 août 2017

INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Examen de août – ***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 (3 points)

La fonction suivante teste si le paramètre (entier strictement positif) passe le "test de Syracuse".

```
def test_syracuse(n):
    """Teste si n n'est pas un contre-exemple à la conjecture de syracuse."""
    if  n == 1:
        res = True
    else:
        if n%2 == 0:
            n = n//2
        else:
            n = 3*n+1
        res = test_syracuse(n)
    return res
```

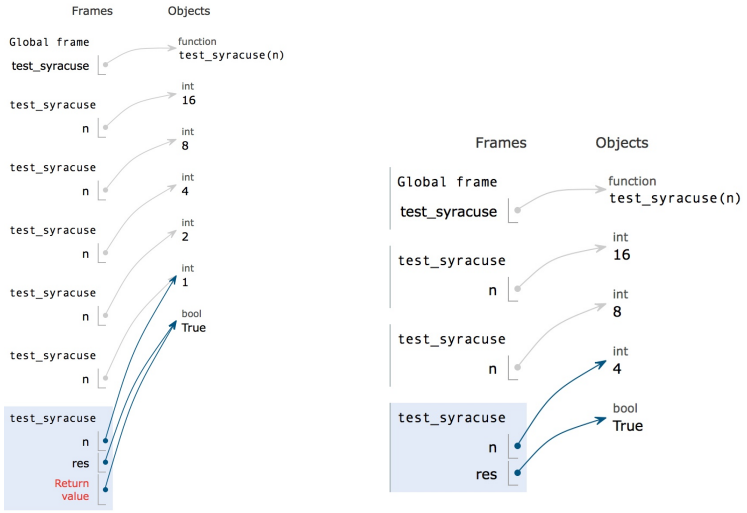
Montrez, grâce à des diagrammes d'état comment s'effectue la gestion de la mémoire lors de l'exécution de cette fonction appelée avec le paramètre 5 : `test_syracuse(5)`.

Dans vos diagrammes d'états, veillez à donner complètement les objets existant en mémoire.

Expliquez, grâce à cet exemple, comment fonctionne la *pile* et le *tas* d'exécution (*run time stack and heap*).

Solution. Explications : voir syllabus section 11.5.2.

(Premier diagramme d'état) état de la mémoire au 6ème appel récursif de `test_syracuse` et (second diagramme) au retour pour $n = 4$:



Question 2 - Code Python (4 points)

Pour chacun des codes suivants,

- donnez ce qu'il imprime (avec les appels qui sont donnés) et expliquez de façon générale ce que fait la fonction (si elle résout un problème, lequel ...) ;
- donnez l'état du programme lors de chaque (groupe de) `print` grâce à des diagrammes d'état (comme fait au cours) ;
- sachant que les types des paramètres sont,
  - n : nombre naturel,
  - $p1$, $p2$: listes de float que vous pouvez supposer de taille n, qui encodent des polynomes : chaque élément d'indice i représente le coefficient de degré i du polynome encodé.

  donnez la complexité moyenne et maximale ($\mathcal{O}()$) de chacune des fonctions **foo_1, add, multiply_one, multiply)** 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).

Notez bien : pour expliquer ce que le code fait et donner la complexité, vous pouvez annoter les codes fournis (compléter les docstrings, commenter, ...)

```
def foo_1(n):                                   #  O(n^2)  O(n^2)
    dico = dict(zip(list(range(n)), [['x']*n for i in range(n)]))    # O(n^2)  O(n^2)
    d2 = dico                                   # O(1)  O(1)
    z = [i*2 for i in dico]                     # O(n)  O(n^2)
    if 7 in d2:                                 # O(n)  O(n^2) (test en O(1)  O(n))
        print(d2[7])                            #   O(n)  O(n^2)
    print(dico)                                 # O(n^2)  O(n^2)
    print(d2)                                   # O(n^2)  O(n^2)
    print(z)                                    # O(n)  O(n)
foo_1(3)
```

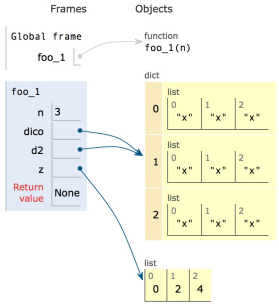
Solution :

Imprime :

```
{0: ['x', 'x', 'x'], 1: ['x', 'x', 'x'], 2: ['x', 'x', 'x']}
{0: ['x', 'x', 'x'], 1: ['x', 'x', 'x'], 2: ['x', 'x', 'x']}
[0, 2, 4]
```

Ce code ne semble pas résoudre de problème spécifique.

La création de la liste est en moyenne en $O(n)$ (nombre d'éléments). Diagramme d'état :



Complexité de la fonction `foo_1` : moyenne = $O(n^2)$; max = $O(n^2)$ (voir code). En effet, les dictionnaires ont n éléments qui sont chacun une liste de n éléments. L'insertion moyenne dans un dictionnaire est en $O(1)$ et il y en a $O(n)$, mais chaque insertion demande d'abord la création d'une liste de $O(n)$ éléments donc au total une complexité moyenne en $O(n^2)$.
Pour la complexité maximale, chaque insertion peut au pire des cas prendre $O(n)$, mais comme de toute façon, la création de chaque liste prend $O(n)$, on obtient également $O(n^2)$.
l'assignation `d2 = dico` est en $O(1)$.
Pour la création de la liste `z`, la complexité moyenne est en $O(n)$ puisque elle fait $O(n)$ éléments simples. Au pire des cas, lors de chaque insertion il faut recopier la liste (soucis de place mémoire) et cele prend $O(n)$ donc un total de $O(n^2)$.
Le test est en moyenne en $O(1)$ et au maximum en $O(n)$. Le print de `d2[7]` est en $O(n)$ dans les 2 cas, car l'accès est en $O(1)$ (moyenne) $O(n)$ (max) suivi de l'impression qui de toute façon est en $O(n)$ (taille de chaque liste de `d2`).
Le print de `dico` et de `d2` sont en $O(n^2)$ dans les 2 cas, car $O(n)$ accès en $O(1)$ suivi de l'impression qui est en $O(n)$ (taille de chaque liste de `dico` et `d2`).
Le print de `z` est en $O(n)$ (nombre d'éléments)

```
def add(p1,p2):                                          # O(n)  (avec n = max(len(p1),len(p2))
    """Renvoie la somme de 2 polynomes"""
    if len(p1) < len(p2):                                #   O(1)
        p1,p2 = p2,p1                                    #     O(1)
    new = p1[:]                                          #   O(n)
    for i in range(len(p2)):                             #   O(n)
        new[i] += p2[i]                                  #     O(1)
    return new                                           #   O(1)

def mult_one(p,c,i):                                     # pire des cas : O(n^2)  (moyenne O(n))
    """Renvoie le produit du polynome p avec le terme c*x^i"""
    new = [0]*i #termes 0 jusque i-1 valent 0            #   O(n)
    for pi in p:                                         #   O(n^2)  (moyenne O(n))
        new.append(pi*c)                                 #     O(n)  (moyenne O(1))
    return new                                           #   O(1)

def multiply(p1,p2):                                     # pire des cas: O(n^3)  (moyenne O(n^2)
    """ Renvoie le produit de 2 polynomes"""
    if len(p1) > len(p2):                                #   O(1)
        p1,p2 = p2,p1                                    #     O(1)
    new = []                                             #   O(1)
    for i in range(len(p1)):                             #   pire cas: O(n^3)  (n appels)  (moyenne O(n^2))
        new = add(new,mult_one(p2,p1[i],i))              #     pire des cas: O(n^2)
                                                         #    (moyenne O(n))  (mult_one suivi de add)
    return new                                           #   O(1)

print(multiply([1.0, 2.0], [1.0, 1.0, 3.0]))
```

Solution imprime

```
[1.0, 1.0, 3.0]
[1.0, 1.0, 3.0]
[0, 2.0, 2.0, 6.0]
[1.0, 3.0, 5.0, 6.0]
[1.0, 3.0, 5.0, 6.0]
[1.0, 3.0, 5.0, 6.0]
```

but du code
La fonction multiply réalise de produit de deux polynômes avec la convention donnée plus haut.

complexité
Voir code et syllabus pour les explications détaillées de la complexité de chaque instruction.

diagrammes d'état

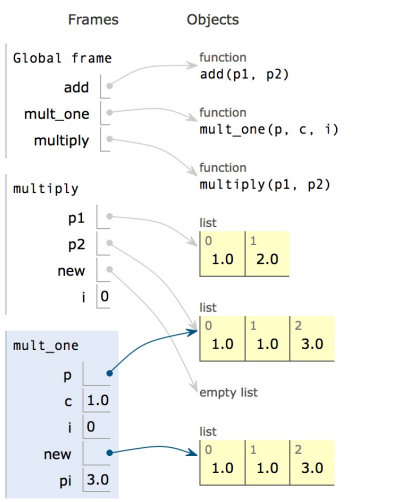
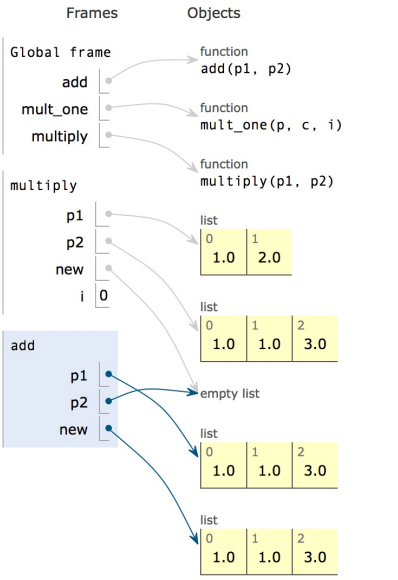


FIGURE 1 – à la fin de mult_one appelé par multiply



et à la fin de add appelé par multiply

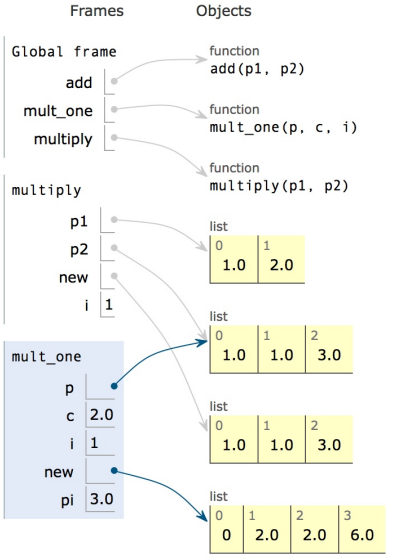
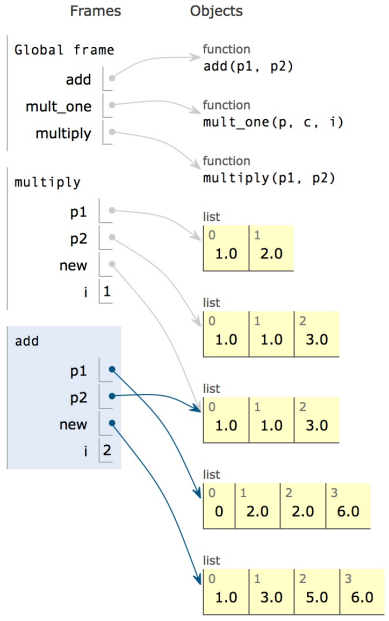


FIGURE 2 – multiply appelle mult_one (retour à la fin de mult_one)



et multiply appelle add (retour à la fin de add)

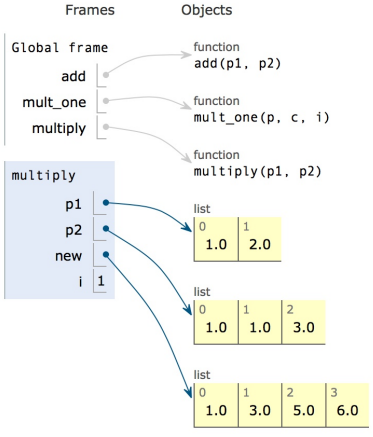


FIGURE 3 – retour de l'appel à multiply

Question 3 - Fichier (5 points)

Considérons le fichier d'une société représentant (de façon extrêmement simplifiée) les achats de clients. Les informations sont mémorisées dans un fichier texte. Chaque ligne du fichier porte sur un achat effectué par le client correspondant.
Il s'agit de la même question que lors de l'examen de Juin mais, cette fois, **le fichier n'est pas trié**.

Par exemple le fichier suivant indique que le Client2 a effectué à différents moments trois achats pour des montants respectifs de 300, 120 et 440 €.

```
Client4  210
Client2  300
Client4  100
Client2  120
Client3  500
Client1  200
Client5  520
Client1  140
Client6  200
Client2  440
```

Ecrire la fonction `pack(fNameIn, fNameOut)` qui écrit dans le fichier `fNameOut` les achats regroupés de façon à ce que les clients n'apparaissent plus qu'une fois avec le montant cumulé (la somme) de leurs achats L'ordre des clients dans le fichier résultat peut être quelconque.

```
Client1  340
Client5  520
Client4  310
Client2  860
Client6  200
Client3  500
```

Remarque : Il n'est pas raisonnable (acceptable) de lire plusieurs fois le même fichier.

Solution.

```python
def lecture(fNameIn):
    dico = {}
    f = open(fNameIn)
    for line in f:
        line = line.strip().split()
        dico[line[0]] = dico.get(line[0], 0) + int(line[1])
    return dico

def ecriture(ls, fNameOut):
    f = open(fNameOut, 'w')
    for line in ls:
        f.write(line[0] + ' ' + str(line[1]) + '\n')
    f.close()

def pack(fNameIn, fNameOut):
    dico = lecture(fNameIn)
    ecriture(dico.items(), fNameOut)
```

Question 4 - Tri (4 points)

Dans une entreprise, de nombreuses demandes sont envoyées au service technique pour diverses raisons. Afin de savoir dans quel ordre il faut traiter ces demandes, il est nécessaire de les trier. Ces demandes sont représentées sous forme de tuples de 3 éléments : *(attente, ancienneté, nom)*. Le premier élément est un entier représentant le nombre de jours écoulés depuis l'envoi de la demande. Le second élément est un entier représentant les années d'ancienneté de l'employé envoyant la demande. Le troisième élément est une chaîne de caractères représentant le nom de l'employé en question.
Le tri doit se faire selon les règles suivantes :
- Les demandes formulées par des employés ayant au moins 10 ans d'ancienneté sont traîtées avant les autres.
- Si des employés appartiennent à la même catégorie (tous les deux plus de 10 ans d'ancienneté, ou tous les deux moins de 10 ans), les demandes les plus anciennes doivent être traîtées en premier.

– Si les deux points suivants n'arrivent pas à départager deux demandes, elles seront triées selon l'ordre lexicographique du nom de l'employé en question

Écrivez une fonction *tri_demandes(data)* où data est une liste comprenant des tuples dans le format décrit plus haut. Utiliser le tri par insertion ou le tri par sélection. La fonction doit être codée de la manière la plus optimale possible.

```
data=[(3,9,'Jean'),(2,11,'Zaza'),(5,7,"Jerome"),(3,8,'Anna'),(2,14,'Yves')]
tri_demandes(data)
print(data) # Affiche : [(2, 14, 'Yves'), (2, 11, 'Zaza'), (5, 7, 'Jerome'), (3, 8, 'Anna'), (3, 9, 'Jean')]
```

Solution.

```
def tri_demandes(data):

    for i in range(len(data)-1):
        first=i
        for j in range(i+1,len(data)):
            if is_before(first,j,data):

                first=j
        data[i],data[first]=data[first],data[i]
def is_before(a,b,data):
    return ((data[b][1]>=10) and not (data[a][1]>=10)) or ((data[b][1]>=10) == (data[a][1]>=10) and data[b][0]>data[a][0]) or
```

Question 5 - Récursivité (4 points)

La fonction de Sudan est une fonction récursive analogue à celle de Ackerman, utilisée en informatique théorique. Elle est définie comme suit :

– $F_0(x, y) = x + y$
– $F_{n+1}(x, 0) = x,\ n \geq 0$
– $F_{n+1}(x, y + 1) = F_n(F_{n+1}(x, y), F_{n+1}(x, y) + y + 1),\ n \geq 0.$

Veuillez écrire une fonction $sudan(n, x, y)$ qui renvoie la valeur du nombre de Sudan, $F_n(x, y)$, pour n, x, y donnés en arguments.

Solution.

```
def sudan(n,x,y):
    if n == 0:
        res = x + y
    elif y==0:
        res = x
    else:
        res = sudan(n-1,sudan(n,x,y-1),sudan(n,x,y-1)+y)
    return res

result = sudan(2,1,2)
print(result) #10228
```