Mines 2017 : corrigé

MP Lycée Camille Jullian

15 janvier 2022

I.Préliminaires.

Q1 : Il suffit de créer une liste de booléen dont la longueur correspond à celle de la file qu'on souhaite représenter, et de considérer que True représente un emplacement où se situe une voiture, et False un emplacement vide.

Q2 : A=[True,False,True,True,False,False,False,False,False,False,True]

Q3 : La liste étant par hypothèse constituée de booléens, pas besoin de se fatiguer, on renvoie simplement ce que contient la case i :

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

Q4 : Il en existe exactement 2^n (deux possibilités pour chaque élément).

Q5 : Encore une question extrêmement difficile :

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

Les plus courageux feront une boucle pour vérifier élément par élément mais ce n'était pas demandé par l'énoncé.

Q6 : Même si on a (un peu) triché en écrivant un seul test dans notre fonction, elle est tout de même de complexité linéaire, le test L1==L2 nécessitant en pratique de tester un par un l'égalité des éléments des deux listes.

Q7 : Là, on atteint vraiment des sommets puisque la réponse est écrite noir sur blanc dans l'énoncé, la fonction renvoie un booléen.

II. Déplacement de voitures dans la file.

Q8 : Elle renvoie [True, False, True, False, True, True, False, False, False, False]

Q9 : Puisqu'on a déjà sous la main un programme permettant de faire avancer une liste, il suffit de l'appliquer à la fin de notre file (sans ajouter de voiture, bien entendu) sans toucher au début :

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

Q10 : C'est quasiment la même chose que précédemment mais cette fois on ne fait avancer que le début (la case m étant inoccupée, aucun risque de collision entre le début de la liste et la fin, on fait simplement attention à inclure la case m dans la procédure d'avancement) :

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

Q11 : Puisqu'on doit renvoyer le résultat dans une nouvelle liste, on va commencer par recopier la liste L puis déplacer les voitures à gauche de la case m si elles ne sont pas bloquées, en effectuant

une boucle descendante à partir de la case numéro $m-2$ (par hypothèse, la voiture éventuellement présente en case $m-1$ sera bloquée par celle en case m, pas besoin de la considérer) :

```
def avancer_debut_bloque(L,b,m) :
    liste=L[ :]
    for i in range(m-2,-1,-1) :
        if not liste[i+1] :
            liste[i+1]=liste[i]
            liste[i]=False
    liste[0]=b
    return liste
```

III. Une étape de simulation à deux files.

Q12 : On fait avancer la file $L1$ normalement puisqu'elle ne peut pas être bloquée, et on fait avancer $L2$ en deux temps : d'abord la fin, puis le début en tenant compte d'un bloquage éventuel :

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

Q13 : L'avancée de la première liste va placer une voiture bloquante en position médiane (mais la voiture en position médiane dans la liste $L2$ proposée va bien sûr avancer, par contre celle qui la précède sera bloquée) , on obtiendra à l'arrivée $[[False, False, True, False, True], [False, True, False, True, False]]$.

IV. Transitions.

Q14 : Il suffit de créer une file $L1$ pleine de voitures et d'ajouter systématiquement une voiture dans la file $L1$ à chaque nouvelle étape, la voiture située juste avant la case m dans la liste $L2$ ne pourra jamais avancer.

Q15 : On est obligés de commencer par faire avancer les voitures de la file $L1$, ce qui nécessite quatre étapes. Seulement ensuite, on pourra faire avancer les voitures de la file $L2$ jusqu'à la position finale souhaitée, ce qui nécessitera 5 étapes de plus (bien sûr on n'aura pas introduit de voitures dans la file $L1$ lors des premières étapes pour ne pas bloquer à nouveau $L2$. Il faudra penser à ajouter des voitures dans $L1$ aux quatre dernières étapes, et on aura la configuration souhaitée en 9 étapes.

Q16 : Non, c'est impossible, car en « remontant le temps », les quatre étapes précédent la situation $4(c)$ auraient placé une voiture bloquante dans la file $L1$, donc la file $L2$ aurait du rester immobile depuis quatre étapes, ce qui est complètement contradictoire avec la position des quatre voitures de la file $L2$ dans la situation $4(c)$.

V. Atteignabilité.

Q17 : Il suffit de parcourir la liste L une seule fois (d'où la complexité linéaire) et de ne conserver que les éléments différents de leur prédécesseur dans la liste L (et bien sûr le premier élément de la liste) :

```
def elim_double(L) :
    return [L[0]]+[L[i] for i in range(1,len(L)) if L[i] !=L[i-1]]
```

Q18 : Il renvoie $[1, 2, 3, 5]$ (en effet, on commence par tester si les deux premiers éléments de la liste sont les mêmes, ce qui est le cas ici. On supprime alors l'élément 1 puis on relance le programme. Cette fois-ci les deux éléments initiaux sont différents, on conserve le premier, et on applique récursivement la fonction au reste de la liste. On va ainsi supprimer toutes les occurrences successives d'éléments identiques).

Q19 : Non, comme la fonction que nous avons nous-même programmée plus haut, on ne fait que comparer deux éléments consécutifs de la liste initiale, on ne supprimera donc pas des doublons éloignés l'un de l'autre. Par exemple, l'appel $doublons([0, 1, 0])$ renverra la liste $[0, 1, 0]$.

Q20 : La fonction recherche renvoie un booléen, les variables but et espace représentent respectivement une situation possible et une liste de situations possibles. Comme chaque situation est constituée d'une liste de deux files de voitures, but est donc une liste de deux listes de booléens, alors que espace est une liste de listes de deux listes de booléens (oui c'est un peu moche). La fonction successeurs renvoie aussi une liste de listes de listes de booléens puisqu'elle calcule toutes les situations atteignables à partir d'une situation donnée (c'est donc une liste de quatre listes de deux listes de booléens si on veut être extrêmement précis).

Q21 : Le meilleur choix est évidemment $in2$ puisque $in1$ effectue la recherche de façon naïve en parcourant la liste élément par élément (complexité linéaire dans le cas moyen), alors que $in2$ effectue une dichotomie de complexité logarithmique.

Q22 : Plein de façons de faire, la plus naturelle pour vous étant probablement d'additionner directement des puissances de 2. Pour le fun, je propose une version récursive du programme :

```
def versEntier(L) :
    if L==[] :
        return 0
    n=len(L)
    if L[0] :
        return 2**(n-1)+versEntier(L[1 :])
    return versEntier(L[1 :])
```

Q23 : On crée une file de longueur égale à la variable taille, si on veut qu'elle puisse contenir l'équivalent du nombre n, celui-ci doit donc contenir au maximum « taille » chiffres en base 2, autrement dit on doit avoir $n < 2^{taille}$, ou encore taille $\geqslant \lfloor \log_2(n) + 1 \rfloor$.

Il suffit de compléter par un $whilen! = 0$ (on divise par 2 tant qu'il reste des chiffres à gérer dans l'écriture binaire de n).

Q24 : Par construction, si espace n'évolue pas lors du passge dans la boucle while, la variable stop va devenir égale à True et on sortira de la boucle while. Sinon, c'est que le nombre de configurations disponibles dans espace croit strictement, ce qui ne pourra évidemment pas se faire plus de 2^n fois, nombre total de configurations disponibles (on supprime les doublons éventuels à chaque étape). La boucle s'arrêtera donc au pire après 2^n étapes.

Q25 : On initialise une variable n à la valeur 0 après la première ligne du programme, puis on ajoute une incrémentation $n = n+1$ après la ligne 9 du programme initial, et on modifie le return de la ligne 11 par un return n. On aura ainsi compté le nombre de fois où on doit faire avancer les files pour que la situation recherchée apparaisse dans la liste des situations atteignables, ce qui est exactement ce qu'on souhaite faire.

VI. Base de données.

Q26 : SELECT id_croisement_fin FROM croisement WHERE id_croisement_debut = c

Q27 : SELECT longitude, latitude FROM croisement INNER JOIN voie ON id_croisement_fin = croisement.id WHERE id_croisement_debut = c

Q28 : La requête renvoie tous les croisements atteignables à partir du croisement c.