u = f[i-1][v-k*lv[?]]+k*lu[i]
                if u > u_max:
                    u_max = ?
                    kk = ?
            f[i][v] = ?
            tk[i][v] = ? # indice

    return f, tk, f[?][?]
```

16. Maintenant que ce tableau tk est construit, complèter le code précédent pour effectuer un backtracing permettant d'obtenir une composition optimale du sac à dos.

17. Appliquer le backtracing à l'exemple, le tableau des valeurs de k étant :

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2
1	0	0	0	0	0	1	0	0	1	1	2	1	1	2	2	2
2	0	0	1	0	2	1	3	2	1	3	2	3	3	3	3	2
3	0	0	0	0	0	0	0	0	1	0	2	1	3	2	4	3

Exercice : problème du sac à dos

Les applications numériques se feront avec l'exemple suivant : $n = 4$, $V = 15$.

objet	1	2	3	4
v_i	3	5	2	2
n_i	2	2	3	4
u_i	10	15	9	7

1. Méthode gloutonne extrême. Monsieur glouton extrême ne voit que le gain immédiat, il choisit le type d'objet le plus utile soit le 2, et en met le plus possible soit les 2 disponibles, qui occupent alors le volume 10 sur les 15 disponibles.
   Il reste un volume de 5, il prend alors 1 objet de type 1, reste un volume de 2, puis 1 objet de type 3.
   L'utilité obtenue est alors $U = 2 \times 15 + 1 \times 10 + 1 \times 9 = 49$.

   Elle n'est pas optimale : en prennant les 3 objets de type 3 et les 4 objets de type 4, on obtient un volume admissible de 14, avec une utilité de $3 \times 9 + 4 \times 7 = 27 + 28 = 55$.

   Cette méthode donne une solution optimale si par exemple (c'est le plus simple) nous disposons d'un seul type d'obet.

2. Méthode gloutonne avec retenue. On complète le tableau avec les quantité u_i/v_i :

objet	1	2	3	4
v_i	3	5	2	2
n_i	2	2	3	4
u_i	10	15	9	7
u_i/v_i	10/3=3.33	15/5=3	9/2=4.5	7/2=3.5

   Ici, on choisit l'objet 3 en premier. On peut prendre les 3, pour un volume de 6, et reste un volume disponible de 9. Ensuite, on choisit l'objet 4, on peut prendre les 4, pour un volume de 8, il reste alors 1 volume, et c'est fini.
   Nous obtenons une utilité de 55.

   Cette méthode n'est pas optimale non plus, nous pouvons choisir 1 objet 1, 3 objets 3, et 1 de type 4, volume de $3 + 6 + 6 = 15$ et une utilité de $10 + 3 \times 9 + 3 \times 7 = 10 + 21 + 27 = 58$.

3. Par exemple

```
def n_max(vo, v):
    n = 0  # initialisation nombre d'objets
    sv = 0 # somme des volumes des n objets
    while sv <= v
        n += 1
```

```
        sv += vo
    return n-1 # attention
```

ou encore (mais on calcule v-vo à chaque tour de boucle, il faudrait le calculer une fois avant et stocker le résultat dans une variable)

```
def n_max(vo, v):
    n = 0  # initialisation nombre d'objets
    sv = 0 # somme des volumes des n objets
    while sv <= v-vo
        n += 1
        sv += vo
    return n
```

4. On complète le squelette de la fonction gloutonne extrême :

```
def glouton_extreme(ln, lv, lu, V):
    # recherche d'utilité maximum
    n = len(ln)
    u_max = lu[0]  # initialisation
    i_max = 0      # initialisation

    for i in range(1,n):
        if lu[i] > u_max :
            i_max= i
            u_max = lu[i]
    if u_max == -1: # plus de types d'objets possibles, fini
        return
    nb = n_max(lv[i_max], V) # nombre possible en volume
    # on renvoie le numéro d'objet et le nombre
    k = min(nb, ln[i_max])
    if k != 0 :
        print("type "+str(i_max+1)+" : "+str(k)+" objet(s)")
    # on met à jour les données pour la récursivité
    V -= k*lv[i_max]
    lu[i_max] = -1
    glouton_extreme(ln, lv, lu, V)
```

La fonction affiche à l'écran les objets choisis, mais les résultats ne sont pas stockés. Nous avons une complexité en $\mathcal{O}(n^2V)$ (n appels récursifs, et dans chacun, une boucle de taille n, la fonction n_max pouvant être considéré à temps constant $\mathcal{O}(V)$).

5. On complète le squelette de la fonction gloutonne avec retenue :

```
def glouton_retenue(ln, lv, lu, V):
    n = len(ln)
    # tableau des [u_i/v_i,i]
    lr = [[lu[i]/lv[i],i] for i in range(n)]
    # on trie par décroissance selon la première coordonnée
```

```
    lr.sort(reverse=True) # trie en place
    u = 0 # initialisation utilité totale
    v = V # initialisation volume restant
    liste = [] # initialisation liste des [objets, nombre]
    for i in range(n):
        indice = lr[i][1]
        vo = lv[indice]
        nb = min(n_max(vo, v), ln[indice])
        if nb != 0 and nb*lv[indice] <= v : # on les r(m)ange
            liste.append([indice, nb])
            v -= nb*lv[indice] # mise à jour volume
            u += nb*lu[indice] # mise à jour utilité
    # on renvoie la liste, le volume pris et l'utilité
    return liste, V-v, u
```

6. L'utilité totale du sac noté $U(x_1, \ldots, x_n)$ est

$$U(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i u_i$$

7. Le volume $V(x_1, \ldots, x_n)$ utilisé avec la configuration $(x_1, \ldots, x_n)$ est

$$V(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i v_i$$

La contrainte de volume est donnée par

$$V(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i v_i \leqslant V$$

8. La quantité qui permet de résoudre le problème du sac à dos de volume V avec les types d'objets de 1 à n est donc $f(n, V)$.

9. Le nombre maximum d'objets de type 1 que l'on peut mettre dans le sac de volume v est l'entier

$$k = \left\lfloor \frac{v}{v_1} \right\rfloor$$

puisque

$$pv_1 \leqslant v \iff p \leqslant \frac{v}{v_1}$$

et que k désigne le plus grand entier inférieur ou égal à $\frac{v}{v_1}$.

Sachant qu'il y a n_1 objets de type 1 au maximum, on a donc le nombre qui est le minimum de k et de n_1, et ainsi l'utilité asssociée est

$$f(1, v) = \min\left(\left\lfloor \frac{v}{v_1} \right\rfloor, n_1 \right) u_1$$

10. Soit i, avec $2 \leqslant i \leqslant n$. Soit un choix optimisé $(x_1, \ldots, x_i)$ associé aux objets de type de 1 à i, pour un volume maximum v, le choix des objets $(x_1, \ldots, x_{i-1})$ de 1 à $i-1$, pour le volume $v - x_i v_i$ est forcément optimisé par un argument de **sous-problème optimal**. En effet, sinon, si on pourrait faire mieux avec les objets $(x'_1, \ldots, x'_{i-1})$ associé aux objets de type de 1 à $i-1$ et un volume de $v - x_i v_i$, le choix $(x'_1, \ldots, x'_{i-1}, x_i)$ associé aux objets de type de 1 à i et un volume de v serait meilleur que le choix $(x_1, \ldots, x_i)$ ce qui contredit le caractère optimisé du choix $(x_1, \ldots, x_i)$.

11. On en déduit ainsi que

$$f(i, v) = \max_{k=0 \cdots \min\left(\left\lfloor \frac{v}{v_i} \right\rfloor, n_i\right)} f(i-1, v - kv_i) + ku_i$$

On explique le rôle de chaque terme :

$$\min\left(\left\lfloor \frac{v}{v_i} \right\rfloor, n_i\right)$$

est le nombre maximum K d'objets de type i que l'on peut utiliser pour remplir un volume v_i, avec la contrainte n_i d'où attention, le minimum des 2 valeurs.
Nous devons alors maximiser l'utilité, maximum entre toutes les valeurs

$$f(i-1, v - kv_i) + ku_i$$

où $0 \leqslant k \leqslant K$, $f(i-1, v - kv_i)$ étant l'utilité optimisée pour les types de 1 à $i-1$, pour un volume restant de $v - kv_i$, conformément à la question précédente, ku_i étant l'utilité des objets de type i si on en choisit k.

12. La formule de récurrence ci-dessus incite à écrire une fonction récursive :

```
def sac_a_dos(ln, lv, lu, V):
    n = len(ln)
    return sac_a_dos(ln, lv, lu, n, V)

def sac_a_dos_r(ln, lv, lu, i, v):
    if i == 1:
        return min(n_max(lv[0], v),m[0])*lu[0]
    u_max = -1
    K = min(n_max(lv[i-1], v), lm[i-1])
    for k in range(K+1):
        u = sac_a_dos(lm, lv, i-1, v-k*lv[i-1]) + k*lu[i-1]
        if u > u_max:
            u_max = u
    return u_max
```

La récursivité n'est pas efficace vu les multiples appels récursifs et les problématiques de chevauchement. On ne demande pas de faire un calcul de la complexité car elle est peut être très complexe à calculer (cf devoir maison pour un cas beaucoup plus simple).

13. On se propose donc d'utiliser la formule de récurrence par une méthode dynamique.

```
def sac_a_dos_dyn(ln, lv, lu, V):
    n = len(ln)
    tf = [[0]*(V+1) for _ in range(n)]
    for v in range(1, V+1):
        f[0][v] = min(n_max(lv[0], v), ln[0])*lu[0])
    for i in range(1,n):
        for v in range(1, V+1):
            u_max = -1
            K = min(n_max(lv[i], v), ln[i])
            for k in range(K+1):
                u = f[i-1][v-k*lv[i]] + k*lu[i]
                if u > u_max:
                    u_max = u
            f[i][v] = u_max
    return f[n-1][V]
```

La boucle interne en k est de taille K, majoré par $\max(n_1, \dots, n_n) = N$ et on considère le corps de cette boucle en temps constant, mais en prenant en compte n_max, en $\mathcal{O}(V)$.
La boucle supérieure en v, de taille V.
La boucle supérieure en i, de taille n.
La première boucle en v de taille V est indépendante des 3 autres qui sont imbriquées, on obtient une complexité majorée par nV^2N.

14. On applique la méthode sur l'exemple :

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	10	10	10	20	20	20	20	20	20	20	20	20	20
1	0	0	0	10	10	15	20	20	25	25	30	35	35	40	40	40
2	0	0	9	10	18	19	27	28	29	37	38	42	47	47	52	53
3	0	0	9	10	18	19	27	28	34	37	41	44	48	51	55	58

15. Il s'agit maintenant de déterminer une composition optimale du sac à dos. Le tableau de f ne permet pas d'obtenir une telle composition. Nous devons construire en parallèle le tableau noté tk des valeurs de k obtenues pour effectuer le calcul de $f(i, v)$. Compléter alors le code suivant :

```
def sac_a_dos_dyn(ln, lv, lu, V):
    n = len(ln)
    f = [[0]*(V+1) for _ in range(n)]
    tk = [[0]*(V+1) for _ in range(n)]
    for v in range(1, V+1):
        f[0][v] = min(n_max(lv[0], v), ln[0])*lu[0]
        tk[0][v] = min(n_max(lv[0], v), ln[0])
    for i in range(1,n):
        for v in range(1, V+1):
            u_max = -1
            K = min(n_max(lv[i], v), ln[i])
            for k in range(K+1):
```

```
                u = f[i-1][v-k*lv[i]]+k*lu[i]
                if u > u_max:
                    u_max = u
                    kk = k
            f[i][v] = u_max
            tk[i][v] = kk # indice du maximum
    return f, tk, f[n-1][V]
```

16. On complète le code précédent pour effectuer un backtracing permettant d'obtenir une composition optimale du sac à dos :

```
def sac_a_dos_dyn(ln, lv, lu, V):
    n = len(ln)
    f = [[0]*(V+1) for _ in range(n)]
    tk = [[0]*(V+1) for _ in range(n)]
    for v in range(V+1):
        f[0][v] = min(n_max(lv[0], v), ln[0])*lu[0]
        tk[0][v] = min(n_max(lv[0], v), ln[0])
    for i in range(1,n):
        for v in range(V+1):
            u_max = -1
            K = min(n_max(lv[i], v), ln[i])
            for k in range(K+1):
                u = f[i-1][v-k*lv[i]]+k*lu[i]
                if u > u_max:
                    u_max = u
                    kk = k
            f[i][v] = u_max
            tk[i][v] = kk # indice
    # backtracing
    liste = []  # liste des [numero, nombre]
    j = n - 1
    v = V
    while j >= 0:
        k = tk[j][v]
        if k != 0 :  # sinon inutile
            liste = [[j, k]] + liste
        v -= k*lv[j]
        j -= 1

    return f, tk, f[n-1][V], liste
```

Sur l'exemple, le tableau des valeurs de k :

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2
1	0	0	0	0	0	1	0	0	1	1	2	1	1	2	2	2
2	0	0	1	0	2	1	3	2	1	3	2	3	3	3	3	2
3	0	0	0	0	0	0	0	0	1	0	2	1	3	2	4	3

On part de la valeur $f(3, 15) = 58$, $k = 3$, soit 3 objets 4, de volume 6.
Le volume restant est 15-6 = 9.

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2
1	0	0	0	0	0	1	0	0	1	1	2	1	1	2	2	2
2	0	0	1	0	2	1	3	2	1	3	2	3	3	3	3	2
3	0	0	0	0	0	0	0	0	1	0	2	1	3	2	4	3

On regarde la case $(2, 9)$, $f(2, 9) = 37$, $k = 3$, soit 3 objets 3, de volume 6.
Le volume restant est $9 - 6 = 3$.

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2
1	0	0	0	0	0	1	0	0	1	1	2	1	1	2	2	2
2	0	0	1	0	2	1	3	2	1	3	2	3	3	3	3	2
3	0	0	0	0	0	0	0	0	1	0	2	1	3	2	4	3

On regarde la case $(1, 3)$, $f(1, 3) = 10$, $k = 0$, soit 0 objet 2.
Le volume restant est encore 3.

i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2
1	0	0	0	0	0	1	0	0	1	1	2	1	1	2	2	2
2	0	0	1	0	2	1	3	2	1	3	2	3	3	3	3	2
3	0	0	0	0	0	0	0	0	1	0	2	1	3	2	4	3

On regarde la case $(0, 3)$, $f(0, 3) = 10$, $k = 1$, soit 1 objet 1, de volume 3.
Nous avons un sac composé de 1 objet 1, 3 objets 3 et 3 objets 4, soit un volume de $3+3.2+3.2 = 15$, et d'une utilité de $1.10 + 3.9 + 3.7 = 10 + 27 + 21 = 58$.