Exercice I : *CCP 2017 MP*

On peut proposer les solutions suivantes :

1.
```
A = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
```

2.
```
len(A) = 4
A[1] = [4,5,6]
A[2][1] = 8
```

3. Classiquement ou par compréhension
```
def difference(x,y):
    result = []
    for i in range(len(x)):
        result.append( x[i] - y[i]
        ↪  )
    return result
```
```
def difference(x,y):
    return [ x[i]-y[i] for i in range(len(x)) ]
```

4. Classiquement ou par compréhension
```
def norme(x):
    result = abs(x[0])
    for i in range(1,len(x)):
        if abs(x[i]) > result:
            result = abs(x[i])
    return result
```
```
def norme(x):
    return max( [ abs(xi) for xi in x ] )
```

5. Classiquement ou par compréhension
```
def itere(x,A):
    p = len(x)
    result = []
    for i in range(p):
        y = 0
        for j in range(p):
            y = y + x[j]*A[j][i]
        result.append(y)
    return result
```
```
def itere(x,A):
    p = len(x)
    return [ sum( [x[j]*A[j][i] for j in range(p)] )
    ↪  for i in range(p) ]
```

6.
```
def probainvariante(A,eps):
    p = len(A)
    mu0 =[ 1/p ]*p
    mu1 = itere( mu0 , A )
    while norme ( difference ( mu1 , mu0 ) ) > eps :
        mu0 = mu1
        mu1 = itere( mu0 , A )
    return mu1
```

Exercice II : *Mines 2017*

À chaque question d'implémentation, plusieurs codes sont possibles.

Le sujet rappelant les notations `L[i:j]` et `3*[0]`, on aura tendance à privilégier ces approches. Par ailleurs, on remarque que `L[:]` effectue une copie de la liste (ce que ne fait pas `A=L`).

1 Préliminaires

☐ **Q1 -** À l'aide d'une liste de la longueur de la file de voitures, on code True s'il existe une voiture à la position i, False sinon.

☐ **Q2 -**
```
A=[True,False]+2*[True]+6*[False]+[True]
```

☐ **Q3 -**
```
def occupe(L,i):
    return L[i]
```

☐ **Q4 -** Il existe 2^n listes possibles (un raisonnement par récurrence nous permettrait de le montrer proprement mais la question n'invite pas à une preuve exhaustive).

Succinctement cela donnerait : pour une case il y a 2 choix possibles. Ensuite, à chaque case ajoutée on multiplie par 2 (True ou False) le nombre de listes, soit 2^n pour une liste de longueur n.

☐ **Q5 -**
```
def egal(L1,L2):
    if len(L1)==len(L2):
        for i in range(len(L1)):
            if L1[i]!=L2[i] :
                return False
        return True
    return False
```

```
def egal2(L1,L2): # 2e possibilite (moins lisible pour la complexite)
    return L1==L2
```

Dans les deux cas, la complexité temporelle est

- au mieux un $\mathcal{O}(1)$ lorsque les listes sont de longueurs distinctes,
- au pire un $\mathcal{O}(n)$ (avec n la lonqueur des deux listes) lorsque les listes sont égales.

2 Déplacement de voitures dans la file

☐ **Q6 -**
```
def avancer(L,b):
    return [b]+L[:-1] # equivaut a : return [b]+L[0:len(L)-1]
```

```
def avancer2(L,b):      # solution avec une boucle
    M=[b]
    for i in range(len(L)-1):
        M.append(L[i])
    return M
```

☐ **Q7 -** Avec la liste `A` déjà définie, l'appel `avancer(avancer(A,False),True)` renvoie `[True, False, True, False, True, True, False, False, False, False, False]`.

☐ **Q8 -**
```
def avancer_fin(L,m):
    return L[:m]+[False]+L[m :-1]
```

☐ **Q9 -**
```
def avancer_debut(L,b,m):
    return [b]+L[:m]+L[m+1:]
```

☐ **Q10 -**
```
def avancer_debut_bloque(L,b,m):  # fonction recursive
    if m==0 :
        return L[:]                # renvoie une copie de L
    if occupe(L,m-1) :             # initialement L[m] est occupee
        return avancer_debut_bloque(L,b,m-1)
    else :
        return avancer_debut(L,b,m-1)
```

```
def avancer_debut_bloqueI(L,b,m): # fonction iterative
    i=m-1                         # L[m] est occupee
    while i>=0 and occupe(L,i):
        i=i-1
    if i<0:
        return L[:]                # renvoie une copie de L
    else :
        return avancer_debut(L,b,i)
```

3 Une étape de simulation à deux files

☐ **Q11 -**

```
def avancer_files(L1,b1,L2,b2):
    R1=avancer(L1,b1) # la file L1 est prioritaire
    m=len(R1)//2      # (len(L1)+1)/2
    if occupe(R1,m) :
        R2=avancer_debut_bloque(L2,b2,m)
        R2=avancer_fin(R2,m)
    else :
        R2=avancer(L2,b2)
    return [R1,R2]
```

☐ **Q12 -** L'appel `avancer_files(D, False, E, False)` renvoie `[[False, False, True, False, True],[False, True, False, True, False]]`

4 Transitions

☐ **Q13 -** Puisque la file 1 est prioritaire sur la file 2, L2 sera bloquée si à chaque étape une voiture est ajoutée en début de liste L1.

☐ **Q14 -** Il faudra 9 étapes : 4 étapes pour que la 4ème voiture de L1 soit au croisement (libérant les voitures L2 pour le tour d'après), 4 étapes pour que la 4ème voiture de L2 soit au croisement, 1 étape pour libérer le croisement. Afin d'avoir les 4 nouvelles voitures sur L1, on ajoute à chaque étape à partir de la 6ème une voiture en L1.

☐ **Q15 -** On ne peut pas passer de 4a) à 4c) car L1 est prioritaire sur L2. En effet pour avoir 4 voitures de L2 en cette position, il faut au moins 4 étapes durant lesquelles il n'y a pas de voiture sur L1 au croisement. Pour se faire la liste L1 à droite du croisement serait nécessairement devenue vide. Ce n'est pas le cas ↯.

5 Atteignabilité

☐ **Q16 -**

```
def elim_double(L):
    M=[L[0]]          # L est supposee non vide
    i=1
    while i<len(L):
        if L[i]!=M[-1]:
            M.append(L[i])
        i=i+1
    return M
```

Attention si on utilise l'instruction `del(L[i])` il faudra utiliser une boucle `while` puisque la longueur de la liste évolue au fur et à mesure des tours de boucle. Toutefois, quelle est la complexité de la fonction `del()` ?

☐ **Q17 -** L'appel `doublons([1, 1, 2, 2, 3, 3, 3, 5])` renvoie la liste `[1, 2, 3, 5]`.

☐ **Q18 -** Elle n'est pas utilisable si `L` n'est pas triée. Comme contre-exemple, on peut considérer la liste [1,2,1].

☐ **Q19 -** La fonction `recherche()` renvoie un booléen. La variable `but` est un couple de listes. La variable `espace` est une liste de couple de 2 listes (tout comme ce que renvoie successeurs). Enfin la fonction `successeurs()` renvoie un arbre quadrinaire de couple de 2 listes, écrit sous forme d'une liste.

☐ **Q20 -** La fonction `in1()` effectue une recherche linéaire en $\mathcal{O}(n)$ tandis que `in2()` effectue une recherche dichotomique en $\mathcal{O}(ln(n))$. On préfèrera donc utiliser la fonction `in2()`.

☐ **Q21 -**

```
def versEntier(L):
    s=0
    for i in range(len(L)):
        if L[i]:
            s=2*s+1
        else :
            s=2*s
    return s
```

☐ **Q22 -** Pour que le codage de n soit consistant, il faut que la longueur de la liste soit strictement supérieure à la partie entière du logarithme en base 2 de n, *i.e.* $len(L) = 1 + E[log_2(n)]$. On peut alors compléter par :

```
def versFile(n, taille):
    res = taille * [False]
    i = taille - 1
    while i>=0 :          # ou de maniere equivalente n>0
        if (n % 2) != 0:  # % est le reste de la division entiere
            res[i] = True
        n = n // 2        # // est la division entiere
        i = i - 1
    return res
```

☐ **Q23 -** Les deux listes ne contiennent que des booléens et possèdent un total de 21 cases, ainsi il faudra parcourir au maximum 2^{21} possibilités.

La fonction `successeurs()` permet d'ajouter 4 cas à chaque appel et `elim_double()` élimine les cas déjà parcourus. Ainsi `espace` est une suite d'états parcourus croissante (pour l'inclusion) et majorée (par les 2^{21} possibilités) : cette liste converge nécessairement. Au pire (lorsque le but n'est pas atteignable), on s'arrête lorsque ancien=espace (*i.e.* lorsque la suite `espace` devient stationnaire). Ainsi la fonction `recherche()` se termine toujours.

☐ **Q24 -** On complétera ainsi (on utilise également la fonction `in2()` de la Q-20) :

```
def recherche(but, init):
    espace = [init]
    etape = 0                # initialisation du nombre d'etape
    stop = False
    while not stop:
        ancien = espace
        espace = espace + successeurs(espace)
        espace.sort()        # permet de trier espace par ordre croissant
        espace = elim_double(espace)
        etape += 1           # incrementation
        stop = egal(ancien,espace) # fonction definie a la question 5
        if in2(but,espace):        # (ligne 10) - question Q25
            return True, etape     # on renvoie le nombre d'etapes
    return False
```

Le nombre d'étapes augmente de 1 à chaque itération de boucle. En effet, chaque situation en double est effacée par `elim_double()` ce qui permet de s'assurer le minimum pour atteindre la-dite situation (jusqu'à ce que l'on atteigne 'but' ou que l'on sorte).

6 Base de données

☐ **Q25 -** On propose

```
1 SELECT id_croisement_fin
2 FROM Voie
3 WHERE id_croisement_debut = idc
```

☐ **Q26 -** On modifie la requête précédente en ajoutant une jointure

```
1 SELECT longitude, latitude
2 FROM Croisement as C
3 JOIN Voie as V
4 ON C.id = V.id_croisement_fin
5 WHERE V.id_croisement_debut = idc
```

☐ **Q27 -** La requête SQL suivante renvoie les identifiants des croisements atteignables en utilisant exactement deux voies à partir du croisement ayant l'identifiant *idc*.