Mathématiques supérieures — le Lundi 3 Novembre 2024

Correction du DS d'Informatique n°2

Ce sujet est composé d'un exercice et un problème. Vous prendrez soin de bien justifier vos calculs. Veillez à bien respecter l'indentation des programmes Python que vous écrirez sur votre copie.

Exercice 1. *Extrait de Centrale 2015*

I Quelques fonctions utilitaires

I.A – Donner la valeur des expressions python suivantes :

I.A.1) `[1, 2, 3] + [4, 5, 6]`

I.A.2) `2 * [1, 2, 3]`

I.B – Écrire une fonction python `smul` à deux paramètres, un nombre et une liste de nombres, qui multiple chaque élément de la liste par le nombre et renvoie une nouvelle liste : `smul(2, [1, 2, 3])` → `[2, 4, 6]`.

I.C – Arithmétique de listes

I.C.1) Écrire une fonction python `vsom` qui prend en paramètre deux listes de nombres de même longueur et qui renvoie une nouvelle liste constituée de la somme terme à terme de ces deux listes : `vsom([1, 2, 3], [4, 5, 6])` → `[5, 7, 9]`.

I.C.2) Écrire une fonction python `vdif` qui prend en paramètre deux listes de nombres de même longueur et qui renvoie une nouvelle liste constituée de la différence terme à terme de ces deux listes (la première moins la deuxième) : `vdif([1, 2, 3], [4, 5, 6])` → `[-3, -3, -3]`.

Correction.

I.A.1) Le `+` réalise la concaténation entre deux listes :

```
>>> [1,2,3]+[4,5,6]
[1,2,3,4,5,6]
```

I.A.2) L'instruction `n*L` est équivalente à `L+...+L` avec `n` termes `L` donc l'instruction `2*[1,2,3]` est équivalente à `[1,2,3]+[1,2,3]` :

```
>>> 2*[1,2,3]
[1,2,3,1,2,3]
```

I.B

```python
def smul(a,L):
    Res = []
    for i in range(len(L)):
        Res.append(a*L[i])
    return Res
```

I.C.1)

```python
def vsom(L1,L2):
    """ L1 et L2 sont supposées de même taille """
    Res = []
    for i in range(len(L1)):
        Res.append(L1[i]+L2[i])
    return Res
```

I.C.2)

```python
def vdif(L1,L2):
    """ L1 et L2 sont supposées de même taille """
    Res = []
    for i in range(len(L1)):
        Res.append(L1[i]-L2[i])
    return Res

# ou en utilisant les fonctions précédentes (mais on parcourt deux fois la liste L2 contre une seule fois avec la fonction précédente)

def vdif(L1,L2):
    return vsom(L1,smul(-1,L2))
```

Problème. Mines 2017 (début)

Ce sujet concerne la conception d'un logiciel d'étude de trafic routier. On modélise le déplacement d'un ensemble de voitures sur des files à sens unique (voir Figure 1). C'est un schéma simple qui peut permettre de comprendre l'apparition d'embouteillages et de concevoir des solutions pour fluidifier le trafic.
Le sujet comporte des questions de programmation. Le langage à utiliser est Python.
Notations : soit L une liste,

- on note $len(L)$ sa longueur ;
- pour i entier, $0 \leqslant i < len(L)$, l'élément de la liste d'indice i est noté L[i] ;
- pour i et j entiers, $0 \leqslant i < j \leqslant len(L)$, $L[i : j]$ est la sous-liste composée des éléments $L[i], ..., L[j-1]$;
- $p * L$, avec p entier, est la liste obtenue en concaténant p copies de L. Par exemple, $3 * [0]$

est la liste $[0, 0, 0]$.

(a) Représentation d'une file de longueur onze comprenant quatre voitures, situées respectivement sur les cases d'indices 0, 2, 3 et 10.

FIGURE 1 – File de circulation

1 - Préliminaires

On considère le cas d'une seule file, illustré par la Figure 1. Une file de longueur n est représentée par n cases. Une case peut contenir au plus une voiture. Les voitures présentes dans une file circulent toutes dans la même direction (sens des indices croissants, désigné par les flèches sur la Figure 1) et sont indifférenciées.

1. Expliquer comment représenter une file de voitures à l'aide d'une liste de booléens.

   On rappelle qu'un booléen est un objet Python admettant deux valeurs possibles : soit `True` *soit* `False`*, où* `True` *correspond à "Vrai" et* `False` *correspond à "Faux".*

2. Donner une ou plusieurs instructions `Python` permettant de définir une liste A représentant la file de voitures illustrée par la Figure 1.
3. Soit L une liste représentant une file de longueur n et i un entier tel que $0 \leqslant i < n$. Définir en Python la fonction `occupe(L, i)` qui renvoie `True` lorsque la case d'indice i de la file est occupée par une voiture et `False` sinon.
4. Combien existe-t-il de files différentes de longueur n ? Justifier votre réponse.
5. Écrire une fonction `egal(L1, L2)` retournant un booléen permettant de savoir si deux listes $L1$ et $L2$ sont égales.
6. Préciser le type de retour de cette fonction.

2 - Déplacement de voitures dans la file

On identifie désormais une file de voitures à une liste. On considère le schéma de la Figure 2 représentant un exemple de file. Une étape de simulation pour une file consiste à déplacer les voitures de la file, à tour de rôle, en commençant par la voiture la plus à droite, d'après les règles suivantes :

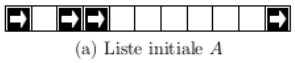
— une voiture se trouvant sur la case la plus à droite de la file sort de la file ;

— une voiture peut avancer d'une case vers la droite si elle arrive sur une case inoccupée ;

— une case libérée par une voiture devient inoccupée ;

— la case la plus à gauche peut devenir occupée ou non, selon le cas considéré.

On suppose avoir écrit en Python la fonction `avancer` prenant en paramètres une liste de départ, un booléen indiquant si la case la plus à gauche doit devenir occupée lors de l'étape de simulation, et renvoyant la liste obtenue par une étape de simulation.
Par exemple, l'application de cette fonction à la liste illustrée par la Figure 2(a) permet d'obtenir soit la liste illustrée par la Figure 2(b) lorsque l'on considère qu'aucune voiture nouvelle n'est introduite, soit la liste illustrée par la Figure 2(c) lorsque l'on considère qu'une voiture nouvelle est introduite.



(a) Liste initiale A

(b) $B =$ `avancer`$(A,$ `False`$)$

(c) $C =$ `avancer`$(A,$ `True`$)$

FIGURE 2 – Étape de simulation

7. Étant donnée A la liste définie à la question 2, que renvoie l'instruction suivante :

```
>>>avancer(avancer(A, False),True)
```

8. On considère L une liste et m l'indice d'une case de cette liste ($0 \leqslant m < len(L)$). On s'intéresse à une étape partielle où seules les voitures situées sur la case d'indice m ou à droite de cette case peuvent avancer normalement, les autres voitures ne se déplaçant pas.

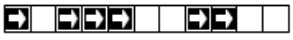
   Par exemple, la file 

devient 

.

   Définir en Python la fonction `avancer_fin(L, m)` qui réalise cette étape partielle de déplacement et renvoie le résultat dans une nouvelle liste sans modifier L.

9. Soient L une liste, b un booléen et m l'indice d'une case inoccupée de cette liste. On considère une étape partielle où seules les voitures situées à gauche de la case d'indice m se déplacent, les autres voitures ne se déplacent pas. Le booléen b indique si une nouvelle voiture est introduite sur la case la plus à gauche.

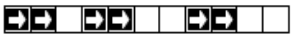
   Par exemple, la file 

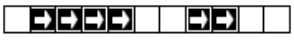
devient 

lorsque aucune nouvelle voiture n'est introduite.

   Définir en Python la fonction `avancer_debut(L, b, m)` qui réalise cette étape partielle de déplacement et renvoie le résultat dans une nouvelle liste sans modifier L.

10. On considère une liste L dont la case d'indice $m > 0$ est temporairement inaccessible et bloque l'avancée des voitures. Une voiture située immédiatement à gauche de la case d'indice m ne peut pas avancer. Les voitures situées sur les cases plus à gauche peuvent avancer, à moins d'être bloquées par une case occupée, les autres voitures ne se déplacent pas. Un booléen b indique si une nouvelle voiture est introduite lorsque cela est possible.

    Par exemple, la file 

devient 

,lorsque aucune nouvelle voiture n'est introduite.

    Définir en Python la fonction `avancer_debut_bloque(L, b, m)` qui réalise cette étape partielle de déplacement et renvoie le résultat dans une nouvelle liste.

Correction.

1. On représente une file de longueur n par une liste `L` de longueur n où chaque élément `L[i]` (pour i entre 0 et $n-1$) vaut `True` s'il y a une voiture sur la case d'indice i ($i+1$-ième case) de la file et `False` s'il n'y en a pas.

2. Par exemple :

```
>>>A=[True,False,True,True,False,False,False,False,False,False,True]
```

3. Version très concise (mais parfaitement juste) :

```
def occupe(L,i):
    return L[i]
```

Version peut-être plus facile à assimiler au départ

```
def occupe(L,i):
    if L[i]==True: #ici "if L[i]:" suffisait aussi :)
        return True
    else:
        return False
```

4. Pour chaque case d'une file de longeur n, on a deux possibilités : soit il y a une voiture, soit il n'y en a pas ; on peut représenter l'ensemble des files de longueur n par l'ensemble des n-uplets de $\{0,1\}^n$ où la présence d'une voiture correspond à 1 et l'absence à 0. Le nombre de files de longueur n est donc égal au nombre d'éléments de $\{0,1\}^n$, en l'occurence, 2^n.

5. Algorithme de comparaison de deux listes. On part du fait suivant : deux listes `L1`, `L2` sont égales si, et seulement si, elles sont de même longueur `n` et que pour tout indice `i` entre 0 et `n`-1, `L1[i]` et `L2[i]` sont égaux puis on utilise sa contraposée pour coder notre fonction :

```
def egal(L1,l2):
    n=len(L1)
    m=len(L2)
    if n!=m: # si les tailles sont différentes : False
        return False
    else: #si elles sont égales ...
        for i in range(n):
            if L1[i]!=L2[i]: # ... mais que deux termes diffèrent : False
                return False
        #... et boucle terminée, tous les éléments sont donc égaux terme
            à terme : True
        return True
```

6. Cette fonction renvoie `True` ou `False` ; le type de retour est donc **booléen**.

7. Pour `A=[True,False,True,True,False,False,False,False,False,False,True]`, on obtient :

```
>>>avancer(avancer(A,False),True)
[True,False,True,False,True,True,False,False,False,False,False]
```

8. L'idée est d'utiliser la fonction `avancer` sur la sous-liste des éléments d'indice plus grand ou égaux à m (`L[m:]`) et de ne pas toucher à la sous-liste des éléments d'indice entre 0 et $m-1$ (`L[:m]`) :

```
def avancer_fin(L,m):
    return L[:m]+avancer(L[m:],False) #False car bien-sûr on ne rajoute
        pas de voiture
```

9. L'idée est essentiellement la même que pour la question précédente : on utilise la fonction `avancer` sur la sous-liste des éléments d'indice de 0 à m (`L[:m+1]`) et on ne touche pas à la sous-liste des éléments d'indice plus grand ou égal à $m+1$ (`L[m+1:]`) :

```
def avancer_debut(L,b,m):
    return avancer(L[:m+1],b)+L[m+1:]
```

10. L'idée est de déterminer l'indice `j` de la dernière case vide (strictement) avant celle d'indice `m`. Si cet indice `j` existe, on applique la fonction précedente `avancer_debut` en `j` ; sinon, la liste ne change pas.

```
def avancer_debut_bloque(L,b,m):
    j=m-1
    while j>=0 and L[j]: #tant que l'indice j est valide et qu'il y a une
         voiture à la case j
        j=j-1 #on teste avec indice d'avant
    if j>=0: #si j>=0 c'est que la boucle s'est arrêtée "à cause" du fait
         que L[j] vaut False, et donc que la case j est vide
        return avancer_debut(L,b,j)
    else: #sinon, pas de case vide entre 0 et m-1
        return L
```