

## DS1 – Étude du trafic routier

*Durée 3h. Calculatrice interdite.*

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

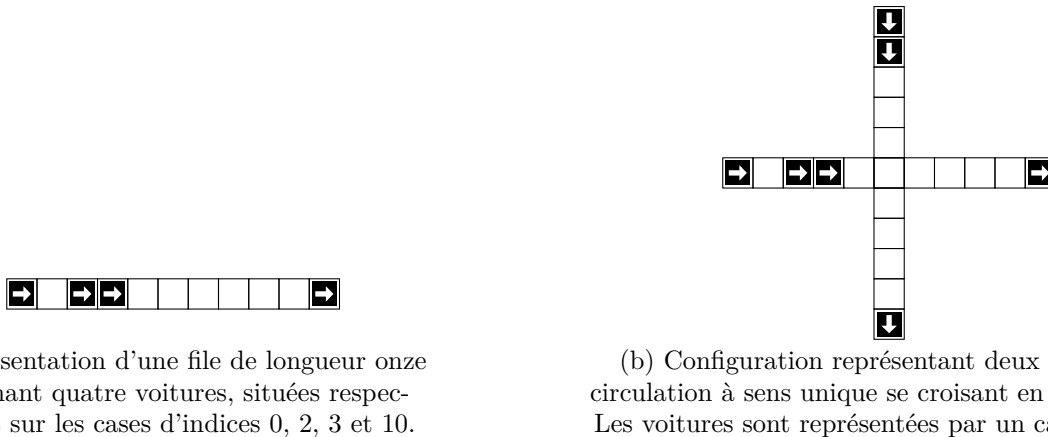


FIGURE 1 – Files de circulation.

## 1 Préliminaires

Dans un premier temps, on considère le cas d'une seule file, illustré par la Figure 1a). 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 1a) et sont indifférenciées.

**Q1** – Expliquer comment représenter une file de voitures à l'aide d'une liste de booléens.

**Q2** – Donner une ou plusieurs instructions **Python** permettant de définir une liste  $A$  représentant la file de voitures illustrée par la Figure 1a).

**Q3** – 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  $\text{occupe}(L, i)$  qui renvoie **True** lorsque la case d'indice  $i$  de la file est occupée par une voiture et **False** sinon.

**Q4** – Combien existe-t-il de files différentes de longueur  $n$ ? Justifier votre réponse.

**Q5** – Écrire une fonction  $\text{egal}(L1, L2)$  retournant un booléen permettant de savoir si deux listes  $L1$  et  $L2$  sont égales.

**Q6** – Que peut-on dire de la complexité de cette fonction ?

**Q7** – 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 les schémas de la Figure 2 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(L, occ)` prenant en paramètres une liste de départ `L`, un booléen `occ` 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.

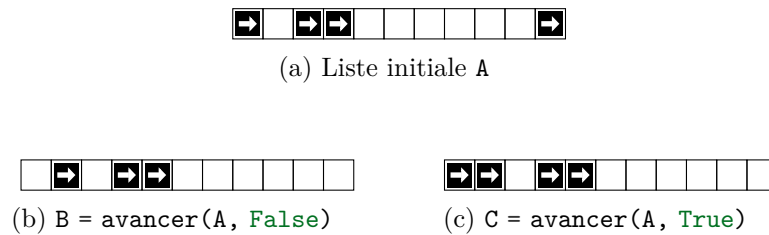
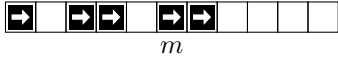
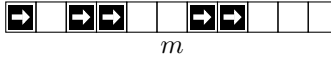


FIGURE 2 – Étape de simulation

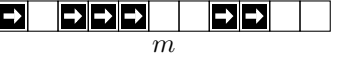

**Q8** – Étant donnée `A` la liste définie à la question 2, que renvoie `avancer(avancer(A, False), True)` ?

**Q9** – On considère `L` une liste et `m` l'indice d'une case de cette liste ( $0 \leq m < \text{len}(L)$ ). 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    
 $m$   $m$

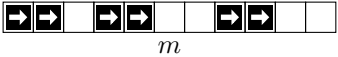
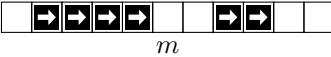
Définir en Python la fonction `avancer_fin(L, m)` qui réalise cette étape partielle de déplacement et renvoie le résultat dans une nouvelle liste sans modifier `L`.

**Q10** – 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.   
 $m$   $m$

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`.

**Q11** – 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.   
 $m$   $m$

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.

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 3(a), les configurations successives sont données par les Figures 3(b), 3(c), 3(d), 3(e) et 3(f), en considérant qu'aucune nouvelle voiture n'est introduite.

### 3 Une étape 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.

**Q12** – 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.

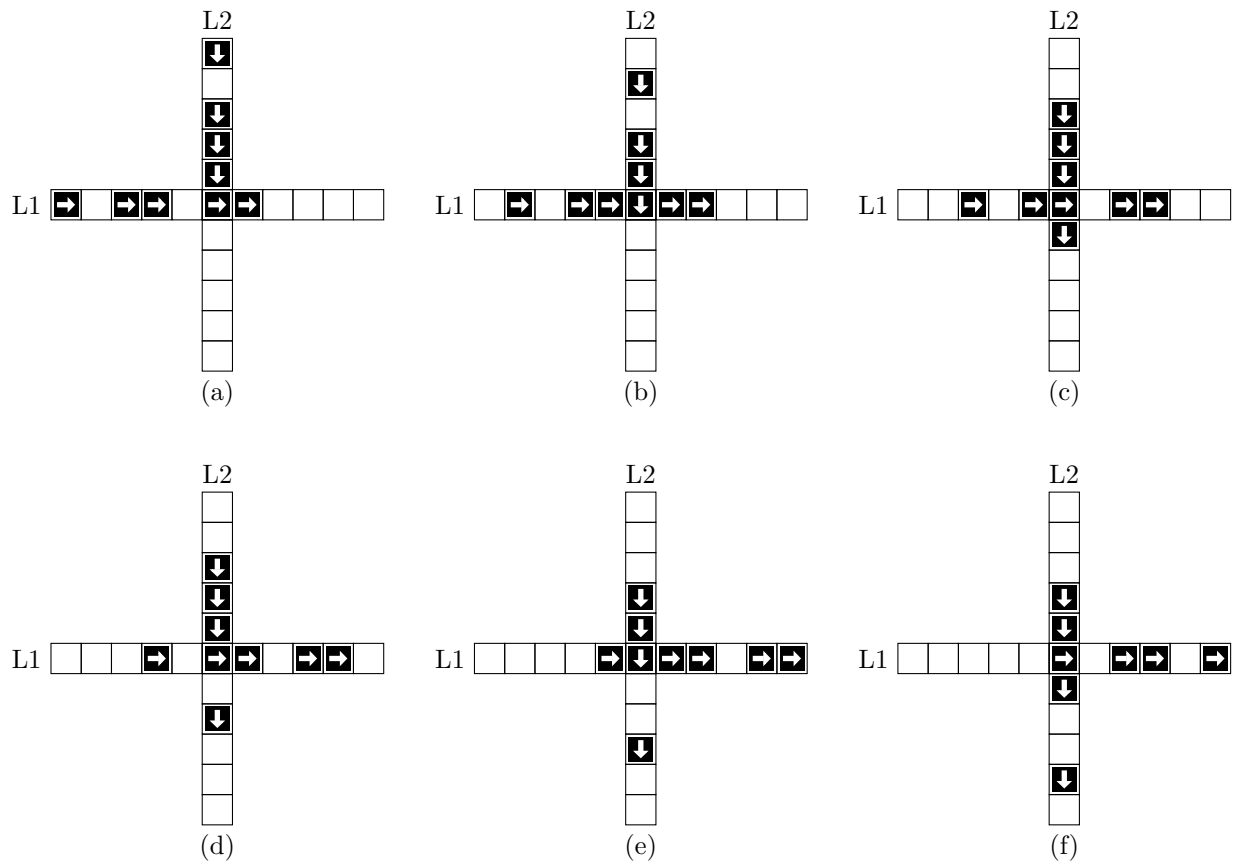


FIGURE 3 – Étapes de simulation à deux files

**Q13** – On considère les listes

$D = [\text{False}, \text{True}, \text{False}, \text{True}, \text{False}]$ ,  $E = [\text{False}, \text{True}, \text{True}, \text{False}, \text{False}]$

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

## 4 Transitions

**Q14** – 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.

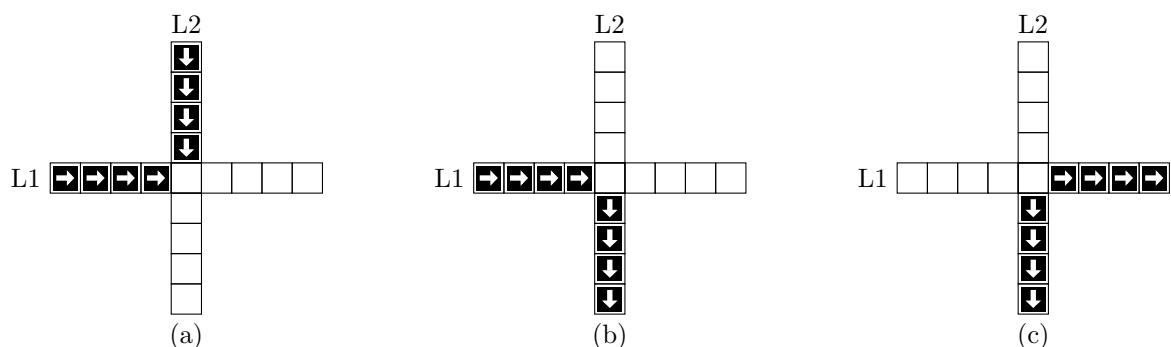


FIGURE 4 – Étude de configurations

**Q15** – Étant données les configurations illustrées par la Figure 4, combien d'étapes sont nécessaires (on demande le nombre minimum) pour passer de la configuration 4(a) à la configuration 4(b) ? Justifier votre réponse.

**Q16** – Peut-on passer de la configuration 4(a) à la configuration 4(c) ? Justifier votre réponse.

## 5 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.

**Q17** – Écrire en langage Python une fonction `elim_double(L)`, de complexité linéaire en la taille de `L`, qui élimine les éléments apparaissant plusieurs fois dans une liste triée `L` et renvoie la liste triée obtenue.

Par exemple `elim_double([1, 1, 3, 3, 3, 7])` doit renvoyer la liste `[1, 3, 7]`.

**Q18** – 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`.

**Q19** – 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.

Quel est le meilleur choix ? Justifier.

```
def in1(element, liste):
    a = 0
    b = len(liste)-1
    while a <= b and element >= liste[a]:
        if element == liste[a]:
            return True
        else:
            a = a + 1
    return False
```

```
def in2(element, liste):
    a = 0
    b = len(liste)-1
    while a < b:
        pivot = (a+b) // 2 # l'opérateur // est la division entière
        if liste[pivot] < element:
            a = pivot + 1
        else:
            b = pivot
    if element == liste[a]:
        return True
    else:
        return False
```

**Q20** – Afin de comparer plus efficacement les files représentées par des listes de booléens on remarque que ces listes représentent un codage binaire où `True` correspond à 1 et `False` à 0.

Écrire la fonction `versEntier(L)` prenant une liste de booléens en paramètre et renvoyant l'entier correspondant. Par exemple, l'appel `versEntier([True, False, False])` renverra 4.

**Q21** – On veut écrire la fonction inverse de `versEntier`, transformant un entier en une liste de booléens. Que doit être au minimum la valeur de taille pour que le codage obtenu soit satisfaisant ? On suppose que la valeur de taille est suffisante. Quelle condition booléenne faut-il écrire en ligne 4 du code ci-dessous ?

```
1 def versFile(n, taille):
2     res = taille * [False]
3     i = taille - 1
4     while ...:
5         if (n % 2) != 0: # % est le reste de la division entière
6             res[i] = True
7             n = n // 2 # // est la division entière
8             i = i - 1
9     return res
```

**Q22** – Montrer qu’un appel à la fonction **recherche** de l’annexe se termine toujours.

**Q23** – 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.

## 6 Base de données

On modélise ici un réseau routier par un ensemble de croisements et de voies reliant ces croisements. Les voies partent d’un croisement et arrivent à un autre croisement. Ainsi, pour modéliser une route à double sens, on utilise deux voies circulant en sens opposés. La base de données du réseau routier est constituée des relations suivantes :

- Croisement(id, longitude, latitude)
- Voie(id, longueur, id\_croisement\_debut, id\_croisement\_fin)

Dans la suite on considère  $c$  l’identifiant (id) d’un croisement donné.

**Q24** – Écrire la requête SQL qui renvoie les identifiants des croisements atteignables en utilisant une seule voie à partir du croisement ayant l’identifiant  $c$ .

**Q25** – Écrire la requête SQL qui renvoie les longitudes et latitudes des croisements atteignables en utilisant une seule voie, à partir du croisement  $c$ .

**Q26** – Écrire une requête SQL qui renvoie le nombre de voies qui sont reliées au croisement  $c$ .

**Q27** – Que renvoie la requête SQL suivante ?

```
SELECT V2.id_croisement_fin
FROM Voie as V1
JOIN Voie as V2
ON V1.id_croisement_fin = V2.id_croisement_debut
WHERE V1.id_croisement_debut = c
```

## Annexe

```
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
# debut 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))
```