Contrôle classant – Informatique 431

Georges Gonthier   Jean-Jacques Lévy [*]

Ecole polytechnique, 2 juillet 2003

Tous les documents du cours sont autorisés. On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction.

Partie I – Partitions

Dans tout le problème, on posera $E = \{0, 1, \ldots n-1\}$ $(n > 0)$. On considère des partitions de E. Une partition $P = \langle B_0, B_1, \ldots B_{t-1} \rangle$ de *taille* t $(0 < t \leq n)$ est une suite de t sous-ensembles B_0, B_1, $\ldots B_{t-1}$ de E tels que $B_0 \cup B_1 \cup \ldots B_{t-1} = E$ et $B_i \cap B_j = \emptyset$ pour $0 \leq i < j < t$. Le *numéro de bloc* de chaque élément x de E est l'indice i du bloc B_i le contenant $(x \in B_i)$. Les partitions ont deux représentations (courte et longue) correspondant aux classes suivantes :

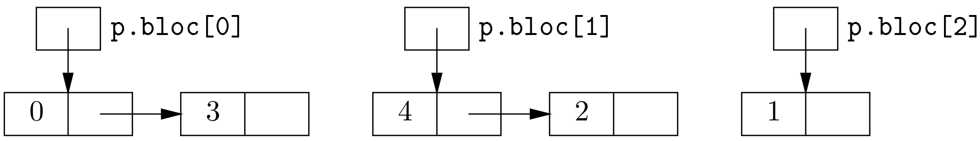
```
class Partition {
  int taille;
  int[ ] numBloc;
}
```

```
class PartitionBloc extends Partition {
  Liste[ ] bloc;
}
```

```
class Liste {
  int val;
  Liste suivant;
}
```

Le champ *taille* est la taille t de la partition ; le champ *numBloc* est un tableau donnant le numéro de bloc de tout x dans E ; le champ *bloc* est un tableau de t listes correspondant aux t blocs B_0, B_1, $\ldots B_{t-1}$.

Ainsi pour $n = 5$, la partition $P = \langle \{0, 3\}, \{2, 4\}, \{1\} \rangle$ est représentée par l'objet p de la classe `Partition` dont les deux champs vérifient *taille* $= 3$, *numBloc* $= \langle 0, 2, 1, 0, 1 \rangle$. Dans la version longue, le champ *bloc* est un tableau de 3 listes :



Remarque : dans la version longue (objets de la classe `PartitionBloc`), on s'arrangera pour que les blocs B_i ne soient jamais vides ; dans la version courte (objets de la classe `Partition`), certains B_i pourront être vides, par exemple dans la partition où *taille* $= 5$ et *numbloc* $= \langle 0, 4, 3, 0, 3 \rangle$ pour $n = 5$.

Question 1 Ecrire les constructeurs suivants :

a) `Partition(int n)` retournant la partition $\langle E \rangle$.

b) `Partition(Partition p)` retournant la même partition que P en supprimant les blocs vides inutiles.

Question 2 Ecrire le constructeur `PartitionBloc(Partition p)` retournant une représentation longue et sans blocs vides de la partition P.

Question 3 Ecrire une méthode `toString()` dans la classe `Partition` qui retourne, pour toute partition P, représentée par l'objet courant (*this*), une chaîne de caractères décrivant P sous une forme compréhensible pour l'impression.

[*]avec la participation de Luc Maranget

a) Peut-on écrire `System.out.print(p)` pour tout objet `p` de la classe `Partition` ? Pourquoi ?

b) Peut-on écrire `System.out.print(p)` pour tout objet `p` de la classe `PartitionBloc` ? Pourquoi ?

Mathématiquement, une partition P de E est un ensemble $P = \{B_0, B_1, \ldots B_{t-1}\}$ de t sous-ensembles non-vides B_0, B_1, $\ldots B_{t-1}$ de E tels que $B_0 \cup B_1 \cup \ldots B_{t-1} = E$ et $B_i \cap B_j = \emptyset$ pour $0 \leq i < j < t$ $(t > 0)$. L'intersection de deux partitions P et Q d'un même ensemble E est la partition de E définie par

$$P \sqcap Q = \{B \cap C \mid B \in P,\ C \in Q,\ B \cap C \neq \emptyset\}$$

Question 4 Calculer $P \sqcap Q$ quand $P = \{\{0, 1, \ldots n-2\}, \{n-1\}\}$ et $Q = \{\{0, 1, \ldots n-3\}, \{n-2, n-1\}\}$.

La partition $Q = \{C_0, C_1, \ldots C_{u-1}\}$ est un raffinement de $P = \{B_0, B_1, \ldots B_{t-1}\}$ si et seulement si pour tout C_j $(0 \leq j < u)$, il existe un B_i $(0 \leq i < t)$ tel que $C_j \subseteq B_i$. On écrira alors $P \sqsubseteq Q$. Remarque : on a $P \sqsubseteq Q$, si et seulement si $P \sqcap Q = Q$.

Question 5 Montrer que :

a) $P \sqsubseteq P \sqcap Q$ et $Q \sqsubseteq P \sqcap Q$,

b) $P \sqsubseteq X$, $Q \sqsubseteq X$ implique $P \sqcap Q \sqsubseteq X$

Etant donné deux partitions P et Q de E, on peut raffiner la partition P par la partition Q en remplaçant P par $P \sqcap Q$. On fait donc l'opération $P \leftarrow P \sqcap Q$.

Question 6 Ecrire la méthode `void raffiner(Partition q)` de la classe `PartitionBloc` qui modifie la partition P, représentée par l'objet courant (*this*), pour la remplacer par son raffinement par Q représentée par l'objet q.

(Indication : on cherche une solution de complexité $O(n)$ en temps où n est le nombre d'éléments de E. Pour cela, on peut parcourir chaque bloc de P en maintenant une table de correspondance entre les numéros de blocs de la partition Q et les nouveaux numéros de blocs de la partition résultat)

Partie II – Partitions compatibles avec une fonction

Dans cette partie, on suppose que f est une fonction de E dans E. L'image inverse de la partition P par la fonction f est définie par : $f^{-1}(P) = \{f^{-1}(B) \mid f^{-1}(B) \neq \emptyset,\ B \in P\}$. On remarque que l'image inverse d'une partition est également une partition.

Informatiquement, on représente la fonction f par le tableau `f` tel que `f[i]`$= f(i)$ pour $0 \leq i < n$.

Question 7 Ecrire la méthode `imageInv(int[ ] f)` de la classe `Partition` qui retourne, en temps linéaire en n, la partition $f^{-1}(P)$, image inverse de la partition P, représentée par l'objet courant (*this*), par la fonction f.

Si P est une partition de E, on note $\sim_P$ la relation d'équivalence associée définie par

$$x \sim_P y \quad \text{ssi} \quad x \in B,\ y \in B,\ B \in P$$

La partition P est *compatible* avec la fonction f si $x \sim_P y$ implique $f(x) \sim_P f(y)$ pour tout $x, y \in E$. On remarque que ceci est équivalent à $f^{-1}(P) \sqsubseteq P$, et on admet donc que P est compatible avec f si et seulement si P est un raffinement de l'image inverse de P par f.

Si on se donne une partition P de E, pour construire le plus petit raffinement de P compatible avec f, on itère le raffinement de P par $f^{-1}(P)$ jusqu'à obtenir une partition qui ne se raffine plus. On a alors $P = P \sqcap f^{-1}(P)$, c'est-à-dire $f^{-1}(P) \sqsubseteq P$; on dit qu'on a effectué le *raffinement P par la fonction f*.

Question 8 Soit $P = \{\{0, 1, \ldots n-2\}, \{n-1\}\}$. Soit f la fonction telle que $f(i) = i+1$ pour $0 \leq i < n-1$ et $f(n-1) = n-1$. Calculer le raffinement de P par f.

Question 9 Ecrire la méthode `raffiner(int[ ] f)` de la classe `PartitionBloc` qui modifie la partition P, représentée par l'objet courant (*this*), pour la remplacer par son raffinement par f. Donner un ordre de grandeur de la complexité en temps et en espace de cette fonction.

Partie III – Minimisation d'un automate fini

On suppose à présent que E est l'ensemble des états d'un automate fini déterministe $\mathcal{A} = (E, \Sigma, \delta, x_0, F)$. L'alphabet d'entrée Σ contient m caractères dont les codes (Unicode) sont compris entre le code de `'a'` et celui de `'a'`$+m-1$. La fonction de transition δ prend, en arguments, un caractère de Σ et un état de E et retourne un état de E. L'état initial est x_0 ; les états de fin forment un sous-ensemble F de E.

La classe `Automate` est ainsi définie :

```
class Automate {
  int x0; // état initial x0 (0 ≤ x0 < n)
  boolean[ ] terminal; // terminal[x] = true ssi x est un état de fin (0 ≤ x < n)
  int[ ][ ] delta; // matrice m × n donnant la fonction de transition δ
}
```

Soit f_c la fonction définie pour tout c de Σ par : $f_c(x) = \delta(c, x)$ pour $0 \leq x < n$.

Pour construire l'automate minimal équivalent à $\mathcal{A}$, on procède en deux étapes :

1. on élimine les états inaccessibles à partir de l'état initial x_0,
2. on construit la partition P de E qui est le raffinement de la partition $\{F, E-F\}$ par toutes les fonctions f_c $(c \in \Sigma)$. Les blocs de P seront les états de l'automate minimal.

Considérons le graphe dont les sommets sont les états de l'automate, et les arcs relient les sommets x et y tels que $y = f_c(x)$ pour un caractère c $(c \in \Sigma)$. Un état x est *accessible* s'il existe, dans ce graphe, un chemin entre le sommet x_0 (état initial de $\mathcal{A}$) et le sommet x.

Question 10 Ecrire la méthode `partitionAccessible()` de la classe `Automate` qui retourne la partition $\langle E - A, A\rangle$ où A est l'ensemble des états accessibles dans l'automate $\mathcal{A}$, représenté par l'objet *this*. Donner sa complexité en temps en fonction de n et de m.

On suppose à présent que tous les états de E sont accessibles dans l'automate $\mathcal{A}$.

Question 11 Ecrire la méthode `partitionMin()` de la classe `Automate` qui retourne la partition des états de l'automate $\mathcal{A} = (E, \Sigma, \delta, x_0, F)$, représenté par l'objet *this*, qui est le plus petit raffinement de la partition $\langle F,\, E-F\rangle$ compatible avec les fonctions f_c $(c \in \Sigma)$. (Le résultat sera un objet de la classe `partitionBloc`). Donner sa complexité en temps en fonction de n et de m.

(Indication : il faut être compatible simultanément avec toutes les fonctions f_c, et non compatible successivement par rapport à chacune des fonctions f_c.)

Question 12 Ecrire la fonction statique `automateMin(Automate a)` qui retourne l'automate minimal équivalent à l'automate $\mathcal{A}$ correspondant à l'objet a de la classe `Automate`.

Partie IV – Optimisation

Si $B \subseteq E$ et $D \subseteq E$, un *diviseur* D de B (par f) est un ensemble tel que $B \cap f^{-1}(D) \neq \emptyset$ est vrai et $B \subseteq f^{-1}(D)$ est faux. Un diviseur D de B coupe donc B en deux ensembles non vides B_1 et B_2 tels que $B_1 = B \cap f^{-1}(D)$ et $B_2 = B - B_1$ $(B_1 \neq \emptyset$ et $B_2 \neq \emptyset)$.

L'algorithme de Hopcroft et Ullman optimise le calcul du raffinement d'une partition P par une fonction f en maintenant une liste pas trop grande W de blocs de P par lesquels on doit diviser la partition P.

On effectue l'algorithme suivant :

1. $W = P = \{B_0, B_1, \dots B_{t-1}\}$ (en fait, $W = \{B_0, B_1, \dots B_{t-2}\}$ est suffisant).
2. Tant que W n'est pas vide, faire (3) et (4) :
3. on retire un D quelconque de W.
4. Pour tout bloc B_i de P dont D est un diviseur,
   (a) Dans P, on remplace B_i par les deux blocs B_i^1 et B_i^2 produits par cette division.

(b) Si $B_i \in W$, on remplace B_i par B_i^1 et B_i^2 dans W.

(c) Si $B_i \notin W$, on rajoute le plus petit des deux ensembles B_i^1 et B_i^2 à W.

Question 13 Donner une suite des W obtenus en partant de $W = P = \{\{0, 1, \ldots n-2\}, \{n-1\}\}$ quand f est la fonction définie par $f(i) = i + 1$ pour $0 \leq i < n-1$ et $f(n-1) = n-1$.

Soit $[x]_P$ le bloc de P contenant x. On compte, pour tout x de E, le nombre de D choisis à la ligne (3) de l'algorithme précédent tels que $x \in D$. A chaque exécution de la ligne (3), si $D = [x]_P$, on avait $[x]_P \in W$ avant, on a $[x]_P \notin W$ après jusqu'à ce qu'on rajoute $[x]_P$ à W en (4-c). Lorsque cela se produit, la taille de $[x]_P$ est au moins divisée par 2 ; comme la taille de $[x]_P$ ne peut que décroitre par ailleurs, il n'y a au plus que $1 + \log n$ blocs D contenant x à la ligne (3).

Question 14 Donner un ordre de grandeur de la complexité en temps de cet algorithme si on sait faire chaque itération de (4) en temps $O(|D| + |f^{-1}(D)|)$, où $|B|$ est le nombre d'éléments de tout ensemble B.

L'algorithme précédent calcule bien le raffinement de P par f. En effet, si D_k est l'élément choisi dans W à l'itération k, et si on pose $S = \{E\} \sqcap \widetilde{D}_1 \sqcap \widetilde{D}_2 \sqcap \ldots \widetilde{D}_k$, on peut montrer que $f^{-1}(S) \sqsubseteq P$ et $P = S \sqcap \widetilde{W}$ sont des invariants de boucle à la ligne (4). D'où le résultat quand $W = \emptyset$. (La partition $\widetilde{D}_i$ est formée par D_i et son complément dans E ; de même $\widetilde{W}$ est la partition obtenue en rajoutant le complément de W dans E ; donc $\widetilde{D} = \{D, E - D\} - \emptyset$, $\widetilde{W} = \{W, E - \cup W\} - \emptyset$).

Question 15 Ecrire la fonction statique `Graphe grapheInv(int[ ] f)` qui retourne le graphe représentant la fonction inverse de f. (Le graphe résultat est représenté avec des listes de successeurs comme dans le cours ; les successeurs de tout sommet y de E sont les sommets x tels que $y = f(x)$).

Le reste du problème consiste à démontrer qu'on peut réaliser pour tout D chaque itération de (4) en temps $O(|D| + |f^{-1}(D)|)$. Pour cela, on introduit les classes `Bloc`, et `PartitionHU` suivantes :

```
class Bloc {
  int longueur;
  Liste contenu;
  Bloc (int x) { ajouter(x); }
  void ajouter (int x) {
    contenu = new Liste(x, contenu);
    ++longueur;
} }
```

```
class PartitionHU extends Partition {
  Bloc[ ] bloc;
  Bloc[ ] sousBloc;
  PartitionHU (Partition p) { ... }
}
```

La classe `Bloc` représente les sous-ensembles de E par des listes d'entiers avec leur cardinalité stockée dans le champ *longueur*. La méthode *ajouter* ajoute un élément à un bloc en temps $O(1)$. La classe `PartitionHU` est une sous-classe de `Partition` avec deux champs supplémentaires *bloc* et *sous-bloc* représentant les blocs de la partition et une liste annexe utilisée par la suite.

Question 16 Ecrire la méthode `Liste blocsImage(int d, Graphe fInv)` de la classe `PartitionHU` qui retourne la liste des numéros des blocs de P qui intersectent $f^{-}1(D)$, où P est la partition représentée par l'objet *this* et d est le numéro de bloc de D dans P, en complexité $O(|D| + |f^{-1}(D)|)$ en temps. (Le champ *sousBloc[i]* pour tout bloc B_i de P sera modifié pour représenter $B_i \cap f^{-1}(D)$)

Question 17 Ecrire la méthode `Liste nouveauxDiviseurs (Liste sb, Liste w, boolean[ ] dansW)` de la classe `PartitionHU` qui retourne la liste des nouveaux blocs dans W et qui prend en arguments la liste des blocs de P intersectant D, calculée à la question précédente, l'ensemble W avec sa fonction caractéristique `dansW`. (P est la partition représentée par l'objet *this*; le calcul doit se faire en temps $O(|f^{-1}(D)|)$; seuls les champs *longueur* des blocs B_i de P seront modifiés).

On en déduit que le calcul de chaque itération de la ligne (4) de l'algorithme de Hopcroft et Ullman prend un temps $O(|D|+|f^{-1}(D)|)$. Paige et Tarjan ont traité le cas où la fonction f devient une relation $\mathcal{R}$.

Corrigé

Question 1

```
Partition (int n) { taille = 1; numBloc = new int[n]; }

Partition (Partition p) {
  int n = p.numBloc.length, t = p.taille;
  int[ ] phi = new int[t];
  for (int k = 0; k < t; ++k) phi[k] = -1;
  numBloc = new int[n];
  for (int x = 0; x < n; ++x) {
    int k = p.numBloc[x]; int j = phi[k];
    if (j == -1)
      j = phi[k] = taille++;
    numBloc[x] = j;
  }
}
```

Question 2

```
private void calculerBlocs() {
  bloc = new Liste [taille];
  for (int i = 0 ; i < numBloc.length ; ++i) {
    int k = numBloc[i];
    bloc[k] = new Liste (i, bloc[k]);
} }

PartitionBloc (Partition p) {
  super(p);
  calculerBlocs() ;
}
```

Question 3

```
public String toString() {
  String s = taille + " [ ";
  for (int x = 0; x < numBloc.length; ++x)
    s = s + numBloc[x] + " ";
  return s + "]";
}
```

a) `System.out.print(p)` imprime la chaîne de caractères `p.toString()`. Donc la méthode précédente est appelée pour imprimer `p`.

b) `System.out.print(p)` appelle la méthode `toString` de la plus petite classe contenant `p`. Comme elle n'est pas définie dans la classe `PartitionBloc`, on appelle la méthode de la classe `Partition`. Ce qui donne la même impression que précédemment.

Question 4 $P \sqcap Q = \{\{0, 1, \dots n-3\}, \{n-2\}, \{n-1\}\}$.

Question 5

a) Si $D \in P \sqcap Q$, il existe $B \in P$ et $C \in Q$, tel que $D = B \cap C$. Donc on a $D \subseteq B$ pour un B de P. On a donc $P \sqsubseteq P \sqcap Q$. Par symétrie $Q \sqsubseteq P \sqcap Q$.

b) Si $P \sqsubseteq X$ et $Q \sqsubseteq X$, alors pour tout $D \in X$, on sait que $D \neq \emptyset$ et qu'il existe B dans P et C dans Q tels que $D \subseteq B$ et $D \subseteq C$. Donc $D \subseteq B \cap C \neq \emptyset$. D'où $P \sqcap Q \sqsubseteq X$.

Question 6

```
void raffiner (Partition q) {
```

```
    int t = taille, u = q.taille, n = numBloc.length;
    int[ ] phi = new int[u];
    for (int k = 0; k < u; ++k) phi[k] = -1;
    for (int k = 0; k < t; ++k) {
      for (Liste b = bloc[k]; b != null; b = b.suivant) {
        int i = b.val, kqi = q.numBloc[i], ki = phi[kqi];
        if (ki < 0) ki = phi[kqi] = taille++;
        numBloc[i] = ki;
      }
      for (Liste b = bloc[k]; b != null; b = b.suivant)
        phi[b.val] = -1;
    }
    calculerBlocs();
  }
```

La boucle interne avec l'indice `bk` prend un temps $O(|B_k|)$ où B_k est le bloc de P correspondant. Au total, cela prend un temps $O(\Sigma_{k=0}^{t-1}|B_k|) = O(n)$.

Question 7

```
  Partition imageInv (int[ ] f) {
    int n = numBloc.length;
    Partition p = new Partition(n);
    for (int i = 0; i < n; ++i)
      p.numBloc[i] = numBloc[f[i]];
    p.taille = taille;
    return p;
  }
```

Question 8 On a $P = P_0 = \{\{0, 1, \ldots n-2\}, \{n-1\}\}$ et $f^{-1}(P) = \{0, 1, \ldots n-3\}, \{n-2, n-1\}\}$. Alors $P_1 = P \sqcap f^{-1}(P) = \{0, 1, \ldots n-3\}, \{n-2\}, \{n-1\}\}$ et $f^{-1}(P_1) = \{0, 1, \ldots n-4\}, \{n-3\}, \{n-2, n-1\}\}$, etc. On finit avec la partition $P_{n-1} = \{\{0\}, \{1\}, \ldots \{n-2\}, \{n-1\}\}$. Cette partition est compatible avec la fonction f. Par ailleurs, toute partition compatible avec f ne peut identifier que $n-2$ et $n-1$. Mais alors elle n'est pas un raffinement de P. La seule partition compatible avec à P est P_{n-1}.

Question 9

```
  void raffiner (int f[ ]) {
    int t;
    do {
      t = taille;
      raffiner(imageInv(f));
    } while (taille > t);
  }
```

Les méthodes *imageInv* et *raffiner* ont une complexité en temps $O(n)$. On ne peut raffiner plus que n fois la partition avant d'atteindre un état stationnaire. Au total le temps pris est $O(n^2)$. L'espace pris par ces fonctions n'est jamais plus que $O(n)$.

Question 10

```
  Partition partitionAccessible() {
    int n = terminal.length;
    Partition r = new Partition(n);
    r.taille = 2;
    dfs(x0, r);
    return r;
  }
```

```
void dfs (int x, Partition r) {
  if (r.numBloc[x] == 0) {
    r.numBloc[x] = 1;
    for (int c = 0; c < delta.length; ++c)
      dfs(delta[c][x], r);
} }
```

Le temps pris par cette fonction est celle d'un parcours dfs du'un graphe, donc en complexite est en $O(E+V) = O(n + n \times m) = O(n \times m)$.

Question 11

```
Partition partitionFin() {
  int n = terminal.length;
  Partition r = new Partition(n); r.taille = 2;
  for (int i = 0 ; i < n ; ++i) r.numBloc[i] = terminal[i] ? 1 : 0;
  return r;
}

PartitionBloc partitionMin() {
  int n = terminal.length, m = delta.length;
  PartitionBloc r = new PartitionBloc (partitionFin());
  int t;
  do {
    t = r. taille;
    for (int c = 0 ; c < m ; ++c)
      r.raffiner(delta[c]);
  } while (r.taille < t);
  return r ;
}
```

Question 12

```
static Automate automateMin (Automate a) {
  int n = a.terminal.length, m = a.delta[0].length;
  PartitionBloc p = a.partitionMin(); int n1 = p.taille;
  Automate r = new Automate();
  r.terminal = new boolean[n1]; r.delta = new int [m][n1];
  r.x0 = p.numBloc[a.x0];
  for (int k = 0 ; k < n1; ++k) {
    int i = p.bloc[k].val;
    r.terminal[k] = a.terminal[i];
    for (int c = 0; c < m; ++c)
      r.delta[c][k] = p.numBloc[a.delta[c][i]];
  }
  return r;
}
```

Question 13 Par exemple $W_0 = P = \{\{0,1,\ldots n-2\},\{n-1\}\}$. Puis $W_1 = \{\{n-2\},\{n-1\}\}$. Alors $P = \{\{0,1,\ldots n-3\},\{n-2\},\{n-1\}\}$. Puis $W_2 = \{\{n-3\},\{n-1\}\}$. Alors $P = \{\{0,1,\ldots n-4\},\{n-3\},\{n-2\},\{n-1\}\}$. Etc. Puis $W_{n-1} = \{\{0\},\{n-1\}\}$. Alors $P = \{\{0\},\{1\},\ldots n-1\}$. Et finalement $W_n = \emptyset$.

Question 14 Si D_i sont les blocs successifs pris dans W, le nombre d'opérations de l'algorithme est :

$$\begin{aligned}
\Sigma_{D_i}|D_i| + |f^{-1}(D_i)| &= \Sigma_x \Sigma_{x \in D_i}(1 + f^{-1}(x)) \\
&= \Sigma_{x \in D_i} \Sigma_x (1 + f^{-1}(x)) \\
&\leq (\log n)(n + \Sigma_x(f^{-1}(x))) \quad = \quad 2n \log n
\end{aligned}$$

La complexité en temps de l'algorithme est en $O(n \log n)$.

Question 15

```
static Graphe grapheInv (int[ ] f) {
  int n = f.length;
  Graphe r = new Graphe (n);
  for (int x = 0; x < n; ++x)
    r.succ[f[x]] = new Liste (x, r.succ[f[x]]);
  return r;
}
```

Question 16

```
Liste blocsImage (int d, Graphe fInv) {
  Liste sb = null;
  for (Liste b = bloc[d].contenu; b != null; b = b.suivant) {
    int x = b.val;
    for (Liste a = fInv.succ[x]; a != null; a = a.suivant) {
      int y = a.val, j = numBloc[y];
      if (sousBloc[j] == null) {
        sousBloc[j] = new Bloc(y);
        sb = new Liste(j, sb);
      } else sousBloc[j].ajouter(y);
    }
  }
  return sb;
}
```

Question 17

```
Liste nouveauxDiviseurs (Liste sb, Liste w, boolean dansW[ ]) {
  for (; sb != null; sb = sb.suivant) {
    int i = sb.val;
    Bloc sbi = sousBloc[i]; sousBloc[i] = null;
    int m = sbi.longueur, dm = bloc[i].longueur - m;
    if (dm > 0) {
      bloc[taille++] = sbi; bloc[i].longueur = dm;
      for (Liste b = sbi.contenu; b != null; b = b.suivant)
        numBloc[b.val] = taille - 1;
      int j = (dansW[i] || dm <= m) ? taille - 1 : i;
      w = new Liste (j, w); dansW[j] = true;
    }
  }
  return w;
}
```