DS1 – Étude du trafic routier – corrigé

Partie I. Préliminaires

Q1 – On peut représenter une file de longueur n par une liste L de même longueur. Pour chaque indice i valide, $L[i]$ vaut `True` si la $i^{\text{ème}}$ case de la file est occupée et `False` sinon.

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

Q3 –

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

Q4 – Pour chaque case de la file de longueur n il y a deux possibilités, ce qui donne un total de 2^n files différentes possibles.

Q5 –

```
def egal(L1, L2):
  return L1 == L2
```

Q6 – Pour déterminer si deux listes sont égales, il faut en comparer tous les éléments, la complexité est donc linéaire en la taille des listes à comparer.

Q7 – Cette fonction retourne un booléen.

Partie II. Déplacement de voitures dans la file

Q8 – La fonction renvoie

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

Ce qui correspond à la file suivante :

Q9 –

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

Q10 –

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

Q11 –

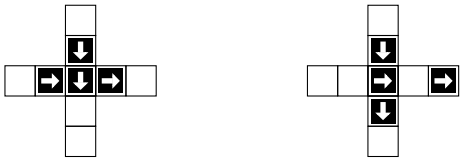
```
def avancer_debut_bloquer(L,b,m):
  i = m-1
  # Recherche le première case non occupé avant la case m
  while L[i] and i>0:
    i-=1
  return avancer_debut(L, b, i)
```

Q12 –

```
def avancer_files(L1, b1, L2, b2):
  m  = len(L1) // 2
  R2 = avancer_fin(L2, m)
  R1 = avancer(L1, b1)
  if occupe(R1,m) :
    R2 = avancer_debut_bloque(R2, b2, m)
  else :
```

```
        R2 = avancer_debut(R2, b2, m)
    return [R1, R2]
```

Q13 – Les files évoluent de la manière suivante :



L'appel renvoie `[[False, False, True, False, True], [False, True, False, True, False]]`

Partie III. Transitions

Q14 – Considérons la situation suivante :

— La file $L1$ est pleine

— à chaque étape de la simulation, on ajoute une nouvelle voiture à $L1$,

— une voiture est sur la case $m-1$ de la file $L2$

La voiture sur la case $m-1$ de la file $L2$ est indéfiniment bloquée.

Q15 – Examinons le temps nécessaire pour que les voitures de la file $L2$ soient dans la position voulue :

— Les voitures de la file $L2$ doivent lausser la priorité aux voitures de la file $L1$, elles restent donc immobiles jusqu'à l'étape 4 incluse.

— Les voitures de la file $L2$ commencent à se déplacer à l'étape 5 et arrivent dans la position voulue à l'étape 9.

Il n'est pas possible d'atteindre la configuration demandée en moins de 9 étapes.

Il suffit d'ajouter des voitures à $L1$ aux étapes 6, 7, 8 et 9 (et de n'ajouter aucune voiture à $L2$) pour obtenir la configuration souhaitée en 9 étapes.

Conclusion : Il faut 9 étapes.

Q16 – Pour obtenir la configuration 4(c) en une étape à partir d'une configuration C, il faut que les voitures en position 6 de la configuration 4(c) soient, dans la configuration C toutes les deux en position 5 dans leur liste, donc toutes les deux sur le croisement, ce qui est impossible.

Conclusion : Il est impossible de passer de la configuration 4(a) à la configuration 4(c)

Q17 – On peut par exemple écrire la fonction suivante :

```
def elim_double(L):
    L2 = [L[0]]
    for k in range(1,len(L)):
        if L[k] != L[k-1]:
            L2.append(L[k])
    return L2
```

Q18 – La variable `but` est une liste deux listes de booléens, la variable `espace` est une liste de listes de booléens. La fonction `recherche` renvoie un booléen et la fonction `successeur` renvoie une liste de listes de listes de booléens.

Q19 – `in1` parcourt la liste élément par élément, d'où une complexité $O(n)$ (n est la longueur de la liste)

`in2` fait une recherche dichotomique, sa complexité est donc $O(\ln(n))$ (voir cours).

Le meilleur choix est donc évidemment `in2`.

Q20 – On peut écrire la fonction suivante :

```
def versEntier(L):
    n=0
    for b in L:
        n *= 2
        if b:
            n += 1
    return n
```

Q21 –

— `taille` doit valoir au moins le nombre de chiffres de n en base 2. C'est à dire $\lfloor \log_2(n) \rfloor + 1$

— Si `taille` est suffisante, alors il suffit d'écrire dans la condition du `while` : `n!=0`

Q22 – Considérons k le nombre de configurations qui ne sont pas dans `espace`. On remarque que k est un variant de boucle (la suite des valeurs de k est une suite d'entiers naturels strictement décroissante), ainsi la boucle ne peut pas être infinie et se termine.

Q23 – On ajoute entre les lignes 1 et 2 les lignes suivantes :

```
      if egal(init, but):
        return 0
      n = 0
```

On ajoute entre les lignes 9 et 10 la ligne : `n = n+1`
Le `return True` de la ligne 11 est remplacé par `return n`
La fonction modifiée est correcte car l'invariant de boucle suivant est satisfait : « `espace` contient toutes les configurations atteignables en n étapes ou moins ».

Partie IV. Base de données

Q24 – `SELECT id_croisement_fin FROM Voie WHERE id_croisement_debut = c`

Q25 –

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

Q26 –

```
SELECT COUNT(*) FROM Voie WHERE id_croisement_debut=c OR id_croisement_fin=c;
```

Q27 – Cette requête renvoie les identifiants des croisements atteignables en utilisant exactement deux voies à partir du croisement c.