INFORMATIQUE

D'après Mines - Ponts 2017.
Rédigé par Jérémy Larochette.

On utilise Python 3.

A. Préliminaires

❑ **Q1** – Une file de voiture peut être représentée par une liste de booléen : l'élément d'indice i contient `True` si et seulement si la case d'indice i contient une voiture.

❑ **Q2** – `A = [True, False, True, True] + 6 * [False] + [True]`

❑ **Q3** –

```
def occupe(L, i):
    """renvoie True si la case d'indice i de la file représentée par L contient une
voiture, False sinon."""
    return L[i]
```

❑ **Q4** – Chacune des n cases contenant ou non une voiture, il y a exactement 2^n files différentes de longueur n.

❑ **Q5** –

Peut-on se contenter de :

```
def egal(L1, L2):
    "teste si L1 et L2 sont égales"
    return L1 == L2
```

```
def egal(L1, L2):
    "teste si L1 et L2 sont égales"
    if len(L1) != len(L2):
        return False
    for i in range(len(L1)):
        if L1[i] != L2[i]:
            return False
    return True
```

Les questions suivantes laissent penser que non.

❑ **Q6** – La complexité est en $O(1)$ si les listes ne sont pas de la même longueur et en $O(n)$ si elles sont toutes deux de longueur n (le cas le pire étant atteint lorsque les listes sont égales).

❑ **Q7** – Cette fonction renvoie un booléen de type `bool`.

B. Déplacement de voitures dans la file

❑ **Q8** – `avancer(avancer(A, False), True)` renvoie

`[True, False, True, False, True, True, False, False, False, False, False]`.

❑ **Q9** –

Version avancer :

```
def avancer_fin(L, m):
    return L[:m] + avancer(L[m:], False)
```

Version itérative :

```
def avancer_fin(L, m):
    n = len(L)
    L1 = L[:] # Copie de L
    for i in range(n - 1, m, -1):
        L1[i] = L1[i - 1]
    L1[m] = False
    return L1
```

Version récursive :

```
def avancer_fin(L, m):
    if m == len(L) - 1:
        L1 = L[:] # Copie de L
        L1[-1] = False
        return L1
    else:
        # Déplacement à partir de m + 1
        L1 = avancer_fin(L, m + 1)
        # Déplacement de m
        L1[m + 1] = L1[m]
        L1[m] = False
        return L1
```

❑ **Q10** –

Version avancer :

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

Version itérative :

```
def avancer_debut(L, b, m):
    n = len(L)
    L1 = L[:] # Copie de L
    for i in range(m, 0, -1):
        L1[i] = L1[i - 1]
    L1[0] = b
    return L1
```

Version récursive :

```
def avancer_debut(L, b, m):
    if m == 1
        L1 = L[:] # Copie de L
        L1[1] = L1[0]
        L1[0] = b
        return L1
    else:
        # Déplacement de m - 1
        L1[m] = L1[m - 1]
        # Déplacement jusqu'à m - 1
        L1 = avancer_debut(L, b, m - 1)
        return L1
```

❑ **Q11** –

```
def avancer_debut_bloque(L, b, m):
    L1 = L[:]
    for i in range(m - 1, 0, -1):
        if not occupe(L1, i) \
           and occupe(L1, i - 1):
            # Voiture en i-1 mais pas en i
            L1[i] = True
            L1[i - 1] = False
    L1[0] = b or L1[0]
    return L1
```

```
def avancer_debut_bloque(L, b, m):
    for i in range(m - 1, -1, -1):
        if not occupe(L, i):
            return avancer_debut(L, b, i)
    return L[:]
```

C. Une étape de simulation à deux files

❑ **Q12** –

```
def avancer_files(L1, b1, L2, b2):
    m = len(L1) // 2
    R1 = avancer(L1, b1) # Dans tous les cas, L1 avance normalement
    R2 = avancer_fin(L2, m)
    if R1[m]: # une voiture bloque le croisement
        R2 = avancer_debut_bloque(R2, b2, m)
    else:
        R2 = avancer_debut(R2, b2, m)
    return [R1, R2]
```

❏ **Q13** – L'appel renvoie `[[False, False, True, False, True], [False, True, False, True, False]]`.

D. Transitions

❏ **Q14** – Si toutes les cases de `L1` sont toujours occupées, les voitures de `L2` avant le croisement sont indéfiniment bloquées.

❏ **Q15** – La file représentée par `L1` avançant normalement, ses 4 premières voitures doivent passer le croisement (5 étapes) et être remplacées par des cases vides, puis les 4 voitures de `L2` qui étaient bloquées vont passer (4 nouvelles étapes) et être remplacées par des cases vides, ces 4 dernières étapes introduisant une voiture dans `L1`. Cela donne donc au minimum 9 étapes.

❏ **Q16** – On ne peut pas arriver à la configuration 4(c) à partir de 4(a) (ni à partir d'aucune autre d'ailleurs), car à l'étape précédente, il faudrait nécessairement que les 4 voitures des deux files soit décalées d'une case à gauche et en haut pour qu'il en reste 4 ensuite, et donc deux voitures occuperaient le croisement ce qui n'est pas possible.

E. Atteignabilité

❏ **Q17** –

```
def elim_double(L):
    "renvoie une liste correspondant à la liste triée L sans répétition."
    if L == []:
        return []
    L1 = [L[0]]
    for i in range(1, len(L)):
        if L[i] != L1[-1]:
            L1.append(L[i])
    return L1
```

❏ **Q18** – L'appel de `doublons([1, 1, 2, 2, 3, 3, 3, 5])` renvoie `[1, 2, 3, 5]`. `doublons` est une version récursive de `elim_double`.

❏ **Q19** – Cette fonction n'est pas utilisable pour éliminer les éléments apparaissant plusieurs fois dans une liste non triée. Par exemple, `doublons([3, 2, 3])` renvoie `[3] + [2] + [3]` soit `[3, 2, 3]`.

❏ **Q20** –

- `recherche` renvoie un booléen.
- `successeur` renvoie une liste de listes de 2 listes de même longueur impaire correspondant à des files.
- `espace` pointe sur une liste de listes de 2 listes de même longueur impaire correspondant à des files.
- `but` pointe sur une liste de 2 listes de même longueur impaire correspondant à des files.

❏ **Q21** – La recherche par `in1` est linéaire tandis que celle par `in2` est dichotomique, donc de complexité logarithmique. La deuxième est donc préférable.

❏ **Q22** – Voici deux versions (directement ou avec schéma de Hörner) :

```
def versEntier(L):
    n = len(L)
    puissance_de_2 = 1 # puissances de 2
    m = 0          # va contenir l'entier
    for i in range(n - 1, -1, -1):
        if L[i]:
            m += puissance_de_2
        puissance_de_2 *= 2
    return m
```

```
def versEntier(L):
    n = len(L)
    m = 0          # va contenir l'entier
    for i in range(n):
        m *= 2
        if L[i]:
            m += 1
    return m
```


❏ **Q23** – `taille` doit au moins être égal au nombre de chiffres de l'écriture binaire de n soit $\lfloor \log_2 n \rfloor + 1$. La condition à écrire est `n > 0`.

❏ **Q24** – Par construction de la boucle, la suite des tailles de `espace` est entière strictement croissante. Or le nombre de files possibles est fini donc elle est bornée. Cette suite est donc finie et la boucle se termine.

❏ **Q25** –

```
def recherche(but, init):
    espace = [init]
    stop = egal(but, init)
    if stop:
        return 0
    nb_etapes = 0 # On initialise le nombre d'étapes
    while not stop:
        ancien = espace
        espace = espace + successeur(espace)
        espace.sort()
        espace = elim_double(espace)
        stop = egal(ancien, espace)
        nb_etapes += 1 # on incrémente le nombre d'étapes
        if but in espace:
            return nb_etapes
    return -1
```

Un invariant d'entrée de boucle est : `espace` contient toutes les configurations possibles, à partir de `init`, en au plus `nb_etapes` étapes, et ne contient pas `but`.

La fonction s'arrête à la première apparition de `but` dans `espace` et renvoie bien le nombre minimal d'étapes pour y arriver (on ne sort de la boucle qu'à la première apparition de `but` dans `espace` s'il y est), et -1 s'il n'y apparaît pas.

F. Base de données

❏ **Q26** –

```
SELECT id_croisement_fin FROM voie
WHERE id_croisement_debut = c;
```

❏ **Q27** –

```
SELECT longitude, latitude FROM croisement cr
JOIN voie ON cr.id = id_croisement_fin
WHERE id_croisement_debut = c;
```

❏ **Q28** – Cette requête renvoie les identifiants des croisements atteignables en utilisant deux voies à partir du croisement ayant l'identifiant c (donc avec un croisement intermédiaire).

G. Simulation dynamique

❏ **Q29** –

```
def Q(t):
    """Renvoie le nombre de véhicules quittant le stationnement par unité de temps à
l'instant t."""
    if 0 < t < 1:
        return db_stat_max * np.exp(1 - 1 / (4 * t * (1 - t)))
    else:
        return 0
```

❑ **Q30** – Avec la formule $\int_a^b f(t)\mathrm{d}t \approx h\sum_{k=0}^{n-1}\frac{f(a_k)+f(a_{k+1})}{2} = h\left(\frac{f(a)+f(b)}{2}+\sum_{k=1}^{n-1}f(a_k)\right)$:

```
def integrale(f, a, b, n):
    """Renvoie l'approximation par la méthode des trapèzes de l'intégrale entre a et b
 de f."""
     h = (b - a) / n
     return h * sum(f(a + k * h) + f(a + (k + 1) * h) for k in range(n)) / 2
```

ou bien

```
def integrale(f, a, b, n):
    """Renvoie l'approximation par la méthode des trapèzes de l'intégrale entre a et b
 de f."""
     h = (b - a) / n
     T = np.linspace(a, b, n + 1)
     return h * ((f(a) + f(b)) / 2 + sum(f(ak) for ak in T[1:-1]))
```

ou encore

```
def integrale(f, a, b, n):
    """Renvoie l'approximation par la
méthode des trapèzes de l'intégrale entre
a et b de f."""
     h = (b - a) / n
     ak = a
     somme = 0
     for k in range(n):
         ak += h
         somme += f(ak) + f(ak + h)
     return h * somme / 2
```

voire

```
def integrale(f, a, b, n):
    """Renvoie l'approximation par la
méthode des trapèzes de l'intégrale entre
a et b de f."""
     h = (b - a) / n
     ak = a
     somme = (f(a) + f(b)) / 2
     for k in range(1, n):
         ak += h
         somme += f(ak)
     return h * somme
```

et bien d'autres versions encore...

❑ **Q31** –

```
def Nb(t, n) =
    """Nombre total de véhicules ayant quitté le stationnement à l'instant t."""
    return integrale(Q, 0, t, n)
```

❑ **Q32** –

```
def V(K):
    """renvoie la vitesse en fonction de l'entier K."""
    Vdiff = (Vmax - Vmin) / 2
    return Vdiff * (1 + np.cos(np.pi * K / Ksat)) + Vmin
```

❑ **Q33** – Le schéma d'Euler, pour $y_k \approx y(t_k)$, est

$$y_{k+1} = y_k + (t_{k+1} - t_k)F(y_k, t_k),$$

soit, avec un pas Δt constant,

$$y_{k+1} = y_k + \Delta_t \cdot F(y_k, t_k).$$

❑ **Q34** –

```
def F(Y, t):
    N, K, S = Y
    return np.array([Q(t), Q(t) - K * V(K), K * V(K)])
```


❑ **Q35** –

- La forme de la courbe de K « en cloche »indique que la concentration de véhicules voulant sortir augmente progressivement, dépassant K_{sat} à environ $t = 0{,}4$, atteint son maximum à $t \approx 0{,}7$ puis décroît, désaturant la sortie à $t \approx 1{,}0$ (donc plus de départ de l'usine) et finit par être nulle (tout le monde est sorti !)
- La courbe du nombre N de véhicules ayant quitté le stationnement est logiquement croissante de 0 jusqu'au nombre total de voitures (120), déjà atteint à $t = 1$, ce qui est cohérent.
- La courbe du nombre S de véhicules sortis du parking est également strictement croissante de 0 à 120, avec un impact sur ses variations lorsque la concentration passe par K_{sat}, ce qui est également cohérent avec le modèle.