X2022 — INF411
Contrôle classant / 14 novembre 2023 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les FUI et FUI-FF. Les calculatrices, ordinateurs, tablettes et téléphones portables sont interdits.

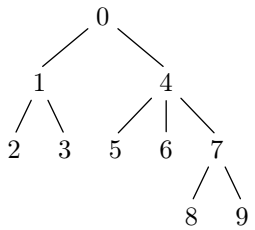
L'énoncé est composé de deux problèmes indépendants, que vous pouvez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pouvez, quand vous traitez une question, considérer comme déjà traitées les questions précédentes du même problème.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Les questions de code n'appellent aucune justification. Une indication du nombre de lignes attendues est donnée entre parenthèses. Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Plus proche ancêtre commun

Dans ce problème, on s'intéresse à la question suivante : étant donné un arbre et deux nœuds dans cet arbre, quel est le plus proche ancêtre commun de ces deux nœuds ? Une solution efficace à ce problème a de nombreuses applications, notamment en bio-informatique. Ainsi, dans l'arbre



(1)

le plus proche ancêtre commun des nœuds 5 et 8 est le nœud 4, et celui des nœuds 3 et 7 est la racine 0. De manière générale, on se donne un arbre quelconque, sur lequel on s'autorise un prétraitement, puis on souhaite calculer le plus efficacement possible le plus proche ancêtre commun pour des couples de nœuds dans cet arbre.

Les arbres que l'on considère ici ont des nœuds étiquetés par des entiers distincts. Plus précisément, un arbre de taille n a ses nœuds étiquetés par les entiers $0, 1, \ldots, n-1$, avec une numérotation qui vérifie la propriété suivante :

pour tout i, les nœuds du sous-arbre i portent des numéros consécutifs à partir de i (P)

On note que la racine est donc nécessairement le nœud 0. On pourra vérifier que l'arbre (1) ci-dessus vérifie bien la propriété (P).

Si un nœud j appartient au sous-arbre i, on dit que i est un *ancêtre* de j. En particulier, chaque nœud est son propre ancêtre. Le *plus proche ancêtre commun* de deux nœuds i et j, noté $\mathrm{LCA}(i, j)$, est défini formellement comme la racine du plus petit sous-arbre pour l'inclusion contenant i et j. Dit autrement,

- si i est un ancêtre de j, alors $\mathrm{LCA}(i, j) = i$;
- symétriquement, si j est un ancêtre de i, alors $\mathrm{LCA}(i, j) = j$;
- sinon, $\mathrm{LCA}(i, j)$ est l'unique nœud a possédant au moins deux sous-arbres distincts t_1 et t_2 tel que i appartient au sous-arbre t_1 et j au sous-arbre t_2.

```
1  class Tree {
2    final int v; // l'étiquette du noeud
3    final LinkedList<Tree> f; // les sous-arbres
4  }
5  class LCA {
6    final Tree t; // l'arbre considéré
7    final int n; // et sa taille
8    ... // éventuellement ici d'autres choses
9
10   LCA(int n, Tree t) {
11     this.n = n;
12     this.t = t;
13     ... // éventuellement ici un pré-traitement de l'arbre t
14   }
15
16   // le plus proche ancêtre commun de i et j dans t
17   int lca(int i, int j) { ... }
18 }
```

FIGURE 1 – Structure du code.

Mise en œuvre en Java. La figure 1 contient deux classes Java `Tree` et `LCA`. Un objet de la classe `Tree` représente un nœud d'un arbre, avec son étiquette `v` et la liste `f` de ses sous-arbres. Cette liste n'est jamais `null` mais elle peut être vide (lorsque le nœud est une feuille). En particulier, on pourra donc toujours la parcourir avec une boucle `for (Tree u: t.f)` pour un arbre `t`.

Un objet de la classe `LCA` stocke un arbre `t` et sa taille `n` (lignes 6–7), et offre une méthode `lca` pour calculer le plus proche ancêtre commun de deux nœuds `i` et `j` (ligne 17), en supposant $0 \leq \texttt{i}, \texttt{j} < \texttt{n}$. La classe `LCA` peut contenir d'autres champs (ligne 8) et un pré-traitement arbitraire réalisé dans le constructeur (ligne 13). Nous allons maintenant considérer deux implémentations différentes de la classe `LCA`, avec des pré-traitements différents.

Une solution simple mais inefficace. On ajoute à la classe `LCA` un tableau d'entiers `size` de taille `n`, c'est-à-dire qu'on déclare `int[] size;` comme un troisième champ de la classe `LCA` et qu'on ajoute `this.size = new int[n]` dans le constructeur. Ce tableau va contenir, dans sa case i, le nombre de nœuds du sous-arbre i. Ainsi, pour l'arbre (1), on aura le tableau $[10, 3, 1, 1, 6, 1, 1, 3, 1, 1]$.

Question 1 Écrire une méthode récursive `int computeSize(Tree u)` qui remplit le tableau `size` pour tous les nœuds de l'arbre `u`, supposé être un sous-arbre de `t`, et renvoie la taille de l'arbre `u`. La complexité doit être linéaire en la taille de `u`. (6 lignes)

Par la suite, on suppose avoir appelé `computeSize(t)` dans le constructeur de la classe `LCA`, de sorte que le tableau `size` est maintenant rempli pour l'arbre `t`.

Question 2 Écrire une méthode `boolean belongsTo(int i, int j)` qui prend en arguments deux entiers i et j et détermine, en temps constant, si i appartient au sous-arbre j. (2 lignes)

Question 3 Écrire la méthode `int lca(int i, int j)` qui prend en arguments deux entiers i et j et détermine $\mathrm{LCA}(i, j)$ en temps $O(\texttt{n})$. Indication : écrire une méthode auxiliaire récursive qui prend également un arbre en argument. (7 lignes)

Une meilleure solution. Dans cette seconde partie, on va effectuer un pré-traitement de l'arbre, en temps $O(\mathtt{n}\log\mathtt{n})$, qui permettra de déterminer ensuite en temps constant le plus proche ancêtre commun pour tout couple de nœuds.

On commence par construire une séquence de nœuds en effectuant un *tour eulérien* de l'arbre à partir de sa racine. D'une manière générale, le tour eulérien à partir du nœud i est ainsi défini :

1. ajouter le nœud i à la fin de la séquence résultat ;
2. pour chaque sous-arbre j de i,
   (a) effectuer un tour eulérien à partir de j,
   (b) ajouter le nœud i à la fin de la séquence résultat.

Ainsi, sur l'exemple de l'arbre (1), on obtient la séquence suivante :

0, 1, 2, 1, 3, 1, 0, 4, 5, 4, 6, 4, 7, 8, 7, 9, 7, 4, 0

Question 4 Montrer que le tour eulérien d'un arbre de n nœuds contient exactement $2n-1$ éléments.

Par la suite, on note `m` la valeur $2\mathtt{n}-1$ et on suppose avoir déclaré dans la classe `LCA` un tableau d'entiers `euler` de taille `m` destiné à contenir le résultat du tour eulérien. On déclare également un tableau d'entiers `index` de taille `n`, destiné à être un index inverse du tableau `euler`, c'est-à-dire que, pour tout $0 \leq i < \mathtt{n}$, on a $\mathtt{euler}[\mathtt{index}[i]] = i$ (s'il existe plusieurs valeurs possibles pour $\mathtt{index}[i]$, on pourra choisir arbitrairement).

Question 5 Écrire une méthode `int eulerTour(int i, Tree t)` qui effectue un tour eulérien de l'arbre `t`, en remplissant le tableau `euler` à partir de l'indice `i` et en remplissant également le tableau `index`. Elle renvoie l'indice immédiatement après le dernier indice qu'elle a rempli dans le tableau `euler`. (9 lignes)

On suppose que le constructeur de la classe `LCA` se poursuit avec les lignes

```
int m = 2*n-1;
this.euler = new int[m];
this.index = new int[n];
assert eulerTour(0, t) == m;
```

de telle sorte que les tableaux `euler` et `index` sont maintenant alloués et remplis pour l'arbre `t` qui nous intéresse.

Question 6 Montrer que le plus proche ancêtre commun de i et j dans `t` est égal au plus petit élément du tableau `euler` compris entre les indices $\mathtt{index}[i]$ et $\mathtt{index}[j]$.

Question 7 Écrire une méthode `int log2(int n)` qui prend en argument un entier n, avec $n \geq 1$, et calcule le plus grand entier k tel que $2^k \leq n$. (4 lignes)

On pose $\mathtt{k} = \mathtt{log2}(\mathtt{m})$. On suppose avoir déclaré dans la classe `LCA` une matrice d'entiers `int[][] mat` et l'avoir alloué avec la taille $\mathtt{m} \times (\mathtt{k}+1)$ dans le constructeur, c'est-à-dire

```
int k = log2(m);
this.mat = new int[m][k+1];
```

Question 8 Écrire du code qui remplit la matrice `mat` de telle sorte que, pour tout $0 \leq j \leq \texttt{k}$ et tout $0 \leq i \leq \texttt{m} - 2^j$, on ait

$$\texttt{mat}[i][j] = \min_{i \leq l < i+2^j} \texttt{euler}[l]$$

On garantira une complexité $O(\texttt{n} \log \texttt{n})$, c'est-à-dire au plus proportionnelle à la taille de la matrice `mat`. (6 lignes)

Question 9 Écrire une méthode `int minimum(int i, int j)` qui prend en arguments deux entiers i et j, avec $0 \leq i \leq j < \texttt{m}$, et qui détermine le plus petit élément du segment $\texttt{euler}[i..j]$ (les deux bornes étant incluses) en temps constant. Indication : on pourra avantageusement se resservir de la méthode `log2`. (4 lignes)

Question 10 Écrire la méthode `int lca(int i, int j)` qui prend en arguments deux entiers i et j et détermine $\mathrm{LCA}(i, j)$ en temps constant. (4 lignes)

```
class State {
  LinkedList<State> moves() { ... }
  boolean success() { ... }
  static State e0 = ...; // état initial
}
```

FIGURE 2 – Structure d'un jeu à un joueur.

2 Jeux à un joueur

Nous nous intéressons ici à des jeux à un joueur où une configuration initiale est donnée et où le joueur effectue une série de déplacements pour parvenir à une configuration gagnante. Des casse-tête tels que le *Rubik's Cube*, le solitaire, l'âne rouge ou encore le taquin entrent dans cette catégorie. L'objectif de ce problème est d'étudier différents algorithmes pour trouver des solutions à de tels jeux qui minimisent le nombre de déplacements effectués.

Un jeu à un joueur peut être vu comme un graphe, avec un ensemble de sommets E, un élément $e_0 \in E$, une fonction $a : E \to \mathcal{P}(E)$ et un sous-ensemble F de E. L'ensemble E représente les états possibles du jeu. L'élément e_0 est l'état initial. Pour un état e, l'ensemble $a(e)$ représente tous les états atteignables en un coup à partir de e. Enfin, F est l'ensemble des états gagnants du jeu. On dit qu'un état e_p est à la *profondeur* p s'il existe une séquence finie de $p+1$ états

$$e_0 \; e_1 \; \ldots \; e_p$$

avec $e_{i+1} \in a(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une *solution* du jeu, de profondeur p. Une solution *optimale* est une solution de profondeur minimale. On notera qu'un même état peut être à plusieurs profondeurs différentes.

Voici un exemple de jeu :

$$\begin{array}{rcl} E & = & \mathbb{N}^\star \\ e_0 & = & 1 \\ a(n) & = & \{\, 2n,\, n+1 \,\} \end{array} \tag{2}$$

Question 11 Donner une solution optimale pour ce jeu lorsque $F = \{\, 42 \,\}$. Il n'est pas demandé de justification.

Mise en œuvre en Java. On peut représenter un jeu en Java avec une classe `State` dont la structure est donnée figure 2. Les instances de la classe `State` sont les éléments de E. La méthode `moves` correspond à la fonction a et la méthode `success` définit le sous-ensemble F.

Parcours en largeur. Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un code Java pour un tel parcours est donné figure 3.

Question 12 Illustrer le contenu du tableau redimensionnable `a` pendant les trois premières itérations de la boucle `while` lorsque `BFS(State.e0)` est lancé sur le jeu (2).

Question 13 Expliquer le rôle des deux tableaux redimensionnables `a` et `b` dans le code de la méthode `BFS`.

Question 14 Montrer que `BFS(State.e0)` renvoie un entier p si et seulement si une solution optimale de profondeur p existe.

```
static int BFS(State e0) {
  int p = 0;
  Vector<State> a = new Vector<>();
  a.add(e0);
  while (!a.isEmpty()) {
    Vector<State> b = new Vector<>();
    for (State x: a) {
      if (x.success())
        return p;
      b.addAll(x.moves());
    }
    a = b;
    p++;
  }
  throw new Error("pas de solution");
}
```

FIGURE 3 – Parcours en largeur.

```
static boolean DFS(int m, State s, int p) {
  if (p > m)
    return false;
  if (s.success())
    return true;
  for (State x: s.moves())
    if (DFS(m, x, p+1))
      return true;
  return false;
}
```

FIGURE 4 – Parcours en profondeur.

Question 15 Donner un exemple de jeu où la méthode `BFS` nécessite un espace proportionnel à 2^p avant de renvoyer un entier p.

Parcours en profondeur. Comme on vient de le montrer, le parcours en largeur permet de trouver une solution optimale mais il peut consommer un espace important pour cela. On peut y remédier en utilisant plutôt un parcours en profondeur. La figure 4 contient le code Java d'une méthode `DFS` effectuant un parcours en profondeur à partir d'un état `s` de profondeur `p`, sans dépasser une profondeur maximale `m` donnée.

Question 16 Illustrer les différents appels à la méthode `DFS` lorsqu'on appelle `DFS(2, State.e0, 0)` sur le jeu (2) et avec $F = \{42\}$.

Question 17 Majorer l'espace utilisé par la méthode `DFS`, en fonction de `m` et `p` (et en ignorant l'espace utilisé par les listes renvoyées par `moves`). Attention, la méthode `DFS` est récursive.

Question 18 Montrer que `DFS(m, State.e0, 0)` renvoie `true` si et seulement si une solution de profondeur inférieure ou égale à `m` existe.

Recherche itérée en profondeur. Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec $\texttt{m} = 0$, puis avec $\texttt{m} = 1$, puis avec $\texttt{m} = 2$, etc., jusqu'à ce que `DFS(m, State.e0, 0)` renvoie `true`.

Question 19 Écrire une méthode `int IDS(State e0)` qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette méthode ne termine pas. (5 lignes)

Question 20 On suppose $E = \mathbb{N}^\star$, $e_0 = 1$ et $F = \{p\}$ avec $p \geq 1$. Comparer les complexités en temps et en espace, en fonction de p, du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants :

1. $a(n) = \{n + 1\}$ (il y a exactement un état à chaque profondeur) ;
2. $a(n) = \{2n, 2n + 1\}$ (il y a exactement 2^n états à la profondeur n).

On demande de justifier les complexités qui seront données.

A Bibliothèque standard Java

`class LinkedList<E>`

`void add(E e)`	ajoute l'élément `e` à la fin de la liste
`void addFirst(E e)`	ajoute l'élément `e` au début de la liste

On peut parcourir tous les éléments d'une liste `l` avec la construction

```
for (E x: l) ...
```

`class Vector<E>` un tableau redimensionnable avec des éléments de type `E`

`Vector()`	renvoie un nouveau tableau redimensionnable, vide
`boolean isEmpty()`	renvoie `true` si et seulement si le tableau est vide
`void add(E x)`	ajoute l'élément `x` à la fin du tableau
`void addAll(Collection<E> c)`	ajoute tous les éléments de la collection `c` à la fin du tableau (la collection `c` peut être une `LinkedList`)

On peut parcourir tous les éléments d'un tableau redimensionnable `v` avec la construction

```
for (E x: v) ...
```

* *
*