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

En python, la méthode `deepcopy` est utile dans certaines conditions. Expliquez ce que fait cette méthode, donnez un code où elle est utile, en expliquant le problème si l'on ne fait pas de `deepcopy`.

Solution. → Voir syllabus.

Question 2 - Code Python (5 points)

1. Que font chacune des 2 fonctions `foo1` et `foo2`, qui reçoivent toutes deux les deux listes `a` et `b` ?

```
def foo1(a,b):                                          # moyenne : O(n)  maximale : O(n^2)
    res = False                                           #  O(1)              O(1)
    if len(a)==len(b):                                    #  O(1)              O(1)
       dic_a = {}                                         #  O(1)              O(1)
       dic_b = {}                                         #  O(1)              O(1)
       for i in a:                          #  O(n) itérations  O(n)              O(n^2)
           dic_a[i]=dic_a.get(i,0)+1                      #  O(1)              O(n)
       for i in b:                          #  O(n) itérations  O(n)              O(n^2)
           dic_b[i]=dic_b.get(i,0)+1                      #  O(1)              O(n)
       res = True                                         #  O(1)              O(1)
       for j in dic_a:              # m irération (max n) :  O(m)              O(n^2)
           res = res and (dic_a[j] == dic_b.get(j,0))     #  O(1)              O(n)
       for j in dic_b:              # m irération (max n) :  O(m)              O(n^2)
           res = res and (dic_b[j] == dic_a.get(j,0))     #  O(1)              O(n)
    return res                                            #  O(1)              O(1)

def foo2(a,b):                                          # moyenne : O(n^2)  maximale : O(n^2)
    res = False                                           #  O(1)              O(1)
    if len(a)==len(b):                        # test en O(1) O(n^2)              O(n^2)
       res = True                                         #  O(1)              O(1)
       for i in a:                         # O(n) itérations  O(n^2)              O(n^2)
           if i in b:                                     #  O(n)              O(n)
               a.remove(i)                                #  O(n)              O(n)
               b.remove(i)                                #  O(n)              O(n)
           else:                                          #  O(1)              O(1)
               res = False                                #  O(1)              O(1)
    return res                                            #  O(1)              O(1)
```

Solution. Les deux fonction renvoient toutes les deux vrai si et seulement si les deux listes `a` et `b` ont les

même éléments à l'ordre prêt.

La fonction `foo_1` construit un dictionnaire pour chacune des deux listes avec comme clés les valeurs dans `a` (resp. `b`) et comme valeur la fréquence de la valeur dans la liste. Ensuite, le code regarde si les deux dictionnaires sont égaux (`dic_a == dic_b`).

La fonction `foo_2` teste si les deux listes ont le même nombre d'éléments. Ensuite, elle prend chaque élément de `a` et s'il est également dans `b` le retire des deux listes ; si cela se passe jusqu'au bout, renvoie vrai sinon faux.

2. En supposant que les éléments soient entiers, donner la complexité moyenne et maximale en $O()$ du temps d'exécution de chacune des deux fonctions. Vous pouvez annoter le code pour répondre à la question. Mais veillez bien à détailler vos réponses en justifiant la complexité de chaque partie du code.

Solution. Voir code. Pour la complexité moyenne de `foo_2` l'hypothèse est que la boucle for a `O(n)` itérations.

Question 3 - Opérations sur les fichiers (4 points)

Ecrivez une fonction `import_dict(fname, key_idx)` qui lit un fichier texte dont le nom est donné par le paramètre `fname`. Chaque ligne du fichier contient le même nombre de colonnes, celles-ci étant séparées par une ligne verticale ("`|`", cf. exemple ci-dessous). Le résultat de la fonction sera un dictionnaire mappant des strings à des listes de listes de strings. Le paramètre `key_idx` représente l'indice (de base 0) de la "colonne" dans le fichier texte servant de clef dans le dictionnaire. A chaque clef sera donc associée une liste contenant autant de sous-listes que d'occurrences de la clef dans le fichier lu. Chaque sous-liste reprend l'ensemble des colonnes de la ligne correspondante.

Exemple. La fonction `import_dict` appliquée au fichier "`import_dict_example.txt`" ci-dessous produira les dictionnaires suivants, en fonction de la valeur du deuxième paramètre `key_idx` valant respectivement `1` (deuxième colonne) ou `2` (troisième colonne).

Fichier `import_dict_example.txt`

```
Antoine|ULB|Polytech|BA1
Issam|UCL|Philo|BA1
Wassim|ULB|Philo|MA1
Dounia|ULG|Polytech|BA2
Aurélie|ULB|Polytech|BA3
Boris|ULB|Droit|MA2
```

```
>>> print(import_dict("import_dict_example.txt",1))
{'ULB': [['Antoine', 'ULB', 'Polytech', 'BA1'],
         ['Wassim', 'ULB', 'Philo', 'MA1'],
         ['Aurélie', 'ULB', 'Polytech', 'BA3'],
         ['Boris', 'ULB', 'Droit', 'MA2']],
 'UCL': [['Issam', 'UCL', 'Philo', 'BA1']],
 'ULG': [['Dounia', 'ULG', 'Polytech', 'BA2']]}

>>> print(import_dict("import_dict_example.txt",2))
{'Polytech': [['Antoine', 'ULB', 'Polytech', 'BA1'],
              ['Dounia', 'ULG', 'Polytech', 'BA2'],
              ['Aurélie', 'ULB', 'Polytech', 'BA3']],
 'Droit': [['Boris', 'ULB', 'Droit', 'MA2']],
 'Philo': [['Issam', 'UCL', 'Philo', 'BA1'],
           ['Wassim', 'ULB', 'Philo', 'MA1']]}
```

Note. Une chaine de caractères `s` peut être transformée en liste en utilisant `s.split(sep)`, où `sep` représente le (ou les) caractère(s) de séparation. Par exemple, l'instruction `"a|b|c".split('|')` renverra la liste `['a', 'b', 'c']`.

Solution.

```
def import_dict(fname, key_idx):
    res = {}
    f = open(fname)
    for line in f:
        data = line.strip().split('|')
        key = data[key_idx]
        if not key in res:
            res[key] = []
        res[key].append(data)
    f.close()
    return res
```

La solution ci-dessus reprend un shéma courant avec les dictionnaires : Si une clé n'y apparait pas déjà, la rajouter avec une valeur associée par défaut, sinon, modifier la valeur associée. La méthode `setdefault` permet justement ça. Les trois lignes

```
        if not key in res:
            res[key] = []
        res[key].append(data)
```

peuvent donc être simplement remplacées par :

```
        res.setdefault(key, []).append(data)
```

Remarque : Une solution produisant un résultat correct mais néanmoins non satisfaisante serait :

```
def import_dict(fname, key_idx):
    """
    Mauvaise solution qui n'utilise pas les vertus des dictionnaires
    """
    f = open(fname)
    list_file = []
    for line in f:
        list_file.append(line.strip().split('|'))
    f.close()

    res = {}
    for ls1 in list_file:
        key = ls1[key_idx]
        res[key] = []
        for ls2 in list_file:
            if ls2[key_idx] == key:
                res[key].append(ls2)
    return res
```

Cette solution est de complexité quadratique en le nombre de lignes du fichier alors que la complexité moyenne de la solution proposée plus haut est linéaire. Par ailleurs, cette solution nécessite de mémoriser l'entièreté du fichier sous forme de liste (`list_file`).

Question 4 - Tri (5 points)

Considérez que vous ayez un sac à dos, pouvant contenir des objets qui pèsent, en tout, au maximum C kg. C sera la capacité du sac.

Considérez ensuite un ensemble d'objets, chacun étant défini par son indice i et caractérisé par sa masse m_i. On suppose, pour que la question ne soit pas triviale, que le poids cumulé de tous les objets dépasse la capacité du sac à dos : $\sum_i m_i > C$.

Nous cherchons à déterminer la combinaison d'objets à placer dans le sac pour que la somme des masses de ces objets sélectionnés s'approche au plus près de la capacité C du sac, sans la dépasser. (Il s'agit de l'énoncé simplifié du problème du sac à dos, l'archétype d'une famille de problèmes combinatoires.)

Une manière d'obtenir simplement une solution approximative à ce problème est nommée "l'approche gloutonne". Elle consiste à trier l'ensemble des objets "candidats" dans l'ordre décroissant de leur poids et de remplir le sac dans cet ordre, tout en veillant à ne pas dépasser la capacité du sac.

Ecrivez une fonction `greedy_knapsack(capacity, weights)` qui résout le problème du sac à dos par l'approche gloutonne. Le paramètre `capacity` représente la capacité C du sac à dos (en kg) ; le paramètre `weights` représente une liste (non triée) de poids des objets que nous voudrions placer dans le sac. Le résultat renvoyé par la fonction sera un tuple contenant la masse totale des objets sélectionnés, ainsi que la liste des indices qui identifieront ces objets.

Il s'agit d'une question sur le tri. On vous demande donc d'écrire vous-même la fonction de tri en utilisant uniquement l'une des méthodes vues aux séances pratiques : tri par insertion ou par sélection. Vous ne pouvez pas utiliser la fonction `sort`.

Exemples.

```
>>> weights = [2, 5, 7, 12, 4]
>>> print(greedy_knapsack(13, weights))
(12, [3])

>>> print(greedy_knapsack(15, weights))
(14, [3, 0])

>>> print(greedy_knapsack(22, weights))
(21, [3, 2, 0])
```

Note. Prenez garde de bien renvoyer les indices des objets de la liste non triée, passée en paramètre (`objects`). Pour garder la trace de ces indices, nous vous suggérons de construire (en variable locale) une nouvelle liste de *tuples* qui associe à chaque élément (poids) de `objects` son indice dans la liste originale : `[(objects[0],0), (objects[1],1), ...]`.

Solution.

```
def make_idx_weights(objects):
    return list(zip(objects, range(len(objects))))

def sort_ls(ls):
    for i in range(len(ls)-1):
        jmax = i
        for j in range(i+1, len(ls)):
            if ls[j] > ls[jmax]:
                jmax = j
        ls[i], ls[jmax] = ls[jmax], ls[i]

def greedy_knapsack(capacity,objects):
    idx_weights = make_idx_weights(objects)
    sort_ls(idx_weights)
    selected = []
    total = 0
    i = 0
    while total<capacity and i<len(objects):
        if total + idx_weights[i][0] <= capacity:
            selected.append(idx_weights[i][1])
            total += idx_weights[i][0]
        i += 1
    return (total, selected)
```

Une autre solution possible consiste à "intégrer" le tri (par sélection) et la sélection des objets à placer dans le sac dans une même boucle. Cette solution exploite le fait que la boucle de la sélection des objets est quasiment la même que la boucle extérieure du tri. Par contre, cette solution est plus spécifique et donc moins modulaire que la première. Notez enfin que cette approche ne fonctionne que pour la méthode de tri par sélection.

```
def make_idx_weights(weights):
    return list(zip(weights, range(len(weights))))

def greedy_knapsack(capacity,weights):
    ls = make_idx_weights(weights)
    selected = []
    total = 0
    i = 0
    while total<capacity and i<len(ls):
        jmax = i
        for j in range(i+1, len(ls)):
            if ls[j] > ls[jmax]:
                jmax = j
        if total+ls[jmax][0] <= capacity:
            selected.append(ls[jmax][1])
            total += ls[jmax][0]
        ls[jmax] = ls[i]
        i += 1
    return (total, selected)
```

Question 5 - Récursivité (4 points)

Ecrivez la fonction `count_sublists(ls)` qui reçoit en paramètre une liste `ls`, dont chaque élément peut être un entier ou une sous-liste, et qui renvoie le nombre total de sous-listes dans `ls`. Notez que chaque sous-liste peut, à son tour, contenir des entiers ou des sous-listes.

Exemples

```
>>> count_sublists([0,1,2,3])
0

>>> count_sublists([0,1,[2,4,8],3])
1

>>> count_sublists([0,1,[2,4,8],[3,6,9]])
2

>>> count_sublists([0,1,[2,[4,8,16],8],[3,6,9]])
3
```

Note. Si le paramètre `ls` est une liste d'entiers qui ne contient aucune sous-liste, la fonction renvoie 0 (zéro). Vous pouvez tester si une variable `x` est une liste en utilisant l'expression booléenne `(type(x) == list)`.

Solution.

```
def count_sublists(ls):
    res = 0
    for x in ls:
        if type(x) == list:
            res += 1 + count_sublists(x)
    return res
```