Algorithmique, programmation

Lionel GUEZ*
guez@lmd.ens.fr
Bureau E224

23 juillet 2018
École normale supérieure – L3 sciences de la planète

Table des matières

1 Introduction

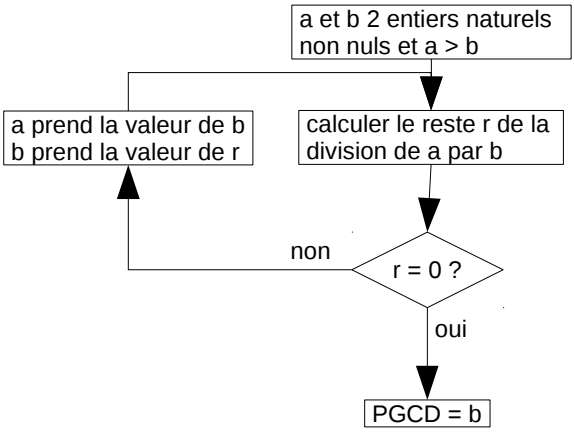
Ce cours présente des concepts communs aux divers langages de programmation utilisés en calcul scientifique et des conseils généraux de programmation.

*Emprunts nombreux au cours de Philippe FACON, Institut d'informatique d'entreprise, 1988.

Nous nous plaçons donc à un niveau logique, que nous distinguons de l'écriture du code des programmes. L'idée est de repousser à une étape ultérieure les problèmes de programmation spécifiques à un langage de programmation particulier et les problèmes de détail.

Le produit du travail au niveau logique est un "algorithme". Un algorithme est une suite finie, séquentielle, de règles que l'on applique à un nombre fini de données, pour résoudre des classes de problèmes semblables. Par exemple : l'algorithme d'Euclide permet de trouver le plus grand commun diviseur de deux nombres entiers.

Exemple : algorithme d'Euclide



Ce cours introduit un outil pour travailler au niveau logique : le langage de description d'algorithme. Le langage proposé ici est textuel (il existe aussi des descriptions graphiques d'algorithmes, comme illustré pour l'algorithme d'Euclide). On trouve aussi l'appellation pseudo-code pour ce langage textuel de description d'algorithme.

Ce cours : langage textuel de description d'algorithme ("pseudo-code")

entrer (a, b)
{a et b 2 entiers naturels non nuls et a > b}
r = reste de la division de a par b
tant que r $\neq$ 0 **faire**
  a = b
  b = r
  r = reste de la division de a par b
fin tant que
écrire (b)

2 Concepts de base des langages de programmation impératifs

Les langages de programmation classiquement utilisés en calcul scientifique sont dits "impératifs". Dans un tel langage, un programme est une suite d'ins-

tructions, dans un ordre bien défini, qui modifient “l’état de la mémoire”. La mémoire est essentiellement un ensemble de “variables”, qui contiennent des valeurs, en général des nombres ou du texte. On peut imaginer les variables comme des cases portant des noms. Le programme lit ou modifie les valeurs contenues dans ces cases. Lorsqu’un programme impératif est exécuté, la mémoire passe donc par une suite finie d’états. Cf. figure (1).

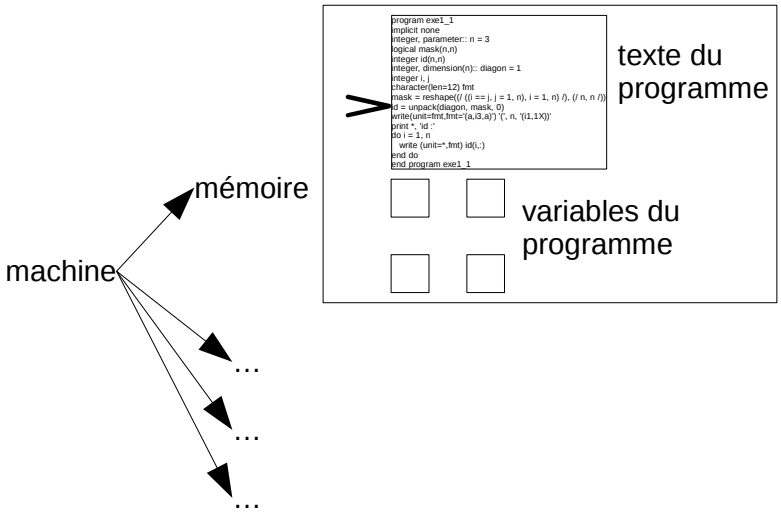


FIGURE 1 – Exécution d’un programme. La mémoire de la machine passe par une suite finie d’états. Le grand rectangle contenant le texte du programme avec des variables représente un état de la mémoire. Le pointeur à gauche du texte du programme représente la position courante dans ce texte.

Définitions

- *Type* d’une variable : type de la valeur qu’elle contient. Exemples : nombre entier, ou texte. Le type détermine les opérations possibles sur la variable. Exemple : + - × / pour les entiers.
- *Constante littérale* : une valeur, écrite littéralement. Exemple : 2,718 282.
- *Constante symbolique* : un nom donné à une valeur littérale. Exemple : e pour la base des logarithmes népériens. Non modifiable par le programme.
- *Expression* : combinaison d’opérations appliquées à des variables ou des constantes.
- *Procédure* : suite d’instructions à laquelle on donne un nom.

Typage statique, déclarations. Dans certains langages de programmation, le type d’une variable donnée est fixé à l’écriture du programme. On dit que le typage est statique dans ce langage de programmation. Par exemple, les langages Fortran, C et C++ sont à typage statique. Le type d’une variable est alors fixé dans une ligne particulière du programme qu’on appelle une ligne de “déclaration”. En plus des déclarations de types des variables, on peut aussi trouver des déclarations qui définissent de nouveaux types (ne pas confondre la définition d’un nouveau type et la déclaration d’une nouvelle variable avec un

type pré-existant), des déclarations de constantes symboliques, des procédures. Les lignes de déclarations ne modifient pas l'état de la mémoire, elles définissent ou caractérisent des objets que le programme manipule.

Typage dynamique Dans d'autres langages de programmation, une variable de nom donné peut voir son type changer en cours d'exécution. Le type de la variable est déterminé par la valeur affectée à cette variable. On dit que le typage est dynamique dans ce langage de programmation. Le langage Python par exemple est à typage dynamique.

Types entier et réel. Les langages de programmation scientifique permettent d'utiliser des valeurs de type entier ou de type réel. La distinction entre entiers et réels n'est pas la même en informatique qu'en mathématiques. Mathématiquement, si un nombre est entier, il est aussi réel, tandis que, informatiquement, une valeur est ou bien entière ou bien réelle. Chaque langage offre donc une possibilité de distinguer une valeur entière d'une valeur réelle. Par exemple, dans les langages scientifiques classiques, les écritures `2` et `2.` (avec un point) distinguent la valeur 2 considérée comme entière ou réelle. En outre, pour les langages à typage statique, la déclaration d'une variable distingue le type entier et le type réel. Une valeur entière n'est pas traitée comme une valeur réelle. La différence est dans le "codage" de la valeur. Le codage est la représentation d'un nombre dans la mémoire de l'ordinateur. Les nombres sont représentés par leur développement en base 2. Le développement est exact pour les nombres entiers. Pour les nombres réels, le codage inclut une mantisse et un exposant, avec un nombre de chiffres binaires (des "bits") fixé, et le développement est approché. Par exemple, le développement binaire du nombre 0,1 est infini donc son codage informatique ne peut qu'en donner une approximation. De même, les opérations sur les entiers donnent des résultats exacts tandis que les opérations sur les réels donnent des résultats approchés. En conclusion, le choix entre l'utilisation d'une valeur informatique entière ou réelle n'est pas anodin. Préférez les valeurs informatiques entières si vous n'avez besoin mathématiquement que de valeurs entières.

Structures algorithmiques. Le programme enchaîne des opérations dans des structures. On distingue trois structures :

la séquence :

instruction 1
instruction 2
instruction 3

l'alternative :

si a $>$ b **alors**
...
sinon
...
fin si

l'itération : répéter

(La signification de ces structures sera détaillée plus bas.) Ces trois structures sont suffisantes. C'est-à-dire que, théoriquement, tout algorithme peut être écrit

avec seulement ces trois structures.

Mémoire vive versus espace disque. L'information nécessaire à l'exécution d'un programme inclut la liste des instructions à exécuter et l'ensemble des objets manipulés par le programme, variables, constantes, avec leurs valeurs. Cf. figure (1). Lors de l'exécution d'un programme, toute cette information est conservée dans une partie de l'ordinateur appelée "mémoire vive", ou encore "mémoire centrale". Cette information n'existe que le temps de l'exécution du programme. La mémoire vive, dédiée à l'exécution des programmes, n'a donc en général rien à voir avec l'espace disque sur lequel vous stockez vos fichiers.

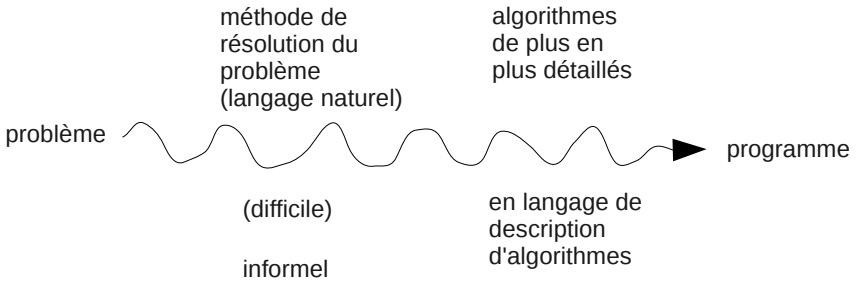
L'affichage à l'écran et la lecture sur le clavier sont analogues à l'écriture et la lecture dans un fichier. On considère en général dans les langages de programmation l'écran et le clavier comme des fichiers particuliers. L'écran est un fichier sur lequel on ne peut qu'écrire et le clavier est un fichier sur lequel on ne peut que lire.

Les notions définies dans cette partie que vous devez avoir bien comprises : variable, type d'une variable, constante littérale, constante symbolique, expression, typage statique ou dynamique, déclaration, mémoire vive.

3 Langage de description d'algorithme

À partir de l'énoncé d'un problème de programmation, avant d'arriver au programme, nous pouvons d'abord décrire dans un langage naturel la méthode de résolution du problème. Nous pouvons passer ensuite à la description dans un "langage de description d'algorithme".

Intérêt d'un langage de description d'algorithme



Langage de description d'algorithme

- Pour n'importe quel problème (intégration numérique, échecs, paye, ...).
- Caractéristiques principales d'un langage de programmation mais avec une syntaxe plus libre, et des parties informelles à détailler plus tard.
- Ce cours : un langage possible. Non universel (ici : notations proches du Fortran). Ce qui importe : le niveau de formalisation du langage, intermédiaire entre un langage naturel et un langage de programmation.

3.1 Variables et types

On utilise des types de base :

Type entier : Ensemble de valeurs : $\mathbb{Z}$, ensemble des entiers relatifs. Opérations arithmétiques usuelles.

Type réel : Ensemble de valeurs : $\mathbb{R}$. Opérations arithmétiques usuelles.

Type texte : Ensemble des suites de caractères. Une suite de caractères est notée entre guillemets :

```
"Il␣fait␣beau."
```

Une opération de base sur les suites de caractères est la concaténation, notée // :

```
texte1 // texte2
```

Type logique : Seulement deux valeurs possibles : vrai, faux. Les opérations sur ce type sont : et logique, ou logique, négation, implication ...

On peut aussi utiliser, en plus des types de base, des types qui sont construits à partir des types de base. Entre autres :

Type partie d'ensemble : Les valeurs sont les parties d'un type donné. Les opérations sont : réunion $\cup$, intersection $\cap$, appartenance $\in$ ...

Type cartésien : Le type est défini par une succession particulière de champs, chaque champ pouvant avoir un type quelconque. On peut définir, par exemple, un type appelé "personne" contenant un champ nom, de type texte, et un champ `nombre_enfants`, de type entier.

En langage de description d'algorithme, on adopte un typage statique. On déclare donc le type des variables. On notera la déclaration sous la forme "type variable". Par exemple :

```
entier i
```

Ce qui signifie que i est une variable de type entier. On peut déclarer des variables, des types (autres que les types de base) et des constantes symboliques. En langage de description d'algorithme, on écrira les constantes symboliques avec le mot-clef "constante". Par exemple :

```
constante réelle pi = 3.141592654
```

Dans une expression, les constantes ou variables sont combinées avec des opérateurs. Les opérateurs utilisés doivent être cohérents avec le type des objets combinés. Une expression peut être plus ou moins compliquée. Elle peut se réduire simplement à une valeur littérale, par exemple la valeur entière 3, ou la valeur chaîne de caractères "bonjour". Une expression peut aussi être simplement une variable, par exemple la variable réelle x. Une expression un cran plus compliquée, contenant une opération, est par exemple l'expression x + 3.

3.2 Les tableaux

Le tableau est une structure de données. Les *éléments* d'un tableau contiennent des valeurs d'un même type. Chaque élément du tableau a un *indice*, qui est une valeur entière. Les indices prennent toutes les valeurs comprises entre une borne inférieure et une borne supérieure. La plage d'indices peut éventuellement avoir une partie négative. Exemple de déclaration de tableau :

```
réel tab(1: 10)
```

qui signifie que tab est une variable tableau, qui contient des éléments de type réel, avec des indices entre 1 et 10 (tab contient donc 10 éléments). On peut généraliser à plusieurs indices :

```
réel tab2(1: 10, -2: 3)
```

qui signifie que tab2 est un tableau de réels, avec un premier indice variant de 1 à 10 et un second indice variant de -2 à 3 (tab2 contient donc 60 éléments). On dira que tab2 a deux *dimensions*.

Notation pour l'accès à un élément du tableau, par exemple pour un tableau à une dimension :

nom_tableau(expression entière)

L'expression entière donne une valeur d'indice et détermine donc l'élément de tableau référencé. Exemples :

```
tab(i * j - 3)
tab(i) + tab2(j, - 2)
```

où i et j sont des variables entières.

3.3 Les instructions simples

Les instructions simples sont les entrées-sorties et l'affectation.

Les entrées-sorties sont des échanges entre la mémoire centrale et l'extérieur : écran, clavier, mémoire de stockage. Pour les entrées, la syntaxe du langage de description d'algorithme est :

```
entrer (ma_variable)
```

qui signifie : attendre une valeur de l'extérieur, l'affecter à `ma_variable`. On pourra aussi écrire l'instruction avec plusieurs variables, par exemple :

```
entrer (x, y, z)
```

Pour les sorties, la syntaxe est :

```
écrire (expression)
```

qui signifie : calculer la valeur de l'expression et l'écrire. On pourra aussi écrire plusieurs expressions :

```
écrire (expression1, expression2, ...)
```

La syntaxe pour l'affectation est :

```
ma_variable = expression
```

qui signifie : calculer la valeur de l'expression et l'affecter à `ma_variable`. Bien distinguer l'affectation du test d'égalité de deux valeurs. On notera == le test d'égalité. Exemple :

```
x = y == z
```

où `x` est une variable de type logique. Dans cet exemple, y et z peuvent par exemple être des variables de type entier, et $y == z$ est une expression de type logique.

3.4 Assertions

Les assertions sont des propositions vraies (au sens de la logique mathématique, on dit encore des prédicats) qui décrivent l'état de la mémoire, ou d'une partie de la mémoire. Comme déjà indiqué dans le § 2, chaque instruction peut être interprétée comme une transformation de l'état de la mémoire. Les assertions permettent donc de caractériser formellement l'effet des instructions. Il est possible de pousser très loin cette idée de caractériser formellement des instructions, pour arriver à prouver que des algorithmes ou des programmes sont corrects. Mais il est aussi utile d'insérer dans un algorithme des assertions comme de simples commentaires, qui permettent de se repérer dans le cours d'un algorithme.

On notera les assertions entre accolades. Par exemple :

```
{x + q * y == a et y > q}
x = x - y
```

Nous avons une assertion avant l'instruction d'affectation. Quelle assertion pouvons-nous écrire après l'affectation ? La valeur de x a été modifiée par l'affectation. Si nous notons x' l'ancienne valeur, nous avons :

```
{x' + q * y == a et y > q et x == x' - y}
```

Éliminons l'ancienne valeur pour obtenir une assertion entre les variables qui existent dans l'algorithme. Nous pouvons écrire après l'affectation :

```
{x + (q + 1) * y == a et y > q}
```

On parlera aussi de "schémas", de la forme :

```
{P} I {Q}
```

où I est une instruction, l'assertion P s'appelle la pré-condition du schéma et l'assertion Q s'appelle la post-condition du schéma. Le schéma signifie que si on a P avant l'exécution de I alors on aura forcément Q après exécution de I. Par exemple, on a le schéma :

```
{x + q * y == a et y > q} x = x - y {x + (q + 1) * y == a et y > q}
```

3.5 Les instructions composées

Les instructions composées sont la séquence, l'alternative et l'itération.

3.5.1 La séquence

Syntaxe de la séquence : on écrira simplement une instruction par ligne.

```
instruction_1
instruction_2
instruction_3
```

qui signifie : exécuter `instruction_1` puis, de l'état de la mémoire obtenu, exécuter `instruction_2`, etc.

Exemple d'assertions avec une séquence :

```
{x == a et y == b}
x = x + y
{x == a + b et y == b}
y = x - y
{x == a + b et y == a}
```

L'ordre d'écriture des instructions dans le langage de description d'algorithme (et dans les langages de programmation impératifs) est donc fondamental.

3.5.2 L'alternative

Syntaxe de l'alternative :

si expression logique **alors**
  instruction_1
sinon
  instruction_2
fin si

qui signifie : évaluer l'expression logique ; si elle est vraie, exécuter `instruction_1`, sinon exécuter `instruction_2`.

On peut caractériser formellement l'effet d'une alternative en utilisant des assertions. Sachant qu'on a les schémas :

```
{P et C} I1 {P1}
{P et non C} I2 {P2}
```

alors on a aussi le schéma :

{P}
si C **alors**
  I1
sinon
  I2
fin si
{P1 **ou** P2}

Prenons un autre exemple de raisonnement avec des assertions et une alternative. Sachant qu'on a les schémas :

```
{Q1} I1 {Q}
{Q2} I2 {Q}
```

alors on a aussi le schéma :

{(Q1 **et** C) **ou** (Q2 **et non** C)}
si C **alors**
  I1
sinon
  I2
fin si
{Q}

On peut utiliser les variantes suivantes de l'alternative. Sans la partie "sinon" :

si expression logique **alors**

instruction_1
fin si

Avec plusieurs parties “sinon” :

si expression logique 1 **alors**
instruction_1
sinon si expression logique 2 **alors**
instruction_2
sinon
instruction_3
fin si

3.5.3 L’itération

Syntaxe de l’itération (qu’on appelle aussi “boucle”) :

tant que condition **faire**
instruction
fin tant que

où la condition est une expression logique. Ce qui signifie : évaluer la condition ; si elle est vraie, exécuter l’instruction puis revenir à l’évaluation de la condition ; si la condition est fausse, ne rien faire.

Attention : la condition ne reste pas forcément vraie lors de l’exécution de l’instruction intérieure, c’est ce qui permet de sortir de l’itération après un certain nombre de passages. La valeur de la condition : vraie ou fausse, n’est testée qu’à chaque passage de la ligne “tant que”.

Deux modes de raisonnement correspondent à l’itération. Premier mode : répétition d’un même traitement sur des objets différents. Par exemple :

tant que lecture d’une ligne de mon fichier réussie **faire**
traitement à partir de la ligne lue
essai de lecture d’une nouvelle ligne de mon ficher
fin tant que

Deuxième mode de raisonnement : calcul du terme final d’une suite définie par récurrence. Dans ce cas, on utilise à chaque passage le résultat du passage précédent. Cf. exemple de l’algorithme (1).

Algorithme 1 Exemple d’algorithme de calcul du terme final d’une suite définie par récurrence.

```
entrer (a, b)
i = 0
s = 0
tant que i < b faire
    i = i + 1
    s = s + a
fin tant que
écrire (s)
```

Comme pour l’alternative, nous pouvons caractériser formellement l’effet d’une itération à l’aide des assertions. Notons C la condition de l’itération et I l’instruction à l’intérieur de l’itération :

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

`P` désignant une proposition, si on a le schéma :

```
{P et C} I {P}
```

alors on a aussi le schéma :

{P}
tant que C **faire**
  I
fin tant que
{P **et non** C}

`P` est appelé un invariant de la boucle.

Exhibons un invariant de boucle sur l'exemple trivial de l'algorithme (1) :

entrer (a, b)
i = 0
s = 0
{s == a * i}
tant que i ≠ b **faire**
  i = i + 1
  s = s + a
  {s == a * i}
fin tant que
{s == a * i **et** i == b}
écrire (s)

L'utilisation des assertions a ainsi permis de prouver l'effet de l'algorithme : l'algorithme multiplie a par b.

Lorsque l'algorithme itératif ne semble pas évident à écrire, l'utilisation des invariants de boucle permet de bien poser le problème et d'arriver à une solution sûre. La méthode conseillée pour écrire un algorithme itératif est donc :

1. Trouver une idée informelle de la boucle.
2. La traduire en invariant `INV`.
3. Trouver une condition de sortie `CS` telle que : (invariant et condition de sortie) ⇒ résultat final.
4. Initialiser avec une suite d'instructions `INIT` telle qu'on ait l'invariant après cette initialisation :
   INIT {INV}
5. Trouver une instruction `I` qui permette de se rapprocher de `CS` en gardant `INV` vrai.
6. Il ne reste qu'à assembler les éléments précédents. L'algorithme s'écrit :

   INIT
   tant que non CS **faire**
     I
   fin tant que

Nous avons vu l'itération avec la structure "tant que". L'itération peut aussi s'écrire sous quelques formes variantes. La variante "répéter" :

répéter
  I
jusqu'à C

qui signifie : exécuter l'instruction I ; évaluer la condition C ; si elle est fausse, revenir à l'exécution de l'instruction I. La seule différence logique avec la structure "tant que" est que, puisque le test sur C est fait à la fin de la structure "répéter", l'instruction I est exécutée au moins une fois. Une itération avec la structure "répéter" peut toujours s'écrire de façon équivalente comme une itération avec "tant que" :

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

L'écriture de l'itération avec "répéter" est plus concise si elle est possible (c'est-à-dire si l'instruction I doit être exécutée au moins une fois).

Une autre variante de l'itération est la variante "pour" :

pour X = V1 **à** V2 **faire**
  I
fin pour

où `X` est un nom de variable d'un type discret, ordonné, et `V1`, `V2` sont des valeurs du type de `X`, avec `V1` $\leq$ `V2`, et `X` ne doit pas être modifié par l'instruction I. Cette syntaxe signifie exactement :

X = V1
tant que X $\leq$ V2 **faire**
  I
  X = successeur(X)
fin tant que

Noter que dans l'écriture avec "pour", le changement de valeur de X est implicite à chaque passage à la ligne "pour".

Le plus souvent, on utilisera la construction "pour" avec une variable sur laquelle on itère de type entier. Par exemple :

pour i = 1 **à** 10 **faire**
  I
fin pour

qui est équivalent à :

i = 1
tant que i $\leq$ 10 **faire**
  I
  i = i + 1
fin tant que

L'écriture avec "pour" est plus concise ; on ne peut l'utiliser que si on connaît avant le début de la boucle le nombre d'itérations à réaliser.

4 Conseils de présentation des algorithmes et programmes

Cette partie donne quelques conseils de présentation.

En langage de description d'algorithme, on écrira toutes les déclarations avant les instructions. De manière générale, en langage de description d'algorithme et dans n'importe quel langage de programmation, faites l'effort de donner un nom significatif aux variables. Il est conseillé aussi de regrouper les déclarations selon leur sujet (plutôt que selon le type, ou autre attribut de forme).

La clarté de l'algorithme ou du programme repose en partie sur l'indentation des instructions. Le principe de l'indentation est de faire commencer les instructions d'une séquence à la même colonne, tandis que, dans les alternatives et les itérations, les "blocs intérieurs" sont décalés. Par exemple :

```
1: X = 3
2: si C alors
3:    instruction
4:    instruction
5: sinon
6:    instruction
7: fin si
8: Y = 2
```

Les lignes numéros 1, 2, 5, 7 et 8 commencent à la même colonne. Les lignes 3, 4 et 6 sont décalées. Si une alternative ou une itération est imbriquée dans une autre, on décale à nouveau le bloc correspondant. Par exemple :

```
1: X = 3
2: si C alors
3:    instruction
4:    tant que C2 faire
5:       instruction
6:    fin tant que
7:    instruction
8: sinon
9:    instruction
10: fin si
11: Y = 2
```

La ligne 5 est encore décalée par rapport à la ligne 4, qui est déjà décalée par rapport à la ligne 2.

Les commentaires sont en général nécessaires tout au long de l'algorithme ou du programme. Ils peuvent être de plusieurs types et il est utile de choisir des notations différentes en langage de description d'algorithme pour les distinguer :

Commentaire titre : titre d'une partie de l'algorithme. Notation possible : entre étoiles. Par exemple :

```
* Tri du tableau *
```

Commentaire assertion Notation : entre accolades `{...}`.

Autre commentaire : explication du rôle d'une variable au moment de sa dé-

claration ; explications sur un passage délicat, etc. Notation possible : marquer le début du commentaire par un point d'exclamation. Par exemple :

```
réel omega ! vitesse verticale, en Pa s-1
```

L'interface avec l'utilisateur peut être oubliée dans l'écriture de l'algorithme mais requiert un minimum d'effort dans le programme final. C'est-à-dire que, dans le programme final, chaque demande de donnée, chaque sortie de résultat, devra être précédée d'un message explicite.

5 Conception descendante

La conception descendante est une méthode de conception d'algorithme. Principe : concevoir l'algorithme niveau par niveau, de plus en plus détaillé. Chaque niveau s'écrit à l'aide des trois structures algorithmiques, plus des parties simplement spécifiées, dont le contenu n'est pas donné.

Prenons un exemple de problème : le tri d'un tableau, que nous allons résoudre par conception descendante. Nous avons le tableau :

```
entier t(1:20)
```

Nous voulons trier les valeurs par ordre croissant. Nous avons l'intuition que l'algorithme doit être itératif. Suivons la méthode conseillée plus haut pour écrire un algorithme itératif.

Première étape : nous devons trouver une idée informelle de l'algorithme. Considérons par exemple une séquence de quatre valeurs. Nous progressons dans le tri case par case, vers la droite. Si par exemple nous avons au départ la séquence de valeurs :

```
3 7 5 2
```

Nous cherchons la plus petite valeur dans la séquence et nous la plaçons en première position :

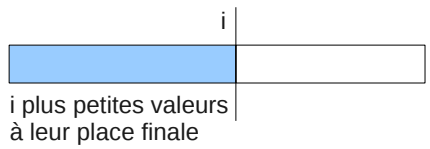
```
2 3 7 5
```

À l'itération suivante, le 3 est déjà à la bonne position. Enfin, avec une itération supplémentaire, nous obtenons :

```
2 3 5 7
```

(Ce tri est appelé tri par sélection. C'est en fait un algorithme de tri très lent, il a juste l'avantage d'être simple.)

Seconde étape : nous devons formaliser l'idée précédente en la traduisant en invariant. Quel est l'invariant ? À un moment donné de la progression du tri, les `i` plus petites valeurs sont à leur place finale dans le tableau (en bleu dans le schéma ci-dessous) :



La proposition :

```
{les i plus petites valeurs sont à leur place}
```

est l'invariant. La condition de sortie est :

```
i == n - 1
```

L'initialisation s'écrit simplement :

```
i = 0
```

Après cette initialisation, nous avons bien l'invariant.

Nous devons trouver ensuite une suite d'instructions qui nous permette de nous rapprocher de la condition de sortie en gardant l'invariant vrai :

```
i = i + 1
p = la place du plus petit élément entre i et n inclus
permuter t(i) et t(p)
```

Dans les lignes ci-dessus, nous n'avons pas précisé comment nous allions trouver la place du plus petit élément entre i et n inclus. De même, nous n'avons pas précisé les instructions qui permettent de permuter t(i) et t(p). C'est en écrivant ainsi un premier algorithme de haut niveau avec des parties dont le contenu n'est pas encore donné, que nous suivons une conception descendante.

Nous regroupons l'initialisation, la condition de sortie et les instructions du corps de la boucle pour obtenir l'algorithme de plus haut niveau : l'algorithme (2). Pour le niveau suivant de l'algorithme, nous explicitons ce que nous avions

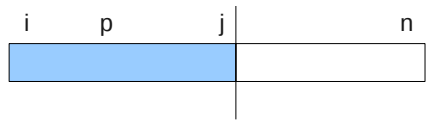
Algorithme 2 Algorithme de plus haut niveau pour le tri d'un tableau.

```
i = 0
tant que i ≠ n - 1 faire
    i = i + 1
    p = la place du plus petit élément entre i et n inclus
    permuter t(i) et t(p)
fin tant que
```

simplement spécifié :

```
p = la place du plus petit élément entre i et n inclus
```

À nouveau, nous suivons la méthode du paragrahe sur l'itération. Idée informelle : nous progressons vers la droite à partir de l'élément i.



Invariant :

```
{p est la place du petit élément entre i et j}
```

Condition de sortie :

```
j == n
```

Initialisation :

```
j = i
p = i
```

Les instructions à placer à l'intérieur de la boucle, qui nous rapprochent de la condition de sortie, en gardant l'invariant vrai, sont :

```
j = j + 1
si t(p) > t(j) alors
  p = j
fin si
```

D'où l'algorithme (3), de second niveau (algorithme final dans cet exemple).

Algorithme 3 Algorithme de second niveau pour le tri d'un tableau.

```
i = 0
tant que i ≠ n - 1 faire
  i = i + 1

  * recherche de la place p du plus petit élément entre i et n inclus *
  j = i
  p = i
  tant que j ≠ n faire
    j = j + 1
    si t(p) > t(j) alors
      p = j
    fin si
  fin tant que

  * permutation de t(i) et t(p) *
  c = t(i)
  t(i) = t(p)
  t(p) = c
fin tant que
```

6 Idéaux de la programmation

Un programme peut avoir pour ambition d'être :

- Correct. Le programme doit donner la bonne réponse à la question posée, ou annoncer son échec, mais ne pas présenter une réponse erronée. On dit aussi que le programme ne doit pas avoir de "bogue".
- Économe en mémoire vive. Pour pouvoir être exécuté sur la plus grande variété de machines possibles, ou ne pas bloquer toute une machine. Notamment, on peut faire attention à ne pas utiliser plus de tableaux que nécessaire.
- Rapide. Sa rapidité est mesurée par le temps écoulé entre le moment où il est lancé et le moment où il se termine.
- Clair. L'algorithme doit être aussi simple que possible, et aussi apparent que possible dans le programme, via la présentation (noms des variables, indentation, etc.).

Ces quatre qualités sont des qualités idéales vers lesquelles un programme peut tendre, plutôt que des points d'arrivée. Même le fait d'être correct est plutôt un idéal pour un gros programme : on ne peut en général garantir l'absence de bogues, et on continue à en trouver des années après avoir commencé à utiliser le programme.

Entre ces quatre idéaux, on peut donner la priorité à celui d'être correct. Les trois autres idéaux ne sont en général pas solidaires. C'est-à-dire que, souvent, en tentant de se rapprocher de l'un, on s'éloigne de l'autre. Cf. figure (2). Par

FIGURE 2 – Tensions entre les idéaux vers lesquels tendent un programe.

exemple, le programme peut être plus rapide en stockant plus de résultats en mémoire vive. Ou le programme peut être plus rapide via une astuce algorithmique, au prix de sa clarté. Vous pourrez donc avoir à faire, pour un programme donné, des arbitrages entre économie de mémoire, rapidité et clarté.

7 Procédures

L'idée est de décomposer un problème en sous-problèmes. La "procédure" isole un sous-problème. Pour définir la procédure, on explicite ce dont elle dépend et quels résultats elle produit : ce sont les données et les résultats de la procédure. À chaque procédure on donne un nom. En un point quelconque de l'algorithme principal, on peut décider d'exécuter les instructions de la procédure en "appelant" la procédure par son nom.

Certains langages de programmation (par exemple Fortran et Pascal) distinguent deux types de procédures : les sous-algorithmes et les fonctions. D'autres langages (par exemple Python et C) ne possèdent que des fonctions. En langage de description d'algorithme, nous ferons cette distinction entre sous-algorithme et fonction, et nous utiliserons donc le terme de procédure pour recouvrir ces deux entités. Nous expliquerons plus bas la distinction.

On peut relever plusieurs intérêts à isoler des procédures :

— la *non duplication* de lignes si on appelle plusieurs fois une procédure dans un même algorithme ;
— la possibilité de *ré-utilisation* dans un autre algorithme ;
— l'amélioration de la clarté de l'algorithme, en rendant la *décomposition apparente* ;

— la *simplicité* des procédures isolées ;
— la facilité de *mise au point* de l'algorithme et des procédures.

L'isolation complète signifierait que l'algorithme appelant et la procédure forment deux mondes complètement séparés : les variables de l'un ne seraient pas connues de l'autre et ne pourraient pas être modifiées dans l'autre. L'isolation ne peut en général être complète puisque les données et les résultats doivent passer entre l'algorithme appelant et la procédure.

Distinguons maintenant sous-algorithme et fonction. Pour un sous-algorithme, le moyen recommandé pour la communication restreinte entre appelant et appelé est de passer par des "arguments". Cf. figure (3). Un "argument effectif" est

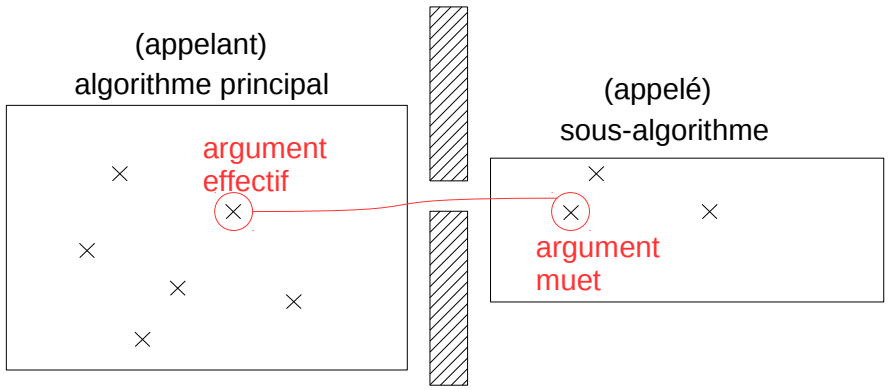


FIGURE 3 – La communication entre algorithme appelant et sous-algorithme doit normalement être réduite à l'association d'arguments.

un objet de l'appelant qui est associé à un objet de l'appelé, "l'argument muet" correspondant. En anglais : *actual argument* pour argument effectif, *dummy argument* ou *formal argument* pour argument muet. On choisit des arguments effectifs à chaque appel du sous-algorithme. On dit qu'un argument muet est une donnée du sous-algorithme si la valeur provenant de l'argument effectif correspondant est seulement utilisée dans le sous-algorithme, ou en d'autres termes si la valeur de l'argument muet n'est pas modifiée dans le sous-algorithme. Si le sous-algorithme n'a pas besoin d'une valeur initiale pour un argument et donne lui-même une valeur à l'argument alors l'argument muet est un argument résultat du sous-algorithme. Un argument muet donnée-résultat est un argument pour lequel le sous-algorithme a besoin d'une valeur initiale et que le sous-algorithme modifie. Il est important de déclarer pour chaque argument muet s'il est donnée, résultat ou donnée-résultat. Le style de programmation le plus clair consiste à ne laisser que le passage d'arguments comme communication entre appelant et sous-algorithme.

Le concept de fonction est très proche du concept de sous-algorithme. La fonction, comme le sous-algorithme, a une liste d'arguments muets. La différence est que la fonction a en outre une "valeur de retour" (qui n'est pas un argument). La valeur de retour est un des résultats de la fonction, qui a un statut particulier, une sorte de résultat principal.

Le style de programmation le plus clair consiste à ne laisser que des arguments d'entrée à une fonction. On peut appeler "fonction pure" une fonction dont tous les arguments sont des arguments d'entrées et qui ne fait pas d'entrées-sorties dans son corps (pas de lecture au clavier ou dans un fichier, pas de sortie à l'écran ou dans un fichier). La valeur de retour d'une fonction pure est non seulement le principal mais le seul résultat de la fonction. En langage de description d'algorithme, toutes les fonctions que l'on écrira seront pures.

Pour insister encore sur la différence entre fonction et sous-algorithme, pour un sous-algorithme : il n'existe pas de valeur de retour, tous les éventuels résultats doivent normalement passer par les arguments.

Le fait qu'il existe une valeur de retour pour une fonction et non pour un sous-algorithme implique qu'on appelle l'un et l'autre de manières différentes. Prenons un exemple. On a isolé le sous-problème du calcul du minimum de trois nombres. On peut définir un sous-algorithme, auquel on donne le nom `min3`, qui a trois arguments muets données, les trois nombres dont on cherche le minimum, et qui a un argument muet résultat, le minimum cherché. Cf. figure (4). Ou bien

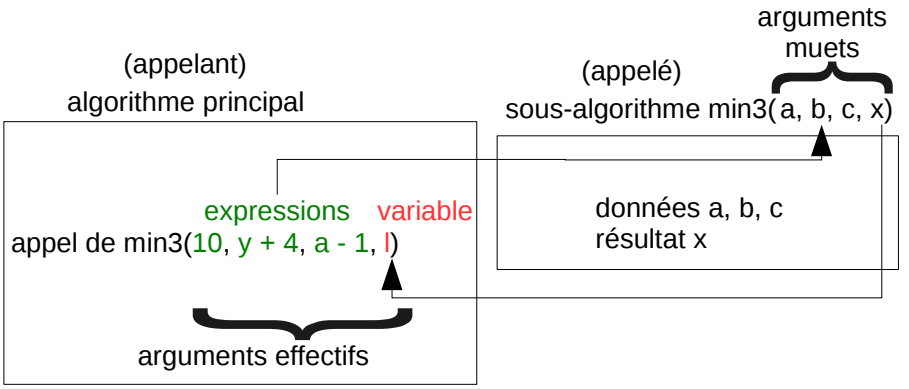


FIGURE 4 – Appel d'un sous-algorithme. Exemple d'un sous-algorithme cherchant le minimum de trois nombres.

on peut définir une fonction `min3` qui a trois arguments muets données et dont la valeur de retour est le minimum cherché. En langage de description d'algorithme, dans l'algorithme appelant, on appelle une procédure en faisant apparaître son nom suivi, entre parenthèses, de la liste d'arguments effectifs pour cet appel. Pour une fonction, on écrit l'appel comme on écrirait une valeur quelconque, étant entendu que la valeur référencée ainsi est la valeur de retour de la fonction. En d'autres termes, l'appel de fonction doit être utilisé directement, par exemple dans une affectation, ou comme partie d'une expression, ou dans une sortie :

```
r = min3(a, b, c) * 3
```

ou :

```
écrire(min3(a, b, c))
```

Mais, pour un sous-algorithme, on ne peut écrire l'appel comme on écrirait une valeur, on écrit simplement l'appel seul sur une ligne. Si min3 est un sous-algorithme et non une fonction, on ne peut qu'écrire :

```
appel de min3(a, b, c, r)
```

Ce qui ne sous-entend l'écriture d'aucune valeur. L'apparition de l'appel de sous-algorithme a seulement pour effet de donner une valeur aux arguments effectifs résultats. Dans l'exemple ci-dessus, l'appel de min3 donne une valeur à r. L'appel du sous-algorithme est une instruction entière à lui tout seul.

Si un argument muet est une donnée, l'argument effectif correspondant ne doit pas nécessairement être une variable, ce peut être une expression quelconque. On peut écrire par exemple :

```
appel de min3(10, y + 4, a - 1, l)
```

Ci-dessus : 10, y + 4 et a - 1 sont des expressions qui ne sont pas réduites à une variable. Par contre, si un argument muet est un résultat du sous-algorithme, alors l'argument effectif correspondant doit être un nom de variable. Cf. figure (4).

Les variables utilisées dans une procédure autres que les arguments muets sont appelées *variables locales*. Les variables locales à une procédure ne sont pas connues de l'extérieur de la procédure et ne peuvent pas être modifiées à l'extérieur de la procédure. Cf. par exemple l'algorithme (4). À la ligne 1, on

Algorithme 4 sous-algorithme min3(a, b, c, x)

Déclarations :

1: données entier a, b, c
2: résultat entier x
3: entier m

Instructions :

si a $\leq$ b **alors**
  m = a
sinon
  m = b
fin si
si m $\leq$ c **alors**
  x= m
sinon
  x = c
fin si

déclare a, b et c, comme des arguments de type entier, et comme des données du sous-algorithme. À la ligne 2, on déclare x comme un argument résultat de type entier. À la ligne 3, on déclare une variable entière m qui n'est pas dans la liste des arguments muets figurant entre parenthèses dans la ligne de titre du sous-algorithme. m est donc une variable locale du sous-algorithme. `m` n'est pas connue à l'extérieur de min3. Il peut y avoir une variable `m` dans l'algorithme principal, qui n'interfère pas avec la variable `m` de `min3`.

Remarque : nous avons relevé plus haut les intérêts à isoler une procédure, nous voyons qu'un inconvénient est l'écriture de lignes de déclaration pour les arguments muets de la procédure. Ces lignes apparaissent en plus des lignes de déclaration des éventuelles variables apparaissant dans les arguments effectifs

de l'appel de la procédure, dans l'algorithme appelant. Ce sont parfois des déclarations identiques. Ces lignes de déclaration supplémentaires à écrire sont le prix à payer pour avoir isolé et généralisé la tâche à réaliser.

Dans la déclaration d'une fonction en langage de description d'algorithme, on utilise le mot clef "fonction". Dans le corps de la fonction seulement, le nom de la fonction est considéré comme une variable, dans laquelle on doit placer le résultat de l'appel de la fonction. Dans l'exemple du minimum de trois nombres, on pourrait imaginer une fonction calculant le minimum de deux nombres et min3 pourrait appeler min2. Cf. par exemple algorithmes (5) et (6). À l'intérieur de la fonction min2, min2 est une variable comme une autre. min3

Algorithme 5 fonction min2(x, y)

entier min2
données entier x, y

Instructions :

si x $\leq$ y **alors**
  min2 = x
sinon
  min2 = y
fin si

Algorithme 6 fonction min3(a, b, c)

1: entier min3
2: données entier a, b, c

Instructions :

min3 = min2(min2(a, b), c)

ne contient qu'une instruction. À l'intérieur de la fonction min3, min3 est une variable comme une autre.

Les notions définies dans cette partie que vous devez avoir bien comprises : différence entre sous-algorithme et fonction, fonction pure, argument effectif, argument muet, argument donnée, argument résultat, argument donnée-résultat, variable locale.

7.1 Choix entre sous-algorithme et fonction pure

Une fonction peut toujours être récrite sous forme de sous-algorithme : il suffit d'ajouter un argument muet résultat qui remplace la valeur de retour de la fonction. La réciproque n'est pas toujours vraie pour une fonction pure. Puisqu'une fonction pure n'a que des arguments muets données, pour qu'un sous-algorithme puisse être récrit sous forme de fonction pure, il faut qu'il ne fournisse qu'un seul résultat, qui puisse devenir la valeur de retour de la fonction. Cf. figure (5). Selon les langages de programmation, il n'est pas toujours possible de regrouper tous les résultats sous la forme d'un objet unique. En langage de description d'algorithme, nous nous limitons à des structures simples (scalaires ou tableaux). On peut envisager de définir un type cartésien pour regrouper des

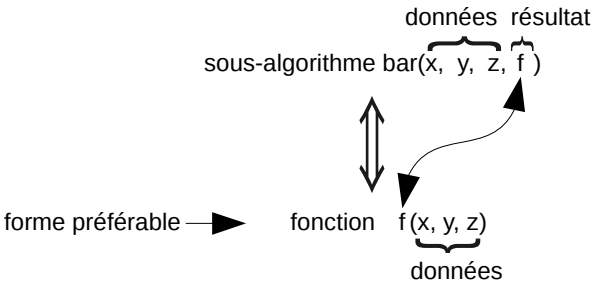


FIGURE 5 – Cas d'équivalence entre un sous-algorithme et une fonction pure. Un seul argument muet du sous-algorithme est un résultat, tous les autres arguments muets du sous-algorithme sont des données.

données hétérogènes mais cela ne se justifie pas si le seul but est de transformer un sous-algorithme en fonction.

Lorsqu'elle est possible simplement, l'écriture sous forme de fonction pure est avantageuse. Données et résultat sont plus clairement séparés. En outre, puisque le résultat de la fonction est utilisé comme une expression, l'utilisation de variables intermédiaires dans l'algorithme appelant peut parfois être évitée. Comparer par exemple les algorithmes (7) et (8) avec les algorithmes (5) et (6).

Algorithme 7 sous-algorithme min2(x, y, r)

```
données entier x, y
résultat entier r
si x ≤ y alors
  r = x
sinon
  r = y
fin si
```

Algorithme 8 sous-algorithme min3(a, b, c, x) avec appel de min2

```
données entier a, b, c
résultat entier x
entier m
appel de min2(a, b, m)
appel de min2(m, c, x)
```

Prenons un autre exemple : nous souhaitons lire des suites de trois caractères et écrire, pour chaque suite, si elle est dans l'ordre alphabétique croissant, décroissant ou ni l'un ni l'autre. L'utilisateur indiquera la dernière suite par ***. Pour effectuer ce traitement, nous écrivons une fonction et un algorithme principal. Cf. algorithmes (9) et (10). Noter dans cet exemple l'économie en variables intermédiaires grâce à l'utilisation d'une fonction.

Algorithme 9 Algorithme principal pour l'ordre d'une suite de trois caractères.

```
entrer(a,b, c)
tant que a // b // c ≠ "***" faire
  si ordc(a, b, c) alors
    écrire("ordre croissant")
  sinon si ordc(c, b, a) alors
    écrire("ordre décroissant")
  sinon
    écrire("ni croissant ni décroissant")
  fin si
  entrer(a,b, c)
fin tant que
```

Algorithme 10 fonction ordc(x, y, z)

```
logique ordc
données caractère x, y, z
Instructions :
ordc = x ≤ y et y ≤ z
```

8 Conception des algorithmes avec procédures

C'est une forme de conception descendante. On décompose un problème en sous-problèmes et on associe une procédure à chaque sous-problème. Chaque procédure doit être paramétrée, c'est-à-dire que ses arguments, et son éventuelle valeur de retour si c'est une fonction, doivent être décrits. En particulier, la paramétrisation indique si les arguments sont des données, des résultats ou des données-résultats. La spécification de la procédure précise l'effet de la procédure : les résultats (arguments résultats et éventuelle valeur de retour), les écritures dans un fichier par la procédure, et les valeurs finales des arguments données-résutats en fonction des arguments données de la procédure, des lectures dans un fichier faites dans la procédure, et des valeurs initiales des arguments données-résutats. Cf. figure (6). En d'autres termes, la paramétrisation décrit l'interface de la procédure avec l'extérieur, et la spécification indique quel est la mission de la procédure (sans décrire comment elle va remplir cette mission).

Par exemple, un sous-algorithme de tri peut avoir pour arguments :

```
tableau entier donnée-résultat t
```

et comme spécification "t final = t initial trié dans l'ordre croissant" (nous avons développé cet exemple dans le § 5). Ce sous-algorithme peut lui-même utiliser une fonction dont l'interface serait :

```
fonction pmin(t, i)
tableau entier donnée t
entier donnée i
```

et la spécification : " valeur de retour = l'indice du plus petit élément entre t(i) et le dernier élément de t".

Les étapes à suivre, dans l'ordre, dans l'esprit de la conception descendante, sont :

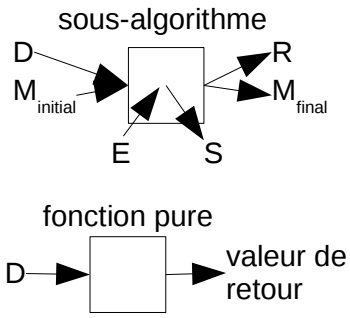


FIGURE 6 – Spécification d'une procédure. D : arguments données, E : entrées directes (lectures dans un fichier, dans le corps du sous-algorithme), R : arguments résultats, S : écritures directes (écritures dans un fichier, dans le corps du sous-algorithme), M : arguments données-résultats. La spécification d'un sous-algorithme indique comment (R, M_{final}, S) dépend de (D, M_{initial}, E). La spécification d'une fonction pure indique comment la valeur de retour dépend de D.

1. préciser les arguments ;
2. écrire la spécification ;
3. utiliser le sous-algorithme dans l'algorithme appelant ;
4. écrire le contenu du sous-algorithme.

Prenons encore un autre exemple de conception descendante avec sous-algorithme. Nous voulons dessiner la forme représentée sur la figure (7), à l'aide de simples caractères, par exemple des étoiles : *.

Idée. Une solution rigide et laborieuse consisterait à réaliser le dessin ligne par ligne. Une meilleure solution consiste à superposer des rectangles. Nous pouvons d'abord réaliser le dessin dans un tableau puis afficher le tableau ligne à ligne. Tableau utilisé :

```
caractère d(70, 70)
```

Algorithme principal. Après l'initialisation du tableau d, l'algorithme principal décompose juste le problème en deux sous-problèmes décrits en langage naturel :

```
d = ' ' ! mise à blanc du tableau
dessin des rectangles dans le tableau
écriture du tableau
```

Il faut maintenant formaliser les sous-problèmes, c'est-à-dire faire éventuellement apparaître des procédures, avec des spécifications.

Partie écriture du tableau. Elle peut s'écrire sans être encapsulée dans une procédure. En considérant que le premier indice de d est l'indice de ligne et que le "haut" du tableau correspond à l'indice de ligne 1 :

```
* écriture du tableau *
pour i = 1 à 70 faire
```

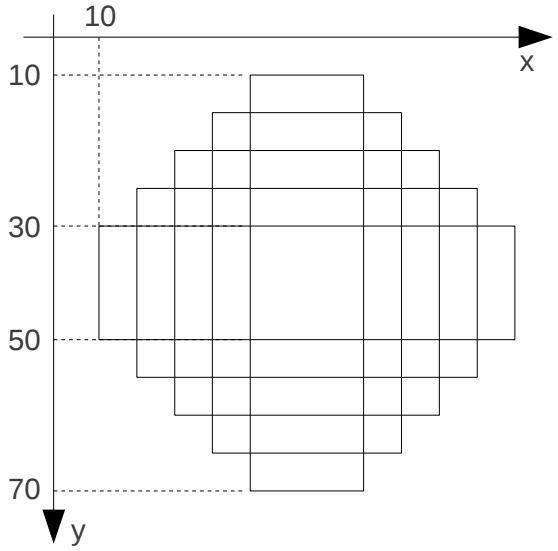


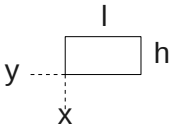
FIGURE 7 – Exemple de conception descendante : forme à dessiner.

écrire(d(i, :))
fin pour

Pour la partie dessin des rectangles, nous décidons de créer un sous-algorithme rectangle. Nous commençons par préciser son interface. Elle s'écrit :

```
sous-algorithme rectangle(x, y, l, h, t)
entier données x, y, l, h
caractère donnée-résultat t(70, 70)
```

Spécification du sous-algorithme rectangle : le sous-algorithme met dans le tableau `t` un rectangle dont le coin inférieur gauche est au couple d'indices (x, y),



la largeur est `l` et la hauteur est `h`.

Utilisation du sous-algorithme rectangle. Cf. algorithme (11).

Il ne reste qu'à écrire le contenu, ou encore la réalisation, du sous-algorithme rectangle.

Références

[1] Sandrine Blazy. *Sémantiques formelles*. Habilitation à diriger des recherches, Université d'Évry Val d'Essonne, 2008.

[2] Franck Ebel and Sébastien Rohaut. *Algorithmique - Techniques fondamentales de programmation – Exemples en Python*. Ressources informatiques. ENI, 2014.

Algorithme 11 Algorithme principal pour le dessin de rectangles.

```
d = ' '! mise à blanc du tableau
* dessin des rectangles *
x = 10
y = 50
l = 60
h = 20
pour i = 0 à 4 faire
   appel de rectangle(x + 5i, y + 5i, l - 10i, h + 10i, d)
fin pour
* écriture du tableau *
pour i = 1 à 70 faire
   écrire(d(i, :))
fin pour
```

[3] Christophe Haro. *Algorithmique – Raisonner pour concevoir.* ENI, 2nd edition, 2015.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–583, 1969.

Vous pouvez accéder aux références de l'éditeur ENI via Sorbonne universités.