Mines Informatique commune MP 2016 — Corrigé

Ce corrigé est proposé par Virgile Andreani (ENS Ulm) ; il a été relu par Julien Dumont (Professeur en CPGE) et Jean-Julien Fleck (Professeur en CPGE).

Ce sujet d'informatique propose d'étudier la propagation des épidémies. Ses trois parties sont indépendantes.

- La première partie, très classique, est l'occasion d'étudier un algorithme de tri. Elle comporte quelques questions de difficulté croissante sur les bases de données relationnelles.
- Dans la deuxième partie, il faut compléter un code permettant d'intégrer un système d'équations différentielles modélisant la propagation d'une épidémie. Le système est ensuite modifié pour introduire un retard, puis on passe à un ensemble d'équations intégro-différentielles dont on doit calculer l'intégrale par la méthode des rectangles.
- La troisième partie, un peu plus originale, propose d'ajouter des dimensions spatiales au modèle, simulé cette fois-ci par un automate cellulaire non déterministe.

Peu de programmation est demandée. Hormis une ou deux questions, le sujet est d'une difficulté modérée ; il est accessible dès la première année de prépa. Les réponses attendues sont souvent plus courtes que les questions. C'est un bon sujet d'entraînement pour s'assurer que l'on sait appliquer le cours sur des exemples simples.

Indications

Partie I

2 Raisonner par récurrence pour prouver l'invariant de boucle.

7 Il faut faire une jointure : associer les colonnes ayant la même signification entre les tables.

Partie II

13 `Itau` est une information qui figure dans la liste `XX`.

14 Utiliser `XX` pour calculer la valeur de l'intégrale, inutile de modifier la fonction `f`.

Partie III

17 Il existe deux manières de répondre à cette question, l'une est plus efficace que l'autre et serait probablement valorisée lors de la correction.

20 Attention à faire en sorte que la mise à jour de toutes les cellules soit simultanée : on ne doit pas appeler la fonction `est_exposee` sur une grille partiellement mise à jour.

21 L'énoncé demande ici les proportions de cases dans les différents états et non pas leur nombre comme précédemment.

I. Tri et bases de données

1 Lors de l'itération i de la boucle `for`, l'élément i de la liste de départ est descendu jusqu'à sa place dans la liste triée constituée des $i+1$ premiers éléments. Par conséquent, le contenu de la liste évolue de la manière suivante :

- Début de la fonction : `[5, 2, 3, 1, 4]` ;
- Fin de l'itération $i = 1$: `[2, 5, 3, 1, 4]` ;
- Fin de l'itération $i = 2$: `[2, 3, 5, 1, 4]` ;
- Fin de l'itération $i = 3$: `[1, 2, 3, 5, 4]` ;
- Fin de l'itération $i = 4$: `[1, 2, 3, 4, 5]` ;
- Sortie de la fonction : `[1, 2, 3, 4, 5]`.

> Vous aurez sûrement reconnu le tri par insertion !

2 Soit $\mathscr{P}(i)$ la propriété

> La liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération i.

On va montrer par récurrence sur i que $\mathscr{P}(i)$ est vraie pour tout $i \in [\![1\,;n-1]\!]$.

- Lors de la première itération, $i = 1$ donc la valeur du deuxième élément de la liste est affectée à `x`. Deux cas sont alors possibles : soit `L[0]` $>$ `x` et le corps de la boucle `while` est effectué une seule fois, ce qui a pour effet d'échanger `L[0]` et `L[1]`. Soit `L[0]` $\leqslant$ `x` et le corps de la boucle n'est jamais exécuté. Dans les deux cas, les deux premiers éléments de la liste se retrouvent triés par ordre croissant à la fin de la première itération, ce qui satisfait $\mathscr{P}(1)$.
- $\mathscr{P}(i-1) \Longrightarrow \mathscr{P}(i)$: $\mathscr{P}(i-1)$ nous permet de considérer que les i premiers éléments de `L` (soit de 0 à $i-1$ inclus) sont triés à la fin de l'itération $i-1$, et donc également au début de l'itération i. On peut alors distinguer deux cas :
  - `L[i]` est supérieur ou égal à tous les éléments de `L[0:i]`. Dans ce cas, le programme n'entre pas dans la boucle `while` puisque sa deuxième condition est fausse, et l'affectation de la ligne 9 ne fait que remplacer `L[j]` par lui-même puisque j est toujours égal à i. La liste `L[0:i+1]` est par conséquent elle aussi triée.
  - Il existe au moins un élément de `L[0:i]` strictement supérieur à `L[i]`. La boucle `while` est exécutée et, à sa sortie, on sait que `L[i]` $<$ `L[j]` (la condition de la boucle était vraie à l'itération précédente), et que soit $j = 0$, soit `L[j-1]` $\leqslant$ `L[i]` (la condition de la boucle est fausse). Comme tous les éléments de `L` d'indices compris entre j et $i-1$ avant la boucle ont été décalés d'un cran vers la droite, la place est libre en `L[j]` pour accueillir l'ancien `L[i]`. La liste `L[0:i+1]` est donc également triée.

Par conséquent | La propriété $\mathscr{P}(i)$ est un invariant de boucle. |

La dernière itération correspond à $i = n-1$, ainsi $\mathscr{P}(n-1)$ assure que **la liste `L[0:n]`, donc toute la liste, est triée**.

> Une manière mathématiquement équivalente mais un peu plus succincte de démontrer le cas de base de la récurrence aurait été de partir de $\mathscr{P}(0)$: bien que l'itération $i = 0$ ne soit pas explicite dans le code, elle correspond à l'état de la liste au début de la fonction, avant la première itération. La liste `L[0:1]` ne contient qu'un seul élément, on peut donc la considérer comme

triée. Donc $\mathscr{P}(0)$ est vraie.

Par ailleurs, cela n'était sûrement pas demandé (la démonstration est assez longue ainsi), mais pour être complet, on aurait dû démontrer deux propriétés supplémentaires. D'une part, que l'on indice toujours la liste `L` dans les limites de sa taille et d'autre part que la fonction change uniquement l'ordre des éléments et pas leur valeur. Concernant cette dernière propriété, l'énoncé commet un abus par hypothèse implicite en invitant à déduire de $\mathscr{P}(n)$ la conclusion que `tri(L)` trie, au sens de l'énoncé, la liste `L`. En effet, cette condition, bien que nécessaire, n'est pas suffisante : imaginons que la boucle `for` écrase progressivement chaque élément de `L` avec son premier élément : $\mathscr{P}(i)$ est toujours vraie pour tout i, la liste finale est bien triée puisqu'elle ne contient que n copies de son premier élément, mais elle n'a plus rien à voir avec la liste initiale. Il est donc nécessaire de démontrer également la propriété $\mathscr{Q}(i)$:

> À l'issue de l'itération i, la liste `L[0:i+1]` contient les mêmes éléments qu'au début de la fonction.

Cette démonstration, à faire également par récurrence, est laissée en exercice au lecteur !

3 À taille de liste fixée, seul le corps de la boucle `while` est exécuté un nombre variable de fois. Dans le meilleur cas, cette portion de code n'est jamais exécutée : c'est ce qui se passe pour les listes déjà triées. En définissant la complexité de cet algorithme par le nombre de comparaisons entre éléments de la liste qu'il effectue, on trouve dans le meilleur cas que celui-ci est égal à $n-1$ (une par itération de `for`, dans la condition du `while`), par conséquent

Le meilleur cas, obtenu pour les listes déjà triées, est linéaire ($\mathrm{O}(n)$).

Dans le pire cas, la boucle `while` s'exécute le maximum de fois à chaque itération, soit i fois à l'itération i. C'est ce qui se passe lorsque `L` est initialement triée dans l'ordre décroissant. Le nombre de comparaisons d'éléments de la liste est alors

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

Le pire cas, celui des listes triées dans l'ordre inverse, est quadratique ($\mathrm{O}(n^2)$).

Le **tri fusion** a une complexité quasi-linéaire en $\mathbf{O(n\log n)}$ dans le meilleur comme dans le pire cas.

> Attention à ne pas citer le tri rapide, qui, bien que quasi-linéaire en moyenne, est quadratique dans le pire cas.

4 On dispose déjà d'une fonction de tri, il suffit de l'adapter très légèrement pour effectuer la comparaison sur le deuxième élément de chaque liste. On reprend donc entièrement le code de `tri` en modifiant seulement la condition 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]:
```