Algorithmique pour le BTS SIO

Alexandre Meslé

28 février 2022

Table des matières

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Le principe

Exemple 1 **- La surprise du chef**

Considérons la suite d'instructions suivante :
— Faites chauffer de l'eau dans une casserole
— Une fois que l'eau boue, placez les pâtes dans l'eau
— Attendez dix minutes
— Versez le tout dans un écumoire
— Vos pâtes sont prêtes.

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

Exemple 2 **- Ikea**

`http://www.videobuzzy.com/m/n5213`

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

Définition

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

algorithme est une séquence d'instructions exécutée de façon logique mais non intelligente.

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

Utilisation en informatique

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

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

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

1.1.2 Variables

Définition

Un algorithme se présente comme une liste d'instructions, elles sont écrites les unes au dessus des autres et elles sont exécutées **dans l'ordre**, lorsque l'on conçoit une algorithme, il faut toujours avoir en tête le fait que l'ordre des instructions est très important. Le premier concept nécessaire pour concevoir un algorithme est celui de variable.

Une **variable** est un emplacement de la mémoire dans lequel est stockée une valeur.

— Une variable porte un **nom** , ce nom est laissé au choix du concepteur de l'algorithme, il doit commencer par une lettre et ne pas comporter d'espace.
— Une variable ne peut contenir qu'une seule valeur à la fois
— Le nom d'une variable permet de lire sa valeur ou la modifier

Notion de type

Une valeur est

— `numérique` s'il s'agit d'un nombre
— `alphanumérique` s'il s'agit d'une succession de symboles, par exemple des mots

Une variable a un **type**, qui détermine les valeurs que l'on pourra y placer.

Si une variable est de type numérique, il n'est possible d'y mettre que des valeurs numériques, si une variable est de type alphanumérique, il n'est possible d'y stocker que des valeurs alphanumériques.

L'affectation

L'**affectation** est une opération permettant de modifier la valeur d'une variable :

```
nomvariable ⟵ valeur
```

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

$A \longleftarrow 5$

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

$A \longleftarrow 5 + 2$

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

$A \longleftarrow B$
$A \longleftarrow B + 2$

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

$A \longleftarrow A + 1$

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

Exemple

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

$A \longleftarrow 1$
$B \longleftarrow 2$
$C \longleftarrow 3$
$D \longleftarrow A$
$A \longleftarrow C + 1$
$B \longleftarrow D + C$
$C \longleftarrow D + 2 * A$

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

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

`n.i` signifie ici **non initialisée**. Une variable est non initialisée si aucune valeur ne lui a été explicitement affectée. $A \longleftarrow 1$ modifie la valeur contenue dans la variable A. A ce moment-là de l'exécution, les valeurs des autres variables sont inchangées. $B \longleftarrow 2$ modifie la valeur de B, les deux variables A et B sont maintenant initialisées. $C \longleftarrow 3$ et $D \longleftarrow A$ initialisent les deux variables C et D, maintenant toutes les variables sont initialisées. Vous remarquerez que

D a été initialisée avec la valeur de A, comme A est une variable initialisée, cela a un sens. Par contre, si l'on avait affecté à D le contenu d'une variable non initialisée, nous aurions exécuté une instruction qui n'a pas de sens. Vous noterez donc qu'il est **interdit de faire figurer du coté droit d'une affectation une variable non initialisée**. Vous remarquez que l'instruction $D \longleftarrow A$ affecte à D la valeur de A, et que l'affectation $A \longleftarrow C + 1$ n'a pas de conséquence sur la variable D. Les deux variables A et D correspondent à **deux emplacements distincts de la mémoire**, modifier l'une n'affecte pas l'autre.

1.1.3 Littéraux

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

— numérique : $1, 2, 0, -4, \ldots$

— alphanumérique : "toto", "toto01", "04", ...

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

1.1.4 Convention d'écriture

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

— Le **titre**, tout algorithme porte un titre. Choisissez un titre qui permet de comprendre ce que fait l'algorithme

— La **déclaration de variables**, vous préciserez dans cette partie quels noms vous avez décidé de donner à vos variables et de quel type est chacune d'elle.

— Les **instructions**, aussi appelé le corps de l'algorithme, cette partie contient notre succession d'instructions.

Par exemple,

Algorithme : Exemple d'algorithme
Variables :
numériques : A, B, C
alphanumériques : t
Début
  $A \longleftarrow 1$
  $B \longleftarrow A + 1$
  $C \longleftarrow A$
  $A \longleftarrow A + 1$
  $t \longleftarrow$"this algorithm is over"
Fin

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

1.1.5 Entrées-sorties

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

Saisie

Il est possible de demander à un utilisateur du programme de **saisir** une valeur :

Saisir < *nomvariable* >

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

Saisir *A, B, C*

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

Affichage

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

Afficher < *message* >

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

Afficher *"Hello World"*

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

Afficher *A*

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

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

ont le même effet que l'instruction

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

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

Exemple

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

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

1.1.6 Types numériques et alphanumériques

Types numériques

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

— `réel`, que nous utiliserons pour représenter des nombres réels.

Types alphanumériques

Nous affinerons aussi les types alphanumériques en leur substituant les deux types suivants :
— `caractère`, qui est un type permettant de représenter un symbole, et un seul.
— `chaîne`, que nous utiliserons lorsque l'on voudra représenter zéro, un ou plusieurs caractères.
Les littéraux de type caractères seront délimités par des simples quotes (apostrophes) et les chaînes de caractères seront délimitées par des double-quotes (guillemets).

1.1.7 Algobox

Le logiciel **algobox** permet de tester les algorithmes. Vous êtes invités à vous le procurer sur le site d'algobox : `http://www.xm1math.net/algobox/`.

1.2 Traitements conditionnels

On appelle traitement conditionnel une portion de code qui n'est pas exécutée systématiquement.

1.2.1 `SI ... ALORS`

La syntaxe d'un traitement conditionnel est la suivante :

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

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

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

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

Conditions

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

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

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

Si étendu

Le traitement conditionnel peut être étendue de la sorte :

Si < *condition* > **alors**
| < *instructions* >
Sinon
| < *autresinstructions* >
Fin si

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

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

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

Imbrication

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

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

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

Connecteurs logiques

Les connecteurs logiques permettent de d'évaluer des conditions plus complexes. Deux sont disponibles :
— **et** : la condition `<condition1> et <condition2>` est vérifiée si les deux conditions `<condition1>` et `<condition2>` sont vérifiées simultanément.
— **ou** : la condition `<condition1> ou <condition2>` est vérifié si au moins une des deux conditions `<condition1>` et `<condition2>` est vérifiée.

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

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

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

1.2.2 Suivant cas

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

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

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

Exemple

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

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

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

1.2.3 Variables Booléennes

Une variable de type **booléen** ne peut contenir que les valeurs **vrai** et **faux**.
Il serait tout à fait judicieux de se demander quel est l'intérêt d'un tel type ? Observons tout d'abord cet exemple :

```
Algorithme : Lendemain
Variables :
booléen : ok
chaînes : jour, lendemain
Début
    Afficher "Saisissez un jour de la semaine"
    Saisir jour
    ok ⟵ vrai
    Suivant jour faire
        Cas "lundi" faire : lendemain ⟵ "mardi"
        Cas "mardi" faire : lendemain ⟵ "mercredi"
        Cas "mercredi" faire : lendemain ⟵ "jeudi"
        Cas "jeudi" faire : lendemain ⟵ "vendredi"
        Cas "vendredi" faire : lendemain ⟵ "samedi"
        Cas "samedi" faire : lendemain ⟵ "dimanche"
        Cas "dimanche" faire : lendemain ⟵ "lundi"
        autres cas faire ok ⟵ faux
    Fin
    Si ok alors
        Afficher "Le lendemain du ", jour, " est ", lendemain, "."
    Sinon
        Afficher "Erreur de saisie"
    Fin si
Fin
```

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

1.3 Boucles

Nous souhaitons créer un programme nous affichant tous les nombres de 1 à 5, donc dont l'exécution serait la suivante :

```
1 2 3 4 5
```

Une façon particulièrement vilaine de procéder serait d'écrire 10 `sysout` successifs, avec la laideur des copier/coller que cela impliquerait. Nous allons étudier un moyen de mettre au point ce type de programme avec un peu plus d'élégance.

1.3.1 Définitions et terminologie

Une **boucle** permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou plus informellement un **passage** dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on **rentre** dans la boucle, lorsque la dernière itération est terminée, on dit qu'on **sort** de la boucle.

Il existe trois constructions de boucles :

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

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

1.3.2 Tant que

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

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

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

Algorithme : 1 à 5 Tant que
Variables :
entier : i
Début
| $i \longleftarrow 1$
| **Tant que** $i \leq 5$
| | **Afficher** i
| | $i \longleftarrow i + 1$
| **Fin tant que**
Fin

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

Lorsqu'une boucle fonctionne avec un compteur, n'oubliez pas :

— D'initialiser le compteur avant d'entrer dans la boucle
— D'incrémenter le compteur à la fin du corps
— De contrôler la valeur du compteur dans la conditon de boucle

1.3.3 Répéter ... jusqu'à

Répéter
| < *instructions* >
Jusqu'à < *condition* >

Le fonctionnement est analogue à celui de la boucle `tant que` à quelques détails près :
— la condition est évaluée **après** chaque passage dans la boucle.
— On exécute le corps de la boucle jusqu'à ce que la condition soit vérifiée, donc tant que la condition est fausse
— On passe toujours **au moins une fois** dans une boucle `répéter jusqu'à`

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

Variables :
entier : i
Début
| $i \longleftarrow 1$
| **Répéter**
| | **Afficher** i
| | $i \longleftarrow i + 1$
| **Jusqu'à** $i > 5$
Fin

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

Répéter
| **Afficher** *"Saisir un nombre strictement positif"*
| **Saisir** i
| **Si** $i \leq 0$ **alors**
| | **Afficher** *"J'ai dit STRICTEMENT POSITIF !"*
| **Fin si**
Jusqu'à $i > 0$

1.3.4 Pour

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

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

< *variable* >$\longleftarrow$< *premierevaleur* >
Tant que < *variable* >$<>$< *dernierevaleur* > + < *pas* >
| < *instructions* >
| < *variable* >$\longleftarrow$< *variable* > + < *pas* >
Fin tant que

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

Algorithme : 1 à 5 Pour
Variables :
entier : i
Début
  Pour i *allant de* 1 *à* 5
    Afficher i
  Fin pour
Fin

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

1.4 Tableaux

Considérons un algorithme dont l'exécution donnerait :

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

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

1.4.1 Définition

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

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

1.4.2 Déclaration

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

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

Par exemple,

$entier : T[4]$

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

1.4.3 Accès aux éléments

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

1.4.4 Exemple

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

$entier : E[10]$

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

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

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

Pour i *allant de* 1 *à* 10
| **Saisir** $E[i]$
Fin pour

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

Afficher *"Saisissez une valeur : "*
Saisir t

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

$i \longleftarrow 1$
Tant que $E[i] <> t$
| $i \longleftarrow i + 1$
Fin tant que

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

$i \longleftarrow 1$
Tant que $i <= 10$ *et* $E[i] <> t$
| $i \longleftarrow i + 1$
Fin tant que

Il existe donc deux façons de sortir de la boucle :
— En cas de débordement d'indice, la condition `i <= 10` ne sera pas vérifiée. Une fois sorti de la boucle, `i` aura la valeur 11.
— Dans le cas où `t` se trouve dans le tableau à l'indice `i`, alors la condition `E[i] <> t` ne sera pas vérifiée et on sortira de la boucle. Un fois sorti de la boucle, `i` aura comme valeur l'indice de l'élément de `E` qui est égal à `t`, donc une valeur comprise entre 1 et 10.

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

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

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

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

1.5 Sous-Programmes

Dès qu'un programme nécessite une grande quantité de code, on se retrouve vite dans une situation où le code ressemble à un pâté. Posant ainsi de nombreux problèmes lorsqu'il s'agit de trouver des erreurs ou d'effectuer des modifications. Existe-t-il des moyens d'organiser ses instructions et de subdiviser un programme en plusieurs petits programmes ?

Nous allons utiliser des **sous-programes** pour subdiviser un programme. Il en existe deux types :

— Les **procédures**
— Les **fonctions**

1.5.1 Les procédures

Définition

Une **procédure** est un ensemble d'**instructions** portant un **nom**
La syntaxe permettant d'en créer une est la suivante :

Procédure *nomprocedure()*
| corps de la procédure
Fin procédure

Par exemple,

Procédure *afficheBonjour()*
| **Afficher** *"Bonjour !"*
Fin procédure

Appel

On **exécute** une procédure en utilisant son nom.
On dit aussi que l'on **appelle** la procédure, ou encore qu'on l'**invoque**.
Par exemple, l'instruction.

afficheBonjour()

exécute la procédure `afficheBonjour`, on dit aussi qu'on appelle la procédure `afficheBonjour`. Ainsi il existe deux façons de rédiger le même algorithme :
Sans procédure

Algorithme : Affiche bonjour
Début
| **Afficher** *"Bonjour !"*
Fin

et avec une procédure

Algorithme : Affiche bonjour
Procédure *afficheBonjour()*
| **Afficher** *"Bonjour !"*
Fin procédure
Début
| $affichеBonjour()$
Fin

Exemple

Vous pouvez définir autant de procédures que vous le voulez et vous pouvez appeler des procédures depuis des procédures :

```
Algorithme : Affiche bonjour
Procédure afficheBonjour()
| Afficher "Bonjour !"
Fin procédure
Procédure afficheAuRevoir()
| Afficher "Au revoir !"
Fin procédure
Procédure afficheBonjourEtAuRevoir()
| afficheBonjour()
| afficheAuRevoir()
Fin procédure
Début
| afficheBonjourEtAuRevoir()
Fin
```

1.5.2 Variables locales

Définition

Une **variable locale** est une variable déclarée dans une procédure.

On peut par exemple réaliser une procédure affichant les dix premiers nombres entiers en déclarant un compteur dans la procédure :

```
Procédure afficheUnADix()
| entier : i
| Pour i ⟵ 1 à 10
| | Afficher i
| Fin pour
Fin procédure
```

Visibilité

Attention : une procédure ne peut pas utiliser les variables locales d'une autre procédure. Par exemple, **ceci est mal** :

```
Procédure afficheUn()
| entier : i
| j ⟵ 1
| Afficher j
Fin procédure
Procédure afficheDeux()
| entier : j
| i ⟵ 2
| Afficher i
Fin procédure
```

Identificateur

Un **identificateur** est un nom choisi par le programmeur.

— Les noms des variables sont des identificateurs parce qu'ils choisis par le programmeur pour représenter des variables
— Les noms des procédures sont aussi des identificateurs parce qu'ils sont choisis par le programmeur.

Mots-clés

Tout ce qui n'est pas choisi par le programmeur entre dans la catégorie suivante : Un **mot-clé** (ou **mot réservé**) est un mot imposé par le langage de programmation.

Par exemple,

Les mots *Si*, fin, *alors procedure...* sont des mots-clés. Vous vous douterez que les mots réservés le sont parce qu'il est impossible de les choisir comme nom de variable.

1.5.3 Passage de paramètres

Définitions

— Une procédure ne peut pas utiliser les variables locales d'une autre procédure
— Mais il existe un moyen de contourner cette restriction
— Les procédures se transmettent des informations avec des **passages de paramètre**.

Il y a **passage de paramètre** lorsque la procédure appelante transmet une information à la procédure appelée.

L'exemple suivant permet d'afficher une valeur déterminée en dehors de la procédure :

```
Procédure afficheValeur(entier : x)
  | Afficher "La valeur de l'entier passé en paramètre est ", x
Fin procédure
Début
  | afficheValeur(4)
Fin
```

Dans l'exemple ci-avant, la valeur 4 est **recopiée** dans la variable x puis est affichée par la procédure $afficheValeur$. Cet extrait affiche donc la valeur 4.

Passage de paramètre par valeur

On aurait tout à fait pu utiliser une variable à la place du 4 :

```
Procédure afficheValeur(entier : x)
  | Afficher "La valeur de l'entier passé en paramètre est ", x
Fin procédure
Procédure afficheQuatre()
  | entier : titi
  | titi ⟵ 4
  | afficheValeur(titi)
Fin procédure
```

Attention : *titi* et x sont deux variables différentes ! Modifier l'une n'a aucun effet sur l'autre. Certes *titi* est recopiée dans x lors de l'appel de la procédure $afficheValeur$, mais il ne s'agit de rien de plus que d'une affectation entre ces deux variables.

Paramètres formels et effectifs

Pour éviter toute confusion entre *titi* et x, nous utiliserons la terminologie suivante :
— Les paramètres déclarés dans l'entête de la procédure sont les **paramètres formels.**
— Les paramètres utilisés lors de l'appel de la procédure sont les **paramètres effectifs.**

Dans l'exemple suivant, $formel$ est le paramètre formel et $effectif$ le paramètre effectif.

```
Procédure afficheValeur(entier : formel)
| Afficher "La valeur de l'entier passé en paramètre est ", formel
Fin procédure
Procédure afficheQuatre()
| entier : effectif
| effectif ⟵ 4
| afficheValeur(effectif)
Fin procédure
```

Passage de plusieurs paramètres

Il est possible de passer plusieurs valeurs en paramètre :

```
Procédure afficheSomme(entier : x, y)
| Afficher x + y
Fin procédure
```

Cette procédure peut s'appeller de la façon suivante :

```
afficheSomme(A, B)
```

Lors de cet appel, A est recopié dans x et B est recopié dans y.

1.5.4 Passage de paramètres par référence

Exemple

Tentons d'échanger les valeurs de deux variables dans une procédure :

```
Procédure echange(entier : x, y)
| entier : temp
| temp ⟵ x
| x ⟵ y
| y ⟵ temp
Fin procédure
```

On peut appeler cette procédure de la façon suivante :

```
echange(A, B)
```

L'exécution de ce sous-programme laisse les valeurs de A et B inchangées :
— x et y sont des copies de A et B
— Les valeurs des **copies** (x et y) sont echangées
— Mais les **originaux** (A et B) ne sont pas affectés par cette modification.

Alias

Deux identificateurs de variables i et j sont des **alias** s'ils représentent la même variable.
Autrement dit, les valeurs de i et j sont liées, dans le sens ou modifier l'une revient aussi à modifier l'autre.
Dans le cas de l'échange de deux variables, pour que des modifications sur x et y soient répercutées sur A et B, il faudrait que x soit un **alias** de A et que y soit un **alias** de B. Pour ce faire, nous allons utiliser un passage de paramètre par référence :

Définition

Il y a passage de paramètre par :
— **valeur** lorsque le paramètre effectif est recopié dans le paramètre formel.
— **référence** lorsque le paramètre effectif est un alias du paramètre formel.

Lors d'un **passage de paramètres par référence**, on transmet non pas les valeurs des variables, mais les variables elles-mêmes.

Passage de paramètres par valeur :

```
Procédure echange(entier : x, y)
    entier : temp
    temp ⟵ x
    x ⟵ y
    y ⟵ temp
Fin procédure
Début
    entier : a, b
    a ⟵ 1
    b ⟵ 2
    echange(a, b)
    Afficher a, b ;
Fin
```

Passage de paramètres par référence.

```
Procédure echange(entier : x e/s, y e/s)
    entier : temp
    temp ⟵ x
    x ⟵ y
    y ⟵ temp
Fin procédure
Début
    entier : a, b
    a ⟵ 1
    b ⟵ 2
    echange(a, b)
    Afficher a, b ;
Fin
```

— a et x sont des alias
— b et y sont des alias

Conventions

— Par défaut, un paramètre est passé par valeur
— Un paramètre est passé par référence s'il porte l'indication **e/s**

On résumera les choses ainsi :
— La transmission de valeur depuis la procédure appelante vers la procédure appelée se fait avec des passages de paramètres.
— La transmission de valeur depuis la procédure appelée vers la procédure appelante se fait avec des passages de paramètres par référence.

Effet de bord

— Lorsqu'une procédure modifie les valeurs de variables passées par référence, il se produit ce que l'on appelle **effet de bord**

— Plus un programme contient d'effets de bords, plus il est difficile de trouver les erreurs, le maintenir et le faire évoluer
— N'utilisez donc les passages par référence que lorsque vous ne pouvez pas faire autrement.

1.5.5 Fonctions

Définition

Il existe un autre mécanisme permettant au sous-programme appelé de transmettre une information au sous-programme appelant : celui des **valeurs de retour**. Une **fonction** est un sous-programme fait pour transmettre une unique valeur au sous-programme appelant. On dit que cette valeur est **retournée**.

Fonction $Un()$: *entier*
| **Retourner** 1
Fin fonction

La fonction $Un()$ transmet 1 au sous-programme appelant.
Une fonction se déclare

Fonction $nom(parametres)$: $typeDeRetour$
| ...
Fin fonction

On retourne une valeur avec l'instruction

Retourner *valeur*

La fonction *carre* prend en paramètre une valeur x et retourne la valeur x^2.

Fonction $carre(entier : x)$: *entier*
| **Retourner** $x * x$
Fin fonction

Appel

On peut appeler une fonction de plusieurs façons :

$y \longleftarrow carre(x)$

Afficher x, " * ", x, " = ", $carre(x)$

Afficher x, "^4 = ", $carre(carre(x))$

On remarque que l'utilisation des fonctions est davantage souple que celle des procédures :
— on peut imbriquer des appels de fonctions dans des expressions
— ce que l'on ne peut pas faire avec des procédures, qui elles sont des instructions.

Parallèle avec les mathématiques

La notion de fonction en tant que sous-programme est à mettre en parallèle avec celle de fonction mathématique.

Fonction $f(entier : x)$: *entier*
| **Retourner** $3 * x + 1$
Fin fonction

La fonction $f(x) = 3x + 1$ est une correspondance entre une valeur quelconque x et celle que l'on obtient en multipliant x par 3 et en lui ajoutant 1.

On l'appelle :

Afficher *"L'image de* 4 *par* f *est "*, $f(4)$, *"."*

On peut considérer que l'**antécédent** est x et que la valeur retournée est l'**image** de la fonction.

Effets de bords

— La seule chose qui nous intéresse avec une fonction est la valeur qu'elle retourne
— donc on **évitera les effets de bords**
— et l'on préfèrera des passages de paramètres par valeur dans les fonctions.

1.6 Matrices

1.6.1 Définition

Une matrice est un tableau à deux dimensions, c'est à dire un regroupement de variables de même type auxquelles on accède à l'aide de deux indices.

$$\begin{pmatrix} 1 & 4 & 2 \\ 2 & 5 & -3 \\ 4 & 7 & 15 \\ 2 & 1 & 6 \end{pmatrix}$$

La matrice ci-dessus est une matrice 4×3, parce qu'elle est formée de 4 lignes et de 3 colonnes. On accède à une des variables à l'aide de la notation à deux indices $t[i, j]$, où t est le nom de la matrice, i l'indice de la ligne et j l'indice de la colonne. Par exemple, la valeur se trouvant dans la variable $t[2, 3]$ est -3, parce que -3 de trouve à la deuxième ligne et à la troisième colonne.

1.6.2 Déclaration

Plusieurs syntaxes existent pour déclarer des matrices, dans ce cours nous utiliserons :

$< type >:< nom > [< nombredelignes >, < nombredecolonnes >]$

Par exemple, pour déclarer une matrices de booléens à 10 lignes et 20 colonnes se nommant w, on écrira :

$booleen : w[10, 20]$

1.6.3 Parcours

Comme une matrice se manipule avec deux indices, il est nécessaire d'utiliser deux boucles imbriquées pour en parcourir une. Par exemple :

```
entier : mat[5, 7]
Pour i ⟵ 1 à 5
|   Pour j ⟵ 1 à 7
|   |   Afficher mat[i, j]
|   Fin pour
Fin pour
```

Chapitre 2

Exercices

2.1 Introduction

2.1.1 Affectations

Exercice 1 Jeu d'essai

Après les affectations suivantes :

$A \longleftarrow 1$
$B \longleftarrow A + 1$
$A \longleftarrow B + 2$
$B \longleftarrow A + 2$
$A \longleftarrow B + 3$
$B \longleftarrow A + 3$

Quelles sont les valeurs de A et de B ?

Exercice 2 Permutation des valeurs de 2 variables

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

2.1.2 Saisie, affichage, affectations

Exercice 3 Nom et âge

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

Exercice 4 Permutation de 2 variables saisies

Saisir deux variables et les permuter avant de les afficher.

Exercice 5 Moyenne de 3 valeurs

Saisir 3 valeurs, afficher leur moyenne.

Exercice 6 Aire du rectangle

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

Exercice 7 Permutation de 4 valeurs

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

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

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

Exercice 8 Permutation de 5 valeurs

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

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

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

Exercice 9 Permutation ultime

Même exercice avec :

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

Exercice 10 Pièces de monnaie

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

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

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

2.2 Traitements conditionnels

2.2.1 Exercices de compréhension

Exercice 1 Jeu d'essai

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

$A \longleftarrow 1$
$B \longleftarrow 3$
Si $A \geq B$ **alors**
  $A \longleftarrow B$
Sinon
  $B \longleftarrow A$
Fin si

Exercice 2 Instructions à compléter

Pourriez-vous compléter les instructions suivantes ?

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

2.2.2 Conditions simples

Exercice 3 Majorité

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

Exercice 4 Valeur absolue

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

Exercice 5 Admissions

Saisir une note, afficher "ajourné" si la note est strictement inférieure à 8, oral entre 8 et 10, admis si la note est au moins égale à 10.

Exercice 6 Assurances

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

Exercice 7 Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 8 Recherche de doublons

Écrire un algorithme demandant à l'utilisateur de saisir trois valeurs et lui disant s'il s'y trouve un doublon.

Exercice 9 Tri de 3 valeurs

Écrire un algorithme demandant à l'utilisateur de saisir 3 valeurs et les affichant dans l'ordre croissant.

2.2.3 Conditions imbriquées

Exercice 10 Signe du produit

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

Exercice 11 Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs distinctes.

Exercice 12 $ax + b = 0$

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

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

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

Exercice 14 Opérations sur les heures

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

— heures de début
— minutes de début
— heures de fin
— minutes de fin

Exercice 15 Le jour d'après

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

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

2.2.4 L'échiquier

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

Exercice 16 Couleurs

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

Exercice 17 Cavaliers

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

Exercice 18 Autres pièces

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

2.2.5 Suivant Cas

Exercice 19 Calculatrice

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

2.3 Boucles

2.3.1 Utilisation de toutes les boucles

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

Exercice 1 Compte à rebours

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

Exercice 2 Factorielle

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

Exercice 3 Choix de boucles

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

2.3.2 Choix de la boucle la plus appropriée

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

Exercice 4 Table de multiplication

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

Exercice 5 Puissance

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

Exercice 6 Somme des entiers

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

Exercice 7 Nombres premiers

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

Exercice 8 Somme des inverses

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

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

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

Exercice 9 n^n

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

Exercice 10 Racine carrée par dichotomie

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

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

2.4 Tableaux

Exercice 1 Initialisation et affichage

Ecrire un algorithme plaçant les valeurs $2, 3, 4, \ldots, 8$ dans un tableau T à 7 éléments, puis affichant les éléments de T en partant de la fin.

Exercice 2 Contrôle de saisie

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

Exercice 3 Choix des valeurs supérieures à t

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

Exercice 4 Somme

Quelles instructions permettent de calculer la somme des éléments d'un tableau.

Exercice 5 Permutation circulaire

Quelles instructions permettent d'effectuer une permutation circulaire des éléments d'un tableau vers la droite ?

Exercice 6 Miroir

Quelles instructions permettent d'inverser l'ordre des éléments d'un tableau.

Exercice 7 Minimum

Quelles instructions permettent d'afficher le plus petit élément d'un tableau.

2.5 Sous-programmes

2.5.1 Procédures

Exercice 1 Carré

Écrire une procédure affichant le carré du nombre passé en paramètre. Utilisez-là pour afficher les carrés des entiers de 1 à 10.

Exercice 2 Volume du parallélépipède

Écrire une procédure affichant le volume du parallélépipède dont les dimensions sont passées en paramètre. Utilisez-là pour afficher le volume d'un parallélépipède dont les dimensions sont saisies par l'utilisateur.

Exercice 3 Plus grande valeur

Écrire une procédure demandant à l'utilisateur de saisir une suite de valeurs se terminant par un nombre négatif, puis lui affichant la plus grande valeur saisie.

2.5.2 Fonctions

Exercice 4 Carré

Écrire une fonction retournant le carré du nombre passé en paramètre. Utilisez-là pour afficher les carrés des entiers de 1 à 10.

Exercice 5 Volume du parallélépipède

Écrire une fonction retournant le volume du parallélépipède dont les dimensions sont passées en paramètre. Utilisez-là pour afficher le volume d'un parallélépipède dont les dimensions sont saisies par l'utilisateur.

Exercice 6 Plus grande valeur

Écrire une fonction demandant à l'utilisateur de saisir une suite de valeurs se terminant par un nombre négatif, puis lui affichant la plus grande valeur saisie.

2.5.3 Analyse combinatoire

Exercice 7 Factorielle

On note $n!$ le nombre $1.2.3.4 \dots (n-1).n$. Par exemple, $5! = 1.2.3.4.5 = 120$ et on a par convention $0! = 1$. Écrivez une fonction $factorielle(n : entier) : entier$ retournant la factorielle du nombre n passé en paramètre.

Exercice 8 Puissance

Écrivez une fonction $puissance(b, n : entier) : entier$ retournant b^n, où b est un entier non nul et n un entier positif. N'oubliez pas que $b^0 = 1$.

Exercice 9 Arrangements

On note $\mathcal{A}_n^p$ le nombre $(n-p+1)(n-p+2)\dots(n-1)n$. Par exemple, $\mathcal{A}_6^4 = 3.4.5.6 = 360$. Écrivez une fonction $arrrangements(p, n : entiers) : entier$ retournant $\mathcal{A}_n^p$ si $p \leq n$ et -1 sinon.

Exercice 10 Combinaisons

On note $\mathcal{C}_n^p$ le nombre $\dfrac{(n-p+1)(n-p+2)\dots(n-1)n}{1.2\dots(p-1).p}$. Par exemple, $\mathcal{C}_6^4 = \frac{3.4.5.6}{1.2.3.4} = 15$. Écrivez une fonction $combinaisons(p, n : entiers) : entier$ retournant $\mathcal{C}_n^p$ si $p \leq n$ et -1 sinon.

2.5.4 Sous-programmes et tableaux

Exercice 11 Appartenance

Ecrire la fonction $contient(T[N], x : entiers) : boolen$. $contient(T, x)$ retourne vrai si et seulement si le tableau T contient l'élément x.

Exercice 12 Substitution

Ecrire la procédure $remplace(T[N], x, y : entiers)$. $remplace(T, x, y)$ remplace dans T toutes les occurrences de x par des y.

Exercice 13 Valeurs distinctes

Ecrire la fonction $valeursDistinctes(T[n] : entier) : entier$. Cette fonction retourne le nombre de valeurs distinctes contenues dans le tableau T, c'est-à-dire le nombre de valeurs contenues dans T une fois tous les doublons supprimés. Par exemple, les valeurs distinctes de $[2, 4, 8, 2, 7, 3, 8, 2, 5, 0, 8, 4, 1]$ sont $\{2, 4, 8, 7, 3, 5, 0, 1\}$, le nombre de valeurs distinctes est donc 8.

Exercice 14 Passage par référence

Ecrire la procédure $carres(T[n] : entiers)$. Cette procédure place dans T les carrés dans n premiers nombres entiers. Vous utiliserez le fait que k^2 est la somme des k premiers entiers impairs. Par exemple, $3^2 = 1 + 3 + 5 = 9$, $5^2 = 1 + 3 + 5 + 7 + 9 = 25$, etc.

Exercice 15 Identités remarquables

Ecrire la procédure $cubes(T[n] : entiers)$. Cette procédure place dans T les cubes dans n premiers nombres entiers. Vous utiliserez la procédure $carrs$ et le fait que $k^3 = (k-1)^3 + 3(k-1)^2 + 3(k-1) + 1$. Par exemple, si $k = 3$, on a $3^3 = (2)^3 + 3(2)^2 + 3(2) + 1 = 8 + 12 + 6 + 1 = 27$.

Exercice 16 Interclassement

Ecrire la procédure $interclasse(S[n], T[p], Q[n + p] : entiers)$ cette fonction prend deux tableaux triés S et T en paramètre. Elle place dans Q les éléments de S et de T dans l'ordre croissant. Si par exemple, $S = [1, 4, 6, 8, 23, 54]$ et $T = [2, 3, 7, 12, 17, 45, 52, 77]$, alors $interclasse$ place $[1, 2, 3, 4, 6, 7, 8, 12, 17, 23, 45, 52, 54, 77]$ dans Q.

2.6 Matrices

2.6.1 Opérations sur les matrices

Exercice 1 Somme

Écrire une procédure calculant la somme de deux matrices.

Exercice 2 Transposition

Écrire une procédure calculant la matrice transposée d'une matrice A de dimensions $(n \times p)$ de deux façons :

1. En plaçant dans une matrice B la transposée de A.
2. En modifiant A de sorte qu'elle devienne sa propre transposée.

Exercice 3 Échange de lignes

Écrire une procédure échangeant deux lignes d'une matrice.

Exercice 4 Multiplication

Écrire une procédure calculant le produit de deux matrices.

Exercice 5 Plus grande colonne

Écrire une calculant l'indice de la colonne dont la somme des valeurs est la plus élevée.

2.6.2 Morceaux choisis

Exercice 6 Triangle de Pascal

Un triangle de Pascal peut être placé dans une matrice, dont seule la partie triangulaire inférieure est renseignée. La première ligne et la première colonne d'une triangle de Pascal ne contiennent que des 1. Et, si on note $P(i, j)$ la valeur se trouvant dans la i-ème ligne et la j-ème colonne de cette matrice, alors on a

$$m(i, j) = m(i - 1, j - 1) + m(i - 1, j)$$

pour tous i et j supérieurs ou égaux à 1. Écrire une procédure initialisant un triangle de Pascal à n lignes.

Exercice 7 Puissances

Écrire une procédure remplissant une matrice m de la façon suivante :

$$m(i, j) = i^{j-1}$$

Vous utiliserez le fait que

$$i^j = P(0, j)(i - 1)^0 + P(1, j)(i - 1)^1 + \ldots + P(k, j)(i - 1)^k + \ldots + P(j, j)(i - 1)^j$$

où $P(a, b)$ est l'élément se trouvant dans la ligne $b+1$ et la colonne $a+1$ du triangle de Pascal calculé dans l'exercice précédent.

Exercice 8 Matrices de Toepliz

Soit $\mathcal{M}$ une matrice à n lignes et à p colonnes, on note $m(i,j)$ l'élément de $\mathcal{M}$ de trouvant à la i-ème ligne et à la j-ème colonne. $\mathcal{M}$ est une matrice de Toepliz si pour tout $i \in \{2,\ldots,n\}$ et pour tout $j \in \{2,\ldots,p\}$,

$$m(i,j) = m(i-1,j) + m(i,j-1)$$

Par exemple :

$$\begin{pmatrix} 1 & 4 & 2 & 1 & 7 \\ 2 & 6 & 8 & 9 & 16 \\ 1 & 7 & 15 & 24 & 40 \\ -2 & 5 & 20 & 44 & 84 \\ 2 & 7 & 27 & 71 & 155 \end{pmatrix}$$

Écrire la procédure `toepliz(entier : T[m, n] e/s)` prenant en paramètre une matrice dont la première ligne et la première colonne sont initialisées. Cette procédure initialise tous les autres éléments de la matrice de sorte que T soit une matrice de Toepliz.

Exercice 9 Rotation

1. on appelle rotation de matrice l'opération qui transforme la matrice

$$\begin{pmatrix} 1 & 4 & 2 & 7 \\ 2 & 6 & 9 & 16 \\ 1 & 7 & 24 & 40 \end{pmatrix}$$

en

$$\begin{pmatrix} 7 & 16 & 40 \\ 2 & 9 & 24 \\ 4 & 6 & 7 \\ 1 & 2 & 1 \end{pmatrix}$$

Écrire la procédure `rotation(entier : T[m, n], Q[n, m] e/s)` affectant à la matrice `Q` le résultat de la rotation de la matrice `T`.

2. Écrire la procédure `rotation(entier : T[n, n] e/s)`, prenant en paramètre une matrice carrée `T`, et modifiant cette matrice de sorte qu'elle contienne le résultat de sa rotation.

2.7 Récursivité

2.7.1 Sous-programmes récursifs

Tous les sous-programmes dans les exercices ci-dessous doivent être rédigés de façon récursive, ils ne doivent pas contenir de boucle.

Exercice 1 Compte à rebours

Écrire une procédure affichant un compte à rebours en partant du nombre passé en paramètre.

Exercice 2 Somme

Écrire une fonction retournant la somme de deux nombres entiers positifs passés en paramètre sans les additionner. Vous n'utiliserez que des incrémentations et des décrémentations.

Exercice 3 Factorielle

Écrire une fonction retournant la factorielle du nombre passé en paramètre.

Exercice 4 Produit

Écrire une fonction retournant le produit de deux nombres entiers passés en paramètre sans les multiplier.

Exercice 5 Série arithmétique

Écrire une fonction retournant la somme des n premiers entiers (où n est passé en paramètre).

Exercice 6 Puissance

Écrire une fonction retournant b^n (où b et n sont tous deux passés en paramètre).

Exercice 7 Nombre de chiffres

Écrire une fonction retournant le nombre de chiffres d'un entier n passé en paramètre. *Ne pas convertir le nombre en chaîne de caractères !*

Exercice 8 p-ème chiffre

Écrire une fonction retournant le p-ième chiffre d'un entier n en partant de la droite. Si par exemple $n = 13489765123$ et $p = 4$, alors la fonction retourne 5 car il s'agit du 4 chiffre de n en partant des unités.

Exercice 9 Somme des chiffres

Écrire une fonction retournant la somme des chiffres d'un entier n passé en paramètre.

2.7.2 Morceaux choisis

Exercice 10 Recherche par dichotomie

Écrire une fonction $recherche(t[], x, i, j : entier) : booleen$ retournant vrai si et seulement si il existe un élément du tableau t qui soit égal à x et dont l'indice se trouve entre i et j. On suppose que le tableau t est trié.

Exercice 11 Suite de Fibonacci

Écrire une fonction permettant de calculer le n-ème terme de la suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$ et $u_n = u_{n-1} + n_{n-2}$. Est-ce efficace pour de grandes valeurs de n ? Est-il possible de rédiger un algorithme itératif ayant de meilleures performances ?

Exercice 12 Tri fusion

Écrire une procédure appliquant la méthode du tri fusion à un tableau passé en paramètre.

Exercice 13 Belle marquise...

Écrire une procédure affichant toutes les permutations de *Belle marquise, vos beaux yeux me font mourir d'amour.*

2.8 Révisions

2.8.1 Problèmes divers

Les corrigés sont visibles sur `http://github.com/alexandreMesle/U22_Python/`.

Exercice 1 - Nombres impairs

Nous nous intéressons ici à la somme des n premiers nombres impairs.

Question 1 (Exemple) *Quelle est la somme des* 4 *premiers nombres impairs ?*

Question 2 (Pseudo-code) *Écrire une fonction* `sommeImpairs(n : entier) : entier` *retournant la somme des n premiers nombres impairs.*

Question 3 (Programmation) *Programmer cet algorithme.*

Question 4 (Test) *Testez-le avec tous les n de* 1 *à* 20.

Question 5 (Conclusion) *Qu'observez-vous ?*

Exercice 2 - Interpolation

Étant donnés deux points $A(x_A, y_A)$ et $B(x_B, y_B)$, comment calculer l'équation $y = ax + b$ de la droite Δ passant par les points A et B ?

Question 1 (Exemple) *Quelle est l'équation de la droite passant par les points $A(0,1)$ et $B(1,2)$?*

Question 2 (Coefficient directeur) *Exprimer a en fonction des coordonnées x_A, x_B, y_A et y_B de A et B.*

Question 3 (Intersection de Δ avec les ordonnées) *Exprimer b en fonction de a et des coordonnées de A.*

Question 4 (Pseudo-code) *Écrire deux fonctions* `interpoleA(xA, yA, xB, yB : réel) : réel` *et* `interpoleB(a, xA, yA : réel) : réel` *retournant les valeurs de a et de b.*

Question 5 (Programmation) *Construire un programme demandant la saisie de x_A, x_B, y_A et y_B, et affichant les valeurs de a et b.*

Question 6 (Test) *Testez-le avec les points $A(0,1)$ et $B(1,2)$.*

Question 7 (Cas particuliers) *Quels cas particuliers peuvent mettre cet algorithme en échec ? Corrigez le programme pour les prendre en compte.*

Exercice 3 - Réglage d'horloge

Une horloge digitale se règle avec deux boutons : un pour passer à l'heure suivante, et un pour passer à la minute suivante. On règle donc l'horloge par pressions successives sur ces deux boutons. Si par exemple l'horloge affiche 11 : 59 et que l'utilisateur presse le bouton des minutes, alors l'horloge sera réglée sur 11 : 00.

Question 1 (Exemple) *Si l'horloge affiche* 11 : 22*, combien de pressions doivent être effectuées pour passer à* 13 : 15 *?*

Question 2 (Pseudo-code) *Écrire un algorithme demandant à l'utilisateur la saisie des heures et minutes de deux moments A et B de la journée. L'algorithme devra par la suite déterminer le nombre de pressions pour régler sur l'heure B une horloge initialement réglée sur l'heure A.*

Question 3 (Programmation) *Programmer cet algorithme.*

Question 4 (Distance) *Quels sont les cas dans lesquels le nombre de pressions sera maximal ? Est-ce que l'un d'eux peut partir de* 11 : 22 *?*

Exercice 4 - Horloge et miroir

Si vous regardez une horloge à aguilles sans chiffres dans un miroir, les positions des aiguilles sont disposées à l'envers. Quel calcul faire pour savoir quelle heure il est sans se tromper ? *On supposera que l'aiguille des heures est toujours positionnée de façon exacte sur un des* 12 *chiffres.*

Question 1 (Exemple) *Si l'horloge que vous observez indique* $14:37$*, quelle heure est-il réellement ?*

Question 2 (Pseudo-code) *Écrire une procédure prenant en paramètre les heures et minutes d'un moment de la journée et affichant le reflet de cette heure dans le miroir.*

Question 3 (Programmation) *Programmer cet algorithme.*

Question 4 (Test) *A quels moments de la journée l'horloge affiche-t-elle une heure correcte à travers le miroir ?*

Exercice 5 - Nombres amis

Deux nombres sont amis si chacun est égal à la somme des diviseurs stricts de l'autre.

Question 1 (Pseudo-code) *Écrire un algorithme demandant à l'utilisateur la saisie d'un nombre, lui disant si celui-ci a un nombre ami, et si oui lequel.*

Question 2 (Programmation) *Programmer cet algorithme.*

Question 3 (Test) *Avec quel nombre* 220 *est-il ami ?*

Question 4 (Test) *Avec quel nombre* 6 *est-il ami ?*

Question 5 (Conclusion) *Affichez les 5 plus petits couples de nombres amis.*

Exercice 6 - Clé RIB

La clé d'un RIB (Relevé d'identité bancaire) se calcule de la façon suivante :

$$97 - ((89 * codeBanque + 15 * codeGuichet + 3 * numeroCompte) \ mod \ 97)$$

Question 1 (Pseudo-code) *Écrire un fonction prenant en paramètre des données bancaires et retournant le RIB du compte.*

Question 2 (Programmation) *Programmer cette fonction.*

Question 3 (Test) *Votre clé RIB est-elle correcte ?*

Exercice 7 - Diététique

Un utilisateur possède dans son placard des pâtes en conserves, chaque boîte est numérotée en fonction de la marque. Si par exemple il y a 2 boîtes de Lustucru et 3 boîtes de Barilla, il est possible que les boîtes soient numérotées, 1, 1, 2, 2 et 2. Les numéros choisis sont arbitraires, ce qui importe est que les boîtes aient le même numéro si et seulement si elles sont de la même marque.

Afin de varier les repas, l'utilisateur souhaite éviter de prendre la même boîte de pâtes à deux repas consécutifs. Dans quel ordre consommer les boîtes de pâtes pour ce faire ?

Les numéros des boîtes seront placées dans un tableau passé en paramètre (par référence) à une fonction qui sera chargée d'en modifier l'ordre de sorte que deux numéros identiques ne seront jamais côte à côte.

Nous appliquerons le principe suivant :

— Déterminer la marque apparaissant le plus souvent dans le tableau.

— La permuter avec le premier élément.

On appliquera le même principe à tous les autres éléments. Soit i un indice du tableau :

— Déterminer la marque apparaissant le plus souvent dans le tableau parmi les éléments d'indice supérieur à i, et dont le numéro est différent de celui d'indice $i-1$.
— Le permuter avec l'élément d'indice i.

Question 1 (Exemple) *Appliquez ce principe au tableau suivant :* $[1, 3, 2, 3, 1, 1, 2, 3, 1]$

Question 2 (Pseudo-code) *Écrire en pseudo-code une procédure* `ordonne(t : tableau de n entiers)` *appliquant ce principe au tableau* t.

Question 3 (Programmation) *Programmez cette procédure et testez-là.*

Question 4 (Limites) *Y a-t-il des cas dans lesquels cette procédure ne fonctionne pas ?*

Question 5 (Autre algorithme) *Existe-t-il un meilleur algorithme permettant de résoudre ce problème ?*

Chapitre 3

Quelques corrigés

3.1 Boucles

3.1.1 Compte à rebours

Algorithme : Compte à rebours
Variables : entier : n
Début
  Afficher *"Saisissez une valeur"*
  Saisir n
  Tant que $n \geq 0$
    Afficher n
    $n \longleftarrow n - 1$
  Fin tant que
Fin

ou bien

Répéter
  Afficher n
  $n \longleftarrow n - 1$
Jusqu'à $n < 0$

ou bien

Pour $i \longleftarrow n$ *à* 0 *par pas de* -1
  Afficher n
Fin pour

3.1.2 Factorielle

Algorithme : Factorielle
Variables : entier : n, i, f
Début
  Afficher *"Saisissez une valeur"*
  Saisir n
  $f \longleftarrow 1$
  $i \longleftarrow 2$
  Tant que $i \leq n$
    $f \longleftarrow f * i$
    $i \longleftarrow i + 1$
  Fin tant que
Fin
Afficher *"La factorielle de ", n, " est ", f, "."*

ou bien

$f \longleftarrow 1$
$i \longleftarrow 1$
Répéter
  $f \longleftarrow f * i$
  $i \longleftarrow i + 1$
Jusqu'à $i > n$

ou bien

$f \longleftarrow 1$
Pour $i \longleftarrow 2$ *à* n *pas pas de* 1
  $f \longleftarrow f * i$
Fin pour

3.1.3 Tables de multiplication

Algorithme : Table de multiplication
Variables : entier : n, i
Début
  Afficher *"Saisissez une valeur"*
  Saisir n
  Pour $ilongleftarrow 1$ *à* n
    Afficher i, *" * ", n, " = ", n * i*
  Fin pour
Fin

3.1.4 Puissance

```
Algorithme : Puissance
Variables : entier : b, n, i, r
Début
    Afficher "Saisissez une valeur positive"
    Saisir b
    Répéter
        Afficher "Saisissez une valeur positive"
        Saisir n
    Jusqu'à n ≥ 0
    r ⟵ 1
    Pour i ⟵ 1 à n
        r ⟵ r * b
    Fin pour
    Afficher b, " puissance ", n, " = ", r
Fin
```

3.1.5 Somme des entiers

```
Algorithme : Somme entiers
Variables : entier : somme, i, n
Début
    Afficher "Saisissez une valeur positive"
    Saisir n
    somme ⟵ 0
    Pour i ⟵ 1 à n
        somme ⟵ somme + i
    Fin pour
    Afficher "La somme des ", n, " premiers entiers est ", somme, "."
Fin
```

3.2 Tableaux

3.2.1 Initialisation et affichage

Algorithme : Initialisation et affichage
Variables :
entiers : i, $T[7]$
Début
  Pour i *allant de* 1 *à* 7
    $T[i] \longleftarrow i + 1$
  Fin pour
  Pour i *allant de* 7 *à* 1 *par pas de* -1
    Afficher $T[i]$
  Fin pour
Fin

3.2.2 Contrôle de saisie

Algorithme : Contrôle de saisie
Variables :
entiers : i, $t[7]$
Début
  Afficher *"Saisir* 20 *valeurs positives ou nulles."*
  $i \longleftarrow 1$
  Tant que $i \leq 20$
    Saisir $t[i]$
    Si $t[i] \geq 0$ **alors**
      $i \longleftarrow i + 1$
    Sinon
      Afficher *"Vous avez saisir une valeur négative."*
    Fin si
  Fin tant que
Fin

3.2.3 Choix des valeurs supérieures à t

```
Algorithme : supérieures à t
Variables :
entiers : v[10], t, i, nb
Début
    Pour i allant de 1 à 10
        Afficher "Saisissez une valeur"
        Saisir v[i]
    Fin pour
    Afficher "Saisissez t"
    Saisir t
    nb ⟵ 0
    Pour i allant de 1 à 10
        Si v[i] > t alors
            nb ⟵ nb + 1
        Fin si
    Fin pour
    Afficher "Vous avez saisi ", nb, "valeurs supérieures à ", t, "."
Fin
```

3.2.4 Somme

```
s ⟵ 0
Pour i ⟵ 1 à 20
    s ⟵ s + t[i]
Fin pour
Afficher "La somme des elements du tableau est ", s, "."
```

3.2.5 Permutation circulaire

Pour un tableau t à n éléments :

```
temp ⟵ t[n]
Pour i ⟵ n − 1 à 1 par pas de −1
    t[i + 1] ⟵ t[i]
Fin pour
t[1] ⟵ temp
```

3.2.6 Miroir

Pour un tableau t à n éléments :

```
i ⟵ 1
j ⟵ n
Tant que i < j
    temp ⟵ t[i]
    t[i] ⟵ t[j]
    t[j] ⟵ temp
Fin tant que
```

3.2.7 Minimum

Pour un tableau t à n éléments :

```
min ⟵ t[1]
Pour i ⟵ 2 à n
|  Si t[i] < min alors
|  |  min ⟵ t[i]
|  Fin si
Fin pour
Afficher "La plus petite valeur est ", min
```

3.3 Sous-programmes

3.3.1 Substitution

```
Procédure remplace(T[n], x, y : entier)
    i : entier
    Pour i ⟵ 1 à n
        Si T[i] = x alors
            T[i] ⟵ y
        Fin si
    Fin pour
Fin procédure
```

```
Fonction appartient(T[n], x, i : entier) : booleen
    j : entier
    Pour j ⟵ i à n
        Si T[j] = x alors
            Retourner vrai
        Fin si
    Fin pour
    Retourner faux
Fin fonction
Fonction valeursDistinctes(T[n] : entier) : entier
    i, v : entier
    v ⟵ 0
    Pour i ⟵ 1 à n
        Si non(contient(T, T[i], i + 1)) alors
            v ⟵ v + 1
        Fin si
    Fin pour
    Retourner v
Fin fonction
```