Samedi 15 janvier MPSI/PCSI

# Devoir surveillé n° 2

## Informatique – durée : 2 heures

L'utilisation de la calculatrice est **interdite**

***Consignes :*** *Le seul langage de programmation autorisé dans cette épreuve est Python. Les copies rendues seront numérotées.*

## Problème 1 : Étude de trafic routier

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 Figures 1 et 2). C'est un schéma simple qui peut permettre de comprendre l'apparition d'embouteillages et de concevoir des solutions pour fluidifier le trafic.

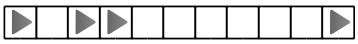


**Figure 1 – Représentation d'une file de longueur onze comprenant quatre voitures, situées respectivement sur les cases d'indice 0, 2, 3 et 10.**

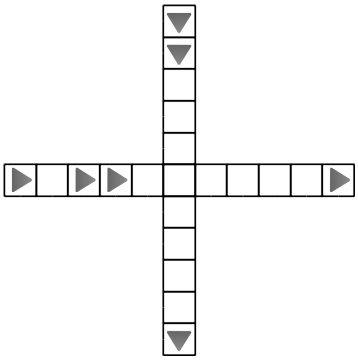


**Figure 2 – Configuration représentant deux files de circulation à sens unique se croisant en une case. Les voitures sont représentées par une flèche.**

**Notations :**

Soit `L` une liste,

- on note `len(L)` sa longueur ;
- pour `i` entier, `0 <= i < len(L)`, l'élément de la liste d'indice `i` est noté `L[i]` ;
- pour `i` et `j` entiers, `0 <= i < j <= 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]`.
- on peut concaténer deux listes `L1` et `L2` avec `+`. Par exemple, `L = [1, 2, 3] + [4, 5, 2]` est la liste `[1, 2, 3, 4, 5, 2]`.

D'autres fonctions Python éventuellement utiles sont rappelées en fin d'énoncé.

## Partie I. Préliminaires

Dans un premier temps, 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.

**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 $0 \leq i < n$. 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)` permettant de savoir si deux listes `L1` et `L2` sont égales.

**6.** Que peut-on dire de la complexité de cette fonction ?

**7.** Préciser le type de retour de cette fonction.

**8.** Écrire une fonction `nombre_occ(L)` renvoyant le nombre de places occupées dans une file représentée par la liste `L`.

**9.** Écrire une fonction `max_succ(L)` qui renvoie le plus grand nombre de places occupées à la suite dans la file représentée par la liste `L`. Par exemple, pour la liste `A` représentant la file illustrée par la Figure 1, cette fonction doit renvoyer 2.

**Remarque :** Les fonctions `nombre_occ(L)` et `max_succ(L)` étudiées dans les questions précédentes ne seront pas utilisées dans la suite du sujet.

## Parie II. Déplacement de voitures dans la file

On identifie désormais une file de voitures à une liste. On considère les schémas de la Figure 3 représentant des exemples de files. 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 et 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 l'image du haut de la Figure 3 permet d'obtenir soit la liste illustrée par l'image du milieu de la Figure 3 lorsque l'on considère qu'aucune voiture nouvelle n'est introduite, soit la liste illustrée par l'image du bas de la Figure 3 lorsque l'on considère qu'une voiture nouvelle est introduite. **NB :** La fonction `avancer` pourra être librement utilisée dans la suite.

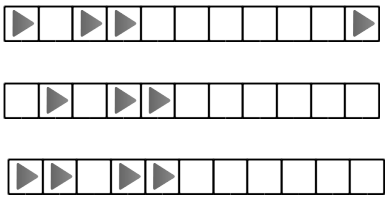
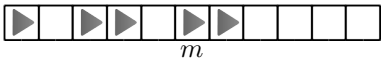
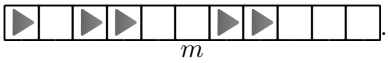


**Figure 3 – Étape de simulation.** En haut : Liste initiale `A`. Au milieu : `B = avancer(A, False)`. En bas : `C = avancer(A, True)`.

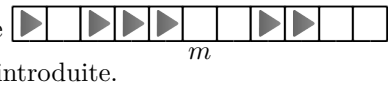
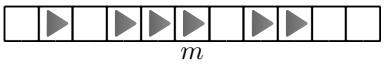
**10.** Étant donnée `A` la liste définie à la question 2, que renvoie `avancer(avancer(A, False), True)` ?

**11.** On considère `L` une liste et `m` l'indice d'une case de cette liste (`0 <= 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 de ne déplaçant pas.

Par exemple, la file  devient  .

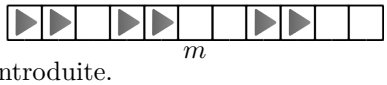
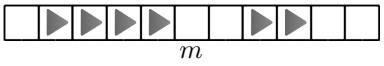
Définir en Python la fonction `avancer_fin(L, m)` qui réalise cette étape partielle de déplacement et renvoie le résultat dans une nouvelle liste sans modifier `L`.

**12.** 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`.

**13.** 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.

## Partie III. Une étape de simulation à deux files

On considère dorénavant deux files `L1` et `L2` de même longueur impaire se croisant en leur milieu ; on note `m` l'indice de la case du milieu. La file `L1` est toujours prioritaire sur la file `L2`. Les voitures ne peuvent pas quitter leur file et la case de croisement ne peut être occupée que par une seule voiture. Les voitures de la file `L2` ne peuvent accéder au croisement que si une voiture de la file `L1` ne s'apprête pas à y accéder. Une *étape de simulation à deux files* se déroule en deux temps. Dans un premier temps, on déplace toutes les voitures situées sur le croisement ou après. Dans un second temps, les voitures situées avant le croisement sont déplacées en respectant la priorité. Par exemple, partant d'une configuration donnée par la Figure 4a), les configurations successives sont données par les Figures 4b), 4c), 4d), 4e) et 4f) en considérant qu'*aucune nouvelle voiture n'est introduite.*

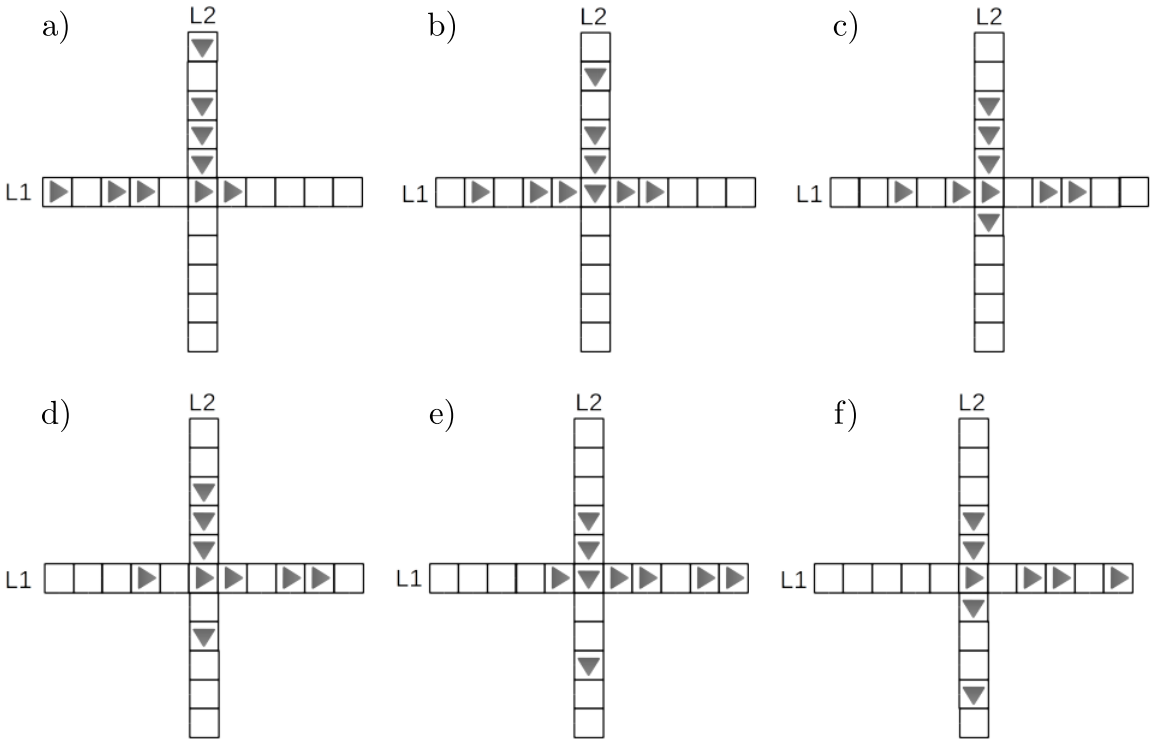



**Figure 4 – Étapes de simulation à deux files.**

L'objectif de cette partie est de définir en Python l'algorithme permettant d'effectuer une étape de simulation pour ce système à deux files.

**14.** En utilisant le langage Python, définir la fonction `avancer_files(L1, b1, L2, b2)` qui renvoie le résultat d'une étape de simulation sous la forme d'une liste de deux éléments notée `[R1, R2]` sans changer les listes `L1` et `L2`. Les booléens `b1` et `b2` indiquent respectivement si une nouvelle voiture est introduite dans les files `L1` et `L2`. Les listes `R1` et `R2` correspondent aux listes après déplacement.

**15.** On considère les listes

```
D = [False, True, False, True, False], E = [False, True, True, False, False]
```

Que renvoie l'appel `avancer_files(D, False, E, False)` ?

## Partie IV. Transitions

**16.** En considérant que de nouvelles voitures peuvent être introduites sur les premières cases des files lors d'une étape de simulation, décrire une situation où une voiture de la file `L2` serait indéfiniment bloquée.

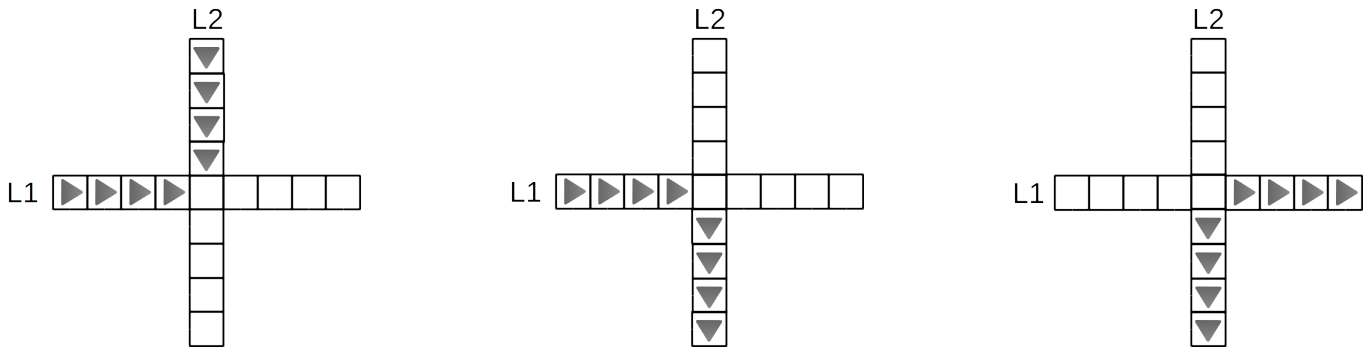



**Figure 5 – Étude de configurations.**

**17.** Étant données les configurations illustrées par la Figure 5, combien d'étapes sont nécessaires (on demande le nombre minimum) pour passer de la configuration de gauche à celle du milieu ? Justifier votre réponse.

**18.** Peut-on passer de la configuration de gauche à celle de droite ? Justifier votre réponse.

## Partie V. Atteignabilité

Certaines configurations peuvent être néfastes pour la fluidité du trafic. Une fois ces configurations identifiées, il est intéressant de savoir si elles peuvent apparaître. Lorsque c'est le cas, on dit qu'une telle configuration est *atteignable*. Pour savoir si une configuration est atteignable à partir d'une configuration initiale, on a écrit le code *incomplet* donné en annexe.

Le langage Python sait comparer deux listes de booléens à l'aide de l'opérateur usuel `<`, on peut ainsi utiliser la méthode `sort` pour trier une liste de listes de booléens.

**19.** Écrire en langage Python une fonction `elim_double(L)` non récursive, qui élimine les éléments apparaissant plusieurs fois dans une liste triée `L` et renvoie la liste triée obtenue.
Par exemple l'appel `elim_double([1, 1, 3, 3, 3, 7])` doit renvoyer la liste `[1, 3, 7]`.

**20.** On dispose de la fonction suivante :

```
1 def doublons(liste):
2     if len(liste) > 1:
3         if liste[0] != liste[1]:
4             return([liste[0]] + doublons(liste[1:]))
5         del liste[1]
6         return(doublons(liste))
7     else:
8         return(liste)
```

Que retourne l'appel `doublons([1, 1, 2, 2, 3, 3, 3, 5])` ?

**21.** Cette fonction est-elle utilisable pour éliminer les éléments apparaissant plusieurs fois dans une liste *non* triée ? Justifier.

**22.** La fonction `recherche` donnée en annexe permet d'établir si la configuration correspondant à `but` est atteignable en partant de l'état `init`. Préciser le type de retour de la fonction `recherche`, le type des variables `but` et `espace`, ainsi que le type de retour de la fonction `successeurs`.

**23.** Afin d'améliorer l'efficacité du test `if but in espace`, ligne 10 de l'annexe, on propose de le remplacer par `if in1(but, espace)` ou bien par `if in2(but, espace)`, avec `in1` et `in2` deux fonctions définies ci-dessous. On considère que le paramètre `liste` est une liste triée par ordre croissant.

```
1  def in1(element, liste):
2      a = 0
3      b = len(liste)-1
4      while a <= b and element >= liste[a]:
5          if element == liste[a]:
6              return True
7          else:
8              a = a + 1
9      return False
```

```
1  def in2(element, liste):
2      a = 0
3      b = len(liste)-1
4      while a < b:
5          pivot = (a+b) // 2 # l'opérateur // est la division entière
6          if liste[pivot] < element:
7              a = pivot + 1
8          else:
9              b = pivot
10     if element == liste[a]:
11         return True
12     else:
13         return False
```

Quel est le meilleur choix ? Justifier.

**24.** Montrer qu'un appel à la fonction `recherche` de l'annexe se termine toujours.

**25.** Compléter la fonction `recherche` pour qu'elle indique le nombre minimum d'étapes à faire pour passer de `init` à `but` lorsque cela est possible. Justifier la réponse.

***Uniquement s'il vous reste du temps :***

**26.** Écrire la fonction `avancer` décrite dans la partie II.

# Annexe

```python
def recherche(but, init):
    espace = [init]
    stop = False
    while not stop:
        ancien = espace
        espace = espace + successeurs(espace)
        espace.sort() # permet de trier espace par ordre croissant
        espace = elim_double(espace)
        stop = egal(ancien, espace) # fonction définie à la question 5
        if but in espace:
            return True
    return False

def successeurs(L):
    res = []
    for x in L:
        L1 = x[0]
        L2 = x[1]
        res.append( avancer_files(L1, False, L2, False) )
        res.append( avancer_files(L1, False, L2, True) )
        res.append( avancer_files(L1, True, L2, False) )
        res.append( avancer_files(L1, True, L2, True) )
    return res

# dans une liste triée, elim_double enlève les éléments apparaissant
# plus d'une fois
# exemple : elim_double([1, 1, 2, 3, 3]) renvoie [1, 2, 3]
def elim_double(L):
  # code à compléter

# exemple d'utilisation
# début et fin sont des listes composées de deux files
# de même longueur impaire,
# la première étant prioritaire par rapport à la seconde
debut = [5 * [False], 5 * [False]]
fin = [3 * [False] + 2 * [True], 3 * [False]+ 2 * [True]]
print(recherche(fin, debut))
```

## • • • Quelques opérations et fonctions Python utiles • • •

**Fonctions :**

- `range(n)` renvoie la séquence des $n$ premiers entiers $(0 \to n-1)$ :
  `list(range(5))` -> `[0, 1, 2, 3, 4]` ; `list(range(0))` -> `[ ]`

**Opérations sur les listes :**

- `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` -> `3` ; `len([[1,2], [3,4]])` -> `2`
- `L[::-1]` construit une liste constituée des éléments de `L` dans l'ordre inverse :
  `[1,2,3][::-1]` -> `[3,2,1]`
- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u`
- `del u[i]` supprime de la liste `u` son élément d'indice `i`
- `u[i], u[j] = u[j], u[i]` permute les éléments d'indice `i` et `j` dans la liste `u`