Terminaison des programmes

I PL : un langage de programmation simplifié

Dans ce chapitre ainsi que les suivants, on considère un langage de programmation simplifié. Les instructions que l'on retient sont les suivantes :

- **l'affectation** : on la note $X \leftarrow e$ où X est un nom de variable et e est une expression (par exemple une opération arithmétique). L'expression e peut contenir la variable X ou non.
- **la composition** : on la note $c_1 ; c_2$ où c_1 et c_2 sont deux blocs d'instructions. On remplacera souvent le ; par un passage à la ligne.
- **l'instruction conditionnelle** : on la notera **si** b **alors** c_1 **sinon** c_2 **fin**, où b est un test (comparaison, égalité) et c_1 et c_2 sont deux blocs d'instructions. Le **sinon** est optionnel.
- **la boucle pour** : on la notera **pour** i de p à q **faire** c **fin** où p et q sont deux entiers et c est un bloc d'instructions.
- **la boucle tant que** : on la notera **tant que** b **faire** c **fin** où b est un test (appelé condition d'arrêt) et c un bloc d'instructions.
- **l'instruction de retour** : on la notera **retourner** e qui termine l'exécution de l'algorithme en retournant la valeur de e.

On ajoute toutes les opérations classiques (arithmétiques, sur les chaînes, les tableaux, etc.) dont on aura besoin. Le langage PL (pour Pseudo-Langage) est *typé statiquement*. Par le type *entier*, on entend *entier relatif*, sinon, on précisera *entier naturel*. Les algorithmes exprimés en PL et examinés dans ce cours sont supposés correctement typés. On peut commencer à se familiariser avec PL.

On privilégie la clarté d'exposition des algorithmes

1 – On considère les algorithmes suivants. Que font-ils ? Considérer les cas limites. Par exemple, pour les algorithmes 1 et 2, que se passe-t-il si n est un entier négatif ou nul ? Pour l'algorithme 3, que se passe-t-il si L est vide ? Affiner au besoin les réponses précédentes.

Algorithme 1 :

```
Entrées : n, un entier
Sorties : un entier
somme : entier ← 0
pour i de 1 à n faire
    somme ← somme + 1
fin
retourner somme
```

Algorithme 2 :

```
Entrées : n, un entier
Sorties : une chaine de caractères
ch : chaine ← ""
nombre : entier ← n
tant que nombre ≠ 0 faire
    ch ← str(nombre%2) + ch
    nombre ← nombre//2
fin
retourner ch
```

Algorithme 3 :

```
Entrées : L[0..], un tableau d'entiers
Sorties : un entier
n : entier ← len(L)
mini : entier ← L[0]
pour i de 1 à n faire
    si L[i] < mini alors
        mini ← L[i]
fin
retourner mini
```

2 – Écrire un algorithme qui prend en argument un tableau non-vide d'entiers et renvoie l'indice de son plus grand élément.

3 – Écrire un algorithme qui prend en argument un tableau non-vide d'entiers et renvoie la moyenne de ses éléments.

4 – Écrire un algorithme qui prend en argument un tableau d'entiers ainsi qu'un entier, et qui renvoie un booléen indiquant si l'entier fait partie des éléments du tableau.

5 – Écrire un algorithme qui prend en argument un entier n et qui renvoie la valeur $\sum_{1 \leq i,j \leq n} \min(i,j)$.

II Le problème de l'arrêt

Il est important de pouvoir montrer qu'un programme termine, ne serait-ce que pour le tester sereinement et le qualifier d'algorithme.

> Un algorithme est un programme qui se termine pour toute entrée conforme

Par *entrée conforme*, on entend qui respecte le type (par exemple `n:entier`) et l'éventuelle précondition associée (par exemple `n non-nul`). Ce qui survient si l'entrée n'est pas conforme ne nous concerne pas.

6 – Parmi les instructions retenues, quelles sont celles qui ne peuvent pas engendrer d'exécution infinie d'un programme PL ? Quelle est la seule instruction problématique ?

On suppose maintenant l'existence d'un algorithme pour une fonction appelée `Termine` qui répond `vrai` ssi un programme termine. Ainsi, l'appel `Termine(P:prog)` renvoie en un temps fini soit `vrai` si le programme `P` termine soit `faux` si `P` boucle.

7 – On considère le programme suivant, qu'on appelle `Turing`. Termine-t-il ? *Indication :* supposer qu'il termine et montrer une contradiction. Supposer qu'il ne termine pas et montrer une contradiction. Que peut-on conclure concernant l'hypothétique existence de l'algorithme `Termine` ?

Algorithme 4 : Turing

```
tant que Termine(Turing :prog) faire
    rien
fin
```

On comprend donc que le problème de la terminaison des programmes n'est pas si simple :

Théorème 1: Indécidabilité du problème de l'arrêt (Alan Turing, 1936)

Il n'existe pas de d'algorithme permettant de dire si un programme termine toujours ou non.

Autrement dit, l'algorithme `Termine` ne peut pas exister. Comme nous avons montré que le probléme de l'arrêt : *un programme termine-t-il, oui ou non ?* n'admet pas de solution algorithmique, on dit que c'est un problème *indécidable*.

III La fonction de Syracuse

Le cas de la fonction de Syracuse est symptomatique.

Algorithme 5 : fonction syracuse

```
Entrées : un entier n ≥ 1
Syr : entier ← n;
(*) tant que Syr ≠ 1 faire
    si Syr est pair alors
        Syr ← Syr/2
    sinon
        Syr ← 3 * Syr + 1
    fin
fin
retourner Syr
```

8 – Lister les valeurs prises par la variable `Syr` au point de programme (*) lors de l'exécution de la fonction `syracuse` pour $n = 16$ puis $n = 15$.

9 – Si l'exécution de la fonction termine, quelles sont les valeurs retournées ?

En faisant tourner cette fonction avec différentes valeurs de n, on se rend compte qu'on arrive à chaque fois à 1 et qu'on sort de la boucle. Mais il n'existe pour le moment aucune justification mathématique permettant de le garantir pour l'ensemble des entiers $n \geq 1$. En l'état actuel des connaissances, c'est un problème *ouvert* : on ne sait pas si cette fonction termine toujours. La qualifier d'algorithme est donc abusif.

IV Démontrer la terminaison

Bien que le problème de l'arrêt soit indécidable, dans l'immense majorité des cas on sait démontrer la terminaison de programme et on dispose pour cela d'un éventail de techniques de preuve. Nous allons en étudier quelques unes dans la suite de ce chapitre. Commençons par examiner l'algorithme très simple suivant :

Algorithme 6 : Algorithme très naïf

```
i : entier ← 0
tant que i < 5 faire
|  (*) i ← i + 1
fin
```

10 – Expliquer informellement pourquoi cet algorithme termine.

11 – On pose $f(i) = 5 - i$ et on définit i_n comme étant la valeur de i au point de programme (*) et au tour de boucle numéro n. Que peut-on dire de la suite $(f(i_n))$?

12 – Donner un minorant de f tant que l'on est dans la boucle. Que peut-on en déduire ?

Les preuves de terminaison que nous examinons ici reposent sur le fait que la relation $<$ est bien fondée sur $\mathbb{N}$:

Propriété 1:

Il n'existe pas de suite infinie strictement décroissante d'entiers naturels.

13 – Montrer cette propriété en procédant en deux temps. Montrer tout d'abord que toute partie non-vide de $\mathbb{N}$ admet un plus petit élément. *Indication :* par l'absurde en supposant l'existence d'un ensemble E non vide qui n'admet pas de plus petit élément, montrer par récurrence la propriété $\forall n \in \mathbb{N}, \forall e \in E, n \leq e$ et en déduire une contradiction. Dans un second temps, en déduire que toute suite strictement décroissante d'entiers naturels est finie.

Prouver la terminaison consiste à exhiber une *fonction de rang* (parfois nommée *variant de boucle*).

Définition 1 : Fonction de rang à valeur dans $\mathbb{N}$

Une *fonction de rang* pour une boucle est une fonction à valeur dans $\mathbb{N}$ qui dépend des variables et paramètres du programme et qui décroît strictement à chaque passage dans la boucle.

La fonction de rang peut parfois se déduire du test de la boucle comme nous le verrons sur des exemples.

Pour valider une fonction de rang, on vérifie les trois conditions suivantes :
T0 : la fonction de rang est a priori à valeur dans $\mathbb{Z}$
et pour *chaque passage* dans la boucle :
T1 : la fonction de rang est en fait à valeur dans $\mathbb{N}$ au début du corps de la boucle
T2 : sa valeur décroît strictement entre le début et la fin du corps de la boucle

Si la boucle devait ne pas terminer pour un jeu d'entrées donné, alors une telle fonction de rang permettrait de construire une suite infinie d'entiers naturels strictement décroissante, ce qui contredirait la propriété 1. Donc l'existence d'une fonction de rang implique la terminaison. Sous certaines conditions, la réciproque est également vraie : si la boucle termine pour chaque jeu de données, alors il existe une fonction de rang à valeur dans $\mathbb{N}$. Pour construire la fonction de rang, il suffit en effet d'associer à chaque jeu de données le nombre de passages dans la boucle.

14 – Proposer et valider une fonction de rang pour l'algorithme 6.

15 – Proposer et valider une fonction de rang pour l'algorithme 7.

Algorithme 7 : fonction produit(a,b)

Entrées : deux entiers a et b avec $a \geq 0$
Sorties : un entier
p : *entier* $\leftarrow 0$;
x : *entier* $\leftarrow a$;
tant que $x > 0$ **faire**
  $p \leftarrow p + b$;
  $x \leftarrow x - 1$;
fin
retourner p

16 – Montrer $\forall n \in \mathbb{N}^*, n > n//2 \geq 0$, par exemple en distinguant suivant la parité de n. Puis proposer et valider une fonction de rang pour l'algorithme 2.

17 – Montrer que l'application $\rho : \mathbb{Z} \to \mathbb{N}$ définie par

$$\rho(x) = \begin{cases} 0 \text{ si } x \leq 0 \ ; \ 2 \text{ si } x = 1 \ ; \\ 2 \text{ si } x = 2 \ ; \ 4 \text{ si } x = 3 \ ; \\ 3 \text{ si } x = 4 \ ; \ 1 \text{ si } x \geq 5 \ ; \end{cases}$$

est une fonction de rang qui garantit la terminaison de l'algorithme 8.

Algorithme 8 : -2x+10

Entrées : n un entier
x : *entier* $\leftarrow n$;
tant que $x > 0$ **faire**
  $x \leftarrow -2x + 10$
fin

Pour toutes les valeurs possibles de n, comparer $\rho(n)$ et le nombre de passage dans la boucle quand x est initialisée avec n. Qu'observe-t-on ?

18 – Exécuter l'algorithme 9 sur les entrées suivantes : $([1, 3, 5, 7, 8], 4)$, $([1, 3, 5, 7, 8], 7)$. Que fait-il ?

Algorithme 9 : fonction rd(L,a)

Entrées : L[0..] un tableau trié non vide
d'entiers et a un entier
Sorties : un booléen
n : *entier* $\leftarrow len(L)$;
g : *entier* $\leftarrow 0$;
d : *entier* $\leftarrow n - 1$;
tant que $d - g > 0$ **faire**
  m : *entier* $\leftarrow (d + g)//2$
  si $L[m] < a$ **alors**
    $g \leftarrow m + 1$
  sinon
    $d \leftarrow m$
  fin
fin
retourner *L[g]*$==a$

19 – Montrer $\forall d, g \in \mathbb{N}, d > g \Rightarrow d > (d + g)//2 \geq g$. En supposant que d et g sont à valeur dans $\mathbb{N}$ lors de toute exécution, en déduire que $f(d, g) = d - g$ est une fonction de rang pour la boucle de l'algorithme 9.

20 – Intuitivement, quelle propriété à propos de u et v doit-on supposer au point de programme (*) pour éviter la non-terminaison de l'algorithme 10 ? *Indication :* que se passe-t-il si u est nul ? Et si v est nul ? La propriété doit exprimer que ces cas ne surviennent pas. On verra ci-dessous comment justifier cette propriété.

Algorithme 10 : fonction pgcd(a,b)

Entrées : a et b deux entiers > 0
Sorties : un entier
u : *entier* $\leftarrow a$;
v : *entier* $\leftarrow b$;
(*) **tant que** $u \neq v$ **faire**
  si $u > v$ **alors**
    $u \leftarrow u - v$
  sinon
    $v \leftarrow v - u$
  fin
fin
retourner u

Définition 2 : Invariant

Un *invariant* associé à un point de programme p est une propriété des valeurs des variables du programme qui est vérifiée à chaque passage sur ce point p lorsqu'on exécute le programme.

En particulier, pour montrer la terminaison, on s'intéresse aux invariants associés aux points de programme qui se situent à l'entrée d'une boucle **tant que**, comme par exemple le point de programme (*) de l'algorithme 10. De tels invariants sont nommés *invariants de boucle*.

Pour montrer qu'une propriété I est un invariant de boucle au point de programme p à l'entrée d'une boucle **tant que**, on peut raisonner comme suit :
Cas de base : montrer que I est vraie la première fois que l'exécution passe au point p.
Cas inductif : supposer que la propriété I est vraie en ce point p, exécuter le corps de la boucle une fois et montrer que la propriété reste vraie quand l'exécution revient sur ce point p.

Les propriétés montrées comme ci-dessus sont qualifiées d'*inductives*. Un invariant au sens de la définition 2 n'est pas nécessairement inductif (cf. l'algorithme 11 et l'exercice associé). En revanche, tout invariant inductif est un invariant au sens de la définition 2. On retrouvera cette notion d'invariant au chapitre suivant, notion cruciale pour garantir la correction d'algorithme (i.e., la valeur retournée est-elle bien celle attendue ?).

21 – Montrer que les valeurs de u et v sont toujours des entiers strictement positifs au point de programme (*) de l'algorithme 10 en argumentant que la formule $u \in \mathbb{N}^* \wedge v \in \mathbb{N}^*$ est un invariant inductif en (*).

22 – Montrer que $f(u, v) = u + v$ est une fonction de rang pour la boucle de l'algorithme 10.

23 – Montrer que $f(u, v) = max(u, v)$ est une fonction de rang pour la boucle de l'algorithme 10.

24 – Montrer que $f(u, v) = min(u, v)$ n'est *pas* une fonction de rang pour la boucle de l'algorithme 10.

25 – En général, si une boucle termine, il y a-t-il *unicité* de la fonction de la rang ?

26 – Soit $f(x_1, \ldots, x_n)$ une fonction de rang pour une boucle opérant sur les variables $x_1, \ldots, x_n$. Proposer une famille infinie de fonctions de rang pour cette boucle.

27 – Considérer le programme suivant.

Algorithme 11 : invariant et invariant inductif

Sorties : un entier
x : *entier* $\leftarrow 1$
(*) **tant que** $x \leq 50$ **faire**
  $x \leftarrow x + 2$
fin
retourner x

A partir de la trace de l'exécution de ce programme, déterminer au point de programme (*) un invariant I_0 de boucle sous forme d'un intervalle pour `x` *aussi précis que possible*. Déterminer un invariant *inductif* I_1 de boucle sous forme d'un intervalle pour `x` *aussi précis que possible*. En terme de précision, quel lien observe-t-on entre I_0 et I_1 ? Déterminer une fonction de rang montrant la terminaison de ce programme. En utilisant I_0 (resp. I_1) et la négation du test de la boucle, quelle est la plage des valeurs que retourne cet algorithme ?

28 – Considérer le programme suivant. Tracer pour $x = 4, y = 3$ et $x = 7, y = 3$. Déterminer un invariant inductif pour le point de programme (*) décrivant la plage de valeurs de a. Déterminer une fonction de rang.

Algorithme 12 : multiplication russe

```
Entrées : x et y, deux entiers naturels
Sorties : un entier
a : entier ← x
b : entier ← y
r : entier ← 0
(*) tant que a > 0 faire
    si a% 2 =0 alors
        b ← 2 * b
        a ← a//2
    sinon
        r ← r + b
        a ← a − 1
    fin
fin
retourner r
```

29 – Considérer le programme suivant. Tracer pour $n = 10$ et $n = 16$. Déterminer une fonction de rang. *Indication :* un invariant de boucle à propos des valeurs de c est indispensable.

Algorithme 13 : racine carrée entière

```
Entrées : n, un entier naturel
Sorties : un entier
c : entier ← 0
s : entier ← 1
(*) tant que s ≤ n faire
    c ← c + 1
    s ← s + 2 * c + 1
fin
retourner c
```

30 – Considérer le programme suivant. Tracer pour $z = 5$. Déterminer un invariant de boucle pour le point de programme (*) décrivant les plages de valeurs de c et p. Montrer que $f(c, p) = 3 * c + 2 * p$ est une fonction de rang. Quelle est l'information minimale extraite de l'invariant qu'il est nécessaire d'utiliser pour montrer que f est une fonction de rang ? Déterminer la plage des valeurs de sortie de ce programme.

Algorithme 14 : 3c2p

```
Entrées : z, un entier naturel
Sorties : un entier
p : entier ← z
c : entier ← 0
(*) tant que p > 0 faire
    si c = 0 alors
        p ← p − 2
        c ← 1
    sinon
        p ← p + 1
        c ← 0
    fin
fin
retourner p
```

V Fonction de rang à valeur dans ℕ et complexité en temps

La valeur initiale à l'entrée de la boucle d'une fonction de rang à valeur dans ℕ détermine par définition un *majorant* du nombre de passage dans la boucle. Si le temps d'une exécution du corps de la boucle est connu, on dispose alors d'une information, éventuellement imprécise mais correcte, sur la *complexité en temps* de la boucle considérée.

Par exemple, considérons à nouveau l'algorithme 6. Nous avons vu que la fonction $f(i) = 5 - i$ est une fonction de rang valide. Pour déterminer un majorant du nombre de passages dans la boucle, il suffit de déterminer la valeur initiale de celle-ci juste avant le premier passage : $f(0) = 5$. Donc il y a au plus cinq passages dans cette boucle.

En général, le nombre de passages dépend des entrées de l'algorithme. Considérons par exemple l'algorithme 7. La fonction $f(x, p) = x$ est une fonction de rang. Sa valeur initiale est $f(a, 0) = a$. Le nombre de passages dans la boucle est donc majoré par a.

31 – Reprendre tous les algorithmes de cette section et majorer le nombre de passages dans les boucles.

VI Terminaison et ordre lexicographique

On peut généraliser la définition de fonction de rang en imposant qu'elle soit à valeur dans un ensemble bien fondé donné. Prouver la terminaison d'une boucle consiste donc à exhiber un ensemble bien fondé E et une fonction de rang à valeur dans E qui dépend des variables de la boucle et qui, relativement à l'ordre bien fondé sur E, décroit strictement à chaque passage dans la boucle.

Un cas fréquent concerne les ordres lexicographiques. Rappelons qu'à partir de deux ensembles ordonnés $(E_1, \leq_1)$ et $(E_2, \leq_2)$, on définit l'ordre lexicographique par $(x_1, x_2) \leq_{lex} (y_1, y_2)$ si et seulement si $x_1 <_1 y_1 \vee (x_1 = y_1 \wedge x_2 \leq_2 y_2)$ où $<_1$ est l'ordre strict associé à $\leq_1$. On note $<_{lex}$ l'ordre strict associé à $\leq_{lex}$, i.e., $(x_1, x_2) <_{lex} (y_1, y_2)$ si et seulement si $x_1 <_1 y_1 \vee (x_1 = y_1 \wedge x_2 <_2 y_2)$. Si $(E_1, <_1)$ et $(E_2, <_2)$ sont bien fondés, le produit cartésien $E_1 \times E_2$ muni de l'ordre lexicographique strict $<_{lex}$ est également bien fondé. Cette notion est généralisable pour tout $n > 2$.

32 – Montrer que la relation $\leq_{lex}$ est bien une relation d'ordre.

33 – Montrer que la relation $<_{lex}$ est bien fondée.

34 – Montrer la terminaison de l'algorithme 10 via une fonction de rang à valeur dans $\mathbb{N} \times \mathbb{N}$ muni de l'ordre lexicographique.

35 – Considérer l'algorithme ci-dessous. Donner un invariant de boucle aussi précis que possible au point (*). Montrer la terminaison. Déterminer les valeurs du couple (x, y) que retourne $lex(a, b)$.

Algorithme 15 : Fonction lex(a,b)

```
Entrées : deux entiers naturels a et b
Sorties : un couple d'entiers
x : entier ← a
y : entier ← b
(*) tant que x > 0 ∨ y > 0 faire
    si y > 0 alors
        y ← y − 1
    sinon
        x ← x − 1
        y ←? ∈ ℕ
    fin
fin
retourner (x, y)
```

VII Ordre lexicographique et fonction de rang à valeur dans $\mathbb{N}$

Pour un ordre lexicographique, en général on perd le lien entre terminaison et complexité. Par exemple, pour l'algorithme 15, il n'est pas possible de majorer le nombre de passages dans la boucle. L'objet de cette section est d'examiner un cas fréquent où il est possible de retrouver ce lien.

Si les composantes du tuple d'expressions qui décroît lexicographiquement sont bornées, il est alors possible de construire une fonction de rang à valeur dans $\mathbb{N}$ au sens de la définition 13.

Théorème 2:

Soit un couple (u, v) où u et v sont des expressions contenant des variables utilisées dans une boucle. Supposons que la valeur du couple (u, v) soit décroissante à chaque répétition de la boucle pour l'ordre lexicographique. Supposons de plus que l'on puisse encadrer u et v respectivement par $min_u \leq u$ et $min_v \leq v \leq max_v$. Alors l'exécution de la boucle termine et une fonction de rang est :

$$f(u, v) = (u - min_u) * (1 + max_v - min_v) + v - min_v.$$

Un résultat similaire peut être établi pour tout type de tuple d'expressions : triplet, quadruplet, etc.

36 – Montrer ce théorème, c'est à dire que si $(u', v') <_{lex} (u, v)$, alors $f(u', v') < f(u, v)$. Indication : évaluer la quantité $f(u, v) - f(u', v')$ suivant les cas.

37 – Qu'effectue l'algorithme suivant ? Montrer qu'au point de programme (*), la propriété $I : 1 \leq i \leq m + 1 \wedge 1 \leq j \leq n$ est un invariant de boucle. Montrer que le couple $(m - i + 1, n - j)$ est bien une fonction de rang à valeur dans $\mathbb{N} \times \mathbb{N}$ qui décroit lexicographiquement. En s'aidant du théorème précédent, donner une fonction de rang à valeur dans $\mathbb{N}$. Déterminer un majorant du nombre de passages dans la boucle.

Algorithme 16 : fonction r2d(b,x)

```
Entrées : une matrice d'entiers b rectangulaire m × n non
          vide et un entier x
Sorties : un booléen
i : entier ← 1
j : entier ← 1
(*) tant que (i ≤ m) ∧ (j ≤ n) ∧ (x ≠ b[i, j]) faire
    j ← j + 1
    si j = n + 1 alors
        i ← i + 1
        j ← 1
    fin
fin
retourner i ≤ m
```

38 – Considérer le programme suivant. Tracer pour $m = 2$ et $n = 3$ (6 affichages). Pour des valeurs de m et n génériques, qu'affiche ce programme ? Trouver un invariant de boucle pour le point de programme (*) décrivant les plages de valeurs de i et j. Trouver un couple (u, v) décroissant qui garantit la terminaison de l'algorithme. En déduire une fonction de rang à valeur $f(i, j)$ à valeur dans $\mathbb{N}$. Déterminer un majorant du nombre de passages dans la boucle. Quelle est la plage des valeurs que retourne cet algorithme ? Si on appelle l'algorithme avec deux entiers naturels, que peut-il se produire ?

Algorithme 17 : parcours étagé

```
Entrées : deux entiers strictement positifs m et
          n
Sorties : un entier
i : entier ← m
j : entier ← n
(*) tant que j ≠ 0 faire
    afficher(i,j)
    j ← j − 1
    si i ≠ 1 ∧ j = 0 alors
        i ← i − 1
        j ← n
    fin
fin
retourner j
```

39 – Considérer le programme suivant. Tracer pour $n = 2$ et $m = 3$ (11 affichages). En s'aidant de la trace précédente, déterminer un invariant de boucle pour le point de programme (*) décrivant les plages de valeurs de i et j. Montrer que le couple $((n+m)-(i+j), i)$ est bien une fonction de rang à valeur dans $\mathbb{N}\times\mathbb{N}$ qui décroit lexicographiquement. En déduire une fonction de rang à valeur dans $\mathbb{N}$. Déterminer un majorant du nombre de passages dans la boucle. Quelle est la plage des valeurs que retourne cet algorithme ?

Algorithme 18 : parcours diagonal

```
Entrées : deux entiers naturels n et m
Sorties : un couple d'entiers
i : entier ← 0
j : entier ← 0
(*) tant que (i < n) ∨ (j < m) faire
    afficher(i,j)
    j ← j + 1
    i ← i − 1
    si (i < 0) ∨ (j > m) alors
        i ← i + j + 1
        j ← 0
        si i > n alors
            j ← i − n
            i ← n
        fin
    fin
fin
retourner (i,j)
```