Info2A-PSI - Informatique de Tronc Commun – DS02

Exercice 1.

On se propose d'implémenter l'algorithme de tri par insertion pour trier dans l'ordre croissant une liste d'entiers.

Q1. On considère la fonction `insere` ci-dessous qui prend en argument un entier `a` et une liste `L` d'entiers **triés** par ordre croissant. Cette fonction insère la valeur `a` dans une copie de la liste `L` et modifie la liste obtenue de sorte qu'elle soit triée. Les tableaux seront représentés sous la forme de listes python.
On rappelle la méthode `.copy()`, applicable à une liste L d'éléments de type non mutable, et renvoyant une copie de la liste L.

```
def insere(a, L):
    L = L.copy()
    L.append(a)
    i = ...
    while a < ... and i >= 0:
        L[i+1] = ...
        L[i] = a
        i = ...
    return L
```

- exemples d'appels à la fonction `insere` :

```
>>> insere(3, [1, 2, 4, 5])
[1, 2, 3, 4, 5]
>>> insere(10, [1, 2, 7, 12, 14, 25])
[1, 2, 7, 10, 12, 14, 25]
>>> insere(1, [2, 3, 4])
[1, 2, 3, 4]
```

a. Compléter le code de la fonction `insere`.

b. Évaluer la complexité de la fonction `insere` en fonction du nombre d'éléments, n, de la liste `L`, dans le meilleur et dans le pire cas, s'il en existe.

Q2. On souhaite utiliser la fonction `insere` pour obtenir une copie triée d'une liste d'entiers L.

a. Écrire une fonction `tri_insertion`, prenant en argument une liste `L` à n éléments, et renvoyant, grâce à n appels à la fonction `insere`, une nouvelle liste qui soit une version triée de L.

b. Évaluer la complexité de la fonction `tri_insertion` en fonction du nombre d'éléments, n, de la liste `L`, dans le meilleur et dans le pire cas, s'il en existe.

Q3. Redonner l'implémentation du tri insertion en place d'une liste L vue en classe, sous la forme d'une fonction `tri_insertion2(L)`, et indiquer en quoi sa complexité est meilleure que celle de la fonction `tri_insertion` de la question **Q2**.

Q4. Citer deux autres algorithmes de tri, dont le tri fusion, et leurs complexités respectives en fonction du nombre d'éléments n de la liste à trier.

Q5. On souhaite donner une implémentation récursive du tri fusion sous la forme d'une fonction récursive `tri_fusion(L)`.

a. On se donne une fonction `fusion` qui prend en entrée deux listes `L1,L2` d'entiers triées par ordre croissant et les fusionne en une liste triée `L12` qu'elle renvoie.
Par exemple, on aura :

```
>>> fusion([1,6,10],[0,7,8,9])
[0, 1, 6, 7, 8, 9, 10]
```

Compléter sur votre copie le code de la fonction fusion donné ci-après.

```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0] *(n1 + n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1 < n1 and ... :
        if L1[i1] < L2[i2]:
            L12[i] = ...
            i1 = ...
        else:
            L12[i] = L2[i2]
            i2 = ...
        i += 1
    while i1 < n1:
        L12[i] = ...
        i1 = i1 + 1
        i = ...
    while i2 < n2:
        L12[i] = ...
        i2 = i2 + 1
        i = ...
    return L12
```

b. Définir une fonction récursive `tri_fusion(L)`, faisant appel à la fonction fusion, et renvoyant une copie triée de la liste L en argument.

Exercice 2.

On s'intéresse au problème dit de « rendu de monnaie ».
Un ensemble de pièces et de billets étant choisi, dont les valeurs sont données, en euros, par une liste triée d'entiers, V, par exemple V = [1, 2, 5, 10, 20, 50, 100], on souhaite déterminer, pour une somme S, égale à un nombre entier d'euros, le nombre minimal de pièces ou billets, dont les valeurs sont dans V, que l'on peut utiliser pour réaliser la somme S.
On suppose dans tout l'exercice que la somme S est égale à un nombre entier d'euros et que la valeur minimale dans V est 1, de sorte qu'il est toujours possible de rendre la somme S, au pire en utilisant S pièces de 1 euro.
On suppose que le stock de pièces ou de billets pour chaque valeur $v \in V$ est infini.

Partie A Utilisation d'un algorithme glouton

On applique ici la démarche « gloutonne » qui consiste à choisir les pièces à utiliser, les unes après les autres, en choisissant à chaque fois la pièce ou le billet de plus grande valeur possible, jusqu'à ce que la somme S à rendre soit atteinte.
On suppose pour l'exemple que $V = [1,\ 2,\ 5,\ 10,\ 20,\ 50,\ 100]$
On recherche une implémentation récursive de cet algorithme, sous la forme de la fonction suivante :

```
V = [1, 2, 5, 10, 20, 50, 100] ## la liste V est ici définie en tant que
                               ## variable globale

def rendu_glouton(S, L = [], i = len(V) - 1):
    if S == 0:
       return ...
    p = V[i]
    if p <= ... :
        L.append(...)
        return rendu_glouton(S - p, L, i)
    else :
        return rendu_glouton(S, L, ...)
```

La fonction `rendu_glouton` prend en argument une somme restant à rendre en euros, S, ainsi qu'une liste L de valeurs $v \in V$, donnant les valeurs de toutes les pièces ou billets déjà utilisés, et un entier i indiquant la position dans V de la dernière valeur de pièce ou de billet utilisée.

Par exemple, un appel `rendu_glouton(23, [20, 100, 100], 4)` est adapté s'il reste à rendre une somme de 23 € - car S vaut 23, et que l'on a déjà utilisé deux billets de 100 euros, et un billet de 20 euros pour rendre la monnaie - car L vaut [20, 100, 100], et que la somme restant rendre sera rendue avec des pièces ou billets de valeur inférieur ou égale à 20 € - car i vaut 4 et V[4] vaut 20. Si l'appel récursif `rendu_glouton(23, [20, 100, 100], 4)` advient, on peut en déduire que la somme originelle à rendre était de 243 €.

Une somme à rendre S étant donnée, c'est l'appel `rendu_glouton(S, [], len(V)-1)` qui renverra la liste des valeurs de chaque pièce ou billets à rendre.

- exemples d'appels à la fonction (si V = [1, 2, 5, 10, 20, 50, 100])

```
>>> rendu_glouton(68, [], 6) # ou simplement rendu_glouton(68) compte-tenu des
                             # valeurs par défaut des paramètres L et i
[50, 10, 5, 2, 1]
>>> rendu_glouton(291, [], 6) # ou rendu_glouton(291)
[100, 100, 50, 20, 20, 1]
```

Q1. Pour l'implémentation récursive proposée, préciser quel est ou quels sont le(s) cas de base (ne nécessitant pas d'appel récursif), et de quelle façon le cas général est traité.

Q2. Compléter sur votre copie le code de la fonction `rendu_glouton`.

Q3. Un ensemble de valeurs de pièces ou de billets, V, étant donné, sous la forme d'une liste décroissante d'entiers distincts, et une somme à rendre, S, étant donnés, par quelle instruction peut obtenir le nombre `np` de pièces ou billets à utiliser pour rendre la somme S à l'aide de la fonction `rendu_glouton` ?

Note : l'algorithme glouton ne donne pas la meilleure solution pour S =9 € et V = [1, 3, 5]

Partie B Algorithme récursif exact

L'algorithme glouton précédent ne donne pas toujours une façon optimale de rendre la monnaie (cela dépend du choix de l'ensemble de valeurs V).

Afin de s'assurer que le nombre de pièces ou billets utilisés soit bien minimal, on peut appliquer le raisonnement suivant et implémenter l'algorithme qui en découle.

On se donne, comme précédemment l'ensemble des valeurs des pièces ou billets disponibles, sous la forme d'une liste V de valeurs entières distinctes **triée dans l'ordre croissant** et **telle que v_0 vaut 1**.

On note ici $V = [v_0, v_1, \dots, v_{n-1}]$. **On suppose que $v_0 = 1$.**

Pour tout i compris entre 0 et $n-1$, on note $V_i = [v_0, v_1, \dots, v_i]$, c'est-à-dire, avec les notations du *slicing* : $V_i = V[0:i+1]$ ou encore $V_i = V[:i+1]$.

La liste V étant donnée, on note $f(S,i)$ la fonction renvoyant le nombre minimal de pièces ou billets que l'on peut utiliser pour rendre une somme S, égale à un nombre entier d'euros, en utilisant des pièces ou billets de valeurs $v \in V_i$.

Q4. Donner, pour $S = 54$ € et $V = [1,\ 5,\ 10]$, les valeurs de $f(S,0)$, $f(S,1)$ et $f(S,2)$.

On cherche une implémentation récursive de la fonction f, notée `frec`.

On distingue comme cas de base, le cas où i vaut 0.

Q5. Donner, pour tout entier $S \geq 0$, la valeur de $f(S,0)$.

Dans le cas général où $i > 0$, on peut faire les observations suivantes.

Q6. Une somme S et un entier i compris entre 1 et $n-1$ étant donnés, on cherche à rendre la somme S à l'aide de pièces ou de billets dont la valeur est dans V_i.

a. Donner une condition sur S et V[i] pour qu'il soit possible d'utiliser une pièce ou un billet de valeur V[i] pour rendre la somme S.

b. Si la condition précédente est réalisée, justifier que $f(S,i) = \min\big(1 + f(S - V[i], i), f(S, i-1)\big)$.

c. Si la condition précédente n'est pas réalisée, indiquer, à l'aide de i et V et en justifiant, les valeurs de pièce ou billets est-il possible d'utiliser pour rendre la somme S ?

d. Justifier que si la condition du **Q6.a** n'est pas réalisée, alors $f(S,i) = f(S,i-1)$.

On a ainsi déterminé, pour tout entier S, la valeur de $f(S,0)$ (question **Q5**), et, pour tout i compris entre 1 et $n-1$, que $f(S,i)$ vaut (*cf.* question **Q6**),

- soit $f(S,i-1)$,
- soit $\min\big(1 + f(S - V[i], i), f(S, i-1)\big)$.

Q7. En déduire une implémentation récursive de `frec`, de la fonction f, en supposant que la liste V des valeurs de pièces ou billets est définie en tant que variable globale.

On autorise de faire appel à la fonction `min` et on pourra utiliser le squelette suivant :

```
def frec(S, i):
    ## cas de base
    if ... :
        return ...
    ## cas général
    if ... :
        ...
    else:
        ...
```

La liste V ayant été préalablement définie, un appel `frec(S, len(V) - 1)` devra renvoyer le nombre minimal de pièces ou billets pour rendre une somme de S euros.

Q8. En supposant que V = [1, 5, 10], donner par un schéma, l'arbre des appels récursifs pour un appel initial `frec(17, 2)`.
Indiquer le nombre total d'appels récursifs et la proportion d'appels redondants (appels identiques effectués plusieurs fois).

Partie C Solution de programmation dynamique

Afin d'améliorer la complexité de la solution précédente, on souhaite adapter une démarche de programmation dynamique.

Pour cela, une liste V et une somme à rendre S étant données, on définit un tableau T à deux dimensions pour y stocker les valeurs $f(s,i)$ pour tout s compris entre 0 et S et tout i compris entre 0 et $n-1$ où n est le nombre d'éléments de V.

On suppose encore que S est égale à un nombre entier d'euros et que V est une liste strictement croissante de valeurs entières distinctes et que son plus petit élément vaut 1.

On suppose que des variables `S` et `V` représentant la somme totale à rendre, S, et la liste des valeurs de pièces ou billets ont été préalablement définies.

Q9. Écrire une instruction permettant de construire un tableau T à deux dimensions, ne contenant que des zéros, et ayant la forme d'une liste de listes Python, à n lignes et $S+1$ colonnes.

À terme, la valeur de T[i][s] devra donner la valeur de $f(s,V_i)$, c'est-à-dire le nombre minimal de pièces ou billets utilisables pour rendre une somme s comprise entre 0 et S à l'aide de pièces ou billets dont les valeurs sont dans V_i.

Q10. Pour $i=0$, quelle doit être la valeur de T[i][s] pour tout $s \in [\![0,S]\!]$?
Écrire une séquence d'instructions permettant de donner leur bonne valeur aux éléments de T[0].

Q11. En supposant, pour un i tel que $1 \leq i \leq n-1$, que la ligne $T[i-1]$ a été correctement remplie justifier que l'on doit avoir, pour la ligne $T[i]$ et pour tout s compris entre 0 et S :

$$T[i][s] = \begin{cases} \min\left(1+T[i]\left[s-V[i]\right], T[i-1][s]\right) & \text{si } s \geq V[i] \\ T[i-1][s] & \text{sinon} \end{cases}.$$

Donner une séquence d'instructions permettant de compléter correctement la ligne i du tableau T si l'on suppose la ligne $T[i-1]$ de T correctement remplie.

Q12. Lorsque l'ensemble du tableau T aura été correctement rempli, quelle cellule du tableau donnera le nombre minimal de pièces ou billets à utiliser pour rendre une somme S avec l'ensemble de valeurs possibles V ?

Q13. En reprenant les réponses aux questions **Q9** à **Q13**, proposer une implémentation d'une fonction `fdyn(S, V)`, non récursive, renvoyant le nombre minimal de pièces ou billets à utiliser pour rendre une somme S avec l'ensemble de valeurs possibles V.

On utilisera le squelette suivant

```
def fdyn(S, V):
    n = ...
    T = ...
    # remplissage de la dernière ligne de T
    ...
    # remplissage des lignes 1, 2, ..., n - 1 de T
    ...
    return ...
```