Informatique PYTHON**. Rendu de monnaie**

On considère un système monétaire, c'est-à-dire un ensemble S de type de pièces.

Par exemple, le système (simplifié) de l'euro est $\{100, 50, 20, 10, 5, 2, 1\}$.

L'ensemble S contient toujours la pièce de valeur 1. Étant donné un entier n, on cherche une décomposition en une somme de pièces appartenant à S et dont le nombre de pièces est minimal.

On suppose que S est donnée par la liste des types de pièces, **classées par ordre décroissant**.

Partie I. Algorithmes gloutons

La méthode gloutonne consiste à décomposer n en choisissant toujours la plus grande pièce possible.

1) Écrire une fonction `glouton(n,S)` qui prend en argument un entier n et le système monétaire P renvoie la liste des espèces dont la somme vaut n correspondant à la méthode gloutonne.

Par exemple, si `glouton(99,[100,50,20,10,5,2,1])` renvoie la liste `[50,20,20,5,2,2]`.

On demande un algorithme de complexité $O(r + m)$, où m est le nombre de pièces (la longueur de S) et où r est la longueur de la liste renvoyée à la fin (on a $r \leq n$).

2) Les systèmes monétaires usuels sont conçus pour que la méthode gloutonne soit optimale.
Donner un contre-exemple avec le système monétaire $S = [1, 4, 5]$.

Partie II. Programmation dynamique

Principe : On construit le tableau $P = (p_i)_{0 \leq i \leq n}$ tel que pour tout $i \in \{0, 1, ..., n\}$, l'entier p_i est le nombre minimal de pièces dont la somme fait i.

3) a) On a $p(0) = 0$. Justifier brièvement que $\forall n > 0$, $\boxed{p(n) = 1 + \min_{s \in S \text{ et } s \leq n} p(n - s)}$.

b) En déduire une fonction `rendu(n,S)` qui construit itérativement le tableau P et renvoie p_n.

On demande une fonction de complexité $O(nm)$, où $m = \operatorname{card} S$.

c) Écrire une variante utilisant une fonction auxiliaire récursive et utilisant le tableau P pour mémoriser les résultats calculés au fur et à mesure des appels récursifs.

4) On souhaite obtenir la liste des pièces de monnaie donnant le rendu optimal. On propose deux méthodes :

a) Le tableau P étant connu, **on peut retrouver les pièces utilisées** : par exemple, partant de p_n, on cherche $i < n$ tel que $p_i = p_n - 1$ et $(n - i) \in S$. Puis on itère le procédé sur p_i.

Implémenter cette méthode en définissant une fonction `pieces(P,S)` qui renvoie une liste de pièces associée à un rendu optimal pour un montant de n, où P est de longueur $(n + 1)$.

On demande une fonction de complexité $O(nm)$.

b) Au lieu d'attribuer à p_i le nombre de pièces du rendu optimal, on mémorise dans p_i, pour $1 \leq i \leq n$, le couple (m_i, s_i), où m_i est le nombre de pièces utilisées et s_i la dernière pièce de monnaie utilisée. Par exemple, $p_1 = (1, 1)$.

Implémenter cette méthode en définissant une fonction `renduSomme(n,S)` qui construit le nouveau tableau P et renvoie une liste de pièces associée à un rendu optimal.

On demande une fonction de complexité $O(nm)$.

Partie III. Plus court chemin dans un graphe

5) a) Ramener le problème de rendu de monnaie à un problème de plus court-chemin dans un graphe dont on précisera les sommets et les arêtes.

b) Préciser la complexité (dans le cas le pire) si on applique l'algorithme de Dijkstra (en supposant qu'on a implémenté une file de priorité par un tas de sorte que les opérations (insertions, suppressions et modifications des priorités) ont une en complexité $O(\ln n)$.

On rappelle que chaque sommet est sélectionné au plus une fois, et que de ce fait, chaque arête est lue au plus une seule fois. Le tas contient des sommets distincts, et l'ordre de priorité d'un sommet i correspond à la distance $d(i)$ à l'origine.

c) Proposer sans justification une heuristique $h(i)$ qui permet d'obtenir la solution optimale tout en limitant un tant soit peu le nombre de sommets traités (il s'agit ici de l'algorithme A*).

Partie IV. Une variante de la partie II

On suppose dans cette question que les éléments de S sont classés par ordre croissant.

On a en particulier $S\ [0] = 1$.

On note S_k la liste des $(k + 1)$ pièces de S d'indice $\leq k$. Par exemple, $S_0 = [1]$ et $S_{m-1} = S$.

Pour $n \in \mathbb{N}$ et $0 \leq k < m$, on note $p(n, k)$ le nombre minimal de pièces appartenant à la sous-liste S_k permettant d'obtenir la somme n.

6) a) Donner la valeur des $p(n, 0)$.

Pour $2 \leq k < m$, exprimer sans justification $p(n, k)$ en fonction de $p(n, k - 1)$ et $p(n - S[k], k)$.

b) En déduire une fonction `renduBis(n,S)` qui, en construisant le tableau de deux dimensions, $(p(i, k))_{0 \leq i \leq n, 0 \leq k < m}$, renvoie $p(n, m - 1)$. Préciser la complexité.

Partie V. Extension en dimension 2

On considère un ensemble fini S de couples d'entiers naturels (a, b), avec $(a, b) \neq (0, 0)$.

On suppose que les couples $(1, 0)$ et $(0, 1)$ appartiennent à S.

Étant donné un entier $(a, b) \in \mathbb{N}^2$, on cherche le plus petit entier p pour lequel il existe une famille $((a_i, b_i)_{1 \leq i \leq p})$ de p couples appartenant à S telle que $(a, b) = \sum_{i=1}^{p} (a_i, b_i)$.

Informatique PYTHON**. Rendu de monnaie.Corrigé**

1)
```
def glouton(n,S) :
    r = n ; m = len(S) ; L = [] ; k = 0
    while r>0 :      # il est inutile de vérifier k < m, car le dernier élément de P vaut 1
        if S[k] <= r :  r = r - S[k] ; L.append(P[k])
        else :  k = k+1
    return L
```

A chaque étape, $r + k$ augmente exactement de 1, où r est la longueur de la liste S.

2) On prend $n = 8$ et $S = [1, 4, 5]$.

La méthode gloutonne donne $[5, 1, 1, 1]$ alors que la décomposition optimale est $[4, 4]$.

3) a) Supposons $n > 0$.

Tout jeu de pièces de somme n est constitué d'une pièce $s \leq n$ et d'un jeu de pièces de somme $n - s$.

D'où $p(n) = \min_{s \in S \text{ et } s \leq n}(1 + p(n - s)) = 1 + \min_{s \in S \text{ et } s \leq n} p(n - s)$.

b)
```
def rendu(n,S) :
    P = [0]*(n+1) ; m = len(S)
    for i in range(1,n+1) :
         min = i ; k = m-1     # on a toujours p(i) ≤ i
         while k >= 0 and S[k]< i :
              if P[i-S[k]] < min :  min = P[i-S[k]]
              k = k-1
         P[i] = 1+min)
    return P[n]
```

c)
```
def renduRec(n,S) :
    P = [0]*(n+1) ; m = len(S)
    def aux(i) :
        if i > 0 :
             min = i ; k = m-1
             while k >= 0 and S[k]< i :
                  aux(i-S[k])
                  if P[i-S[k]] < min :  min = P[i-S[k]]
```

```
                    k = k-1
                P[i] = 1+min
    aux(n) ; return P[n]
```

4) a)
```
def pieces(P,S) :
    n = len(P)-1 ; L = [] ; m = len(S) ; i = n     # L liste des pièces
    while i>=0 :
        k = m-1
        while k>=0 and i>=S[k] and P[i-S[k]] > P[i]-1 :  k = k-1
          # en fait, il est inutile de vérifier k ≥ 0 et i ≥ S[k].
        L.append(S[k]) ; i = i-S[k]
    return L
```

b)
```
def renduSomme(n,S) :
    m = len(S) ; P = [ (0,0) ]
    for i in range(1,n+1) :
        min = i ; k = m-1 ; kMin = m-1
        while k >= 0 and S[k]< i :
            if P[i-S[k]] < min :  min = P[i-S[k]] ; kMin = k
            k = k-1
        P.append((1+min,kMin))
    L = [] ; i = n
    while i>0 :
        m,s = P[i] ; L.append(s) ; i = i-s
    return L
```

5) On considère un graphe dont les sommets sont les entiers $0, 1, ..., n$.

Les arêtes sont les $i \to j$ lorsque $(j - i) \in S$, valuées par 1. On cherche un plus court chemin reliant 0 à n.

Comme il y a $(n + 1)$ sommets et chaque sommet admet $O(m)$ successeurs, la complexité par l'algorithme de Dijkstra (où la file de priorité est implémentée par un tas) est en $O(nm + n \log n)$.

6) a)

On a $p(n, 1) = n$ et pour $1 \leq k < m$, $p(n, k) = \begin{cases} \min(p(n, k-1), p(n - S[k], k)) & \text{si } S[k] \leq n \\ p(n, k-1) & \text{si } S[k] > n \end{cases}$

b)

```
def renduBis(n,S) :
    m = len(S) ; P = [[0]*(n+1) for k in range(m)]
```

```
    for i in range(n+1) :  P[i][0] = i
    for i in range(n+1) :
        for k in range(1,m) :
            if S[k]<=i :  P[i][k] = min(P[i-S[k]][k],P[i][k-1])
            else :  P[i][k] = P[i][k-1]
    return P[n][m-1]
```

La complexité est à nouveau en $O(nm)$.