Dauphine | PSL
UNIVERSITÉ PARIS

2019 - 2020

Sciences des organisations
L2 - Semestre 4

UE Y06 - Sciences du digital (Informatique)

Cours

Lucie GALAND
Florian SIKORA

Reproduction effectuée par l'Université Paris-Dauphine avec l'autorisation du CFC (20, rue des Grands Augustins 75006 Paris). La reproduction de ce document par tout moyen que ce soit est interdite conformément aux articles L. 111-1 et L. 122-4 du code de la propriété intellectuelle.

Table des matières

1 Bases de la programmation en VBA 6

1.1 Programmation en VBA . . . 7

1.1.1 Environnement de programmation VBE . . . 7

1.1.2 Programmation impérative . . . 7

1.2 Variables . . . 7

1.3 Interactions avec l'utilisateur . . . 9

1.3.1 Affichage . . . 9

1.3.2 Saisie de valeurs . . . 11

1.4 Structures de contrôle . . . 12

1.4.1 Conditionnelles . . . 12

1.4.2 Boucles . . . 14

1.5 Procédures, macros et fonctions . . . 17

1.5.1 Procédures . . . 17

1.5.2 Fonctions . . . 19

1.6 Structure d'un programme VBA . . . 20

1.6.1 Structure du code . . . 20

1.6.2 Portée des variables . . . 21

1.7 Les entrées-sorties en VBA . . . 21

2 Traitement de données Excel avec VBA 27

2.1 Notion d'objets en VBA . . . 28

2.2 Quelques classes en VBA . . . 29

2.3 Exemples d'utilisation des classes VBA . . . 32

2.3.1 Instruction **With** . . . 32

2.3.2 Instruction **For Each** . . . 33

3 Représentation des données et algorithmique 34

3.1 Représentation des données et calcul binaire . . . 35

3.1.1 Les unités de base . . . 35

3.1.2 Représentation d'un entier en binaire . . . 36

3.1.3 Additions binaires . . . 36

3.1.4 Nombres négatifs . . . 37

3.2 Algorithmique et temps d'exécution . . . 38

3.2.1 Algorithmes de recherche d'un élément . . . 39

3.2.2 Algorithmes de tri des données . . . 41

3.3 Programmation récursive . . . 42

Il existe aujourd'hui de nombreux outils informatiques sophistiqués permettant de traiter des données complexes. Les tableurs (comme Excel par exemple) en font partie. Cependant, malgré toutes les possibilités offertes par de tels outils, ils ne peuvent répondre à toutes les situations. La programmation permet de personnaliser les solutions apportées par ces outils afin qu'ils puissent répondre à des besoins spécifiques. En particulier, la programmation en *VBA (Visual Basic for Applications)* offre la possibilité, entre autres, d'ajouter de nouvelles fonctions à Excel, d'automatiser des actions répétitives et de créer des interfaces personnalisées. L'objectif de ce polycopié de cours est de présenter au lecteur des notions fondamentales d'informatique théorique telles que la représentation des données et l'algorithmique, et de montrer comment l'application de ces notions à la programmation VBA permet d'augmenter considérablement l'utilisation d'outils informatiques, tels que des tableurs, et ainsi de gérer des projets de tailles plus ou moins importantes.

Introduction

Considérons la feuille Excel de la figure 1 dans laquelle sont indiquées les notes obtenues par un étudiant (Bob) aux UE du premier semestre (Droit privé, Microéconomie, Anglais, Sociologie, Macroéconomie, Satistique, Histoire de l'Entreprise et de la Gestion, Informatique, Option) ainsi que les coefficients associés.

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3		Droit	Micro	Anglais	Socio	Macro	Stats	Histoire	Info	Option	Moyenne
4	notes UE	10	12	15	8	9	6	8	16	11	
5	coefficients	0,5	0,5	0,5	0,5	1	1	0,5	1	0,5	
6											
7											
8											
9											

Figure 1 – Notes de Bob

Bob aimerait savoir s'il a validé son premier semestre. Supposons qu'il existe un programme VBA (sous la forme d'une *macro* nommée `Moyenne`) qui, à partir d'un tel tableau de notes et de coefficients, détermine la moyenne obtenue au premier semestre et l'inscrit dans la cellule (4,11). Pour savoir s'il a validé son semestre, Bob n'a qu'à exécuter ce programme. L'exécution de la macro `Moyenne` sur les notes de Bob donne le résultat illustré dans la figure 2 dans la feuille Excel :

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3		Droit	Micro	Anglais	Socio	Macro	Stats	Histoire	Info	Option	Moyenne
4	notes UE	10	12	15	8	9	6	8	16	11	10,5
5	coefficients	0,5	0,5	0,5	0,5	1	1	0,5	1	0,5	
6											
7											
8											

Figure 2 – Notes et moyenne de Bob

Bob a donc obtenu la moyenne ! Exécuter la macro `Moyenne` signifie qu'une suite d'instructions définies dans un programme VBA (probablement écrit par Bob) ont été exécutées. Ce programme VBA est accessible dans l'éditeur de macro *VBE (Visual Basic Editor)* (voir la figure 3).

Le texte de la macro `Moyenne` commence par l'instruction **Sub** `Moyenne()` et se termine par l'instruction **End Sub**. Ces deux instructions délimitent la macro dans laquelle les instructions sont définies.

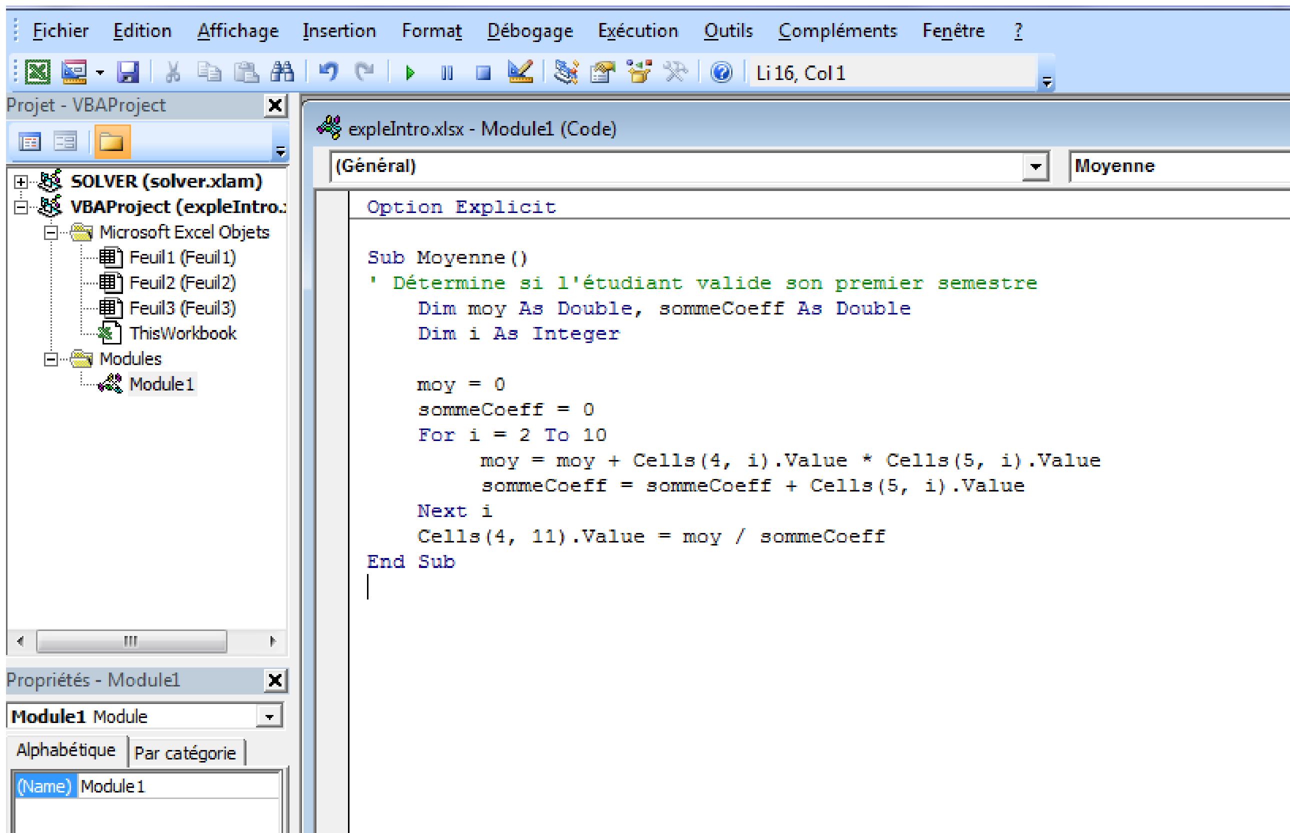


FIGURE 3 – Éditeur de macros

Le mot clé **Sub** est l'abréviation de *subroutine*, qui signifie *sous-routine* ou *sous-programme*. Les macros sont en effet des sous-programmes de l'application hôte. Dans ce programme, une ligne correspond à une instruction. La première ligne (commençant par une apostrophe) est une ligne de commentaires dans laquelle le programme est décrit. Les deux lignes suivantes (commençant par le mot clé **Dim**) permettent de déclarer des variables utiles pour le calcul, et les deux lignes suivantes permettent d'initialiser les variables (cf section 1.2 pour les variables). Une structure de boucle (**For**) est ensuite utilisée pour effectuer le calcul de la moyenne (cf section 1.4 pour les structures de contrôles). L'accès aux valeurs des cellules s'effectue par les instructions `Cells(4,i).Value` ou `Cells(5,i).Value`. Ces instructions signifient que l'on accède à la valeur de la cellule (4,i) (ou (5,i)) pour une valeur de `i` variant de 2 à 10. La notation pointée est typique de la *programmation orientée objet*. On accède ici à la valeur de l'objet cellule (cf section 2.1 pour le concept d'objet). Enfin, la dernière instruction (`Cells(4,11).Value = moy / sommeCoeff`) permet de modifier la valeur de la cellule (4,11) dans la feuille de calcul. Les concepts fondamentaux de la programmation en VBA, tels que ceux illustrés dans cet exemple (procédure, boucle, notion d'objet...), sont présentés dans la première partie de ce polycopié.

Première partie

Bases de la programmation en VBA

1.1 Programmation en VBA

1.1.1 Environnement de programmation VBE

Un programme VBA est lié au classeur Excel (qui peut être vu comme un ensemble de feuilles de calcul) dans lequel il est écrit. Il est constitué de *modules standard* (ou *modules de code*), de *modules de classes* et de *feuilles*. Un module est une feuille de calcul ne recevant que du code VBA et permettant de l'exécuter. Ces éléments interagissent et s'appellent pour constituer un programme complet.

L'environnement d'édition de VBA, appelé *VBE (Visual Basic Editor)*, permet d'écrire du code VBA dans des modules standards, des modules de classes ou des feuilles. Les constituants d'un projet VBA sont indiqués dans l'arborescence de la fenêtre **Explorateur de projets** en haut à gauche de VBE (voir par exemple l'arborescence de la fenêtre appelée *Projet - VBAProject* de la figure 3).

Le code standard se trouve dans des modules de code, stockés dans le dossier **Modules**. Dans l'exemple donné en introduction, **Module1** est le nom du module dans lequel est écrit la macro `Moyenne`. On pourrait aussi écrire du code VBA dans des modules de classes ou des feuilles. Du code VBA écrit dans une feuille ne peut être utilisé que dans la feuille de calcul Excel de même nom. Par exemple, le code VBA écrit dans la feuille **Feuil1** du projet de la figure 3 ne peut être utilisé que dans la feuille appelée **Feuil1** du classeur Excel. En revanche, tout code écrit dans un module de code peut être exécuté sur n'importe quelle feuille Excel du classeur. Pour cette raison, **tous les programmes VBA écrits dans le cadre de ce cours doivent être écrits dans des modules de code**.

1.1.2 Programmation impérative

VBA est un langage de *programmation impérative*, ce qui signifie que les opérations à effectuer sont codées sous la forme d'une suite d'instructions qui s'exécutent les unes après les autres. En VBA les différentes instructions sont séparées par un passage à la ligne ou par le signe `:`. Par exemple, considérons le programme suivant :

```
instruction1
instruction2
instruction3
```

L'exécution de ce programme entraîne l'exécution de `instruction1`, puis celle de `instruction2` et enfin celle de `instruction3`. Ce programme pourrait s'écrire de manière équivalente :

```
instruction1 : instruction2 : instruction3
```

Si une instuction est trop longue pour tenir sur une seule ligne, il suffit de passer à la ligne en écrivant le symbole _ précédé d'un espace pour préciser que l'instruction n'est pas finie. Ainsi l'exécution du programme suivant entraîne l'exécution de l'instruction `instruction1` :

```
instruc _
  tion1
```

1.2 Variables

Les variables permettent de stocker des informations à tout moment de l'exécution d'un programme et de les réexploiter à n'importe quel autre moment. En VBA, une variable possède :

- un *nom* qui permet d'accéder aux données qu'elle contient ;
- un *type de données*.

En VBA, ce nom ne peut contenir que des caractères alphanumériques (pas d'espace, de caractère spécial...) et doit commencer par une lettre. En général, les noms de variables commencent par une minuscule (la première lettre des mots clefs du langage sont automatiquement mis en majuscule par l'éditeur).

Types de données

Les différents types de données en VBA sont les suivants :

- types numériques :
  - type **Byte** : entiers compris entre 0 et 255 (8 bits)
  - type **Integer** : entiers compris entre -32768 et 32 767 (16 bits)
  - type **Long** : entiers compris entre -2 147 483 648 et 2 147 483 647 (32 bits)
  - type **Single** : nombres décimaux compris entre $-3,402823 \times 10^{38}$ et $-1,401298 \times 10^{-45}$ pour les nombres négatifs et entre $1,401298 \times 10^{-45}$ et $3,402823 \times 10^{38}$ pour les nombres positifs
  - type **Double** : nombres décimaux compris entre $-1,79769313486231 \times 10^{308}$ et $-4,94065645841247 \times 10^{-324}$ pour les nombres négatifs et entre $4,94065645841247 \times 10^{-324}$ et $1,79769313486231 \times 10^{308}$ pour les nombres positifs
- type **Boolean** : qui vaut **False** ou **True**
- type **String** : chaîne de caractères (texte mis entre guillemets)
- type **Date** : permet de mémoriser dates et heures ainsi que des durées
- type **Variant** : union de tous les types différents

Opérateurs

Les principaux opérateurs VBA que l'on peut appliquer en fonction des types des données sont récapitulés dans le tableau 1.

Type	Opérateurs
numérique	$+, -, *, /$, \ (division entière), ^ (élévation à la puissance), `mod` (modulo)
String	concaténation de chaînes de caractères : &
Date	$+, -$
Boolean	Not, And, Or
	opérateurs de comparaison : >, <, >=, <=, = (égalité), <> (différence)

Table 1 – Opérateurs VBA selon le type des données

Déclaration d'une variable

Pour utiliser une variable, il faut la déclarer, c'est-à-dire lui affecter un nom qu'il suffira par la suite d'utiliser pour exploiter la valeur qui y est stockée. La déclaration de variables s'effectue à l'aide du mot clé **Dim** :

```
Dim nomVar As typeVar
```

où `nomVar` est le nom de la variable et `typeVar` son type.

Affectation d'une variable

Une fois une variable déclarée, on peut lui affecter une valeur à l'aide du signe =. La valeur peut être une constante ou le résultat d'une expression.

```
Dim n As Integer
Dim x As Integer
Dim s As String
n = 10
```

```
x = n + 10
s = "bonjour !"
```

Notons que la déclaration de plusieurs variables peut s'écrire en une instruction. Ainsi les instructions précédentes peuvent s'écrire :

```
Dim n As Integer, x As Integer, s As String
n = 10
x = n + 10
s = "bonjour !"
```

Les trois variables `n`, `x` et `s` sont déclarées en une seule instruction.

Constantes

Les constantes permettent d'attribuer un nom à une valeur fixe. L'exploitation de cette valeur à travers son nom permet de faciliter la compréhension du programme. De plus, lorsqu'une valeur est susceptible d'être modifiée (une valeur de TVA par exemple), l'affectation de cette valeur à une constante simplifiera les éventuelles mises à jour. Il suffira en effet de modifier la valeur de la constante, plutôt que de modifier la valeur à chaque fois qu'elle est utilisée dans le code.

La déclaration d'une constante s'effectue à l'aide du mot clé **Const** :

```
Const nomConst As typeConst = valConst
```

où `nomConst` est le nom de la constante, `typeConst` son type et `valConst` la valeur qui lui est affectée. Par exemple, la déclaration d'une constante TVA à 20% peut s'effectuer par l'instruction :

```
Const TVA As Single = 0.2
```

1.3 Interactions avec l'utilisateur

Les interactions avec l'utilisateur peuvent s'effectuer à travers des *boîtes de dialogues* qui sont des fenêtres particulières dans lesquelles il est possible d'afficher des valeurs (valeurs dites *sorties* du programme) ou de saisir des valeurs (valeurs dites *entrées* du programme). Les deux métodes VBA que l'on va utiliser dans le cadre de ce cours sont `MsgBox` pour l'affichage et `Application.InputBox` pour la saisie de valeurs.

1.3.1 Affichage

L'affichage de valeurs s'effectue dans une boîte de dialogue s'ouvrant à l'exécution de l'instruction d'affichage suivante :

```
MsgBox msg
```

où `msg` est le message qui s'affiche dans la boîte de dialogue. `msg` est une chaîne de caractères ou une valeur.

Par exemple, l'exécution de l'instruction :

```
MsgBox "Bonjour !"
```

provoque l'affichage d'une boîte de dialogue contenant le texte « Bonjour ! », comme dans la figure 4. L'exécution des instructions suivantes :

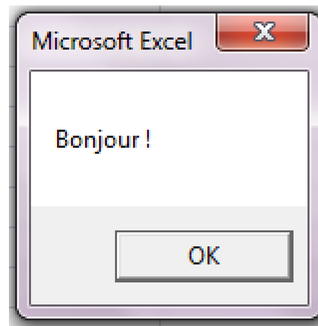


FIGURE 4 – Affichage de « Bonjour ! »

```
Dim maVar1 As String, maVar2 As String
maVar1 = "Bonjour, "
maVar2 = "vous allez bien ?"
MsgBox maVar1 & maVar2      'affichage de maVar1 & maVar2
```

provoque l'affichage du message « Bonjour, vous allez bien ? » dans une boîte de dialogue. Ce message est une chaîne de caractères construite par la concaténation de deux chaînes de caractères (instruction `maVar1 & maVar2`) stockées dans les variables `maVar1` et `maVar2`. Notons que l'on peut aussi bien afficher la valeur d'une variable, qu'une valeur évaluée directement. Ainsi,l'instruction

```
MsgBox "Bonjour, " & "vous allez bien ?"
```

est équivalente à l'exemple précédent, tout comme l'instruction suivante :

```
MsgBox "Bonjour, vous allez bien ?"
```

On peut aussi afficher des valeurs numériques avec `MsgBox`, comme dans l'exemple suivant :

```
Dim x As Integer
x = 10
MsgBox x
MsgBox 4
```

qui affiche 10 dans une boîte de dialogue, puis 4 dans une deuxème boîte de dialogue. Notons que chaque appel de `MsgBox` ouvre une boîte de dialogue.

Enfin, il est aussi possible d'afficher un texte contenant des chaînes de caractères et des valeurs numériques comme dans l'exemple suivant :

```
Dim s1 As String, s2 As String, x As Integer
s1 = "Vous avez "
s2 = " ans"
x = 13 + 5
MsgBox s1 & x & s2
```

où le message « Vous avez 18 ans » est affiché dans une boîte de dialogue. De manière équivalente, on aurait pu écrire l'instruction :

```
MsgBox "Vous avez " & 18 & " ans"
```

`MsgBox` est présentée de manière plus détaillée dans la section 1.7 sur les entrées-sorties.

1.3.2 Saisie de valeurs

La saisie de valeurs au cours de l'exécution d'un programme s'effectue à l'aide de l'instruction `Application.InputBox`. La valeur saisie par l'utilisateur est récupérée dans une variable (déjà déclarée) de la manière suivante :

```
res = Application.InputBox(msg)
```

où `res` est la variable qui contiendra la valeur saisie par l'utilisateur, et `msg` est le message qui s'affiche dans la boîte de dialogue lors de l'ouverture de celle-ci. Par exemple, l'exécution des instructions suivantes :

```
Dim res As Integer
res = Application.InputBox("Veuillez saisir un entier")
```

provoque l'affichage de la boîte de dialogue de la figure 5. L'utilisateur saisit alors une valeur dans le

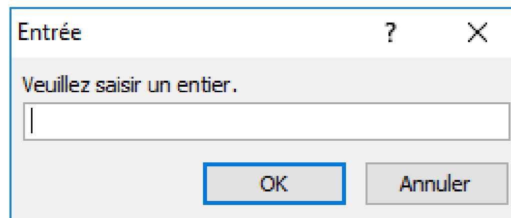


Figure 5 – Exécution de `Application.InputBox`

champ dédié, comme le montre la figure 6 où l'utilisateur a saisi la valeur 5. En cliquant sur le bouton

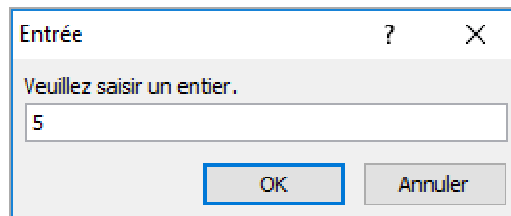


Figure 6 – Saisie par `Application.InputBox`

"OK", la boîte de dialogue se ferme et la variable `res` vaut alors 5.

Il est possible de contrôler le type de la valeur saisie par l'utilisateur (numérique, chaîne de caractères, etc.). Pour cela, on ajoute `Type:= x` dans la parenthèse de `Application.InputBox` où `x` est une valeur codant un type particulier. Quelques valeurs possibles pour `Type` sont présentées dans le tableau 2. Ainsi, la précision du type attendu dans les instructions suivantes permet de vérifier que la

Valeur	Type de données renvoyé
1	Valeur numérique
2	Chaîne de caractères
4	Valeur booléenne (**True** ou **False**)

Table 2 – Quelques valeurs du paramètre `Type` de `Application.InputBox`

valeur saisie par l'utilisateur est bien un nombre :

```
Dim res As Integer
res = Application.InputBox("Veuillez saisir un entier.", Type:=1)
```

Si ce n'est pas le cas, un message indiquant que le type est erronné est affiché dans une autre boîte de dialogue, puis la boîte de dialogue initiale attendant la saisie de l'utilisateur est à nouveau proposée à l'utilisateur. `Application.InputBox` est présentée de manière plus détaillée dans la section 1.7 sur les entrées-sorties.

1.4 Structures de contrôle

1.4.1 Conditionnelles

Conditionnelles simples

En VBA, l'écriture des alternatives est la suivante :

```
If condition1 then
  instruction1
[Else
  instruction 2]
End If
```

Les crochets ([...]) signifient que les instructions se trouvant à l'intérieur sont optionnelles. Le fonctionnement de cette structure est classique : si la `condition1` est vérifiée alors l'`instruction1` est exécutée, sinon l'`instruction2` est exécutée.

Par exemple, considérons un programme qui affecte comme valeur à une variable `b` (déjà déclarée) le nombre de chiffres d'un entier compris entre 0 et 99 stocké dans une variable `a` :

```
If a > 9 then
  b = 2
Else
  b = 1
End If
```

Ainsi si la valeur stockée dans la variable `a` est strictement supérieure à 9 (c'est-à-dire qu'elle est comprise entre 10 et 99), `b` vaut 2 (puisque `a` a deux chiffres), sinon `b` vaut 1 (puisque `a` n'a qu'un chiffre). Puisqu'il n'est pas obligatoire de définir l'instruction **Else**, on pourrait très bien écrire le programme précédent de la manière suivante :

```
b = 1
If a > 9 then
  b = 2
End If
```

Les deux programmes sont équivalents.

Conditionnelles multiples

Il est possible de tester plusieurs conditions à la suite à l'aide du mot clé **ElseIf** :

```
If condition1 Then
   instruction1
ElseIf condition2 Then
   instruction2
ElseIf condition3 Then
   instruction3
  .
  .
[Else
   instructionElse]
End If
```

Le fonctionnement de cette structure est le suivant : si la `condition1` est vérifiée alors l'`instruction1` est exécutée, sinon si la `condition2` est vérifiée alors l'`instruction2` est exécutée, sinon si la `condition3` est vérifiée alors l'`instruction3` est exécutée, et ainsi de suite. Si aucune de ces conditions n'est vérifiée, alors l'`instructionElse` est exécutée (si elle est définie).

Considérons par exemple un programme qui affecte comme valeur à la variable `b` (déjà déclarée) le nombre de chiffres d'un entier compris entre 0 et 100 stocké dans la variable `a` :

```
If a = 100 then
   b = 3
ElseIf a > 9 then
   b = 2
Else
   b = 1
End If
```

Ainsi si la valeur stockée dans la variable `a` est égale à 100, `b` vaut 3, sinon si elle est strictement supérieure à 9, `b` vaut 2, sinon `b` vaut 1.

Lorsque pour tester la valeur d'une même variable il existe plusieurs conditions possibles (trois dans l'exemple précédent), on peut simplifier l'écriture des **ElseIf** à l'aide des instructions **Select Case**.

```
   Select Case Expression
      Case valeur1
            instructions1
      Case valeur2
            instructions2
            .
            .
      Case valeurN
            instructionsN
      [Case Else
            instructionsElse]
   End Select
```

où `Expression` est une expression renvoyant une valeur. Lorsque la valeur renvoyée par `Expression` correspond à l'une des `valeurs` (`valeur1` ou `valeur2` ou ... ou `valeurN`), les instructions correspondantes sont exécutées (`instructions1` ou `instructions2` ou ... ou `instructionsN`). Si aucune de ces valeurs ne correspond à la valeur renvoyée par `Expression`, les instructions `instructionsElse` sont exécutées. L'utilisation de `Case Else` est facultative.

Par exemple, le programme précédent qui affecte comme valeur à une variable `b` (déjà déclarée) le

nombre de chiffres d'un entier compris entre 0 et 100 stocké dans une variable `a` peut s'écrire :

```
 Select Case a
   Case 100
        b=3
   Case 10 To 99
        b=2
   Case Else
        b=1
 End Select
```

Dans le cas où la valeur de `a` est 100, `b` vaut 3, dans celui où elle est comprise entre 10 et 99, `b` vaut 2, sinon `b` vaut 1 (la valeur de `a` est comprise entre 0 et 9). Notons que le mot clé **To** permet de définir un intervalle de valeurs possibles. Si l'une des valeurs de l'intervalle correspond à la valeur testée (la valeur de `a` dans l'exemple) alors les instructions associées à ce cas sont exécutées. Dans l'exemple, si la valeur de `a` est comprise entre 10 et 99 alors l'instruction `b=2` est exécutée.

1.4.2 Boucles

Boucles **Do Loop**

En VBA, les différentes boucles `Do Loop` sont les suivantes :

```
Do While condition
   instructions
Loop
```

Les instructions à l'intérieur de la boucle sont exécutées tant que la `condition` est vérifiée.

```
Do
   instructions
Loop While condition
```

Les instructions à l'intérieur de la boucle sont exécutées une fois, puis répétées tant que la `condition` est vérifiée.

```
Do Until condition
   instructions
Loop
```

Les instructions à l'intérieur de la boucle sont exécutées tant que la `condition` n'est pas vérifiée.

```
Do
   instructions
Loop Until condition
```

Les instructions à l'intérieur de la boucle sont exécutées une fois, puis répétées tant que la `condition` n'est pas vérifiée.

Par exemple, pour calculer le nombre de chiffres d'un entier `a` strictement positif, on peut écrire le programme suivant :

```
b = 0
Do While a > 0
        b = b + 1
        a = a \ 10 'Division entière de a par 10
Loop
```

Considérons l'exécution de ces instructions avec a ayant comme valeur 132 par exemple : b est initialisé à 0, puis le test du **While** est effectué : 132>0 est vrai. On rentre alors dans la boucle avec comme valeurs (provisoires) de a et b 132 et 0 respectivement. Le tableau suivant détaille les valeurs des variables a et b à l'issue de chacune des trois itérations de la boucle **While** :

	b	a
itération 1	1	13
itération 2	2	1
itération 3	3	0

La valeur de a étant 0 à l'issue de la troisième itération, le programme sort alors de la boucle **While** pour exécuter les instructions éventuelles qui suivent le mot clé **Loop**. La valeur de b est 3, ce qui correspond bien au nombre de chiffres de 132.

Ce programme peut être écrit de manière équivalente comme les programmes suivants :

```
b = 0
Do
        b = b + 1
        a = a \ 10 'Division entière de a par 10
Loop While a > 0
```

Notons qu'ici, la valeur de b est incrémentée au moins une fois. Ce programme est donc équivalent au précédent uniquement lorsque a est strictement positif. Si a valait 0, le programme affecterait à b la valeur 1 alors que le précédent affecterait à b la valeur 0.

```
b = 0
Do Until a = 0
        b = b + 1
        a = a \ 10 'Division entière de a par 10
Loop
```

```
b = 0
Do
        b = b + 1
        a = a \ 10 'Division entière de a par 10
Loop Until a = 0
```

Tout comme les boucles **While**, les deux programmes précédents n'effectuent la même opération que si la valeur de a est strictement positive. Si la valeur de a était 0, la valeur de b après exécution du premier programme utilisant la boucle **Do Until** serait 0, alors qu'elle serait de 1 après exécution du deuxième programme utilisant la boucle **Loop Until**.

Boucles **For**
En VBA, les boucles `For` suivent la structure suivante :

```
For var = valDeb To valFin [Step pas]
        instructions
Next var
```

Les instructions à l'intérieur de la boucle `For` sont exécutées pour la variable `var` valant `valDeb`, puis `valDeb`+ `pas`, puis `valDeb`+2×`pas`,... jusqu'à `valFin`. Les instructions sont donc exécutées ⌊(`valFin` − `valDeb` +1)/`pas`⌋ fois. Si le pas n'est pas précisé par l'option **Step** (qui est facultative), le pas est de 1 (tous les termes compris entre `valDeb` et `valFin` sont considérés successivement). Si `valFin` < `valDeb`, les instructions ne sont pas exécutées.

Par exemple, si l'on souhaite calculer la somme des n premiers entiers naturels, on peut écrire les instructions suivantes (`n` est déjà initialisée par ailleurs) :

```
Dim somme As Long
Dim i As Long           'i est la variable de boucle
somme = 0               'initialisation de la variable somme
For i = 1 To n
    somme = somme + i
Next i
```

A l'issue de ces instructions, la variable `somme` contient la somme des n premiers entiers. Pour cela, il faut commencer par déclarer les variables nécessaires au calcul (`somme` et `i`) puis initialiser la variable `somme` dans laquelle le résultat sera stocké au fur et à mesure du calcul. Initialement, la somme des entiers est nulle. On remarque qu'il n'est pas nécessaire d'initialiser la variable de boucle `i`. Son initialisation sera effectuée lors de l'entrée dans la boucle **For**. La variable `i` vaudra 1, puis 2, puis 3,..., jusqu'à n. Dans la boucle **For** 1 est ajouté à 0, puis 2 à la valeur courante de `somme`, puis 3,..., jusqu'à n qui est ajouté à la valeur courante de `somme` (qui contient alors les somme des $(n-1)$ premiers entiers). A la sortie de la boucle, la variable `somme` contient donc bien la somme des n premiers entiers.

Si l'on souhaite calculer la somme des n premiers entiers naturels impairs par exemple, on peut écrire les insctructions suivantes :

```
Dim somme As Long, i As Long
somme = 0
For i = 1 To n Step 2
    somme = somme + i
Next i
```

Sortie de boucles
Pour chacune des boucles **Do Loop** et pour la boucle **For**, il est possible de sortir de la boucle sans que la condition de sortie ne soit vérifiée en utilisant le mot clé **Exit**. Ainsi, l'instruction **Exit Do** à l'intérieur d'une boucle **Do Loop** permet de sortir de la boucle même si la condition de sortie n'est pas vérifiée, et l'instruction **Exit For** à l'intérieur d'une boucle **For** permet de sortir de la boucle même si le nombre d'itérations effectuées est inférieur à (`valFin` − `valDeb` + 1).

Utilisation des boucles
Les boucles permettent de répéter une instruction n fois sans avoir à écrire n instructions. Le choix entre l'utilisation d'une boucle **For** plutôt qu'une boucle **Do Loop** se fait en fonction de l'opération que l'on souhaite programmer. Le tableau 3 résume le choix du type de boucle à utiliser en fonction

du contexte (si l'on connaît à l'avance le nombre d'itérations à effectuer et/ou si l'on souhaite exécuter l'instruction au moins une fois ou non).

Exécution des instructions	nombre d'itérations connu	nombre d'itérations inconnu
au moins 1 fois	boucle **For**	boucle **Do Loop** condition à la fin
0, 1 ou plusieurs fois	boucle **For**	boucle **Do Loop** condition au début

TABLE 3 – Choix du type de boucle

1.5 Procédures, macros et fonctions

A l'intérieur d'un même module, le code est structuré en *procédures* et/ou *fonctions*. Nous allons voir à présent comment s'effectue la définition des procédures ou des fonctions en VBA.

1.5.1 Procédures

Déclaration d'une procédure

Une procédure est une suite d'instructions qui modifie l'environnement mais ne renvoie aucune valeur. La déclaration d'une procédure en VBA s'effectue à l'aide du mot clé **Sub** selon le format suivant :

```
Sub nomProc (par As typePar ,...)
'Commentaires
    instructions
End Sub
```

où `nomProc` désigne le nom de la procédure et `par` le nom d'un paramètre de type `typePar`. Afin d'améliorer la lisibilité du programme, on décrit en quelques mots ce que fait la procédure à l'aide de *commentaires* (texte qui ne sera pas interprété par l'ordinateur). Les commentaires en VBA s'écrivent après une apostrophe (') comme dans l'exemple précédent ou après le mot clé **Rem**. Notons que l'on peut aussi commenter certaines instructions dans le but de rendre le programme plus lisible.

Une procédure peut être déclarée avec un ou plusieurs paramètres, mais aussi sans paramètre :

```
Sub nomProc ()
'Commentaires
    instructions
End Sub
```

Par exemple, la déclaration d'une procédure nommée `bonjour` affichant le message "Hello world!" peut s'écrire en VBA :

```
Sub bonjour ()
'Affiche un message
   MsgBox "Hello world!"
End Sub
```

Cette procédure est constituée d'une seule instruction : **MsgBox** `"Hello world!"`. Cette instruction est un appel à la procédure VBA **MsgBox** qui affiche le message passé en paramètre (ici, "Hello world!") à travers une boîte de dialogue (cf section 1.7).

Une procédure afficheNb qui affiche un nombre passé en paramètre s'écrit en VBA :

```
Sub afficheNb (nb As Integer)
'Affiche le nombre nb
    MsgBox nb
End Sub
```

Le nombre `nb` à afficher est passé en paramètre, **As** `Integer` signifie que le paramètre est de type `Integer`.

Appel d'une procédure

Pour exécuter une procédure, il suffit de faire appel à elle en écrivant son nom (en respectant la casse) suivi des arguments éventuels séparés par une virgule. L'appel d'une procédure sans argument consiste donc à écrire son nom. Par exemple, l'appel de la procédure `bonjour` s'effectue à l'aide de l'instruction suivante :

```
bonjour
```

Il provoque l'affichage d'une boîte de dialogue contenant le message "Hello world!" dans la feuille de calcul Excel comme l'illustre la figure 7.

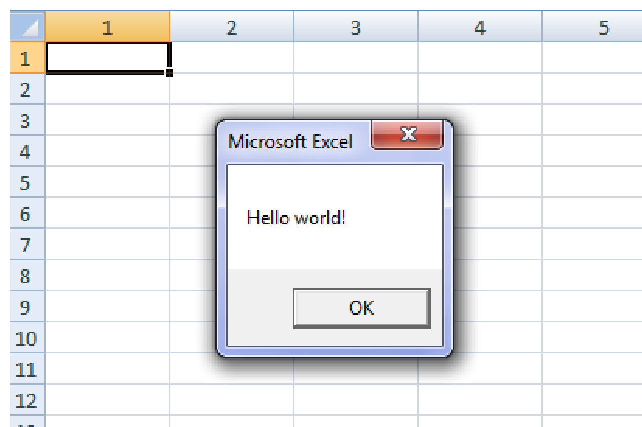


FIGURE 7 – Affichage du message "Hello world!"

Les paramètres d'une procédure déclarée avec des paramètres doivent être définis lors de son appel. Leur définition consiste à leur donner une valeur, soit directement (résultat d'une expression), soit à travers le contenu d'une variable déjà initialisée. L'appel de la procédure `afficheNb` pour afficher le nombre 10 par exemple peut s'effectuer à l'aide de l'instruction suivante :

```
afficheNb 10
```

ou, de manière équivalente :

```
Dim x As Integer
x = 10
afficheNb x
```

Il est important de noter que le nom de la variable en paramètre lors de l'appel de la procédure n'a pas

besoin d'être le même que le nom du paramètre dans de la déclaration de la procédure. En revanche, ces deux variables doivent être de même type.

Macros

Une macro VBA est une procédure sans paramètre. La procédure `bonjour` par exemple est donc une macro. Les macros sont les seules procédures VBA qui peuvent être exécutées depuis Excel.

Sortie de procédure

Il peut être utile de quitter une procédure avant la fin de son exécution. Pour cela, on peut utiliser l'instruction **Exit Sub**. Par exemple, dans la macro suivante, l'exécution de **Exit Sub** entraîne la sortie de la macro, l'instruction suivante (**MsgBox** `"Good bye world!"`) ne sera donc pas exécutée :

```
Sub expleSortie()
    MsgBox "Hello world!"
    Exit Sub                   'sortie de la macro
    MsgBox "Good bye world!"   'instruction jamais exécutée
End Sub
```

1.5.2 Fonctions

Déclaration d'une fonction

Une fonction est une suite d'instructions qui renvoie une et une seule valeur (résultat d'une expression). La déclaration d'une fonction VBA s'effectue à l'aide du mot clé **Function** selon le format suivant :

```
Function nomFonc (par As typePar,...) As typeRes
'Commentaires
    instructions
    nomFonc = expression
End Function
```

où `nomFonc` désigne le nom de la fonction et `par` le nom d'un paramètre de type `typePar`. Une fonction peut être déclarée avec un, plusieurs ou sans paramètres. **As** `typeRes` permet de spécifier le type de valeur renvoyée par la fonction.

Par exemple, pour calculer la somme des n premiers entiers naturels, on peut définir la fonction `sommeEnt` suivante :

```
Function sommeEnt(n As Long) As Long
'Fonction retournant la somme des n premiers entiers
    sommeEnt = (n * (n + 1)) / 2
End Function
```

Ainsi, étant donné un entier `n`, la fonction `sommeEnt` retourne la valeur `(n * (n + 1)) / 2`, ce qui correspond à la somme des `n` premiers entiers naturels.

Appel de fonctions

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (en respectant la casse) suivi de parenthèses contenant les arguments éventuels. L'appel d'une fonction sans paramètres consiste donc à écrire son nom suivi de parenthèses vides. L'appel de la fonction `sommeEnt` avec 10 comme paramètre s'écrit donc `sommeEnt(10)`.

Passage des paramètres

Les remarques sur le passage des paramètres des procédures restent valables pour les fonctions. Ainsi, `sommeEnt(10)` retourne la même valeur que `sommeEnt(x)` où `x` est préalablement initialisée à 10. Si une fonction prend plusieurs arguments, on les sépare à l'aide d'une **virgule**.

Sortie de fonction

Comme pour les procédures, il peut être utile de quitter une fonction avant la fin de son exécution. Pour cela, on peut utiliser l'instruction **Exit Function**. Dans ce cas, la fonction retourne la valeur 0.

Utilisation dans Excel

Les fonctions déclarées dans un module peuvent être utilisée dans une feuille Excel. Pour cela, comme toutes les autres fonctions d'Excel, on fait précéder le nom de la fonction par le signe = (par exemple `=sommeEnt(A1)` affichera la somme des entiers jusqu'à la valeur située dans la cellule A1). Remarque : lors d'un appel dans Excel, les différents arguments de la fonction doivent être séparés par des **points-virgules**.

1.6 Structure d'un programme VBA

1.6.1 Structure du code

Un programme VBA est un ensemble de *procédures* ou *fonctions* écrites dans le but d'effectuer une ou plusieurs opérations. Les instructions sont écrites à l'intérieur de ces procédures (ou fonctions). On parle alors de *déclaration* de procédures (ou fonctions). A l'intérieur d'un même module, le code est donc structuré en une succession de déclaration de procédures et/ou fonctions. Il s'écrit dans un module de code selon la structure suivante, où les instructions sont écrites à l'intérieur des différentes fonctions et procédures :

```
Sub Proc1 ...
...
End Sub

Sub Proc2 ...
...
End Sub

Function Fonc1 ...
...
End Function

Sub macro1()
Proc1 ...
...
End Sub
```

Dans cet exemple, le code VBA est constitué de la déclaration des procédures `Proc1` et `Proc2`, de la fonction `Fonc1` et de la macro `macro1`.

Une fois déclarées, les procédures (ou fonctions) sont exécutées lorsqu'elles sont *appelées*. Pour lancer l'exécution d'un programme VBA il faut exécuter une *macro* (procédure sans paramètre) dont les instructions peuvent être des appels à d'autres procédures (ou fonctions) déclarées dans le module. Dans l'exemple précédent, la première instruction de la macro `macro1` est un appel à la procédure

`Proc1`. La première opération effectuée lors de l'exécution de la macro `macro1` sera donc l'exécution de la procédure `Proc1`.

L'exemple suivant est un petit programme VBA permettant l'affichage de la somme des 10 premiers entiers naturels à l'aide de la macro `afficheSom10` qui affiche le résultat de l'appel de la fonction `sommeEnt` avec 10 comme paramètre :

```
Function sommeEnt(n As Long) As Long
'Fonction retournant la somme des n premiers entiers
    sommeEnt = (n * (n + 1)) / 2
End Function

Sub afficheSom10 ()
'Affiche la somme des 10 premiers entiers
    MsgBox sommeEnt(10)
End Sub
```

1.6.2 Portée des variables

Outre leur type et leur valeur, les variables sont caractérisées par leur *portée*. La portée d'une variable (ou d'une constante) désigne son accessibilité pour les procédures et les modules du projet. Elles peuvent être accessibles :

- **au niveau d'une procédure :** c'est le cas de toute variable/constante déclarée au sein d'une procédure
- **au niveau de l'ensemble des procédures d'un même module :** c'est le cas de toute variable/constante déclarée au niveau du module, c'est-à-dire en dehors de toute procédure ou fonction du module
- ou encore **au niveau de l'ensemble des modules du projet en cours :** c'est le cas de toute variable/constante déclarée *publique* au niveau du module.

Dans l'exemple suivant, la variable `varProc` a une portée de niveau procédure, la variable `varModPriv` a une portée de niveau module privée, et la variable `varModPub` a une portée de niveau module publique.

```
Public varModPub As Integer
Dim varModPriv As Integer
varModPub = 10 : varModPriv = 100

Sub maProc()
    Dim varProc As Integer
     varProc= 50
    MsgBox varProc 'affiche 50
    MsgBox varModPub + varModPriv 'affiche 110
End Sub

Sub maProc2()
    MsgBox varModPub + varModPriv 'affiche 110
    MsgBox varProc 'Erreur !! Variable non déclarée dans maProc2 !
End Sub
```

1.7 Les entrées-sorties en VBA

L'utilisation de boîtes de dialogue au cours de l'exécution d'un programme permet d'interagir avec l'utilisateur, soit en lui donnant des informations (valeurs dites *sorties* du programme) soit en lui demandant des informations (valeurs dites *entrées* du programme). Deux méthodes sont définies en VBA afin de permettre l'affichage de boîtes de dialogue :

— `Application.InputBox` : affiche une boîte de dialogue présentant une zone de texte dans laquelle l'utilisateur est invité à entrer des informations

— `MsgBox` : affiche un message à l'intention de l'utilisateur dans une boîte de dialogue et lui propose éventuellement de choisir entre différentes possibilités en cliquant sur l'un des boutons de commande affichés

Fonction `Application.InputBox`

```
Function Application.InputBox (Prompt As String, _
                               Optional Title As String, _
                               Optional Default As String, _
                               Optional Left As Integer, _
                               Optional Top As Integer, _
                               Optional HelpFile As String, _
                               Optional HelpContextId As Variant, _
                               Optional Type As Integer) As Variant
```

La fonction `Application.InputBox` affiche une boîte de dialogue qui comporte un cadre de saisie et des boutons OK et Annuler et renvoie les informations saisies. Elle prend en paramètre une chaîne de caractères (`Prompt`) qui correspond au message d'invite affiché dans la boîte de dialogue. Les autres paramètres sont optionnels, ils peuvent être omis lors de l'appel de la fonction `Application.InputBox`. `Title` correspond au titre de la boîte de dialogue (affiché dans la barre de titre). S'il est omis, le titre par défaut de la boîte de dialogue sera "Microsoft Excel". `Default` correspond au texte apparaissant par défaut dans la zone de texte lors de l'affichage de la boîte de dialogue. Les paramètres `Left` et `Top` permettent de spécifier l'emplacement de la boîte de dialogue sur l'écran au moment de son affichage, et les paramètres `HelpFile` et `HelpContextId` servent à associer des fichiers d'aide à la boîte de dialogue. Enfin le paramètre `Type` permet de préciser le type de données renvoyé par `Application.InputBox`. Le tableau 4 présente quelques valeurs possibles pour ce paramètre.

Valeur	Type de données renvoyé
1	Valeur numérique
2	Chaîne de caractères
4	Valeur booléenne (**True** ou **False**)
8	Référence de cellule (objet **Range**)

TABLE 4 – Valeurs du paramètre `Type` de `Application.InputBox`

Dans l'exemple suivant, seule la valeur des deux premiers paramètres optionnels (`Title` et `Default`) est précisée lors de l'appel de la fonction `Application.InputBox`.

```
Function age() As Integer
    age = Application.InputBox("Quel est votre âge ?", "Age","0")
End Function
```

La fonction `age` retourne l'âge de l'utilisateur. Pour cela, l'exécution de cette fonction provoque l'affichage d'une boîte de dialogue demandant à l'utilisateur de saisir son âge (figure 8).

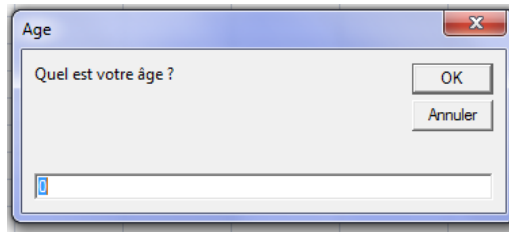


FIGURE 8 – Exécution de age

La valeur saisie par l'utilisateur est retournée par la fonction `age`. Les paramètres optionnels `Title` et `Default` étant optionnels, on pourrait définir la fonction `age2` suivante, équivalente à `age` :

```
Function age2() As Integer
  age2 = Application.InputBox ("Quel est votre âge ?")
End Function
```

L'appel de la fonction `age2` provoque l'affichage illustré dans la figure 9.

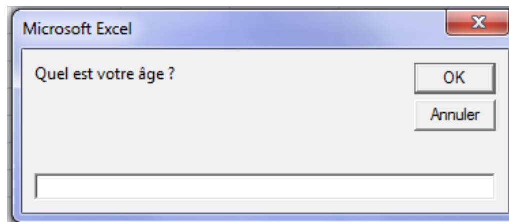


FIGURE 9 – Exécution de age2

Un seul des paramètres optionnels peut être précisé. Considérons par exemple la fonction `age3` :

```
Function age3() As Integer
  age3 = Application.InputBox ("Quel est votre âge ?","Age")
End Function
```

Si le nom du paramètre optionnel n'est pas précisé lors de l'appel de la fonction, alors, il s'agit du premier paramètre dans la définition de `Application.InputBox`, c'est-à-dire `Title` ici. L'appel de la fonction `age3` provoque l'affichage de la figure 10.

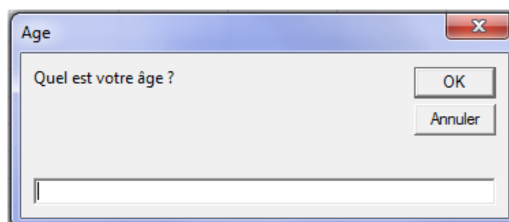


FIGURE 10 – Exécution de age3

Si l'on souhaite ne définir que les paramètres `Default` et `Type` lors d'un appel à `Application.InputBox`, il faut alors préciser devant la valeur du paramètre le nom du paramètre optionnel suivi du symbole :=. Ainsi, la fonction `age4` suivante fait appel à `Application.InputBox` en ne précisant que les valeurs des paramètres optionnels `Default` et `Type` :

```
Function age4() As Integer
  age4 = Application.InputBox ("Quel est votre âge ?",Default:="0",Type:=1)
End Function
```

Remarquons que la valeur de retour de la méthode `Application.InputBox` doit être de type **Integer** (`Type:=1`) dans la fonction `age4`. L'appel de la fonction `age4` provoque l'affichage de la figure 11.

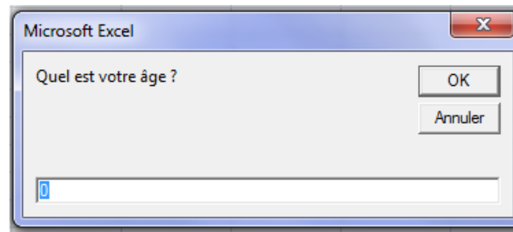


FIGURE 11 – Exécution de `age4`

Méthode `MsgBox`

`MsgBox` peut être utilisée comme une procédure (pour afficher un message d'information via une boîte de dialogue) ou comme une fonction (pour permettre à un utilisateur de spécifier un choix à travers des boutons).

```
Function MsgBox (Prompt As Variant, _
                 Optional Buttons As Long,_
                 Optional Title As String) As Integer
```

La méthode `MsgBox` permet d'afficher le message `Prompt` dans une boîte de dialogue ayant comme titre `Title`, la valeur de `Buttons` précisant les boutons qui sont affichés. Si le paramètre `Title` est omis lors de l'appel de `MsgBox`, le nom de l'application apparaît dans la barre de titre. Si le paramètre `Buttons` est omis lors de l'appel, un seul bouton libellé `OK` est affiché. Le tableau 5 présente les valeurs des paramètres `Buttons`.

Valeur	Description
0	OK
1	OK et Annuler
2	Abandonner, Recommencer et Ignorer
3	Oui, Non et Annuler
4	Oui et Non
5	Réessayer et Annuler

TABLE 5 – Valeurs du paramètre `Buttons` de `MsgBox`

La valeur de retour de la fonction `MsgBox` dépend du bouton sur lequel l'utilisateur a cliqué. La table 6 présente les valeurs possibles.

Valeur	Description
1	OK
2	Annuler
3	Abandonner
4	Recommencer
5	Ignorer
6	Oui
7	Non

TABLE 6 – Valeurs renvoyées par `MsgBox`

L'exécution de la macro `afficheMsg` suivante :

```
Sub afficheMsg ()
'Affiche un message
  MsgBox "Hello world!", 0, "Affichage d'un message"
End Sub
```

provoque l'affichage de la figure 12. Considérons maintenant la macro `majeur` qui permet d'afficher un message si l'utilisateur est majeur.

```
Sub majeur()
'Affiche un message si l'utilisateur est majeur
  Dim plus18 As Integer
   plus18 = MsgBox("Avez-vous plus de 18 ans ?", 4)
   If plus18 = 6 Then 'l'utilisateur a cliqué sur le bouton Oui
```

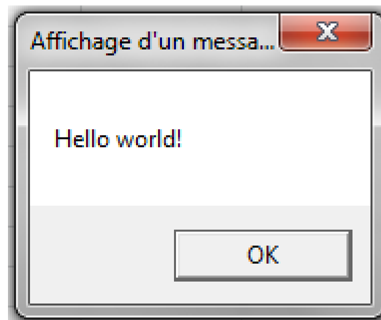
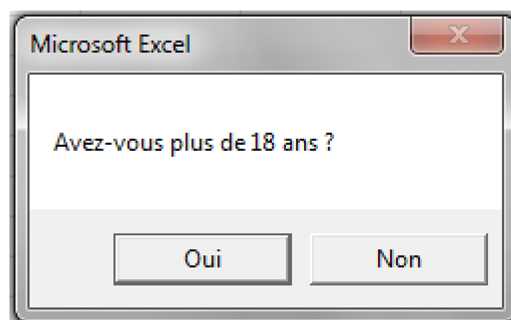


FIGURE 12 – Affichage de « Hello world! »

```
        MsgBox "Vous êtes majeur !"
    End If
End Sub
```

Dans cette macro, la méthode `MsgBox` est utilisée en tant que fonction (avec une valeur de retour) lors du premier appel, alors qu'elle est utilisée en tant que procédure lors du deuxième appel. L'exécution de la macro `majeur` provoque l'affichage suivant :



Deuxième partie

Traitement de données Excel avec VBA

La programmation en VBA permet de manipuler des données stockées dans des feuilles Excel. L'accès à ces données s'effectue par la manipulation des *objets* que représentent les différentes composantes d'un classeur Excel. Cette partie du polycopié est dédiée à une présentation rapide de la notion d'objets en VBA et notamment à la présentation de quelques classes en VBA permettant une manipulation simple de données stockées dans une feuille Excel.

2.1 Notion d'objets en VBA

Le VBA est un langage *orienté objet* : un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore un livre. Il possède une structure interne et un comportement, et il sait communiquer avec ses pairs.

Propriétés et méthodes d'un objet

L'objet est une entité que l'on peut distinguer grâce à ses *propriétés*, et sur laquelle on peut effectuer des *actions*. Prenons comme exemple un appartement : il est caractérisé par ses *propriétés* : nombre de pièces, superficie, étage, etc. et on peut y effectuer des *actions* (on parle alors de *méthodes*) : dormir, déjeuner, nettoyer l'appartement, regarder la télévision, etc. À partir de ces propriétés et méthodes, on peut imaginer de nombreux appartements différents, en faisant varier le nombre de pièces par exemple. Les propriétés permettent d'identifier l'objet (l'appartement ici) en le caractérisant. Les méthodes forment toutes les actions que l'on peut exécuter à partir de cet objet. Tous ces appartements ont été fabriqués à partir d'un plan : on parle d'une *classe*. Un objet construit à partir d'une classe est une *instance de classe*.

Notation pointée

Les méthodes ou les propriétés d'un objet sont accessibles à l'aide de la notation pointée :

```
<nom objet>.<nom méthode>
<nom objet>.<nom propriété>
```

Ainsi, pour effectuer l'action de regarder la télévision dans son appartement (instance `MonAppart` d'une classe `Appartement`, il suffit alors d'appliquer la méthode `RegarderTv` de l'objet `Appartement` :

```
MonAppart.RegarderTv
```

Un objet peut contenir un autre objet, et cet objet peut contenir un autre objet, etc. On dispose alors d'une hiérarchie d'objets. Par exemple, un appartement peut contenir des pièces, elles-même contenant des meubles. En VBA, Excel est la mère de tous les objets VBA, sa classe est appelée `Application`. Un fichier Excel est un objet de la classe `Workbook`, il peut contenir une ou plusieurs feuilles Excel (classe `Worksheet`). Une feuille Excel contient par exemple l'objet plage de cellules (classe `Range`). Pour accéder à ses différents niveaux de hiérarchie d'objets on utilise là encore la notation pointée. Ainsi, pour accéder à la cellule L3C4 d'une feuille Excel nommée « MaFeuille »(onglet « MaFeuille ») d'un classeur Excel nommé « maMacro.xlsm »(nom du fichier Excel) on pourrait écrire :

```
Application.Workbooks("maMacro.xlsm").Worksheets("MaFeuille").Cells(3, 4)
```

où `Cells(i,j)` désigne la cellule à la ligne `i` et colonne `j` (cf section 2.2). Heureusement, il n'est généralement pas nécessaire de préciser le chemin d'accès complet pour accéder à un objet donné. Sans précisions, il s'agit en effet de la feuille Excel active (celle sur laquelle on se trouve, i.e. onglet actif du classeur) du classeur actif (fichier sur lequel on se trouve). Ainsi

```
Cells(3, 4)
```

désigne la cellule L3C4 de la feuille Excel courante (onglet actif au moment de l'exécution de l'instruction) du classeur courant (fichier actif, à partir duquel l'instruction a été exécutée).

Collection

Dans l'instruction `Application.Workbooks("maMacro.xlsm").Worksheets("MaFeuille").Cells(3, 4)`, on peut noter que `Workbooks` et `Worksheets` sont au pluriel. Il s'agit en fait de *collections* d'objets. Une collection est un ensemble ordonné d'éléments auquel il est possible de faire référence en tant qu'unité. La collection `Workbooks` contient tous les classeurs (objet `Workbook`) ouverts et la collection `Worksheets` contient toutes les feuilles Excel (objet `Worksheet`) du classeur Excel désigné. Il existe différentes manières d'accéder à un membre d'une collection : par son nom ou son numéro par exemple. Ainsi `Worksheets("Mafeuille")` désigne la feuille Excel de nom « MaFeuille » (onglet « MaFeuille » du fichier Excel), et `Worksheets(1)` désigne la première feuille Excel du fichier courant (i.e. premier onglet en partant de la gauche).

2.2 Quelques classes en VBA

Dans ce cours, nous nous intéressons en particulier au traitement de données contenues dans une feuille Excel. Sauf mention contraire, nous pouvons supposer qu'il s'agit des cellules de la feuille Excel active du fichier courant. Ainsi, nous sommes amenés à utiliser essentiellement la classe `Range` permettant d'accéder aux cellules d'une feuille de calcul Excel. Nous présentons dans cette section différentes propriétés, méthodes et classes liées à la classe `Range`, qui peuvent être utiles pour ce cours.

Classe `Range`

La classe `Range` est la classe des plages de cellules. Entre autres, elle possède les propriétés suivantes :

```
Property Cells(i As Long, j As Long) As Range
' Lecture
' Cellule de la ligne i et de la colonne j relativement à l'objet Range

Property Range(Cell1 As Range, Cell2 As Range) As Range
' Lecture
' Plage dont les coins (supérieur gauche et inférieur droit) sont Cell1
' et Cell2 relativement à l'objet Range

Property Value As Variant
' Lecture-écriture
' Valeur de la cellule

Property FormulaR1C1Local As String
' Lecture-écriture
' Formule de la cellule (affichée dans la barre de formule)

Property Interior As Interior
' Lecture-écriture
' Objet qui représente le fond des cellules de la plage

Property Font As Font
' Lecture
' Style des caractères affichés dans les cellules de la plage
```

```
Property Row As Long
' Lecture
' Numéro de la première ligne de la première zone de la plage

Property Column As Long
' Lecture
' Numéro de la première colonne de la première zone de la plage
```

Certaines de ces propriétés peuvent être modifiées, elles sont dites en *lecture-écriture* (comme `Interior` par exemple), et d'autres ne peuvent pas être modifiées, elles sont dites en *lecture* (comme `Row`). En effet, si l'on peut modifier l'intérieur d'une cellule, on ne peut pas modifier le numéro de ligne d'une cellule d'une feuille de calcul.

Plusieurs méthodes sont aussi définies pour la classe `Range`, permettant ainsi d'appliquer des actions à une plage de cellules. Entre autres, la classe `Range` possède les méthodes suivantes :

```
Sub Clear()
' Efface le contenu et le formatage des cellules de la plage

Sub ClearContents()
' Efface le contenu des cellules de la plage

Sub Select()
' Sélectionne la plage dans la feuille de calcul active
```

Classe `Font`

La classe `Font` est la classe des styles de caractères. Entre autres, elle possède les propriétés suivantes :

```
Property Color As Long
' Lecture-écriture
' Couleur (codée en RGB) des caractères.

Property Bold As Boolean
' Lecture-écriture
' Graisse des caractères.

Property Italic As Boolean
' Lecture-écriture
' Inclinaison des caractères.

Property Size As Integer
' Lecture-écriture
' Taille (en points) des caractères.

Property Name As String
' Lecture-écriture
' Nom de la police des caractères.
```

Classe `Interior`

La classe `Interior` est la classe des fonds de cellules. Entre autres, elle possède les propriétés suivantes :

```
Property Color As Long
' Lecture-écriture
' Couleur (codée en RGB) du fond.

Property Pattern As Long
' Lecture-écriture
' Nombre représentant le motif du fond.

Property PatternColor As Long
' Lecture-écriture
' Couleur (codée en RGB) du motif du fond.
```

Codage RGB

Les propriétés et méthodes de certaines classes Excel peuvent faire intervenir des couleurs (c'est le cas des classes `Interior` et `Range` par exemple). Si l'on souhaite modifier la couleur de fond de cellule par exemple, on modifie la propriété `Color` de la classe `Interior` de la cellule concernée. Le codage de couleurs adopté ici est le *codage RGB* (pour *Red Green Blue*). Il permet de définir une couleur en donnant le niveau d'intensité de chacune des trois couleurs primaires (rouge, vert et bleu) : `RGB(r,g,b)` désigne donc une couleur dont le niveau d'intensité de rouge est `r`, le niveau d'intensité de vert est `g` et le niveau d'intensité de bleu est `b`. Les nombres `r`, `g` et `b` varient entre 0 et 255.

Ainsi, en VBA, `RGB(255,0,0)` correspond au codage de la couleur rouge car le niveau d'intensité de rouge est 255, et ceux de vert et bleu sont 0. Le codage de la couleur blanc est `RGB(255,255,255)`, et celui de la couleur noir est `RGB(0,0,0)`. En faisant varier le niveau d'intensité de chacune des trois composantes, on peut décrire un nombre considérable de couleurs. A titre d'exemples, la couleur orange a pour code `RGB(237,127,16)`, la couleur lavande a pour code `RGB(150,131,236)` et la couleur aubergine a pour code `RGB(55,0,40)`. Pour modifier la couleur de fond de la cellule L4C5 en couleur lavande par exemple, il suffit d'exécuter l'instruction suivante :

```
Cells(4,5).Interior.Color = RGB(150,131,236)
```

Affectation d'objets à une variable

Il est tout à fait possible d'utiliser des variables ayant comme type un objet. Par exemple, la variable nommée `c` de l'instruction suivante est de type `Range`.

```
Dim c As Range
```

Contrairement à l'affectation d'une valeur à une variable de type simple (numérique, booléen, chaîne de caractères, etc.), l'affectation d'une valeur à une variable ayant comme type un objet s'effectue impérativement à l'aide du mot clé `Set`. Ainsi, l'affectation de la plage L1C1 :L5C5 à une variable `c` de type `Range` s'écrit :

```
Dim c As Range
Set c = Range(Cells(1, 1), Cells(5, 5))
```

2.3 Exemples d'utilisation des classes VBA

Les instructions suivantes permettent de modifier le style de caractères sur une plage de cellules :

```
Range(Cells(1, 1), Cells(4, 8)).Font.Bold = True
Range(Cells(1, 1), Cells(4, 8)).Font.Size = 12
Range(Cells(1, 1), Cells(4, 8)).Font.Name = "Courier New"
```

`Range(Cells(1, 1), Cells(4, 8))` désigne la plage de cellules délimitée en haut à gauche par la cellule L1C1 et en bas à droite par la cellule L4C8 et `Range(Cells(1, 1), Cells(4, 8)).Font` désigne l'objet style de caractères de cette plage de cellules (L1C1 :L4C8). Après exécution de ces instructions, les caractères sur la plage L1C1 :L4C8 seront en gras (première instruction), auront une taille de 12 (deuxième instruction) et auront comme police de caractères « Courier New » (troisième instruction).

L'instruction suivante permet d'insérer une formule Excel dans la feuille de calcul active :

```
Cells(1, 3).FormulaR1C1Local = "=LC1-LC2"
```

Après exécution de cette instruction, la cellule L1C3 contiendra la formule Excel suivante : "=LC1-LC2". Cette formule signifie que le résultat du calcul (valeur apparaissant dans L1C3) est calculé en effectuant la différence entre la valeur de la cellule désignée par `LC1`, c'est-à-dire sur la même ligne à la colonne 1 (cellule L1C1) et la valeur de la cellule désignée par `LC2`, c'est-à-dire sur la même ligne à la colonne 2 (cellule L1C2). La formule apparaît dans la barre des formules et le résultat dans la cellule L1C3 comme le montre la figure 13.

L1C3 | fx | =LC1-LC2

	1	2	3	4	5
1	10	8	2		
2					
3					

Figure 13 – Exemple d'utilisation de `FormulaR1C1Local`

2.3.1 Instruction With

Pour modifier les différentes propriétés d'un objet, il peut être fastidieux de répéter le chemin d'accès à cet objet pour chaque propriété que l'on souhate modifier. Supposons par exemple que dans l'exemple précédent de modification du style de caractères de la plage de cellules L1C1 :L4C8, on souhaite aussi mettre les caractères en italique et de couleur rouge. Le code serait alors le suivant :

```
Range(Cells(1, 1), Cells(4, 8)).Font.Bold = True
Range(Cells(1, 1), Cells(4, 8)).Font.Size = 12
Range(Cells(1, 1), Cells(4, 8)).Font.Name = "Courier New"
Range(Cells(1, 1), Cells(4, 8)).Font.Italic = True
Range(Cells(1, 1), Cells(4, 8)).Font.Color = RGB(255,0,0)
```

L'instruction **With** permet d'effectuer tous ses changements de propriétés du même objet sans avoir à répéter l'objet dans le code. Elle suit la syntaxe suivante :

```
With objet
  Instructions
End With
```

L'exemple précédent peut donc s'écrire :

```
With Range(Cells(1, 1), Cells(4, 8)).Font
    .Bold = True
    .Size = 12
    .Name = "Courier New"
    .Italic = True
    .Color = RGB(255,0,0)
End With
```

Une propriété d'un objet peut être elle-même un objet (c'est le cas par exemple de la propriété `Font` de la classe `Range`). L'instruction **With** permet de modifier les différentes propriétés des objets qui peuvent eux-même être des propriétés d'une classe en imbriquant les **With**. Dans l'exemple suivant, les deux propriétés `Value` et `Font` de la classe `Range` sont modifiées. La modification de la propriété `Font` s'effectuant par la modification de plusieurs de ses propriétés (`Bold`, `Size` et `Name`).

```
With Range(Cells(1, 1), Cells(4, 8))
    .Value = 10
    With .Font
        .Bold= True
        .Size = 12
        .Name = "Courier New"
    End With
End With
```

2.3.2 Instruction For Each

Les boucles `Do Loop` et `For` permettent de parcourir les cellules d'un objet de type `Range`. Mais cela nécessite cependant une manipulation d'indices i et j afin d'accéder aux différentes cellules LiCj de la plage. En VBA, il existe l'instruction **For Each** qui permet de parcourir les différentes cellules d'une plage sans devoir explicitement gérer leurs numéros de ligne et colonne. De manière plus générale, l'instruction **For Each** permet d'itérer sur les éléments d'une collection, selon la syntaxe suivante :

```
For Each var In Collection
    Instructions
Next var
```

Par exemple, les instructions suivantes modifient la couleur de police de caractères des valeurs contenues dans la plage L1C1 :L8C8 en fonction de la valeur de ses cellules :

```
Dim c As Range
For Each c In Range(Cells(1,1),Cells(8,8))
    If c.Value < 10 Then
        c.Font.Color = RGB(255, 0, 0)
    ElseIf c.Value > 15 Then
        c.Font.Color = RGB(0, 255, 0)
    End If
Next c
```

A l'issue de l'exécution de ces instructions, les nombres compris dans les cellules de la plage L1C1 :L8C8 seront en rouge s'ils sont strictement plus petits que 10 et en vert s'ils sont strictement plus grand que 15. Dans les autres cas, leur couleur n'est pas modifiée.

Troisième partie

Représentation des données et algorithmique

3.1 Représentation des données et calcul binaire [1]

Les ordinateurs calculent en binaire. Il s'agit de la base 2 dans laquelle seuls les symboles 0 et 1 sont utilisés pour écrire les nombres, sans bien sûr que cela ne limite la taille des nombres représentables. Ceux-ci vont être stockés dans des cases qui, elles, ont une taille fixée. L'ordinateur est donc limité dans ses capacités à exprimer n'importe quel nombre.

Dans nos activités quotidiennes nous avons l'habitude de nous servir de la base 10, ce choix étant très naturel, dicté par le nombre des doigts des deux mains. L'idée de représentation par des bases est apparu par la nécessité de représenter des grandes quantités de valeur (imaginez si on devait marquer un trait pour chaque unité pour représenter 1.000.000 !) et est basée sur le fait de faire des paquets avec des unités. Par exemple, quand on travaille avec la base 10 on fait des paquets de dix, si on a plus de 10 paquets on fait des paquets de ces paquets et ainsi de suite. Quand on dit 423 en base 10 nous entendons 3 unités, plus 2 paquets de dix et quatre paquets de 10 paquets de 10.

Bien que la base 10 soit la base la plus utilisée de nos jours, on trouve également :

- la base sexagésimale (60), utilisée par les Sumériens et parfois au moyen âge en France ou dans le calcul des angles ou encore en comptage du temps. Cette base est intéressante car 60 est divisible par 2, 3, 4, 5 et 6
- la base vicésimale (20), utilisée par les Mayas (il y a aussi des restes dans les nombres français, réfléchissez par exemple à quatre-vingt-dix-sept)
- la base duodécimale (12), utilisée dans la langue du Népal, dans le passé de nombreuses populations en ont fait usage. En latin par exemple, il existe un grand nombre de noms et d'adjectifs pour désigner des ensembles de douze (le mot douzaine est encore souvent utilisé)
- la base quinaire (5), utilisée aussi par les Mayas
- la base binaire (2), utilisée par les informaticiens.

Cette section sera dédiée à la base binaire. Le système binaire n'est évidemment pas naturel mais il est bien adapté aux contraintes des circuits électroniques des ordinateurs. Il est en effet simple de distinguer deux valeurs de tension sur un fil. Si on avait souhaité reproduire directement le calcul décimal, dix valeurs différentes de la tension auraient été nécessaires, ce qui aurait rendu les circuits beaucoup plus complexes.

Le but de cette section est bien évidemment de présenter la représentation des symboles dans un ordinateur, mais aussi de montrer que le choix d'une représentation a des conséquences sur l'utilisation des nombres : comment les opérations sont-elles effectuées ? Est-ce le même principe qu'avec la base 10 ? Comment peut-on représenter les nombres négatifs ? Peut-on tout représenter en sachant que le mémoire d'ordinateur est limitée ?...

3.1.1 Les unités de base

Le terme **bit** signifie "BInary digiT, chiffre binaire", c'est-à-dire 0 ou 1 en numérotation binaire. Il s'agit de la plus petite unité d'information manipulable par une machine numérique. Avec un bit il est ainsi possible d'obtenir deux états : soit 1, soit 0. Grâce à 2 bits, il est possible d'obtenir quatre états différents (00, 01, 10 et 11), avec 3 bits 8 états (000, 001, 010, 100, 110, 101, 011, 111) et d'une manière générale avec n bits on pourrait représenter 2^n états. Ceci signifie par exemple que si dans la représentation binaire on utilise 5 bits le plus grand nombre que l'on peut représenter est 31 ($2^5 - 1$, car il ne faut pas oublier de représenter zéro).

1. *Remarque :* nous remercions pour cette partie Jean-Marie Janod et Emmanuel Lazard. Certaines parties de cette section sont tirées du livre *Architecture de l'ordinateur*, Collection Syntext, Pearson Education France, 2006 d'Emmanuel Lazard et du polycopié de cours 2010/2011 de Jean-Marie Janod pour l'UE 47.

L'**octet** (en anglais byte) est une unité d'information composée de 8 bits. Il est considéré comme l'unité de base et permet de coder tous les caractères du clavier. Pour un octet, le plus petit nombre est 0 (représenté par huit zéros 00000000), et le plus grand est 255 (représenté par huit chiffres "un" 11111111), ce qui représente 256 possibilités de valeurs différentes. En effet, 26 lettres composent l'alphabet, soit 52 codes, si l'on distingue les majuscules des minuscules. Mais il faut aussi les ponctuations, les signes diacritiques propres à chaque langue, les symboles représentant les chiffres et les opérations. Enfin, il faut aussi coder le retour en début de ligne, le passage à la ligne, le saut de page, et réserver certaines configurations pour générer des protocoles de communication entre l'UC (unité centrale) et les périphériques : un minimum de 128 symboles. Signalons qu'historiquement ce choix ne fut pas motivé par les caractères, mais par la représentation des nombres et des considérations technologiques.

Ce regroupement de nombres par série de 8 permet une lisibilité plus grande, au même titre que l'on apprécie, en base décimale, de regrouper les nombres par trois pour pouvoir distinguer les milliers. Le nombre "1 256 245" est par exemple plus lisible que "1256245".

Les autres unités couramment utilisées sont :

— le kilo-octet noté ko = 2^{10} octets= 1024 octets
— le Mega-octet noté Mo = 2^{10} ko = 2^{20} octets
— le Giga-octet noté Go = 2^{10} Mo = 2^{30} octets
— le Téra-octet noté To = 2^{10} Go = 2^{40} octets

Les capacités des machines ainsi que la taille des fichiers sont données avec ces unités.

Remarque : il existe d'autres systèmes de numération. Dans un système de numération en base B si B est inférieur ou égal à 10, on utilise évidemment les chiffres arabes classiques ; ainsi en base 8, on se sert des chiffres de 0 à 7. Si B est supérieur à 10, il faut définir de nouveaux symboles pour "compter" entre 10 et $B-1$. La seule base d'utilisation courante dans laquelle le problème se pose est la base 16, qui utilise les lettres a, b, c, d, e et f pour les nombres de 10 à 15.

3.1.2 Représentation d'un entier en binaire

Comme expliqué précédemment pour la base 10, en numération en base 2 on fait des paquets de 2. Par exemple pour le nombre 5 (on sait que l'on a besoin de 3 bits : - - -), on fait des paquets de deux, il reste 1 unité (on écrit 1 tout à la fin de notre représentation : - -1) et deux paquets de deux, appelons les a et b. On fait alors des paquets de deux avec a et b, il reste zéro (on écrit 0 à droite : - 01), a et b faisant un seul paquet, on écrit à droite 1. La représentation de 5 est donc 101 en base binaire. D'une manière générale regrouper par deux revient, pour trouver son écriture en base deux, à diviser le nombre décimal par 2, garder le reste et recommencer avec le quotient, jusqu'à arriver à un quotient égal à 1. Par exemple pour trouver la représentation de 117, on divise 117 par 2, ce qui donne un reste de 1 et un quotient de 58. On poursuit la division jusqu'à aboutir à un quotient de 1. On lit alors les bits en "remontant les divisions" : $117_{10} = 1110101_2$.

A l'inverse, pour avoir la valeur décimale d'un nombre binaire, il suffit d'additionner les puissances de 2 correspondant aux bits mis à 1 dans l'écriture binaire : $10011 = 1\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0 = 16+2+1 = 19$

3.1.3 Additions binaires

De même que l'on additionne deux nombres décimaux chiffre à chiffre en obtenant un résultat et une retenue à reporter, pour additionner deux nombres binaires, il faut définir l'addition de 2 bits. Il n'y a que quatre cas à examiner :

— 0 + 0 donne 0 de résultat et 0 de retenue.
— 0 + 1 donne 1 de résultat et 0 de retenue.
— 1 + 0 donne 1 de résultat et 0 de retenue.
— 1 + 1 donne 0 de résultat et 1 de retenue (de même qu'il y a une retenue lors de l'addition de deux chiffres décimaux quand le résultat est supérieur à 10).

On effectue ensuite l'addition binaire bit à bit, de droite à gauche, en reportant les retenues, comme dans l'exemple suivant où on additionne 101011 et 1101110 :

$$\begin{array}{ccccccccc} & {}^{1} & {}^{1} & 1 & 0^{1} & 1^{1} & 0^{1} & 1 & 1 \\ + & & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}$$

On peut vérifier le résultat en décimal : $43 + 110 = 153$.

3.1.4 Nombres négatifs

La représentation que nous avons vue jusqu'à présent n'est valable que pour les nombres positifs, alors comment faire pour travailler avec des nombres négatifs ? Dans l'arithmétique classique, ces nombres sont précédés d'un signe moins, mais c'est ici impossible car on ne peut mettre que des bits dans une case mémoire. Il faut donc trouver une écriture, une représentation des nombres négatifs, utilisant les bits disponibles.

Représentation « signe et valeur absolue »

Directement issue de nos habitudes manuelles, il suffit de coder le signe + par 0 et le signe - par 1 dans le premier bit, puis d'utiliser la représentation précédente. L'implantation fixe alors le nombre total de bits ; ainsi avec 2 octets (16 bits) le premier est utilisé pour le signe et les 15 suivants permettent de coder tous les nombres entiers positifs de 0 à 32767. On obtient donc l'implantation des entiers relatifs de -32767 à 32767. D'une manière plus générale, avec n bits on peut représenter les nombres appartenant à l'intervalle $[-2^{n-1} + 1, 2^{n-1} - 1]$. Cette représentation offre l'avantage de la simplicité, car elle est conforme à nos habitudes. En revanche, elle conduit à deux représentations de zéro (+0 et -0) et a des méthodes d'addition plus complexes utilisant notamment la soustraction si les deux nombres sont de signes différents (on prend le plus grand en valeur absolue, on lui soustrait le plus petit et le résultat a le signe du plus grand en valeur absolue). La représentation que nous allons présenter dans la section suivante propose des opérations plus simples pour la soustraction. De plus si le résultat est en dehors de l'intervalle des nombres qui peuvent être représentés par le nombre de bits utilisés, on peut avoir des surprises étonnantes. Par exemple si on travaille avec deux octets, 32767+1 peut fournir, selon l'algorithme d'addition, le résultat -0.

Représentation en complément à 2

C'est la représentation standard sur les ordinateurs pour exprimer les nombres entiers négatifs, néanmoins elle est assez éloignée de nos habitudes.

Voici le principe : sur n bits, on exprime les nombres de l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$. On retrouve bien les 2^n nombres possibles. Un nombre positif est représenté de façon standard par son écriture binaire. On représente un nombre négatif en ajoutant 1 à son complément à 1 (obtenu en inversant tous les bits) et en ne tenant pas compte d'une éventuelle retenue finale.

Cette description peut paraître compliquée à première vue. Pour clarifier le concept de "complémentarité", commençons tout d'abord par présenter le complément à 10 qui est plus naturel pour nous. Raisonnons donc avec la base dix en introduisant une représentation des nombres de -10 à 9 à l'aide de deux chiffres. La représentation des nombres 0,1,2,3,4,5,6,7,8,9 est alors 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, sans changement. Pour les nombres négatifs utilisons une toute autre implantation : -1 sera représenté par 99, -2 par 98, -3 par 97, -4 par 96, -5 par 95, -6 par 94, -7 par 93, -8 par 92, -9 par 91, -10 par 90. Ainsi $-i$ est représenté par les deux derniers chiffres de $100-i$. Une autre façon de trouver la représentation d'un nombre négatif est de prendre le complément à 9 de chaque chiffre et d'ajouter au résultat final 1 (par exemple pour -5, on l'écrit d'abord -05, le complément à 9 de 0 est 9, le complément de 5 est 4, on obtient donc 94 ; on y ajoute 1 et à la fin on obtient 95 comme la représentation de -5).

Cette représentation peut surprendre mais elle offre une simplification remarquable pour les additions. Prenons un exemple simple : comment additionner -2 et -3 ? Dans l'arithmétique classique on obtient -5. Avec le nouveau codage -2 sera 98 et -3 sera 97. On additionne 98 et 97, on obtient 195 comme on n'a que deux chiffres à notre disposition le 1 est une retenue, la solution est 95 qui représente bien -5. Cela montre que la soustraction qui peut paraître comme une opération plus compliquée que l'addition devient une opération d'addition classique où on ne tient pas compte des retenues résiduelles éventuelles.

Le principe est similaire en base 2 sauf que comme on n'a que deux chiffres (0 et 1) on fait le complément à 2 au lieu de 10. La représentation des nombres positifs reste la même (comme 5 qui avait 05 en base 10) et celle des nombres négatifs se fait en prenant le complément à 1 de chaque chiffre et en ajoutant 1 au résultat final.

Donnons quelques exemples en envisageant des entiers courts codés sur un octet (8 bits). On peut alors représenter tous les entiers compris entre -128 et 127 :

nombre positif	représentation binaire	nombre négatif	représentation en complément à 2
0	0 0 0 0 0 0 0 0		
1	0 0 0 0 0 0 0 1	-1	1 1 1 1 1 1 1 1
2	0 0 0 0 0 0 1 0	-2	1 1 1 1 1 1 1 0
3	0 0 0 0 0 0 1 1	-3	1 1 1 1 1 1 0 1
4	0 0 0 0 0 1 0 0	-4	1 1 1 1 1 1 0 0
5	0 0 0 0 0 1 0 1	-5	1 1 1 1 1 0 1 1
122	0 1 1 1 1 0 1 0	-122	1 0 0 0 0 1 1 0
123	0 1 1 1 1 0 1 1	-123	1 0 0 0 0 1 0 1
124	0 1 1 1 1 1 0 0	-124	1 0 0 0 0 1 0 0
125	0 1 1 1 1 1 0 1	-125	1 0 0 0 0 0 1 1
126	0 1 1 1 1 1 1 0	-126	1 0 0 0 0 0 1 0
127	0 1 1 1 1 1 1 1	-127	1 0 0 0 0 0 0 1
		-128	1 0 0 0 0 0 0 0

Remarquons que comme dans la représentation précédente (signe et valeur absolue) nous sommes limités en taille et quand le résultat d'une opération dépasse les bornes des nombres représentables on risque d'avoir de mauvaises surprises. Par exemple sur un octet (donc les nombres doivent être entre -128 et 127), la somme de 125+5 va avoir comme représentation 1000000010 qui veut dire -126 !

Malgré ce dernier défaut, la représentation en complément à deux enlève l'ambiguité de zéro (qui n'a qu'une seule representation) et offre un même algorithme pour l'addition des nombres entiers positifs ou négatifs, ce qui est un avantage pour l'informatique.

3.2 Algorithmique et temps d'exécution

Dans cette partie du cours, nous nous intéressons à l'amélioration de la performance d'un programme grâce à l'utilisation d'algorithmes plus sophistiqués. Le temps de calcul est un sujet très important dans le domaine de l'informatique (plusieurs centres de recherche sont spécialisés sur ce sujet) car il entraîne des conséquences majeures dans des applications réelles. A titre d'exemple, mentionnons ici les automates qui sont utilisés dans certains services pour le passage d'ordres en bourse (des particuliers peuvent également acheter des logiciels qui automatisent les ordres de bourse en ligne). Le métier d'arbitragiste consiste à détecter des opportunités intéressantes sur l'achat et la vente de produits financiers. Dans le cadre d'arbitrage « haute fréquence » les opportunités sont détectées de manière automatique par un programme informatique, qui passe les ordres (également de manière automatique) sur le marché. L'arbitrage ne génère du profit que si vos ordres sont passés avant ceux de

vos concurrents. Les banques de marché ont des services spécialisés dans ce secteur très rémunérateur ; l'efficacité des algorithmes joue un rôle essentiel dans ce monde où une différence d'une milliseconde peut avoir des conséquences monétaires importantes.

Dans cette section, nous allons étudier en particulier l'impact de l'algorithmique sur le temps d'exécution d'un programme pour deux problèmes simples : la recherche d'un élément dans un ensemble d'éléments et le tri des éléments.

3.2.1 Algorithmes de recherche d'un élément

On suppose que des données sont stockées dans un tableau T, qui est une plage de cellules de la feuille de calcul Excel. On s'intéresse à la recherche d'un élément x dans le tableau de données T. Pour répondre à ce problème, on s'intéresse à l'écriture d'une fonction qui retourne un booléen nous indiquant la présence ou non de x dans T.

Recherche séquentielle

La façon la plus naturelle de rechercher un élément x dans un tableau T est de parcourir un à un tous les éléments de T jusqu'à trouver x. C'est le principe de la *recherche séquentielle*, qui effectue un parcours de tous les éléments du tableau dans l'ordre jusqu'à avoir trouvé x ou avoir parcouru tout le tableau. Notons qu'il est inutile de continuer à parcourir T si x a déjà été trouvé.

L'algorithme de recherche séquentielle d'un élément `x` dans un tableau T de taille n peut s'écrire :

```
Pour i de 1 à n
    Si T[i] = x Alors
        Retourner Vrai
    FinSi
FinPour
Retourner Faux
```

Recherche dans un tableau de données triées

On se place maintenant dans le cas où les données de T sont triées. Supposons par exemple que T contienne n entiers placés dans l'ordre croissant (par exemple $T = [3, 5, 7, 11, 13, 17, 19, 23, 29, 31]$ pour $n = 10$). Si l'élement x est petit par rapport aux données du tableau, il est inutile de le chercher parmi les derniers éléments du tableau. Par exemple, si l'on cherche le nombre 5 dans le tableau T donné précédemment, il est inutile de le chercher dans le sous tableau $[7, 11, 13, 17, 19, 23, 29, 31]$. On peut donc tirer parti du fait que les données sont triées dans T pour limiter le nombre d'éléments à examiner dans T. L'algorithme de recherche séquentielle d'un élément x dans un tableau T de taille n peut s'écrire en pseudo-code :

```
i=1
Tant que i <= n et T[i] < x
    i <- i + 1
FinTantQue
Si T[i] = x Alors
    Retourner Vrai
Sinon
    Retourner Faux
FinSi
```

Si l'élement est à la fin du tableau, on va cependant devoir parcourir quasiment tout le tableau. Par exemple, le nombre 29 sera trouvé dans le tableau T donné précédemment au bout de 9 itérations, ce qui signifie que 9 des 10 éléments de T ont été examinés. Pour éviter cela, on peut adopter une

stratégie de recherche plus sophistiquée : la *recherche dichotomique*. Le principe de la dichotomie ("couper en deux" en grec) est de diviser en deux parties l'espace de recherche à chaque étape et de ne rechercher que dans un de ces deux espaces. Cela permet d'éviter de parcourir tout l'espace de recherche. Dans le cadre de la recherche d'un élement dans un tableau, l'espace de recherche est défini par l'ensemble des élements à examiner dans le tableau. Sachant que le premier élément de T dans l'exemple précédent est 3, le dernier est 31, et que l'élément au milieu est 13, il suffit de chercher 29 dans la deuxième partie du tableau (qui contient des éléments compris entre 13 et 31). Examiner le sous-tableau $[3, 5, 7, 11, 13]$ est en effet complètement inutile pour chercher 29. Pour rechercher ensuite l'élément 29 dans le sous-tableau $[17, 19, 23, 29, 31]$, il suffit d'appliquer le même principe, c'est-à-dire rechercher 29 dans le sous-tableau $[23, 29, 31]$, et ainsi de suite jusqu'à avoir trouvé l'élement ou avoir un espace de recherche vide (i.e. il n'y a plus d'éléments à examiner dans le sous-tableau).

L'algorithme de recherche dichotomique d'un élément x dans le tableau trié T de taille n peut s'écrire en pseudo-code :

```
prem <- 1
dern <- n
Tant que prem <= dern
    mil <- ⌊(prem + dern)/2⌋
    Si T[mil] = x Alors
        Retourner Vrai
    SinonSi T[mil] < x Alors
        prem = mil + 1
    Sinon
        dern = mil - 1
    FinSi
FinTantQue
Retourner Faux
```

Le tableau suivant indique les valeurs des variables de l'algorithme de recherche dichotomique lors de son exécution sur le tableau $T = [3, 5, 7, 11, 13, 17, 19, 23, 29, 31]$ pour la recherche du nombre 29 :

	prem	dern	mil	T[mil]
itération 1	0, puis 5	9	4	13
itération 2	5, puis 8	9	7	23
itération 3	8	9	8	29

L'élement 29 est trouvé au bout de 3 itérations avec une recherche dichotomique, alors qu'il en a fallu 9 avec une recherche séquentielle.

De manière générale, on peut montrer que la recherche dichotomique nécessite au maximum $\lceil \log_2(n) \rceil + 1$ itérations pour trouver l'élement x dans un tableau T de taille n ou déterminer qu'il n'est pas dans T. La recherche séquentielle nécessite quant à elle au maximum n itérations (le pire cas étant le cas où l'élément x n'est pas présent dans T mais est plus grand que le dernier élément de T ou le cas où x est à la dernière position de T). Par exemple, en supposant que chaque itération dure 1ms, la recherche d'un élément dans un tableau de 4096 données triées prendra au maximum 13ms pour la recherche dichotomique alors que la recherche séquentielle pourrait durer plus de 4s. Lorsque l'on manipule d'importantes quantités de données, il est donc nécessaire de réfléchir à des stratégies algorithmiques permettant de traiter les données dans des temps raisonnables.

3.2.2 Algorithmes de tri des données

A nouveau, on suppose dans cette section que des données sont stockées dans un tableau T, qui est une plage de cellules de la feuille de calcul Excel. On s'intéresse au tri de ces données dans un ordre donné (croissant ou décroissant par exemple pour les nombres). Pour répondre à ce problème, on considère l'écriture d'une procédure qui, étant donné le tableau T, replace les éléments de T dans T dans un ordre donné. Pour la suite de cette section, on considère que T est un tableau d'entiers et que l'on souhaite trier ses éléments dans l'ordre croissant.

Tri par sélection

Le *tri par sélection* est probablement l'algorithme de tri le plus naturel. Le principe est simple : à chaque itération, on place l'élement le plus petit de T au début de T (en l'échangeant avec l'élément présent en première position de T), puis on recommence sur le sous-tableau de T dans lequel on ne considère pas le premier élément, et ainsi de suite jusqu'à avoir trié tous les éléments de T.

Ainsi, sur le tableau $T = [3, 7, 2, 1, 9, 5]$, on sélectionne en premier l'élement 1 (élément minimal) qui est à la position 4, et on l'échange avec l'élément en première position de T (3 ici). On obtient le tableau $T = [1, 7, 2, 3, 9, 5]$. On continue la procédure en considérant à présent le sous-tableau $[7, 2, 3, 9, 5]$. On échange le plus petit élément avec l'élément en première position dans le sous-tableau, ce qui donnerait $[2, 7, 3, 9, 5]$. A cette étape, le tableau T est donc $[1, 2, 7, 3, 9, 5]$. Les deux premiers éléments de T sont les deux plus petits et ils sont triés. On continue ainsi de suite jusqu'à obtenir le tableau $T = [1, 2, 3, 5, 7, 9]$. A l'issue de l'étape i de cette procédure, les i premiers éléments de T sont les i plus petits éléments placés dans l'ordre croissant. Au bout de n étapes (où n est la taille de T), les éléments de T sont donc triés. Cet algorithme peut s'écrire en pseudo-code de la manière suivante :

```
Pour i de 1 à n − 1
    min <− i
    Pour j de i + 1 à n
       Si T[j] < T[min] Alors
           min <− j
       FinSi
    FinPour
    Echanger T[i] et T[min]
FinPour
```

Dans cette procédure, `min` est une variable locale qui permet de conserver la position de l'élément minimum dans le sous-tableau en cours d'examen. La première boucle `Pour` (boucle sur `i`) permet de placer l'élément minimum du sous-tableau à la `i`-ème position. La seconde boucle `Pour` (boucle sur `j`) permet de rechercher l'élement minimum dans le sous-tableau en cours d'examen (allant de `i`+1 à `n`).

Tri à bulles

La méthode *Tri à bulles* consiste à parcourir la suite de nombres en commençant par la fin, et en effectuant un échange à chaque fois que l'on rencontre deux nombres successifs qui ne sont pas dans le bon ordre. Le nom de ce tri vient de ce que les éléments les plus petits (les plus "légers") remontent vers le début de la suite, comme des bulles dans un tube à essai.

Par exemple, sur le tableau $T = [10, 11, 3, 6, 4, 2]$:

— en commençant par la fin, 2 est échangé avec 4, puis avec 6, puis avec 3, puis avec 11, puis avec 10 ; on obtient donc $T = [2, 10, 11, 3, 6, 4]$, où le nombre le plus petit est bien en première position

— on recommence ensuite sur le sous-tableau $[10, 11, 3, 6, 4]$: 4 est échangé avec 6, ne bouge pas par rapport à 3, mais 3 est échangé ensuite avec 11 et 10 ; on obtient donc $T = [2, 3, 10, 11, 4, 6]$

— on continue ensuite sur le sous tableau $[10, 11, 4, 6]$: au bout de cette troisième étape on obtient $T = [2, 3, 4, 10, 11, 6]$

— en continuant de la même manière on obtient $T = [2,3,4,6,10,11]$ au bout de la quatrième étape, puis $T = [2,3,4,6,10,11]$ au bout de la cinquième étape (dans laquelle aucun échange n'est effectué ici)

A chaque étape i de cet algorithme, les i premiers éléments de T sont les i plus petits éléments placés dans l'ordre croissant. Au bout de n étapes (où n est la taille de T), les éléments de T sont donc triés. Cet algorithme peut s'écrire en pseudo-code de la manière suivante :

```
Pour i allant de 1 à n−1
  Pour j allant de n à i+1 par pas de −1
    Si T[j] < T[j−1] Alors
      Echanger T[j] et T[j−1]
    FinSi
  FinPour
FinPour
```

3.3 Programmation récursive

La méthode de programmation présentée jusqu'à présent dans ce polycopié suit le principe de la *programmation itérative*, fondée sur une série d'instructions à exécuter dans l'ordre afin d'effectuer un calcul. Le programme se décompose à travers des schémas itératifs (comme les boucles) au sein desquels on effectue un calcul en répétant des opérations. Ce concept de programmation peut s'avérer difficile à utiliser, et même totalement inefficace, dans certaines situations. C'est par exemple le cas lorsque les données sont naturellement représentées à l'aide de structures arborescentes. Prenons l'exemple de l'arbre généalogique qui fournit une représentation de la relation « enfant de » entre deux individus sur tout un ensemble d'individus. À partir de cette simple représentation, on peut déterminer tous les liens familiaux entre ces individus en suivant les chemins adéquats dans l'arbre. Ce type de calcul (déterminer les cousins jusqu'au 3e degré par exemple) s'effectue de manière très efficace grâce à cette représentation arborescente et aux parcours que l'on peut y faire. La mise en œuvre automatique de ce type de calcul sur de telles structures est difficile à réaliser en suivant le paradigme de la programmation impérative, mais elle est en revanche totalement naturelle à travers le paradigme de la *programmation récursive*.

La programmation récursive est un des concepts de base en informatique, qui permet d'effectuer de manière très efficace des calculs sur certains types de problèmes, de nature récursive, surpassant ainsi les meilleurs performances, en termes d'utilisation de la mémoire et de temps d'exécution, que l'on pourrait obtenir avec la programmation itérative. Le but de cette partie du cours est de faire une rapide introduction à ce concept.

Principe

La programmation récursive est fondée sur la décomposition du calcul en calculs sur des problèmes de même nature mais un peu plus simples (notamment sur de plus petites instances du même problème). Elle peut donc être vue comme la mise en œuvre du principe de récurrence, fondé sur le raisonnement par induction. Pour illustrer de manière très simple ce principe, considérons le calcul de la somme des n premiers entiers positifs. On peut utiliser le fait que cette somme est égale à n ajouté à la somme des $n-1$ premiers entiers positifs (pour $n \geq 1$). Ainsi, si l'on appelle $s(n)$ cette somme, elle peut être calculée à l'aide de la formule récurrente suivante :

$$s(n) = n + s(n-1)$$

La valeur de $s(n-1)$ peut aussi être calculée par la formule suivante :

$$s(n-1) = n-1 + s(n-2)$$

et ainsi de suite. Pour que ce calcul puisse se finir, il faut définir un *cas de base*, c'est-à-dire un (ou plusieurs) cas pour lequel on connaît directement la valeur sans avoir besoin de la calculer. Par exemple, on peut utiliser ici le fait que $s(0) = 0$.

Programmation récursive sous VBA

Ce principe de récurrence peut tout à fait être utilisé en programmation. On est alors amené à écrire des fonctions ou procédures qui s'appellent elles-mêmes. On parle alors de *fonctions récursives* ou *procédures récursives*. La définition d'une fonction (ou procédure) récursive consiste essentiellement à définir les deux éléments suivants :

— un *appel de la même fonction (ou procédure)* pour une valeur de paramètre différente,

— et un *cas de base*.

Attention, si le cas de base n'est pas défini, la fonction (ou procédure) récursive est vouée à s'exécuter indéfiniment !

Par exemple, on peut écrire en VBA une fonction récursive qui retourne la somme des entiers positifs de 0 à n pour un n passé en paramètre de la manière suivante :

```
Function sommeRec(n As Long) As Long
'Fonction récursive retournant la somme des n premiers entiers positifs
        If n = 0 Then
                sommeRec = 0
        Else
                sommeRec = n + sommeRec(n - 1)    'Appel récursif
        End If
End Function
```

L'exécution de `sommeRec(3)` déclenche le calcul de `3 + sommeRec(2)`. D'après la définition de la fonction `sommeRec(2)` vaut 2 + `sommeRec(1)`, et `sommeRec(1)` vaut 1 + `sommeRec(0)`. Enfin, `sommeRec(0)` vaut 0. En remplaçant dans les calculs successifs les valeurs calculées on obtient donc :

— `sommeRec(1)` $= 1 + \mathbf{0} = 1$

— `sommeRec(2)` $= 2 + \mathbf{1} = 3$

— `sommeRec(3)` $= 3 + \mathbf{3} = 6$

Le résultat calculé par `sommeRec(3)` est donc 6.

Cet exemple très simple est à but essentiellement pédagogique. Nous pourrions écrire le même calcul en version itérative à l'aide d'une simple boucle `Pour`. Nous avons même vu précédemment dans ce polycopié que ce calcul s'écrivait en une seule instruction en utilisant le résultat mathématique suivant : $s(n) = \frac{n(n+1)}{2}$. Il n'y a donc a priori pas de raison de privilégier la version récursive pour ce calcul. Son principal intérêt étant peut-être qu'elle est conceptuellement simple à mettre en oeuvre. Cependant, dans certaines situations, en plus d'être naturelle la programmation récursive va se révéler être beaucoup plus efficace que la programmation itérative. C'est par exemple le cas lorsque l'on manipule des structures de données, telles que les listes ou les arbres, qui permettent une représentation simple de problèmes (et donc de données) de très grosse taille. Ces notions de structures abstraites pour représenter les données dépassent le cadre de ce cours, elles ne seront donc pas abordées dans ce polycopié. Cependant, la programmation récursive peut se révéler pertinente, même sans utiliser de structures de données naturellement récursives, comme on peut le voir dans la partie suivante où l'on va observer que le recours à la programmation récursive permet de traiter plus efficacement que vu précédemment le problème du tri des données.

Tri fusion

La récursivité peut aussi être utilisée pour trier des données. C'est le cas de l'algorithme de *tri fusion* dont le principe est le suivant (on suppose à nouveau que l'on dispose d'un tableau T d'entiers

à trier, comme dans la section 3.2.2) :

— s'il n'y a qu'un seul élément à trier, il n'y a rien à faire, sinon
  — on trie récursivement la première moitié du tableau
  — on trie récursivement la deuxième moitié du tableau
  — on interclasse (fusionne) les deux moitiés ainsi obtenues

L'opération d'interclassement de deux tableaux consiste à examiner les premiers éléments des deux tableaux, placer le plus petit des deux dans le tableau résultat et le « supprimer » du sous-tableau dans lequel il était. La même opération est appliquée sur les deux sous-tableaux jusqu'à ce qu'ils ne contiennent plus d'élément.

Sur le tableau $T = [10, 2, 3, 11, 4, 6]$, le tri fusion commence par trier le sous-tableau $T_1 = [10, 2, 3]$ et le sous-tableau $T_2 = [11, 4, 6]$. Pour trier le sous-tableau T_1, le tri-fusion trie le sous-tableau $T_{11} = [10, 2]$ et le sous-tableau $T_{12} = [3]$. Pour trier T_{11}, l'algorithme de tri commence par trier les sous-tableaux $T_{111} = [10]$ et $T_{112} = [2]$. Ces deux sous-tableaux ne contiennent qu'un élément, ils sont donc triés. Il faut alors interclasser T_{111} et T_{112}, ce qui finira le tri de T_{11}. L'interclassement donne $T_{11} = [2, 10]$. L'interclassement de T_{11} et T_{12} donne $T_1 = [2, 3, 10]$. Le tri récursif de T_2 donne ensuite $T_2 = [4, 6, 11]$. L'interclassement de T_1 et T_2 donne enfin $T = [2, 3, 4, 6, 10, 11]$. Le tableau T est trié.

Pour mesurer l'intérêt de ce tri récursif, on s'intéresse à sa *complexité*. La complexité d'un algorithme permet de mesurer le nombre d'operations qu'il effectue. L'algorithme de tri fusion examine les éléments de T lors de l'opération d'interclassement. La complexité de cet algorithme dépend donc du nombre d'éléments examinés lors de chacun des interclassements. Sur l'exemple précédent, l'exécution de l'algorithme de tri fusion nécessite 3 comparaisons d'élements lors des interclassements pour trier T_1 et 3 comparaisons lors des interclassements pour trier T_2. L'interclassement de T_1 et T_2 nécessite 5 comparaisons. L'algorithme effectue donc en tout 11 opérations sur cet exemple. L'algorithme de tri par sélection et l'algorithme de tri à bulles effectuent $5 + 4 + 3 + 2 + 1 = 15$ comparaisons sur ce même exemple.

De manière générale, on peut montrer que le nombre d'opérations du tri fusion est de l'ordre de $n \log_2(n)$ pour un tableau à n éléments, alors qu'il est de l'ordre de n^2 pour les deux autres algorithmes de tri de la section 3.2.2. Si l'on doit trier un tableau de 8 192 ($= 2^{13}$) éléments par exemple, les méthodes de tri par sélection ou de tri à bulles nécessiteraient plus de 67 millions ($\approx 2^{26} = 8192^2$) d'opérations, alors que le tri fusion n'en nécessiterait que 106 496. Supposons que l'on dispose d'un ordinateur capable d'effectuer 1 000 opérations en 1 milliseconde. Il faudrait alors à peine plus d'un dixième de seconde ($\approx$ 0.11s) au programme pour trier les données s'il utilise le tri fusion, alors qu'il lui faudrait plus d'une minute ($\approx$ 67s) s'il utilise le tri par sélection ou le tri à bulles.