Algorithmique & Programmation (INF431)
Contrôle des connaissances CC2
CORRIGÉ

28 juin 2013

Ce sujet est composé de trois parties entièrement **indépendantes** les unes des autres. La couleur des copies utilisées est indifférente.

Première partie

Rendu de monnaie

Dans cette partie, on demande d'écrire **du pseudo-code** aussi clair et rigoureux que possible.

Dans cette partie, on considère qu'un nombre entier occupe en mémoire un espace $O(1)$. On considère de plus que les opérations arithmétiques usuelles (à savoir addition, soustraction, multiplication, division euclidienne) ont une complexité en temps $O(1)$.

Une banque centrale a mis en circulation n types de pièces de monnaie, dont les valeurs faciales sont notées $v_1, \ldots, v_n$. Quitte à adopter une unité appropriée, on suppose que ces valeurs sont des entiers strictement positifs. On fait en sorte que ces valeurs forment une suite strictement croissante. Enfin, on suppose $n \geq 1$ et $v_1 = 1$. Dans la suite, on suppose la suite $v_1, \ldots, v_n$ fixée et connue.

Par exemple, la « zone euro » a mis en circulation des pièces de 1, 2, 5, 10, 20, 50, 100, et 200 centimes. On ignore les billets et on prend le centime comme unité. On a donc $n = 8$ et les nombres précédents forment la suite $v_1, \ldots, v_8$.

Si s est un entier positif ou nul, on note $m(s)$ le nombre minimal de pièces (de tous types) nécessaires pour former la somme s. Par exemple, dans la zone euro, $m(9)$ est égal à 3, car on peut obtenir 9 centimes à l'aide de 3 pièces (de 2, 2, et 5 centimes) et il est impossible d'obtenir cette somme à l'aide de deux pièces seulement.

Le problème du *rendu de monnaie* consiste, étant donné s, à calculer $m(s)$.

On cherche d'abord un algorithme simple et efficace capable d'exhiber **une** manière, **pas nécessairement optimale**, d'atteindre la somme s.

Question 1 Proposez un algorithme $\mathcal{G}$ qui, étant donné un entier positif ou nul s, produit un entier $\mathcal{G}(s)$ tel qu'il soit possible de former la somme s à l'aide de $\mathcal{G}(s)$ pièces de tous types. On demande que la complexité asymptotique en temps de cet algorithme, exprimée en fonction de n et s, soit $O(n)$. Ceci implique que cette complexité doit être **indépendante de** s. On souhaite que, dans le cas du système de pièces de la zone euro, cet algorithme produise toujours une solution optimale, i.e., $\mathcal{G}(s) = m(s)$. Cependant, on ne demande pas de démontrer cette affirmation. ⋄

Solution. L'idée est de proposer un algorithme glouton, qui s'approche d'abord aussi près de s que possible à l'aide de la pièce dont la valeur est la plus grande, à savoir v_n, puis utilise ensuite les pièces de valeur inférieure. On peut penser que c'est l'algorithme utilisé intuitivement par les humains.

On peut exprimer cet algorithme sous forme récursive. On écrit une fonction $\mathcal{G}\textsc{Rec}$ paramétrée par deux entiers s et i, où s est la somme à atteindre et où l'intervalle $[1, i]$ indique quels types de pièces il est permis d'utiliser. $\mathcal{G}(s)$ est alors $\mathcal{G}\textsc{Rec}(s, n)$.

fonction $\mathcal{G}\textsc{Rec}(s, i)$
  si $i = 0$ **alors**
    renvoyer 0
  sinon
    soit $k = \lfloor s/v_i \rfloor$
    afficher « rendre k pièces de type i »
    renvoyer $k + \mathcal{G}\textsc{Rec}(s - kv_i, i - 1)$

Notons que, lorsque i vaut 0, il faut que s soit nul également, sans quoi il est impossible d'atteindre s. On peut dire que $i = 0 \Rightarrow s = 0$ est une précondition de la fonction $\mathcal{G}\textsc{Rec}$. L'appel initial $\mathcal{G}\textsc{Rec}(s, n)$ satisfait cette propriété car n est non nul, et l'appel récursif $\mathcal{G}\textsc{Rec}(s - kv_i, i - 1)$ la satisfait également car si $i - 1$ vaut 0, alors i vaut 1, donc v_i vaut 1, donc k vaut s, et $s - kv_i$ vaut 0.

On peut aussi écrire cet algorithme sous forme itérative. Au début de chaque itération, les variables s, i et c indiquent respectivement la somme restant à produire, les types de pièces qu'il est encore permis d'utiliser, et le nombre de pièces utilisées jusqu'ici.

fonction $\mathcal{G}(s)$
  $i \leftarrow n$
  $c \leftarrow 0$
  tant que $i \geq 1$ **faire**
    soit $k = \lfloor s/v_i \rfloor$
    afficher « rendre k pièces de type i »
    $s \leftarrow s - kv_i$
    $i \leftarrow i - 1$
    $c \leftarrow c + k$
  renvoyer c

La complexité en temps de l'algorithme est bien, pour les deux formulations, $O(n)$. En effet, il faut n appels récursifs, ou n itérations ; et à chaque appel, ou à chaque itération, on effectue un nombre fixe d'opérations arithmétiques.

La complexité en espace de la version récursive est $O(n)$, à cause de la pile implicite qu'elle exige. La complexité en espace de la version itérative est $O(1)$, car l'algorithme n'a besoin que d'un nombre fixe de variables. Cette analyse n'était pas demandée. □

Question 2 Ajoutez à votre algorithme $\mathcal{G}$ des instructions **afficher** « rendre k pièces de type i » (où $k \geq 0$ et $1 \leq i \leq n$) de façon à ce que l'algorithme affiche la manière dont il atteint la somme s. On permet, si besoin, que deux instructions « **afficher** » concernent un même type i. ◇

Solution. Voir la solution à la question 1, où les instructions d'affichage apparaissent déjà. □

Question 3 Votre algorithme $\mathcal{G}$ produit-il toujours le résultat optimal $m(s)$, quelle que soit la suite $(v_1, \ldots, v_n)$? Justifiez votre réponse. ◇

Solution. Posons $n = 3$ et $(v_1, v_2, v_3) = (1, 3, 4)$. On constate que pour atteindre la somme 6 l'algorithme glouton propose d'employer 3 pièces ($4 + 1 + 1 = 6$), alors que 2 pièces suffisent ($3 + 3 = 6$). L'algorithme glouton n'est donc pas optimal dans ce cas.

On peut démontrer que, pour certaines suites de valeurs faciales bien choisies, comme la suite « 1, 2, 5, 10, 20, 50, 100, 200 », l'algorithme glouton est optimal. Les systèmes de pièces qui satisfont cette propriété sont appelés canoniques. La quasi-totalité des systèmes utilisés dans le monde sont bien sûrs canoniques. □

On cherche maintenant un algorithme capable d'exhiber une manière **optimale** d'atteindre la somme s.

Question 4 Proposez un algorithme $\mathcal{M}$ qui, étant donné un entier positif ou nul s, calcule $m(s)$. Indiquez quelle est sa complexité asymptotique en temps et en espace en fonction de n et s. On exige que cette complexité soit polynomiale, c'est-à-dire de la forme $O(n^a s^b)$ pour des constantes a et b bien choisies. ◇

Solution. On utilise la programmation dynamique : identifier une famille de sous-problèmes, exprimer les équations qui relient ces sous-problèmes, puis résoudre ces problèmes dans un ordre approprié.

Pour $i \in [0, n]$, notons $m(s, i)$ le nombre minimal de pièces nécessaires pour former la somme s lorsque seules les pièces de types 1 à i sont disponibles.

Pour atteindre une somme nulle, aucune pièce n'est nécessaire. Donc, nous avons :

$$m(0, i) = 0$$

Atteindre une somme non nulle est impossible si l'on ne dispose d'aucune pièce. Donc, nous avons :

$$m(s, 0) = \infty \qquad \text{si } s > 0$$

Enfin, pour atteindre une somme quelconque s, soit on utilise au moins une pièce de type i, auquel cas les autres pièces sont de types 1 à i et doivent constituer une manière optimale d'atteindre la somme $s - v_i$; soit on n'en utilise aucune, auquel cas les pièces utilisées sont de types 1 à $i - 1$. Donc, nous avons :

$$m(s, i) = \min \begin{cases} 1 + m(s - v_i, i) & \text{si } s - v_i \geq 0 \\ m(s, i - 1) & \text{si } i \geq 1 \end{cases}$$

On constate que l'on peut calculer tous les $m(s, i)$, d'abord pour i croissant, puis pour s croissant. (On s'appuie sur le fait que les v_i sont strictement positifs.) On les stocke dans un tableau à deux dimensions. La valeur recherchée, $m(s)$, est égale à $m(s, n)$.

Le pseudo-code peut s'écrire de la façon suivante :

fonction $\mathcal{M}(s)$
  soit m un tableau indicé par $[0, s] \times [0, n]$
  pour $i = 0$ **à** n **faire**
    pour $t = 0$ **à** s **faire**
      si $t = 0$ **alors**
        $m[t][i] \leftarrow 0$
      sinon si $i = 0$ **alors**
        $m[t][i] \leftarrow \infty$
      sinon
        $m[t][i] \leftarrow \min \begin{cases} \textbf{si } t - v_i \geq 0 \textbf{ alors } 1 + m(t - v_i, i) \textbf{ sinon } \infty \\ \textbf{si } i \geq 1 \textbf{ alors } m(t, i - 1) \textbf{ sinon } \infty \end{cases}$
  — on insérera ici le code d'affichage (voir solution à la question 5)
  renvoyer $m[s][n]$

La complexité de l'algorithme en espace est $O(ns)$, c'est-à-dire la taille du tableau. Sa complexité en temps est la même, car il suffit d'un nombre constant d'opérations pour calculer la valeur de chaque case du tableau. □

Question 5 Ajoutez à votre algorithme $\mathcal{M}$ des instructions **afficher** « rendre k pièces de type i » (où $k \geq 0$ et $1 \leq i \leq n$) de façon à ce que l'algorithme affiche une manière optimale d'atteindre la somme s. On permet, si besoin, que deux instructions « **afficher** » concernent un même type i. Cet ajout affecte-t-il la complexité asymptotique en temps ou en espace de l'algorithme ? ◇

Solution. Dans le cas de l'algorithme glouton (question 1), il était possible d'afficher la solution au fur et à mesure qu'elle était calculée. Ici, on ne peut pas faire cela, car on ne sait pas, pendant la construction du tableau m, quel chemin à travers ce tableau correspondra finalement à la solution optimale. On doit attendre que ce tableau soit entièrement rempli pour déterminer, a posteriori, quel est ce chemin. Le pseudo-code qui détermine et affiche ce chemin peut s'écrire de la façon suivante :

```
i ← n
t ← s
tant que vrai faire
    si t = 0 alors
        sortir de la boucle        — on a terminé
    sinon si i = 0 alors
        impossible        — car m[t][i] < ∞
    sinon si t − v_i ≥ 0 et m[t][i] = 1 + m[t − v_i][i] alors
        afficher « rendre 1 pièce de type i »
        t ← t − v_i
    sinon si i ≥ 1 et m[t][i] = m[t][i − 1] alors
        i ← i − 1
    sinon
        impossible        — car m[t][i] < ∞
```

Un invariant de la boucle ci-dessus est $m[t][i] < \infty$. En effet, on a initialement $m[s][n] < \infty$, car on sait que le problème de rendu de monnaie admet une solution ; et à chaque itération, on met à jour t ou i en suivant le chemin qui « explique » pourquoi $m[t][i]$ a reçu cette valeur, donc on préserve la propriété que $m[t][i]$ est fini. Cet invariant permet de démontrer que les instructions « **impossible** » ne peuvent pas être atteintes.

Dans l'écriture de cette boucle, dans un objectif de clarté, nous avons paraphrasé la structure du code qui initialise le tableau m. Cependant, a posteriori, il est possible de simplifier ce code. Certains tests peuvent être supprimés, puisqu'il est impossible qu'ils échouent. De plus, on constate que la boucle termine lorsque la condition $t = 0$ est satisfaite ; on peut donc employer une boucle **tant que** ordinaire, plutôt qu'une boucle infinie munie d'une instruction **sortir** (« *break* »). On pouvait donc écrire également :

```
i ← n
t ← s
tant que t > 0 faire
    si t − v_i ≥ 0 et m[t][i] = 1 + m[t − v_i][i] alors
        afficher « rendre 1 pièce de type i »
        t ← t − v_i
    sinon
        i ← i − 1
```

Le code ci-dessus n'affiche jamais « rendre k pièces de type i » pour $k > 1$. Pour faire un peu mieux, on peut fusionner les instructions d'affichage qui concernent le même type i. Comme l'indice i ne fait que décroître, ces instructions sont nécessairement consécutives, et il est facile de les fusionner. Pour cela, on remplace la construction **si** / **alors** / **sinon** par une boucle **tant que** imbriquée. On pouvait donc écrire également :

$i \leftarrow n$
$t \leftarrow s$
tant que $t > 0$ **faire**
  $k \leftarrow 0$
  tant que $t - v_i \geq 0$ **et** $m[t][i] = 1 + m[t - v_i][i]$ **faire**
    $t \leftarrow t - v_i$
    $k \leftarrow k + 1$
  si $k > 0$ **alors**
    afficher « rendre k pièces de type i »
  $i \leftarrow i - 1$

Quelle que soit l'écriture de cette boucle, elle termine car, à chaque itération, soit t décroît strictement et i est inchangé, soit t est inchangé et i décroît strictement. De plus, cet argument montre que la complexité en temps de cette boucle est $O(n + s)$. Ce coût est dominé par $O(ns)$. À une constante près, l'affichage n'affecte donc pas la complexité asymptotique en temps de l'algorithme. Sa complexité en espace est également inchangée, puisqu'aucune nouvelle structure de données n'est introduite. □

Deuxième partie

Graphes et model-checking

Dans cette partie, on demande d'écrire **du pseudo-code** aussi clair et rigoureux que possible. Vous pourrez vous appuyer sur des structures de données de la librairie standard de Java, à condition de rappeler brièvement ce qu'elles font.

On désigne par *model-checking* une famille de techniques visant à vérifier formellement des systèmes informatiques (schémas de matériel, protocoles, voire programmes) en examinant chaque état du système.

Un système est spécifié sous la forme d'un état initial $\sigma_0 \in \Sigma$ et d'une relation de transition $\tau \subseteq \Sigma^2$. Ici, Σ désigne l'ensemble des états. Un état σ peut consister en un vecteur de k bits (par exemple dans le cas d'un circuit électronique doté de k bits de stockage), mais on peut avoir aussi des entiers ($\mathbb{N}$, $\mathbb{Z}$) ou d'autres types de données comme des chaînes de caractères.

Une *trace* d'exécution est une suite (finie ou infinie) d'états $\sigma_0, \sigma_1, \ldots$ telle que $\forall i, (\sigma_i, \sigma_{i+1}) \in \tau$.

On parle de vérification d'une propriété de *sûreté* quand il s'agit de montrer que certains états indésirables sont inaccessibles.

1 Model-checking énumératif

Nous supposons que tout élément de Σ peut être désigné de façon unique par une chaîne de caractères (autrement dit, deux éléments distincts de Σ sont identifiés par deux chaînes différentes ; en revanche il peut exister des chaînes ne correspondant à aucun élément de Σ). Nous supposerons que nous disposons d'une fonction qui, étant donné $\sigma \in \Sigma$, renvoie un tableau des états *successeurs* de σ, c'est-à-dire un tableau contenant les états $\sigma' \in \Sigma$ tels que $(\sigma, \sigma') \in \tau$.

Nous programmons cela en Java ainsi :

```
interface Etat {
  String identifiant();
```

```
  Etat[] successeurs();
}
```

La méthode `identifiant()` renvoie l'identifiant de l'état (rappel : deux états différents ont des identifiants différents). La méthode `successeurs()` renvoie un tableau contenant les successeurs de l'état (dans un ordre quelconque).

Dans un premier temps, **nous ne faisons aucune autre hypothèse sur l'ensemble Σ, qui peut être infini**.

Question 6 Donnez une fonction :

```
static boolean estAccessible (Etat initial, String destination)
```

Cette fonction doit satisfaire les propriétés suivantes :

1. S'il existe une trace $\sigma_0, \ldots, \sigma_n$ telle que σ_0 est l'état `initial` et σ_n est un état désigné par l'identifiant `destination`, alors `estAccessible(initial, destination)` **doit terminer** et renvoyer «vrai».
2. S'il n'existe pas de telle trace, alors `estAccessible(initial, destination)` peut terminer et renvoyer «faux», ou ne pas terminer. On impose cependant qu'il termine forcément si Σ est fini. ◇

Solution. Un parcours en profondeur d'abord ne convient pas : en effet, il peut très bien partir dans une direction infructueuse et ignorer une solution. Prenons par exemple un système d'ensemble d'états $\Sigma = \mathbb{N} \cup \{F\}$, d'état initial 0, dont la relation de transition τ est $\{(i, i+1) \mid i \in \mathbb{N}\} \cup \{(0, F)\}$ et pour lequel on veut tester l'accessibilité de l'état F. Un parcours en profondeur d'abord peut, s'il choisit la transition $0 \to 1$, partir dans une branche infinie $0 \to 1 \to 2 \ldots$, alors que s'il avait choisi la branche $0 \to F$ il aurait immédiatement terminé en répondant «vrai».

Nous utiliserons donc plutôt un parcours en largeur d'abord, qui va trouver une trace de longueur minimale. Il faut marquer les états déjà parcourus, afin d'éviter de «boucler» : pour cela nous utiliserons une table de hachage pour mémoriser les identifiants des états déjà vus.

```
static boolean estAccessible(Etat initial, String destination) {
  Queue<Etat> file = new LinkedList<Etat>();
  AbstractSet<String> hash = new HashSet<String>();
  file.offer(initial);
  hash.add(initial.identifiant());
  while(file.peek() != null) {
    Etat courant = file.remove();
    if (courant.identifiant().equals(destination)) return true;
    for(Etat successeur : courant.successeurs()) {
      if (!hash.contains(successeur.identifiant())) {
        file.offer(successeur);
        hash.add(successeur.identifiant());
      }
    }
  }
  return false;
}
```

Nous examinons maintenant le cas où non seulement Σ peut être infini, mais de plus **l'ensemble des successeurs d'un état peut être infini**. Nous devons donc modifier la signature de la méthode `successeurs()`.

```
interface Etat {
  String identifiant();
  Iterator<Etat> successeurs();
}
```

La méthode `successeurs()` fournit maintenant un *itérateur* sur les successeurs de l'état, c'est-à-dire un objet qui permet de les énumérer (dans un ordre quelconque). D'après la définition de l'interface `Iterator<Etat>`, les deux méthodes offertes par un itérateur sont :

1. `boolean hasNext()` indique si oui ou non il y a encore au moins un successeur ;
2. `Etat next()` obtient le prochain successeur (ou lève une exception s'il n'y en a plus).

Question 7 Même question que la précédente, dans ce nouveau cas où l'ensemble des successeurs d'un état peut être infini. Nous vous rappelons que l'algorithme **doit terminer lorsqu'il existe une trace vers la destination**, même si certains sommets ont un nombre infini de successeurs. Expliquez le fonctionnement de l'algorithme. ◇

Solution. Il est tentant de prendre le programme précédent et de l'adapter en changeant le parcours de tableau en parcours de l'itérateur des successeurs :

```
static boolean estAccessibleIncorrect(Etat initial, String destination) {
  Queue<Etat> file = new LinkedList<Etat>();
  AbstractSet<String> hash = new HashSet<String>();
  file.offer(initial);
  hash.add(initial.identifiant());
  while(file.peek() != null) {
    Etat courant = file.remove();
    System.out.println(courant.identifiant());
    if (courant.identifiant().equals(destination)) return true;

    // ajout des successeurs au bout de la file
    Iterator<Etat> successeurs = courant.successeurs();
    while (successeurs.hasNext()) {
      Etat successeur = successeurs.next();
      if (!hash.contains(successeur.identifiant())) {
        file.offer(successeur);
        hash.add(successeur.identifiant());
      }
    }
  }
  return false;
}
```

Ce programme **ne fonctionne cependant pas correctement** s'il rencontre un état dont la liste de successeurs est infinie, car dans ce cas il va boucler en ajoutant cette liste à la file !

Nous suggérons une méthode plus subtile. La file contient non plus des états, mais des itérateurs ; chaque itérateur permet de parcourir les fils d'un état. Au début, on met dans la file l'itérateur sur les fils du état initial (on traite spécialement le cas où le état initial est aussi le état destination). Lorsque l'on récupère un itérateur dans la file, soit il est épuisé (`hasNext()==false`) et on le jette, soit on le fait avancer en récupérant le état `courant` et on le remet au bout de la file, ainsi qu'un itérateur sur les fils du état `courant`.

```
static boolean estAccessible(Etat initial, String destination) {
  if (initial.identifiant().equals(destination)) return true;
```

```
    Queue<Iterator<Etat>> file = new LinkedList<Iterator<Etat>>();
    AbstractSet<String> hash = new HashSet<String>();
    file.offer(initial.successeurs());
    hash.add(initial.identifiant());
    while(file.peek() != null) {
      Iterator<Etat> courantSuccesseur = file.remove();
      if (courantSuccesseur.hasNext()) {
        file.offer(courantSuccesseur);
        Etat courant = courantSuccesseur.next();
        if (!hash.contains(courant.identifiant())) {
          hash.add(courant.identifiant());
          System.out.println(courant.identifiant());
          if (courant.identifiant().equals(destination)) return true;
          file.offer(courant.successeurs());
        }
      }
    }
    return false;
  }
```

C'est presque (mais pas exactement) un parcours en largeur d'abord sur le graphe dont les sommets sont Σ, et dont les arêtes sont construites ainsi : pour tout σ qui a des successeurs $\sigma'_1, \sigma'_2, \ldots$ dans l'ordre renvoyé par l'itérateur σ.`successeurs()`, mettre une arête (σ, σ'_1) et pour tout i une arête $(\sigma'_i, \sigma''_{i+1})$.

Plus précisément, c'est un parcours en largeur d'abord sur le graphe dont les sommets sont les couples $(\sigma, \sigma') \in \tau$, et dont les arêtes sont construites ainsi : pour tout (σ, σ') tel que σ' a des successeurs $\sigma''_1, \sigma''_2, \ldots$ dans l'ordre renvoyé par l'itérateur σ'.`successeurs()`, mettre une arête $\Big((\sigma, \sigma'), (\sigma', \sigma''_1)\Big)$ et pour chaque i mettre une arête $\Big((\sigma', \sigma''_i), (\sigma', \sigma''_{i+1})\Big)$. L'algorithme termine dès qu'il tombe sur un sommet (σ, σ') avec $\sigma =$ `destination`. Par ailleurs, comme tous les sommets (σ, σ') de même σ' ont les mêmes successeurs, nous considérons qu'ils sont tous équivalents (on dit parfois *bisimilaires*) et que parcourir l'un revient à tous les parcourir.

Notons qu'avec cette méthode, l'algorithme ne va pas forcément trouver la trace la plus courte. Ce serait de toute façon impossible sans des hypothèses plus fortes : même pour savoir s'il existe une trace de longueur 1, l'algorithme devra tester tous les successeurs du état initial, qui peuvent être en nombre infini et ne pas contenir l'état destination ! □

2 Model-checking implicite

On suppose maintenant que Σ est de la forme $\{0,1\}^k$, pour un entier k fixé. (Donc, en particulier, Σ **est fini**.) On s'intéresse toujours au graphe orienté dont les sommets sont les états et dont les arêtes sont données par la relation τ.

La relation τ est représentée de façon compacte sous forme d'une fonction T qui attend $\sigma, \sigma' \in \{0,1\}^k$ et qui renvoie 1 si $(\sigma, \sigma') \in \tau$ et 0 sinon.

Nous ne définissons pas ici précisément comment T est représentée. Nous nous contentons de supposer qu'on peut lui associer un entier naturel N, sa «taille», tel que $k \leq N$. Nous supposons qu'évaluer $T(\sigma, \sigma')$ demande un temps $O(N)$ et un espace $O(N)$. Ceci correspond par exemple au cas où T est représentée par un circuit de N portes logiques dont les entrées sont les $2k$ booléens représentant σ et σ'.

Question 8 Comment peut-on programmer une méthode `successeurs` conforme à la première version de l'interface `Etat`, qui précède la question 6 ? On rappelle que cette méthode attend

$\sigma \in \{0,1\}^k$ et renvoie un tableau des $\sigma' \in \{0,1\}^k$ tels que $(\sigma, \sigma') \in \tau$. Il est inutile de de fournir du pseudo-code. ◇

Solution. Il suffit d'énumérer tous les σ' et tester pour chacun $T(\sigma, \sigma')$: si 1, ajouter σ' à la liste.

```
import java.util.Collection;
import java.util.ArrayList;

interface Transitions {
    int dimensionY();
    boolean estUneTransition(boolean[] x, boolean[] y);
}

class ChercheSolutionsBooleennes {
    static void chercheSolutionsRec(Collection<boolean[]> accumulateur,
                        Transitions t,
                        boolean[] x,
                        boolean[] y,
                        int remplissage) {
        assert(remplissage <= y.length);
        if (remplissage == y.length) {
            if (t.estUneTransition(x, y)) accumulateur.add(y.clone());
        } else {
            y[remplissage] = false;
            chercheSolutionsRec(accumulateur, t, x, y, remplissage+1);
            y[remplissage] = true;
            chercheSolutionsRec(accumulateur, t, x, y, remplissage+1);
        }
    }

    static Collection<boolean[]> chercheSolutions(Transitions t,
                                          boolean[] x) {
        Collection<boolean[]> accumulateur = new ArrayList<boolean[]>();
        chercheSolutionsRec(accumulateur, t, x, new boolean[t.dimensionY()], 0);
        return accumulateur;
    }
}
```

Question 9 Quelle est la complexité en temps et en espace des algorithmes classiques de parcours en largeur d'abord et de parcours en profondeur d'abord, lorsqu'on les applique au graphe qui nous intéresse ? Exprimez-la en fonction de k et N. Justifiez. ◇

Solution. Dans les algorithmes classiques de parcours de graphe, pour savoir si l'on a déjà parcouru un état, il faut stocker pour chaque état possible un booléen disant si l'on a déjà parcouru cet état ; ici il y a 2^k états donc une mémoire de 2^k bits. La file d'attente ou la pile peut également atteindre 2^k états stockés ; si chacun est mémorisé avec k bits cela fait $k2^k$ bits. Pour chacun des 2^k états σ éventuellement accessibles, il faut énumérer les 2^k successeurs σ' et payer à chaque fois $O(N)$ en temps et en espace pour tester si $(\sigma, \sigma') \in \tau$. Nous en concluons :

- Espace $O(k2^k + N)$
- Temps $O(N4^k)$ □

Nous voudrions maintenant un algorithme permettant de déterminer s'il existe un chemin d'un état σ à un état σ' et de complexité en espace *polynomiale* (en fait, quadratique) vis-à-vis de N.

Question 10 Donnez un algorithme qui, étant donnés σ, σ', d ($\sigma, \sigma' \in \{0,1\}^k$, $d \in \mathbb{N}$), détermine s'il existe de σ à σ' un chemin de longueur au plus 2^d. Sa complexité en espace doit être $O(dk+N)$. Nous vous suggérons très fortement de procéder par récurrence sur d. Vous pourrez soit donner du pseudo-code, soit fournir une explication claire et précise de ce que l'algorithme fait. Justifiez de sa complexité en espace. ◇

Solution. Pour $d = 0$ il suffit d'appeler $T(\sigma, \sigma')$.

Pour $d > 0$: énumérer tous les $\sigma'' \in \{0,1\}^k$ et se rappeler récursivement sur $\sigma, \sigma'', d-1$ et $\sigma'', \sigma', d-1$; répondre «vrai» si l'on a obtenu «vrai» aux deux appels récursifs pour au moins un σ'' et «faux» sinon.

Cet algorithme exige un espace $O(dk+N)$ parce qu'il faut stocker la pile des d appels récursifs, dont chaque niveau est un état σ'' courant, donc de taille k, et aux feuilles on appelle T, qui exige un espace $O(N)$. □

Question 11 Déduisez-en un algorithme de complexité en espace polynomiale vis-à-vis de N qui, étant donnés deux états σ et σ', détermine s'il existe un chemin de σ à σ'. Justifiez sa complexité. ◇

Solution. Le graphe étant de taille 2^k, s'il y a un chemin entre σ et σ' il y en a un d'au maximum $2^k - 1$ pas, donc on peut se contenter d'appeler la procédure de la question précédente avec une profondeur d'appels $d = k$. La complexité en espace sera donc $O(k^2 + N)$, donc (a fortiori) $O(N^2)$. □

Troisième partie

Barrières

Dans cette partie, on demande d'écrire non pas du pseudo-code, mais **du code Java**.

On s'intéresse à l'algorithme de Floyd et Warshall, dont on souhaite construire une version parallélisée.

Étant donnée une matrice g de dimensions $n \times n$ à coefficients dans $\mathbb{Z} \cup \{\infty\}$, l'algorithme de Floyd et Warshall calcule successivement les matrices d_k, pour k variant de 0 inclus à n inclus, définies par les équations suivantes :

$$\begin{aligned} d_0(i,j) &= g(i,j) \\ d_{k+1}(i,j) &= \min\begin{pmatrix} d_k(i,j) \\ d_k(i,k) + d_k(k,j) \end{pmatrix} \end{aligned}$$

Les opérations + et min sont celles de $\mathbb{Z} \cup \{\infty\}$. Le résultat de l'algorithme est la matrice d_n.

La figure 1 présente une version de cet algorithme en Java. La valeur ∞ est représentée par `Integer.MAX_VALUE`. La fonction `add` effectue une addition dans $\mathbb{Z} \cup \{\infty\}$. La fonction `line` initialise la ligne i de la matrice d_{k+1}. Elle est paramétrée par `src` et `dst`, qui représentent les matrices d_k et d_{k+1}, ainsi que par les entiers `k` et `i`. La fonction principale `fw` alloue d'abord suffisamment d'espace pour stocker les matrices d_k. Ensuite, elle initialise d_0. Enfin, pour k variant de 0 inclus à n exclus, elle calcule d_{k+1} à l'aide des données contenues dans d_k.

On rappelle qu'une matrice de dimension 2 est représentée en Java par un tableau de tableaux. Une matrice à coefficients entiers de taille $m \times n$ est allouée à l'aide de l'expression `new int [m][n]`.

Les questions 12 et 13 concernent deux modifications que l'on souhaite apporter à ce code. Elles sont indépendantes l'une de l'autre.

```
static int add (int x, int y)
{
  if (x == Integer.MAX_VALUE || y == Integer.MAX_VALUE)
    return Integer.MAX_VALUE;
  else
    return x + y;
}

static void line (int[][] src, int[][] dst, int k, int i)
{
  final int n = src.length;
  for (int j = 0; j < n; j++)
    dst[i][j] = Math.min(
      src[i][j],
      add(src[i][k], src[k][j])
    );
}

static int[][] fw (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];

  d[0] = g;

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      line(d[k], d[k + 1], k, i);

  return d[n];
}
```

FIGURE 1 – L'algorithme de Floyd et Warshall

Question 12 Le code de la figure 1 alloue $n + 1$ matrices en mémoire en plus de la matrice `g` qui est donnée initialement. Modifiez la fonction `fw` de la figure 1 pour allouer exactement une matrice en plus de la matrice `g`. Ainsi, l'algorithme modifié utilise exactement deux matrices, dont `g`. Il lui est permis de modifier le contenu de la matrice `g`. Une fois le calcul terminé, il renvoie (l'adresse de) la matrice qui représente d_n.

On souligne que **l'on souhaite bien utiliser deux matrices** et non pas une seule. En d'autres termes, les arguments `src` et `dst` que l'on fournit à la fonction `line` doivent être deux matrices stockées à des emplacements distincts en mémoire. Cela sera nécessaire plus loin. ◇

Solution. Si cette optimisation est possible, c'est parce que, pendant la phase k, l'algorithme consulte uniquement d_k, et écrit dans d_{k+1}. Les matrices $d_{k'}$, pour $k' < k$, ne sont plus utiles. Il suffit donc de stocker en mémoire deux matrices, qui représentent d_k et d_{k+1}. La complexité en espace de l'algorithme passe ainsi de $O(n^3)$ à $O(n^2)$.

Une première possibilité, pour réaliser cela, consiste à allouer un tableau `d` de longueur 2 seulement, et d'appliquer l'opération « modulo 2 » aux indices utilisés pour accéder au tableau `d`. Le code est ainsi très peu différent de celui de la figure 1 :

```
static int[][] fw1a (int[][] g)
```

```
{
  final int n = g.length;
  final int[][][] d = new int [2][][];
  d[0] = g;
  d[1] = new int [n][n];

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      line(d[k % 2], d[(k + 1) % 2], k, i);

  return d[n % 2];
}
```

Une seconde possibilité est de se passer entièrement du tableau `d`. On le remplace par deux variables locales `src` et `dst`, qui chacune contiennent (l'adresse d')une matrice. Initialement `src` vaut `g` et `dst` contient l'adresse d'une matrice nouvellement allouée. Puis, à chaque nouvelle phase, on échange les rôles des deux matrices, simplement en échangeant les adresses contenues dans les variables locales `src` et `dst`.

```
static int[][] fw1b (int[][] g)
{
  final int n = g.length;
  int[][] src, dst, tmp;
  src = g;
  dst = new int [n][n];

  for (int k = 0; k < n; k++, tmp = dst, dst = src, src = tmp)
    for (int i = 0; i < n; i++)
      line(src, dst, k, i);

  return src;
}
```

À la fin, la matrice `src` contient le résultat. □

Question 13 On fixe une constante `T`, entier strictement positif. On souhaite que la seconde boucle, où `i` évolue de 0 inclus à n exclus, soit sub-divisée en `T` segments de tailles approximativement égales. Modifiez la fonction `fw` de la figure 1 pour remplacer cette boucle par deux boucles imbriquées, à savoir une première boucle où un indice `t` désigne l'un des `T` segments, puis une seconde boucle où un indice `i` évolue à l'intérieur du segment désigné par `t`.

On ne demande pas de combiner cette modification avec celle que vous avez proposée lors de la question précédente. On demande de repartir de la fonction `fw` de la figure 1. ◇

Solution. On calcule d'abord la largeur des segments, que l'on note `w`. Il faut bien sûr traiter correctement le cas où `n` n'est pas multiple de `T`. Dans ce cas, on décide que les segments seront tous de même taille, sauf le dernier, qui pourra être plus court. On définit donc `w` comme la partie entière supérieure de `n / T`. Cette définition garantit que le produit `T * w` est supérieur ou égal à `n`.

```
static int[][] fw2 (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];
```

```
    d[0] = g;

    final int w =
      n % T == 0 ?
      n / T :
      n / T + 1;

    for (int k = 0; k < n; k++)
      for (int t = 0; t < T; t++)
        for (int i = t * w; i < (t + 1) * w && i < n; i++)
          line(d[k], d[k + 1], k, i);

    return d[n];
  }
```

Cela fait, il reste à exploser la boucle qui concerne `i` en deux boucles imbriquées, qui concernent `t` et `i`. Il doit y avoir `T` segments, donc `t` évolue de 0 inclus à `T` exclus. Ensuite, `i` évolue à l'intérieur du segment d'indice `t`, donc de `t * w` inclus à `(t + 1) * w` exclus. De plus, afin de traiter correctement le cas du dernier segment lorsque `n` n'est pas multiple de `T`, on ajoute la contrainte `i < n`.

Cette transformation de programme, qui peut être effectuée automatiquement par certains compilateurs optimisants, est appelée « *loop tiling* ». Elle est utile ici car, dans la suite, chaque segment va pouvoir être confié à un thread distinct. □

On met maintenant de côté les deux modifications étudiées dans les questions 12 et 13. On y reviendra lors de la question 17.

Pour les questions 14 à 17, par souci de concision, **on propose (et on exige) d'utiliser Java auquel on a ajouté `spawn` et `sync`**. L'instruction `spawn { ... }` crée et lance un nouveau thread. L'instruction `sync` attend la terminaison des threads lancés plus haut par `spawn`. Ainsi, par exemple, le code pseudo-Java suivant :

```
for (int t = 0; t < T; t++)
  spawn { System.out.println("Je suis le thread " + t); }
sync;
```

s'écrirait ordinairement en Java :

```
// Allocate an array of thread identifiers.
Thread[] thread = new Thread [T];
// Create and start the threads.
for (int t = 0; t < T; t++) {
  final int value_of_t = t;
  thread[t] = new Thread (new Runnable () {
    public void run () {
      int t = value_of_t;
      System.out.println("Je suis le thread " + t);
    }
  });
  thread[t].start();
}
// Wait for them to finish.
```

```
for (int t = 0; t < T; t++)
  thread[t].join();
```

On rappelle que chaque thread a ses propres variables locales, qui ne peuvent être ni lues ni modifiées par les autres threads. Dans l'exemple ci-dessus, on voit que le thread créateur (le code à l'extérieur de `spawn`) a une variable locale nommée `t`. Le thread créé (le code à l'intérieur de `spawn`) a également une variable locale nommée `t`, qui en pseudo-Java n'est pas déclarée explicitement. Ces variables sont **distinctes**. Cependant, nous adoptons la convention que la valeur de la première est copiée, au moment où le thread est créé, vers la seconde. C'est pourquoi le code pseudo-Java ci-dessus a un sens, et chaque thread affiche bien son propre identifiant, bien que la variable locale `t` du thread créateur soit modifiée (elle varie de 0 à `T`).

Question 14 Modifiez la fonction `fw` de la figure 1 pour que la boucle interne (qui concerne `i`) soit exécutée en parallèle. À chaque itération de la boucle externe (qui concerne `k`), il faut donc créer `n` threads, dont chacun traite une valeur de `i`, puis attendre leur terminaison. Quelles structures en mémoire sont partagées entre les différents threads ? Justifiez pourquoi votre code ne provoque pas de race condition. ◇

Solution. Seule la boucle interne est modifiée. Le mot-clef `spawn` entoure le corps de la boucle, de sorte que ce n'est plus le thread principal qui traite les `n` valeurs de `i`, mais ce sont threads `n` différents qui traitent chacun une valeur de `i`.

Une fois ces `n` threads lancés, on attend leur terminaison, à l'aide du mot-clef `sync`. On répète cela à chaque itération de la boucle externe.

```
static int[][] fw3 (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];

  d[0] = g;

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++)
      spawn { line(d[k], d[k + 1], k, i); }
    sync;
  }

  return d[n];
}
```

En général, seuls les variables globales et les objets alloués dans le tas sont partagés entre plusieurs threads. Il n'y a pas de variables globales ici. Les objets alloués dans le tas sont les (tableaux qui constituent les) matrices `g` et `d`.

Un examen du code de la fonction `line` montre que le thread auxiliaire numéro `i` accède en lecture à toute la matrice `d[k]` et accède en écriture **uniquement à la `i`-ème ligne** de la matrice `d[k + 1]`. Les matrices `d[k]` et `d[k + 1]` sont distinctes (i.e., stockées à des emplacements distincts en mémoire). On constate donc que deux threads distincts ne peuvent jamais lire/écrire, écrire/lire, ou écrire/écrire au même emplacement. Notons que deux threads distincts peuvent lire/lire au même emplacement, à savoir (l'une des cases de) la matrice `d[k]`, mais par définition, cela ne constitue pas une race condition. □

Le code écrit lors de la question 14 présente un inconvénient. La création et la destruction d'un thread coûtent cher. Or, à chaque itération de la boucle externe, donc pour chaque valeur

```
static int[][] fwpara (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];

  d[0] = g;

  for (int i = 0; i < n; i++)
    spawn {
      for (int k = 0; k < n; k++)
        line(d[k], d[k + 1], k, i);
    }
  sync;

  return d[n];
}
```

FIGURE 2 – Une version modifiée de l'algorithme (question 15)

de `k`, on crée puis on détruit `n` threads. Il semble naïf de détruire `n` threads pour en recréer aussitôt `n` nouveaux ; pourquoi plutôt ne pas les laisser continuer ? Il suffirait alors de créer `n` threads tout au début et de les détruire tout à la fin.

Question 15 La figure 2 présente une version de l'algorithme modifiée pour utiliser `n` threads auxiliaires en tout. Expliquez pourquoi ce code est incorrect. ◇

Solution. Par comparaison avec la fonction `fw` de la figure 1, les boucles internes et externes ont été interverties, et la boucle qui concerne `i`, devenue la boucle externe, a été parallélisée.

Le problème provient du fait que chaque thread auxiliaire exécute la boucle interne, qui concerne `k`, à sa vitesse propre. Si l'un des threads progresse plus vite que les autres, il peut atteindre `k = 10` (par exemple) et se trouver en train de construire l'une des lignes de la matrice `d[11]` à l'aide de toutes les lignes la matrice `d[10]`, alors que d'autres threads n'ont même pas encore atteint `k = 9` et (par conséquent) la matrice `d[10]` n'a pas encore été totalement initialisée. Ainsi, le résultat obtenu n'a en général aucun sens, car certains emplacements de la mémoire sont lus avant d'avoir été initialisés.

On peut aussi remarquer que, si ce code était correct, alors on pourrait effacer les mots-clef `spawn` et `sync`, et on obtiendrait encore un code correct, car un calcul parallèle peut toujours être exécuté de façon séquentielle. Or, si on efface ces mots-clef, on constate que la seule différence entre les figures 1 et 2 est l'interversion des boucles. C'est cette interversion qui est fondamentalement incorrecte.

Enfin, on peut également remarquer que le code de la figure 2 provoque des race conditions, puisqu'un thread peut être en train d'écrire dans `d[k]` au moment où un autre thread est en train d'y lire. □

Afin de réparer ce problème, on se donne un nouveau mécanisme de synchronisation, connu sous le nom de **barrière**. Une barrière est destinée à faciliter la synchronisation entre `n` threads. Intuitivement, elle permet de bloquer ceux qui « arrivent les premiers » à la barrière : elle ne s'ouvre que lorsque les `n` threads sont arrivés, et les débloque alors tous d'un seul coup.

En Java, cette notion peut être représentée par une classe `Barrier` (figure 3) qui offre deux opérations :

1. L'opération **`new`** permet de créer une nouvelle barrière. On fixe alors la valeur de `n`, qui indique combien de threads partageront cette barrière.

```
public class Barrier {
  public Barrier (int n);
  public void await (int phase);
}
```

FIGURE 3 – Squelette de la classe `Barrier`

2. La méthode `await` permet à l'un des `n` threads participants d'indiquer qu'il a « atteint la barrière ». Il est alors bloqué jusqu'à ce que **tous** les threads participants aient appelé `await`. Le paramètre `phase` est utile si la barrière doit servir plusieurs fois. On adopte la convention que lorsqu'un thread atteint la barrière pour la première fois, il appelle `await(0)` ; la fois suivante, `await(1)` ; et ainsi de suite.

Question 16 À l'aide d'une barrière, corrigez le code de la figure 2. ◇

Solution. L'idée est d'empêcher chacun des `n` threads auxiliaires de progresser à sa propre vitesse, sans aucun contrôle, à travers les `n` itérations de la boucle interne (qui concerne `k`). Il est nécessaire que les `n` threads passent au même moment d'une valeur de `k` à la suivante. Ainsi, la matrice `d[k]` ne sera pas lue avant d'avoir été entièrement initialisée, et les race conditions disparaîtront.

Une barrière constitue le mécanisme idéal pour cela. On crée initialement une barrière `b`, qui sera partagée par les `n` threads auxiliaires.

```
static int[][] fw5 (int[][] g)
{
  final int n = g.length;
  final int[][][] d = new int [n + 1][n][n];
  final Barrier b = new Barrier (n);

  d[0] = g;

  for (int i = 0; i < n; i++)
    spawn {
      for (int k = 0; k < n; k++) {
        line(d[k], d[k + 1], k, i);
        b.await(k);
      }
    }
  sync;

  return d[n];
}
```

Le thread auxiliaire `i`, après avoir terminé son travail pour la phase `k`, c'est-à-dire après avoir exécuté l'appel `line(d[k], d[k + 1], k, i)`, signale qu'il a atteint la barrière. Ainsi, il est bloqué jusqu'à ce que tous aient terminé la phase `k`. □

Question 17 Combinez le code que vous avez proposé lors de la question précédente avec les idées des questions 12 et 13 de façon à obtenir une version de l'algorithme qui utilise en tout deux matrices et `T` threads. ◇

Solution. Comme dans la solution à la question 13, on explose la boucle qui concerne `i` pour obtenir deux boucles imbriquées, portant sur les indices `t` et `i`. Ensuite, comme dans la solution à la question 16, on intervertit les boucles portant sur `k` et `t`, et on introduit une barrière pour conserver une synchronisation au niveau de chaque phase `k`.

```
static int[][] fw6 (int[][] g)
{
  final int n = g.length;
  final int w = n % T == 0 ? n / T : n / T + 1;
  final int[][] x = new int [n][n];
  final Barrier b = new Barrier (n);

  for (int t = 0; t < T; t++)
    spawn {
      int[][] tmp, src, dst;
      src = g;
      dst = x;
      for (int k = 0; k < n; k++, tmp = dst, dst = src, src = tmp) {
        for (int i = t * w; i < (t + 1) * w && i < n; i++)
          line(src, dst, k, i);
        b.await(k);
      }
    }
  sync;

  return n % 2 == 0 ? g : x;
}
```

Enfin, on incorpore la solution à la question 12, qui a montré que deux matrices suffisent. Notons que chaque thread a ses propres variables locales `src` et `dst`. Chaque thread met à jour ces variables de la même manière (leurs valeurs sont échangées après chaque passage de la barrière) donc leur valeur est en fait la même dans tous les threads. À la fin, le résultat recherché est stocké soit dans la matrice `g`, soit dans la matrice auxiliaire `x`, suivant la parité de `n`. On ne peut pas utiliser `src` à la dernière ligne du code car il s'agit d'une variable locale de chaque thread auxiliaire.

Contrairement à ce qui est possible dans le cas purement séquentiel, on ne peut pas se contenter d'une seule matrice, car cela impliquerait de lire et d'écrire dans cette même matrice au même moment, ce qui créerait des race conditions. □

On souhaite maintenant construire plusieurs implémentations de la classe `Barrier`, à l'aide de différents mécanismes étudiés pendant le cours.

Pour les questions 18 et 19, on fait l'hypothèse que **la barrière ne sert qu'une fois**. Autrement dit, si une barrière `b` a été créée par `new Barrier (n)`, alors on attend que chacun des `n` threads appelle exactement une fois `b.await(0)`, après quoi la barrière `b` n'est plus utilisée.

On rappelle que la méthode `decrementAndGet` de la classe `AtomicInteger` (figure 4) décrémente le compteur et renvoie sa nouvelle valeur, le tout de façon atomique. De même, `incrementAndGet` incrémente le compteur et renvoie sa nouvelle valeur, le tout de façon atomique. On souligne qu'un objet de classe `AtomicInteger` **peut** être utilisé simultanément par plusieurs threads ; cela n'est pas considéré comme une race condition.

Question 18 À l'aide de la classe `AtomicInteger` (figure 4), proposez une implémentation de la classe `Barrier` (figure 3). L'attente active est autorisée. ◇

Solution. L'idée est simple. Lorsqu'on crée une nouvelle barrière, on alloue un objet de classe `AtomicInteger`, dont on stocke l'adresse dans la barrière. Cet objet va servir de compteur. On choisit de compter, par exemple, le nombre de threads qui n'ont pas encore atteint la barrière. Ce compteur est donc initialisé à la valeur `n`, et doit être décrémenté à chaque fois qu'un nouveau thread atteint la barrière.

```
public class Barrier {

  private final AtomicInteger count;

  public Barrier (int n)
  {
    count = new AtomicInteger (n);
  }

  public void await (int phase)
  {
    if (count.decrementAndGet() == 0)
      return;
    while (count.get() != 0) ;
  }

}
```

Le compteur est donc décrémenté dans la méthode `await`. On utilise `decrementAndGet` pour décrémenter le compteur et en obtenir la nouvelle valeur, le tout de façon atomique, c'est-à-dire sans interférence possible de la part des autres threads.

Si `decrementAndGet` renvoie 0, cela signifie que le thread qui exécute cet appel à `await` est le dernier arrivé à la barrière. Dans ce cas, il est inutile de bloquer ce thread ; il peut immédiatement « passer la barrière » et continuer son exécution. La méthode `await` termine donc immédiatement, via **`return`**.

Si au contraire `decrementAndGet` renvoie une valeur non nulle, cela signifie que certains threads n'ont pas encore atteint la barrière. Dans ce cas, le thread qui exécute cet appel à `await` doit attendre que la barrière s'ouvre, c'est-à-dire attendre que le compteur partagé prenne la valeur 0. On implémente cette attente via la boucle **`while`** `(count.get() != 0);`. □

On change maintenant d'outil et on souhaite implémenter une barrière à l'aide d'un verrou et d'une variable de condition.

Question 19 À l'aide de la classe `ReentrantLock` et des interfaces `Lock` et `Condition` (figure 5), et sans utiliser la classe `AtomicInteger`, proposez une implémentation de la classe `Barrier` (figure 3). ◇

Solution. L'idée est la même : on maintient un compteur du nombre de threads qui n'ont pas encore atteint la barrière. Cependant, le compteur est maintenant stocké dans un champ ordinaire, de type **`int`**. Pour éviter les race conditions, il faut que les accès à ce champ soient protégés par un verrou. De plus, pour éviter l'attente active, il faut une variable de condition, qui permet au dernier thread arrivé d'envoyer un signal à tous les autres. À chaque barrière, on associe donc un verrou `l` et une variable de condition `c`.

```
public class Barrier {

  private final Lock l = new ReentrantLock ();
  private final Condition c = l.newCondition ();
  private int count;

  public Barrier (int n)
  {
    count = n;
```

```
  }

  public void await (int phase)
  {
    l.lock();
    try {
      if (--count == 0)
        c.signalAll();
      else
        while (count != 0) c.awaitUninterruptibly() ;
    } finally {
      l.unlock();
    }
  }

}
```

On n'oublie pas d'utiliser `try/finally` pour garantir que le verrou est toujours correctement relâché. Une fois le verrou pris, une expression de décrémentation ordinaire, `--count`, est utilisée pour décrémenter le compteur et en obtenir la nouvelle valeur.

Si `--count` renvoie 0, cela signifie que le thread qui exécute cet appel à `await` est le dernier arrivé à la barrière. Dans ce cas, comme précédemment, il est inutile de bloquer ce thread. De plus, il faut penser à réveiller tous les autres, qui sont arrivés plus tôt et se sont suspendus. On utilise `signalAll` pour cela.

Si au contraire `--count` renvoie une valeur non nulle, cela signifie que certains threads n'ont pas encore atteint la barrière. Dans ce cas, le thread qui exécute cet appel à `await` doit attendre que `count` prenne la valeur 0. Pour cela, il faut relâcher le verrou et se suspendre jusqu'à l'arrivée d'un signal ; c'est ce que permet `awaitUninterruptibly`.

On pourrait penser que, au moment où l'appel à `awaitUninterruptibly` termine, un signal a nécessairement été reçu, donc un signal a été envoyé, donc `count` est nécessairement égal à 0. Cependant, la documentation de Java indique (et on a dit en cours) que `awaitUninterruptibly` rend parfois la main (i.e., réveille parfois le thread qui l'a appelé) sans raison. On parle alors de « *spurious wakeup* ». Pour cette raison, il faut utiliser une boucle `while (count != 0) c.awaitUninterruptibly()` ; qui a pour effet de suspendre à nouveau le thread jusqu'à ce que le compteur ait effectivement atteint la valeur 0. □

On suppose maintenant que **la barrière sert plusieurs fois**, c'est-à-dire que chacun des `n` threads appelle `await(0)`, puis `await(1)`, etc.

Question 20 Modifiez votre solution à la question 19 de façon à ce que la barrière puisse servir plusieurs fois. ◇

Solution. Pour que la barrière puisse être ré-utilisée, il faudrait, une fois que le compteur a atteint la valeur 0, ré-initialiser la barrière, donc redonner au compteur la valeur `n`. Cependant, on voit dans la solution à la question précédente que les threads suspendus attendent via une boucle `while (count != 0) ...`. Par conséquent, il faudrait leur laisser l'opportunité d'observer que le compteur vaut 0, sans quoi ils resteront suspendus, alors qu'ils devraient être débloqués. Ces exigences sont contradictoires.

Pour éviter ce problème, il faut modifier légèrement l'organisation du code. On conserve l'idée que le champ `count` compte le nombre de threads qui n'ont pas encore atteint la barrière. Le dernier thread qui atteint la barrière ré-initialise donc ce champ, via l'instruction `count = n`. Pour cela, on a pris soin de stocker l'entier `n` dans un champ.

Du coup, il faut modifier la condition attendue par un thread qui a déjà atteint la barrière pour décider qu'il peut repartir. La boucle `while (count != 0) ...` ne doit pas être utilisée, puisque le champ `count` passe maintenant de 1 à 0 puis aussitôt de 0 à `n` sans qu'il soit possible, de l'extérieur, d'observer la valeur intermédiaire 0.

Il faut donc une autre façon d'indiquer aux threads suspendus le fait que l'attente est terminée et que la barrière a été ré-initialisée. Pour cela, on choisit de stocker dans la barrière, en plus du champ `count`, un champ `currentPhase`, qui indique le numéro de la phase actuelle. Lorsqu'un thread atteint la barrière, i.e., lorsqu'il appelle `await(phase)`, on peut être certain que `phase` et `currentPhase` sont égaux. Si ce thread constate qu'il est le dernier à atteindre la barrière, alors il incrémente `currentPhase`. Sinon, il attend que `currentPhase` soit incrémenté. Ainsi, l'attente ne se fait plus via `while (count != 0)` mais via `while (currentPhase != phase)`. Rappelons que `currentPhase` est un champ de l'objet barrière, donc est partagé entre tous les threads, tandis que `phase` est une variable locale, donc est propre à chaque thread.

```
public class Barrier {

  private final Lock l = new ReentrantLock ();
  private final Condition c = l.newCondition ();
  private final int n;

  private int count;
  private int currentPhase;

  public Barrier (int n)
  {
    this.n = n;
    count = n;
    currentPhase = 0;
  }

  public void await (int phase)
  {
    l.lock();
    try {
      assert (phase == currentPhase);
      if (--count == 0) {
        count = n;
        currentPhase++;
        c.signalAll();
      }
      else
        while (currentPhase != phase) c.awaitUninterruptibly() ;
    } finally {
      l.unlock();
    }
  }

}
```

Il n'est pas essentiel que `currentPhase` et `phase` soient des entiers. En réalité, leur parité suffit : ce pourraient être des booléens. L'important, c'est que la notion de « phase courante » change à chaque fois que l'on passe la barrière, pour que les threads suspendus puissent observer avec certitude que ce passage a eu lieu. Lorsque l'on utilise un booléen, ce type de barrière est connu sous le nom « *sense reversing barrier* ».

Notons que, pour l'utilisateur de la barrière, il est relativement désagréable de devoir fournir un argument `phase` à `await`. En effet, si jamais on fournit une valeur incorrecte pour cet argument, un blocage peut avoir lieu. On préférerait que `await` n'attende aucun argument. Pour cela, il faudrait que la valeur de `phase`, **pour chaque thread** , soit mémorisée dans l'objet barrière. En Java, cela est possible à l'aide de la classe `ThreadLocal`. Un objet de cette classe est une table qui à chaque thread associe une valeur. □

```
public class AtomicInteger {
  public AtomicInteger ();
  public int get ();
  public int decrementAndGet ();
  public int incrementAndGet ();
  // the other methods are omitted
}
```

FIGURE 4 – La classe `AtomicInteger`

```
public class ReentrantLock implements Lock {
  // methods omitted
}
public interface Lock {
  void lock ();
  void unlock ();
  Condition newCondition();
  // the other methods are omitted
}
public interface Condition {
  void awaitUninterruptibly ();
  void signal ();
  void signalAll ();
  // the other methods are omitted
}
```

FIGURE 5 – La classe `ReentrantLock` et les interfaces `Lock` et `Condition`