Informatique de Tronc Commun

Devoir surveillé n°3

La calculatrice n'est pas autorisée

durée : 2 h

Toutes les réponses sont à donner sur la copie.

Exercice 1. Algorithmes gloutons.

On dispose d'une clé USB qui est déjà bien remplie et sur laquelle il ne reste que 5 Go de libre. On souhaite copier sur cette clé des fichiers vidéo pour l'emporter en voyage. Chaque fichier a un poids, exprimé en mégaoctets ou en gigaoctets, et chaque vidéo a une durée, exprimée sous la forme d'un nombre entier de minutes. La durée n'est pas proportionnelle à la taille car les fichiers sont de formats différents, certaines vidéos sont de grande qualité, d'autres sont très compressées. Le tableau qui suit présente les 7 fichiers disponibles avec les durées données en minutes.
Les fichiers vidéo dont on dispose sont décrits dans un fichier **videos.csv**, qui, lorsqu'on l'ouvre dans un tableur a l'allure suivante :

	A	B	C	D
1	Nom	durée (min)	poids (Go)	unité
2	Vidéo A	114	4,57	Go
3	Vidéo B	32	630	Mo
4	Vidéo C	20	1,65	Go
5	Vidéo D	4	85	Mo
6	Vidéo E	18	2,15	Go
7	Vidéo F	80	2,71	Go
8	Vidéo G	5	320	Mo

Le séparateur « ; » a été utilisé pour enregistrer ce fichier au format .csv.

- On rappelle les éléments suivants utilisables en Python pour lire dans un fichier :
  - `f = open(nom_fichier, mode)` crée un objet-fichier, `f`, permettant d'accéder au fichier dont le nom est `nom_fichier`, soit en lecture, si le mode est `'r'`, soit en écriture, si le mode est `'w'`.
  - `f.close()` ferme l'accès au fichier ouvert avec `open`.
  - `f.readlines()` avec `f` un objet-fichier vers un fichier ouvert en lecture, renvoie une liste de chaînes de caractères dont chacune correspond aux caractères composant une ligne du fichier, caractère de saut de ligne inclus ;
  - `'\n'` : caractère spécial de saut de ligne.

1. Rappeler la suite d'instructions permettant de récupérer le contenu du fichier **videos.csv** sous la forme de la liste `L` suivante :

```
>>> L
['Vidéo A;114;4,57;Go\n', 'Vidéo B;32;630;Mo\n',..., 'Vidéo G;5;320;Go\n']
```

- On rappelle que la méthode `.strip()`, appliquée à une chaîne de caractères terminant par un caractère de saut de ligne, renvoie une copie de cette chaîne de caractères privée de ce dernier caractère de saut de ligne :

```
>>> ch = 'abc\n'
>>> ch
'abc'
```

  - On rappelle que la méthode `.split(sep)`, appliquée à une chaîne de caractères renvoie une liste de toutes les sous-chaînes situées de part et d'autre des occurrences de la chaîne `sep`, servant de séparateur :

```
>>> 'abcdxefgxh'.split('x')
['abcd', 'efg', 'h']
```

2. Écrire une séquence d'instructions permettant de construire à partir de la liste `L` précédente, la liste `L1` suivante :

```
L1 = [['Vidéo A', '114', '4,57', 'Go'],
      ['Vidéo B', '32', '630', 'Mo'],
      ...,
      ['Vidéo G', '5', '320', 'Go']]
```

- On rappelle que la méthode `.replace(old, new)`, appliquée à une chaîne de caractères, renvoie une chaîne de caractères dans laquelle toutes les occurrences de la sous-chaîne `old` ont été remplacées par la sous-chaîne `new` :

```
>>> 'abracadabra'.replace('a', 'x')
'xbrxcxdxbrx'
```

- On rappelle les fonctions de conversion de type, `int` et `float`, permettant, respectivement, de convertir une chaîne de caractères correspondant à l'écriture d'un entier ou d'un flottant en une valeur de type `int` ou `float`, respectivement :

```
>>> int('25')
25
>>> float('2.5')
2.5
```

3. Écrire une séquence d'instructions permettant de modifier la liste `L1` précédente de sorte qu'elle soit transformée en la liste suivante :

```
>>> L1
[['Vidéo A', 114, 4.57, 'Go'],
 ['Vidéo B', 32, 630.0, 'Mo'],
 ...,
 ['Vidéo G', 5, 320.0, 'Go']]
```

4. Écrire une séquence d'instructions permettant de construire, à partir de la liste `L1` obtenue à la question précédente, la liste `L2` des quadruplets (nom, durée en minutes, poids en Go, durée (min)/poids des vidéos (Go)) pour toutes les vidéos.
On aura ainsi :

$$L2 = \left[\left('\text{Vidéo A}', 114, 4.57, \frac{114}{4.57}\right), \left('\text{Vidéo B}', 32, 630 \times 10^{-3}, \frac{32}{630 \times 10^{-3}}\right), \ldots, \left('\text{Vidéo G}', 5, 320.0, \frac{5}{320}\right)\right]$$

5. Écrire une fonction `dureepoids_total (L)` prenant en entrée une liste `L` de la même forme que la liste `L2` et renvoyant la durée totale en minutes et le poids total en Go des vidéos de la liste `L`.

6. Modifier le code de la fonction précédente, pour écrire une fonction `choix(L, pmax)`, prenant en argument une liste `L` de la même forme que la liste `L2`, mais **supposée triée dans l'ordre décroissant des rapports durée (min)/poids des vidéos (Go)**, et renvoyant une liste de noms de vidéos dont le poids total n'excède pas le poids `pmax` (en Go), ainsi que la durée totale et le poids total des vidéos sélectionnées.

Exercice 2.

Q2.1. Écrire une fonction `rechMax` prenant en argument une liste non vide de valeurs entières ou flottantes et renvoyant la valeur du maximum des éléments de cette liste.

Q2.2. Évaluer la complexité temporelle de la fonction `rechMax` en fonction de la longueur de la liste en entrée dans le pire cas. Caractériser le pire cas.
Déterminer le meilleur cas, le caractériser et donner l'ordre de grandeur de la complexité temporelle dans ce cas.
Conclure de façon générale.

Exercice 3.

La fonction définie ci-dessous prend pour argument une liste de flottants.
On rappelle que l'instruction `dd=float('inf')` affecte à la variable `dd` une valeur de type flottant que l'on peut assimiler à « $+\infty$ » qui a pour propriété d'être supérieure strictement à toute autre valeur de type flottant.

La valeur `None` (de type `NoneType`) est utilisée ici comme valeur par défaut pour les variables `xx` et `yy`.
On rappelle que la fonction valeur absolue a pour nom `abs` (*elle est accessible sans importation de module*).
On considère la fonction `f` implémentée ci-dessous, de deux façons équivalentes :
On rappelle que l'instruction `dd=float('inf')` affecte à la variable `dd` une valeur de type flottant que l'on peut assimiler à « $+\infty$ » qui a pour propriété d'être supérieure strictement à toute autre valeur de type flottant.

```
def f(L):
    xx, yy = None, None
    dd = float('inf')
    for x in L:
        for y in L:
            if x != y:
                d = abs(x-y)
                if d < dd :
                    xx, yy, dd = x, y, d
    return xx, yy
```

```
def f(L):
    xx, yy = None, None
    dd = float('inf')
    for i in range(len(L)):
        for j in range(len(L)):
            if L[i] != L[j]:
                d = abs(L[i] - L[j])
                if d < dd:
                    xx, yy, dd = L[i], L[j], d
    return xx,yy
```

Q3.1. Que renvoie la fonction `f` si on l'applique :

- **a.** à liste `L1 = [10, 20, 12, 1, 5]`
- **b.** à liste `L2 = [10, 10, 12]`

Q3.2. Que fait l'algorithme implémenté par `f` ? (à quel problème répond-il ?)

Q3.3. Déterminer la complexité de l'algorithme implémenté par la fonction `f`, en fonction de la longueur n de la liste `L`.

Q3.4. Expliquer pourquoi l'algorithme suivant appliqué à une liste **triée**, résout le même problème que le précédent :

```
def f2(L):
    dd = float('inf')
    for i in range(len(L) - 1):
        x, y = L[i], L[I + 1]
        if x !=y :
            d = abs(x - y)
            if d < dd:
                xx, yy, dd = x, y, d
    return xx, yy
```

Q3.5. Évaluer la complexité de l'algorithme implémenté par la fonction `f2`.

Exercice 4. **Somme et moyenne**

Q3.1. Écrire une fonction `somme` qui prend en argument une liste de nombres et renvoie la somme des éléments de cette liste. Il s'agit d'implémenter une fonction analogue à la fonction prédéfinie `sum()`.

Exemple de résultat d'un appel à la fonction :

```
>>> somme([3,4,12])
19
```

Q4.2. Utiliser la fonction `somme` précédente pour calculer la somme des entiers de 1 à 100 (inclus).

Q4.3. En s'inspirant de la fonction `somme`, créer une fonction `moyenne` calculant la moyenne des éléments d'une liste.

Q4.4. Évaluer la complexité de la fonction moyenne.

On rappelle la définition de la variance v d'une liste de nombres $(n_1, n_2, \dots, n_N)$:

$$v = \frac{1}{N}\sum_{i=1}^{N}(n_i - \mu)^2$$

où N est le nombre d'éléments de la liste, μ est la moyenne de la liste.

Q4.1. En s'inspirant de la fonction `somme` précédente, écrire une fonction `var` calculant la variance d'une liste de nombres.

Q4.2. Écrire une fonction `var2` utilisant la formule suivante (formule d'Huyghens) :

$$v = \frac{1}{N}\sum_{i=1}^{N} n_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} n_i\right)^2$$

Q4.3. Évaluer la complexité des fonctions var1 et var2.

Exercice 5. Recherche d'une valeur dans une liste

Recherche séquentielle dans une liste quelconque

1. Écrire une fonction `recherche1` qui prend en argument une liste `L` et une valeur `v` et qui renvoie `True` si la valeur est présente dans la liste `L` et `False` si ce n'est pas le cas.

Exemples d'appels à la fonction :

```
>>> recherche1([1, 10, 5, -6, 0], 5)
True
>>> recherche1(['toto', 'titi', 'tata'], 'tonton')
False
```

2. Évaluer la complexité d'un appel à la fonction `recherche1` en fonction de la longueur n de la liste en entrée, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

3. Écrire une fonction `recherche2` qui prend en argument une liste `L` et une valeur `v` et qui renvoie la liste des indices des occurrences de la valeur `v` dans la liste `L` .
La liste renvoyée sera une liste vide si la valeur n'est pas présente dans la liste.

Exemples d'appels à la fonction :

```
>>> recherche2([1, 10, 5, -6, 0], 4)
[]
>>> recherche2(['A', 'B', 'C', 'E', 'D', 'C', 'D', 'C'], 'C')
[2, 5, 7]
```

Dans le second exemple, en effet, le caractère `'C'` est présent dans la liste, aux positions 2, 5 et 7.

4. Évaluer la complexité d'un appel à la fonction `recherche2` en fonction de la longueur n de la liste en entrée, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

Recherche par dichotomie

La fonction `recherche3` définie ci-dessous permet une recherche par dichotomie d'un élément dans une liste triée. Cette fonction prend deux arguments, le premier est une liste dont les éléments sont classés dans l'ordre croissant et le second la valeur à rechercher dans la liste.

```
1.  def recherche3(L,v):
2.      bas, haut = 0, len(L) - 1
3.      while bas <= haut:
4.          milieu = (bas + haut) // 2    #détermination de la valeur de milieu
5.          print(bas, milieu, haut)      #affichage des valeurs bas, milieu, haut
6.          if L[milieu] == v:      #
7.              return True         #valeur trouvée
8.          elif L[milieu] < v:     #valeur v supérieure strictement à L[milieu]
9.              bas = milieu + 1    #
10.         else:                   #valeur v inférieure à L[milieu]
11.             haut = milieu - 1   #
12.     return False                #valeur non trouvée dans la liste
```

On définit une liste `L` comme suit :

```
>>> L = [10, 20, 30, 50, 60, 80, 90, 100, 120]
```

L'exécution de l'instruction `print(recherche3(L, 30))` (recherche de la valeur 30 dans la liste `L`), induit les affichages suivants :

```
>>> print(recherche3(L, 30))
0 4 8
```

```
0 1 3
2 2 3
True
```

On a donc 3 itérations de la boucle `while`, au cours desquelles les valeurs affichées des variables `bas`, `milieu` et `haut` sont les suivantes :

	`bas`	`milieu`	`haut`
Itération 1	0	4	8
Itération 2	0	1	3
Itération 3	2	2	3

À l'issue de la 3ème itération, la valeur 30 est trouvée, donc la fonction renvoie `True`.

5. À l'exemple de ce qui précède, déterminer quels sont les affichages produits et la valeur renvoyée lors de l'exécution de l'appel `recherche3(L, 20)`.

6. Faire de même pour l'appel `recherche3(L, 0)`.

On peut s'assurer de la correction d'un algorithme utilisant une boucle, en produisant pour cette boucle un invariant de boucle.

7. Rappeler ce qu'est un invariant de boucle et en donner un pour la boucle de la fonction `recherche3`.

La fonction `recherche3` proposé comportant une boucle `while`, il est primordial de s'assurer de la fin de celle-ci, pour démontrer que l'algorithme termine.

8. Considérant l'implémentation de la fonction `recherche3`, indiquer à quelle(s) condition(s) une exécution de la fonction termine.

9. En considérant les valeurs initiales, b_0 et h_0, des variables `bas` et `haut`, et en notant respectivement, b_i et h_i, leurs valeurs, à la fin de la $i^{\text{ème}}$ itération de la boucle `while`, quelle propriété de la suite $(h_i - b_i)_{i\geq 0}$ assure que la boucle termine ? Pourquoi ? (il n'est pas demandé de démontrer cette propriété).

10. Déduire de la question **8.** un variant de boucle pour l'algorithme et lui donner un sens concret.

Complexité de la recherche dichotomique dans une liste triée.

11. Pour une liste de 1000 éléments, peut-on estimer le nombre d'itérations nécessaires pour que la boucle `while` termine ? Donner et justifier par des calculs votre estimation.

12. Pour une liste L de longueur n, donner, sans la démontrer, une majoration du nombre d'itérations nécessaires à la fonction `recherche3` pour renvoyer le résultat.

13. Donner alors la complexité d'exécution de cette fonction `recherche3`.

14. En comparant la complexité de la fonction `recherche3` à celle de la fonction `recherche1` définie à la question 1, expliquer en une ou deux phrases, l'intérêt de la recherche dichotomique.
On pourra en particulier comparer la façon dont on peut s'attendre à ce que varie le temps d'exécution lorsque la longueur de la liste en entrée double en considérant le nombre d'itérations nécessaires dans le pire cas.

Exercice 6. Algorithme de tri sélection

PARTIE A

On donne ici une implémentation du tri sélection légèrement différente de celle du cours

```
1. def tri_selection1(L):
2.     n = len(L)
3.     for i in range(n - 1):
4.         p = 0
5.         for j in range(1, n - i):
6.             if L[j] > L[p]:
7.                 p = j
8.         L[n - 1 - i], L[p] = L[p], L[n - 1 - i]
9.         #print(L)
10.     return L
```

1. On décommente la ligne 9 et on exécute l'appel `tri_selection1([3, 5, 2, 8, 3, 4, 6])`. Donner tous affichages produits par l'instruction en ligne 9.
2. Quel est le rôle des instructions lignes 4 à 7 ?
3. Donner un invariant de boucle pour la boucle d'indice i.
4. Expliquer en une ou deux phrases pour quelle raison, si l'on se limite à appeler la fonction `tri_selection1` sur des listes auxquelles on a préalablement attaché un nom de variable, l'instruction `return L` en ligne 10 peut être omise.

Exercice 7. Algorithme de tri insertion

On donne la fonction suivante qui est version altérée de l'implémentation du tri insertion donnée en classe :

```
1.  def tri_insertionF(L):
2.      n = len(L)
3.      for i in range(1, n):
4.          tmp = L[i]
5.          j = i
6.          while L[j - 1] > L[i]:
7.              L[j] = L[j - 1]
8.              j = j - 1
9.          L[j] = tmp
```

On indique qu'en Python, une position négative, j, dans une liste L, et comprise entre $-\text{len}(L)$ et -1 désigne la position $i = \text{len}(L) - 1 - j$. Ainsi, par exemple, si $L = [5,\ 2,\ 3]$, $L[-1]$ vaudra 3 et $L[-3]$ vaudra 5.

1. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertionF` sur la liste $L = [1,\ 4,\ 3,\ 2,\ 1]$.
2. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertionF` sur la liste $L = [4,\ 1,\ 5]$.
3. Apporter des modifications minimales entre les lignes 4 à 8 de la fonction `tri_insertionF` pour obtenir une fonction `tri_insertionT` dans laquelle l'invariant de boucle du tri insertion, à savoir le fait qu'avant toute itération et à la fin de chaque itération d'indice i, la portion de liste $L[:i+1]$ est triée.

On propose l'implémentation suivante du tri insertion

```
1.  def tri_insertion2(L):
2.      n = len(L)
3.      for i in range(n - 2, -1, -1):
4.          j = i
5.          while j < n - 1 and L[j] > L[j + 1]:
6.              L[j + 1], L[j] = L[j], L[j + 1]
7.              j = j + 1
```

On indique que pour a entier positif, `range(a, -1, -1)` génère la liste de valeurs $[a, a-1, \ldots, 1, 0]$.

4. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertion2` sur la liste $L = [7,\ 4,\ 1,\ 6,\ 5]$.
5. Donner l'invariant de boucle vérifié par la liste L au fil des itérations de la boucle `for` d'indice i.
6. Quel est le rôle des lignes 4 à 7 ?

Exercice 8. Diviser pour régner

On considère la fonction `g` suivante :

```
def g(L1, L2):
    n1, n2 = len(L1), len(L2)
    i1, i2 = 0, 0
    L1 = L1 + [float('inf')]
    L2 = L2 + [float('inf')]
    L = []
    while i1 + i2 <= n1 + n2:
        if L1[i1] < L2[i2]:
            L = L + [L1[i1]]
            i1 = i1 + 1
        else:
            L = L + [L2[i2]]
            i2 = i2 + 1
    L.pop()
    return L
```

On indique que la valeur `float('inf')` est une valeur flottante spéciale ayant la propriété d'être supérieure ou égale à toute autre valeur flottante ou entière.

Remarque : cette valeur est affichée sous la forme de la chaîne de caractère `inf`

```
>>> float('inf')
inf
```

On rappelle que l'instruction `L.pop()` supprime et renvoie la valeur en dernière position dans une liste L non vide.

1. Donner l'état de la liste L au fil des itérations lors d'un appel à la fonction g avec pour argument les listes $L_1 = [4, 1, 2]$ et $L_2 = [3, 4, 5]$.
2. Reprendre la question **1.**, lorsque $L_1 = [1, 2, 4]$ et $L_2 = [3, 4, 5]$.
3. Quelles que soient les listes L_1 et L_2 en argument de la fonction, quel est, en fonction des longueurs respectives n_1 et n_2 de ces deux listes, le nombre d'itérations effectuées par la boucle `while`. Justifier.
4. Quelles que soient les listes L_1 et L_2 en argument de la fonction, quelle sera le nombre d'éléments de la liste L renvoyée par la fonction `g` ? Justifier.
5. Si les listes L_1 et L_2 sont triées, décrire, en fonction des valeurs i_1 et i_2 des variables `i1` et `i2`, décrire le contenu de la liste L à la fin de chaque itération.
   En déduire l'état de la liste L lorsque la fonction termine.
6. Donner un invariant de boucle pour la boucle `while`, permettant d'établir la correction de l'algorithme de fusion de deux listes triées.
7. Justifier que la boucle `while` termine, en exhibant un variant de boucle.

On souhaite utiliser la fonction `g` précédente pour donner une implémentation récursive sous la forme d'une fonction `tri_fusion` du tri par fusion, qui, on le rappelle, consiste à appeler récursivement la fonction sur les moitiés gauche et droite d'une liste non triée, et à construire une version triée de la liste en entrée à partir des listes triées renvoyées par ces deux appels récursifs.

8. Compléter sur votre copie le squelette de fonction suivant, de sorte à obtenir une telle implémentation du tri par fusion

```
1.  def tri_fusion(L):
2.      n = len(L)
3.      ## traitement des cas de base
4.      ...
5.
6.      ## traitement du cas général par deux appels récursifs
7.      Lgauche = ... # Lgauche est une copie de la partie gauche de la liste,
8.                    # obtenue par slicing
9.      Ldroite = ... # Lgauche est une copie de la partie droite de la liste
10.                    # obtenue par slicing
11.     ...
12.     return ...
```

9. Quelle est la complexité, en fonction de la longueur n de la liste L des opérations en lignes 7 et 9 ?

10. Rappeler sans justification la complexité du tri fusion et la comparer à celle du tri insertion.

L'algorithme de tri par fusion utilise le principe « diviser pour régner ».

11. Proposer une fonction `maximum`, dont le squelette est le même que celui proposé en question **6.** et implémentant récursivement la recherche du maximum d'une liste L en appliquant le principe « diviser pour régner », c'est-à-dire en ramenant la recherche du maximum d'une liste L à la recherche du maximum de la première moitié de la liste et du maximum de la seconde moitié de la liste.

12. Que pensez-vous de l'intérêt de la fonction `maximum` proposée en termes de complexité ?