Algorithmique pour le BTS

Alexandre Meslé

5 octobre 2012

Table des matières

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Le principe

Exemple 1 **- La surprise du chef**

Considérons la suite d'instructions suivante :

1. Faites chauffer de l'eau dans une casserole
2. Une fois que l'eau boue, placez les pâtes dans l'eau
3. Attendez dix minutes
4. Versez le tout dans un écumoire
5. Vos pâtes sont prêtes.

Vous l'aurez deviné, il s'agit des grandes lignes de la recette permettant de préparer des pâtes (si vous les voulez *al dente*, attendez un petit peu moins de 10 minutes). Cette recette ne vous expose pas le détail des réactions chimiques qui font que les pâtes cuisent en dix minutes, ni pourquoi il faut les égoutter. Il s'agit seulement d'une suite d'instructions devant être exécutées à la lettre. Si vous ne les suivez pas, vous prenez le risque que le résultat ne soit pas celui que vous attendez. Si vous décidez de *suivre une recette*, vous décidez de vous conformer aux instructions sans poser de questions. Par opposition, vous pouvez décider de créer vous-même une recette, cela vous demandera davantage de réflexion, et vous serez amené à élaborer d'une suite d'instructions qui vous permettra de retrouver le même résultat.

Exemple 2 **- Ikéa**

Considérons comme autre exemple une notice de montage. Elle est composée d'un ensemble d'étapes à respecter scrupuleusement. Il ne vous est pas demandé de vous interroger sur la validité de ces instructions, on vous demande juste de les suivre. Si vous vous conformez aux indications de la notice, vous parviendrez à monter votre bibliothèque Louis XVI. Si vous ne suivez pas la notice de montage, il vous restera probablement à la fin une pièce entre les mains, et vous aurez beau chercher où la placer, aucun endroit ne conviendra. Vous aurez alors deux solutions : soit vous démontez tout pour reprendre le montage depuis le début, soit vous placez cette pièce dans l'assiette qui est dans l'entrée en attendant le prochain déménagement, et en sachant que la prochaine fois, vous suivrez la notice... Cet exemple est analogue au premier, vous avez entre vos mains une suite d'instructions à exécuter, si vous les suivez, vous obtenez le résultat attendu, sinon, il y a de très fortes chances que n'obteniez pas le résultat escompté. De la même façon, le but n'est pas que vous vous demandiez pourquoi ou comment ça marche, la notice est faite pour que vous n'ayez pas à vous poser ce type de question. Si jamais vous décidez de créer un meuble (par exemple, une bibliothèque Nicolas Premier) à monter soi-même, il vous faudra fournir avec une notice de montage. C'est-à-dire une succession d'étapes que l'acquéreur de ce meuble devra suivre à la lettre.

Définition

On conclut de de la façon suivante : nous avons vu qu'il existait des séquences d'instructions faites pour être exécutée à la lettre et sans se poser de questions, c'est le principe de l'algorithme. Nous retiendrons donc que **Un algorithme est une séquence d'instructions exécutée de façon logique mais non intelligente.**

- **Logique** parce que la personne (ou la machine) qui exécute les instructions est capable de comprendre et exécuter sans erreur chacune d'elles.
- **Non intelligente** parce que la personne qui exécute l'algorithme n'est pas supposée apte à comprendre pourquoi la succession d'étapes décrite par l'algorithme donne bien un résultat correct.

Utilisation en informatique

Les premiers algorithmes remontent à l'antiquité. Par exemple l'algorithme de calcul du plus grand commun diviseur de deux nombres, appelé maintenant "algorithme d'Euclide". Il s'agissait en général de méthodes de calcul semblables à celle que vous utilisez depuis le cours élémentaire pour additionner deux nombres à plusieurs chiffres. Notez qu'à l'époque, on vous demandait juste d'appliquer la méthode sans vous tromper, on ne vous a pas expliqué pourquoi cette méthode marchait à tous les coups. Le principe était donc le même, vous n'aviez pas le niveau en mathématiques pour comprendre pourquoi la succession d'étapes qu'on vous donnait était valide, mais vous étiez capable d'exécuter chaque étape de la méthode. Avant même d'avoir dix ans, vous connaissiez donc déjà des algorithmes.

Le mot algorithme prend étymologiquement ses racines dans le nom d'un mathématicien arabe du moyen âge : Al-Kawarizmi. Les algorithmes sont extrêmement puissants : en concevant un algorithme, vous pouvez décomposer un calcul compliqué en une succession d'étapes compréhensibles, c'est de cette façon qu'on vous a fait faire des divisions (opération compliquée) en cours moyen, à un âge où votre niveau en mathématiques ne vous permettait pas de comprendre le fonctionnement d'une division.

Contrairement aux mythe Matrix-Terminator-L'Odyssée de l'espace-I, Robot-R2D2 (et j'en passe) un ordinateur fonctionne de la même façon qu'un monteur de bibliothèque (rien à voir avec l'alpinisme) ou votre cuisinier célibataire (il y a quand même des exceptions), il est idiot et pour chaque chose que vous lui demanderez, il faudra lui dire comment faire. Vous aller donc lui donner des successions d'instructions à suivre, et lui les respectera à la lettre et sans jamais se tromper. Une suite d'instructions de la sorte est fournie à l'ordinateur sous la forme de **programme**. Pour coder un programme, on utilise un **langage de programmation**, par exemple C, Java, Pascal, VB... Selon le langage utilisé, une même instruction se code différemment, nous ferons donc dans ce cours abstraction du langage utilisé. Nous nous intéresserons uniquement à la façon de combiner des instructions pour former des programmes, indépendamment des langages de programmation. Le but de ce cours est donc de vous apprendre à créer des algorithmes, c'est-à-dire à décomposer des calculs compliqués en successions d'étapes simples.

1.1.2 Variables

Une algorithme se présente comme une liste d'instructions, elles sont écrites les unes au dessus des autres et elles sont exécutées **dans l'ordre**, lorsque l'on conçoit une algorithme, il faut toujours avoir en tête le fait que l'ordre des instructions est très important. Le premier concept nécessaire pour concevoir un algorithme est celui de variable. Une **variable** est un emplacement dans la mémoire où est stockée une valeur. Une variable porte un **nom**, ce nom est laissé au choix du concepteur de l'algorithme, il doit commencer par une lettre et ne pas comporter d'espace. On se sert du nom d'une variable pour lire la valeur qui s'y trouve ou bien pour la modifier, une

variable ne peut contenir qu'une seule valeur à la fois. Une valeur est `numérique` s'il s'agit d'un nombre, ou bien `alphanumérique` s'il s'agit d'une succession de symboles, par exemple des mots. Toute variable a un **type** : si une variable est de type numérique, il n'est possible d'y mettre que des valeurs numériques, si une variable est de type alphanumérique, il n'est possible d'y stocker que des valeurs alphanumériques.

L'**affectation** est une opération permettant de modifier la valeur d'une variable. La syntaxe de l'affectation est la suivante :

```
nomvariable ⟵ valeur
```

`<nomvariable>` est le nom de la variable dont on souhaite modifier la valeur, `<valeur>` est la valeur que l'on veut placer dans la variable. Notez bien que cette valeur doit être de même type que la variable. Par exemple,

$$A \longleftarrow 5$$

place la valeur 5 dans la variable A. Si A contenait préalablement une valeur, celle-ci est écrasée. Il est possible d'affecter à une variable le résultat d'une opération arithmétique.

$$A \longleftarrow 5 + 2$$

On peut aussi affecter à une variable la valeur d'une autre variable

$$\begin{array}{l} A \longleftarrow B \\ A \longleftarrow B + 2 \end{array}$$

La première instruction lit la valeur de B et la recopie dans A. la deuxième instruction, donc exécutée après la première, lit la valeur de B, lui additionne 2, et recopie le résultat dans A. Le fait que l'on affecte à A la valeur de B ne signifie pas que ces deux variables auront dorénavant la même valeur. Cela signifie que la valeur contenue dans B est écrasée par la valeur que contient A **au moment de l'affectation**. Si la par la suite, la valeur de A est modifiée, alors la valeur de B restera inchangée. Il est possible de faire figurer une variable simultanément à gauche et à droite d'une affectation :

$$A \longleftarrow A + 1$$

Cette instruction augmente de 1 la valeur contenue dans A, cela s'appelle une **incrémentation**.

Exemple

Quelles sont les valeurs des variables après l'exécution des instructions suivantes ?

$$\begin{array}{l} A \longleftarrow 1 \\ B \longleftarrow 2 \\ C \longleftarrow 3 \\ D \longleftarrow A \\ A \longleftarrow C + 1 \\ B \longleftarrow D + C \\ C \longleftarrow D + 2 * A \end{array}$$

Construisons un tableau nous montrant les valeurs des variables au fil des affectations :

instruction	A	B	C	D
début	n.i	n.i	n.i	n.i
$A \longleftarrow 1$	1	n.i	n.i	n.i
$B \longleftarrow 2$	1	2	n.i	n.i
$C \longleftarrow 3$	1	2	3	n.i
$D \longleftarrow A$	1	2	3	1
$A \longleftarrow C + 1$	4	2	3	1
$B \longleftarrow D + C$	4	4	3	1
$C \longleftarrow D + 2 * A$	4	4	9	1

`n.i` signifie ici **non initialisée**. Une variable est non initialisée si aucune valeur ne lui a été explicitement affectée. $A \longleftarrow 1$ modifie la valeur contenue dans la variable A. A ce moment-là de l'exécution, les valeurs des autres variables sont inchangées. $B \longleftarrow 2$ modifie la valeur de B, les deux variables A et B sont maintenant initialisées. $C \longleftarrow 3$ et $D \longleftarrow A$ initialisent les deux variables C et D, maintenant toutes les variables sont initialisées. Vous remarquerez que D a été initialisée avec la valeur de A, comme A est une variable initialisée, cela a un sens. Par contre, si l'on avait affecté à D le contenu d'une variable non initialisée, nous aurions exécuté une instruction qui n'a pas de sens. Vous noterez donc qu'il est **interdit de faire figurer du coté droit d'une affectation une variable non initialisée**. Vous remarquez que l'instruction $D \longleftarrow A$ affecte à D la valeur de A, et que l'affectation $A \longleftarrow C + 1$ n'a pas de conséquence sur la variable D. Les deux variables A et D correspondent à **deux emplacements distincts de la mémoire**, modifier l'une n'affecte pas l'autre.

1.1.3 Littéraux

Un **littéral** est la représentation de la valeur d'une variable. Il s'agit de la façon dont on écrit les valeurs des variables directement dans l'algorithme.

- numérique : $1, 2, 0, -4, \ldots$
- alphanumérique : "toto", "toto01", "04", ...

Attention, 01 et 1 représentent les mêmes valeurs numériques, alors que "01" et "1" sont des valeurs alphanumériques distinctes. Nous avons vu dans l'exemple précédent des littéraux de type numérique, dans la section suivante il y a un exemple d'utilisation d'un variable de type alphanumérique.

1.1.4 Convention d'écriture

Afin que nous soyons certains de bien nous comprendre lorsque nous rédigeons des algorithmes, définissons de façon précise des règles d'écriture. Un algorithme s'écrit en trois parties :

1. Le **titre**, tout algorithme porte un titre. Choisissez un titre qui permet de comprendre ce que fait l'algorithme.
2. La **déclaration de variables**, vous préciserez dans cette partie quels noms vous avez décidé de donner à vos variables et de quel type est chacune d'elle.
3. Les **instructions**, aussi appelé le corps de l'algorithme, cette partie contient notre succession d'instructions.

Par exemple,

Algorithme : Meanless
Variables :
numériques : A, B, C
alphanumériques : t
DEBUT
$A \longleftarrow 1$
$B \longleftarrow A + 1$
$C \longleftarrow A$
$A \longleftarrow A + 1$
$t \longleftarrow$"this algorithm is dumb"
FIN

La lisibilité des algorithmes est un critère de qualité prépondérant. Un algorithme correct mais indéchiffrable est aussi efficace qu'un algorithme faux. Donc c'est un algorithme faux. Vous devrez par conséquent soigner particulièrement vos algorithmes, ils doivent être faciles à lire, et rédigés de sorte qu'un lecteur soit non seulement capable de l'exécuter, mais aussi capable de le comprendre rapidement.

1.1.5 Entrées-sorties

De nombreux algorithmes ont pour but de communiquer avec un utilisateur, cela se fait dans les deux sens, les sorties sont des envois de messages à l'utilisateur, les entrées sont des informations fournies par l'utilisateur.

Saisie

Il est possible de demander à un utilisateur du programme de **saisir** une valeur. La syntaxe de la saisie est la suivante :

Saisir < *nomvariable* >

La saisie interrompt le programme jusqu'à ce que l'utilisateur ait saisi une valeur au clavier. Une fois cela fait, la valeur saisie est placée dans la variable *nomvariable*. Il est possible de saisir plusieurs variables à la suite,

Saisir *A, B, C*

place trois valeurs saisies par l'utilisateur dans les variables `A`, `B` et `C`.

Affichage

Pour afficher un message à destination de l'utilisateur, on se sert de la commande

Afficher < *message* >

Cette instruction affiche le `<message>` à l'utilisateur. Par exemple,

Afficher *"Hello World"*

affiche "Hello World" (les guillemets sont très importantes !). Il est aussi possible d'afficher le contenu d'une variable,

Afficher *A*

affiche l'écran le contenu de la variable `A`. On peut entremêler les messages et les valeurs des variables. Par exemple, les instructions

Afficher *"La valeur de la variable A est"*;
Afficher *A*;

ont le même effet que l'instruction

Afficher *"La valeur de la variable A est", A*

Lorsque l'on combine messages et variables dans les instruction d'affichage, on les sépare par des virgules. Notez bien que ce qui est délimité par des guillemets est affiché tel quel, alors tout ce qui n'est pas délimité par des guillemets est considéré comme des variables.

Exemple

Cet algorithme demande à l'utilisateur de saisir une valeur numérique, ensuite il affiche la valeur saisie puis la même valeur incrémentée de 1.

Algorithme : Affichage incrément
Variables :
numériques : a, b
DEBUT
  Afficher *"Saisissez une valeur numérique"*
  Saisir a
  $b \longleftarrow a + 1$
  Afficher *"Vous avez saisi la valeur ", a, "."*
  Afficher *a, "+ 1 = ", b*
FIN

1.1.6 Types numériques et alphanumériques

Types numériques

Nous mettrons maintenant de côté le type `numérique` pour privilégier les deux types suivants :
- `entier`, que nous utiliserons pour représenter des nombres entiers, éventuellement négatifs.
- `réel`, que nous utiliserons pour représenter des nombres réels.

Types alphanumériques

Nous affinerons aussi les types alphanumériques en leur substituant les deux types suivants :
- `caractère`, qui est un type permettant de représenter un symbole, et un seul.
- `chaîne`, que nous utiliserons lorsque l'on voudra représenter zéro, un ou plusieurs caractères.

Les littéraux de type caractères seront délimités par des simples quotes (apostrophes) et les chaînes de caractères seront délimitées par des double-quotes (guillemets).

1.2 Traitements conditionnels

On appelle traitement conditionnel un bloc d'instructions dont l'exécution est soumise à la vérification d'un test.

1.2.1 `SI ... ALORS`

La syntaxe d'un traitement conditionnel est la suivante :

Si < *condition* > **alors**
| < *instructions* >
fin si

Les `<instructions>` ne sont exécutées que si `<condition>` est vérifiée. Par exemple,

Si $A = 0$ **alors**
| **Afficher** *"La valeur de la variable A est nulle."*
fin si

Si la variable A, au moment du test, a une valeur nulle, alors l'instruction `Afficher "La valeur de la variable A est nulle."` est exécutée, sinon, elle est ignorée.

Conditions

Une condition peut être tout type de test. Par exemple,

$A = 2$
$A = B$
$B <> 7$
$2 > 7$

La condition `A = 2` est vérifiée si la valeur contenue dans A est 2. `A = B` est vérifiée si les valeurs contenues dans A et dans B sont les mêmes. `B <> 7` est vérifiée si B contient une valeur différente de 7. `2 > 7` est vérifiée si 2 est supérieur à 7, donc jamais, cette condition est donc fausse et ne dépend pas des valeurs des variables.

Si étendu

Le traitement conditionnel peut être étendue de la sorte :

Si < *condition* > **alors**
| < *instructions* >
sinon
| < *autresinstructions* >
fin si

Si `<condition>` est vérifiée, les `<instructions>` sont exécutées. Dans le cas contraire, donc si `<condition>` n'est pas vérifiée, alors ce sont les `<autresinstructions>` qui sont exécutées. Par exemple,

Algorithme : Valeurs Distinctes
Variables :
entiers : a, b
DEBUT
  Afficher *"Saisissez deux valeurs entières"*
  Saisir a, b
  Si $a = b$ **alors**
    Afficher *"Vous avez saisi deux fois la même valeur, à savoir ", a, "."*
  sinon
    Afficher *"Vous avez saisi deux valeurs différentes, ", a, " et ", b, "."*
  fin si
FIN

Dans l'exemple ci-dessus, la condition `a = b` est évaluée. Si à ce moment-là les variables a et b contiennent la même valeur, alors la condition `a = b` sera vérifiée. Dans ce cas, l'instruction `Afficher "Vous avez saisi deux fois la même valeur, à savoir ", a, "."` sera exécutée. Si la condition `a = b` n'est pas vérifiée, donc si les variables a et b ne contiennent pas la même valeur au moment de l'évaluation de la condition, c'est alors l'instruction `Afficher "Vous avez saisi deux valeurs différentes, ", a, " et ", b, "."` qui sera exécutée.

Imbrication

Il est possible d'imbriquer les `SI` à volonté :

Si $a < 0$ **alors**
  Si $b < 0$ **alors**
    Afficher *"a et b sont négatifs"*
  sinon
    Afficher *"a est négatif, b est positif"*
  fin si
sinon
  Si $b < 0$ **alors**
    Afficher *"b est négatif, a est positif"*
  sinon
    Afficher *"a et b sont positifs"*
  fin si
fin si

Si par exemple a et b sont tous deux positifs, alors aucun des deux tests ne sera vérifié, et c'est donc le `sinon` du `sinon` qui sera exécuté, à savoir `Afficher "a et b sont positifs"`.

Connecteurs logiques

Les connecteurs logiques permettent de d'évaluer des conditions plus complexes. Deux sont disponibles :

- **et** : la condition `<condition1> et <condition2>` est vérifiée si les deux conditions `<condition1>` et `<condition2>` sont vérifiées simultanément.
- **ou** : la condition `<condition1> ou <condition2>` est vérifié si au moins une des deux conditions `<condition1>` et `<condition2>` est vérifiée.

Par exemple, écrivons un algorithme qui demande à l'utilisateur de saisir deux valeurs, et qui lui dit si le produit de ces deux valeurs est positif ou négatif sans en calculer le produit.

Algorithme : Signe du produit
Variables :
entiers : a, b
DEBUT
  Afficher *"Saisissez deux valeurs entières"*
  Saisir *a, b*
  Afficher *"Le produit de ", a, " par ", b, " est "*
  Si *($a \leq 0$ et $b \leq 0$) ou ($a \geq 0$ et $b \geq 0$)* **alors**
    Afficher *"positif ou nul"*
  sinon
    Afficher *"négatif"*
  fin si
FIN

L'instruction `Afficher "positif ou nul"` sera exécutée si au moins une des deux conditions suivantes est vérifiée :
- $a \leq 0$ et $b \leq 0$
- $a \geq 0$ et $b \geq 0$

1.2.2 Suivant cas

Lorsque que l'on souhaite conditionner l'exécution de plusieurs ensembles d'instructions par la valeur que prend une variable, plutôt que de faire des imbrications de `si` à outrance, on préférera la forme suivante :

Suivant $<$ *variable* $>$ **faire**
  cas $<$ *valeur*$_1$ $>$ *:* $<$ *instructions*$_1$ $>$
  cas $<$ *valeur*$_2$ $>$ *:* $<$ *instructions*$_2$ $>$
  ...
  cas $<$ *valeur*$_n$ $>$ *:* $<$ *instructions*$_n$ $>$
  autres cas $<$ *instructions* $>$
fin

Selon la valeur que prend la variable `<variable>`, le bloc d'instructions à exécuter est sélectionné. Par exemple, si la valeur de `<variable>` est `<valeur_1>`, alors le bloc `<instructions_1>` est exécuté. Le bloc `<autres cas>` est exécuté si la valeur de `<variable>` ne correspond à aucune des valeurs énumérées.

Exemple

Écrivons un algorithme demandant à l'utilisateur le jour de la semaine. Affichons ensuite le jour correspondant au lendemain.

```
Algorithme : Lendemain
Variables :
entier : erreur
chaînes : jour, lendemain
DEBUT
    Afficher "Saisissez un jour de la semaine"
    Saisir jour
    erreur ⟵ 0
    Suivant jour faire
        cas "lundi" : lendemain ⟵ "mardi"
        cas "mardi" : lendemain ⟵ "mercredi"
        cas "mercredi" : lendemain ⟵ "jeudi"
        cas "jeudi" : lendemain ⟵ "vendredi"
        cas "vendredi" : lendemain ⟵ "samedi"
        cas "samedi" : lendemain ⟵ "dimanche"
        cas "dimanche" : lendemain ⟵ "lundi"
        autres cas erreur ⟵ 1
    fin
    Si erreur = 1 alors
        Afficher "Erreur de saisie"
    sinon
        Afficher "Le lendemain du ", jour, " est ", lendemain, "."
    fin si
FIN
```

Vous remarquez que si l'on avait voulu écrire le même algorithme avec des `Si`, des imbrications nombreuses et peu élégantes auraient été nécessaires.

1.2.3 Variables Booléennes

Le type **Booléen** sera utilisé pour des variables ne pouvant prendre que deux valeurs : **vrai** et **faux**. Il serait tout à fait judicieux de se demander quel est l'intérêt d'un tel type ? Observons tout d'abord cet exemple :

Algorithme : Lendemain

Variables :
booléen : erreur
chaînes : jour, lendemain
DEBUT
  Afficher *"Saisissez un jour de la semaine"*
  Saisir *jour*
  erreur ⟵ *faux*
  Suivant *jour* **faire**
    cas *"lundi"* : lendemain ⟵ "mardi"
    cas *"mardi"* : lendemain ⟵ "mercredi"
    cas *"mercredi"* : lendemain ⟵ "jeudi"
    cas *"jeudi"* : lendemain ⟵ "vendredi"
    cas *"vendredi"* : lendemain ⟵ "samedi"
    cas *"samedi"* : lendemain ⟵ "dimanche"
    cas *"dimanche"* : lendemain ⟵ "lundi"
    autres cas erreur ⟵ *vrai*
  fin
  Si *erreur* **alors**
    Afficher *"Erreur de saisie"*
  sinon
    Afficher *"Le lendemain du ", jour, " est ", lendemain, "."*
  fin si
FIN

Le test opéré lors du `Si ... alors` est effectué sur une expression pouvant prendre les valeurs `vrai` ou `faux`. Par conséquent, un test peut aussi s'opérer sur une variable booléenne. Si la variable `erreur` contient la valeur `vrai`, alors le test est vérifié et l'instruction se trouvant sous la portée du `alors` est exécutée. Si par contre elle contient la valeur `faux`, alors le test n'est pas vérifié et l'instruction de trouvant sous la portée du `sinon` est exécutée.

1.3 Boucles

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou encore un **passage** dans la boucle. Il existe trois types de boucle :

- Tant que
- Répéter ... jusqu'à
- Pour

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

1.3.1 Tant que

La syntaxe d'une boucle `Tant que` est la suivante.

Tant que < *condition* >
| < *instructions* >
fin tant que

La condition est évaluée **avant** chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Un fois que la condition n'est plus vérifiée, l'exécution se poursuit après le `fin tant que`. Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant,

Algorithme : 1 à 5 Tant que
Variables :
entier : i
DEBUT
| $i \longleftarrow 1$
| **Tant que** $i \leq 5$
| | **Afficher** i
| | $i \longleftarrow i + 1$
| **fin tant que**
FIN

Cet algorithme **initialise** i **à** 1 et tant que la valeur de i n'excède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante** ! Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre algorithme ne se comportera pas du tout comme prévu.

1.3.2 Répéter ... jusqu'à

répéter
| < *instructions* >
jusqu'à < *condition* > ;

La fonctionnement est analogue à celui de la boucle `tant que` à quelques détails près :

- la condition est évaluée **après** chaque passage dans la boucle.
- On exécute le corps de la boucle jusqu'à ce que la condition soit vérifiée, donc tant que la condition est fausse.

Une boucle `Répéter ... jusqu'à` est donc exécutée donc **au moins une fois**. Reprenons l'exemple précédent avec une boucle `Répéter ... jusqu'à` :

Algorithme : 1 à 5 Répéter ... jusqu'à
Variables :
entier : i
DEBUT
  $i \longleftarrow 1$
  répéter
    Afficher i
    $i \longleftarrow i + 1$
  jusqu'à $i > 5$
FIN

De la même façon que pour la boucle `Tant que`, le compteur est initialisé avant le premier passage dans la boucle. Par contre, la condition de sortie de la boucle n'est pas la même, on ne sort de la boucle qu'un fois que la valeur 5 a été affichée. Or, i est incrémentée après l'affichage, par conséquent i aura la valeur 6 à la fin de l'itération pendant laquelle la valeur 5 aura été affichée. C'est pour cela qu'on ne sort de la boucle qu'une fois que i a dépassé strictement la valeur 5. Un des usages les plus courant de la boucle `Répéter ... jusqu'à` est le contrôle de saisie :

répéter
  Afficher *"Saisir un nombre strictement positif"*
  Saisir i
  Si $i \leq 0$ **alors**
    Afficher *"J'ai dit STRICTEMENT POSITIF !"*
  fin si
jusqu'à $i > 0$

1.3.3 Pour

Pour < *variable* > *allant de* < *premierevaleur* > *à* < *dernierevaleur* > *[par pas de* < *pas* >*]*
  < *instructions* >
fin pour

La boucle Pour fait varier la valeur du compteur `<variable>` entre `<première valeur` et `<dernière valeur>`. Le `<pas>` est optionnel et permet de préciser la variation du compteur entre chaque itération, le pas par défaut est 1 et correspond donc à une incrémentation. Toute boucle `pour` peut être réécrite avec une boucle `tant que`. On réécrit de la façon suivante :

< *variable* >⟵< *premierevaleur* >
Tant que < *variable* >< >< *dernierevaleur* > + < *pas* >
  < *instructions* >
  < *variable* >⟵< *variable* > + < *pas* >
fin tant que

La boucle `pour` initialise le compteur `<variable>` à la `<première valeur>`, et tant que la dernière valeur n'a pas été atteinte, les `<instructions>` sont exécutées et le compteur incrémenté de `<pas>` si le pas est positif, et décrémenté de `|<pas>|` si le pas est négatif.

Algorithme : 1 à 5 Pour
Variables :
entier : i
DEBUT
  Pour i *allant de* 1 *à* 5
    Afficher i
  fin pour
FIN

Observez les similitudes entre cet algorithme et la version utilisant la boucle `tant que`. Notez bien que l'on utilise une boucle `pour` quand on sait en rentrant dans la boucle combien d'itérations devront être faites. Par exemple, n'utilisez pas une boucle `pour` pour contrôler une saisie !

1.4 Tableaux

Considérons un algorithme dont l'exécution donnerait :

```
Saisissez 10 valeurs :
4
90
5
-2
0
6
8
1
-7
39
Saisissez une valeur :
-7
-7 est la 9-ième valeur saisie.
```

Comment rédiger un tel algorithme sans utiliser dix variables pour stocker les 10 valeurs ?

1.4.1 Définition

Une tableau est un regroupement de variables de même type, il est identifié par un nom. Chacune des variables du tableau est numérotée, ce numéro s'appelle un **indice**. Chaque variable du tableau est donc caractérisée par le nom du tableau et son indice.

Si par exemple, T est un tableau de 10 variables, alors chacune d'elles sera numérotée et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable. Les différentes variables de T porteront des numéros de 1 à 10, et nous appellerons chacune de ces variables un **élément** de T.

Une variable n'étant pas un tableau est appelée variable **scalaire**, un tableau par opposition à une variable scalaire est une variable **non scalaire**.

1.4.2 Déclaration

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la déclaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient. La syntaxe est :

$< type >:< nom > [< taille >]$

Par exemple,

$entier : T[4]$

déclare un tableau `T` contenant 4 variables de type entier.

1.4.3 Accès aux éléments

Les éléments d'un tableau à n éléments sont indicés de 1 à n. On note `T[i]` l'élément d'indice i du tableau `T`. Les quatre éléments du tableau de l'exemple ci-avant sont donc notés `T[1]`, `T[2]`, `T[3]` et `T[4]`.

1.4.4 Exemple

Nous pouvons maintenant rédiger l'algorithme dont le comportement est décrit au début du cours. Il est nécessaire de stocker 10 valeurs de type entier, nous allons donc déclarer un tableau `E` de la sorte :

```
entier : E[10]
```

La déclaration ci-dessus est celle d'un tableau de 10 éléments de type entier appelé `E`. Il convient ensuite d'effectuer les saisies des 10 valeurs. On peut par exemple procéder de la sorte :

```
Afficher "Saisissez dix valeurs : "
Saisir E[1]
Saisir E[2]
Saisir E[3]
Saisir E[4]
Saisir E[5]
Saisir E[6]
Saisir E[7]
Saisir E[8]
Saisir E[9]
Saisir E[10]
```

La redondance des instructions de saisie de cet extrait sont d'une laideur rédhibitoire. Nous procéderons plus élégamment en faisant une boucle :

```
Pour i allant de 1 à 10
| Saisir E[i]
fin pour
```

Ce type de boucle s'appelle un **parcours** de tableau. En règle générale on utilise des boucles pour manier les tableaux, celles-ci permettent d'effectuer un traitement sur chaque élément d'un tableau. Ensuite, il faut saisir une valeur à rechercher dans le tableau :

```
Afficher "Saisissez une valeur : "
Saisir t
```

Nous allons maintenant rechercher la valeur `t` dans le tableau `E`. Considérons pour ce faire la boucle suivante :

```
i ⟵ 1
Tant que E[i] <> t
| i ⟵ i + 1
fin tant que
```

Cette boucle parcourt le tableau jusqu'à trouver un élément de `E` qui ait la même valeur que `t`. Le problème qui pourrait se poser est que si `t` ne se trouve pas dans le tableau `E`, alors la boucle pourrait ne pas s'arrêter. Si `i` prend des valeurs strictement plus grandes que 10, alors il se produira ce que l'on appelle un **débordement d'indice**. Vous devez toujours veiller à ce qu'il ne se produise pas de débordement d'indice ! Nous allons donc faire en sorte que la boucle s'arrête si `i` prend des valeurs strictement supérieures à 10.

```
i ⟵ 1
Tant que i <= 10 et E[i] <> t
| i ⟵ i + 1
fin tant que
```

Il existe donc deux façons de sortir de la boucle :

- En cas de débordement d'indice, la condition `i <= 10` ne sera pas vérifiée. Une fois sorti de la boucle, `i` aura la valeur 11.

– Dans le cas où `t` se trouve dans le tableau à l'indice `i`, alors la condition `E[i] <> t` ne sera pas vérifiée et on sortira de la boucle. Un fois sorti de la boucle, `i` aura comme valeur l'indice de l'élément de `E` qui est égal à `t`, donc une valeur comprise entre 1 et 10.

On identifie donc la façon dont on est sorti de la boucle en testant la valeur de `i` :

```
Si i = 11 alors
| Afficher t, " ne fait pas partie des valeurs saisies."
sinon
| Afficher t, " est la ", i, "-ème valeur saisie."
fin si
```

Si `(i = 11)`, alors nous sommes sorti de la boucle parce que l'élément saisi par l'utilisateur ne trouve pas dans le tableau. Dans le cas contraire, `t` est la `i`-ème valeur saisie par l'utilisateur. Récapitulons :

```
Algorithme : Exemple tableau
Variables :
entiers : E[10], i, t
DEBUT
| Afficher "Saisissez dix valeurs : "
| Pour i allant de 1 à 10
| | Saisir E[i]
| fin pour
| Afficher "Saisissez une valeur : "
| Saisir t
| i ⟵ 1
| Tant que i <= 10 et E[i] <> t
| | i ⟵ i + 1
| fin tant que
| Si i = 11 alors
| | Afficher t, " ne fait pas partie des valeurs saisies."
| sinon
| | Afficher t, " est la ", i, "-ème valeur saisie."
| fin si
FIN
```

Chapitre 2

Exercices

2.1 Introduction

2.1.1 Affectations

Exercice 1 - Jeu d'essai

Après les affectations suivantes :

```
A <- 1
B <- A + 1
A <- B + 2
B <- A + 2
A <- B + 3
B <- A + 3
```

Quelles sont les valeurs de A et de B ?

Exercice 2 - Permutation des valeurs de 2 variables

Quelle série d'instructions échange les valeurs des deux variables A et B déjà initialisées ?

2.1.2 Saisie, affichage, affectations

Exercice 3 - Nom et âge

Saisir le nom et l'âge de l'utilisateur et afficher "Bonjour ..., tu as ... ans." en remplaçant les ... par respectivement le nom et l'âge.

Exercice 4 - Permutation de 2 variables saisies

Saisir deux variables et les permuter avant de les afficher.

Exercice 5 - Moyenne de 3 valeurs

Saisir 3 valeurs, afficher leur moyenne.

Exercice 6 - Aire du rectangle

Demander à l'utilisateur de saisir les longueurs et largeurs d'un rectangle, afficher sa surface.

Exercice 7 - Permutation de 4 valeurs

Ecrire un algorithme demandant à l'utilisateur de saisir 4 valeurs A, B, C, D et qui permute les variables de la façon suivante :

noms des variables	A	B	C	D
valeurs avant la permutation	1	2	3	4
valeurs après la permutation	3	4	1	2

Dans l'exemple ci-dessus, on suppose que l'utilisateur a saisi les valeurs 1, 2, 3 et 4. Mais votre algorithme doit fonctionner quelles que soient les valeurs saisies par l'utilisateur.

Exercice 8 - Permutation de 5 valeurs

On considère la permutation qui modifie cinq valeurs de la façon suivante :

noms des variables	A	B	C	D	E
valeurs avant la permutation	1	2	3	4	5
valeurs après la permutation	4	3	5	1	2

Ecrire un algorithme demandant à l'utilisateur de saisir 5 valeurs que vous placerez dans des variables appelées A, B, C, D et E. Vous les permuterez ensuite de la façon décrite ci-dessus.

Exercice 9 - Permutation ultime

Même exercice avec :

noms des variables	A	B	C	D	E	F
valeurs avant la permutation	1	2	3	4	5	6
valeurs après la permutation	3	4	5	1	6	2

Exercice 10 - Pièces de monnaie

Nous disposons d'un nombre illimité de pièces de 0.5, 0.2, 0.1, 0.05, 0.02 et 0.01 euros. Nous souhaitons, étant donné une somme S, savoir avec quelles pièces la payer de sorte que le nombre de pièces utilisée soit minimal. Par exemple, la somme de 0.96 euros se paie avec une pièce de 0.5 euros, deux pièces de 0.2 euros, une pièce de 0.05 euros et une pièce de 0.01 euros.

1. Le fait que la solution donnée pour l'exemple est minimal est justifié par une idée plutôt intuitive. Expliquez ce principe sans excès de formalisme.
2. Ecrire un algorithme demandant à l'utilisateur de saisir une valeur positive ou nulle. Ensuite, affichez le détail des pièces à utiliser pour constituer la somme saisie avec un nombre minimal de pièces.

Vous choisirez judicieusement les types de vos variables numériques pour que les divisions donnent bien les résultats escomptés.

2.2 Traitements conditionnels

2.2.1 Exercices de compréhension

Exercice 1

Que font les suites d'affectations suivantes ?

$A \longleftarrow 1$;
$B \longleftarrow 3$;
Si $A \geq B$ **alors**
| $A \longleftarrow B$;
sinon
| $B \longleftarrow A$;
fin si

Exercice 2

Pourriez-vous compléter les instructions suivantes ?

```
SI A = 0 ALORS
SI B = 0 ALORS
    Afficher A, " et ", B, " sont nuls."
SINON
        Afficher ...
...
```

2.2.2 Conditions simples

Exercice 3 - Majorité

Saisir l'âge de l'utilisateur et lui dire s'il est majeur.

Exercice 4 - Valeur absolue

Saisir une valeur, afficher sa valeur absolue. On rappelle que la valeur absolue de x est la distance entre x et 0.

Exercice 5 - Admissions

Saisir une note, afficher "ajourné" si la note est inférieure à 8, oral entre 8 et 10, admis dessus de 10.

Exercice 6 - Assurances

Une compagnie d'assurance effectue des remboursements en laissant une somme, appelée franchise, à la charge du client. La franchise représente 10% du montant des dommages sans toutefois pouvoir être inférieure à 15 euros ou supérieure à 500 euros. Ecrire un algorithme demandant à l'utilisateur de saisir le montant des dommages et lui affichant le montant remboursé ainsi que le montant de la franchise.

Exercice 7 - Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 8 - Recherche de doublons

Ecrire un algorithme qui demande à l'utilisateur de saisir trois valeurs et qui lui dit s'il s'y trouve un doublon.

Exercice 9 - Tri de 3 valeurs

Ecrire un algorithme demandant à l'utilisateur de saisir 3 valeurs et qui les affiche dans l'ordre croissant.

2.2.3 Conditions imbriquées

Exercice 10 - Signe du produit

Saisir deux nombres et afficher le signe de leur produit sans les multiplier.

Exercice 11 - Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs distinctes.

Exercice 12 - $ax + b = 0$

Saisir les coefficients a et b et afficher la solution de l'équation $ax + b = 0$.

Exercice 13 - $ax^2 + bx + c = 0$

Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$.

Exercice 14 - Opérations sur les heures

Ecrire un algorithme qui demande à l'utilisateur de saisir une heure de début (heures + minutes) et une heure de fin (heures + minutes aussi). Cet algorithme doit ensuite calculer en heures + minutes le temps écoulé entre l'heure de début et l'heure de fin. Si l'utilisateur saisit $10h30$ et $12h15$, l'algorithme doit lui afficher que le temps écoulé entre l'heure de début et celle de fin est $1h45$. On suppose que les deux heures se trouvent dans la même journée, si celle de début se trouve après celle de fin, un message d'erreur doit s'afficher. Lors la saisie des heures, séparez les heures des minutes en demandant à l'utilisateur de saisir :

- heures de début
- minutes de début
- heures de fin
- minutes de fin

Exercice 15 - Le jour d'après

Ecrire un algorithme de saisir une date (jour, mois, année), et affichez la date du lendemain. Saisissez les trois données séparément (comme dans l'exercice précédent). Prenez garde aux nombre de jours que comporte chaque mois, et au fait que le mois de février comporte 29 jours les années bissextiles. Une année est bissextile si elle est divisible par 4 mais pas par 100 (`http://fr.wikipedia.org/wiki/Ann\%C3\%A9e_bissextile`).

Vous utiliserez l'instruction $a \ mod \ b$ pour obtenir le reste de la division entière de a par b.

2.2.4 L'échiquier

On indice les cases d'un échiquer avec deux indices i et j variant tous deux de 1 à 8. La case (i, j) est sur la ligne i et la colonne j. Par convention, la case $(1, 1)$ est noire.

Exercice 16 - Couleurs

Ecrire un programme demandant à l'utilisateur de saisir les deux coordonnées i et j d'une case, et lui disant s'il s'agit d'une case blanche ou noire.

Exercice 17 - Cavaliers

Ecrire un programme demandant à l'utilisateur de saisir les coordonnées (i, j) d'une première case et les coordonnées (i', j') d'une deuxième case. Dites-lui ensuite s'il est possible de déplacer un cavalier de (i, j) à (i', j').

Exercice 18 - Autres pièces

Donner des conditions sur (i, j) et (i', j') permettant de tester la validité d'un mouvement de tour, de fou, de dame ou de roi.

2.2.5 Suivant Cas

Exercice 19 - Calculatrice

Ecrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques a et b, un opérateur op (vérifier qu'il s'agit de l'une des valeurs suivantes : $+, -, *, /$) de type caractère, et qui affiche le résultat de l'opération $a\ op\ b$.

2.3 Boucles

2.3.1 Utilisation de toutes les boucles

Les exercices suivants seront rédigés avec les trois types de boucle : tant que, répéter jusqu'à et pour.

Exercice 1 - compte à rebours

Ecrire un algorithme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant toutes les valeurs $n, n-1, \ldots, 2, 1, 0$.

Exercice 2 - factorielle

Ecrire un algorithme calculant la factorielle d'un nombre saisi par l'utilisateur.

Exercice 3

Repérer, dans les exercices de la section précédente, les types de boucles les plus adaptées au problème.

2.3.2 Choix de la boucle la plus appropriée

Pour les exercices suivants, vous choisirez la boucle la plus simple et la plus lisible.

Exercice 4

Ecrire un algorithme demandant à l'utilisateur de saisir la valeur d'une variable n et qui affiche la table de multiplication de n.

Exercice 5 - puissance

Ecrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques b et n (vérifier que n est positif) et affichant la valeur b^n.

Exercice 6

Ecrire un algorithme demandant à l'utilisateur de saisir la valeur d'une variable n et qui affiche la valeur $1 + 2 + \ldots + (n-1) + n$.

Exercice 7 - nombres premiers

Ecrire un algorithme demandant à l'utilisateur de saisir un nombre au clavier et lui disant si le nombre saisi est premier.

Exercice 8 - somme des inverses

Ecrivez un algorithme saisissant un nombre n et calculant la somme suivante :

$$\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{(-1)^{n+1}}{n}$$

Vous remarquez qu'il s'agit des inverses des n premiers nombres entiers. Si le dénominateur d'un terme est impair, alors vous l'aditionnez aux autres, sinon vous le soutrairez aux autres.

Exercice 9 - n^n

Écrire un algorithme demandant la saisie d'un nombre n et calculant n^n. Par exemple, si l'utilisateur saisit 3, l'algorithme lui affiche $3^3 = 3 \times 3 \times 3 = 27$.

Exercice 10 - racine carrée par dichotomie

Ecrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques x et p et affichant $\sqrt{x}$ avec une précision p. On utilisera une méthode par dichotomie : à la k-ème itération, on cherche x dans l'intervalle $[min, sup]$, on calcule le milieu m de cet intervalle (à vous de trouver comment la calculer). Si cet intervalle est suffisament petit (à vous de trouver quel critère utiliser), afficher m. Sinon, vérifiez si $\sqrt{x}$ se trouve dans $[inf, m]$ ou dans $[m, sup]$, et modifiez les variables inf et sup en conséquence. Par exemple, calculons la racine carrée de 10 avec une précision 0.5,

- Commençons par la chercher dans $[0, 10]$, on a $m = 5$, comme $5^2 > 10$, alors $5 > \sqrt{10}$, donc $\sqrt{10}$ se trouve dans l'intervalle $[0, 5]$.
- On recommence, $m = 2.5$, comme $\frac{5}{2}^2 = \frac{25}{4} < 10$, alors $\frac{5}{2} < \sqrt{10}$, on poursuit la recherche dans $[\frac{5}{2}, 5]$
- On a $m = 3.75$, comme $3.75^2 > 10$, alors $3.75 > \sqrt{10}$ et $\sqrt{10} \in [2.5, 3.75]$
- On a $m = 3.125$, comme $3.125^2 < 10$, alors $3.75 < \sqrt{10}$ et $\sqrt{10} \in [3.125, 3.75]$
- Comme l'étendue de l'intervalle $[3.125, 3.75]$ est inférieure 2×0.5, alors $m = 3.4375$ est une approximattion à 0.5 près de $\sqrt{10}$.

2.4 Tableaux

Exercice 1 - Initialisation et affichage

Ecrire un algorithme les valeurs $1, 2, 3, \ldots, 7$ dans un tableau T à 7 élément, puis affichant les éléments de T en partant de la fin.

Exercice 2 - Contrôle de saisie

Ecrire un algorithme plaçant 20 valeurs positives saisies par l'utilisateur dans un tableau à 20 éléments. Vous refuserez toutes les valeurs strictement négatives.

Exercice 3 - Choix des valeurs supérieures à t

Écrire un algorithme demandant à l'utilisateur de saisir dix valeurs numériques puis de saisir une valeur t. Il affichera ensuite le nombre de valeurs strictement supérieures à t. Par exemple, si l'utilisateur saisit $4, 19, 3, -2, 8, 0, 2, 10, 34, 7$ puis 3, alors le nombre de valeurs strictement supérieures à 3 parmi les 10 premières saisies est 6 ($4, 19, 8, 10, 34$ et 7).