INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Interrogation de janvier – ***Solutions***

Prénom : Nom : Matricule :

Remarques préliminaires

- On vous demande de répondre à **chaque question sur une feuille séparée**.
- N'oubliez pas d'inscrire votre nom, prénom et numéro de matricule sur chaque feuille.
- Vous disposez de trois heures et vous ne pouvez pas utiliser de notes.
- Vous pouvez utiliser le verso des feuilles pour répondre.
- Si du code vous est demandé,
  - la réponse à la question doit comprendre le code *Python* structuré et conforme aux règles de bonne pratique et conventions,
  - sauf mention contraire, vous ne pouvez utiliser aucune fonction de librairies (pas d'`import`)
  - veillez à découper votre réponse en fonctions de manière pertinente
  - veillez à utiliser des structures de données appropriées.

Question 1 - Fonctionnement de Python (2 points)

Expliquer le mode de fonctionnement d'un compilateur et d'un interpréteur. Python de base est-il interprété ou compilé ? Expliquez quels en sont les avantages et les inconvénients pour ce langage.

Solution. → Voir syllabus.

Question 2 - Code Python (5 points)

Pour chacun des codes suivants,
- donnez ce qu'il imprime et expliquez de façon générale ce que fait la fonction (ce qu'elle résout comme problème, ...) ;
- donnez l'état du programme lors de chaque `print` grâce à des diagrammes d'état (comme fait au cours) ;
- donnez la complexité moyenne et maximale ($\mathcal{O}()$) de la **fonction** `foo_`i ($i = 1..3$) en temps d'exécution. Précisez bien les paramètres utilisés pour exprimer la complexité et justifiez bien vos réponses (le résultat seul ne suffit pas pour obtenir des points).

NB :
- pour expliquer ce que le code fait et donner la complexité, vous pouvez annoter les codes fournis (compléter les docstrings, commenter, ...)
- extrait de l'aide mémoire : `s.count(sub [,start [,end]])` : donne le nombre d'occurrences sans chevauchement de `sub` dans `s[start:end]`
- extrait de l'aide mémoire : `d.setdefault(k [,v])` avec le dictionnaire `d`) : la valeur `d[k]` si elle existe sinon `v` et rajoute `d[k]=v`
- `s.add(x)` rajoute (la valeur de) `x` à l'ensemble `s` ; `s.pop()` enlève une valeur de l'ensemble `s` et la renvoie

1.
```python
def foo_1(liste):
    """

    """
    dico = {}
    for elem in liste:
        dico[liste.count(elem)] = dico.setdefault(liste.count(elem), set({})) | {elem}
    print(dico)
    return dico

u = ['a','b','c','a','c','d','a','d']
print(foo_1(u))
```

2.
```
def foo_2(data):
    """


    """
    if len(data) == 0:
        res = []
        print("donnez le diagramme d'état du programme quand le texte s'imprime")
    else:
        res = [min(data)] + foo_2([x for x in data if x > min(data)])
    return res

print(foo_2([5,3,1,9]))
```

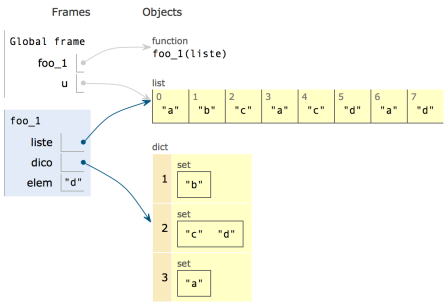
3.
```
def foo_3(g, a, b):
    """


    """
    s = {a}
    to_do = {a}
    while len(to_do) > 0 and b not in to_do:
        z = to_do.pop()
        for y in g[z]:
          if y not in s:
             s.add(y)
             to_do.add(y)
        print(to_do)
    return b in to_do


carte={'c1':{'c2','c3'},
       'c2':{'c1','c4'},
       'c3':{'c1','c4'},
       'c4':{'c2','c3','c5'},
       'c5':{'c4','c6'},
       'c6':{'c5'}} # représente une carte routière avec les carrefours c1..c6
print(foo_3(carte, 'c1', 'c6'))
```

Solution.

1. – Imprime
   `{1: {'b'}, 2: {'c', 'd'}, 3: {'a'}}`
   `{1: {'b'}, 2: {'c', 'd'}, 3: {'a'}}`
   – `foo_1` construit et renvoie un dictionnaire qui contient comme clé des nombres d'occurrences, et comme valeur, l'ensemble (set) des caractères rencontrés dans liste ayant ce nombre d'occurrences.
   – Les deux diagrammes d'états sont semblables exceptés qu'au second print, le namespace de la fonction `foo_1` a été supprimé.



   – Complexités moyenne et maximale
   Hypothèse `n = len(liste)`

```
1 def foo_1(liste):                                          # O(n^2)        O(n^2)
2   """
3   construit et renvoie un dictionnaire qui contient comme
```

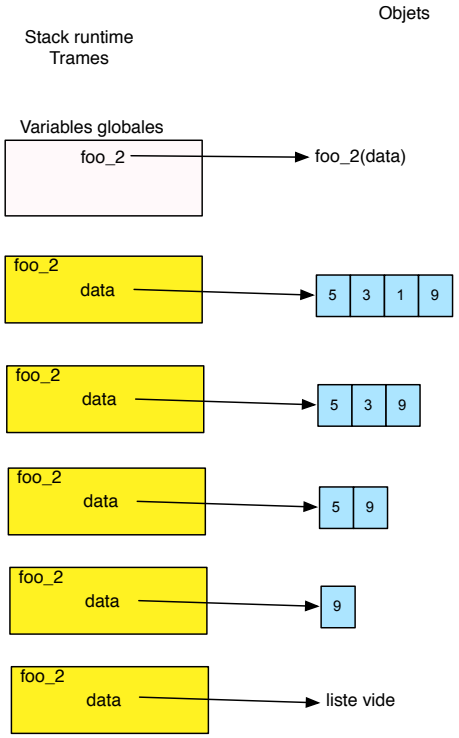
```
4    clé des nombres d'occurrences, et comme valeur, la liste des caractères
5    rencontrés dans liste ce nombre d'occurences
6    """
7    dico = {}                                                    # O(1)          O(1)
8    for elem in liste:                                  # O(n) itérations: O(n^2)      O(n^2)
9       dico[liste.count(elem)] = \
10           dico.setdefault(liste.count(elem), set({})) | {elem}        # O(n)     O(n)
11    print(dico)                                                 # O(n)     O(n)
12    return dico                                                 # O(1)     O(1)
```

Explication de la complexité : compter combien de fois la valeur `elem` se trouve dans une liste a une complexité moyenne ou maximale en $O(n)$; accéder à un élément d'un dictionnaire de taille maximum `n` est en moyenne en $O(1)$ et la complexité maximale est en $O(n)$; l'instruction en lignes 9-10 est donc en $O(n)$ (moyenne et max). Le `for` s'effectue `n` fois : on obtient donc $O(n^2)$.

2. – Imprime

```
donnez le diagramme d'état du programme quand le texte s'imprime
[1, 3, 5, 9]
```

– Le second diagramme d'états ne contient qu'une référence vers la fonction `foo_2`. Le premier diagramme est donné ici :



– Explication et complexités :

```
def foo_2(data):                          # O(n+n-1+n-2...+1) = O(n^2) (idem max)
   """
   Trie data (par sélection mais de façon récursive) si toutes les données
   sont différentes, sinon, trie mais ne conserve qu'un exemplaire
   de chaque valeur différente

   """
   if len(data) == 0:                                  # O(1)    O(1)
      res = []                                         # O(1)    O(1)
      print("...")                                     # O(1)    O(1)
   else:
      res = [min(data)] + \
        foo_2([x for x in data if x > min(data)]) # O(n) plus l'exécution récursive (max:idem)
   return res                                          # O(1)
```

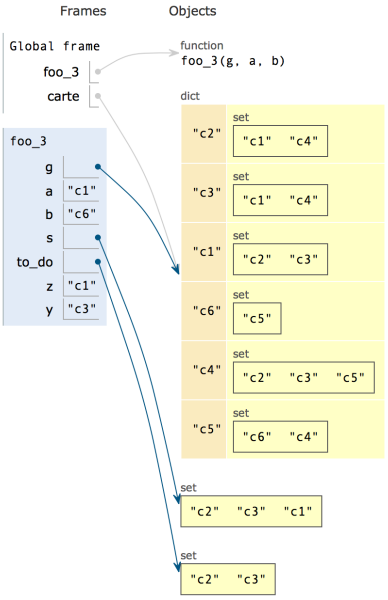
Chaque instance a une complexité linéaire dans la taille de `data`. Comme il y a des appels récursifs pour `data` d'une taille à chaque appel diminuée de un (en supposant ques les éléments sont tous différents), on obtient le résultat.

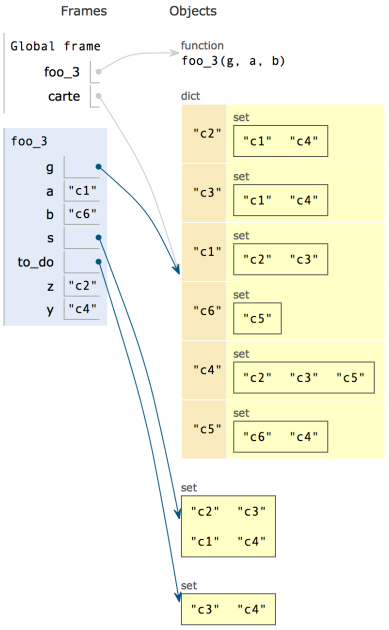
3. – Imprime

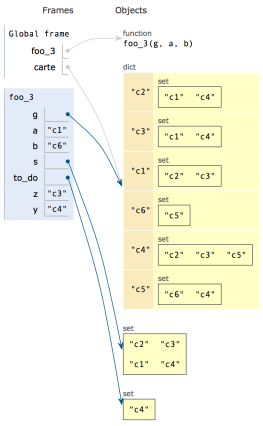
```
{'c2', 'c3'}
{'c3', 'c4'}
{'c4'}
{'c5'}
{'c6'}
True
```

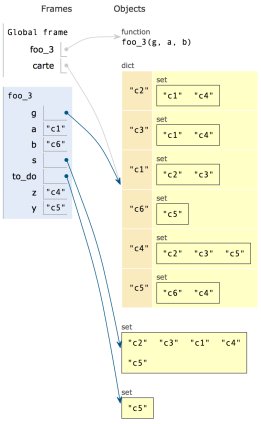
– que fait la fonction : voir docstring

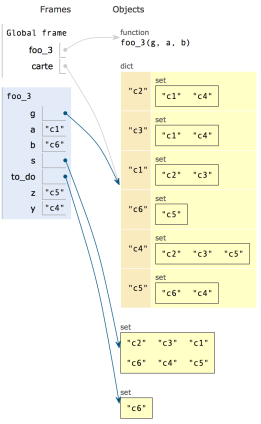
– Diagrammes d'état :











– Complexité : hypothèses chaque élément à `m` voisins dans sa liste et `n` éléments sont accessibles à partir de `a`

```
def foo_3(g, a, b):                  # O(m.n) O(m.n^2)
    """
    calcule l'ensemble des éléments accessibles à partir
    de a jusqu'à ce que tous les éléments accessibles soient
    retenus ou que l'on accède à b.  Renvoie True si b
    est accessible à partir de a.
    """
    s = {a}                          # O(1)      O(1)
    to_do = {a}                      # O(1)      O(1)
    while len(to_do) > 0 and b not in to_do: # n itérations : O(m.n) max: O(m.n^2)
        z = to_do.pop()              # O(1)       O(1)
        for y in g[z]:               # m itérations: O(m)   O(m.n)
            if y not in s:           # O(1)          O(n)
                s.add(y)             # O(1)          O(n)
                to_do.add(y)         # O(1)          O(n)
        print(to_do)                 # O(n)          O(n)
    return b in to_do                # O(n)          O(n)
```

Question 3 - Manipulation de fichiers et de données (5 points)

Ecrivez une fonction `mean_calc(names_file, grade_files, means_file)` qui permet de calculer la moyenne de points individuelle pour un ensemble d'étudiants. La fonction reçoit trois paramètres :

`names_file` contient le nom du fichier dans lequel se trouve une liste de noms d'étudiants (un nom par ligne ; cf. exemple ci-dessous).

`grade_files` est une liste de (noms de) fichiers, chaque fichier contenant une cote par ligne. Il y a correspondance dans les fichiers entre la ligne dans laquelle se trouve le nom de l'étudiant dans le fichier `names_file` et les cotes dans les fichiers nommés par `grade_files`.

`means_file` contient le nom du fichier résultat qui contient, par ligne, le nom d'un étudiant et sa moyenne.

Notez que tous les fichiers contiennent le même nombre de lignes.

Exemple.

Fichier de noms

`noms.txt`

```
Stefan
Anh Vu
Alain
Thierry
Jean
```

Fichiers de cotes

`interro1.txt`

```
5
7
2
9
6
```

`interro2.txt`

```
7
9
6
9
7
```

`interro3.txt`

```
1
5
7
3
8
```

Etant donné les fichiers ci-dessus, l'appel

```
mean_calc("noms.txt", ["interro1.txt", "interro2.txt", "interro3.txt"], "moyennes.txt")
```

produit le fichier de résultats suivant :

Fichier résultat

`moyennes.txt`

```
Stefan    4.333333333333333
Anh Vu    7.0
Alain     5.0
Thierry   7.0
Jean      7.0
```

Note. Dans le fichier résultat `moyennes.txt`, utilisez le signe de tabulation ("`\t`") comme séparateur entre le nom de l'étudiant et sa moyenne.

Solution.

```
def mean_list(ls):
    res = []
    for i in range(len(ls[0])):
        s=0
        for ssl in ls:
            s += ssl[i]
        res.append(s / len(ls))
    return res

def grades_from_file(file_name):
    f = open(file_name)
    grades = []
    for line in f.readlines():
        grades.append(int(line))
    return grades

def mean_calc(names_file, grade_files, means_file):
    grades_lists = []
    for f_name in grade_files:
        grades_lists.append(grades_from_file(f_name))
    means = mean_list(grades_lists)
    fo = open(means_file, "w")
    names = open(names_file).readlines()
    for i in range(len(names)):
        fo.write(names[i].strip() + "\t" + str(means[i]) + "\n")
    fo.close()

mean_calc("noms.txt", ["interro1.txt", "interro2.txt", "interro3.txt"], "moyennes.txt")

#Affichage du fichier resultat
for line in open("moyennes.txt").readlines():
    print(line, end="")
```

Question 4 - Tri (4 points)

Ecrivez une fonction `sort_grades(data)` qui reçoit comme paramètre `data` une liste de tuples $(nom, moyenne)$ (indiquant, pour chaque étudiant identifié par son nom, la $moyenne$ de points obtenus) et qui la trie principalement en ordre décroissant selon la $moyenne$ et puis, en cas d'*ex aequo* (c'est-à-dire si deux ou plus étudiants ont la même moyenne), selon le nom (dans l'ordre lexicographique). Vous ne pouvez pas utiliser les fonctions intégrées de tri offertes par Python, mais vous pouvez choisir la méthode de tri que vous voulez utiliser (parmi celles vues lors des séances de travaux pratiques).

Exemple. Pour le jeu de données suivant :

```
>>> data = [("Stefan", 4.3), ("Anh Vu", 7.0), ("Alain", 5.0), ("Thierry",7.0), ("Jean", 7.0)]
>>> sort_grades(data)
>>> print(data)
[("Anh Vu", 7.0), ("Jean", 7.0), ("Thierry",7.0), ("Alain", 5.0), ("Stefan",4.3)]
```

Solution.

C'est la relation d'ordre (`is_before()`) qui importe pour cette question. Elle est utilisée dans l'une ou l'autre forme de tri (par insertion ou par sélection).

```
def is_before(student1,student2):
        (name1,grade1) = student1
        (name2,grade2) = student2
        return (grade1 > grade2) or (grade1 == grade2 and name1 <= name2)


def sort_grades_selection(data):
        for i in range(len(data)-1):
                jmin = i
                for j in range(jmin+1,len(data)):
                        if is_before(data[j],data[jmin]):
                                jmin = j
                data[i],data[jmin] = data[jmin],data[i]


def sort_grades_insertion(data):
    for i in range(1, len(data)):
        val = data[i]
        j = i
        while j > 0 and is_before(val, data[j - 1]):
            data[j] = data[j - 1]
            j -= 1
        data[j] = val


data = [("Stefan", 4.3), ("Anh Vu", 7.0), ("Alain", 5.0), ("Thierry",7.0), ("Jean", 7.0)]

sort_grades_selection(data)
#sort_grades_insertion(data)

print(data)
```

Question 5 - Récursivité (4 points)

Ecrivez une fonction `equidim(ls1, ls2)` qui, étant donné deux listes (de listes) `ls1` et `ls2`, renvoie un booléen qui vaut `True` si les deux listes ont la même structure (càd le même nombre d'éléments et de sous-listes, mais pas forcément les mêmes valeurs), et `False` sinon.

Exemple

```
>>> ls1 = [1, 2, [3, [4, 5]]]
>>> ls2 = [9, 4, [3, [7, 1]]]
>>> ls3 = [1, [2, 3], [4, 5]]
>>> equidim(ls1,ls2)
True
>>> equidim(ls1,ls3)
False
```

On fait l'hypothèse simplificatrice que chaque élément d'une liste ne peut être qu'un entier ou une autre liste.

Note. Pour rappel, il est possible de vérifier qu'une variable `x` est une liste en utilisant l'expression booléenne `type(x) == list`.

Solution.

```
def equidim(ls1,ls2):
  if type(ls1) == list:
    i = 0
    res = (type(ls2) == list) and (len(ls1) == len(ls2))
    while i < len(ls1) and res:
      res = equidim(ls1[i],ls2[i])
      i += 1
  else:
    res = (type(ls1) == type(ls2))
  return res
```