Algorithmie et Programmation

Chapitre 4 : les instructions de répétition

Etude de deux cas pratiques

Méthode de construction systématique d'une boucle

L'état du système que nous considérons dans nos problèmes est en général constitué d'états de variables et la boucle la plus simple que l'on puisse envisager est usuellement conditionnée par l'état d'une seule variable.

Pour ces raisons, les boucles que nous envisagerons réellement dans la résolution de nos problèmes sont en réalité un schéma algorithmique composé ayant la forme suivante :

```
var = étatInit;
while Condition(var) {
   Séquence;
   Actualiser(var)
}
```

où : - *var* est la *variable de pilotage* de la boucle,
- *étatInit* est l'état initial de *var*,
- *Condition(var)* est la condition sur *var* contrôlant boucle,
- *Séquence* est la séquence itérée,
- *Actualiser(var)* est l'actualisation de la valeur de *var*.

Une fois qu'on a identifié que la solution du problème (ou d'une partie du problème) nécessitait la mise en œuvre d'un processus itératif, on sait que l'algorithme de résolution correspondant s'exprimera sous la forme générale décrite ci-dessus. Cet algorithme étant posé, il nous restera à résoudre les quatre points suivants, spécifiques à chaque problème :

- identifier la *variable de pilotage* et déterminer sa valeur initiale,
- déterminer la condition de continuation de la boucle ; cette condition s'exprime sur un état de la *variable de pilotage*,
- déterminer la nouvelle valeur à attribuer à la variable de pilotage en vue de l'itération suivante (actualisation de la valeur de la *variable de pilotage*),
- définir le code de la séquence itérée.

Remarque:

Suivant la forme de raisonnement adoptée, on peut être amené à déterminer la condition d'arrêt de la boucle lors de la résolution du point 2. Il est important de noter que la condition de continuation est la négation de la condition d'arrêt ; s'il s'agit d'une conjonction et/ou d'une disjonction, il est donc essentiel de connaître les *lois de Morgan* (ou de bien employer l'opérateur de négation **!**).

Cas pratique 1 : exercice 3, Série 3

Énoncé

Afficher les nombres entiers pairs compris entre 0 et une valeur limite donnée.

Analyse du problème

- La solution de notre problème nécessite la mise en œuvre d'un processus itératif : il faut passer en revue les nombres entiers compris entre 0 et la valeur limite donnée. On a alors :
  1. la *variable de pilotage* est *nbre*, sa valeur initiale est *0*,
  2. la boucle continue tant que *nbre <= limite*,
  3. on doit passer en revue les entiers, l'actualisation est donc *nbre++*,
  4. afficher les *nbre* qui sont pairs.

- Les nombre pairs sont, par définition, les multiples de 2 ; en d'autres termes, un nombre *nbre* est pair si et seulement s'il se divise par deux.

  On a donc : *nbre* pair: *nbre % 2 == 0*

// Version 1

```
int limite = lireLimite();
int nbre = 0;
while (nbre <= limite) {
        if (nbre % 2 == 0) {
                System.out.println(nbre);
        }
        nbre++;
}
```

Pour cette version, on applique *stricto sensu* la définition mathématique d'un nombre pair : on teste la parité de chaque entier compris entre *nbre* et *limite*, on affiche ceux qui satisfont le test de parité.

On peut observer que les nombres entiers se partagent en deux groupes, les nombres pairs et les nombres impairs, à l'exclusion de tout autre. Sachant que les nombre entier sont codé **en complément à deux** on peut donc également définir un nombre pair par le fait que son dernier bit est à 0.

On a donc : *nbre* pair: *(nbre & 1) == 0*

// Version 2

```
int limite = lireLimite();
int nbre = 0;
while (nbre <= limite) {
        if ((nbre & 1) == 0) {
                System.out.println(nbre);
        }
        nbre++;
}
```

Remarquons que ces deux versions sont deux implantations différentes du même algorithme : *on énumère les entiers et on affiche les pairs*. Ce qui les distingue, c'est uniquement le test de parité.

À partir du moment où on a deux versions fonctionnelles différentes d'un programme résolvant le même problème, on peut les comparer. Les éléments de comparaison qu'on peut envisager sont principalement :

- **La simplicité de l'expression** de l'algorithme : l'algorithme est exprimé dans une forme facilement compréhensible pour le lecteur du code.

- **La généralité de l'algorithme** obtenu : l'algorithme obtenu est facilement généralisable à un sur-ensemble(ou sous-ensemble) du problème qu'il résoud.

- **L'élégance, la beauté du code obtenu** : ce sont des éléments principalement subjectifs, mais qu'un programmeur expérimenté reconnaît et apprécie.

- **La performance de l'algorithme** : lorsqu'on déclenche le chronomètre, on peut mesurer qu'un algorithme prend moins de temps que l'autre pour produire les mêmes résultats. La performance d'un algorithme peut évidemment être déduite de façon abstraite du code : on compte alors le nombre d'opérations élémentaires nécessaires (tests, assignations, opérations arithmétiques, etc.) à l'obtention des résultats pour chacun des algorithmes qu'on compare.

En ce qui concerne les deux versions ci-dessus, c'est clairement la version 2 qui est la meilleure du point de vue de la performance : en effet, ce qui les différencie, nous l'avons vu, c'est la manière de tester la parité. Or, dans la version 1, on effectue une opération de division alors que dans la version 2, on teste le bit de parité, ce qu'on obtient facilement en testant uniquement le bit 0 du *nbre*.

Un autre algorithme

On peut observer que les nombres pairs sont à distance 2 les uns des autres. Dans ce cas de figure, contrairement à la version 2 que nous venons d'étudier, on ne remet pas seulement en cause la séquence itérée, mais également les autres éléments de la boucle : **on est bien en présence d'un autre algorithme**. Il convient donc de procéder à une nouvelle analyse du problème :

- La solution de notre problème nécessite toujours la mise en œuvre d'un processus itératif : ce qu'il faut maintenant faire, c'est énumérer les nombres entiers compris entre 0 et la valeur limite donnée en effectuant un pas de 2 à chaque itération. On a alors :
  1. la *variable de pilotage* est *nbre*, sa valeur initiale est *0*,
  2. la boucle continue tant que *nbre <= limite*,
  3. on doit passer en revue les entiers par pas de 2, donc *nbre = nbre + 2*,
  4. afficher *nbre* (qui est forcément pair par construction).

Cette approche se différencie fortement des deux précédentes dans la mesure où on a déterminé ici un **algorithme de construction des nombres pairs**, alors que les algorithmes des deux autres versions énuméraient les nombres entiers pour leur appliquer un test de parité.

```
// Version 3

        int limite = lireLimite();
        int nbre = 0;
        while (nbre <= limite) {
                System.out.println(nbre);
                nbre += 2;
        }
```

Du point de vue de la performance, cette version est bien sûr la plus performante des trois, puisqu'il n'y a plus aucun test dans la séquence itérée ; de plus, la boucle est effectuée deux fois moins souvent.

Généralité des algorithmes étudiés

Il est maintenant temps de se poser la question de la généralisation possible de ces algorithmes à des sur-ensembles du problème initial. Cet aspect ne doit pas être sous-estimé ; c'est grâce à cette qualité qu'on peut réutiliser du logiciel (et donc rentabiliser les investissements en matière grise et en espèces sonnantes et trébuchantes). Considérons donc les variantes suivantes de l'énoncé original :

Énoncé – Variante A
Afficher les multiples de 2, 3 et 5 compris entre 0 et une valeur limite donnée.

Énoncé – Variante B
Afficher les nombres entiers pairs compris entre deux valeurs limites données.

Variante A
La version 1 s'adapte facilement à cette généralisation du problème. On obtient:

```
// Version 1.A

        int limite = lireLimite();
        int nbre = 0;
        while (nbre <= limite) {
                if (nbre % 2 == 0 ||  nbre % 3 == 0 || nbre % 5 == 0 ) {
                        System.out.println(nbre);
                }
                nbre++;
        }
```

La version 2 également. On obtient:

```
// Version 2.A

        int limite = lireLimite();
        int nbre = 0;
        while (nbre <= limite) {
                if ((nbre & 1) == 0 ||  nbre % 3 == 0 || nbre % 5 == 0 ) {
                        System.out.println(nbre);
                }
                nbre++;
        }
```

Pour les mêmes raisons que précédemment la version 2.A est plus performante que la 1.A

Pour ce qui est de la version 3, **cette généralisation n'est tout simplement pas possible**. Cela tient principalement au fait que l'algorithme de la version 3 est très spécialisé : c'est un algorithme de construction des nombres pairs.

Variante B (Afficher les nombres entiers pairs compris entre deux valeurs limites données.)

La version 1 s'adapte à nouveau assez facilement à cette nouvelle situation :

```
// Version 1.B
        int limite = lireLimite();
        int limiteMin = lireLimiteMin();
        nbre = limiteMin;
        while (nbre <= limite) {
                if (nbre % 2 == 0) {
                        System.out.println(nbre);
                }
                nbre++;
        }
```

À part les instructions de lecture des données, il suffit de modifier l'affectation de la valeur initiale de la *variable de pilotage* pour obtenir un algorithme fonctionnel pour ce nouvel énoncé. De même pour la version 2 :

```
// Version 2.B
        int limite = lireLimite();
        int limiteMin = lireLimiteMin();
        nbre = limiteMin;
        while (nbre <= limite) {
                if ((nbre & 1) == 0){
                        System.out.println(nbre);
                }
                nbre++;
        }
```

La version 3 nécessite un peu plus d'effort. Il faut en effet affecter une valeur initiale correcte à la *variable de pilotage*. L'algorithme que nous avons développé est basé sur le fait que l'énumération des pairs commence à partir de 0, soit à partir d'un nombre pair. Avec le nouvel énoncé, nous ne connaissons pas d'avance la parité de la borne inférieure, car celle-ci est saisie. Il faut donc développer un petit algorithme qui affecte une valeur initiale paire adéquate à la *variable de pilotage* à partir d'une donnée quelconque.

```
// Version 3.B
        int limite = lireLimite();
        int limiteMin = lireLimiteMin();
        if ((limiteMin & 1) == 0) {
                nbre = limiteMin;
        } else {
                nbre = limiteMin+1;
        }

        while (nbre <= limite) {
                System.out.println(nbre);
                nbre += 2;
        }
```

Conclusions

- Le premier algorithme (exprimé dans les versions 1 & 2) est finalement celui qui se généralise le mieux ; nous avons pu observer qu'il était facilement adaptable à diverses extensions du problème original. De ce point de vue, on peut considérer qu'il est meilleur que le second (exprimé dans la version 3).

- Il est normal, dans un certain sens, que l'algorithme de la version 3 soit celui qui se généralise le moins bien ; en effet, il s'agit d'un algorithme très spécialisé, conçu spécifiquement pour résoudre le problème de l'énoncé original.

- L'algorithme spécialisé (version 3) est aussi le plus performant dans le cas de l'énoncé original. Il l'est également dans la variante B de cet énoncé. En effet, le test supplémentaire nécessaire à l'assignation de la valeur initiale à la *variable de pilotage* n'est effectué qu'une seule fois, à l'extérieur de la boucle alors que les deux autres versions effectuent leur test de parité à chaque tour de cette boucle, boucle par ailleurs effectuée deux fois plus souvent par ces versions.

Cas pratique 2 : exercice 5, Série 3 (variante avec la moyenne)

Énoncé

Calculer et afficher le plus petit et le plus grand nombre d'une liste de 10 nombres entiers. Calculer et afficher également la moyenne des 10 nombres.

Analyse du problème

- Les valeurs extrêmes recherchées sont mémorisées dans les variables *petit* et *grand* qui contiennent, à tout moment du traitement, le plus petit, respectivement le plus grand nombre rencontré dans la liste des valeurs déjà lues. Leurs valeurs initiales sont définies par la valeur du premier nombre de la liste.

- La somme des valeurs déjà lues est mémorisée dans l'***accumulateur*** *somme* ; sa valeur intiale est donc également la valeur du premier nombre de la liste.

- La solution de notre problème nécessite la mise en œuvre d'un processus itératif : il faut lire les 9 nombres restants (le premier est lu à l'extérieur de la boucle pour fixer les valeurs initiales des variables *petit*, *grand* et *somme*). On a alors :

  - la *variable de pilotage* est *i*, sa valeur initiale est *0*,
  - la boucle continue tant que *i < 9* puisqu'il y a 9 itérations à compter,
  - on doit compter les itérations, l'actualisation est donc *i++*,
  - lire le nombre suivant (*nb*) et le traiter.

- Le traitement d'un nombre consiste à :

  1. comparer *nb* à *petit* ; si *nb* est inférieur à *petit*, mémoriser la valeur de *nbreLu* comme la nouvelle valeur de *petit*,

  2. comparer *nb* à *grand* ; si *nb* est supérieur à *grand*, mémoriser la valeur de *nb* comme la nouvelle valeur de *grand*,

  3. accumuler la valeur de *nb* dans *somme*.

- Finalement, la valeur de la moyenne est obtenue en divisant *somme* par *10*.

```
// Version 1
    int nb = lireNb();
    int petit = nb;
    int grand = nb;
    int somme = nb;
    for (int i = 0; i < 9; i++){
        nb = lireNb();
        if (nb < petit) petit = nb;
        if ((nb > grand) grand = nb;
        somme += nb;
    }
    double moyenne = somme / 10d;
```

On peut également approcher le problème des valeurs initiales des variables *petit*, *grand* et *somme* d'une autre manière. Au lieu de lire la première valeur séparément des autres à l'extérieur de la boucle, on affecte à *petit* la plus grande valeur possible pour un entier (*Integer.MAX_VALUE*), à *grand* la plus petite valeur possible pour un entier (*Integer.MIN_VALUE*) et à *somme* la valeur *0*. On doit maintenant effectuer 10 itérations avec la boucle pour traiter les 10 nombres de la liste ; le traitement effectué est celui décrit précédemment. Lors du premier tour de la boucle, les variables *petit* et *grand* prendront automatiquement une valeur adéquate, leurs valeurs initiales ayant judicieusement été choisies. On obtient alors :

```
// Version 2
    int nb;
    int petit = Integer.MAX_VALUE;
    int grand = Integer.MIN_VALUE;
    int somme = 0;
    for (int i = 0; i < 10; i++){
        nb = lireNb();
        if (nb < petit) petit = nb;
        if (nb > grand) grand = nb;
        somme += nb;
    }
    double moyenne = somme / 10d;
```

Considérons à nouveau la version 1 **du point de vue de la performance** et demandons-nous si nous ne pouvons pas optimiser le code que nous avons obtenu. On peut observer que la procédure a comme **invariant** la relation suivante : *petit* < *grand*.

Avoir cette relation comme **invariant** signifie que celle-ci est vraie à tout moment de l'exécution du code de la procédure. On peut donc exploiter ce fait en remarquant que si *nb* est inférieur à *petit*, il n'est pas nécessaire de tester s'il est supérieur à *grand* car on a alors *nb* < *petit* <*grand* . On obtient alors :

```
// Version 1.1
      int nb = lireNb();
      int petit = nb;
      int grand = nb;
      int somme = nb;
      for (int i = 0; i < 9; i++){
        nb = lireNb();
        if (nb < petit) {
                petit = nb;
        } else if (nb > grand) {
                grand = nb;
        }
        somme += nb;
      }
      double moyenne = somme / 10d;
```

Comparaison des performances des versions 1 et 1.1

La différence entre les deux versions réside dans la manière de traiter *nb*, en particulier dans les tests qu'on effectue pour le comparer à *petit* et à *grand*. Nous allons donc compter, pour chacune des deux versions, le nombre de tests effectués lors du traitement des 10 nombres de la liste. Comme les autres opérations sont identiques pour les deux versions, nous pouvons les omettre dans notre analyse.

Version 1
La boucle est effectuée 9 fois ; elle contient deux tests en séquence. Il faut donc 18 tests, quelles que soient les données, pour effectuer le traitement des 10 nombres de la liste.

Version 1.1
La boucle est également effectuée 9 fois. Ici, le problème est un peu plus délicat dans la mesure où, suivant les valeurs actuelles de *nb* et de *petit*, on effectue soit 1 test, soit 2 tests lors d'un tour de la boucle. La performance globale dépend donc des données. Deux cas extrêmes se présentent :

1. le meilleur des cas (best case) : les valeurs des données sont en **ordre décroissant** ; dans ce cas, on effectue un seul test à chaque tour de la boucle et donc 9 tests sont nécessaires pour effectuer le traitement des 10 nombres de la liste,

2. le pire des cas(worst case) : les valeurs des données sont en **ordre croissant** ; dans ce cas, on effectue deux tests à chaque tour de la boucle et donc 18 tests sont nécessaires pour effectuer le traitement des 10 nombres de la liste.

On voit donc qu'au pire, on effectue autant de tests que dans le cas de la version 1, et qu'au mieux, on en effectue 2 fois moins.

Si maintenant on peut établir que toutes les configurations possibles de données sont équiprobables (on a autant de chance de rencontrer n'importe laquelle des configurations possibles de données), on a le droit de calculer une performance moyenne en effectuant la moyenne arithmétique des performances des deux cas extrêmes ; on obtient alors qu'avec la version 1.1, il faut en moyenne 13.5 tests pour traiter les 10 nombres de la liste.

Nous avons observé que la version 1.1 était plus performante que la version 1.
Appliquons donc la même amélioration à la version 2 :

```
// Version 2.1
      int nb;
      int petit = Integer.MAX_VALUE;
      int grand = Integer.MIN_VALUE;
      int somme = 0;
      for (int i = 0; i < 10; i++){
         nb = lireNb();
         if (nb < petit) {
               petit = nb;
         } else if (nb > grand) {
               grand = nb;
         }
         somme += nb;
      }
      double moyenne = somme / 10d;
```

Testons cette nouvelle version en la soumettant à un certain nombre de données de test:

```
Parmi les 10 nombres suivants:   1   2   3   4   5   6   7   8   9  10
   Le plus petit est 1
   Le plus grand est 10
   La moyenne est 5.5

   Parmi les 10 nombres suivants:
    -22   2  34  -4  51  12  45   8 -17  14
   Le plus petit est -22
   Le plus grand est 51
   La moyenne est 12.3

   Parmi les 10 nombres suivants:
     1   2   1  24   5  24   7  12   9  10
   Le plus petit est 1
   Le plus grand est 24
   La moyenne est 9.5
```

Voilà qui a l'air tout à fait satisfaisant ! Examinons encore les résultats de l'exécution de cette version 2.1 avec un autre jeu de données de test :

```
   Parmi les 10 nombres suivants:
     10   8   6   4   2   1   3   5   7   9
   Le plus petit est 1
   Le plus grand est 9
   La moyenne est 5.5

   Parmi les 10 nombres suivants:
     10   9   8   7   6   5   4   3   2   1
   Le plus petit est 1
   Le plus grand est -2147483648
   La moyenne est 5.5
```

Là, les résultats sont carrément faux ! À quoi cela est-il dû ?
Le problème est que dans cette version du programme, la relation *petit* < *grand* **n'est pas un invariant** pour l'ensemble de la procédure ; cette relation n'est en particulier pas

vraie la première fois qu'on entre dans la boucle. Or, le raisonnement que nous avions effectué pour construire la version 1.1 à partir de la version 1 se fondait justement sur le fait que cette relation était un invariant pour toute la procédure.

La version 2.1 n'est donc pas une solution acceptable au problème posé.

Conclusions

- Lors de la construction et/ou de l'amélioration d'un programme, il est essentiel d'appliquer à tout moment un raisonnement rigoureux. Nous avons obtenu un programme faux en transposant un raisonnement que nous avions effectué pour un autre programme et en négligeant le fait que les conditions initiales de ce raisonnement n'étaient plus les mêmes.

- Lors du test d'un programme, il convient de construire le jeu des données de test avec soin. Ici, nous avons exécuté notre programme faux avec diverses données de test pour lesquelles l'exécution donnait des résultats corrects. Il ne suffit donc pas d'essayer le programme conçu une ou deux fois ; il faut en vérifier abstraitement le code et construire un jeu de données de test si possible exhaustif (la construction de jeux de tests est en réalité un problème très complexe).

À propos de la *variable de pilotage* de la boucle

Dans les versions 1 et 1.1 présentées ci-dessus, la *variable de pilotage* de la boucle *i* indique le nombre exact d'itérations effectuées par la boucle. Lorsqu'on en a effectué 9, la boucle s'arrête.
On pourrait également prendre comme *variable de pilotage* de la boucle une variable indiquant le nombre exact de nombres qui ont déjà été traités. On obtient alors :

```
// Version 1.2
        int nb = lireNb();
        int petit = nb;
        int grand = nb;
        int somme = nb;
        for (int nbNbresTraites = 0; nbNbresTraites < 10; nbNbresTraites++){
            int nb lireNb();
            if (nb < petit) {
                    petit = nb;
            } else if (nb > grand) {
                    grand = nb;
            }
            somme += nb;
        }
        double moyenne = somme / 10d;
```

Lorsqu'on change ainsi de point de vue sur le sens de la *variable de pilotage*, il est **fondamental** d'adopter un nom pour cette variable qui corresponde à son sens dans le programme ; ici nous avons choisi de la nommer *nbNbresTraites*. Un code identique avec *i* comme nom de variable serait bien sûr fonctionnel et correct, mais aurait l'inconvénient d'être moins lisible, aussi bien pour un lecteur différent du concepteur du programme que pour le concepteur lui-même, quelques temps après la rédaction du code. D'autres options sont possibles ; on pourrait par exemple prendre comme *variable de pilotage* une variable indiquant le numéro du prochain nombre à traiter . Là encore, on choisirait un nom qui indique clairement le rôle de cette variable.