Schéma itératif [it00]
Support de Cours

Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner

Unisciel algoprog Version 17 mai 2018

Table des matières

Java - Schéma itératif (Cours)

Mots-Clés Schéma itératif, Itération canonique, Notion de récurrence ■
Requis Structures de base, Structures conditionnelles, Algorithmes paramétrés, Structures répétitives ■
Difficulté ●●○ (1 h) ■

Introduction
Ce module traite du **schéma itératif** : notion de récurrence, spécification utilisant les suites récurrentes, déduction du programme itératif.

1 Notion de récurrence

1.1 Induction mathématique

Nous allons utiliser l'induction mathématique pour construire des suites récurrentes exprimant des calculs récurrents ainsi que des algorithmes récurrents associés à une propriété $\mathcal{P}$.

Proposition

Soit $\mathcal{P}(n)$ une propriété pour $n \in \mathbb{N}$.
Si on peut montrer les deux conditions suivantes :

- (H1) $\mathcal{P}(a)$ est vraie pour un entier naturel a.
- (H2) La validité de $\mathcal{P}(n)$ entraîne celle de $\mathcal{P}(n+1)$.

Alors $\mathcal{P}(n)$ est vraie pour tout entier naturel $n \geq a$.

Preuve

Soit M l'ensemble des entiers naturels pour lesquels $\mathcal{P}(n)$ est vraie. D'après l'hypothèse (H1), on a $a \in M$.

Soit alors Q l'ensemble des entiers naturels pour lesquels $\mathcal{P}(n)$ est fausse. Si Q n'est pas vide, il admet un plus petit élément x. On a $x > a$ car $a \in M$. Or $(x-1) \notin Q$. Sinon x ne serait pas le plus petit élément de Q. Ainsi $\mathcal{P}(x-1)$ n'est pas fausse et est donc vraie. Donc $(x-1) \in M$. Mais alors par l'hypothèse (H2), la validité de $\mathcal{P}(x-1)$ entraîne celle de $\mathcal{P}(x)$ qui induit que $(x-1) \in Q$. Contradiction et la conclusion que Q est vide. ■

1.2 Les suites récurrentes

Récurrence directe, d'ordre 1, d'ordre k
Soit une suite $u = (u_n)_{n\in\mathbb{N}}$.
La définition est :

- **Récurrente directe** si l'élément u_n de rang n est fonction de n :

$$u_n = f(n) \text{ pour tout } n \in \mathbb{N}$$

- **Récurrente d'ordre 1** si u_n est fonction de n et de u_{n-1} :

$$\begin{cases} u_n = g(n, u_{n-1}) \text{ pour } n > d \\ u_d = f(d) \text{ est donné} \end{cases}$$

- **Récurrente d'ordre k**, avec $k \geq 1$, si :

$$\begin{cases} u_n = h(n, u_{n-1}, u_{n-2}, ..., u_{n-k}) \text{ pour } n > d + k \\ u_d,\ u_{d+1},\ ...,\ u_{d+k-1} \text{ sont donnés} \end{cases}$$

Exemple
Le factoriel est défini de manière directe par :

$$u_n = n!$$

Et de manière récurrente d'ordre 1 par :

$$\begin{cases} u_n = n\, u_{n-1} \text{ pour } n \geq 1 \\ u_0 = 1 \end{cases}$$

Exemple
Les nombres de Fibonacci sont définis par une suite récurrente d'ordre 2 :

$$\begin{cases} u_n = u_{n-1} + u_{n-2} \quad n \geq 2 \\ u_0 = 0, u_1 = 1 \end{cases}$$

1.3 Systèmes de suites récurrentes

Les problèmes ne s'expriment pas toujours à l'aide d'une suite unique : on est alors amené à considérer des **systèmes** de suites récurrentes. Dans de tels systèmes, le calcul d'un élément d'une suite peut dépendre des éléments d'autres suites.

Définition directe
On rencontre de tels systèmes quand on recherche une définition récurrente équivalente d'une définition directe d'une suite.

Exemple
Le développement limité de l'exponentielle e^x à l'ordre n s'exprime de manière directe par :

$$S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!} \quad x \in \mathbb{R}, \forall n \in \mathbb{N}$$

Ou encore par le système de suites récurrentes :

$$\begin{cases} u_n = x\, u_{n-1} \\ v_n = n\, v_{n-1} \\ S_n = S_{n-1} + \frac{u_n}{v_n} \\ u_0 = 1,\ v_0 = 1,\ S_n = 1 \end{cases}$$

Suite d'ordre k
De tels systèmes apparaissent aussi quand on cherche à transformer une suite récurrente d'ordre k en suites récurrentes d'ordre 1.

Exemple
La suite de Fibonacci, récurrence d'ordre 2 :

$$\begin{cases} u_n = u_{n-1} + u_{n-2} \quad n \geq 2 \\ u_0 = 0, u_1 = 1 \end{cases}$$

peut encore s'écrire :

$$\begin{cases} u_n = u_{n-1} + v_{n-1} \\ v_n = u_{n-1} \\ v_0 = 0,\ u_0 = 1 \end{cases}$$

2 Spécification utilisant les suites récurrentes

La spécification d'un algorithme peut tirer profit de l'existence de suites récurrentes. Dans ce cas, on appliquera la démarche systématique suivante :

1. Définissez le résultat comme un élément d'une suite.
2. Explicitez la récurrence et déterminez les conditions initiales.
3. Prouvez la convergence de l'algorithme.

Définir le résultat

Le résultat doit s'exprimer comme le premier élément d'une suite $u = (u_n)$ vérifiant une condition $B(u_n)$. L'indice n^* de ce premier élément est le plus petit indice n qui vérifie la condition $B(u_n)$.

Si d est la borne inférieure des indices, on doit avoir :

$$\begin{cases} \neg B(u_n) \text{ pour } n \in [d..n^*[\\ B(u_n) \text{ pour } n = n^* \end{cases}$$

Opérateur de minimisation

On formalise cette définition de la manière équivalente suivante :

$$n^* = PPETIT(n \geq d : B(u_n))$$

qui se lit : « n^* est le plus petit n supérieur à d tel que $B(u_n)$ ». L'opérateur `PPETIT` est appelé l'**opérateur de minimisation**. Par commodité d'écriture, on écrit :

$$u^* = u_n^*$$

La solution est le doublet $\langle u^*, n^* \rangle$.

Expliciter la récurrence

Il s'agit d'exprimer le calcul, pas à pas, de chacun des termes de la suite u_{d+1} à partir de u_d, puis u_{d+2} à partir de u_{d+1} et ainsi de suite jusqu'à u^* inclusivement.

Prouver la convergence

Pour prouver la convergence de l'algorithme, il faut montrer que n^* existe. Une méthode consiste à trouver une suite d'entiers $e = (e_n)$ telle que :

$$\begin{cases} 0 < e_n < e_{n-1} \quad \forall n \in [d..n^*[\\ e_{n^*} \leq 0 \end{cases}$$

Résumé de la spécification : SDR

Cette démarche se résume par l'écriture suivante qui définit un **SDR** (Système de Définition Récurrente).

- Critère d'arrêt : $\begin{cases} n^* = PPETIT(n \geq d : B(u_n)) \\ resultat = u^* \end{cases}$
- Itération : $\begin{cases} u_n = g(n, u_{n-1}) \quad \forall n \in [d..n^*[\\ u_d = a \end{cases}$
- Preuve de convergence

Exemple

Cherchons la valeur du plus petit factoriel $u_n = n!$ qui dépasse une valeur L.

1. Définir le résultat : Celui-ci s'exprime comme ceci :

$$\begin{cases} n^* = PPETIT(n \geq 0 : u_n > L) \\ resultat = u^* \end{cases}$$

2. Expliciter la récurrence : La suite $u = (u_n)$ telle que $u_n = n!$ admet la définition récurrente :

$$\begin{cases} u_n = n\, u_{n-1} \quad \forall n > 0 \\ u_0 = 0! = 1 \end{cases}$$

3. Prouver la convergence : La suite d'entiers croît strictement à partir de $n = 1$. Donc quel que soit la valeur L fixée, il est sûr que u_n la dépassera pour un indice n assez grand. La suite d'entiers (e_n) est $e_n = L - u_n$.
4. SDR : Le système qui calcule le plus petit factoriel dépassant $L > 1$ est :
   - Critère d'arrêt : $\begin{cases} n^* = PPETIT(n \geq 0 : u_n > L) \\ resultat = u^* \end{cases}$
   - Itération : $\begin{cases} u_n = n\, u_{n-1} \quad \forall n \in [0..n^*[\\ u_0 = 1 \end{cases}$
   - Preuve de convergence : u_n est une suite d'entiers strictement croissante. Comme $u_0 > 1$, il existe un indice n tel que $u_n > L$.

3 Déduction du programme itératif

L'algorithme sous la forme précédente fournit une solution qui relève du domaine mathématique. Pour passer au programme, il faut :

1. Introduire les variables informatiques en associant une variable à chacune des suites introduites dans l'algorithme.
2. Proposer une séquence de calcul qui respecte l'ordre partiel imposé par les suites et les relations de précédence entre variables informatiques récurrentes.

3.1 Introduction des variables informatiques

Soit (u_n) une suite récurrente d'ordre 1 telle que :

$$u_{n+1} = h(u_n)$$

Si on affecte successivement les valeurs u_n et u_{n+1} à une variable `u`, on obtient :

```
PRECOND u = u_(n)
u <- h(u)
POSTCOND u = u_(n+1)
```

On peut alors :

- Représenter une suite récurrente d'ordre 1 par une variable informatique récurrente : « chaque pas de récurrence affecte une nouvelle valeur à cette variable ».
- Remplacer un système de suites récurrences par autant de variables informatiques qu'il y a de suites récurrentes d'ordre 1 dans le système. (Nous verrons que cela ne suffit pas dans tous les cas et comment y remédier.)

3.2 Séquence de calcul

Lorsque la variable `x` a au moins une occurrence dans la définition de `y|, la valeur de \lstinliney@` dépend de la valeur de `x` et le calcul de `y` n'est possible que si `x` a déjà été calculé.

x précède y

Soient `x` et `y` deux variables. On dit que `x` **précède** `y` lorsque le calcul de `x` doit précéder le calcul de `y`. On représente cette relation par un arc orienté :

Graphe de précédence

On crée ainsi un **graphe de précédence**. Quand le programme ne comporte que des affectations uniques, ce graphe est **sans circuit**.

Cas des variables récurrentes

Soient `x` et `y` deux variables récurrentes.

- Si x_n figure dans la définition de y_n, il faut calculer x_n avant y_n.
  Donc `x` précède `y`.
- Le calcul de y_n détruit la valeur y_{n-1}. Il ne faut donc pas calculer y_n tant que l'on a besoin de y_{n-1}. Si le calcul de x_n dépend de y_{n-1}, il faut calculer x_n avant y_n.
  Donc `y` précède `x`.

On a donc deux relations de précédence :

- `x` précède `y` si x_n figure dans y_n
- `y` précède `x` si y_{n-1} figure dans x_n

L'ensemble des relations pour un système ne sont pas nécessairement compatibles : cette contradiction apparaît sous la forme d'un circuit dans le graphe. Pour lever cette contradiction et obtenir un graphe sans circuit, on introduit une variable supplémentaire par circuit détecté.

Exemple

Soit le système :

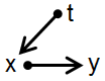
$$\begin{cases} x_n = f(x_{n-1}, y_{n-1}) \\ y_n = g(x_{n-1}, y_{n-1}) \end{cases}$$

Il lui correspond le circuit :

On casse ce circuit en introduisant $t_n = x_{n-1}$. Il vient alors :

$$\begin{cases} t_n = x_{n-1} \\ x_n = f(t_n, y_{n-1}) \\ y_n = g(t_n, y_{n-1}) \end{cases}$$

Le graphe devient alors sans circuit.



Ordonnancement des affectations

Lorsque le graphe est sans circuit, il exprime un ordre partiel entre les diverses affectations du programme. On peut trouver un ordre total compatible en effectuant ce qu'on appelle un **tri topologique**.

Algorithme du tri topologique

Il s'exprime ainsi :

1. Prendre un point sans prédécesseur : c'est un point d'entrée du graphe. L'affectation correspondante est la première du programme représenté par le graphe.
2. Retirer ce sommet du graphe.
3. Recommencer tant que le graphe n'est pas vide.

Si à certaines étapes, il y a plus d'un sommet d'entrée, en choisir un quelconque d'entre eux.

Exemple

L'application du tri topologique à l'exemple précédent donne :

```
t <- x
x <- f(t,y)
y <- g(t,y)
```

3.3 Construction du programme

Démarche systématique

Nous avons maintenant tous les éléments :

1. Déclarer une variable par suite récurrente.
2. Examiner le graphe de précédence et introduire d'autres variables pour rendre le graphe sans circuit.
3. Initialiser les variables.
4. Mettre en place la structure itérative et sa condition de continuation.
   `TantQue Non B(u) Faire S FinTantQue`
5. Écrire le corps de boucle `S`.
6. Définir le résultat.

Règles de précédence entre variables

Tous les systèmes de suites récurrentes ne se ramènent pas à des programmes itératifs. La méthode précédente ne s'applique donc pas toujours. Cependant les systèmes récurrents canoniques permettent de passer directement à l'itération.

Propriété

Un **système récurrent canonique** s'énonce sous la forme :

$$\begin{cases} a_{n+1} = f(a_n, b_n, c_n, ...) \\ b_{n+1} = g(b_n, c_n, ...) \\ c_{n+1} = h(c_n, ...) \\ \vdots \end{cases}$$

Conséquence

Alors pour passer au programme itératif, il suffit d'associer une variable à chacune des suites élémentaires, à condition que le programme respecte l'ordre d'écriture des suites. Si les variables associées sont `A`, `B`, `C`, ..., le corps de boucle d'itération donne :

```
PRECOND A=a_(n), B=b_(n), C=c_(n), ...
A <- f(A, B, C, ...)
B <- g(B, C, ...)
C <- h(C, ...)
...
POSTCOND A=a_(n+1), B=b_(n+1), C=c_(n+1), ...
```

4 Exemples simples

4.1 Somme de nombres

Objectif

Réaliser un programme de type calculatrice pour additionner et soustraire des réels et en calculer la somme. Afin d'éviter à l'utilisateur de compter le nombre de valeurs qu'il souhaite entrer, une valeur spéciale nommée **sentinelle** (ici de valeur nulle) permet de stopper la saisie. Exemple d'exécution :

```
Vos réels [0 pour terminer]
12.5
-10
110.9
-39.22
0
==> La somme totale est 74.18
```

Spécification

On considère la suite $s = (s_j)$ dont la définition directe est :

$$\begin{cases} s_j = nombre_j + s_{j-1} \\ s_0 = 0.0 \\ j^* = PPETIT(j \geq 0 : nombre_j = sentinelle) \end{cases}$$

L'algorithme découle de la spécification.

Programme @[pgsomme.java]

```java
import java.util.Scanner;
import java.util.Locale;

class PGSomme {
final static int sentinelle = 0;

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  input.useLocale(Locale.US);
  double somme = 0.0;
  double nombre;
  System.out.println("Vos reels [" + sentinelle + " pour terminer]");
  nombre = input.nextDouble();
  while (nombre != sentinelle)
  {
    somme += nombre;
    nombre = input.nextDouble();
  }
  System.out.println("==> La somme totale est " + somme);
}

}
```

Explication

Il déclare un réel `somme` pour le résultat, l'initialise à 0.0 puis saisit un réel dans `nombre` et l'ajoute au résultat tant que sa valeur n'est pas égale à la `sentinelle`.

4.2 Somme des entiers de 1 à n

Objectif
Calculer la somme arithmétique de n (entier naturel) définie par :

$$s_n = \sum_{j=1}^{n} j = \frac{1}{2}\, n\,(n+1)$$

Spécification
Cette somme s'obtient en calculant le n^{e} terme d'une suite récurrente définie par :

$$s_n = \left(\sum_{j=1}^{n-1} j\right) + n = s_{n-1} + n \text{ avec } s_0 = 0$$

Programme @[pgsommeN.java]

```java
import java.util.Scanner;

class PGSommeN {

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  int n;
  System.out.print("n? ");
  n = input.nextInt();
  int s = 0;
  for (int j = 1; j <= n; ++j)
  {
    s += j;
    System.out.println("==> somme(1.." + j + ") = " + s);
  }
  System.out.println("==> Par formule " + (n * (n + 1) / 2));
}

}
```

Exemple d'exécution

```
n? 6
==> somme(1..1) = 1
==> somme(1..2) = 3
==> somme(1..3) = 6
==> somme(1..4) = 10
==> somme(1..5) = 15
==> somme(1..6) = 21
==> Par formule 21
```

4.3 Calcul par récurrence d'un maximum

Objectif
Calculer le maximum d'entiers donnés au fur et à mesure.
Exemple d'exécution :

```
n? 5
9
4
12
3
-6
==> Le maximum est 12
```

Spécification
Comment trouver ce maximum ? C'est le plus grand des deux nombres : maximum des $n-1$ premiers entiers donnés, n-ème entier donné.

Une définition par récurrence du maximum est donc :

$$maximum_n = \begin{cases} nombre_n & \text{si } nombre_n > maximum_{n-1} \\ maximum_{n-1} & \text{sinon} \end{cases}$$

Comment initialiser la suite ?
Une solution consiste à initialiser le maximum au premier terme saisi et à commencer la formule générale de récurrence à l'indice 2. Il s'agit d'une **initialisation à un terme utile**. L'algorithme en découle.

Programme @[pgvmax.java]

```java
import java.util.Scanner;

class PGVmax {

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  int n;
  System.out.print("n? ");
  n = input.nextInt();
  int nombre;
  nombre = input.nextInt();
  int vmax = nombre;
  for (int j = 2; j <= n; ++j)
  {
    nombre = input.nextInt();
    if (vmax < nombre)
    {
      vmax = nombre;
```

```
    }
  }
  System.out.println("==> Le maximum est " + vmax);
}

}
```

Explication

`j` est l'indice d'itération, `nombre` le `j`-ème nombre lu et `vmax` le maximum des `j` premiers entiers.

4.4 Sommation

Objectif
Calculer la sommation suivante :

$$s_n = \sum_{j=1}^{n} \sum_{k=1}^{j} (j+k)$$

Spécification
Cette sommation s'obtient en utilisant une boucle imbriquée.

Programme @[pgsommation.java]

```java
import java.util.Scanner;

class PGSommation {

public static void main(String[] args)
{
  Scanner input = new Scanner(System.in);
  int n;
  System.out.print("Sommation jusqu'a? ");
  n = input.nextInt();
  int rs = 0;
  for (int j = 1; j <= n; ++j)
  {
    for (int k = 1; k <= j; ++k)
    {
      rs += j + k;
    }
  }
  System.out.println("==> Sigma(j+k,j=1.." + n + ",k=1..j) vaut " + rs);
}

}
```

Exemple d'exécution

```
Sommation jusqu'à? 4
==> Sigma(j+k,j=1..4,k=1..j) vaut 50
```