Algorithmique & Programmation (INF431)
Contrôle classant CC1
CORRIGÉ

28 avril 2014

Les parties I et II sont indépendantes l'une de l'autre. Elles peuvent être traitées dans l'ordre de votre choix. Elles peuvent être traitées sur la même feuille.

Première partie

Cordes

La classe `String` de Java permet de représenter des chaînes de caractères. On rappelle que cette classe est munie (en particulier) des méthodes suivantes :

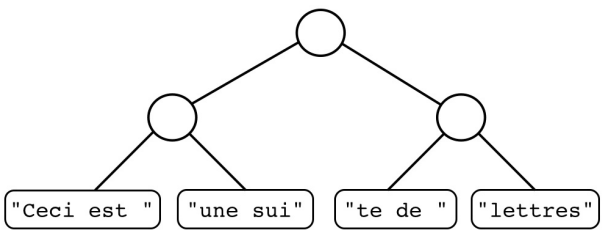
```
   int length ()
String substring (int i, int j)
```

La première méthode renvoie la longueur de la chaîne concernée. Sa complexité en temps est $O(1)$. La seconde méthode exige `0 <= i <= j <= n`, où `n` est la longueur de la chaîne concernée. Cette méthode, appliquée à une chaîne constituée des caractères $c_0, c_1, \ldots c_{n-1}$, construit et renvoie la chaîne $c_i, c_{i+1}, \ldots, c_{j-1}$. On note que `s.substring(0, s.length())` est égal à `s`. On rappelle enfin que l'opérateur `+` permet de concaténer deux chaînes de caractères.

Les *cordes*, en anglais *ropes*, sont une structure de données permettant de représenter les chaînes de caractères. Elles offrent des fonctionalités similaires à la classe `String` mais effectuent plus efficacement certaines opérations, comme l'insertion à l'intérieur d'une corde.

Dans ce qui suit, on appelle **chaîne de caractères** un objet de classe `String`.

Une **corde** est un arbre binaire dont les feuilles contiennent des chaînes de caractères. La chaîne représentée par une corde est obtenue en concaténant les feuilles de gauche à droite. Voici une représentation possible (il y en a plusieurs) de `"Ceci est une suite de lettres"` :



On se donne une classe abstraite `Rope` qui représente une corde. Pour l'implémentation, on définit deux classes concrètes `Leaf` et `Node`, qui représentent respectivement une feuille et un nœud binaire. Notez que les champs sont non mutables.

```
abstract class Rope {}

class Leaf extends Rope {
  private final String s;
  Leaf (String st) { s = st; }
}

class Node extends Rope {
  private final Rope left;
  private final Rope right;
  Node (Rope l, Rope r) { left = l; right = r; }
}
```

Dans les questions qui suivent, on va progressivement étendre ces définitions.

Question 1 Étendez ces définitions de façon à ce que, si `r` est un objet de classe `Rope`, alors `r.toString()` renvoie la chaîne représentée par `r`. Ne vous inquiétez pas de l'efficacité de votre solution. ◇

Solution. Il n'est pas nécessaire d'ajouter à la classe abstraite `Rope` une déclaration de la méthode `toString`, car tous les objets ont une méthode `toString`, définie dans la classe `Object`.

Il faut simplement redéfinir cette méthode dans les classes `Leaf` et `Node`, respectivement comme suit :

```
/* Leaf */
public String toString() { return s; }

/* Node */
public String toString() {
  return left.toString() + right.toString();
}
```

Ce code est inefficace à cause du coût élevé de la concaténation de chaînes (+). En réalité, pour éviter ce problème, on préférerait employer un objet intermédiaire de classe `StringBuffer`. Ce n'était pas demandé ici.

Note du correcteur. On a rencontré parfois des solutions qui consistaient à placer la méthode `toString` dans la classe mère, `Rope`, et à utiliser un test (par exemple, `if this instanceof Leaf`) pour déterminer à quelle sous-classe on a affaire. Ce style n'est pas recommandé et a été pénalisé.

On note que les champs `left` et `right` ne sont jamais `null` (cette convention n'était pas explicitement rappelée dans le sujet, mais était celle employée à plusieurs reprises en cours). Il était donc inutile de tester. □

Question 2 Étendez les définitions de façon à ce que les cordes soient munies d'une méthode `int length()`. On souhaite que, si `r` est une corde, alors `r.length()` renvoie la longueur de la chaîne représentée par `r`. **On exige de plus que cette méthode ait une complexité en temps** $O(1)$ **dans le cas le pire.** Vous pouvez si besoin ajouter un champ (ou des champs) aux classes `Leaf` et `Node` et modifier les constructeurs de ces classes. Toutefois, **on exige que la complexité en temps des constructeurs reste** $O(1)$ **dans le cas le pire.** ◇

Solution. Dans le classe `Leaf`, il est facile d'obtenir la longueur de la chaîne en temps constant, grâce à la méthode `length` de la classe `String`. Dans la classe `Node`, cependant, un problème se pose. On ne peut pas calculer `left.length() + right.length()`, car la complexité de la

méthode `length` serait alors $O(n)$, où n est le nombre de nœuds de l'arbre. L'idée est alors de stocker dans chaque objet de classe `Node` la longueur de la chaîne qu'il représente. Si on suppose que la classe `Node` contient un champ `length`, alors la méthode `length()` est facile à implémenter !

Pour ce qui concerne la méthode `length()`, on ajoute donc dans les classes `Rope`, `Leaf` et `Node`, respectivement :

```
/* Rope */
abstract int length();

/* Leaf */
int length() { return s.length(); }

/* Node */
int length() { return length; }
```

Il reste à déclarer le champ `length` et à faire en sorte que celui-ci soit initialisé. Ce champ peut être déclaré non mutable, et doit alors être initialisé par le constructeur de la classe `Node`. Cette initialisation se fait en temps constant, puisque la méthode `length`, appliquée aux fils gauche et droit, s'exécute en temps constant.

On modifie donc ainsi la classe `Node` :

```
/* Node */
private final int length; // declare the field
Node (Rope l, Rope r) {
  left = l;
  right = r;
  length = left.length() + right.length(); // initialise it
}
```

Cette technique illustre un compromis entre temps et espace. On paie un peu plus cher en espace (en ajoutant un champ `length` dans chaque nœud) pour que la méthode `length()` puisse s'exécuter en temps constant. □

Dans les trois questions qui suivent, on refuse les solutions qui demandent la construction d'une chaîne de caractères inutilement grande. Par exemple, dans la question 3 ci-dessous, on refuse la réponse triviale `System.out.print(r.toString().substring(i, j))`.

Question 3 Munissez les cordes d'une méthode `void printSubstring(int i, int j)` de sorte que, si `r` est une corde, alors `r.printSubstring(i, j)` affiche la sous-chaîne de `r` formée par les caractères compris entre les positions `i` (inclus) et `j` (exclus). Vous pourrez supposer `0 <= i <= j <= r.length()`. Cette convention à propos des indices `i` et `j` est la même que celle de la méthode `substring` de la classe `String`, dont le comportement a été rappelé en préambule. Vous ferez en sorte que `printSubstring` s'exécute en temps constant dans le cas particulier où `i == j`. ◇

Solution. On ajoute dans les classes `Rope`, `Leaf` et `Node`, respectivement :

```
/* Rope */
abstract void printSubstring (int i, int j);

/* Leaf */
void printSubstring (int i, int j) {
  System.out.print(s.substring(i, j));
}

/* Node */
```

```
void printSubstring (int i, int j) {
  if (i == j)
    return;
  int m = left.length();
  if (i < m)
    left.printSubstring(i, Math.min(j, m));
  if (j > m)
    right.printSubstring(Math.max(0, i - m), j - m);
}
```

Dans la classe `Node`, il fallait organiser le code de façon à effectuer un appel récursif à gauche, à droite, ou les deux, suivant les valeurs de `i` et `j`. Plusieurs écritures étaient possibles.

Note du correcteur. Le test `if (i == j) return;` dans la classe `Node` permet d'éviter des appels récursifs inutiles. Il est nécessaire pour obtenir la complexité $O(1)$ dans le cas où `i` et `j` sont égaux. □

On se donne la classe suivante, qui représente une paire de cordes :

```
class PairRope {
  Rope left;
  Rope right;
  PairRope (Rope l, Rope r) { left = l; right = r; }
}
```

Question 4 Munissez les cordes d'une méthode `PairRope split (int i)` qui renvoie une paire formée par deux cordes représentant les sous-chaînes délimitées respectivement par les indices [`0`...`i`[et [`i`...`r.length()`[. Vous pourrez supposer `0 <= i <= r.length()`. ◇

Solution. On ajoute dans les classes `Rope`, `Leaf` et `Node`, respectivement :

```
/* Rope */
abstract PairRope split (int i);

/* Leaf */
PairRope split (int i) {
  return new PairRope (
    new Leaf (s.substring(0, i)),
    new Leaf (s.substring(i, s.length()))
  );
}

/* Node */
PairRope split (int i) {
  int m = left.length();
  if (i < m) {
    PairRope s = left.split(i);
    s.right = new Node (s.right, right);
    return s;
  }
  else if (i == m) {
    return new PairRope (left, right);
  }
  else {
    PairRope s = right.split(i - m);
    s.left = new Node (left, s.left);
    return s;
  }
}
```

Dans la classe `Node`, on a tiré parti du fait qu'un objet de classe `PairNode` est modifiable. On aurait pu également ne pas en tirer parti et allouer un nouvel objet de classe `PairNode` au lieu de modifier l'objet renvoyé par l'appel récursif.

Note du correcteur. Dans la solution ci-dessus, on n'a pas traité spécialement les cas où `i` est égal à `0` ou bien à `this.length()`. On pouvait bien sûr, dans ces cas particuliers, renvoyer (en temps constant) une paire constituée d'une corde vide et de la corde `this`. □

Question 5 À l'aide de la méthode `split` et en modifiant uniquement la classe `Rope`, définissez une méthode `Rope substring (int i, int j)` qui renvoie une corde représentant la sous-chaîne délimitée par les indices [`i` . . . `j`[. Vous pourrez supposer `0 <= i <= j <= r.length()`.◇

Solution. Il suffit de composer deux opération `split`. On ne modifie que la classe parent, `Rope` :

```
Rope substring (int i, int j) {
  PairRope p = this.split(i);
  PairRope q = p.right.split(j - i);
  return q.left;
}
```

Ou, pour plus de concision encore :

```
Rope substring (int i, int j) {
  return split(i).right.split(j - i).left;
}
```

Ou bien, si on veut éviter une soustraction :

```
Rope substring (int i, int j) {
  return split(j).left.split(i).right;
}
```

Les classes `Leaf` et `Node` ne sont pas modifiées, puisque ce comportement est défini dans la classe parent et hérité par toutes les cordes, feuilles ou nœuds. □

Deuxième partie

Calcul des distances dans un graphe

1 Exponentiation rapide de matrices booléennes

On suppose qu'une opération élémentaire sur les nombres entiers (addition, soustraction, multiplication, comparaison) exige un temps $O(1)$. On suppose également qu'une lecture ou une écriture en mémoire exige un temps $O(1)$.

Si A et B sont des matrices carrées de taille n à coefficients entiers, on note $A \cdot B$ leur produit au sens de l'anneau $(\mathbb{Z}, +, \times)$. Pour calculer ce produit, on suppose que l'on dispose d'un algorithme de multiplication dont la complexité en temps est $O(n^\omega)$.

Question 6 Donnez deux entiers p et q pour lesquels on peut supposer $p \leq \omega \leq q$. Justifiez brièvement pourquoi. ◇

Solution. On peut supposer $2 \leq \omega \leq 3$. Pour justifier $2 \leq \omega$, on note qu'une matrice carrée de taille n est composée de n^2 coefficients, et que tout algorithme de multiplication doit lire chacun de ces coefficients. Pour justifier $\omega \leq 3$, on note que l'algorithme naïf de multiplication a une

complexité $O(n^3)$. L'algorithme de Strassen (1969), étudié en cours, donne $\omega = \log_2 7 = 2.807\ldots$ Des algorithmes plus complexes, qui fournissent de meilleures valeurs de ω, ont été mis au point depuis. □

On identifie les booléens **faux** et **vrai** avec les entiers 0 et 1. Une matrice à coefficients booléens peut donc être considérée également comme une matrice à coefficients entiers.

On note $\mathbb{B} = \{0, 1\}$ l'ensemble des booléens. Sur cet ensemble, l'opération « ou » est appelée disjonction et notée $\vee$; l'opération « et » est appelée conjonction et notée $\wedge$. La structure $(\mathbb{B}, \vee, \wedge)$ forme un semi-anneau. (Un semi-anneau doit vérifier les mêmes propriétés qu'un anneau, à ceci près que les éléments n'ont pas forcément d'opposé. Dans $(\mathbb{B}, \vee, \wedge)$, il n'y a pas de notion d'opposé pour l'opération de disjonction.)

Si A et B sont des matrices carrées de taille n à coefficients booléens, on note $A \odot B$ leur produit au sens du semi-anneau $(\mathbb{B}, \vee, \wedge)$. On souligne que $A \cdot B$ et $A \odot B$ sont deux opérations distinctes. On note Id la matrice identité, qui est la même pour ces deux opérations.

On a donc par exemple, pour $n = 2$:

$$Id = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \odot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} (a_{00} \wedge b_{00}) \vee (a_{01} \wedge b_{10}) & (a_{00} \wedge b_{01}) \vee (a_{01} \wedge b_{11}) \\ (a_{10} \wedge b_{00}) \vee (a_{11} \wedge b_{10}) & (a_{10} \wedge b_{01}) \vee (a_{11} \wedge b_{11}) \end{bmatrix}$$

Question 7 Montrez que, pour calculer le produit $A \odot B$ de deux matrices booléennes A et B de taille n, on dispose d'un algorithme dont la complexité en temps est $O(n^\omega)$. On ne demande pas de code ni de pseudo-code. ◇

Solution. Soit A et B deux matrices à coefficients booléens. Puisque $\mathbb{B}$ est inclus dans $\mathbb{Z}$, on peut les considérer comme deux matrices à coefficients entiers positifs ou nuls. On calcule alors le produit $A \cdot B$, ce qui donne à nouveau une matrice à coefficients entiers positifs ou nuls, puis on projette le résultat de $\mathbb{Z}$ dans $\mathbb{B}$, via la fonction g qui à 0 associe 0 et à tout entier strictement positif associe 1.

Si l'on souhaite justifier de façon détaillée le fait que ce calcul est correct, on note que pour tout $x \in \mathbb{B}$, on a :

$$g(x) = x$$

On note, de plus, que les opérations sur les entiers produisent, modulo g, le même résultat que les opérations sur les booléens. Pour tous $x, y \in \mathbb{Z}$, on a :

$$g(x + y) = g(x) \vee g(y)$$
$$g(x \times y) = g(x) \wedge g(y)$$

Il en découle que

$$\begin{aligned} g((A \cdot B)_{ij}) &= g(\textstyle\sum_k A_{ik} \times B_{kj}) \\ &= \textstyle\bigvee_k A_{ik} \wedge B_{kj} \\ &= (A \odot B)_{ij} \end{aligned}$$

La complexité de cet algorithme est $O(n^\omega)+O(n^2)$, où le terme $O(n^\omega)$ est le coût de la multiplication $A \cdot B$ et le terme $O(n^2)$ est le coût de la projection g, qui doit être appliquée à chacun des coefficients. D'après la réponse à la question 6, on a $2 \leq \omega$, donc cette complexité s'écrit plus simplement $O(n^\omega)$. □

On s'intéresse maintenant à des graphes orientés dont les n sommets sont les entiers de l'intervalle $[0, n-1]$. On représente un tel graphe A par sa **matrice d'adjacence**, également notée A. Il s'agit d'une matrice carrée de taille n dont le coefficient A_{ij} vaut 1 s'il existe une arête du sommet i vers le sommet j et 0 dans le cas contraire.

Question 8 Pour cette question, on se place dans un semi-anneau quelconque, par exemple $(\mathbb{Z}, +, \times)$ ou $(\mathbb{B}, \vee, \wedge)$, et on considère des matrices à coefficients dans ce semi-anneau. Donnez (sous forme de pseudo-code précis) un algorithme efficace qui, étant donnés une matrice A et un entier $k \geq 0$, calcule la matrice A^k, c'est-à-dire « A à la puissance k ». Quelle est sa complexité en temps ? ◇

Solution. Notons Id la matrice identité et $A \cdot B$ la multiplication des matrices A et B, au sens du semi-anneau dans lequel nous nous sommes placés. L'algorithme classique « d'exponentiation rapide » s'écrit sous forme récursive, avec analyse par cas de l'exposant k :

fonction POWER(A, k)
  si $k = 0$ **alors renvoyer** Id
  sinon { **soit** B = POWER($A, \lfloor k/2 \rfloor$)
  si k est pair **alors renvoyer** $B \cdot B$
  sinon renvoyer $B \cdot B \cdot A$

On peut également l'écrire ainsi :

fonction POWER(A, k)
  si $k = 0$ **alors renvoyer** Id
  sinon { **soit** B = POWER($A \cdot A, \lfloor k/2 \rfloor$)
  si k est pair **alors renvoyer** B
  sinon renvoyer $B \cdot A$

Parce que k est divisé par deux à chaque appel récursif, le nombre d'appels récursifs requis est $O(\log k)$. À chaque étage, une ou deux multiplications de matrices de taille n sont effectuées. La complexité en temps de cet algorithme est donc $O(n^\omega \log k)$.

Note du correcteur. On a rencontré souvent l'erreur qui consiste à effectuer deux appels récursifs, en écrivant par exemple POWER($A, \lfloor k/2 \rfloor$) · POWER($A, \lfloor k/2 \rfloor$) dans le cas où k est pair. Un tel code produit un résultat correct, mais n'a pas la bonne complexité : sa complexité devient $O(k)$ au lieu de $O(\log k)$. Il faut, comme ci-dessus, effectuer un seul appel à POWER($A, \lfloor k/2 \rfloor$), en nommer le résultat B, puis calculer $B \cdot B$.

Certains ont voulu éviter cet écueil en utilisant la mémoisation. Toutefois,
— les détails étaient souvent flous : même en pseudo-code, on n'admet pas l'utilisation d'un mot-clef magique « remember » dont la signification précise n'est pas définie ;
— il aurait fallu préciser quelle structure de données était employée pour la mémoisation ; par exemple, un tableau ne convenait pas, car le coût de son initialisation aurait été $O(k)$;
— enfin et surtout, la solution la plus simple ne nécessite pas de mémoisation, et nous considérons donc son utilisation comme une erreur. □

Question 9 Étant donné un graphe A, on souhaite déterminer, pour tous les sommets i et j, s'il existe dans A un chemin de i vers j. En vous appuyant sur vos réponses aux questions précédentes, indiquez comment effectuer efficacement ce calcul. On ne demande pas de code ni de pseudo-code. Justifiez brièvement pourquoi ce calcul est correct et quelle est sa complexité en temps. ◇

Solution. La matrice A est à coefficients booléens. On peut dire que le problème est de calculer sa clôture transitive $A^\star$, qui est également une matrice à coefficients booléens.

Or, d'un sommet i vers un sommet j,

1. il existe dans le graphe A un chemin quelconque si et seulement s'il existe dans le graphe A un chemin de longueur au plus n ;
2. il existe dans le graphe A un chemin de longueur au plus n si et seulement s'il existe dans le graphe $Id \vee A$ un chemin de longueur exactement n ;

3. il existe dans le graphe $Id \vee A$ un chemin de longueur exactement n si et seulement s'il existe dans le graphe $(Id \vee A)^n$ une arête.

Ces remarques, combinées, montrent que la matrice $A^\star$ est égale à $(Id \vee A)^n$, où le produit de matrices est compris au sens du semi-anneau $(\mathbb{B}, \vee, \wedge)$.

À l'aide de l'algorithme de multiplication de matrices à coefficients booléens (question 7) et de l'algorithme d'exponentiation rapide (question 8), ce calcul peut être effectué en temps $O(n^\omega \log n)$.

On peut comparer l'efficacité de cet algorithme avec celle de l'algorithme de Floyd et Warshall, qui permet également de calculer la clôture transitive d'une matrice booléenne, et dont la complexité en temps est $O(n^3)$. Parce que $\omega < 3$ (question 6), l'algorithme proposé ici est en principe préférable à l'algorithme de Floyd et Warshall lorsque n tend vers l'infini. Toutefois, en pratique, l'algorithme de Floyd et Warshall peut rester préférable parce qu'il est plus simple est moins coûteux pour de petites valeurs de n.

Fischer et Meyer (1971) ont obtenu un résultat meilleur : il est en fait possible de calculer $A^\star$ en temps $O(n^\omega)$. Pour cela, on se ramène d'abord au cas d'un graphe acyclique, et dans ce cas, quitte à re-numéroter les sommets, on peut supposer que la matrice A est triangulaire supérieure. Puis, sous cette hypothèse, on propose un algorithme récursif pour calculer $A^\star$, basé sur une décomposition en quatre sous-blocs. Voir leur article, « **Boolean matrix multiplication and transitive closure** ».

Note du correcteur. Une erreur fréquente consistait à calculer A^n au lieu de $(Id \vee A)^n$. Cette matrice donnait les chemins de longueur exactement n, et non pas les chemins de longueur au plus n. □

2 Calcul des distances dans un graphe non orienté connexe

On s'intéresse maintenant à un graphe A **non orienté** dont les n sommets sont les entiers de l'intervalle $[0, n-1]$. Sa matrice d'adjacence A est donc symétrique : $A_{ij} = A_{ji}$. On suppose de plus que cette matrice est irréflexive : $A_{ii} = 0$. On suppose enfin que le graphe A est **connexe**.

On appelle distance de i à j la longueur (c'est-à-dire le nombre d'arêtes) du plus court chemin dans A de i vers j. On note cette distance $\hat{A}_{ij}$. On souligne que, puisque le graphe A est connexe, il existe au moins un chemin entre i et j, donc $\hat{A}_{ij}$ est un entier. On souligne que la distance d'un sommet à lui-même est nulle : $\hat{A}_{ii} = 0$.

On appelle **diamètre** du graphe A, et on note $\delta(A)$, la plus grande distance entre deux sommets : $\delta(A) = \max_{i,j} \hat{A}_{ij}$.

Pour calculer toutes les distances entre sommets dans le graphe A, on souhaite se ramener à un graphe B, défini ainsi :

$$\begin{array}{ll} B_{ij} = 1 & \text{si } \hat{A}_{ij} = 1 \text{ ou } \hat{A}_{ij} = 2 \\ B_{ij} = 0 & \text{sinon} \end{array}$$

En termes imagés, à chaque fois que deux sommets i et j sont situés à distance 2 l'un de l'autre dans A, il existe dans B un « raccourci », c'est-à-dire une arête entre i et j.

Question 10 Montrez que la matrice B est irréflexive et symétrique. Montrez que le graphe B est connexe. ◇

Solution. Dans le graphe A, la distance d'un sommet à lui-même est nulle : $\hat{A}_{ii} = 0$. Il en découle $B_{ii} = 0$: la matrice B est irréflexive.

Du fait que A est symétrique, on déduit que $\hat{A}$ est symétrique également. Il en découle $B_{ij} = B_{ji}$: la matrice B est symétrique.

Si $A_{ij} = 1$, alors on a $i \neq j$ (parce que A est irréflexive), donc $\hat{A}_{ij} = 1$. Il en découle $B_{ij} = 1$. Par conséquent, le graphe A est inclus dans le graphe B : ce dernier a plus d'arêtes. Or, A est connexe ; il en découle que B est connexe également. □

Question 11 Indiquez comment calculer efficacement la matrice B à partir de la matrice A. Quelle est la complexité de ce calcul ? ◇

Solution. La matrice A étant irréflexive, les sommets i et j situés à distance 1 l'un de l'autre sont ceux reliés par une arête : $\hat{A}_{ij} = 1$ ssi $A_{ij} = 1$.

Par ailleurs, les sommets i et j situés à distance 2 l'un de l'autre sont les sommets i et j distincts tels qu'il existe un chemin de longueur 2 entre i et j : $\hat{A}_{ij} = 2$ ssi $i \neq j$ et $(A \cdot A)_{ij} > 0$. On utilise ici le fait que la matrice $A \cdot A$ compte les chemins de longueur 2 dans le graphe A.

Par conséquent, on a $B_{ij} = 1$ si et seulement si soit $A_{ij} = 1$, soit $i \neq j$ et $(A \cdot A)_{ij} > 0$.

On peut résumer cela par l'équation :

$$B = A \vee (\neg Id \wedge (A \odot A))$$

Cette équation indique comment calculer B. L'opération la plus coûteuse est la multiplication de matrices $A \odot A$, de complexité $O(n^\omega)$; les autres opérations sont de complexité $O(n^2)$.

Note du correcteur. Au lieu de l'équation ci-dessus, on pouvait également écrire :

$$B = (Id \vee A)^2 \wedge \neg Id$$

En effet, cette expression calcule d'abord les chemins dans A de longueur 2 au plus, puis supprime ceux de longueur 0, pour ne conserver que les chemins de longueur 1 ou 2. □

Question 12 Comment, à l'aide des matrices A et B, peut-on déterminer si le diamètre du graphe A est inférieur ou égal à 1 ? Quelle est, dans ce cas, la matrice $\hat{A}$? ◇

Solution. Si le diamètre de A est inférieur ou égal à 1, alors il n'existe dans A aucune paire de sommets situés à distance 2 l'un de l'autre, donc le graphe B est identique au graphe A.

Réciproquement, si les graphes A et B sont identiques, alors il n'existe dans A aucune paire de sommets situés à distance 2 l'un de l'autre, donc aucune paire de sommets situés à distance 2 ou plus l'un de l'autre. Donc, le diamètre de A est inférieur ou égal à 1.

On a donc $\delta(A) = 1$ si et seulement si $A = B$.

De plus, dans ce cas, les plus courts chemins sont simplement les arêtes : on a $\hat{A} = A$.

Note du correcteur. Sachant que le graphe A est connexe et irréflexif, on pouvait remarquer que la condition $\delta(A) = 1$ est en fait équivalente à affirmer que le graphe est complet, i.e., $A = \neg Id$. □

Pour les estimations de distance qui suivent, on travaille exclusivement avec des nombres entiers. En particulier, on note simplement $n/2$ le quotient de la division **euclidienne** de n par 2.

Donc, l'entier $n/2$ est égal à $\lfloor \frac{n}{2} \rfloor$, et l'entier $(n+1)/2$ est égal à $\lceil \frac{n}{2} \rceil$.

Question 13 Montrez que $\hat{A}_{ij} \leq 2\hat{B}_{ij}$. ◇

Solution. Chaque arête dans le graphe B correspond à une ou deux arêtes consécutives dans le graphe A : si $B_{ij} = 1$, alors $\hat{A}_{ij} \in \{1, 2\}$.

Par conséquent, entre deux sommets i et j, s'il existe dans B un chemin de longueur d, alors il existe dans A un chemin de longueur inférieure ou égale à $2d$.

Soient i et j deux sommets. Par définition de $\hat{B}_{ij}$, il existe entre eux dans B un chemin de longueur $\hat{B}_{ij}$. Donc, il existe entre eux dans A un chemin de longueur $2\hat{B}_{ij}$ au plus. Donc, $\hat{A}_{ij} \leq 2\hat{B}_{ij}$. □

Question 14 Montrez que $\hat{B}_{ij} \leq (\hat{A}_{ij} + 1)/2$. ◇

Solution. Si i, k et j sont trois sommets consécutifs le long d'un plus court chemin dans le graphe A, alors $\hat{A}_{ij} = 2$, donc $B_{ij} = 1$: il existe dans le graphe B une arête de i à j.

Par conséquent, entre deux sommets i et j, s'il existe dans A un plus court chemin π de longueur d, alors il existe dans B un chemin de longueur égale à $(d + 1)/2$, obtenu en groupant deux par deux les arêtes de π.

(Si d est pair, alors $(d + 1)/2$ est égal à $d/2$: les arêtes sont groupées deux par deux. Si d est impair, alors $(d + 1)/2$ est égal à $d/2 + 1$: les arêtes sont groupées deux par deux, sauf une, qui reste isolée.)

Donc, $\hat{B}_{ij} \leq (\hat{A}_{ij} + 1)/2$. □

Question 15 Donnez une équation qui caractérise $\hat{B}_{ij}$ en fonction de $\hat{A}_{ij}$. Inversement, peut-on déduire $\hat{A}_{ij}$ de $\hat{B}_{ij}$? ◇

Solution. La question 13 implique que le nombre rationnel $\frac{\hat{A}_{ij}}{2}$ est inférieur ou égal au nombre entier $\hat{B}_{ij}$, donc que le nombre entier $(\hat{A}_{ij} + 1)/2$ est inférieur ou égal à $\hat{B}_{ij}$.

En combinant ceci avec le résultat de la question 14, on obtient $\hat{B}_{ij} = (\hat{A}_{ij} + 1)/2$.

Inversement, $\hat{A}_{ij}$ est égal soit à $2\hat{B}_{ij}$, soit à $2\hat{B}_{ij} - 1$. Cependant, on ne voit pas, a priori, comment distinguer dans lequel de ces deux cas on se trouve. De fait, on ne peut pas déduire $\hat{A}$ à partir de $\hat{B}$ seule. Voici, par exemple, deux graphes A légèrement différents :

Dans chacun de ces deux graphes, tous les sommets sont à distance 1 ou 2 les uns des autres. Dans les deux cas, le graphe B est donc le même, à savoir le graphe irréflexif complet sur $\{a, b, c\}$. Pourtant, la matrice $\hat{A}$ n'est pas la même dans les deux cas. On ne peut pas déduire $\hat{A}$ de $\hat{B}$. □

Question 16 Donnez une relation qui caractérise $\delta(B)$ en fonction de $\delta(A)$. Déduisez-en que, si $\delta(A) > 1$, alors $\delta(B) < \delta(A)$. ◇

Solution. On a :

$$\delta(B) = \max_{i,j} \hat{B}_{ij} = \max_{i,j}((\hat{A}_{ij} + 1)/2) = ((\max_{i,j} \hat{A}_{ij}) + 1)/2 = (\delta(A) + 1)/2$$

Le diamètre de B est donc la moitié de celui de A, arrondi supérieurement. Il en découle immédiatement que, si $\delta(A) > 1$, alors $\delta(B) < \delta(A)$. □

Question 17 Soient i et j deux sommets distincts. Montrez que, pour tout voisin k de j dans le graphe A, on a $\hat{A}_{ij} - 1 \leq \hat{A}_{ik} \leq \hat{A}_{ij} + 1$. Existe-t-il au moins un voisin k de j telle que l'inégalité de gauche est une égalité ? Même question pour l'inégalité de droite. ◇

Solution. Pour aller de i à k, on peut suivre un plus court chemin de i à j, puis emprunter une arête de j à k. Donc, $\hat{A}_{ik} \leq \hat{A}_{ij} + 1$.

Le même argument, où l'on échange les rôles de j et k, donne $\hat{A}_{ij} \leq \hat{A}_{ik} + 1$, donc $\hat{A}_{ij} - 1 \leq \hat{A}_{ik}$. On s'appuie sur le fait que le graphe est non orienté : l'arête qui relie j et k peut être empruntée dans les deux sens.

Supposons i et j distincts. Considérons un plus court chemin de i à j, et nommons k l'avant-dernier sommet de ce chemin. Le sommet k est voisin de j, et on a $\hat{A}_{ik} + 1 = \hat{A}_{ij}$. L'inégalité de gauche est alors une égalité.

L'inégalité de droite peut être stricte pour tous les voisins k de j. Si par exemple on choisit i et j de sorte que $\hat{A}_{ij} = \delta(A)$, alors clairement $\hat{A}_{ik}$ ne peut pas être strictement supérieur à $\hat{A}_{ij}$. □

Question 18 Soient i et j deux sommets distincts, et k un voisin de j dans le graphe A. Déduisez de la question précédente des inégalités reliant $\hat{B}_{ik}$ et $\hat{B}_{ij}$. Vous distinguerez le cas où $\hat{A}_{ij}$ est pair et le cas où $\hat{A}_{ij}$ est impair. ◇

Solution. D'après la question 17, pour tout voisin k de j dans A, on a :

$$\hat{A}_{ij} - 1 \leq \hat{A}_{ik} \leq \hat{A}_{ij} + 1$$

Si l'on ajoute 1 à chaque membre, on obtient :

$$\hat{A}_{ij} \leq \hat{A}_{ik} + 1 \leq \hat{A}_{ij} + 2$$

et si l'on divise chaque membre par 2, on obtient :

$$\hat{A}_{ij}/2 \leq \hat{B}_{ik} \leq \hat{A}_{ij}/2 + 1$$

On a utilisé la question 15 pour remplacer $(\hat{A}_{ik} + 1)/2$ par $\hat{B}_{ik}$.

Si $\hat{A}_{ij}$ est pair, alors toujours à l'aide du résultat obtenu lors de la question 15, on peut écrire :

$$\hat{B}_{ij} \leq \hat{B}_{ik} \leq \hat{B}_{ij} + 1$$

Si $\hat{A}_{ij}$ est impair, les inégalités deviennent :

$$\hat{B}_{ij} - 1 \leq \hat{B}_{ik} \leq \hat{B}_{ij}$$

Comme on l'a noté lors de la question 17, l'inégalité de gauche est atteinte pour un certain k. □

On note $degré_A(j)$ le nombre de voisins du sommet j dans A.

Question 19 Soient i et j deux sommets distincts. Montrez que si $\hat{A}_{ij}$ est pair, alors on a :

$$\left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik} \right) \geq \hat{B}_{ij} \times degré_A(j)$$

et que si $\hat{A}_{ij}$ est impair, alors on a au contraire :

$$\left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik} \right) < \hat{B}_{ij} \times degré_A(j)$$

◇

Solution. Supposons d'abord que $\hat{A}_{ij}$ est pair. On somme sur tous les k, voisins de j dans A, l'inégalité $\hat{B}_{ij} \leq \hat{B}_{ik}$ obtenue lors de la question 18. On obtient donc :

$$\left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ij} \right) \leq \left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik} \right)$$

Puisque $\hat{B}_{ij}$ ne dépend pas de k, le terme de gauche se simplifie, et on obtient :

$$\hat{B}_{ij} \times degré_A(j) \leq \left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik} \right)$$

Supposons maintenant que $\hat{A}_{ij}$ est impair. On somme sur tous les k, voisins de j dans A, l'inégalité $\hat{B}_{ik} \leq \hat{B}_{ij}$ obtenue lors de la question 18. De plus, on a noté que, pour au moins une valeur de k, cette inégalité est stricte. On obtient donc :

$$\left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik} \right) < \hat{B}_{ij} \times degré_A(j)$$

□

Question 20 On définit la matrice S par $S_{ij} = \left(\sum_{k \text{ voisin de } j \text{ dans } A} \hat{B}_{ik}\right)$. Comment, à partir des matrices A et $\hat{B}$, peut-on calculer facilement et efficacement la matrice S ? ◇

Solution. On a $S_{ij} = \sum_k \hat{B}_{ik} A_{kj}$, puisque A_{kj} vaut 1 si k est voisin de j dans A et vaut 0 sinon. Donc, $S = \hat{B} \cdot A$. Une multiplication de matrices suffit. □

Question 21 À l'aide des résultats accumulés lors des questions 10 à 20, définissez (sous forme de pseudo-code précis) un algorithme récursif qui, étant donnée la matrice d'adjacence d'un graphe A irréflexif, non orienté, connexe, calcule la matrice $\hat{A}$. Démontrez que l'algorithme termine, et donnez sa complexité en temps. ◇

Solution. Le principe de l'algorithme est le suivant. On construit d'abord la matrice B, comme indiqué lors de la question 11. On compare ensuite les matrices A et B : si elles sont égales, alors d'après la question 12, $\hat{A}$ est égale à A, et le calcul est terminé. Dans le cas contraire, on sait que l'on a $\delta(A) > 1$ (question 12), d'où $\delta(B) < \delta(A)$ (question 16). On peut donc effectuer un appel récursif, sans crainte que l'algorithme ne termine pas, et l'on obtient la matrice $\hat{B}$ des plus courtes distances dans le graphe B.

fonction DISTANCES(A)
  soit $B = A \vee (\neg Id \wedge (A \odot A))$
  si $A = B$ **alors renvoyer** A
  sinon
    soit $\hat{B}$ = DISTANCES(B)
    soit $S = \hat{B} \cdot A$
    soit d le vecteur $degré_A$
    soit $\hat{A}$ la matrice définie par $\begin{cases} \hat{A}_{ij} = 2\hat{B}_{ij} & \text{si } S_{ij} \geq \hat{B}_{ij} \times d_j \\ \hat{A}_{ij} = 2\hat{B}_{ij} - 1 & \text{sinon} \end{cases}$
    renvoyer $\hat{A}$

Il reste alors à reconstruire la matrice $\hat{A}$ à partir de la matrice $\hat{B}$. On a vu que $\hat{A}_{ij}$ est égal soit à $2\hat{B}_{ij}$, soit à $2\hat{B}_{ij} - 1$ (question 15). On a vu (question 19) comment déterminer dans lesquels de ces deux cas on se trouve : si $S_{ij} \geq \hat{B}_{ij} \times degré_A(j)$ alors on est dans le premier cas ; sinon on est dans le second cas. On a vu comment calculer la matrice S (question 20). On prend soin de pré-calculer le vecteur $degré_A$, afin d'éviter des calculs redondants. On obtient ainsi le pseudo-code ci-dessus.

On a vu (question 16) que, lors de l'appel récursif, on a $\delta(B) = (\delta(A) + 1)/2$. Le diamètre du graphe est donc divisé par deux (arrondi supérieurement) à chaque appel récursif. Or le diamètre du graphe initial est borné par n. Il en découle que le nombre d'appels récursifs imbriqués est $O(\log n)$.

Si l'on exclut le coût de l'appel récursif DISTANCES(B), le coût de l'appel DISTANCES(A) est dominé par les deux opérations de multiplication de matrices. (Le calcul des degrés, la construction de $\hat{A}$, etc. ont un coût $O(n^2)$.) Ce coût est donc $O(n^\omega)$.

Il découle des deux paragraphes précédents que la complexité en temps de l'algorithme est $O(n^\omega \log n)$.

L'algorithme obtenu ici pour le calcul de $\hat{A}$ est dû à Seidel (« **On the All-Pairs-Shortest-Path problem in unweighted undirected graphs** », 1995). Il a la même complexité asymptotique que

celui proposé pour le calcul de $A^\star$ (questions 8 et 9), et la structure générale des deux algorithmes est la même. On peut considérer le second comme une généralisation du premier. □