Modélisation de la propagation d'une épidémie

Q1. Voici ce que l'on obtient en partant de L = [5, 2, 3, 1, 4]. Déjà, $\boxed{n = 5.}$

- $\boxed{i = 1, x = 2}$ et $j = 1$. On démarre la boucle `while` avec $\underline{j - 1 = 0}$.
  Comme $x <$ L[0] = 5, on remplace L[1] par L[0] et j par 0.
  La variable j vaut 0, donc c'est la fin de la boucle `while`.
  On remplace alors L[0] par x, donc par 2 : ainsi, $\boxed{\text{L} = [2, 5, 3, 1, 4]}$
- $\boxed{i = 2, x = 3}$ et $j = 2$. On démarre la boucle `while` avec $\underline{j - 1 = 1}$.
  Comme $x <$ L[1] = 5, on remplace L[2] par L[1] et j par 1.
  Or L$[j - 1]$ = L[0] = $2 < x$ donc c'est la fin de la boucle `while`.
  On remplace alors L[1] par x, donc par 3 : ainsi, $\boxed{\text{L} = [2, 3, 5, 1, 4]}$

  Ces deux premiers exemples permettent de voir le fonctionnement de la fonction `tri` *: chaque élément est échangé avec son prédécesseur tant que celui-ci lui est supérieur, ce qui, au fur et à mesure, va ranger dans l'ordre croissant les éléments du début de la liste.*

- $\boxed{i = 3, x = 1}$ et $j = 3$. On va échanger L[3] avec tous ses prédécesseurs.

  $$\boxed{\text{L} = [1, 2, 3, 5, 4]}$$

- $\boxed{i = 4, x = 4}$ et $j = 4$. On va échanger L[4] avec L[3].

  $$\boxed{\text{L} = [1, 2, 3, 4, 5]}$$

Q2. À l'issue de la boucle `for`, on a $i = n - 1$. On déduit alors de l'invariant de boucle que la liste `L[0:n]` est triée dans l'ordre croissant. Or `n=len(L)` donc `L[0:n]` vaut `L`. Ainsi,

$$\boxed{\text{L'instruction } \texttt{tri(L)} \text{ trie la liste } \texttt{L}.}$$

Q3. Estimons le nombre d'opérations effectuées dans le pire des cas, c'est-à-dire lorsque toutes les étapes possibles de la boucle `while` ont lieu.

Comme les différentes lignes de la fonction ne comportent qu'un nombre fini et fixé d'opérations, il suffit de compter le nombre de passages dans chaque boucle.

- Si l'on ne s'occupe pas de la boucle `while`, il y a déjà n passages dans la boucle `for` soit une complexité en $\mathcal{O}(n)$.
- La variable i décrit l'ensemble $\{1, \dots, n-1\}$ et, pour chaque valeur de i, la variable j va décrire l'ensemble $\{i, i-1, \dots, 1\}$: on passe donc i fois dans la boucle `while`.

  Le nombre de passages dans cette boucle est donc

  $$\sum_{i=1}^{n-1} i = \frac{(n-1) \times (1 + n - 1)}{2} = \frac{n(n-1)}{2}$$

  ce qui nous donne une complexité en $\mathcal{O}(n^2)$.

Ainsi, $\boxed{\text{La fonction } \texttt{tri} \text{ a une complexité en } \mathcal{O}(n^2).}$

Q4. Il suffit d'adapter la fonction `tri` à une liste de couples : on va comparer les entiers `L[i][1]` et `L[i-1][1]` pour déterminer si l'on échange les couples `L[i]` et `L[i-1]`.

La seule modification concerne ainsi la seconde inégalité sur la ligne du `while` :

```
def tri_chaine(L):
    n = len(L)
    for i in range(1,n):
        j = i
        x = L[i]
        while 0 < j and x[1] < L[j-1][1]:
            L[j] = L[j-1]
            j = j-1
        L[j] = x
```

Q5. Dans une relation, une clef est un ensemble d'attributs dont la valeur permet d'identifier de manière unique chaque enregistrement. Une clef primaire est une clef constituée d'un seul attribut.

Un attribut pouvant servir de clef primaire ne prendra ainsi jamais deux fois la même valeur dans une table : de ce fait, aucun des attributs `nom`, `iso` ou `annee` ne peut servir de clef primaire dans la table `palu` car

- la valeur `Bresil` de l'attribut `nom` apparaît au moins deux fois ;
- la valeur `BR` de l'attribut `iso` apparaît au moins deux fois ;
- la valeur `2010` de l'attribut `annee` apparaît au moins trois fois.

Cela vient tout simplement du fait que cette table recense des données concernant différents pays pendant plusieurs années : il y aura donc plusieurs entrées pour chaque année et pour chaque pays.

Par contre, pour un pays donné et une année donnée, il n'y aura forcément qu'une entrée. Ainsi, on peut prendre pour clef

`(iso,annee)`

Remarque : dans la table `demographie`, la clef est le couple `(pays,periode)`.

Q6. La requête proposée est une sélection composée portant sur les attributs `annee` et `deces`. Elle s'écrit simplement

```
SELECT * FROM palu WHERE annee == 2010 AND deces >= 1000;
```

Q7. Comme on a besoin de combiner des attributs apparaissant dans les tables `palu` et `demographie`, il va falloir effectuer :

- une jointure des deux tables en identifiant les clefs `(iso,annee)` et `(pays,periode)` ;
- une projection pour afficher le nom et le taux d'incidence pour chaque pays ;
- une sélection pour se restreindre à l'année 2011.

Cela donne en `SQL`

```
SELECT nom,100000*cas/pop AS taux FROM palu JOIN demographie
ON iso == pays AND annee == periode WHERE annee == 2011;
```

Q8. On va récupérer le plus grand nombre de nouveaux cas de paludisme pour l'année 2010 et l'utiliser dans une sélection pour obtenir le nom du pays associé :

```
SELECT nom FROM palu WHERE annee == 2010 AND cas ==
( SELECT max(cas) FROM palu WHERE annee == 2010 );
```

Q9. R est la relation obtenue en ne gardant que les noms de pays et les nombres de décès lors de l'année 2010 dans la table `palu`. De ce fait, `deces2010` est la liste des couples (nom de pays, nombre de décès en 2010). Pour la trier dans l'ordre souhaité, il suffit de lui appliquer la fonction `tri_chaine` en tapant

```
tri_chaine(deces2010)
```

On peut aussi le faire en `SQL`, en ajoutant à la requête `R` un tri selon le nombre de décès :

```
SELECT nom,deces FROM palu WHERE annee == 2010 ORDER BY deces;
```

Q10. Prenons – comme dans la suite de l'énoncé – pour vecteur $\boxed{X = (S, I, R, D.}$

Les composantes de ce vecteur sont dans l'intervalle $[\,0\,;1\,]$ puisque ce sont des proportions. La fonction f définissant le second membre du système **(1)** est alors

$$\boxed{\begin{array}{rl} f : & [\,0\,;1\,]^4 \longrightarrow \mathbb{R}^4 \\ & (S, I, R, D) \longmapsto (-r\,S\,I, r\,S\,I - (a+b)\,I, a\,I, b\,I) \end{array}}$$

Q11. Le plus simple est de commencer par capturer les coordonnées de X, avant de renvoyer un tableau `numpy` créé à partir de celles-ci :

```
(S,I,R,D)=X
return np.array([-r*S*I,r*S*I-(a+b)*I,a*I,b*I])
```

Q12. Plus N est grand, plus le pas $\mathrm{d}t$ est petit, meilleure est l'approximation. Ainsi, la simulation avec N = 250 est la plus précise des deux (250 points contre 7). Avec N = 7, on réagit moins vite aux variations des dérivées et les erreurs de calcul s'accumulent.

Par contre, le temps de calcul est plus important avec N = 250 car la complexité de notre programme est en $\mathcal{O}(N)$ à cause de la boucle `for`.

Q13. Dans la fonction f, le paramètre `Itau` vient simplement remplacer `I` dans l'expression `r*S*I`. On va donc mettre ligne 7 :

```
(S,I,R,D)=X
return np.array([-r*S*Itau,r*S*Itau-(a+b)*I,a*I,b*I])
```

Le calcul de `Itau` est à placer ligne 28 : grâce à l'indication de l'énoncé, c'est de la tarte ! En notant que comme `I=X[1]`, on aura `I[0]=XX[0][1]` et `I((i-p)*dt)=XX[i-p][1]` :

```
if i<p:
    Itau=XX[0][1]
else:
    Itau=XX[i-p][1]
X=X+dt*f(X,Itau)
```

On aurait aussi pu écrire `X0` à la place de `XX[0]` ... et même `0.05` à la place de `X0[1]`.

Étude de trafic routier

Q1. On prend une liste `L` de même taille de la liste définie de la manière suivante :
- s'il y a une voiture dans la case d'indice `i`, alors `L[i]=True` ;
- sinon, `L[i]=False`.

Q2. Voici deux façons de procéder :

```
A=[True,False,True,True,False,False,False,False,False,False,True]
```

pour les brutes, ou alors avec un peu plus de raffinement :

```
A=[False]*11
A[0]=A[2]=A[3]=A[10]=True
```

Q3. Il suffit simplement de renvoyer l'élément d'indice `i` de la liste `L`, puisqu'il a précisément la valeur voulue :

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

Q4. Ce sont des n-listes d'éléments de l'ensemble `{True, False}` donc il y en a $\boxed{2^n.}$

Q5. Voici l'algorithme suggéré par l'indication :
- si les listes sont de longueurs différentes, on retourne `False` ;
- sinon, on compare leurs termes un à un à l'aide d'une boucle `for` :
  - si l'on trouve deux éléments différents, on retourne `False` ;
  - si l'on sort de la boucle, cela signifie que les listes sont égales (cf. l'instruction `return` interrompt la fonction) donc on retourne `True`.

Cela s'écrit de la manière suivante en `Python` :

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

Q6. En dehors de la boucle, on a un nombre fini et fixé d'opérations. Ensuite, on a une boucle de `n=len(L1)` étapes contenant chacune contenant un nombre fini et fixé d'opérations.

Ainsi, $\boxed{\text{La complexité de la fonction } \texttt{egal} \text{ est en } \mathcal{O}(n), \text{ où } \texttt{n=len(L1)}.}$

Q8. On fait avancer la file deux fois, sans introduire de nouvelle voiture la première fois. La dernière voiture va donc sortir de la file, les autres avancent de deux cases et une nouvelle voiture occupe la première case. On obtient ainsi la liste

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

On peut noter que la liste A de la question 2 est celle de la figure 2.a, si bien que l'instruction `avancer(A,False)` renvoie la liste B de la figure 2.b.

Q9. On commence par créer une copie de la liste `L` avec l'instruction `L2=L[:]` : on ainsi les positions initiales de toutes les voitures. Ensuite, on fait avancer d'une case toutes les voitures situées dans une case d'indice `i`$\geqslant$`m` en réitérant l'instruction `L2[i+1]=L[i]` pour `m` $\leqslant$ `i` $\leqslant$ `len(L)`-2 ... sans oublier l'instruction `L2[m]=False`, puisque la case d'indice `m` se retrouve forcément vide à l'issue de l'étape.

```
def avancer_fin(L,m):
    L2=L[:]
    L2[m]=False
    for i in range(m,len(L)-1):
        L2[i+1]=L[i]
    return L2
```

Mais on peut faire plus simple en utilisant la fonction `avancer` : il suffit d'appliquer `avancer(.,False)` à la sous-liste `L[m:]` tout en gardant la sous-liste `L[:m]` intacte.

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

Q10. C'est la même idée qu'à la question précédente, sauf que l'on déplace cette fois les cases d'indices `i`$\leqslant$`m`-1 en réitérant l'instruction `L2[i+1]=L[i]` pour 0 $\leqslant$ `i` $\leqslant$ `m`-1 ... sans oublier l'instruction `L2[0]=b` pour gérer l'introduction de la nouvelle voiture.

```
def avancer_debut(L,b,m):
    L2=L[:]
    L2[0]=b
    for i in range(m):
        L2[i+1]=L[i]
    return L2
```

Mais comme précédemment, il est plus malin d'utiliser l'instruction `avancer(.,b)` pour modifier la sous-liste `L[:m+1]` tout en gardant la sous-liste `L[m+1:]` intacte.

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

Q11. Les voitures situées à gauche d'une case inoccupée d'indice `i`$<$`m` vont pouvoir avancer (dans l'exemple, `i`$=2$ et `m`$=5$). Il suffit donc de trouver le plus grand indice `i`$<$`m` d'une case inoccupée ... s'il existe ! On va pour cela initialiser la variable `i` à -1, puis parcourir tous les indices `k` de 0 à `m`-1 et stocker la valeur de `k` dans `i` chaque fois que `L[k]=False`.

Si `i`$\geqslant 0$ en fin de boucle, c'est l'indice recherché et on utilise la fonction `avancer_debut` pour faire progresser la file. Si `i`< 0 en revanche, toutes les cases à gauche du point de blocage sont occupées : la file n'évolue donc pas et on en renvoie une copie.

```
def avancer_debut_bloque(L,b,m):
    i=-1
    for k in range(m):
        if not L[k]:
            i=k
    if i>=0:
        return avancer_debut(L,b,i)
    else:
        return L[:]
```

Q12. La file L1 étant prioritaire, elle avancera dans tous les cas : en effet, les cases de `L2[m:]` n'étant pas bloquées, elles avanceront toujours et la case d'indice `m` se libèrera pour laisser passer L1 si besoin. Autrement dit :

- les listes `L1` et `L2[m:]` avancent toujours ;
- la liste `L2[:m]` avance sans contrainte si `L1[m]=False`, et avec la case d'indice `m` bloquée si `L1[m]=True`.

Il ne faut pas oublier de commencer par calculer `m`, d'où la fonction :

```
def avancer_files(L1,b1,L2,b2):
    m=(len(L1)-1)//2
    R1=avancer(L1,b1)
    R2=avancer_fin(L2,m)
    if occupe(R1,m):
        R2=avancer_debut_bloque(R2,b2,m)
    else:
        R2=avancer_debut(R2,b2,m)
    return [R1,R2]
```

Q13. La case du milieu est ici d'indice 2. Quand la file prioritaire D avance, elle bloque cette case et seules les trois dernières voitures de la file E vont pouvoir avancer. Comme aucune voiture n'est introduite dans les deux files, on obtient après l'instruction `avancer_files(D,False,E,False)`

`D=[False,False,True,False,True]` et `E=[False,True,False,True,False]`

Q14. Il suffit que les files L1 et L2 soient pleines jusqu'au croisement (comme la figure 4.a) et que l'on introduise continuellement de nouvelles voitures dans L1 pour que les voitures de L2 restent bloquées lors de l'appel `avancer_files(L1,True,L2,False)`.

Q15. Voici comment passer de la configuration 4.a à la configuration 4.b.

- Durant les 4 premières étapes : la file L1 avance *sans introduction* de voiture / la file L2 reste bloquée car il y a une voiture de L1 en case 4. On atteint de la sorte (en notant les files comme des molécules !) la configuration L1=F_4T_4F et L2=T_4F_5.
- La 5^e étape : la file L1 avance encore une fois *sans introduction* de voiture et on atteint L1=F_5T_4 / la case du milieu étant enfin libre, la file L2 peut avancer *sans introduction* de voiture et on atteint L2=FT_4F_4.
- Les 4 étapes suivantes : la file L1 avance *avec introduction* de voiture ; les voitures de tête vont alors sortir et on atteint L1=T_4F_5 / il n'y a aucun blocage donc la file L2 peut avancer *sans introduction* de voiture et on atteint L2=F_5T_4.

Q16. On ne peut pas atteindre la configuration 4.c : en effet, les deux files finissant par 4 voitures, cela signifierait que ces 8 voitures ont toutes avancé à l'étape précédente (sinon il y aurait une case vide parmi les 4 dernières d'une des files) ... et donc qu'il y avait à la fois une voiture de L1 et une voiture de L2 sur la case centrale. Ce qui est impossible puisque l'on exclut la possibilité d'un accrochage !