Module I3

ALGORITHMIQUE ET PROGRAMMATION

Introduction à l'algorithmique

Camille Coti
camille.coti@iutv.univ-paris13.fr
IUT de Villetaneuse - Département R&T - 2011-2012

Table des matières

1 Introduction

Un algorithme est une *série d'instructions* qui doit être exécutée par un programme. À partir des données d'entrée, un algorithme doit donner un résultat. Une métaphore fréquemment utilisée consiste à définir un algorithme comme la recette qui, à partir des ingrédients, permet d'obtenir un gâteau.

Un programme informatique est la traduction d'un algorithme dans un langage de programmation. Les algorithmes constituent un formalisme permettant de décrire cette série d'instructions *indépendamment d'un langage de programmation en particulier*. On décrit généralement les algorithmes dans un langage compréhensible par un humain ou *pseudocode*.

Un algorithme est donc constitué des éléments suivants :
- Un début et une fin
- Un nom
- Des données d'entrée
- Des données de sortie, qui sont le résultat du calcul effectué par l'algorithme
- Un ensemble d'instructions exécutées par l'algorithme

La conception correcte d'un programme informatique commence par la *bonne compréhension du problème* et la *conception d'un algorithme qui fait bien ce que l'on attend*. Un excellent programmeur n'obtiendra pas un programme correct si il utilise un algorithme faux. L'étude du problème et sa transcription algorithmique sont donc fondamentales dans la pratique de la programmation. De même, le coût d'un calcul est défini principalement par son algorithme. La résolution efficace d'un problème dépend donc de l'*efficacité de l'algorithme* utilisé par le programme.

2 Les variables

Une *variable* correspond à l'emplacement mémoire d'une donnée. Elle porte un nom, ce qui permet de l'identifier facilement.

Une variable peut être *d'entrée* (une entrée de l'algorithme), *de sortie* (le résultat du calcul effectué par l'algorithme) ou *interne à l'algorithme* (ni d'entrée ni de sortie mais utilisée à l'intérieur de l'algorithme).

2.1 Types de variables

Les variables sont souvent *typées*, c'est-à-dire qu'on définit quel type de donnée pourra être contenu dans cette variable. Par exemple :
- Entier : tout nombre entier
- Booléen : vrai ou faux
- Caractère
- Chaîne de caractères
- Tableau...

Ainsi, on ne peut pas affecter un nombre réel à virgule (par exemple, 3,14159) dans une variable entière. Il est alors nécessaire d'effectuer une conversion : la variable entière contiendra la partie entière du nombre réel à virgule. Attention, les conversions ne sont pas toujours possibles.

2.2 Affectation d'une valeur à une variable

Pour écrire une valeur dans une variable, on dit que l'on *affecte* cette valeur à la variable. On le note de la façon suivante :

variable $\leftarrow$ *valeur*

Par exemple, si on définit une variable maVariable de type entier et que l'on lui affecte la valeur 42 dans l'algorithme suivant :

```
1 début
2 |  maVariable : Entier
3 |  maVariable ⟵ 42
4 fin
```

On commence par déclarer le type de la variable avant de lui affecter une valeur. En général, par soucis de lisibilité, les déclarations de variables sont intégralement faites en début d'algorithme quel que soit l'endroit où elles sont utilisées par la suite dans cet algorithme.

2.3 Les tableaux

Un tableau est un type particulier de variable. Il contient un *ensemble de variables* de même type, stockées de façon contigüe en mémoire. Par exemple, voici un tableau d'entiers :

3	5	2	1	12	5	2	9

La taille d'un tableau est fixe. Ils peuvent être de la dimension que l'on souhaite : unidimensionnels comme dans l'exemple ci-dessus, bidimensionnels (dans le cas d'une matrice, par exemple), etc. Dans ce cas, les données sont identifiées par leur indice dans chacune des dimensions.

On déclare un tableau en donnant sa taille et le type de données qu'il contiendra :
$monTab[10] : Tableau\ d'entiers$ déclare le tableau $monTab$ de taille 10 contenant des entiers.

Les données contenues dans un tableau sont numérotées, à partir de 0 ou de 1 selon la convention. La plupart des langages modernes commencent la numérotation à partir de 0 : c'est pourquoi dans ce cours nous suivrons cette convention. L'indice utilisé pour désigner les données d'un tableau est généralement de type entier. Ainsi, on désigne la huitième case du tableau $monTab$ en utilisant $monTab[7]$.

Dans le cas d'un tableau multidimensionnel, on le déclarera en donnant sa taille dans toutes les dimensions. Par exemple, pour un tableau bidimensionnel de taille 10×10 :
$maMatrice[10][10] : Tableau\ d'entiers$. On accède aux données en donnant leur rang dans chacune des dimensions, par exemple : $maMatrice[4][3]$ donnera l'élément situé en quatrième position sur la cinquième ligne [1].

3 Les structures de contrôle

3.1 Les tests

Un *test* évalue la valeur d'une expression booléenne et effectue une action si cette expression est vraie, une autre action si cette expression n'est pas vraie. On l'appelle également *structure conditionnelle*, car l'action réalisée est soumise à une condition.

L'algorithme suivant présente un exemple de test. Si la condition *condition* est validée, alors on effectue l'action *action1*. Sinon, on effectue l'action *action2*.

```
1 début
2 |  si condition alors
3 |  |  action1
4 |  sinon
5 |  |  action2
6 |  finsi
7 fin
```

1. L'ordre ligne / colonne dépend du langage d'implémentation : par exemple le C donne d'abord la ligne puis la colonne, tandis que le Fortran donne d'abord la colonne puis la ligne. Dans ce cours nous utiliserons la convention ligne-colonne (comme en C), plus répandue actuellement.

La deuxième partie est optionnelle : il est possible de ne pas en définir. Dans ce cas, l'algorithme ne fait rien dans le cas où la condition n'est pas réalisée.

L'algorithme suivant présente un deuxième exemple de test. Dans le cas où la variable $maVariable$ est inférieure à 5, on lui ajoute 1. Dans les autres cas, on ne fait rien.

```
1 début
2 |  si maVariable < 5 alors
3 |  |  maVariable ⟵ maVariable + 1
4 |  finsi
5 fin
```

Dans la condition, on peut tester :
- l'égalité entre deux variables : $var1 == var2$
- la non-égalité entre deux variables : $var1 != var2$
- une relation d'ordre entre deux variables : $var1 > 0$
- ou toute expression renvoyant $Vrai$ ou $Faux$

Ces conditions peuvent être combinées en suivant les règles de l'algèbre de Boole. Attention à mettre des parenthèses autour des expressions pour éviter les ambiguïtés, par exemple : $(var1 == var2) et (var1 > 0)$.

Les règles de syntaxe sont les suivantes :
- La *condition* est donnée entre les mot-clés **si** et **alors**
- L'action réalisée si la condition est vérifiée est donnée après le mot-clé **alors**
- Si on donne une action à réaliser si la condition n'est pas vérifiée, elle est introduite par le mot-clé **sinon**
- Le test est terminé par le mot-clé **finsi**
- Pour plus de lisibilité, le contenu de chaque bloc est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.2 Les boucles

Une boucle sert à *répéter une action*. On distingue deux sortes de boucles :
- La boucle **Pour**
- La boucle **Tant que**

La boucle *Pour* nécessite de connaître le nombre de répétitions à effectuer avant le début de la boucle. L'arrêt de la boucle *Tant que* est déterminé par une condition qui n'est pas intrinsèque à la boucle, mais souvent à son contenu : par exemple, une condition sur une variable modifiée à l'intérieur de la boucle.

3.2.1 Boucle *Pour*

La boucle *Pour* effectue une action en commençant par l'initialisation d'un compteur. À la fin de chaque répétition, on effectue une action sur ce compteur (par exemple, on l'augmente d'une certaine valeur). La boucle se termine lorsque la condition de bouclage n'est plus vraie.

Par exemple, examinons l'algorithme suivant. On a un compteur i initialisé à la valeur 0. L'action est réalisée pour i allant de 0 à 9 par un pas de 1, c'est-à-dire qu'à chaque répétition, la valeur du compteur i est augmentée de 1.

```
1 début
2 |  tab[10] : Tableau d'entiers
3 |  pour i ⟵ 0 à 9 pas 1 faire
4 |  |  tab[i] = 2 * i
5 |  finpour
6 fin
```

Dans le cas de l'algorithme ci-dessus, le compteur i est initialisé à 0 et la condition de la boucle est que i doit être inférieur ou égal à 9. Lorsque la valeur de i dépasse 9, la boucle se termine. Le pas vaut ici 1, mais il peut être différent de 1 (par exemple, pour aller de 2 en 2...), voire être négatif (pour aller à reculons).

Les règles de syntaxe sont les suivantes :
- Les conditions sur le compteur sont définies entre les mot-clés **pour** et **faire**
- L'initialisation du compteur est donnée comme une affectation ($i \longleftarrow 0$), sa valeur finale est donnée après le mot-clé **à**, le pas est donné après le mot-clé **pas**
- L'action à effectuer est donnée entre les mot-clés **faire** et **finpour**
- Pour plus de lisibilité, le contenu du bloc définissant l'action à effectuer est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.2.2 Boucle *Tant que ... Faire*

La boucle *Tant que* effectue une action tant qu'une condition donnée est vraie. Avant chaque répétition, on évalue la valeur de la condition. Si elle est vraie, alors on effectue l'action définie dans la boucle. Sinon, on sort de la boucle.

L'algorithme suivant est un exemple de boucle *tant que ... faire*. Nous définissons une variable entière *puissance* initialisée à la valeur 1. À chaque répétition, on multiplie par 2 la valeur de *puissance*. La boucle s'arrête lorsque la valeur de *puissance* atteint ou dépasse 100.

```
1 début
2 |  puissance : Entier
3 |  puissance ⟵ 1
4 |  tant que puissance < 100 faire
5 |  |  puissance ⟵ puissance * 2
6 |  fintq
7 fin
```

Les règles de syntaxe sont les suivantes :
- La condition de boucle est définie après le mot-clé **Tant que**
- L'action à effectuer est donnée entre les mot-clés **faire** et **fintq**
- Pour plus de lisibilité, le contenu du bloc définissant l'action à effectuer est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.2.3 Boucle *Faire ... Tant que*

À la différence de la boucle *tant que ... faire*, la boucle *faire ... tant que* évalue la valeur de la condition *après* avoir effectué l'action définie à l'intérieur de la boucle. On effectue donc l'action au moins une fois, et on décide ensuite si on la répète.

Par exemple, l'action à effectuer devant un panneau de stop correspond à une boucle *faire ... tant que*. L'algorithme ci-dessus décrit le comportement que doit suivre une voiture lorsqu'elle se trouve face à un panneau de stop. Il faut d'abord s'arrêter (ligne 4), puis regarder s'il y a des voitures. Tant que des voitures arrivent (ligne 5), on répète l'action dans la boucle (ligne 4). Lorsqu'il n'y a plus de voitures, le test situé ligne 5 est faux : on sort de la boucle (ligne 6).

```
1 début
2 |  vitesse : Entier
3 |  faire
4 |  |  vitesse ⟵ 0
5 |  tant que voitures_arrivent == Vrai ;
6 |  vitesse ⟵ 50
7 fin
```

3.2.4 Équivalence entre les boucles *Pour* et *Tant que*

On remarque qu'il est facile d'écrire une boucle *Pour* au moyen d'une boucle *Tant que*. On initialise le compteur avant d'entrer dans la boucle, la condition de la boucle *Tant que* est l'inverse de la condition d'arrêt de la boucle *Pour* (dans la boucle *Pour* il s'agit d'une condition d'arrêt, tandis que dans la boucle *Tant que* il s'agit d'une condition de continuation) et le compteur est modifié en ajoutant le pas à la fin de l'action à effectuer dans la boucle :

```
1 début
2 |  compteur : Entier
3 |  compteur ⟵ 0
4 |  tant que compteur < 10 faire
5 |  |  action
6 |  |  compteur ⟵ compteur + 1
7 |  fintq
8 fin
```

Cependant, une boucle *Pour* est souvent plus lisible et moins source d'erreurs qu'une boucle *Tant que*. Lorsque le nombre de répétitions est connu avant l'entrée dans la boucle et ne dépend pas du résultat de l'action à effectuer, on privilégie donc plutôt une boucle *Pour*. À l'inverse, la boucle *Tant que* est plutôt utilisée lorsque la condition d'arrêt de la boucle dépend du calcul effectué.

Cette équivalence illustre par ailleurs le fait qu'en algorithmique et en programmation en général, il y a souvent plusieurs façons de résoudre un problème. Les critères de choix entre les différentes solutions peuvent être :

- L'efficacité : par exemple, minimiser le nombre d'opérations à effectuer, ou l'espace mémoire utilisé ;
- La simplicité et la lisibilité : un algorithme simple présentera moins de risques d'erreur, s'il est lisible il sera plus facile de s'y replonger plus tard ou, pour une autre personne, de se familiariser avec le travail de quelqu'un d'autre.

3.3 Structuration par blocs et imbrication

Comme vu dans les règles de syntaxe des tests et des boucles, les structures de contrôle obéissent à certaines règles de présentation telles que l'indentation vers la droite et l'utilisation d'une ligne verticale pour bien marquer les actions à effectuer dans les différentes parties de ces structures. Les parties de ces structures sont appelées des *blocs*.

L'algorithme suivant présente deux blocs successifs.

```
1  début
2  |  tableau[10] : Tableau d'entiers
3  |  i : Entier
4  |  pour i ⟵ 0 à 9 pas 1 faire
5  |  |  tableau[i] ⟵ i × i
6  |  finpour
7  |  i ⟵ 0
8  |  tant que tableau[i] < 8 faire
9  |  |  i ⟵ i + 1
10 |  fintq
11 fin
```

La boucle *Pour* est un premier bloc. Ce bloc commence à la ligne 4 et termine à la ligne 6. Il commence avec le mot-clé **pour**, qui marque le début de la boucle, et termine avec le mot-clé **finpour**,

qui marque la fin de la boucle. La boucle *Tant que* est un deuxième bloc. Ce bloc commence à la ligne 8 et termine à la ligne 10. Il commence avec le mot-clé **tant que**, qui marque le début de la boucle, et termine avec le mot-clé **fintq**, qui marque la fin de la boucle. La ligne 5 est un bloc (constitué d'une seule ligne) : il s'agit du bloc d'instructions de la boucle *Pour*. De même, la ligne 9 contient le bloc d'instructions de la boucle *Tant que*.

Des blocs peuvent également être *imbriqués*. Un exemple de blocs imbriqués est illustré par l'algorithme ci-dessous.

```
1  début
2  |  matrice[10][10] : Tableau d'entiers
3  |  x : Entier
4  |  y : Entier
5  |  pour x ⟵ 0 à 9 pas 1 faire
6  |  |  pour y ⟵ 0 à 9 pas 1 faire
7  |  |  |  si x != y alors
8  |  |  |  |  matrice[x][y] ⟵ x × y
9  |  |  |  sinon
10 |  |  |  |  matrice[x][y] ⟵ 0
11 |  |  |  finsi
12 |  |  finpour
13 |  finpour
14 fin
```

Cet algorithme présente deux boucles *Pour* imbriquées. La première boucle répète la deuxième boucle en faisant varier son indice x. La deuxième boucle remplit la matrice en faisant varier son indice y. La première boucle parcourt les lignes de la matrice, la deuxième boucle parcourt les colonnes de chaque ligne. Enfin, un test est imbriqué dans la deuxième boucle et affecte une valeur à la case du tableau correspondant aux indices x et y en fonction de la valeur de x et y.

Le bloc de la boucle principale commence à la ligne 5 et termine à la ligne 13. Le bloc imbriqué dans la boucle principale commence à la ligne 6 et termine à la ligne 12. Le bloc imbriqué dans le bloc imbriqué commence à la ligne 7 et termine à la ligne 11. Enfin, ce bloc contient des blocs d'une ligne à la ligne 8 et à la ligne 10.

4 Programmation structurée

> "... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre."
>
> *Discours de la méthode (1637)*
> René Descartes

Il est possible de découper un problème en sous-problèmes indépendants les uns des autres. On suppose que chacun des sous-problèmes est résolu : on dispose alors d'une abstraction permettant de les combiner et de repousser le plus possible la résolution elle-même.

Un autre avantage de la programmation structurée est qu'elle permet de factoriser et de réutiliser un algorithme plusieurs fois à différents endroits du programme.

Pour cela, on découpe souvent un algorithme en *fonctions* et/ou en *procédures*. Le corps de l'algorithme appelle ces fonctions et procédures pour effectuer les actions et les calculs qui y sont définis. On leur passe souvent des *arguments* ou *paramètres*, qui sont les variables d'entrée de la fonction ou de la procédure.

4.1 Arguments

On distingue trois types d'arguments :
- Les arguments d'entrée : ils sont utilisés par la fonction ou la procédure mais leur valeur n'est pas modifiée par les actions effectuées à l'intérieur de la fonction ou de la procédure ;
- Les arguments de sortie : leur valeur n'est pas utilisée mais on leur affecte une valeur à l'intérieur de la fonction ou la procédure, et la variable utilisée comme paramètre par l'appelant est utilisée ensuite ;
- les arguments d'entrée/sortie : ils sont utilisés par la fonction ou la procédure et leur valeur est modifiée à l'intérieur de la fonction ou de la procédure.

4.2 Fonctions

À l'image d'une fonction mathématique $f : x \mapsto f(x)$, une fonction effectue un calcul et produit un résultat. On dit que ce résultat est *retourné* par la fonction. Il peut être utilisé dans un calcul ou affecté à une variable.

Par exemple, l'algorithme suivant appelle une fonction qui calcule le carré d'une variable passée en argument et retourne la valeur du carré. La fonction `carre` est définie en-dessous : elle prend comme argument un entier (qui est appelé *nombre*) et retourne un entier. Le calcul est effectué ligne 3.

```
1 début programme
2 |  nombreDepart : Entier
3 |  calcul : Entier
4 |  calcul ⟵ carre ( nombreDepart )
5 fin programme
```

```
1 début fonction carre( nombre: Entier ): Entier
2 |  resultat : Entier
3 |  resultat ⟵ nombre × nombre
4 |  retourner resultat
5 fin fonction
```

Le mot-clé *retourner* signifie que l'on sort de la fonction en renvoyant la valeur indiquée. Par exemple, la ligne 4 de la fonction `carre` contient l'instruction **retourner**. On sort donc de la fonction `carre` et elle retourne la valeur de la variable *resultat*, qui a été calculée dans la fonction.

La ligne 3 de l'algorithme principal appelle la fonction `carre`. On passe alors dans la fonction. La valeur retournée par la fonction est affectée à la variable *calcul* à la ligne 3 de l'algorithme principal.

La fonction ci-dessous retourne la plus petite puissance de 2 supérieure à 200. La condition de la boucle **tant que** est toujours vérifiée : la boucle est donc *infinie*. Cependant, il y a un test à l'intérieur de la boucle: si la condition est réalisée (si le nombre dépasse 200), la ligne 6 est exécutée. On appelle alors l'instruction **retourner** : la fonction retourne la valeur de la variable *petitepuissance*. On sort alors de la fonction.

```
1 début fonction puissance( ): Entier
2 |  petitepuissance : Entier
3 |  tant que Vrai faire
4 |  |  petitepuissance ⟵ petitepuissance × 2
5 |  |  if petitepuissance > 200 then
6 |  |  |  retourner petitepuissance
7 |  |  endif
8 |  fintq
9 fin fonction
```

Les règles de syntaxe sont les suivantes :
- Une fonction commence par les mot-clés **début fonction** et se termine par le mot-clé **fin fonction**
- On donne ensuite le *nom* de la fonction et les *paramètres* qui doivent lui être passés, ainsi que leur type
- Pour les paramètres de sortie ou d'entrée/sortie, on précise **sortie** ou **entrée/sortie**. Pour les paramètres d'entrée, on n'a pas à le préciser
- Le *type retourné* par la fonction est précisé à la fin de la ligne de définition de la fonction, après deux points (**:**)
- Une fonction *retourne obligatoirement quelque chose*. On ne peut sortir d'une fonction que par le mot-clé **retourner** suivi du nom de la variable retournée, ou d'une valeur

4.3 Procédures

À l'inverse d'une fonction, une procédure ne retourne rien. Elle est utilisée pour effectuer une action qui ne renvoie pas de résultat (par exemple, rafraîchir un affichage) ou pour modifier la valeur de variables qui lui sont passées en paramètres.

On peut réécrire la fonction *puissance*() définie ci-dessous au moyen d'une procédure en mettant le résultat du calcul dans une variable de sortie passée en paramètre de la procédure. On précise alors dans la déclaration de la procédure que l'argument passé **petitepuissance** est de type **Entier** et qu'il s'agit d'un argument de **sortie** :

```
1 début procédure puissance( petitepuissance: Entier Sortie )
2 |  petitepuissance ⟵ 1
3 |  tant que Vrai faire
4 |  |  petitepuissance ⟵ petitepuissance × 2
5 |  |  if petitepuissance > 200 then
6 |  |  |  retourner
7 |  |  endif
8 |  fintq
9 fin procédure
```

De même que pour une fonction, on peut sortir d'une procédure à tout moment en appelant le mot-clé **retourner**. Cependant, on l'appelle seul : on ne précise pas de valeur à retourner, car une procédure ne retourne *rien*.

Les règles de syntaxe sont les suivantes :
- Une procédure commence par les mot-clés **début procédure** et se termine par le mot-clé **fin procédure**
- On donne ensuite le *nom* de la procédure et les *paramètres* qui doivent lui être passés, ainsi que leur type
- Pour les paramètres de sortie ou d'entrée/sortie, on précise **sortie** ou **entrée/sortie**. Pour les paramètres d'entrée, on n'a pas à le préciser
- On sort de la procédure soit en arrivant à la fin, soit par le mot-clé **retourner**

4.4 Approche descendante

En découpant un problème en sous-problèmes, qui sont eux-mêmes découpés en sous-problèmes, on en vient à concevoir un algorithme qui fait appel à des solutions à ces sous-problèmes. On définit alors ces solutions dans des fonctions ou dans des procédures, qui font elles-mêmes appel à d'autres fonctions et procédures.

Par exemple, considérons un problème de géométrie. On veut calculer la longueur de l'hypoténuse d'un triangle rectangle, en connaissant la longueur de ses deux autres côtés.

L'approche naïve et inefficace conduit à l'algorithme suivant :

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2  |  /* variables internes */
3  |  carre1 : Entier
4  |  carre2 : Entier
5  |  hypotenusecarre : Entier
6  |  hypotenuse : Entier
7  |  /* on calcule la somme des carrés des côtés */
8  |  carre1 ⟵ cote1 × cote1
9  |  carre2 ⟵ cote2 × cote2
10 |  hypotenusecarre ⟵ carre1 + carre2
11 |  /* on prend la racine carrée de la somme et on retourne le résultat */
12 |  hypotenuse ⟵ √hypotenusecarre
13 |  retourner hypotenuse
14 fin fonction
```

Tous les calculs sont définis à la suite les uns des autres. Cependant, le théorème de Pythagore qui calcule de la longueur de l'hypoténuse d'un triangle rectangle se définit comme suit : *"le carré de l'hypoténuse d'un triangle rectangle est égal à la somme des carrés des longueurs des deux autres côtés"*.

On découpe alors notre algorithme comme suit :

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2  |  /* variables internes */
3  |  carre1 : Entier
4  |  carre2 : Entier
5  |  hypotenusecarre : Entier
6  |  hypotenuse : Entier
7  |  /* on calcule la somme des carrés des côtés */
8  |  carre1 ⟵ carré( cote1 )
9  |  carre2 ⟵ carré( cote2 )
10 |  hypotenusecarre ⟵ carre1 + carre2
11 |  /* on prend la racine carrée de la somme et on retourne le résultat */
12 |  hypotenuse ⟵ racine( hypotenusecarre )
13 |  retourner hypotenuse
14 fin fonction
```

Cet algorithme présente l'avantage de ne pas demander de savoir faire les calculs eux-mêmes : *on retarde le plus possible le moment d'effectuer les calculs*. Par exemple, un calcul de racine carrée est compliqué à mettre en place sur une machine. On préfère appeler une fonction qui l'effectue une fois pour toutes, plutôt que de devoir penser tout de suite à la façon dont on va faire ce calcul. À ce moment de la conception de l'algorithme, on n'a pas à savoir comment on va la calculer. On se donne un *niveau d'abstraction* qui permet de penser à la structure générale de l'algorithme, puis de *descendre dans les détails* de la résolution des sous-problèmes.

L'algorithme ci-dessus fait appel à deux fonctions : **carré()** et **racine()**. On donne leur algorithme séparément :

```
1  début fonction carré( nombre: Entier ): Entier
2  |  moncarre : Entier
3  |  moncarre ⟵ nombre × nombre
4  |  retourner moncarre
5  fin fonction
```

```
1 début fonction racine( nombre: Entier ): Entier
2 |   maracine : reel
3 |   maracine ⟵ √nombre
4 |   retourner maracine
5 fin fonction
```

De cette façon, on se concentre spécifiquement et de manière isolée sur les calculs eux-mêmes. Lorsqu'on conçoit un algorithme, il vaut mieux écrire de nombreuses fonctions et procédures courtes, plutôt qu'une seule et longue fonction.