TP 3 – Algorithme glouton

Proposition de corrigé et commentaires

Un algorithme est dit « glouton » lorsqu'il effectue, à chaque étape, le choix qui semble le meilleur à ce moment-là et qui ne revient pas sur sa décision. Ils s'utilisent souvent pour des problèmes d'optimisation, c'est à dire où on cherche à maximiser quelque chose avec des contraintes.

Nous allons illustrer ce type d'algorithme avec quelques exemples.

Question 1
Écrire une fonction pour calculer la somme des éléments d'une liste.
Proposition de script :

```
def somme(liste):
    somme=0 # NÃ©cessaire initialisation
    for i in liste:
        # i prend successivement les valeurs de la liste
        somme+=i # y += x est Ã©quivalent Ã  y = y + x
    return somme
```

Question 2
Écrire une fonction pour déterminer si une liste est triée dans l'ordre décroissant.
Proposition de script :

```
def decroissant(liste):
    for i in range(len(liste)-1):
        if liste[i] < liste[i+1]:
            return False
    return True
```

Question 3
Écrire une fonction pour ranger une liste dans l'ordre décroissant.
*On pourra se souvenir du tri à bulle du **TP2** et utiliser la fonction réalisée à la **question 2**.*
Proposition de script :

```
def rangerdec(liste):
    while decroissant(liste)==False:
        for i in range(len(liste)-1):
            if liste[i]<liste[i+1]:
                liste[i],liste[i+1]=liste[i+1],liste[i]
# cette affection double Ã©vite d'avoir Ã  crÃ©er une variable tampon
# permute deux Ã©lÃ©ments s'ils ne sont pas dans le bon ordre
# chaque itÃ©ration de la boucle "for" amÃ©liore le rangement
# la boucle while amÃ¨ne Ã  le faire autant de fois que nÃ©cessaire (et pas plus!)
    return(liste)
```

Proposition de script qui ne fait pas appel au programme `decroissant`

```
def rangerdec(liste):
    fini=False
    while fini==False:
        fini=True # ... jusqu'Ã  preuve du contraire
        for i in range(len(liste)-1):
            if liste[i]<liste[i+1]:
                liste[i],liste[i+1]=liste[i+1],liste[i]
                fini=False
# s'il y a eu un changement de place Ã  rÃ©aliser, c'est que le rangement n'Ã©tait pas fini
# cet algorithme rÃ©alisera une boucle en trop par rapport au minimum possible
    return(liste)
```

Question 4 : Rendu de monnaie (*premier algorithme glouton*)

On souhaite réaliser une fonction `rendre_monnaie` qui prend en variables :

- un montant à rendre
- la liste des valeurs des pièces ou billets disponibles

et qui renvoie une liste de valeurs des pièces ou billets à rendre.

***Par exemple** :*
`rendre_monnaie(6, [1, 2, 4])` peut renvoyer `[4, 2]` ;
`rendre_monnaie(5, [1, 7, 3])` peut renvoyer `[3, 1, 1]`.

***Instructions qui peuvent s'avérer utiles** :*
L'instruction `liste.append(ajout)` ajoute l'élément `ajout` à la fin de la liste `liste`.
L'instruction `liste=[]` créé une liste vide appelée `liste`.

Est-on assuré que cet algorithme rend le minimum de pièces, i.e. cet algorithme est-il optimal ?

Remarque : Un système de monnaie est dit canonique si cet algorithme glouton est optimal.
La quasi-totalité des monnaies utilisées dans le monde sont canoniques.

Proposition de script :

```
def rendre_monnaie(c, pieces):
    pieces = rangerdec(pieces)
    rendu = []
    for p in pieces:
        while p <= c:
            rendu.append(p)
            c -= p
    return rendu
```

Cet algorithme ne rend pas toujours le minimum de pièces.
Par exemple si on saisit : `rendre_monnaie(6,[4,3,1]`, le programme renverra : `[4,1,1]` alors que `[3,3]` rendait une pièce de moins.

Question 5 : Emploi du temps de salles (*deuxième algorithme glouton*)

1. On cherche à gérer l'emploi du temps d'une salle de cours. Les créneaux des cours sont identifiés par un couple de réels $(d, f) \in [0, 24[^2$ où d est l'heure de début et f l'heure de fin.
   Écrire une fonction `intersecte` pour déterminer si deux cours se chevauchent.

   Par exemple :
   `intersecte`((1, 3), (2, 4)) doit renvoyer `True`
   mais `intersecte`((1, 2), (3, 5)) doit renvoyer `False`.

   Proposition de programme

```
def intersecte(c1, c2):
    if (c1[0]<=c2[0] and c1[1]>=c2[0]) or (c2[0]<=c1[0] and
   c2[1]>=c1[0]):
        return True
    return False
```

Listing 1 – Question 5.1

```
# Autre version, plus subtile
def intersecte(c1, c2):
    return (c1[0] <= c2[0] and c1[1] >= c2[0]) or (c2[0] <=
   c1[0] and c2[1] >= c1[0])
# renvoie 1 (TRUE) si l'heure de dÃ©but d'un cours est
# avant celle de l'autre, mais pas son heure de fin.
```

Listing 2 – Question 5.1

2. Écrire une fonction `intersecte_list` telle que, si c est un cours et L est une liste de cours, `intersecte_list`(c, L) renvoie True si c chevauche un des cours de L.

Par exemple : `intersecte_list`((1, 3), [(2, 5), (4, 6)]) renvoie True
mais `intersecte_list`((1, 3), [(4, 6), (5, 7)]) renvoie False.

Proposition de programme

```
def intersecte_list(c, L):
    for cours in L:
        if intersecte(c, cours):
            return True
    return False
# la crÃ©ation prÃ©alable de la fonction intersecte permet d'
   allÃ©ger la complexitÃ© du programme intersecte_list et d'
   amÃ©liorer la lisibilitÃ© et la maintenabilitÃ© de l'
   ensemble
```

Listing 3 – Question 5.2

3. Écrire une fonction `choisir_salle` telle que, si c est un cours, `choisir_salle`(c, salles) renvoie une salle disponible pour c, ou -1 si aucune salle n'est disponible.

Par exemple :
`choisir_salle`((7, 10), [[(9, 11)], [(12, 15)]]) doit renvoyer 1
mais `choisir_salle`((7, 10), [[(8, 10)], [(9, 12)]]) doit renvoyer -1.

Proposition de programme

```
def choisir_salle(c, salles):
    for s in range(len(salles)):
        if not intersecte_list(c, salles[s]):
            return s
    return -1
```

Listing 4 – Question 5.3

4. Écrire une fonction `allocation_salles(cours)` qui renvoie une liste des salles utilisées par les cours de la liste cours en appliquant l'algorithme glouton. On pourra compléter le code suivant :

```
def allocation_salles(cours):
    cours.sort(key=lambda c: c[0]) # trie les cours par ordre
    de dÃ©but croissant
    salles = []
    for c in cours: # parcours de la liste cours
        for s in range(len(salles)): # teste si on peut
   mettre le cours c en salle s
            if ... # si on peut mettre le cours c en salle s
                salles[s].append(c) # on ajoute le cours c Ã
    la liste des cours de la salle s
                break # on sort de la boucle for
```

```
9
    return salles
```

Listing 5 – Question 5.4

Proposition de programme

```
def allocation_salles(cours):
    cours.sort(key=lambda c: c[0]) # trie les cours par ordre
    de dÃ©but croissant
    salles = []
    for c in cours: # parcours de la liste cours
        s = choisir_salle(c, salles)
        if s == -1:
            salles.append([c])
        else:
            salles[s].append(c)
    return salles
```

Listing 6 – Question 5.4

5. Écrire les trois fonctions suivantes :

- ranger une liste de cours par ordre croissant de leur début
- à partir d'une liste de cours rangés par ordre croissant de leur début, faire une affectation d'une salle sans chevauchement de cours
- à partir d'une liste de cours, affecter autant de salles que nécessaire pour caser tous les cours dans des salles.

Proposition de programme

```
def ranger_croissant(liste):
# cet algorithme scanne la liste et permute deux cours
# adjacents quand ils ne sont pas dans le bon ordre.
# un seul passage ne suffit pas en gÃ©nÃ©ral
# l'opÃ©ration est renouvelÃ©e autant de fois que nÃ©cessaire
    if len(liste)<2:
        return(liste)
    fini=False
    while fini==False:
        for i in range(len(liste)-1):
            fini=True
            if liste[i][0]>liste[i+1][0]:
        # les deux cours ne sont pas dans l'ordre souhaitÃ©
                liste[i],liste[i+1]=liste[i+1],liste[i]
        # on les permute
                fini=False
        # il y a eu une permutation, donc on continue
    return(liste)

def affecter_salle(liste):
# cet algorithme remplit un horaire de salle
# en affectant des cours compatibles
```

```
# Ã  partir d'une liste rangÃ©e par ranger_croissant
    affectation=[liste[0]]
# le premier cours est affectÃ© Ã  la salle ...
    del liste[0]
# ... et supprimÃ© de la liste
    creneau_salle=1
    creneau_liste=1
    taille_liste=len(liste)
    while creneau_liste<taille_liste:
        if affectation[creneau_salle-1][1]>liste[creneau_liste
   -1][0]:
# la fin du cours dans la salle arrive aprÃ¨s le dÃ©but du cours
   de la liste
            creneau_liste = creneau_liste +1
# on passe ce cours pour considÃ©rer le suivant
        else:
            affectation.append(liste[creneau_liste-1])
# on ajoute le cours dans l'affectation de la salle
            del liste[creneau_liste-1]
            taille_liste=taille_liste-1
# on supprime le cours de la liste
            creneau_salle=creneau_salle+1
# on passe au crÃ©neau suivant
    print('renvoie : ',affectation,liste)
    return(affectation,liste)

def affecter_salles(liste):
# on part d'une liste de cours dans un ordre quelconque
    ranger_croissant(liste)
# on range la liste
    salles=[]
# crÃ©Ã© la liste salles (liste de listes de listes Ã  deux
   Ã©lÃ©ments)
    while liste!=[]:
# tant que la liste n'est pas vide
        salle,liste_nouvelle=affecter_salle(liste)
        salles.append(salle)
# appelle la fonction affecter_salle pour remplir une salle
        liste=liste_nouvelle
# met Ã  jour la liste en enlevant les cours affectÃ©s Ã  la
   premiÃ¨re salle
    return(salles)

def presentation(liste):
    affectation=affecter_salles(liste)
    for salle in range(len(affectation)):
        print('La salle nÂ°',salle+1,' est occupÃ©e comme suit :'
   )
        for creneau in range(len(affectation[salle])):
            print('      crÃ©neau ',creneau+1,' : dÃ©but Ã  ', \
                  affectation[salle][creneau][0],' heure et  fin
```

```
    Ã  ', \
                        affectation[salle][creneau][1],' heure.')
70
```

Listing 7 – Question 5.5

$\sim$