Algorithmes itératifs

Olivier Cogis, Jérôme Palaysi

Algorithmes itératifs

Olivier Cogis
oliviercogis@gmx.fr

Jérôme Palaysi
palaysi@lirmm.fr

23 décembre 2021

Résumé

Cet article est une présentation de ce qu'on appelle les algorithmes itératifs en Informatique. Il est destiné aux étudiants de niveau Licence ou Master en Informatique, notamment à ceux préparant un CAPES d'informatique, comme aux enseignants du secondaire qui souhaitent accompagner l'apparition de la discipline Informatique au lycée.

— En guise de préambule, une version formelle de l'algorithme de la multiplication Russe est présentée avec une trace d'exécution.

— La deuxième section définit justement les instructions de base de l'algorithmique pour nos machines.

— La troisième section s'intéresse aux concepts de variants et d'invariants permettant de prouver qu'un algorihtme fonctionne conformément à ses spécifications.

— La quatrième section s'intéresse aux cas des algorithmes utilisant des instructions ou des fonctions qui peuvent avoir des résultats différents pour une même donnée (tout en respectant leur spécification), ce qu'on appelle algorithmes non déterministes.

Il est suivi d'une brève conclusion et d'une annexe contenant quelques implémentations en Python.

Table des matières

1 Préambule

Pour servir de base aux définitions et développements qui vont suivre, nous soumettons d'abord une version en pseudo-code [1] d'une technique de multiplication de deux entiers dite *multiplication russe* [2]. Elle s'inspire d'une technique de décomposition des entiers sur les puissances de 2, connue des Égyptiens il y a quelques 4000 ans [3], lorsqu'il apparut que la méthode des additions successives rencontrait vite ses limites [4]

1. `https://fr.wikipedia.org/wiki/Pseudo-code`. Nous préciserons au fur et à mesure la syntaxe et la sémantique du pseudo-code particulier que nous utilisons.
2. `https://fr.wikipedia.orhg/wiki/Technique_de_multiplication_dite_russe`.
3. `https://fr.wikipedia.org/wiki/Papyrus_Rhind`.
4. Aisé pour 3 fois 12, fastidieux pour 12 fois 12, décourageant pour 123 fois 321.

Données : deux entiers a et b $(a, b \geq 0)$
Résultat : retourne la valeur du produit ab

```
1 x ← a
2 y ← b
3 z ← 0
4 tant que x ≠ 0 faire
5 |   si x est impair alors
6 |   |   z ← z + y
7 |   x ← x ÷ 2
8 |   y ← y × 2
9 retourner z
```

Algorithme 1 : *Multiplication Russe* : une version en pseudo-code de la multiplication du paysan russe.

Les instructions d'un algorithme ainsi donné doivent être exécutées en séquence, dans leur ordre d'écriture (la numérotation des lignes n'est là que pour permettre de les désigner dans d'éventuels commentaires de l'algorithme) :

— le symbole ÷ représente la division entière [5] ;

— la flèche « ← » signifie qu'il faut évaluer ce qui se trouve à sa droite et affecter la valeur trouvée à la variable figurant à sa gauche ;

— « **tant que** ... **faire** ... » signifie qu'il faut répéter sous condition ce qui est sous sa portée (indiqué ici par l'indentation des instructions et par la grande barre verticale) ;

— « **si** ... **alors** ... » signifie qu'il faut exécuter ce qui est sous sa portée (l'indentation et la petite barre verticale indiquent que seule l'affectation $z \leftarrow z + y$ est concernée) à la seule condition que x soit un nombre impair ;

— « **retourner** z » signifie que la valeur de z est à transmettre comme résultat des calculs effectués.

Ces précisions devraient permettre à chacun de faire la *trace* de l'exécution de l'algorithme de multiplication avec 50 et 89 comme valeurs respectives pour a et b, trace que nous représentons au moyen des tableaux des figures 1 page suivante et 2 page 5 (le second est un résumé du premier).

5. Soit $6 \div 2 = 7 \div 2 = 3$.

instruction ou expression	**évaluation**	**x**	**y**	**z**
$x \leftarrow a$		50	—	—
$y \leftarrow b$		50	89	—
$z \leftarrow 0$		50	89	0
$x \neq 0$	$vrai$	50	89	0
x est impair	$faux$	50	89	0
$x \div 2$	25	50	89	0
$x \leftarrow x \div 2$		25	89	0
$y \times 2$	178	25	89	0
$y \leftarrow y \times 2$		25	178	0
$x \neq 0$	$vrai$	25	178	0
x est impair	$vrai$	25	178	0
$z + y$	178	25	178	0
$z \leftarrow z + y$		25	178	178
$x \div 2$	12	25	178	178
$x \leftarrow x \div 2$		12	178	178
$y \times 2$	356	25	178	178
$y \leftarrow y \times 2$		12	356	178
$x \neq 0$	$vrai$	12	356	178
x est impair	$faux$	12	356	178
$x \div 2$	6	12	356	178
$x \leftarrow x \div 2$		6	356	178
$y \times 2$	712	6	356	178
$y \leftarrow y \times 2$		6	712	178

instruction ou expression	**évaluation**	**x**	**y**	**z**
$x \neq 0$	$vrai$	6	712	178
x est impair	$faux$	6	712	178
$x \div 2$	3	6	712	178
$x \leftarrow x \div 2$		3	712	178
$y \times 2$	1424	3	712	178
$y \leftarrow y \times 2$		3	1424	178
$x \neq 0$	$vrai$	3	1424	178
x est impair	$vrai$	3	1424	178
$z + y$	1602	3	1424	178
$z \leftarrow z + y$		3	1424	1602
$x \div 2$	1	3	1424	1602
$x \leftarrow x \div 2$		1	1424	1602
$y \times 2$	2848	1	1424	1602
$y \leftarrow y \times 2$		1	2848	1602
$x \neq 0$	$vrai$	1	2848	1602
x est impair	$vrai$	1	2848	1602
$z + y$	4450	1	2848	1602
$z \leftarrow z + y$		1	2848	4450
$x \div 2$	0	1	2848	4450
$x \leftarrow x \div 2$		0	2848	4450
$y \times 2$	5696	0	2848	3200
$y \leftarrow y \times 2$		0	5696	4450
$x \neq 0$	$faux$	0	5696	4450

FIGURE 1: Trace détaillée de l'exécution de l'algorithme de *Multiplication Russe* avec $a = 50$ et $b = 89$.

2 Instructions

Nous définissons les *instructions composées* comme étant construites à partir d'autres instructions. Il y faut donc des instructions permettant d'initialiser cette construction : les *instructions élémentaires*.

2.1 Une instruction élémentaire : l'affectation

Nous choisissons de disposer d'une seule instruction élémentaire : l'*affectation*.

La structure, ou encore la syntaxe, d'une *affectation* est :

<nom de variable> ← <expression>

dont l'exécution, ou encore la *sémantique*, consiste à :

1) évaluer l'expression figurant au membre droit de l'affectation ;
2) affecter le résultat de cette évaluation à la variable figurant au membre gauche de l'affectation.

	x	y	z
initialisation	50	89	0
itérations			
1	25	178	0
2	12	356	178
3	6	712	178
4	3	1424	178
5	1	2848	1602
6	0	5696	4450

FIGURE 2: Trace résumée de l'exécution de l'algorithme de multiplication avec $a = 50$ et $b = 89$.

2.2 Instructions séquentielles

Une *instruction séquentielle*, ou *séquence d'instructions*, est une suite finie d'instructions pour laquelle il est clairement établi :

— quelles sont les instructions qui la constituent ;

— quel est leur ordre dans la suite.

toute latitude étant laissée quant à la forme du pseudo-code choisie pourvu que ces deux points soient clairement établis [6].

Exécuter une séquence d'instructions consiste à exécuter dans l'ordre prescrit chacune des instructions de la séquence.

2.3 Instructions répétitives

2.3.1 Définition

Une *instruction répétitive* est une instruction de la forme

tant que C **faire** I **fin**

où :

— **tant que**, **faire** et **fin** sont des mots-clés ;

— C, qu'on appelle parfois la *condition de contrôle* de la répétitive, est une expression booléenne quelconque [7]

6. L'indentation, l'usage de barres verticales ou d'encadrés sont des outils graphiques privilégiés. Mis à part *Python* (indentation), les langages de programmation en proposent généralement d'autres (balisage par accolades ouvrantes/fermantes, balisage par mots-clés...).

7. Autrement dit, qui s'évalue à *vrai* ou à *faux*.

— I, qu'on appelle parfois le *corps* de la répétitive, est une instruction quelconque [8].

Exécuter une répétitive consiste à :

1) évaluer l'expression booléenne C ;
2) selon le résultat de cette évaluation :
   — si le résultat obtenu est la valeur *faux*, ne rien faire (c'est-à-dire exécuter l'action de base *rien*) et considérer que la répétitive a bien été exécutée ;
   — si le résultat obtenu est la valeur *vrai*, exécuter l'instruction I – on dit également *exécuter une itération* – et reprendre le processus au point 1.

Ce qu'on peut représenter par le schéma de la figure 3.

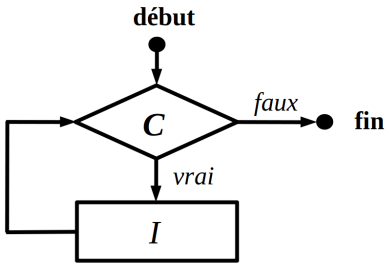

FIGURE 3: Schéma d'exécution d'une instruction répétitive.

2.3.2 Le problème de la terminaison

Considérons la répétitive :

tant que $x > 0$ **faire** $x \leftarrow x + 2$ **fin**

Elle est en soi d'un intérêt douteux, mais elle est syntaxiquement correcte et, en conséquence, elle permet de poser la question : que peut-il se passer lors de son exécution ?

1. Si x a pour valeur -3, l'exécution de la répétitive consistera à :
   (a) évaluer la condition de contrôle de la répétitive, laquelle sera évaluée à *faux* ;
   (b) considérer que l'exécution de la répétitive est terminée.
2. Si x a pour valeur 3, l'exécution de la répétitive consistera à :

8. Soulignons-le : pas nécessairement une instruction élémentaire (voir par exemple la répétitive de l'algorithme 1 page 3).

(a) évaluer la condition de contrôle de la répétitive, laquelle est évaluée à *vrai* ;

(b) exécuter une itération, c'est-à-dire ajouter 2 à la valeur de la variable x qui a maintenant 5 pour valeur ;

(c) évaluer la condition de contrôle de la répétitive, laquelle est évaluée à *vrai* ;

(d) exécuter une itération, c'est-à-dire ajouter 2 à la valeur de la variable x qui a maintenant 7 pour valeur ;

(e) évaluer la condition de contrôle de la répétitive, laquelle est évaluée à *vrai* ;

(f) ...

... et l'exécution de la répétitive n'est jamais terminée.

De ces deux cas limites, l'exécution de 0 itérations et l'exécution d'une infinité d'itérations, seul le premier cas est acceptable tandis que le second invalide l'algorithme.

Ainsi appartient-il à l'auteur d'un algorithme de démontrer que celui-ci *termine*[9], c'est-à-dire que son exécution se termine au bout d'un nombre fini d'itérations pour tout jeu de données valide pour l'algorithme.

Pour enfoncer le clou, signalons que dans les cas que nous envisageons dans cet ouvrage le problème de la terminaison se résout sans effort, voire au moyen d'un effort très raisonnable[10], mais qu'il peut parfois se révéler d'une difficulté hors norme (voir l'encadré sur la suite de Syracuse).

9. Nous avons songé à utiliser l'expression « l'algorithme converge », mais la connotation au modèle du continu nous a paru trop forte. Qu'il nous soit permis d'introduire, au seul bénéfice des algorithmes, un usage intransitif du verbe *terminer*.

10. Pour l'algorithme *Multiplication Russe* (algorithme 1 page 3), la terminaison de la répétitive peut s'argumenter comme suit :

— x est initialisé à une valeur entière positive en respect de la rubrique *donnée* des spécifications ;

— la suite des valeurs prises par x à chaque itération est une suite monotone strictement décroissante d'entiers naturels puisqu'on a toujours $0 \leq x \div 2 < x$ pour tout entier $x > 0$, les seules valeurs de x pour lesquelles l'exécution d'une nouvelle itération est requise ;

— or toute suite d'entiers naturels strictement décroissante est nécessairement finie.

La suite de Syracuse.
La répétitive ci-dessous a pour objectif de calculer les valeurs de termes consécutifs de la célèbre suite dite *suite de Syracuse* [a].

tant que $x \neq 1$ **faire**
  si x est pair **alors**
    $x \leftarrow x \div 2$
  sinon
    $x \leftarrow (3x + 1) \div 2$
  fin

On ne connaît pas de valeur entière positive de x pour laquelle cette répétitive ne *termine* pas et il a été conjecturé qu'il n'en existe pas, une conjecture, à ce jour et à notre connaissance, toujours ouverte.

a. <`https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse`>. Pour éviter toute ambiguïté, signalons que la définition de cette suite est postérieure de quelques 22 siècles au génial Archimède… qui, en l'espèce, n'y est pour rien.

2.4 Instructions conditionnelles

On peut définir des instructions conditionnelles de formes variées. Nous nous contenterons de n'en présenter que les deux formes les plus classiques.

La conditionnelle pure. Une *instruction conditionnelle pure* est une instruction de la forme « **si** C **alors** I **fin** » où :

- **si**, **alors** et **fin** sont des mots-clés ;
- C, qu'on appelle parfois la *condition de contrôle* de la conditionnelle, est une expression booléenne quelconque ;
- I est une instruction quelconque [11] ;

Exécuter une conditionnelle pure consiste à :

1) évaluer l'expression booléenne C ;
2) selon le résultat de cette évaluation :
   - si le résultat obtenu est la valeur *vrai*, exécuter l'instruction I ;
   - si le résultat obtenu est la valeur *faux*, ne rien faire (c'est-à-dire exécuter l'instruction élémentaire *nulle*).

Ce qu'on peut représenter par le schéma de la figure 4 page suivante.

11. Soulignons-le : pas nécessairement une instruction élémentaire.

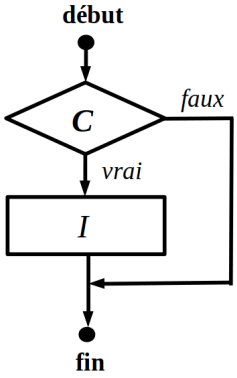

FIGURE 4: Schéma d'exécution d'une instruction conditionnelle pure.

L'alternative. Une *instruction alternative* est une instruction de la forme « **si** C **alors** I_1 **sinon** I_2 **fin** » où :

— **si**, **alors**, **sinon** et **fin** sont des mots-clés ;

— C est une expression booléenne quelconque ;

— I_1 et I_2 sont des instructions quelconques.

Exécuter une alternative consiste à :

1) évaluer l'expression booléenne C ;
2) selon le résultat de cette évaluation :
   — si le résultat obtenu est la valeur *vrai*, exécuter l'instruction I_1 ;
   — si le résultat obtenu est la valeur *vrai*, exécuter l'instruction I_2.

Ce qu'on peut représenter par le schéma de la figure 5 page suivante.

Digression. On constate que les deux instructions :

1. **si** C **alors** I **fin**
2. **si** C **alors** I **sinon** *nulle* **fin**

s'interprètent de la même façon, ce qui fait qu'on peut voir la conditionnelle pure comme un cas particulier de l'alternative. Nous préférons conserver l'idée de conditionnelle pure pour ce qu'elle permet l'expression d'une marque d'intention : proposer une alternative n'a pas tout à fait le même sens que de soumettre une exécution à une éventualité [12].

12. Exemple de conditionnelle pure : « si M. X me téléphone, venez me chercher en salle de réunion ». Et sinon ?

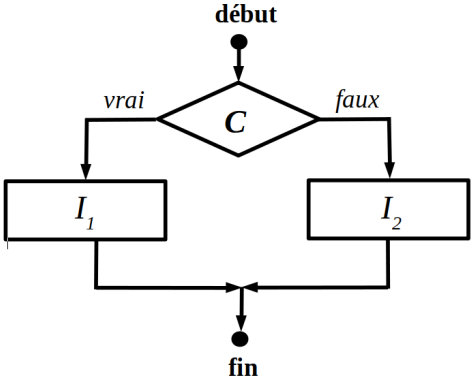

FIGURE 5: Schéma d'exécution d'une instruction alternative.

2.5 Autres instructions

Il est possible de définir d'autres instructions, notamment de types répétitives et conditionnelles [13]. Leur intérêt tient à ce qu'elles permettent de rendre plus directement compte des intentions du concepteur tout en lui en facilitant l'expression.

S'il n'entre pas dans notre intention d'en dresser un panorama, nous nous réservons néanmoins d'en introduire de nouvelles lorsque l'occasion s'en présentera. À titre d'illustration, l'instruction conditionnelle suivante s'interprète plus directement qu'une imbrication équivalente de *si ... alors ... sinon* :

selon *mois*
  cas 1, 3, 5, 7, 8, 10, 12 **alors** $nombreDeJours \leftarrow 31$
  cas 4, 6, 9, 11 **alors** $nombreDeJours \leftarrow 30$
  cas 2 **alors**
    si *année* est bissextile
    alors $nombreDeJours \leftarrow 29$
    sinon $nombreDeJours \leftarrow 28$

2.6 Décomposition des instructions

2.6.1 Décomposition

Les instructions peuvent avoir une structure aussi complexe qu'on veut. Il convient donc de savoir les décomposer.

13. Les langages de programmation en proposent de nombreuses variantes.

À titre d'exemple, décomposons l'instruction constituant le corps de l'algorithme de multiplication sans tables donné dans sa version *Multiplication Russe*[14].

Cette instruction est une séquence de 4 instructions :

1. Les trois premières sont des instructions élémentaires (donc non décomposables), ce sont des affectations.
2. La quatrième est une répétitive dont le corps est une instruction qui est elle-même une séquence de 3 instructions :
   (a) la première est une conditionnelle pure portant sur une instruction élémentaire qui est une affectation ;
   (b) les deux suivantes sont des instructions élémentaires qui sont des affectations.

Ce qui peut être schématisé dans la figure 6.

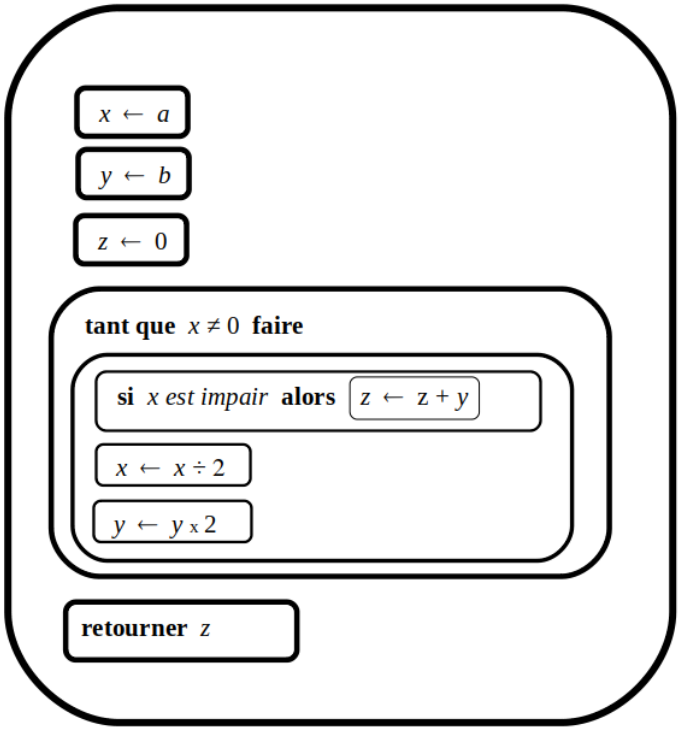

FIGURE 6: Décomposition du corps de l'algorithme de la multiplication du paysan russe reformulée (algorithme 1 page 3).

14. Algorithme 1 page 3.

2.6.2 Pseudo-code et unicité de la décomposition.

Nous avons opté pour une forme particulière de présentation du pseudo-code qui permet de différentier les deux instructions qui diffèrent par leur sémantique :

1.
si $C1$ **alors**
  si $C2$ **alors**
    I_1
  fin
sinon
  I_2
fin

2.
si $C1$ **alors**
  si $C2$ **alors**
    I_1
  sinon
    I_2
  fin
fin

Le lecteur peut se convaincre que supprimer l'usage du mot-clé **fin** en conservant l'indentation ou supprimer l'indentation en conservant les mots-clé **fin** permet encore de différentier les deux instructions.

En revanche, leur suppression conjointe qui aboutit à la forme linéarisée :

$$\textbf{si}\ \ C_1\ \ \textbf{alors si}\ \ C_2\ \ \textbf{alors}\ \ I_1\ \ \textbf{sinon}\ \ I_2$$

est sujette à deux décompositions qu'on peut indifféremment identifier à chacune de ces deux instructions selon qu'on la décode sous l'une des schémas :

1. **si** C_1 **alors** $\left[\ \textbf{si}\ \ C_2\ \ \textbf{alors}\ \ I_1\ \right]$ **sinon** I_2
2. **si** C_1 **alors** $\left[\ \textbf{si}\ \ C_2\ \ \textbf{alors}\ \ I_1\ \ \textbf{sinon}\ \ I_2\ \right]$

Il appartient donc à chacun de veiller à ce que, quel que soit le choix fait pour la présentation d'une instruction, sa décomposition soit non ambiguë.

3 Algorithmes itératifs

3.1 Un cadre pour fixer les idées

Nous proposons de considérer un *algorithme* comme étant constitué :

— d'un *corps* qui est une instruction au sens de la section *Instructions*[15] ;

— de *spécifications* comportant au moins une rubrique *donnée* et une rubrique *résultat* ;

— d'un nom et d'une éventuelle instruction de *retour* utiles à la *composition des algorithmes* sur laquelle nous revenons dans une section ultérieure[16].

En conséquence, le corps de l'algorithme peut être :

— une *instruction élémentaire* ;

— une *instruction séquentielle* ;

— une *instruction répétitive* ;

— une *instruction conditionnelle* ;

ou encore un *appel d'algorithme*[17].

Dans l'esprit de la section *Autres instructions*[18], cette liste peut éventuellement être augmentée.

3.2 Justification des algorithmes itératifs

3.2.1 Motivation

Les spécifications d'un algorithme, pour autant qu'elles comportent les rubriques *donnée* et *résultat*, permettent d'utiliser celui-ci comme une « boîte noire », genre : il résulte ceci et cela de son exécution si on lui fournit telle et/ou telle donnée.

Il appartient à l'auteur de l'algorithme non seulement de se porter garant de ce qu'on pourrait considérer comme un contrat passé avec ses utilisateurs, mais évidemment de pouvoir aussi le justifier.

3.2.2 La justification des algorithmes itératifs en général

Elle relève d'un schéma incontournable, à savoir que la justification d'un algorithme itératif se doit de comporter deux étapes indépendantes, chacune indispensable, qui consistent respectivement à montrer que, quelles que soient les valeurs des données respectant les spécifications :

15. Section 2 page 4.
16. Section *Composition des algorithmes*, section 3.3 page 28.
17. Ibid.
18. Section 2.5 page 10.

1. L'algorithme *termine*[19], c'est-à-dire que chacune des répétitives est assurée de n'exécuter qu'un nombre fini d'itérations.
2. L'algorithme *satisfait*[20], c'est-à-dire que lorsque son exécution est terminée, la clause *résultat* de ses spécifications est vérifiée.

Dans cette perspective, les spécifications des algorithmes se révèlent être un outil fondamental, méritant donc la plus grande attention.

3.2.3 Quand « ça va de soi ».

Considérons l'algorithme ci-dessous :

```
   Données : un entier positif ou nul n
   Résultat : retourne vrai si l'entier n est impair, faux sinon
1  x ← n
2  tant que x ≥ 2 faire
3  |   x ← x − 2
4  retourner x = 1
```

Algorithme 2 : *EstImpairParSoustractionsSuccessives* : calcul de parité à l'aide de la soustraction seule.

Sauf à vouloir en faire un exercice d'école, cet algorithme ne mérite pas qu'on l'accompagne d'une justification, disons que « le pseudo-code parle de lui-même »[21].

3.2.4 De l'usage du commentaire.

En revanche, l'algorithme de multiplication du paysan russe mérite quelques explications, car si l'arithmétique utilisée est élémentaire, nous ne considérons pas comme limpide que la mise en œuvre proposée conduise au calcul du produit $a \times b$.

On peut estimer que la version présentée par l'algorithme *MultiplicationRusse* (algorithme 3 page suivante), généreusement augmentée du commentaire[22] placé entre ses lignes 4 et 5, peut être une aide à qui s'intéresse au pourquoi du comment.

19. Voir la note n°9 page 7.
20. De la même façon, qu'il nous soit permis d'utiliser le verbe *satisfaire* sans lui adjoindre de complément.
21. La dernière instruction de l'algorithme est à comprendre comme suit : **si** $x = 1$ **alors** retourner la valeur *vrai* **sinon** retourner la valeur $faux$ **fin**. Nous revenons sur ce **retourner** dans la digression page 31.
22. Qu'on appelle *invariant*, nous y revenons à la section *Invariants* (section 3.2.6 page 20).

```
   Données : deux entiers positifs a et b
   Résultat : retourne la valeur du produit ab
 1 x ← a
 2 y ← b
 3 z ← 0
 4 tant que x ≠ 0
   ☞ à ce stade, on a z + xy = ab
 5  faire
 6 |  si x est impair alors
 7 |  |  z ← z + y
 8 |  x ← x ÷ 2
 9 |  y ← y × 2
10 retourner z
```

Algorithme 3 : *Multiplication Russe* : l'algorithme avec un *invariant* (voir la note 22 page précédente) en commentaire dans le pseudo-code.

Le fait que l'algorithme ne peut exécuter qu'un nombre fini d'itérations (pourvu que x soit bien entier positif, ce qui est requis par la rubrique *donnée* des spécifications) ne pose pas de problème [23], et donc l'algorithme *termine*.

En revanche, quant au commentaire, choix est laissé au lecteur de s'en convaincre ou de faire confiance au concepteur : avant chaque itération, au moment où le processeur évalue la condition de contrôle de la répétitive pour déterminer si son exécution est achevée ou non, ce commentaire affirme que la relation $z + xy = ab$ est satisfaite [24]. Ce qui permet de conclure que l'algorithme *satisfait* puisqu'il *termine* et

23. Comme déjà remarqué (voir note 10 page 7), tout entier positif est strictement supérieur à sa moitié entière par défaut, elle-même un entier positif ou nul, et on ne peut donc faire décroître strictement x qu'un nombre fini de fois jusqu'à la valeur 0 qui met une fin à l'exécution de la répétitive.

24. Pour s'en convaincre, notons x, y et z les valeurs des variables avant l'exécution d'une itération et x', y' et z' leurs valeurs respectives après l'exécution d'une itération, laquelle a nécessairement lieu sous l'hypothèse $x > 0$. Deux cas sont à envisager.

1 x est pair, c'est-à-dire que $x = 2k$ avec k entier positif. Alors :

$$\begin{aligned} z' &= z \\ x' &= k \\ y' &= 2y \\ z' + x'y' &= z + 2ky \\ &= z + xy \\ &= ab \end{aligned}$$

2 x est impair, c'est-à-dire que $x = 2k + 1$ avec k entier positif ou nul. Alors :

que x est nul à la fin de son exécution, entraînant que la valeur de z retournée vérifie $z = ab$.

3.2.5 De la nécessité d'une justification approfondie

Un secrétariat doit organiser les réservations d'une salle placée sous sa responsabilité. Les demandes affluent pour un même jour, sous la forme « demande de réservation de telle heure à telle heure ». Nous appellerons *requêtes* ces demandes.

Voici un tableau [25] (tableau 7) représentant un ensemble de requêtes, où les heures de début et les heures de fin sont des nombres entiers compris entre 0 et 24 (accordons au lecteur que les horaires, choisis pour les besoins de la cause, sont assez peu réalistes !) :

requêtes	a	b	c	d	e	f	g	h	u	v	w
heures de début	1	5	0	12	6	5	2	8	3	8	3
heures de fin	4	9	6	14	10	7	13	12	5	11	8

FIGURE 7: Tableau des requêtes de réservation de la salle.

Le problème consiste à sélectionner un ensemble de requêtes de sorte qu'aucune des réservations sélectionnées n'ait un horaire qui chevauche celui d'une autre réservation sélectionnée.

On peut trouver commode de représenter chaque requête par un segment, ce que propose le schéma de la figure 8 page suivante pour le tableau de la figure 7.

$$
\begin{aligned}
z' &= z + y \\
x' &= k \\
y' &= 2y \\
z' + x'y' &= z + y + 2ky \\
&= z + (2k+1)y \\
&= z + xy \\
&= ab
\end{aligned}
$$

et donc le commentaire est encore valable au moment où, une fois effectuée l'itération, le processeur évalue à nouveau la condition de contrôle de la répétitive (c.-à-d. l'expression booléenne $x \neq 0$).

25. Ce tableau est extrait (après un réordonnancement des colonnes) de CORMEN, T.H., LEISERSON, E.L., RIVEST, R.L. et al. *Introduction à l'Algorithmique*. 2de éd., Paris : Dunod, 2002, p. 362.

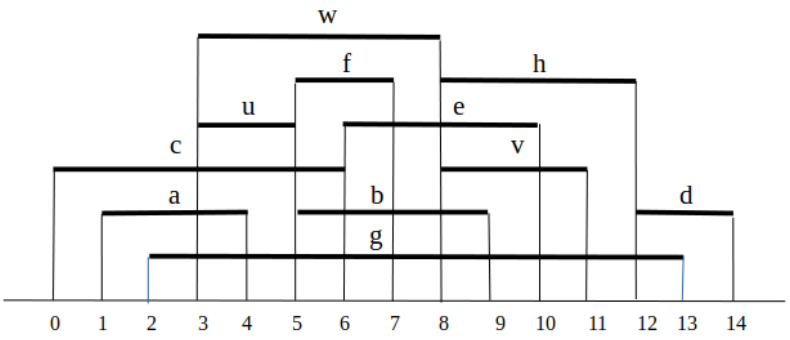

FIGURE 8: Représentation des requêtes du tableau de la figure 7 page précédente.

De façon plus formelle :

1. Une *requête* est un couple de réels (a, b) satisfaisant $a < b$; on dit que la *requête commence* en a et *termine* en b.
2. Deux *requêtes* (a, b) et (a', b') sont dites *incompatibles* lorsque l'intersection des intervalles $[a, b]$ et $[a', b']$ est non vide et non réduite à l'une de leurs extrémités [26], et sont dites *compatibles* sinon.
3. Soit R un ensemble de *requêtes* :
   - une *sélection valide* de R est un ensemble de *requêtes* de R deux à deux *compatibles* ;
   - une *sélection complète* de R est une *sélection valide* de R qu'on ne peut pas compléter dans R sans lui faire perdre sa validité ;
   - une *sélection optimale* de R est une *sélection valide* de R comprenant le plus grand nombre possible de requêtes parmi toutes les *sélections valides* de R.

Il est clair qu'une *sélection optimale* de R est une *sélection complète* de R.

Le problème consiste alors à calculer une sélection valide et, tant qu'à faire, une sélection complète [27], voire une sélection optimale de façon à satisfaire le plus grand nombre de demandeurs.

C'est ce que nous proposons avec l'algorithme *SélectionOptimale* [28].

26. En d'autres termes, les intervalles se chevauchent « vraiment ».
27. De fait, dans la « vraie » vie, une sélection qui ne serait qu'une sélection valide mais pas une sélection complète provoquerait sûrement l'incompréhension, voire l'invective.
28. Algorithme 4 page suivante.

On constate que la propriété « il existe une sélection optimale qui contient toutes les requêtes marquées bleu et aucune des requêtes marquées rouge », appelons la P, donnée en commentaire entre les lignes 1 et 2 de l'algorithme, risque d'apparaître comme un défi lancé au lecteur. D'autant qu'on ne peut s'en convaincre en exécutant « à la main » quelques exemples : s'il est clair que toute sélection retournée est bien valide, comment s'assurer [29] qu'elle est optimale ?

```
   Données : un ensemble de n requêtes (n ≥ 2)
   Résultat : retourne une sélection optimale extraite des requêtes
              données
1  tant que il reste des requêtes non marquées
   ☞ à ce stade : il existe une sélection optimale qui contient toutes les
     requêtes marquées bleu et aucune des requêtes marquées rouge
2  faire
3  |  choisir une requête r non marquée terminant au plus tôt ou
   |    commençant au plus tard
4  |  si la requêtes r est compatible avec chacune des requêtes
   |    marquées bleu alors
5  |  |  la marquer bleu
6  |  sinon
7  |  |  la marquer rouge
8  retourner l'ensemble des requêtes marquées bleu
```

Algorithme 4 : *SélectionOptimale* : calcul d'une *sélection optimale* à partir d'un ensemble de *requêtes* donné.

Comme chaque itération marque une requête non marquée et que les requêtes ne sont jamais démarquées, il est clair que l'algorithme *termine*.

Pour montrer qu'il *satisfait*, on va montrer que la propriété P est un commentaire valide au début et à la fin de l'exécution de chaque itération par le processeur. On le montre en deux temps.

1. On montre que si P est vraie avant l'exécution d'une itération, donc sous l'hypothèse qu'il reste des requêtes non marquées, alors elle reste vraie après l'exécution de cette itération.

   Sans perte de généralité par raison de symétrie, nous le vérifions seulement dans le cas ou la requête r choisie *termine* au plus tôt parmi les requêtes non marquées, et appelons alors OPT une

29. Au contraire du commentaire de l'algorithme *MultiplicationRusse* (algorithme 3 page 15) qu'on peut vérifier directement « à la main » à chaque itération.

sélection optimale contenant toutes les requêtes marquées bleu et aucune des requêtes marquées rouge dont P affirme l'existence.

Le résultat est acquis si r est incompatible avec l'une au moins des requêtes marquées bleu : OPT ne contient sûrement pas r que le processeur marque rouge, et OPT contient donc toujours toutes les requêtes marquées bleu et aucune des requêtes marquées rouge.

Supposons donc que r soit compatible avec chacune des requêtes marquées bleu, qui sont par hypothèse des requêtes de OPT. Comme r est marquée bleu par le processeur, le résultat est encore acquis si r appartient OPT.

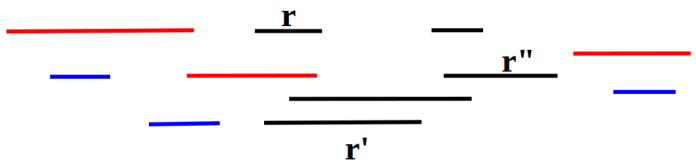

FIGURE 9: Une étape d'exécution de l'algorithme de calcul d'une sélection optimale. La sélection optimale OPT est constituée des requêtes marquées bleu et des requêtes non marquées r' et r''. Elle contient toutes les requêtes marquées bleu et aucune des requêtes marquées rouge. Sa requête non marquée terminant au plus tôt est la requête r'. La requête r est la requête non marquée terminant au plus tôt. Elle est incompatible avec la requête r'. La requête non marquée r'' de OPT n'est pas incompatible avec la requête r puisqu'elle ne peut pas l'être avec la requête r' qui, elle aussi dans OPT, termine avant elle, fût-ce au sens large.

Supposons donc que r ne soit pas une requête de OPT (voir la figure 9). Alors OPT contient d'autres requêtes que les requêtes marquées bleu (au moins une), car sinon en marquant bleu la requête r on fabriquerait une sélection valide qui contredirait le fait que OPT est elle-même une sélection optimale. Soit alors r' une des requêtes de OPT non marquées, et terminant au plus tôt parmi celles-ci. De par le choix de r, r' ne peut terminer plus tôt que r. Il s'ensuit qu'aucune requête non marquée de OPT autre que r' ne peut être incompatible avec r : elle le serait alors aussi avec r', ce qui ne se peut puisque OPT est valide. Alors en échangeant r' avec r dans OPT, on obtient une sélection optimale satisfaisant encore P après que r a été marquée bleu par

le processeur [30] .

2. P est vraie avant l'exécution de la première itération.

   L'initialisation la rend trivialement vraie puisqu'aucune requête n'est marquée [31].

Il s'ensuit que la propriété P, vraie juste avant l'exécution de la première itération [32] et que l'exécution d'aucune itération ne peut faire passer de *vrai* à *faux*, est bien valide au début et à la fin de l'exécution de chaque itération. Comme l'algorithme *termine*, à la fin de son exécution P est encore vraie. Le fait que toutes les requêtes soient alors marquées, soit bleu, soit rouge, permet de conclure que les requêtes marquées bleu constituent nécessairement la sélection optimale dont la propriété P affirme l'existence, autrement dit, l'algorithme *satisfait*.

3.2.6 Invariants

Nous généralisons la démarche que nous avons adoptée pour justifier les algorithmes *MultiplicationRusse* et *SélectionOptimale* [33].

30. Cette requête est bien optimale puisque valide par choix de r et optimale car comptant autant de requêtes que OPT.

31. Ce « trivialement », utilisé ici au sens où il n'y a rien à démontrer, mérite quelques commentaires, sinon à constater qu'il existe au moins une sélection optimale puisque les spécifications imposent que la donnée comporte au moins deux requêtes.

De fait, à ce stade de l'initialisation où aucune requête n'est marquée, il est clair qu'aucune sélection, optimale ou non, ne contient de requête marquée rouge.

En revanche, il se peut que n'aille pas de soi le fait que toutes les sélections contiennent bien toutes les requêtes marquées bleu... puisqu'il n'y a pas de requête marquée bleu.

On peut s'en convaincre plus facilement en reformulant l'assertion : aucune des requêtes marquées bleu ne manque à aucune sélection puisqu'il n'existe pas de requête marquée bleu.

Osons un parallèle avec une situation « de la vraie vie » : au cours d'un voyage scolaire, un autocar doit franchir une frontière, obligation étant faite aux élèves mineurs de présenter une autorisation des parents. Que faire d'un car où tous les élèves sont majeurs ? satisfait-il la condition « les élèves mineurs sont tous en situation de pouvoir présenter une autorisation des parents » ?

Quoi qu'il en soit, une façon de contourner ici l'éventuelle difficulté consiste à vérifier que la propriété P est bien satisfaite après l'exécution de la première itération qui marque nécessairement bleu la première requête sélectionnée.

32. En tout cas juste après l'exécution de la première itération que les spécifications rendent inéluctable. Ceci au cas où on souhaite s'économiser le débat sur la validation d'une propriété pour cause de « vacuité » (voir la note 31), mais il faut le payer au prix d'un raisonnement dont l'analogie avec celui du *maintien* de la propriété par n'importe quelle autre itération fera sans doute reconsidérer la démarche.

33. Respectivement algorithme 3 page 15 et algoritheme 4 page 18.

Invariant pour une répétitive. Soit $\mathcal{R}$ l'instruction répétitive **tant que** $\mathcal{C}$ **faire** $\mathcal{I}$ et soit P une propriété.

On dit que :

> la propriété P est un *invariant* pour la répétitive $\mathcal{R}$ lorsque si on exécute le corps $\mathcal{I}$ de la répétitive sous l'hypothèse que $\mathcal{C}$ et P sont satisfaites, alors, à la fin de l'exécution de l'itération $\mathcal{I}$, la propriété P est toujours satisfaite [34].

Illustration : les propriétés « $z+xy = ab$ » et « il existe une sélection optimale qui contient toutes les requêtes marquées bleu et aucune des requêtes marquées rouge » ont respectivement été montrées être des invariants pour les répétitives des algorithmes *MultiplicationRusse* et *SélectionOptimale*.

N.B. Ce que nous entendons par « invariant » [35] mérite d'être précisé au sens où dire d'une propriété P qu'elle est un *invariant* pour une répétitive $\mathcal{R}$ ne donne aucune indication sur le fait qu'elle puisse être satisfaite ou non au cours d'une exécution de ladite répétitive.

Nous en donnons une illustration à partir de l'algorithme *EstImpairParSoustractionsSuccessives* [36] dont la répétitive admet comme invariants les deux propriétés suivantes qui sont la négation l'une de l'autre :

— P : x est pair ;

— Q : x est impair.

Dire que P et Q sont des invariants pour la répétitive considérée signifie que si n'importe laquelle des deux est satisfaite et qu'on exécute une itération (c'est-à-dire $x \leftarrow x - 2$ sous l'hypothèse $x > 1$), elle reste satisfaite après cette exécution. Maintenant, qu'au cours d'une exécution de l'algorithme, que l'une soit satisfaite, et donc l'autre pas, dépend évidemment de la valeur initiale de x.

Schéma de preuve par invariants Soit $\mathcal{R}$ l'instruction répétitive **tant que** $\mathcal{C}$ **faire** $\mathcal{I}$ et soit P une propriété.

En montrant que :

— la répétitive $\mathcal{R}$ *termine* ;

— chaque itération *maintient* P, c'est-à-dire que la propriété P est un invariant pour la répétitive $\mathcal{R}$;

34. Rappelons qu'il y ait lieu d'exécuter une itération si et seulement si la condition $\mathcal{C}$ est satisfaite.

35. À ne pas confondre avec un invariant en Physique – qui ne varie pas – ni avec un invariant de structure ensembliste – qui se transmet par isomorphisme.

36. Algorithme 2 page 14.

— l'initialisation *établit* la propriété P, c'est-à-dire que la propriété P est satisfaite juste avant l'exécution de la répétitive $\mathcal{R}$;

on montre qu'après l'exécution de l'instruction répétitive $\mathcal{R}$, la propriété P est satisfaite en même temps que la condition $\mathcal{C}$ ne l'est pas.

C'est la démarche que nous avons suivie pour justifier les deux algorithmes évoqués plus haut.

1. S'agissant de la multiplication du paysan russe :
   — la propriété $x \geq 0$ est *établie* par l'initialisation et *maintenue* par chaque itération ;
   — donc la répétitive *termine* puisque x décroît strictement à chaque itération (propriété des entiers positifs : $0 \leq x \div 2 < x$) ;
   — la propriété $z + xy = ab$ est *maintenue* par chaque itération [37] ;
   — l'initialisation *établit* clairement cette propriété.

   On en déduit qu'à la fin de l'exécution de l'algorithme on a à la fois $z + xy = ab$ et $x = 0$, soit encore $z = ab$.

37. Voir la note 24 page 15.

2. S'agissant de la sélection optimale :
   — la répétitive termine puisque chaque itération marque une requête non marquée sans jamais en démarquer aucune ;
   — la propriété « il existe une sélection optimale qui contient toutes les requêtes marquées bleu et aucune des requêtes marquées rouge » est *maintenue* par chaque itération [38] ;
   — l'initialisation *établit* cette propriété « par vacuité » (ou, si on préfère, c'est la première itération qui l'*établit*) [39].

   On en déduit qu'à la fin de l'exécution de l'algorithme on a à la fois « il existe une sélection optimale qui contient toutes les requêtes marquées bleu et aucune des requêtes marquées rouge » et « toutes les requêtes sont marquées », d'où on déduit que « l'ensemble des requêtes marquées bleu constitue une sélection optimale ».

Digressions.

1. L'ordre proposé ici pour raisonner n'a pas à être respecté (*terminaison* de la répétitive, *établissement* et *maintien* de l'invariant). Toutefois, il est motivé par le souci de mettre en évidence que :
   — si la répétitive $\mathcal{R}$ ne *termine* pas, le reste est sans objet ;
   — le cœur de la démonstration consiste à montrer que chaque itération *maintient* l'invariant P ; cette preuve, qui peut parfois être « difficile », peut parfois aussi se dérober et conduire à modifier l'invariant initialement choisi (et souvent à l'enrichir), soit encore, dans le pire des cas, à constater que l'algorithme lui-même mérite qu'on le remette sur le métier, voire qu'on l'abandonne ;
   — *établir* l'invariant P au moyen d'une initialisation adaptée relève le plus souvent d'une démarche simplement technique… qu'il est judicieux de ne mettre en place qu'une fois l'invariant P « stabilisé ».
2. On aura pu reconnaître ici une interprétation du principe de récurrence appliqué aux instructions répétitives en définissant « $P(n)$: la propriété P est vraie après l'exécution de n itérations » :
   — initialisation : $P(0)$ est vraie ;
   — maintien : $P(n) \Rightarrow P(n+1)$.

38. Voir item 1 page 18.
39. Voir notamment la note 31 page 20.

Nous en donnons une formulation prenant en compte le fait que le maintien ne s'entend que lorsque la condition de contrôle de la répétitive est satisfaite, en utilisant les symboles de la logique $\neg$, $\vee$ et $\wedge$ pour représenter respectivement la négation, la disjonction et la conjonction.

Soit $\mathcal{I}$ une instruction et P et Q deux propriétés. La notation $\{P\}\ \mathcal{I}\ \{Q\}$ s'interprète comme suit :

> si on exécute l'instruction $\mathcal{I}$ sous l'hypothèse que la propriété P est satisfaite, alors, sous réserve que l'instruction $\mathcal{I}$ *termine*, la propriété Q est satisfaite à la fin de cette exécution.

Ainsi, un *invariant* pour l'instruction répétitive **tant que** C **faire** $\mathcal{I}$, où C en est la condition de contrôle et $\mathcal{I}$ le corps, est une propriété P telle que $\{P \wedge C\}\ \mathcal{I}\ \{P\}$.

Schéma de preuve par invariant

Soit $\mathcal{R}$ l'instruction répétitive **tant que** C **faire** $\mathcal{I}$:

— si on a montré que $\{P \wedge C\}\ \mathcal{I}\ \{P\}$ et si on a montré que la répétitive termine, alors on a montré que $\{P\}\ \mathcal{R}\ \{P \wedge \neg C\}$;

— si on a montré de plus que la propriété P est établie avant l'exécution de $\mathcal{R}$, alors la propriété $\{P \wedge \neg C\}$ est satisfaite après l'exécution de $\mathcal{R}$.

N.B. Pour raisonner strictement, la preuve ne réclame que de s'assurer que l'invariant prendra la valeur *vrai* lors de l'exécution d'au moins une des itérations effectuées par le processeur, auquel cas l'invariant restera ensuite satisfait jusqu'à l'arrêt de cette dernière [40]. Cependant, dans une grande majorité des cas, c'est bien à l'initialisation qu'incombe la responsabilité d'*établir* l'invariant.

3. Pour montrer que la répétitive $\mathcal{R}$ termine, on choisit une expression e dont la valeur est un entier naturel, parfois appelée *variant*, dont la valeur dépend de l'évolution des variables de l'algorithme au cours de son exécution, et on montre que sa valeur varie de façon strictement décroissante, ce qui ne se peut donc qu'en un nombre fini d'étapes [41].

40. On peut noter à cet égard que nous avons proposé, comme alternative à un raisonnement logique faisant intervenir l'ensemble vide lors de la preuve de l'algorithme de la sélection optimale (algorithme 4 page 18), d'observer que la répétitive exécute nécessairement une première itération qui, elle, installait bien l'invariant.

41. Si on a trouvé une expression entière e dont la valeur croît strictement à chaque

S'agissant de l'algorithme *MultiplicationRusse*[42] l'expression e choisie est réduite à la variable x et en toute rigueur, affirmer que la répétitive *termine* passe par :

(a) La propriété « x est un entier positif ou nul » est *établie* par l'initialisation (compte tenu de la rubrique *donnée* des spécifications).

(b) Elle est maintenue par chaque itération (c'est un invariant pour la répétitive).

(c) La suite des valeurs de x est une suite finie strictement décroissante dont le dernier terme est 0.

4. Exercice corrigé :
   On a exécuté l'algorithme *EstImpairParSoustractionsSuccessives*[43] pour un certain entier n. La réponse retournée par l'algorithme est la valeur $vrai$. Que peut-on dire de la parité de n ?
   Il est demandé de justifier sa réponse au moyen de l'un des deux invariants :

   — P : x est pair ;

   — Q : x est impair.

   La réponse est évidemment « alors n est impair ».
   Mais d'expérience d'enseignants et de formateurs, l'exercice est apparemment piégeux : dans la (très) grande majorité des cas, la justification obtenue est basée sur l'invariant $Q\ldots$ qui permet seulement de prouver que si n est impair, l'algorithme retourne bien la valeur $vrai$.
   Or c'est un raisonnement par l'absurde basé sur l'invariant P qui donne la solution :

   > si n est pair, comme P est un invariant pour la répétitive, celle-ci ne peut terminer qu'avec une valeur paire de x et l'algorithme retourne la valeur $faux$; or l'algorithme a retourné la valeur $vrai$, donc n ne peut pas être pair.

---

itération tout en restant majorée par un entier fixé N, alors $e' = N - e$ peut être utilisée comme variant.

On généralise le principe en remplaçant l'ensemble $\mathbb{N}$ par n'importe quel ensemble ordonné satisfaisant la *condition de la chaîne descendante* (`https://fr.wikipedia.org/wiki/Conditions_de_chaîne`).

En pratique, nous nous en tiendrons le plus souvent à $\mathbb{N}$.

42. Algorithme 3 page 15.
43. Algorithme 2 page 14.

Accordons que la présence plus ou moins consciente de l'invariant « R : x et n ont la même parité » peut avoir troublé le raisonnement… mais il reste que l'invariance de Q seule n'établit pas celle de R, et que l'invariance de P seule suffit à justifier la réponse apportée.

3.2.7 Des invariants pour généraliser et pour spécialiser

La justification d'un algorithme, en mettant en évidence les raisons pour lesquelles son exécution effectue bien ce qu'on en attend, peut ouvrir des perspectives de généralisation comme de spécialisation :

— peut-on relaxer certain détails de la mise en œuvre tout en conservant la justification ?

— est-ce que d'autres instructions relèveraient de la même justification ?

Nous proposons d'illustrer notre propos autour de l'algorithme *MultiplicationRusse*[44]. Sa justification tient à ce qu'il *termine* et que la propriété $z + xy = ab$ est :

— *établie* par l'initialisation ;

— *maintenue* par chaque itération.

Ce qui établit directement que l'algorithme *SchémaGénéralDeMultiplication* (algorithme 5) lui aussi tout à la fois *termine* et *satisfait*.

Données : deux entiers positifs ou nuls a et b
Résultat : retourne la valeur du produit ab
1 initialiser x, y et z de sorte que $z + xy = ab$
2 **tant que** $x \neq 0$
☞ $z + xy = ab$
3 **faire**
4 | diminuer strictement la valeur entière de x et modifier celles de y et de z en conséquence de façon à conserver la propriété $z + xy = ab$
5 **retourner** z

Algorithme 5 : *SchémaGénéralDeMultiplication* : une méthode générale de multiplication de deux entiers positifs ou nuls.

Dans la mesure où l'algorithme *MultiplicationRusse* met en œuvre de façon spécifique le corps de l'algorithme *SchémaGénéralDeMulti-*

44. Algorithme 3 page 15.

plication en en restreignant l'ensemble des données valides [45], on peut voir ce dernier comme une *généralisation* du premier et, inversement, le premier comme une *spécialisation* du second.

Mais on peut penser à d'autres *spécialisations* du *SchémaGénéralDeMultiplication*, comme par exemple l'algorithme *MultiplicationParamétrée* (algorithme 6).

Données : deux entiers positifs ou nuls a et b et un entier k supérieur à 1
Résultat : retourne la valeur du produit ab
☞ a `mod` b représente le reste de la division de a par b
1 $x \leftarrow a$
2 $y \leftarrow b$
3 $z \leftarrow 0$
4 **tant que** $x \neq 0$
☞ $z + xy = ab$
5 **faire**
6 $\quad z \leftarrow z + y \times (x \bmod k)$
7 $\quad x \leftarrow x \div k$
8 $\quad y \leftarrow y \times k$
9 **retourner** z

Algorithme 6 : *MultiplicationParamétrée* : multiplication paramétrée de base k.

Que l'algorithme *termine* tient au fait que la rubrique *donnée* impose que k vaille au moins [46] 2. Que chaque itération *maintienne* P

45. Le lecteur a pu remarquer que la rubrique *Données* des spécifications de l'algorithme*SchémaGénéralDeMultiplication* admet les valeurs nulles pour a et b, ce qui, d'une certaine façon, conforte cet algorithme comme généralisation de l'algorithme *MultiplicationRusse*.

Deux remarques à ce propos qui se complètent sans s'opposer :

— les premières versions de la multiplication ne mentionnaient pas la valeur 0 car nul n'a besoin d'appliquer une méthode sophistiquée pour évaluer un produit dont un facteur est nul (à compter même que le 0 ait déjà acquis ses lettres de noblesse) ;

— signaler que la méthode est encore valable en cas de facteur(s) nul(s) peut être une coquetterie de l'auteur, mais peut aussi être utile à garantir que l'algorithme peut être mis à contribution dans une chaîne de calculs répétés sans que l'utilisateur n'ait à se prémunir contre les cas particulier d'apparition intempestive d'un 0.

Bref, chacun fait comme il préfère.

46. Il est clair que, pour des raisons différentes, les valeurs $k = 0$ et $k = 1$ ne peuvent être retenues : pour cause d'opération non définie dans un cas (division par 0) et de non *terminaison* de l'algorithme dans l'autre (l'exécution d'une itération ne modifie pas la valeur de x puisque $x \div 1 = x$).

résulte de ce que, en posant $x = qk + r$, on a, en notant respectivement x', y' et z' les valeurs des variables x, y et z après leur modification par l'exécution d'un itération :

$$z + xy = z + (qk + r)y = (z + ry) + q(ky) = z' + x'y'$$

Il est clair que plus k est grand, plus la décroissance de x est rapide. En revanche, il est requis du processeur destiné à exécuter cet algorithme qu'il sache calculer :

— le reste de la division d'un entier positif ou nul quelconque par k (ligne 6) ;

— le quotient entier de la division d'un entier positif ou nul quelconque par k (ligne 7) ;

— le produit d'un entier positif ou nul quelconque par n'importe quel entier positif ou nul inférieur ou égal à k. (lignes 6 et 8).

Vu les savoir-faire requis des processeurs éligibles à l'exécution de cet algorithme, deux valeurs particulières de k attirent « naturellement » notre attention :

1. $k = 2$.

   Le calcul de $x \bmod 2$ (en ligne 6) a pour résultat 0 si x est pair et 1 si x est impair. On reconnaît là une variante d'écriture de l'algorithme *MultiplicationRusse*.

2. $k = 10$.

   Papier-crayon, le choix est judicieux car la division par 10, reste ou quotient, est grandement facilitée [47]. Pour ce qui est du calcul du produit de n'importe quel entier par 10, toujours la même facilité, en revanche, pour le produit $y \times (x \bmod 10)$ (dans l'affectation de la ligne 6 de l'algorithme adapté à $k = 10$)... il y faut une méthode et pourquoi pas un peu de par cœur, genre « les tables de multiplication » ?

   De fait, on peut aisément se convaincre que, à la disposition près des calculs, cet algorithme est très exactement celui de la *multiplication à plusieurs chiffres* enseignée dans nos écoles [48].

3.3 Composition des algorithmes

La *composition d'algorithme* est la faculté pour un algorithme A de pouvoir utiliser lors de son exécution, le résultat de l'exécution d'un

47. Du moins pour qui a suivi, avec succès et sous nos latitudes, un parcours primaire conventionnel.

48. On peut remarquer que, du fait de la division entière de x par 10 à chaque itération, $x \bmod 10$ décrit, de droite à gauche, la suite des chiffres de l'écriture de x en base 10.

algorithme B [49].

3.3.1 Appels d'algorithmes

La composition de deux algorithmes A et B s'exprime sous forme d'un *appel* de l'algorithme B par l'algorithme A consistant en la présence dans le corps de l'algorithme A du nom de l'algorithme B suivi d'une liste des valeurs demandées par la rubrique *donnée* des spécifications de ce dernier [50].

Lorsque l'algorithme A appelle l'algorithme B, son exécution s'interrompt pour lancer celle de l'algorithme B, lequel peut être utilisé :

1. pour modifier l'environnement de l'exécution suspendue de l'algorithme A [51] ;
2. ou pour fournir le résultat d'une évaluation [52].

Dans le premier cas, une fois l'exécution de l'algorithme B terminée, celle de l'algorithme A reprend son cours.

Dans le second cas, le corps de l'algorithme B comporte une *instruction de retour* de la forme [53] :

retourner E

où :

— **retourner** est un mot-clé ;

— E est une expression.

et la valeur de l'expression E calculée par l'exécution de l'algorithme B est substituée à l'appel lancé par l'algorithme A dont l'exécution reprend alors son cours.

Pour illustration, on constate bien que :

— l'algorithme *EstImpairParSoustractionsSuccessives* [54], de par sa définition, *retourne* une valeur booléenne, c'est-à-dire soit *vrai* soit $faux$ (voir la digression qui suit) ;

49. Par analogie avec la composition des fonctions où, par exemple, la notation $\sqrt{x^2 + y^2}$ fait appel à la composition des fonction *racine carrée* et *somme des carrés de deux nombres*

50. La liste des valeurs demandées par la rubrique *donnée* des spécifications d'un algorithme s'appelle la liste de ses *paramètres formels*, celles des valeurs fournies lors de l'appel de cet algorithme s'appelle la liste des *paramètres effectifs* de l'appel.

51. Ce que le langage Pascal appelle une *procédure*.

52. Ce que le langage Pascal appelle une *fonction*.

53. Un exemple en est fourni à la ligne 9 de l'algorithme *Multiplication Russe*, algorithme 1 page 3.

54. Algorithme 2 page 14, qui teste la parité de sa donnée.

Données : un entier positif ou nul n
Résultat : retourne *vrai* si l'entier n est impair, $faux$ sinon

```
1 x ← n
2 tant que x ≥ 2 faire
3 |   x ← x − 2
4 retourner x = 1
```

Algorithme 7 : *EstImpairParSoustractionsSuccessives* : calcul de parité à l'aide de la soustraction seule.

Données : deux entiers positifs a et b
Résultat : retourne la valeur du produit ab
☞ fait appel à l'algorithme *EstImpairParSoustractionsSuccessives*

```
1 x ← a
2 y ← b
3 z ← 0
4 tant que x ≠ 0 faire
5 |   si EstImpairParSoustractionsSuccessives(x) alors
6 |   |   z ← z + y
7 |   x ← x ÷ 2
8 |   y ← y × 2
9 retourner z
```

Algorithme 8 : *Multiplication Russe Bis* : multiplication du paysan russe avec appel de l'algorithme *EstImpairParSoustractionsSuccessives*.

— c'est bien une valeur booléenne que l'algorithme *Multiplication Russe Bis*[55] a besoin d'évaluer afin de pouvoir exécuter l'instruction conditionnelle commençant en sa ligne 5.

Digression : alternatives et expressions booléennes, notion d'expression alternative. La dernière instruction de l'algorithme *EstImpairParSoustractionsSuccessives*[56] est :

retourner $x = 1$

Or, notamment en phase d'apprentissage, d'aucuns écrivent plus spontanément l'instruction alternative suivante, appelons la *alternative-de-retour* :

si $x = 1$
  alors retourner $vrai$
  sinon retourner $faux$

Élargissons le débat.

Il est courant, lorsqu'on veut affecter une variable y de la valeur absolue d'une variable x, d'écrire :

si $x \geq 0$
  alors $y \leftarrow x$
  sinon $y \leftarrow -x$

une instruction parfaitement définie dans notre cadre : il s'agit d'une instruction alternative dont les deux branches sont des affectations, c'est-à-dire des instructions élémentaires.

Mais on peut aussi définir la notion d'*expression alternative*[57] :

si C
  alors E_1
  sinon E_2

où :

— C est la condition de contrôle de l'expression alternative, c'est-à-dire une expression booléenne ;

— E_1 et E_2 sont des expressions.

L'évaluation de l'expression se ramène alors à celle de E_1 lorsque la condition de contrôle C s'évalue à $vrai$, et à celle de E_2 lorsque la condition de contrôle C s'évalue à $faux$, ce qui permet de réécrire le calcul de la valeur absolue sous la forme :

55. Algorithme 9 page précédente.

56. Algorithme 2 page 14.

57. Dite plus généralement *expression conditionnelle* (il en existe d'autres formes). Cette notion introduite en son temps par le langage Algol60, a été reprise entre autres par le langage *C* (dans une syntaxe passablement absconse) et par les langages de la famille ML (en particulier OCalm).

$y \leftarrow$

si ≥ 0 **alors** x
sinon $-x$

où il apparaît immédiatement que l'instruction est en première intention une affectation de la variable y, et seulement ensuite que la valeur à lui affecter est soumise à condition [58].

Pour revenir à l'instruction que nous avons appelée *alternative-de-retour*, celle-ci peut dans ces conditions s'écrire :

retourner

si $x = 1$ **alors** $vrai$
sinon $faux$

qui demande de retourner la valeur $vrai$ si x vaut 1 et de retourner la valeur $faux$ si x ne vaut pas 1... ce qui est précisément la valeur de l'expression booléenne $x = 1$.

Considérant que « les expressions booléennes sont des expressions comme les autres », nous avons une préférence pour :

retourner $x = 1$

qui a bien pour sémantique :

1) évaluer l'expression $x = 1$ (qui est une expression booléenne) ;
2) retourner la valeur calculée (une valeur qui est donc soit $vrai$ soit $faux$).

3.3.2 Niveaux d'abstraction

L'algorithme *EstImpairParDivisionPar2* (algorithme 9) propose une autre façon de tester la parité d'un entier positif ou nul.

Données : un entier positif ou nul n
Résultat : retourne $vrai$ si l'entier n est impair, $faux$ sinon
1 **retourner** $(x \div 2) \times 2 = x$

Algorithme 9 : *EstImpairParDivisionPar2* : calcul de parité à l'aide de la division entière par 2.

On constate que, de façon équivalente en ce qui concerne le résultat calculé, l'instruction conditionnelle commençant en ligne 5 de l'algorithme *Multiplication Russe Bis* [59] :

si *EstImpairParSoustractionsSuccessives*(x) **alors** $z \leftarrow z + y$

peut être remplacée par l'instruction conditionnelle :

58. En quelque sorte, on a « mis en facteur » le symbole d'affectation $\leftarrow$.
59. Algorithme 9 page 30.

Si on peut considérer comme brillante l'idée qui fonde la méthode de la multiplication sans tables [60] , il apparaît que la façon de déterminer si un entier est impair n'y est pour rien. Plus précisément, on peut considérer le calcul de la parité comme un détail technique au regard de l'élaboration de la méthode de multiplication, quitte à lui apporter ailleurs et à un autre moment toute la considération qu'il mérite.

Ce que permet précisément la composition des algorithmes au moyen des *appels d'algorithmes*, un outil autorisant la hiérarchisation de différents niveaux d'abstraction lors de la phase de conception : il est possible, voire recommandé, d'élaborer un algorithme *Multiplication Russe* en présumant l'existence d'un algorithme *Impair* à concevoir par ailleurs et pour lui-même, ayant comme spécifications les spécifications commune aux deux algorithmes *EstImpairParSoustractionsSuccessives* et *EstImpairParDivisionPar2*.

L'algorithme *MultiplicationDuPaysanRusseIncluantLaRechercheDeParité* (algorithme 10 page suivante) en donne a contrario une illustration, en intégrant directement dans son code celui de la détermination de la parité de x, au prix de l'utilisation de « boucles imbriquées » [61].

Quand bien même on peut penser formatrice la mobilisation nécessaire des facultés d'abstraction que nécessite l'usage de ces boucles imbriquées, nous y voyons le plus souvent un manque de hiérarchi-

60. Dont une version *papier crayon*, moins formelle et toujours sans tables, peut consister à construire les trois colonnes :

50	89	
25	178	178
12	356	
6	712	
3	1424	1424
1	2848	2848
		4450

pour le calcul de 50 fois 89 en suivant les directives :

1. Écrire une colonne d'entiers sous le nombre 50 dont chaque nombre est la moitié entière par défaut du nombre situé immédiatement au-dessus de lui, et en s'arrêtant au nombre 1.
2. Écrire à la droite de la colonne précédente, une colonne commençant au nombre 89, de longueur égale à celle de la précédente colonne, et où chaque nombre est le double du nombre situé immédiatement au-dessus de lui.
3. Écrire, à la droite de la colonne précédente, une troisième colonne où, pour chaque ligne, si le contenu de la première colonne est impair, on recopie le contenu de la seconde colonne, sinon on laisse en blanc.
4. Additionner les nombres de la troisième colonne.

61. C'est-à-dire d'une répétitive dont le corps contient lui-même une répétitive

sation des concepts à mettre en œuvre lors de la conception d'une solution algorithmique à un problème donné.

À cet égard, nous maintenons que le calcul de parité prend un caractère « technique », même si digne d'intérêt, en regard de la remarquable stratégie de multiplication proposée par le « paysan russe ». Ce que la composition des algorithmes permet de mettre clairement en évidence.

```
   Données : deux entiers positifs a et b
   Résultat : retourne la valeur du produit ab
 1 x ← a
 2 y ← b
 3 z ← 0
 4 tant que x ≠ 0 faire
 5 |   t ← x
 6 |   tant que t > 1 faire
 7 |   |   t ← t − 2
 8 |   si t=1 alors
 9 |   |   z ← z + y
10 |   x ← x ÷ 2
11 |   y ← y × 2
12 retourner z
```

Algorithme 10 : *MultRusseAvecRechercheDeParité* : multiplication du paysan russe reformulée avec calcul direct de parité.

4 Du non déterminisme

L'algorithme *SélectionOptimale*[62] utilise une *instruction de choix* (ligne 3). Ce degré de liberté laissé au processeur se traduit par le fait que plusieurs *traces d'exécution* différentes sont possibles et, par définition du *non déterminisme*, elles sont toutes considérées comme acceptables au sens où toute exécution de l'algorithme est garantie respecter les spécifications[63] : quels que soient les choix effectués par le processeur, il calcule bien une sélection optimale de tout ensemble de n requêtes ($n \geq 2$).

On peut noter à cet égard qu'à partir des données du tableau de la figure 7 page 16, cet algorithme peut, suivant les choix effectués en cours d'exécution, retourner l'une des quatre *sélections* suivantes :

62. Algorithme 4 page 18.
63. Voir la section *De la nécessité d'une justification approfondie*, section 3.2.5 page 16.

1. $\{a, d, f, h\}$
2. $\{a, d, f, v\}$
3. $\{d, f, h, u\}$
4. $\{d, f, u, v\}$

à l'exclusion de toute autre.

Les spécifications de l'algorithme ne revendiquent pas autre chose que l'*optimalité* des *sélections* qu'il calcule. En particulier, elles ne revendiquent pas que ces quatre *sélections* soient les seules à être des *sélections optimales*.

De fait ce sont bien les seules pour notre exemple [64], mais nous laissons au lecteur le soin de vérifier que ce n'est pas toujours le cas, par exemple en considérant les *traces* de l'algorithme *SélectionOptimale* s'exécutant avec la donnée illustrée dans le tableau de la figure 10.

requêtes	a	b	c	d
heures de début	1	2	5	6
heures de fin	3	4	7	8

FIGURE 10: On peut facilement vérifier qu'avec les données de ce tableau, les choix offerts à l'exécution de l'algorithme *SélectionOptimale* conduisent à l'une des trois sélections optimales $\{a, c\}$, $\{a, d\}$ et $\{b, d\}$, à l'exclusion de la sélection $\{b, c\}$ pourtant également optimale. On peut noter qu'un résultat similaire est obtenu avec les deux seules requêtes $a' = (1, 4)$ et $b' = (2, 3)$.

À l'inverse, on peut concevoir des algorithmes non déterministes destinés à calculer un même et unique résultat quels que soient les différents choix effectués lors d'exécutions répétées [65].

Ce qui est assurément le cas de l'algorithme *TriParSuppressions-DInversions* (algorithme 11 page suivante), sans doute l'un des algorithmes de tri les moins sophistiqués et qui illustre bien notre propos pour autant qu'on soit assuré que l'algorithme *termine*, ce qui est plus ou moins intuitif [66].

64. Confiance, ou pas confiance ? Si pas confiance, bon courage... mais c'est quand même faisable à la main. Ou alors programmez... bon courage aussi.

65. Des algorithmes non déterministe qu'on pourrait dire *globalement* déterministes.

66. Si (u_a, u_b) est une inversion, la transposition de u_a avec u_b la détruit mais, en général, elle crée du même coup d'autres inversions aussi en même temps qu'elle en détruit. Une soigneuse étude de cas permet de monter qu'au total, elle en détruit toujours au moins une de plus qu'elle n'en crée, d'où le résultat. Mais montrons sur un exemple une autre façon de s'en persuader :

— soit à trier la liste 8 12 4 7 2 ;

— transposer 12 et 7 conduit à la liste 8 7 4 12 2 ;

Données : une suite d'entiers $S = (u_0, u_1, u_2, \cdots, u_n) = \left(u_i\right)_{0 \leq i \leq n}$
Résultat : permute les éléments de la suite S de sorte qu'elle soit ordonnée croissante, c'est-à-dire que quels que soient deux éléments de la suite, celui de plus petit indice est inférieur ou égal à l'autre

☞ appelons *inversion* d'une suite d'entiers $\left(u_i\right)_{0 \leq i \leq n}$ tout couple de termes (u_i, u_j) tel que $0 \leq i < j \leq n$ avec $u_i > u_j$, c'est-à-dire deux termes distincts tels que le plus grand des deux ait le plus petit indice

```
1 tant que la suite S présente une inversion faire
2 |   choisir une inversion (u_i, u_j)
3 |   transposer dans la suite S les éléments de rangs respectifs i et j
```

Algorithme 11 : *TriParSuppressionsDInversions*

5 Conclusion

Nous avons présenté l'idée du couple « nécessaire » *algorithme/processeur* et proposé une définition pour les *algorithmes itératifs* tout en restant largement permissifs quant aux types de processeurs intervenant, notamment en ce qui concerne les *actions de base* que chacun d'entre eux doit être à même de pouvoir exécuter selon les algorithmes considérés [67].

La nécessité de la *justification* des algorithmes va de soi, en liaison avec leurs *spécifications*, et nous avons, à ce propos, introduit la notion d'*invariant* permettant d'adapter le raisonnement par récurrence aux preuves relatives aux instructions répétitives et celle de *variant* permettant de montrer la finitude de leur exécution.

Nous avons montré que la notion de *composition* des algorithmes, quant à elle, permet d'établir et d'utiliser différents niveaux d'abstraction dans la conception des algorithmes, tout en mettant à nouveau en évidence l'importance des *spécifications*.

Enfin, nous avons proposé une notion d'*algorithme non détermi-*

— si on imagine ces listes comme étant la notation d'entiers écrits en base 13 ou, d'une façon plus générale, dans toute base n avec $n \geq 13$, la seconde liste représente un entier strictement inférieur à celui représenté par la première liste ;

— on en déduit que chaque transposition diminue strictement les entiers représentés par les listes successives obtenues, ce qui ne se peut qu'un nombre fini de fois.

On peut convenir que cette argumentation est généralisable à toutes les listes proposées au *TriParSuppressionsDInversions*.

67. Pour mémoire, aussi bien additionner des entiers que marquer des requêtes, voire faire un roux ou encore crocheter une maille en l'air.

niste.

Pour finir, la section suivante présente des implémentations possibles de l'algorithme de la multiplication russe en python (sans prétendre être un tutoriel de démarrage à la programmation avec ce langage).

6 Programmation en Python : la multiplication russe

Les processeurs éligibles à l'exécution de l'algorithme de la Multiplication Russe [68] doivent posséder comme outil de base la faculté de déterminer la parité d'un entier positif. Peu de langages de programmation proposent une fonction, native ou standard, remplissant ce cahier des charges, et le langage Python n'y fait pas exception [69]. Les variantes proposées ci-après tirent partie des possibilités alternatives offertes par ce dernier.

Listing 1: La multiplication russe avec modulo 2 pour déterminer la parité.

```
def multrusse_avecpariteparmodulo2(a,b) :
    """
    donnée : deux entiers a>=0 et b>=0
    résultat : retourne la valeur du produit de a
       par b
    """
    x = a
    y = b
    z = 0
    while x!=0 : # invariant : z+x*y = a*b
        if x%2 == 1 :
            z = z+y
        x = x//2
        y = y*2
    return z
```

Cependant, le concepteur peut abstraire le calcul de parité en tirant partie de la composition des algorithmes mise en œuvre sous forme d'un appel de fonction.

68. Algorithme 1 page 3.
69. Au contraire du langage Pascal qui propose la fonction booléenne *odd*.

Les fonctions conjointes *MultiplcationRusse* et *estimpair* en donnent une illustration [70].

Listing 2: La multiplication russe avec appel à une fonction *estimpair* pour déterminer la parité.

```
from estimpair import estimpair

def multrusse_avecunefonctionestimpair(a,b) :
    """
    donnée : deux entiers naturels a et b
    résultat : retourne la valeur du produit ab
    commentaire : appelle une fonction estimpair
    """
    x = a
    y = b
    z = 0
    while x!=0 :
        if estimpair(x) :
            z = z+y
        x = x//2
        y = y*2
    return z
```

Listing 3: fonction estimpair

```
def estimpair(x) :
    """
    donnée : un entiers naturel n
    résultat : retourne la valeur vrai si n est
       impair, faux sinon
    """
    return x%2 == 1
```

L'abstraction du calcul de parité peut être poussée plus loin en tirant partie de ce que le langage Python autorise que les fonctions admettent d'autres fonctions comme paramètres. Une illustration en est donnée par la fonction *multrusse_ avecimpairenparametre* [71].

Listing 4: La multiplication russe à l'aide d'une fonction passée en paramètre pour déterminer la parité.

70. Listings 2 et 3.
71. Listing 4.

```
def multrusse_avecimpairenparametre(a,b,impair):
    """
    donnée :
    - deux entiers a>=0 et b>=0
    - une fonction booléenne EstImpair qui pour
        tout entier n>=0 retourne vrai si n est
        impair et faux sinon
    résultat : retourne la valeur du produit de a
        par b
    """
    x = a
    y = b
    z = 0
    while x!=0 :
        if impair(x) :
            z=z+y
        x=x//2
        y = y*2
    return z
```

Ainsi, en ayant défini les deux autres façons de calculer la parité d'un entier naturel que proposent les fonctions *estimpair_bis* et *estimpair_ter*[72], l'évaluation de chacune des trois expressions suivantes :

- *multrusse_avecimpairenparametre(50, 89, estimpair)*,
- *multrusse_avecimpairenparametre(50, 89, estimpair_bis)*
- et *multrusse_avecimpairenparametre(50, 89, estimpair_ter)*

retourne la valeur 4450.

Listing 5: Calcul de parité bis

```
def estimpair_bis(n) :
    """
    donnée : un entiers n>=0
    résultat : retourne la valeur vrai si n est
        impair, faux sinon
    """
    return 2*(n//2) != n
```

Listing 6: Calcul de parité ter

72. Listings 5 et 6.

```
def estimpair_ter(x) :
    """
    donnée: un entier n>=0
    résultat: retourne la valeur vrai si n est
       impair, faux sinon
    """
    while x>1 :
        x = x-2
    return x == 1
```

Enfin, à titre d'illustration, la fonction *multrusse_ aveccalculdepariteintegre*[73] implémente l'algorithme *MultiplicationDuPaysanRusseIncluantLaRechercheDeParité*[74]

Listing 7: La multiplication russe avec un calcul intégré pour déterminer la parité.

```
def multrusse_aveccalculdepariteintegre(a,b) :
    """
    donnée:   deux entiers naturels a et b
    résultat:   retourne la valeur du produit ab
    """
    x = a
    y = b
    z = 0
    while x!=0 :
        t = x
        while t>1 :
            t = t-2
        if t==1 :
            z = z+y
        x = x//2
        y = y*2
    return z
```

73. Listing 7.
74. Agorithme 10 page 34.