Logique de Programmation et Algorithme I

Tshikutu Anaclet Bikengela

REPUBLIQUE DEMOCRATIQUE DU CONGO

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET UNIVERSITAIRE

Faculté d'Informatique

ALGORITHMIQUE ET LOGIQUE DE PROGRAMMATION

Licence 1

Par Anaclet CIKUTU

Chef de Travaux

Année Académique 2023 - 2024

« N'attribuez jamais à la malveillance ce qui s'explique très bien par l'incompétence. » - Napoléon Bonaparte

« A l'origine de toute erreur attribuée à l'ordinateur, vous trouverez au moins deux erreurs humaines. Dont celle consistant à attribuer l'erreur à l'ordinateur. » - Anonyme

Ce Support s'adresse à toute personne désireuse de maîtriser les bases essentielles de la programmation. Pour apprendre à programmer, il faut d'abord comprendre ce qu'est vraiment un ordinateur, comment il fonctionne et surtout comment il peut faire fonctionner des programmes, comment il manipule et stocke les données et les instructions, quelle est sa logique. Alors, au fur et à mesure, le reste devient évidence : variables, tests, conditions, boucles, tableaux, fonctions, fichiers, jusqu'aux notions avancées comme les pointeurs et les objets.

Introduction Générale

Pour obtenir un résultat donné, il faut généralement suivre une méthode, une certaine logique Sauf à être un grand pâtissier dont la science des mélanges des ingrédients est innée(ou fruit d'une longue pratique), vous n'obtiendrez jamais un délicieux gâteau au chocolat même si vous disposez des meilleurs ingrédients et accessoires de cuisson, si vous ne connaissez pas les bonnes proportions, l'ordre
dans lesquels ajouter les ingrédients, le temps de cuisson, la température : bref, la recette.

De même, sans formation de mécanicien ou sans la documentation technique du moteur de votre véhicule, inutile de vous lancer dans un changement de joint de culasse : c'est la casse as surée. Il en est de même de la programmation.

Il existe plusieurs langages de programmation très simples, extrêmement simples parfois, qui peuvent donner un temps l'illusion que vous savez programmer. En entreprise même, certains employés sont bombardés développeurs pour leurs quelques connaissances confuses de Visual Basi c, de Delphi ou de WinDev. Le résultat risque d'être catastrophique. Les publicités sont alléchantes mais trompeuses.

Les programmeurs, y compris les autodidactes, ont tous à un moment ou un autre eu affaire avec les algorithmes, car il existe en programmation une multitude de moyens d'arriver à un résultat, mais très peu pour obtenir le meilleur résultat possible, ce qui explique pourquoi beaucoup de programmes ayant la même fonction, se ressemblent (au niveau de la programmation) alors que ce ne sont pas les mêmes programmeurs qui les ont développés.

Les débutants qui de lancent dans des projets de programmation audacieux se trouvent bloqués, ne maitrisant pas une technique particulière de la logique de programmation. Certains abandonnent, d'autres trouvent un moyen de contournement (souvent peu reluisant).

Les derniers liront peut-être un Support d'algorithmique comme celui-ci, qui à
défaut de donner une solution complète à leur problème, leur fournira les bases et les techniques pour avancer. C'est ainsi que nous mettons à votre disposition, ces notes de cours pour vous permettre entant que débitant en informatique et en particulier en programmation, les notions de bases pour votre décollage.

Pour se faire, ce cours est scindé en trois chapitres, dont le premier parle de notions introductives d'algorithmes, le second aborde les sous-programmes et tableaux, et le dernier parle de Fichiers, Listes chainées et notions de complexité des algorithmes. Toutes les applications seront testées avec les outils Algobox et LARP à l'ordinateur.

Chapitre 1 : Notions d'Algorithmique et Construction d'ordinogrammes

1.1. Définition d'un algorithme

L'algorithmique est un terme d'origine arabe, comme algèbre, amiral ou zénith. Un algorithme, c'est une suite d'instructions qui, une fois exécutée correctement, conduit à un résultat donné. Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller. Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, cette saloperie de répondeur ne veut rien savoir.

Complétons toutefois cette définition. Après tout, en effet, si l'algorithme, comme on vient de le dire, n'est qu'une suite d'instructions menant celui qui l'exécute à résoudre un problème, pourquoi ne pas donner comme instruction unique : « résous le problème », et laisser l'interlocuteur se débrouiller avec ça ? A ce tarif, n'importe qui serait champion d'algorithmique sans faire aucun effort. Pas de ça, ce serait trop facile.

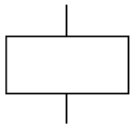
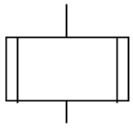
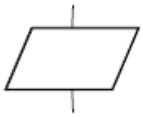
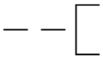
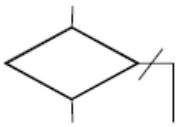
Pour fonctionner, un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter. C'est d'ailleurs l'un des points délicats pour les rédacteurs de modes d'emploi : les références culturelles, ou lexicales, des utilisateurs, étant variables, un même mode d'emploi peut être très clair pour certains et parfaitement abscons pour d'autres.

1.2. Représentation d'un algorithme

Historiquement, il existe deux façons pour représenter un algorithme:

- ✓ L'Ordinogramme: représentation graphique avec des symboles (carrés, losanges, etc.)
  - o offre une vue d'ensemble de l'algorithme
  - o représentation
  - o presque abandonnée aujourd'hui
- ✓ Le pseudo-code: représentation textuelle avec une série de conventions ressemblant à un langage de programmation
  - o plus pratique pour écrire un algorithme
  - o représentation largement utilisée

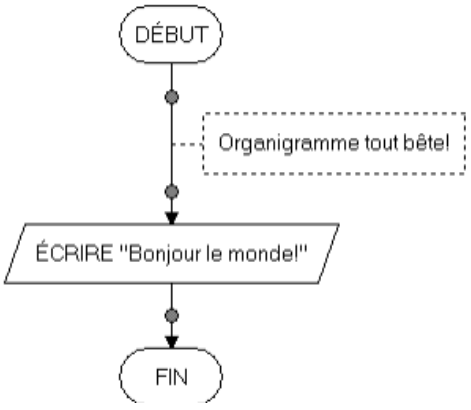
1.3. Quelques symboles utilisé dans la création d'un ordinogramme

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	**Symbole général** Opération ou groupe d'opérations sur des données, instructions, pour laquelle il n'existe aucun symbole normalisé.		**Renvoi** Symbole utilisé deux fois pour assurer la continuité lorsqu'une partie de ligne de liaison n'est pas représentée.
	**Sous-programme** Portion de programme considérée comme une simple opération.		**Début, fin , interruption** Début, fin ou interruption d'un algorigramme.
	**Entrée-Sortie** Mise à disposition d'une information à traiter ou enregistrement d'une information traitée.		**Commentaire** Symbole utilisé pour donner des indications sur les opérations effectuées.
Symbole de test		Les différents symboles sont reliés entre eux par des lignes de liaisons.	
	**Branchement** Exploitation de conditions variables impliquant un choix parmi plusieurs.		
Sens conventionnel des liaisons			

1.4. Premiers ordinogrammes

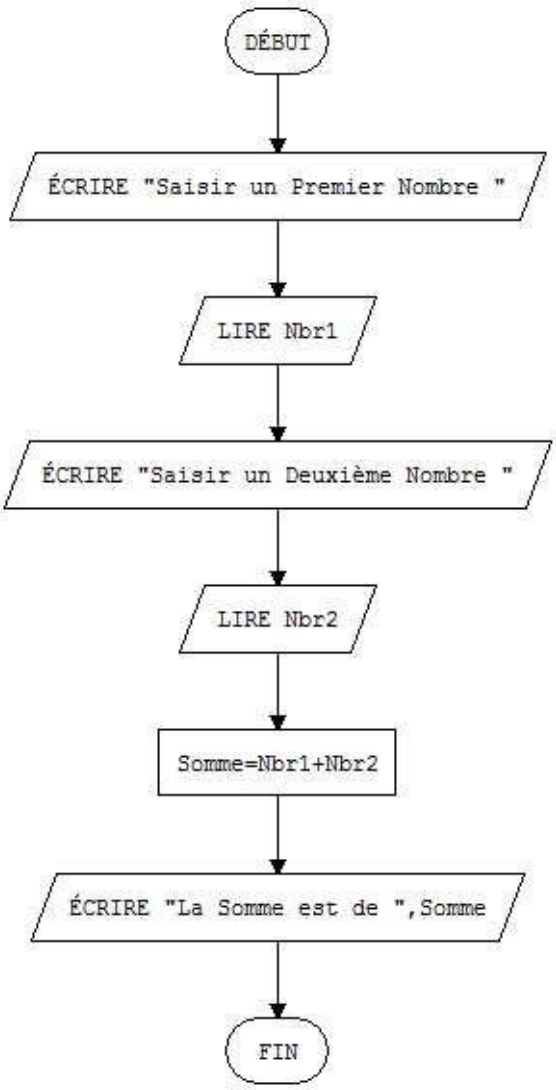
Voici un premier algorithme qui ne fait qu'afficher la chaîne de caractères **Bonjour le monde!** à l'écran. L'algorithme est formulé sous forme de pseudo-code et d'organigramme :

```
\\ Pseudo-code tout bête!
DÉBUT
  ÉCRIRE "Bonjour le monde!"
FIN
```



Autre Exemple : Écrire un Algorithme (Pseudo code et Ordinogramme) qui demande d'entrer deux nombres et qui ensuite détermine et affiche leur somme.

Ordinogramme



Pseudo code

```
Variable Nbr1, Nbr2, Somme en Numérique
DÉBUT
    ÉCRIRE "Saisir un Premier Nombre "
    LIRE Nbr1
    ÉCRIRE "Saisir un Deuxième Nombre "
    LIRE Nbr2
    Somme=Nbr1+Nbr2
    ÉCRIRE "La Somme est de ",Somme
FIN
```

Voici la capture d'écrans d'exécution de cet algorithme

```
Terminé
Saisir un Premier Nombre : 12
Saisir un Deuxième Nombre : 16
La Somme est de  28

Appuyez sur une touche pour fermer la console...
```

1.5. Structures conditionnelles

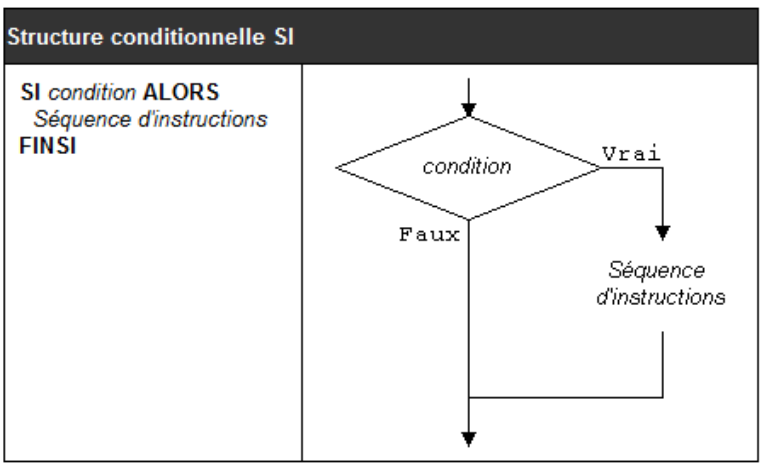
Les algorithmes présentés dans les sections précédentes sont constitués d'instructions exécutées séquentiellement, du début vers la fin.

La plupart des problèmes à résoudre par programmation requièrent l'éventualité d'effectuer un choix dans l'algorithme. Une *structure conditionnelle* est une instruction permettant de spécifier des séquences d'instructions alternatives dans un algorithme.

Quatre structures conditionnelles :

1. La *structure SI*
2. La *structure SI-SINON*
3. La structure *SI-SINON-SI*
4. La structure de *SÉLECTION*

Dans sa forme la plus simple (la structure SI), une structure conditionnelle sous forme pseudo-code est composée des mots réservés **SI**, **ALORS** et **FINSI**, d'une *condition* et d'une *séquence d'instructions* à exécuter lorsque la condition est vraie. Dans un organigramme la structure SI est composée d'une instruction conditionnelle où la *séquence d'instructions* est associée à la branche étiquetée *Vrai* :



Dans la structure pseudo-code ci-dessus (à gauche), le mot réservé **SI** indique le début de la structure conditionnelle, et le mot réservé **FINSI** en indique la fin. Dans la structure en organigramme correspondante (à droite) la condition indique le début de la structure conditionnelle et la convergence des deux branches (branches *Vrai* et *Faux*) en indique la fin.

Les structures conditionnelles sont basées sur l'évaluation d'une condition, dont le résultat est *vrai* ou *faux*. C'est sur la base de cette condition que le flux d'exécution est déterminé.

a. *Les Conditions*

Une condition est une comparaison. Cet énoncé décrit l'essentiel de ce qu'est une condition. En pratique, une *condition simple* est composée d'au moins trois éléments :

1. une première valeur,
2. un opérateur de comparaison, et
3. une seconde valeur.

Les valeurs peuvent être a priori de n'importe quel type (numériques, chaînes de caractères ou conteneurs) et peuvent être spécifiées explicitement sous forme de constantes ou implicitement sous forme d'expressions à évaluer. Si l'on veut que la comparaison ait un sens, il faut cependant que les deux valeurs comparées soient du même type ou de types comparables.

L'opérateur de comparaison dans une condition simple est appelé un opérateur relationnel. Ces opérateurs permettent de comparer l'envergure de deux valeurs.

Une condition simple peut aussi être un test de type : ce test sert à vérifier le type de la valeur résultante de l'évaluation d'une expression.

Enfin, des conditions simples peuvent être regroupées en une *condition composée* à l'aide des opérateurs logiques.

b. Les opérateurs relationnels

Les opérateurs relationnels de *LARP* permettent de comparer deux valeurs :

<	Plus petit : **a < b**
<=	Plus petit ou égal : **a <= b**
>	Plus grand : **a > b**
>=	Plus grand ou égal : **a >= b**
=	Égalité : **a = b**
!=	Inégalité : **a != b**. Le symbole équivalent **<>** est aussi supporté par *LARP*.

Les opérateurs relationnels permettent de comparer deux valeurs de types comparables. Des valeurs sont de types comparables lorsqu'elles peuvent être logiquement comparées. Ainsi, alors qu'il est logique de comparer une valeur entière à une valeur flottante, ça ne l'est pas de comparer une valeur entière à un conteneur.

Voici des exemples de conditions simples :

```
\\ Lire deux valeurs
ÉCRIRE "Entrez deux valeurs: "
LIRE a, b

\\ Identifier la plus petite de deux valeurs lues
SI a < b ALORS
  ÉCRIRE "Minimum = ", a
FINSI
```

```
SI a >= b ALORS
  ÉCRIRE "Minimum = ", b
FINSI

\\ Déterminer si une des deux valeurs est 0
SI a*b = 0 ALORS
  ÉCRIRE "Au moins une valeur est 0"
  ÉCRIRE "Veuillez entrer de nouvelles valeurs: "
  LIRE a, b
FINSI
```

Comme le démontre l'exemple ci-dessus, la condition simple peut en fait être composée d'expressions, et la séquence d'instructions dans la structure conditionnelle peut être constituée d'une ou plusieurs instructions.

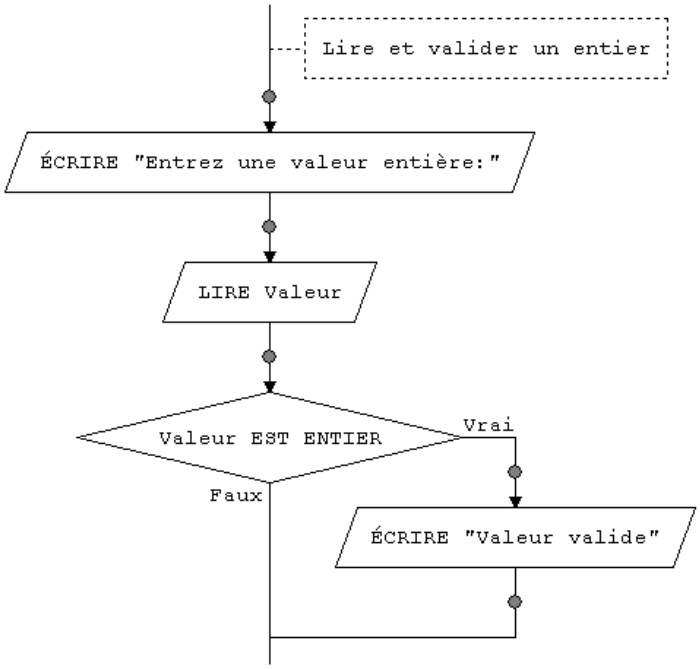
Notez que les opérateurs relationnels peuvent très bien être employés pour comparer des chaînes de caractères ou des conteneurs :

- Lorsque deux chaînes de caractères sont comparées, celles-ci le sont selon l'ordre alphabétique en fonction du codage ASCII. Ainsi, **"abc"<"b"**, mais **"abc">"B"**.
- L'égalité de conteneurs est déterminée selon leurs éléments. La comparaison est récursive lorsque des éléments sont eux-mêmes des conteneurs. Seuls les opérateurs = sont != sont applicables aux conteneurs.

Attention : les opérateurs relationnels ne peuvent pas être enchaînés. Par exemple, la condition **5<a<10** est invalide. Il faut exploiter les *opérateurs logiques* pour exprimer de telles conditions.

c. Test de type

Il est parfois nécessaire de vérifier le type d'une valeur avant de procéder à son traitement. C'est ainsi le cas lorsqu'on veut valider une valeur entrée par l'utilisateur. *LARP* dispose des mots réservés **ENTIER**, **FLOTTANT**, **CHAÎNE** et **CONTENEUR** qui, utilisés conjointement avec le mot réservé **EST**, permettant de vérifier le type de la valeur résultante de l'évaluation d'une expression :



d. Les opérateurs Logiques

Les *opérateurs logiques* permettent de relier des conditions simples en une seule « super-condition ». Le regroupement de conditions est parfois requis pour spécifier qu'un ensemble de conditions doit être satisfait pour procéder à l'exécution d'une séquence d'instructions. Par exemple, une *condition composée* est requise pour exprimer la condition « *la valeur doit être supérieure à zéro et inférieure à 100* » ou « *la couleur doit être rouge ou verte* ». Les opérateurs logiques permettent de tels regroupements de conditions simples.

Trois opérateurs logiques permettent de regrouper des conditions. Ceux-ci sont :

ET	Les deux conditions doivent être satisfaites : **a > 0 ET a < 100**
OU	Au moins une des deux conditions doit être satisfaite : **a < 1 *OU* a > 99**
!	Négation logique de la condition : **!(a EST ENTIER)**. Le mot réservé **NON** est équivalent.

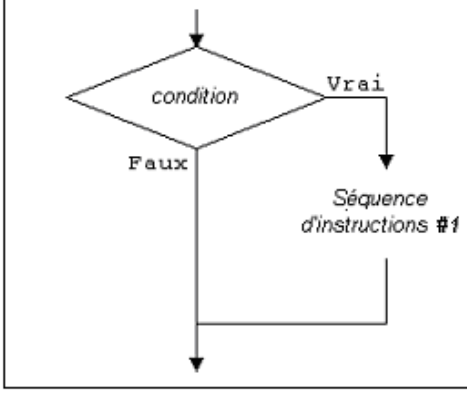
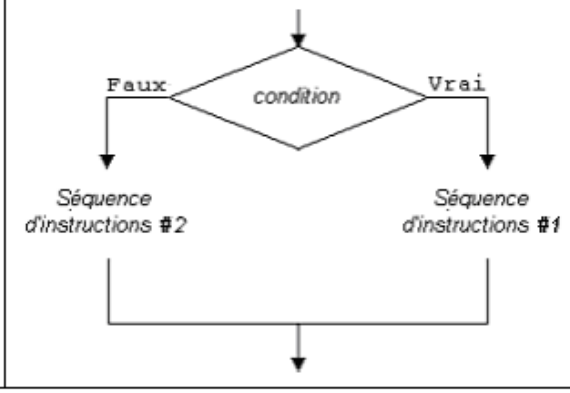
Notez que l'opérateur logique de négation est employé avec les parenthèses; celles-ci permettent de préciser la condition devant être inversée.

Contrairement aux opérateurs relationnels, les opérateurs logiques peuvent être enchaînés :

```
\\ Lire et valider une couleur
ÉCRIRE "Entrez une couleur: "
LIRE couleur
SI couleur="bleu" OU couleur="blanc" OU couleur="rouge" ALORS
  ÉCRIRE "Couleur invalide"
FINSI
```

e. Structures SI et SI-SINON

Il n'y a que deux formes possibles de structures *SI* : la forme de droite du tableau ci-dessous est la forme complète, celle de gauche la forme simple.

Structure SI	Structure SI-SINON
SI *condition* **ALORS** *Séquence d'instructions #1* **FINSI**	**SI** *condition* **ALORS** *Séquence d'instructions #1* **SINON** *Séquence d'instructions #2* **FINSI**
	

Une *condition* est une expression composée d'opérateurs relationnels (parfois aussi d'opérateurs arithmétiques et logiques) dont la valeur est vraie ou fausse. Cela peut donc être :

- une condition, ou
- un test de type.

Ces deux structures conditionnelles sont relativement claires. Lorsque le flux d'exécution atteint la structure (i.e. la ligne **SI** *condition* **ALORS** du pseudo-code, ou le losange de la condition dans l'organigramme), *LARP* examine la valeur de la *condition*. Si cette condition est vraie, la *Séquence d'instructions #1* est exécutée. Cette séquence d'instructions peut être très brève comme très longue, cela n'a aucune importance. À la fin de la *Séquence d'instructions #1*, *LARP* saute à la fin de la structure conditionnelle :

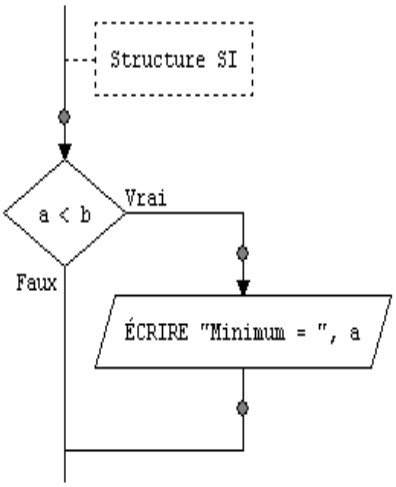
- Au moment où le flux d'exécution arrive au mot **SINON** du pseudo-code, *LARP* saute directement à la première instruction située après le **FINSI**.
- Au moment où le flux d'exécution atteint le point de convergence des branchements de la structure conditionnelle de l'organigramme, *LARP* quitte cette structure.

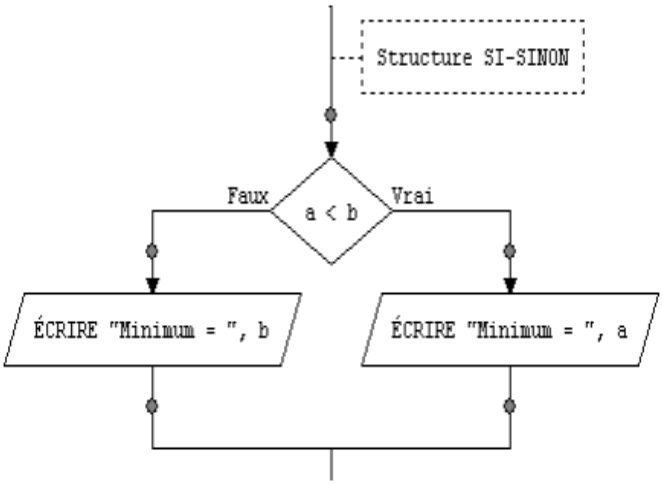
De même, si la *condition* est fausse, le flux d'exécution ignore la *Séquence d'instructions #1* et passe directement à la première ligne située après le **SINON** afin d'exécuter la *Séquence d'instructions #2* dans le cas du pseudo-code, ou après le point de convergence des branchements de la structure conditionnelle de l'organigramme.

```
\\ Structure SI
SI a < b ALORS
  ÉCRIRE "Minimum = ", a
FINSI

\\ Structure SI-SINON
SI a < b ALORS
  ÉCRIRE "Minimum = ", a
SINON
  ÉCRIRE "Minimum = ", b
FINSI
```

La forme simplifiée correspond au cas où la branche fausse du **SI** serait vide. Dès lors, plutôt qu'écrire « sinon ne rien faire du tout », il est plus simple de ne rien écrire.





Exercices sur les structures conditionnelles

Question 1 : Ecrire un algorithme (Code source et ordinogramme) qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut le traitement du cas où le nombre vaut zéro).

Question 2 : Cet algorithme (Code source et ordinogramme) est destiné à prédire l'avenir, et il doit être infaillible !

Il lira au clavier l'heure et les minutes, et il affichera l'heure qu'il sera une minute plus tard. Par exemple, si l'utilisateur tape 21 puis 32, l'algorithme doit répondre : "Dans une minute, il sera 21 heure(s) 33".

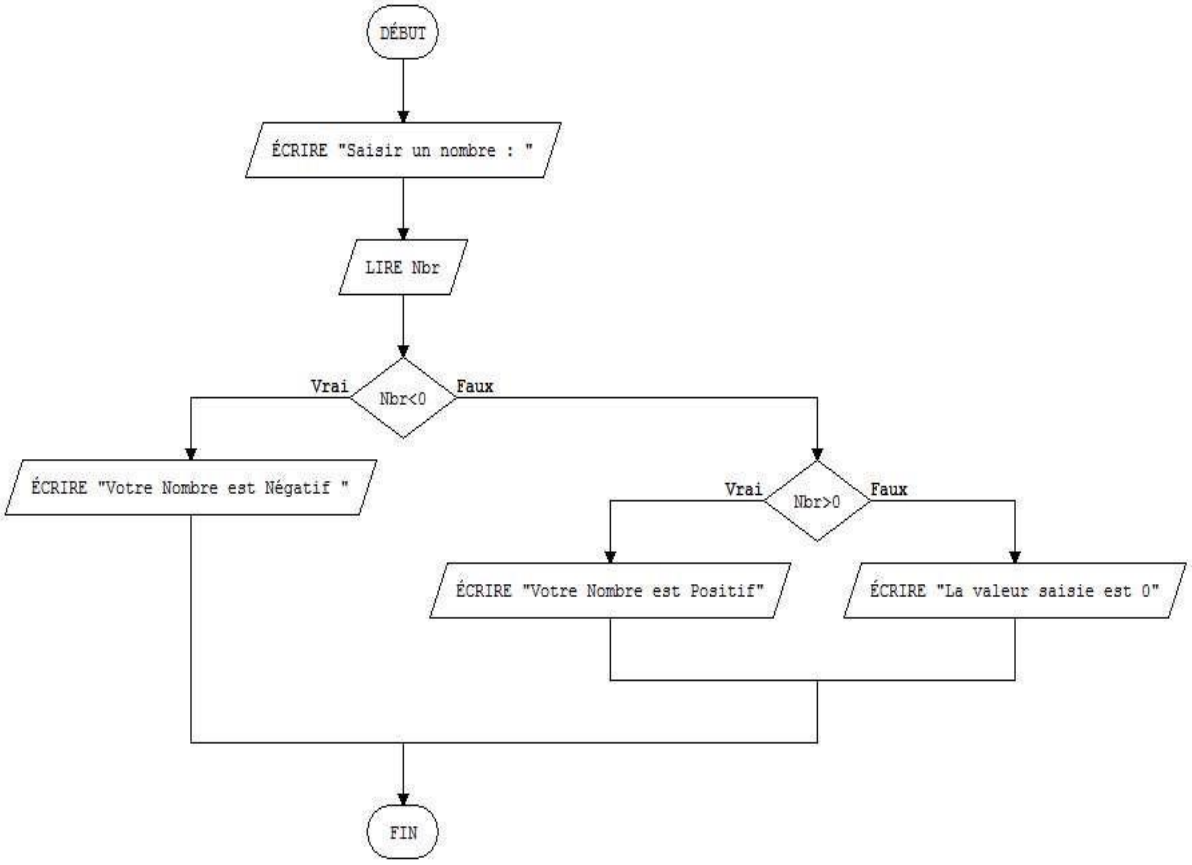
NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.

Question 3 : Un magasin de reprographie facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà.

Ecrivez un algorithme (Code source et ordinogramme) qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

Solution Question 1 :

```
Variable Nbr en Numérique
DÉBUT
    ÉCRIRE "Saisir un nombre : "
    LIRE Nbr
    SI Nbr<0 ALORS
        ÉCRIRE "Votre Nombre est Négatif "
    SINON
        SI Nbr>0 ALORS
            ÉCRIRE "Votre Nombre est Positif"
        SINON
            ÉCRIRE "La valeur saisie est 0"
        FINSI
    FINSI
FIN
```



1.6. Structures répétitives

Il arrive fréquemment qu'un algorithme doive répéter une séquence d'instructions un certain nombre de fois afin d'accomplir une tâche; en fait, la plupart des algorithmes requièrent de telles répétitions. *LARP* offre trois structures à cette fin, généralement appelées *structures répétitives* ou *boucles* :

1. La *structure TANTQUE*
2. La *structure RÉPÉTER-JUSQU'À*
3. La *structure POUR*

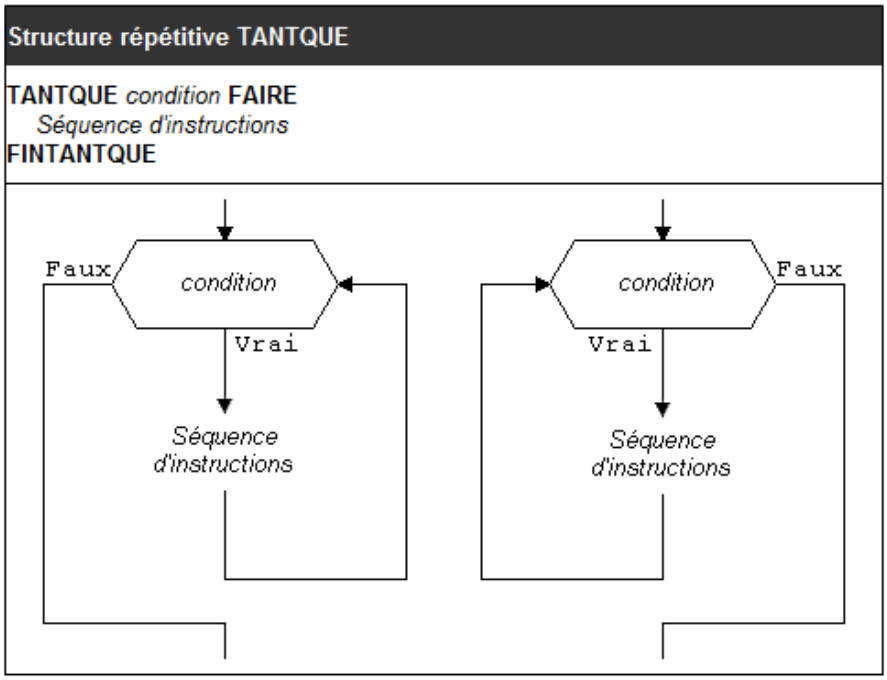
Les structures répétitives sont basées sur l'évaluation d'une condition, dont le résultat est vrai ou faux. C'est sur la base de cette condition que la répétition est déterminée.

1.6.1. Structure TANTQUE

Dans sa forme la plus simple (structure TANTQUE) telle que présentée dans le tableau ci-dessous, une structure répétitive sous forme pseudo-code est composée des mots réservés **TANTQUE**, **FAIRE** et **FINTANTQUE**, d'une condition et d'une *séquence d'instructions* à exécuter tant que la condition est vraie.

Dans un organigramme (voir le tableau ci-dessous), la structure TANTQUE comprend une condition dans un hexagone suivie de la *séquence d'instructions* sur la branche étiquetée *Vrai*.

L'orientation de la branche *Faux* peut être à gauche ou à droite de la condition. Notez que la branche *Vrai* retourne à la condition, indiquant ainsi que le flux d'exécution retourne évaluer la condition une fois la *séquence d'instructions* exécutée.

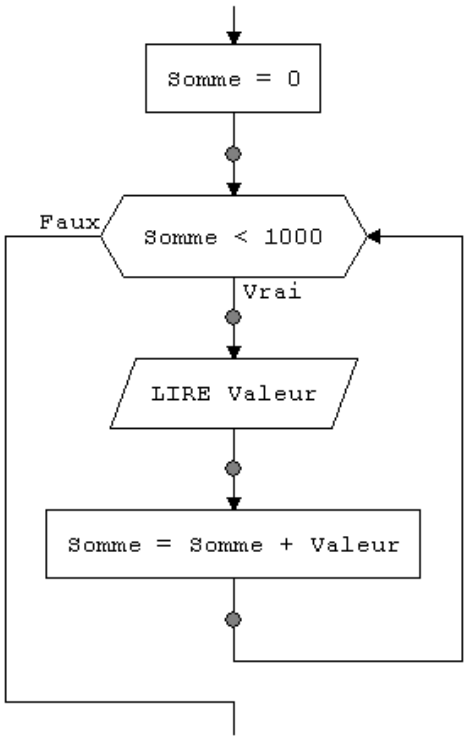


Dans la structure pseudo-code ci-dessus, le mot réservé **TANTQUE** indique le début de la structure répétitive, et le mot réservé **FINTANTQUE** en indique la fin. Dans la structure en organigramme, la condition indique le début de la structure répétitive et la fin de la branche étiquetée *Faux* en indique la fin.

Les structures répétitives sont basées sur l'évaluation d'une condition, dont le résultat est vrai ou faux. C'est sur la base de cette condition que le flux d'exécution est déterminé. Dans le cas d'une structure TANTQUE, la *séquence d'instructions* est exécutée tant que la *condition* est satisfaite (i.e. vraie).

L'exemple ci-dessous (sous forme pseudo-code et d'organigramme) exploite une structure TANTQUE afin d'additionner des valeurs lues jusqu'à ce que leur somme atteigne ou dépasse **1000** :

```
Somme = 0
TANTQUE Somme < 1000 FAIRE
  LIRE Valeur
  Somme = Somme + Valeur
FINTANTQUE
```



La condition d'une structure TANTQUE est évaluée avant chaque *itération* (une itération est une et une seule exécution de la *séquence d'instructions* dans la boucle). Ainsi, l'exécution de la structure peut être résumée ainsi :

1. vérifier la *condition*
2. si elle est vraie alors

    2.1. 2.1. exécuter la *séquence d'instructions*

    2.2.2.2. retourner à l'étape 1.

Ainsi, la condition doit obligatoirement être satisfaite pour que la séquence d'instructions soit exécutée. Dès que la condition devient non satisfaite (i.e. fausse), le flux d'exécution est redirigé vers les instructions suivant la structure répétitive (après le **FINTANTQUE** en pseudo-code).

Une particularité de la structure TANTQUE est que la *séquence d'instructions* peut ne pas être exécutée du tout si la première évaluation de la condition (i.e. lorsque le flux d'exécution entre dans la structure répétitive) retourne faux. En effet, si la condition est fausse dès le départ, aucune itération de la boucle ne sera effectuée.

Puisque qu'une exécution de la *séquence d'instructions* dans une structure répétitive est dite une *itération*, les structures répétitives sont aussi appelées *structures itératives*. Ce sont des synonymes.

Exercices sur la structure Tant que

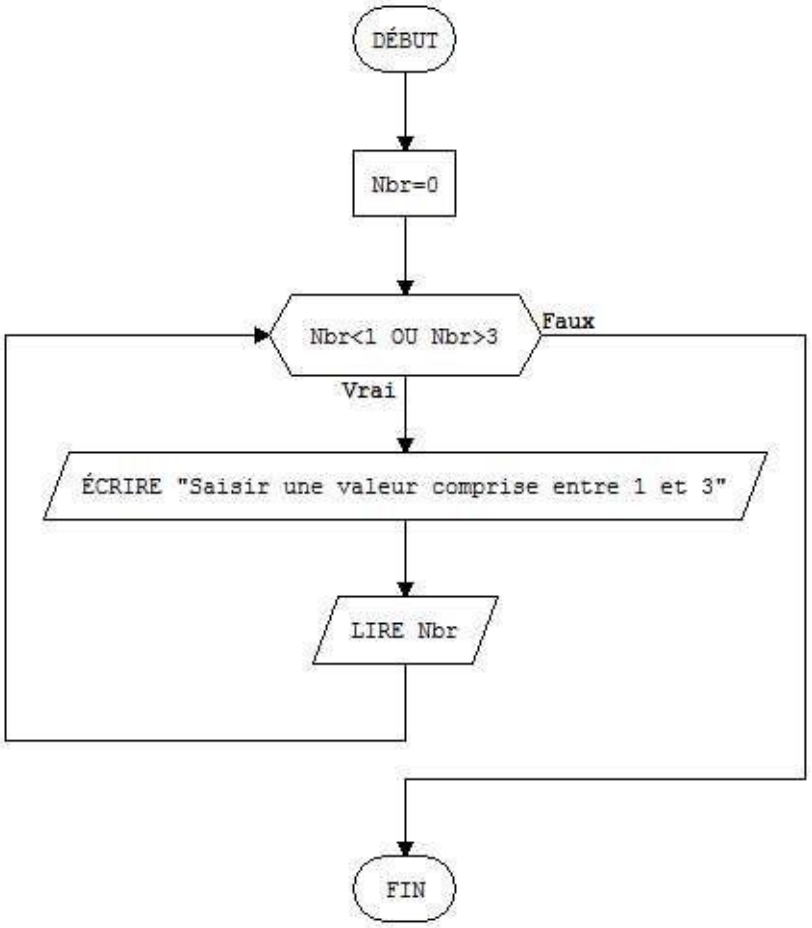
Question 1 :

Ecrire un algorithme (Pseudo code et Ordinogramme) qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Solution :

```
DÉBUT
    Nbr=0
```

```
    TANTQUE Nbr<1 OU Nbr>3 FAIRE
        ÉCRIRE "Saisir une valeur comprise entre 1 et 3"
        LIRE Nbr
    FINTANTQUE
FIN
```



Question 2 :

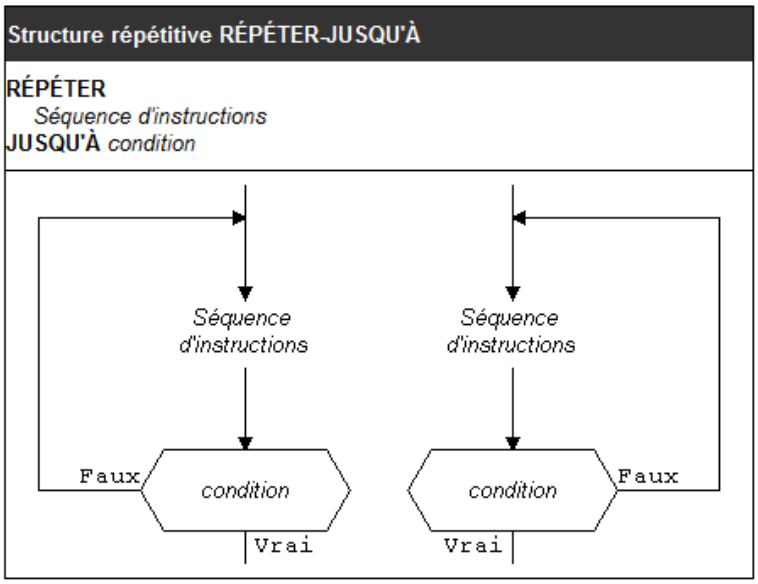
Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

1.6.2. Structure RÉPÉTER-JUSQU'À

La *structure RÉPÉTER-JUSQU'À* est semblable à la structure TANTQUE. Comme cette dernière, elle implique l'exécution d'une *séquence d'instructions* à répétition et en fonction de la valeur d'une condition. Cependant, les structures RÉPÉTER-JUSQU'À et TANTQUE diffèrent sur deux points :

1. La structure TANTQUE exécute la séquence d'instructions tant et aussi longtemps que la condition est satisfaite, alors que la structure RÉPÉTER-JUSQU'À exécute la séquence d'instructions tant et aussi longtemps que la condition *n'est pas* satisfaite. En d'autres mots, la structure RÉPÉTER-JUSQU'À itère *jusqu'à* ce que la condition devienne vraie.
2. La structure TANTQUE vérifie la condition *avant* chaque itération, alors que la structure RÉPÉTER-JUSQU'À vérifie la condition *après* chaque itération.

La différence marquée entre ces deux structures réside dans le fait que, pour la structure RÉPÉTER-JUSQU'À, la séquence d'instructions est exécutée *au moins une fois*, sans égard à la valeur de la condition. Cette caractéristique est mise en évidence par la position de la condition dans la structure : elle est située à la fin de celle-ci (alors que dans la structure TANTQUE la condition est située au début de la structure).



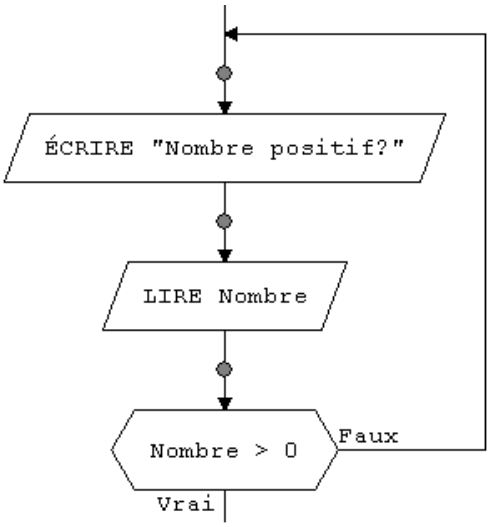
La structure RÉPÉTER-JUSQU'À, présentée dans le tableau ci-dessus, est composée des mots réservés **RÉPÉTER** et **JUSQU'À**, d'une condition et d'une *séquence d'instructions* à exécuter jusqu'à ce que la condition devienne vraie (en d'autres mots, tant qu'elle est fausse).

Dans un organigramme la structure RÉPÉTER-JUSQU'À comprend une condition dans un hexagone précédée de la *séquence d'instructions*.

L'orientation de la branche *Faux*, qui représente l'itération, peut être à gauche ou à droite de la condition. Notez que la branche *Vrai* sort de la boucle, indiquant ainsi que le flux d'exécution cesse d'exécuter la *séquence d'instructions* lorsque la condition est vraie.

L'exemple ci-dessous exploite une structure RÉPÉTER-JUSQU'À afin de lire un entier positif et le valider :

```
RÉPÉTER
  ÉCRIRE "Nombre positif?"
  LIRE Nombre
JUSQU'À Nombre > 0
```



Puisque au moins une lecture doit être effectuée, la structure RÉPÉTER-JUSQU'À est préférable car elle fait obligatoirement une itération (i.e. une lecture) avant de vérifier la condition. La structure RÉPÉTER-JUSQU'À est généralement préférée à la structure TANTQUE lorsque les variables dont dépend la condition reçoivent leur valeur dans la séquence d'instructions, ce qui exige obligatoirement une première itération.

Notez cependant qu'une structure TANTQUE peut remplacer toute structure RÉPÉTER-JUSQU'À, au coût de quelques instructions supplémentaires (dans le pseudo code ci-dessous une opération lecture supplémentaire doit être insérée avant la première itération) :

```
ÉCRIRE "Nombre positif?"
 LIRE Nombre
 TANTQUE Nombre <= 0 FAIRE
   ÉCRIRE "Nombre positif?"
   LIRE Nombre
 FINTANTQUE
```

Exemple 2 : Ecrire un algorithme qui (*Pseudo-code et ordinogramme)* demande deux nombres à l'utilisateur, puis qui calcul et affiche la somme de nombre. Puis qui affiche le message « Voulez-vous continuer »

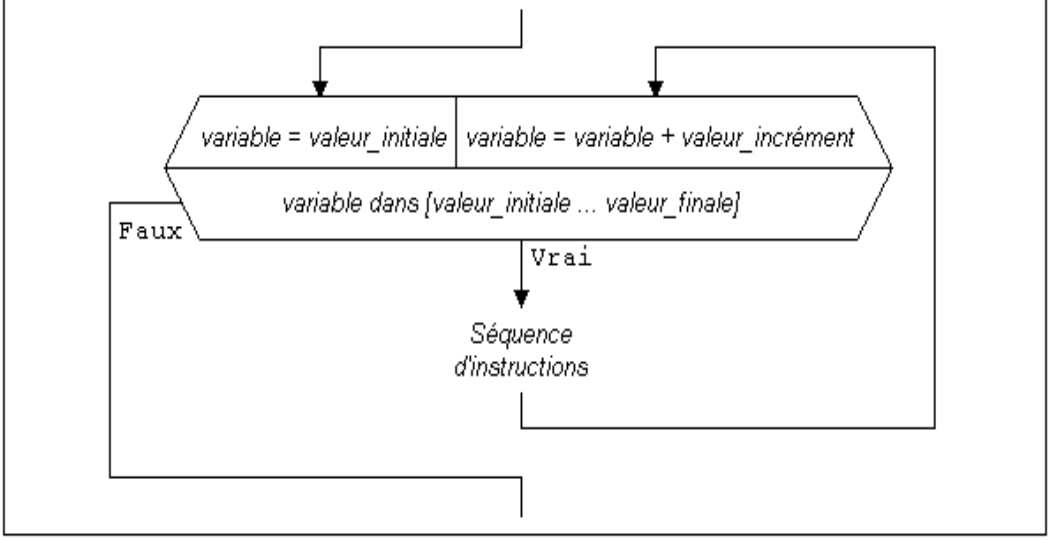
Exemple 3 : Ecrire un algorithme (*Pseudo-code et ordinogramme*) qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

1.6.3. Structure POUR

La troisième structure répétitive est plus sophistiquée que les deux premières (les structures TANTQUE et RÉPÉTER-JUSQU'À). La *structure POUR* a la forme suivante :

Structure répétitive POUR

POUR *variable* = *valeur_initiale* **JUSQU'À** *valeur-finale* **INCRÉMENT** *valeur_incrément* **FAIRE**
Séquence d'instructions
FINPOUR



Dans la structure pseudo-code ci-dessus, le mot réservé **POUR** indique le début de la structure répétitive, et le mot réservé **FINPOUR** en indique la fin.

La structure répétitive POUR est généralement employée lorsqu'on veut répétitivement faire varier la valeur d'une variable (identificateur *variable* ci-dessus) d'une valeur initiale (identificateur *valeur_initiale* ci-dessus) jusqu'à une valeur finale (identificateur *valeur_finale* ci-dessus), tout en exécutant une *séquence d'instructions* pour chaque valeur de cette variable. L'identificateur *valeur_incrément* indique le changement à appliquer à *variable* à la fin de chaque itération.

Ainsi, dans la structure ci-dessus, les identificateurs signifient :

- *variable* : variable dont la valeur varie de *valeur_initiale* à *valeur_finale*, changeant d'une unité à chaque itération.
- *valeur_initiale* : valeur de la variable à la première itération.
- *valeur_finale* : valeur de la variable à la dernière itération.
- *valeur_incrément* : incrément appliqué à la variable à chaque itération.
- *Séquence d'instructions* : instructions exécutées à chaque itération. La *variable* peut être employée dans ces instructions.

Notez que l'incrément peut être omis de la structure POUR. Dans ce cas, l'incrément par défaut appliqué est de 1. La *variable* employée pour compter les itérations dans une structure POUR est généralement appelée une *variable d'itération*.

Voici un exemple d'utilisation d'une telle structure. Supposons qu'on veuille afficher les températures en Fahrenheit correspondant aux 20 premiers degrés Celsius du thermomètre. On peut produire une telle grille de températures avec une structure TANTQUE :

```
 \\ Affiche les températures de 0C à 19C en
Fahrenheit
 Celsius = 0
 TANTQUE Celsius <= 19 FAIRE
   Fahrenheit = Celsius * 9/5 + 32
   ÉCRIRE Celsius, ' = ', Fahrenheit
   Celsius = Celsius + 1
 FINTANTQUE
```

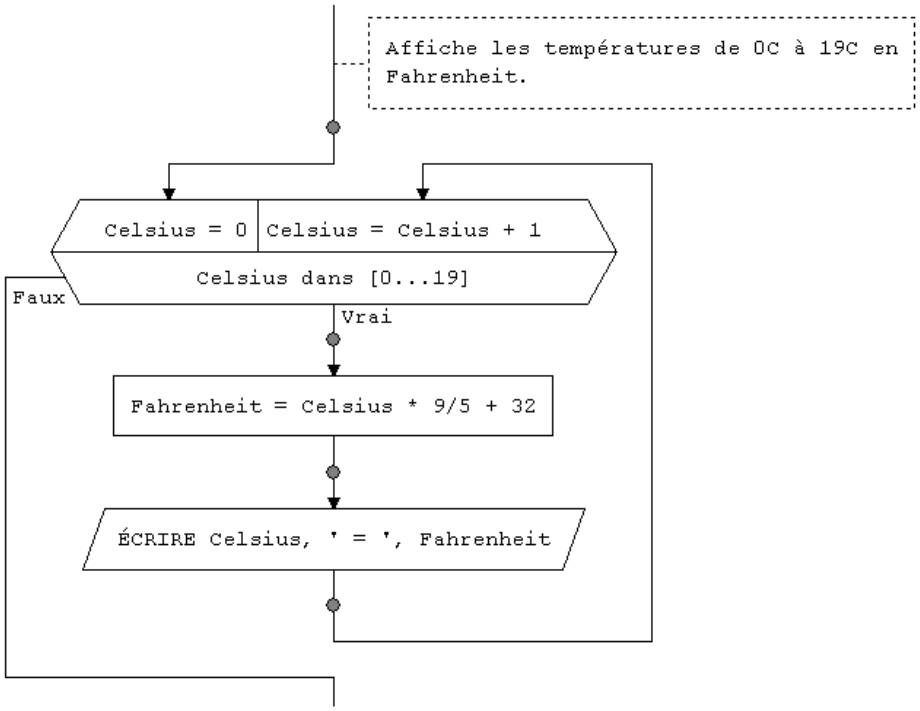
La structure ci-dessus effectue exactement 20 itérations, augmentant de 1 la valeur de la variable **Celsius** à chaque itération.

La structure POUR offre une syntaxe plus compacte et naturelle pour représenter les structures répétitives telles que celle ci-dessus :

```
 \\ Affiche les températures de 0C à 19C en
Fahrenheit
 POUR Celsius = 0 JUSQU'À 19 FAIRE
   Fahrenheit = Celsius * 9/5 + 32
   ÉCRIRE Celsius, ' = ', Fahrenheit
 FINPOUR
```

Dans cette structure, la variable **Celsius** est initialisée à **0** avant la première itération, puis successivement augmentée de **1** à la fin de chaque itération (1 est l'incrément par défaut puisque aucune valeur d'incrémentation n'est fournie dans la structure POUR). On constate donc que l'incrémentation est implicite (nul besoin de l'incrémenter explicitement dans la boucle avec l'instruction **Celsius = Celsius + 1**). Lorsque la variable **Celsius** atteint la valeur **20**, le flux d'exécution quitte la structure POUR et poursuit l'exécution du pseudo-code suivant la boucle, après le **FINPOUR**.

L'organigramme suivant présente le même algorithme que le pseudo-code précédent.



Notez que la structure en organigramme comporte une condition sous une forme différente de celles retrouvées dans les structures TANTQUE et RÉPÉTER-JUSQU'À en organigramme. Le symbole hexagonal de la structure POUR contient tous les éléments de son équivalente en

pseudo-code : l'initialisation de la variable d'itération en entrée de la structure (**Celsius = 0**), une vérification de continuer les itérations (**Celsius dans [0...19]**) et l'incrémentation de la variable d'itération à la fin de chaque itération (**Celsius = Celsius + 1**). Les branchements de la structure POUR en organigramme indiquent clairement le parcours du flux d'exécution dans la structure :

1. 1. En entrant dans la structure, l'instruction d'initialisation (**Celsius = 0**) est exécutée une et une seule fois.
2. 2. La variable d'itération est ensuite validée en fonction des valeurs limites d'itération (**Celsius dans [0...19]**). Si la valeur de la variable d'itération est dans cette intervalle, alors il y a itération :
   - 2.1 2.1 Les deux instructions dans la boucle sont exécutées.
   - 2.2 2.2 La variable d'itération est incrémentée (**Celsius = Celsius + 1**).
   - 2.3 2.3 Enfin le flux d'exécution retourne à l'étape 2 afin de valider la valeur de la variable d'itération.

L'option de spécifier une valeur incrément dans la structure POUR offre la possibilité d'utiliser un incrément autre que 1 dans une boucle. Ainsi, l'exemple suivant convertit seulement les températures Celsius paires en Fahrenheit :

```
\\ Affiche les températures 0C, 2C, 4C, 6C... à 18C en Fahrenheit
POUR Celsius = 0 JUSQU'À 18 INCRÉMENT 2 FAIRE
  Fahrenheit = Celsius * 9/5 + 32
  ÉCRIRE Celsius, ' = ', Fahrenheit
FINPOUR
```

Notez qu'une structure POUR incrémente la valeur de la variable d'une unité à chaque itération. Si cependant la *valeur_initiale* est supérieure à la *valeur_finale*, la variable sera *décrémentée* (diminuée de un) à chaque itération :

```
\\ Affiche les températures de 19C à 0C en Fahrenheit
POUR Celsius = 19 JUSQU'À 0 FAIRE
  Fahrenheit = Celsius * 9/5 + 32
  ÉCRIRE Celsius, ' = ', Fahrenheit
FINPOUR
```

On peut aussi spécifier une valeur d'incrémentation négative afin de réduire la variable (**Celsius**) par bonds autres que d'une seule unité :

```
\\ Affiche les températures de 18C, 16C, 14C ... à 0C en Fahrenheit
POUR Celsius = 18 JUSQU'À 0 INCRÉMENT -2 FAIRE
  Fahrenheit = Celsius * 9/5 + 32
  ÉCRIRE Celsius, ' = ', Fahrenheit
FINPOUR
```

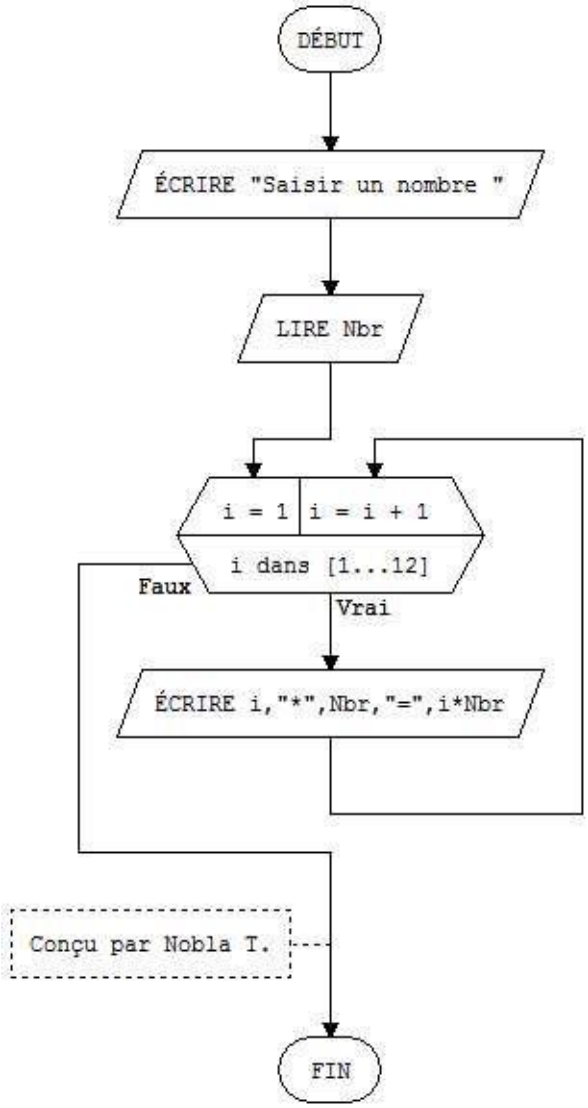
La valeur de la variable d'itération (**Celsius** dans l'exemple ci-dessus) peut être employée dans les instructions incluses dans la structure POUR, mais elle ne peut pas être modifiée par ces instructions. Ainsi, dans le pseudo-code suivant, l'instruction **LIRE i** est interdite car elle vise à modifier la valeur de la variable d'itération **i**. Par contre, l'instruction **Log(i * 100)** est permise dans la boucle, tout comme **Fahrenheit = Celsius * 9/5 + 32** dans le pseudo-code précédent, puisqu'elles ne modifient pas la valeur de la variable d'itération.

```
POUR i = 0 JUSQU'À 10 FAIRE
  ÉCRIRE Log(i * 100)
  LIRE i
FINPOUR
```

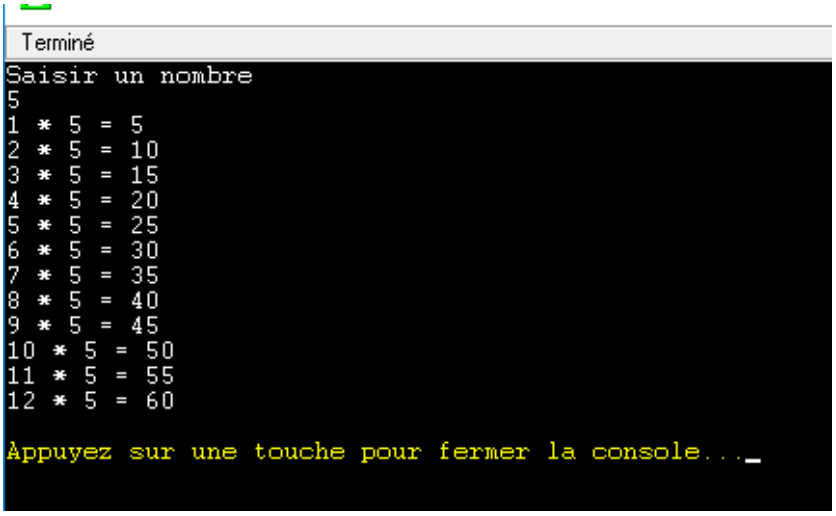
Exemple 2 : Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) : Table de 7 : 7 x 1 = 7 7 x 2 = 14 7 x 3 = 21 … 7 x 10 = 70

```
DÉBUT
    ÉCRIRE "Saisir un nombre "
    LIRE Nbr
    POUR i = 1 JUSQU'À 12 INCRÉMENT 1 FAIRE
        ÉCRIRE i,"*",Nbr,"=",i*Nbr
    FINPOUR

    \\ Conçu par Nobla T.
FIN
```



Voici l'écran d'exécution



Exemple 3 : Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer : 1 + 2 + 3 + 4 + 5 = 15

Chapitre 2 : Sous-Programmes et Tableaux

1.1. Les modules ou sous-programmes

Dans la résolution d'un problème, on peut constater qu'une suite d'actions revient plusieurs fois. Dans ce cas, il serait judicieux de l'écrire une seule fois, et de l'utiliser autant de fois que c'est nécessaire, en effectuant des calculs avec des données différentes.

Cette suite d'actions sera définie dans un sous-programme, qui peut prendre soit la forme d'une *procédure*, soit la forme d'une *fonction*.

D'autre part, on peut observer que certains groupes d'actions se rapportent à des traitements précis et différents. Il est souhaitable alors de représenter chacun d'eux dans un sous-programme, ce qui permettra d'améliorer la conception du programme et sa lisibilité. On perçoit alors un programme comme un ensemble de procédures/fonctions. La structuration d'un programme par morceaux (modules) est la base de la programmation structurée et modulaire.

La *programmation modulaire* est une technique utilisée lors de la conception d'algorithmes complexes. Elle consiste à diviser un algorithme en sections. Chaque section est appelée un *module* et exécute une tâche simple.

Voici des exemples de tâches qu'un module peut exécuter :

- Afficher un menu d'options
- Afficher des résultats
- Calculer une moyenne de données
- Valider des données
- Trier des données

Un sous-programme (ou sous algorithme) est un module algorithmique complet, définit et indépendant qui peut être utilisé ou appelé par un quelconque algorithme principal ou par d'autre sous-programme. Un sous-programme reçoit des valeurs, les arguments, à partir de l'algorithme appelant, il exécute des calculs et transmet ensuite les résultats obtenus au programme appelant.

Dans son utilisation, les modules offrent plusieurs avantages :

- La complexité des programmes est simplifiée pour éviter des recopiés inutiles.
- La division de travail de manière à permettre à plusieurs personnes travaillant indépendant la faciliter de trouver un profit aux problèmes complexes.
- La localisation des erreurs est facile.
- La lisibilité des programmes.

Un module est identifié par un *nom* unique, et consiste en une séquence d'instructions débutant avec le mot réservé **ENTRER** et se terminant avec le mot réservé **RETOURNER**. La séquence d'instructions contenue dans le module est exécutée lorsque le nom du module est rencontré durant l'exécution d'autres modules. On dit alors que le module est *invoqué*.

1.1.1. Noms de modules

Les modules d'un projet *en Algorithmique* doivent être nommés selon les règles suivantes :

- Un nom de module doit débuter par une lettre (**A** à **Z**, **a** à **z**) ou le caractère de soulignement (_).
- Un nom de module peut être constitué de lettres minuscules, de lettres majuscules, de chiffres et du caractère de soulignement.

À noter que tous les modules d'un même projet doivent avoir un nom distinct.

a. Module principal

Lorsqu'un algorithme est divisé en plusieurs modules, un de ces modules doit tenir le rôle de *module principal*. Contrairement aux autres modules (appelés modules auxiliaires) qui débutent par le mot réservé **ENTRER** et se terminent par **RETOURNER**, le module principal débute par le mot réservé **DÉBUT** et se termine par le mot réservé **FIN** :

```
\\ Module principal
DÉBUT
  ÉCRIRE "Bonjour"
FIN
```

Un projet *LARP* doit obligatoirement disposer d'un et un seul module principal, car *LARP* débute l'exécution de l'algorithme à l'instruction **DÉBUT** et termine l'exécution de l'algorithme à l'instruction **FIN**. Seul le module principal peut contenir les instructions **DÉBUT** et **FIN**. Par contre, un projet peut contenir ou non des modules auxiliaires, et ceux-ci doivent débuter avec l'instruction **ENTRER** et se terminer avec l'instruction **RETOURNER**.

b. Modules auxiliaires

Les *modules simples* (i.e. sans paramètre) sont utilisés pour exécuter des tâches simples comme afficher des menus pour l'utilisateur. Le module se compose d'une séquence d'instructions regroupées entre les mots réservés **ENTRER** et **RETOURNER**.

Tous les modules autres que le module principal sont appelés *modules auxiliaires* et ils effectuent généralement des tâches requises par le module principal.

Voici un exemple de module auxiliaire affichant un menu :

```
\\ Module auxiliaire Menu
ENTRER
  ÉCRIRE "Le menu est"
  ÉCRIRE " 1 - Lire le dossier"
  ÉCRIRE " 2 - Sauvegarder le dossier"
  ÉCRIRE " 3 - Afficher le dossier"
  ÉCRIRE " 4 - Modifier le dossier"
  ÉCRIRE " 5 - Quitter"
RETOURNER
```

Le module ci-dessus, appelé **Menu**, exécute les instructions séquentiellement, de **ENTRER** jusqu'à **RETOURNER**. Pour l'invoquer dans un algorithme, le nom du module doit être spécifié dans un module (généralement un autre module du projet), précédé du mot réservé **EXÉCUTER** :

```
\\ Module principal
DÉBUT
  RÉPÉTER            \\ Afficher le menu
    EXÉCUTER Menu
    REQUÊTE "Commande? ", Commande
  JUSQU'À Commande = 5
FIN
```

Dans le module principal ci-dessus, le module simple **Menu** est exécuté (i.e. invoqué) à chaque itération afin d'afficher le menu. Toutes les instructions du module **Menu** sont exécutées à chaque invocation. L'instruction **REQUÊTE** suivant l'invocation du module est exécutée après chaque invocation de **Menu**.

Le résultat à la console de l'exécution du module principal avec les valeurs 1 et 5 fournies par l'utilisateur est :

```
Le menu est
 1 - Lire le dossier
 2 - Sauvegarder le dossier
 3 - Afficher le dossier
 4 - Modifier le dossier
 5 - Quitter
Commande? 1
Le menu est
 1 - Lire le dossier
 2 - Sauvegarder le dossier
 3 - Afficher le dossier
 4 - Modifier le dossier
 5 - Quitter
Commande? 5
```

Évidemment, le module principal ci-dessus est incomplet, puisque aucune action n'est entreprise lorsque l'utilisateur sélectionne les commandes 1 à 4.

c. Variables locales

Un module peut exploiter ses propres variables pour accomplir des tâches. Ces variables, appartenant exclusivement au module et n'étant pas partagées avec les autres modules du projet, sont appelées des *variables locales* car elles sont locales au module.

Les variables locales sont accessibles partout entre les mots réservés **ENTRER** et **RETOURNER** du module auxiliaire ou **DÉBUT** et **FIN** du module principal. Lorsque deux modules utilisent le même nom pour une variable locale, les deux variables sont distinctes. Les modules qui suivent illustrent cette indépendance :

```
\\ Module principal
DÉBUT
  Valeur = 1
  EXÉCUTER Module_A
  ÉCRIRE Valeur       \\ Affiche 1 comme valeur
FIN
```

```
\\ Module auxiliaire Module_A
ENTRER
```

```
  Valeur = 2
RETOURNER
```

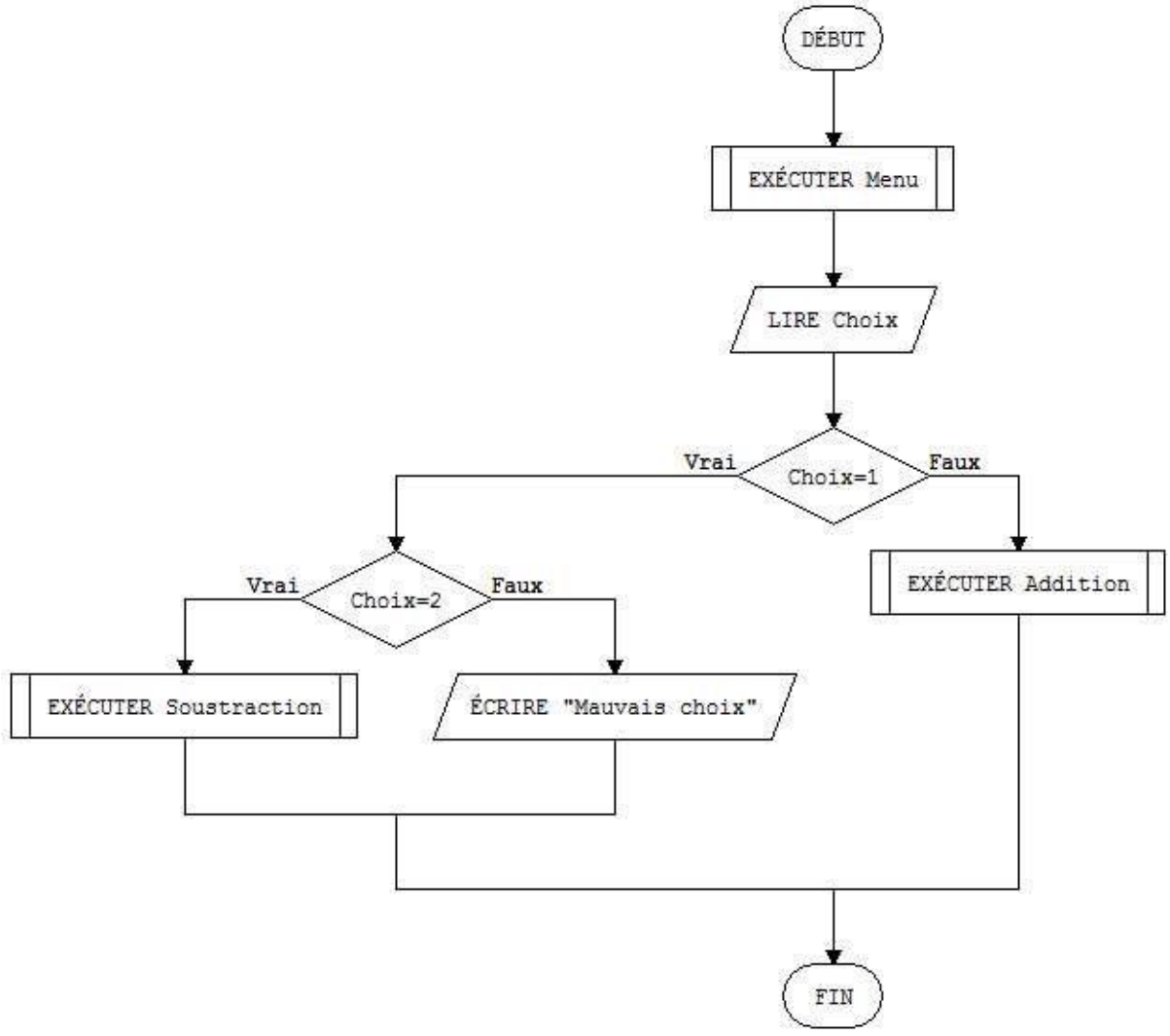
Même si les deux modules exploitent une variable nommée **Valeur**, ces deux variables sont distinctes, celle du module principal n'étant pas modifiée par l'invocation du module auxiliaire.

Exercice sur les sous programmes

Ecrire un algorithme (Pseudo code et Ordinogramme) qui affiche le menu suivant :

1/ Addition
2/ Soustraction
Votre Choix :

Si le choix est 1, l'algorithme demande d'entrer deux nombres et affiche leur somme. Si le choix est 2, il demande d'entrer deux nombres et affiche leur différence. Prévoir un sous-programme pour chaque élément du menu.




```
DÉBUT
    EXÉCUTER Menu
    LIRE Choix
    SI Choix=1 ALORS
        SI Choix=2 ALORS
            EXÉCUTER Soustraction
```

```
        SINON
            ÉCRIRE "Mauvais choix"
        FINSI
    SINON
        EXÉCUTER Addition
    FINSI
FIN
```

Sous-Programme Addition

```
ENTRER
    ÉCRIRE "Premier Nombre "
    LIRE Nbr1
    ÉCRIRE "Deuxième Nombre "
    LIRE Nbr2
    ÉCRIRE "La somme est de : ",Nbr1+Nbr2
RETOURNER
```

Sous-programme soustraction

```
ENTRER
    ÉCRIRE "Premier Nombre : "
    LIRE Nbr1
    ÉCRIRE "Deuxième nombre : "
    LIRE Nbr2
    ÉCRIRE "La différence est de ",Nbr1-Nbr2
RETOURNER
```

Sous-programme Menu

```
ENTRER
    ÉCRIRE "MENU"
    ÉCRIRE "1. Addition"
    ÉCRIRE "2. Soustraction"
    ÉCRIRE "Votre Choix :"
RETOURNER
```

2.2. Tableaux

2.2.1. Tableaux linéaires

Lorsque dans un programme, les données se multiplient, au lieu de créer 40 000 variables, on peut utiliser un tableau qui aura pour effet de nous simplifier grandement la tâche ; où si nous ayons besoin simultanément de 15 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple N1, N2, N3, etc avant d'arriver au calcul, et après une succession de quinze instructions « Lire » distinctes. Cela donnera obligatoirement une situation quelque peu complexe du genre :

Moy← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12+N13+N14+N15)/15

Considérons le cas où nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors le problème devient complexe. Si en plus

on est dans une situation on l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, là les choses se compliquent davantage.

C'est pourquoi la programmation nous permet de **rassembler toutes ces variables en une seule**, au sein de laquelle chaque valeur sera désignée par un numéro.

Définition 1. : un tableau est une variable (possédant un nom) pouvant accueillir un nombre fini de valeurs de même type. La représentation d'un tableau de nom « Tab » de 10 éléments dont chaque valeur est 0 est la suivante :

Tab

0	0	0	0	0	0	0	0	0

Définition 2. : Est un ensemble de valeurs de même type portant le même nom de variable et repérées par un nombre (indice), il s'appelle aussi variable indicée.

Note	**12**	**5**	**14**	**23**	**11**	**15**	**18**	**9**	**24**	**26**	**13**	**22**

Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses. Ex. Note (4) = 23

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables.

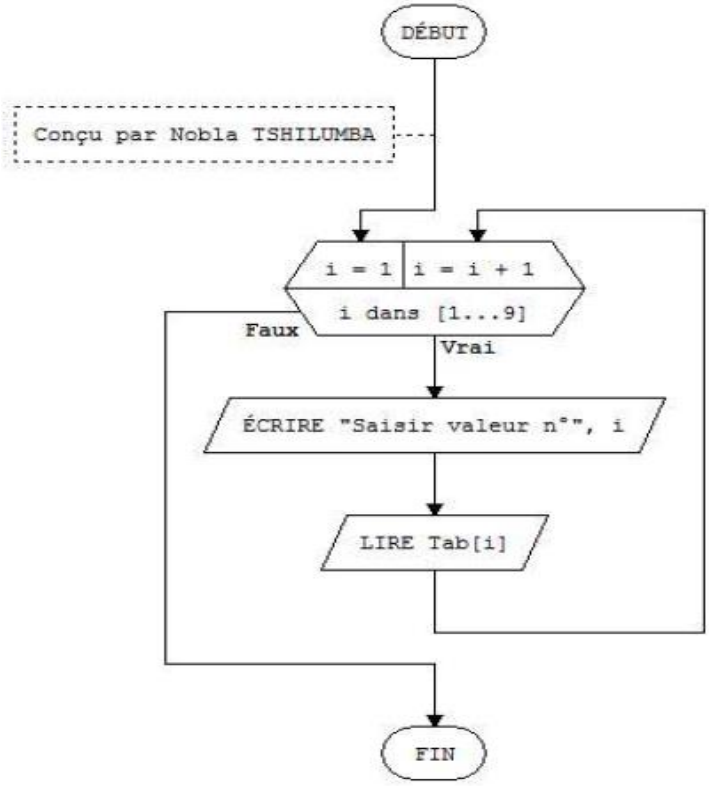
Pour affecter une valeur dans un tableau, il faut préciser l'emplacement exact dans ce tableau. Le rang de cet emplacement est appelé **l'indice du tableau.**

Exemple : Ecrire un algorithme (pseudo-code et ordinogramme) qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Pseudo code

```
DÉBUT
    \\ Conçu par Anaclet TSHIKUTU
    POUR i = 1 JUSQU'À 9 INCRÉMENT 1 FAIRE
        ÉCRIRE "Saisir valeur n°", i
        LIRE Tab[i]
    FINPOUR
FIN
```

Ordinogramme



2.2.2. Quelques algorithmes de traitement courant sur les tableaux

1. Parcours d'un tableau

Tous les éléments d'un tableau possèdent le même type de données. Évidemment, lorsque le type est variant, chaque élément peut contenir différents types de données (objets, chaînes, nombres, etc.). Vous pouvez déclarer un tableau avec n'importe quel type de données fondamental, y compris les types définis par l'utilisateur.

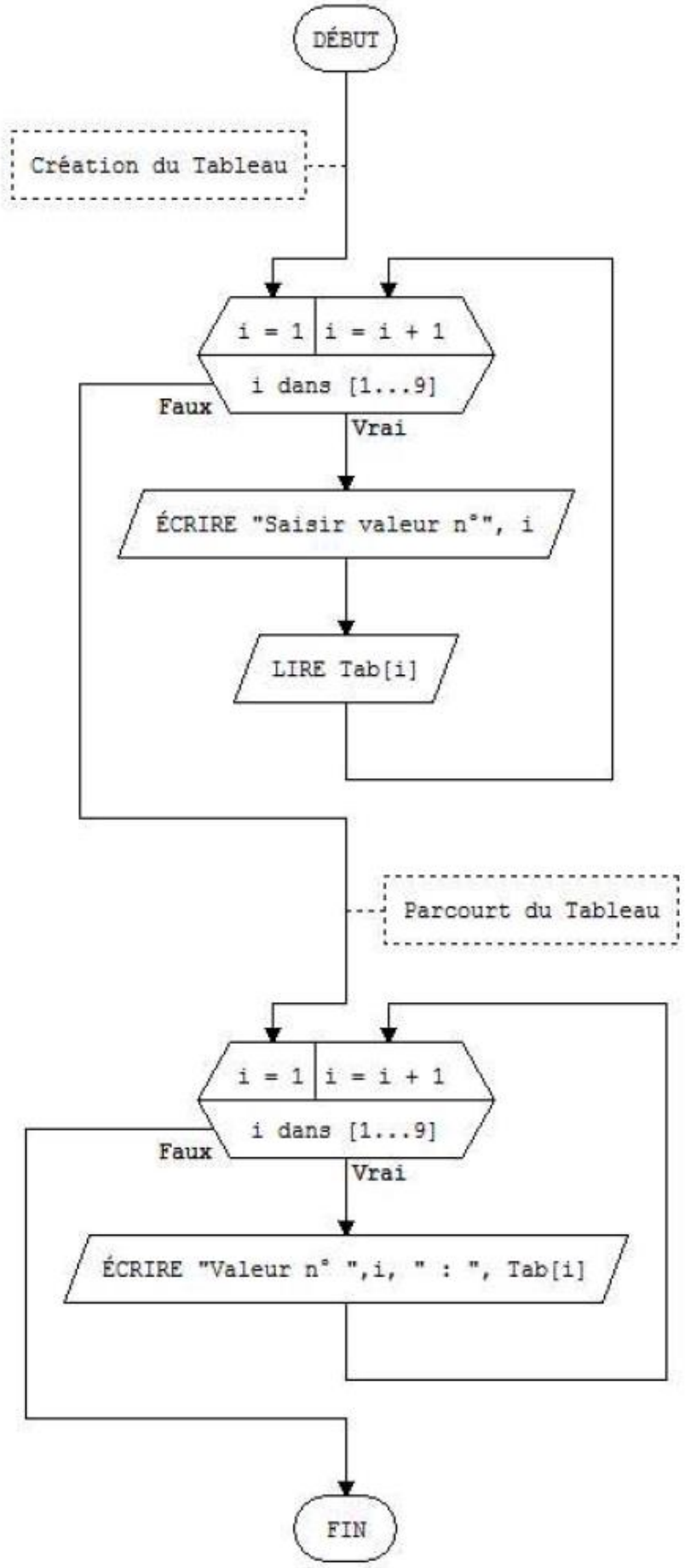
Exemple : Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Pseudo code

```
DÉBUT
    \\ Création du Tableau
    POUR i = 1 JUSQU'À 9 INCRÉMENT 1 FAIRE
        ÉCRIRE "Saisir valeur n°", i
        LIRE Tab[i]
    FINPOUR

    \\ Parcourt du Tableau
    POUR i = 1 JUSQU'À 9 INCRÉMENT 1 FAIRE
        ÉCRIRE "Valeur n° ",i, " : ", Tab[i]
    FINPOUR
FIN
```

Ordinogramme


DÉBUT
Création du Tableau
i = 1
i = i + 1
i dans [1...9]
Faux
Vrai
ÉCRIRE "Saisir valeur n°", i
LIRE Tab[i]
Parcourt du Tableau
i = 1
i = i + 1
i dans [1...9]
Faux
Vrai
ÉCRIRE "Valeur n° ",i, " : ", Tab[i]
FIN

Remarque générale : lors de parcours d'un tableau, l'indice sert à désigner les éléments d'un tableau et doit être souvent un nombre en clair, mais il peut être aussi une variable, ou une expression calculée. La valeur de cet indice doit être égale au moins 1, doit être un nombre entier et doit nécessairement être inférieure et égale au nombre d'élément du tableau.

2. LA RECHERCHE DANS UN TABLEAU

a. Recherche linéaire

Soit un tableau comportant, disons, 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

```
Tableau Tab(19) en Numerique
Variable N en Numerique
Debut
Ecrire "Entrez la valeur a rechercher"
Lire N
Trouve ← Faux
Pour i ← 0 a 19
Si N = Tab(i) Alors
Trouve ← Vrai
FinSi
i suivant
Si Trouve Alors
Ecrire "N fait partie du tableau"
Sinon
Ecrire "N ne fait pas partie du tableau"
FinSi
Fin
```

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie.

Voilà la raison qui nous oblige à passer par une variable booléenne. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

b. Recherche de l'élément maximum et de l'élément minimum dans un tableau linéaire

Soit un tableau Temp contenant 12 températures dans un tableau de single. Lorsqu'on veut chercher la température maximale et minimale dans ce tableau de 12 températures, on procédera comme suit :

- **Recherche de l'élément maximal**

On doit parcourir le tableau et de conserver la valeur maximale dans une variable dite temporaire « TempMax ».

Au départ, on suppose que la température maximale est la première du tableau.

Var TempMax : réel
TempMax = Temp (1)

Pour i ← 2 à 12

Si Temp (i) > TempMax alors TempMax ← Temp (i)

Fin Pour i
A la fin du processus, la variable TEMPMAX contiendra la valeur recherchée.

- **Recherche de l'élément minimal**

Mêmement qu'à la recherche de l'élément maximal, mais ici, il suffit de conserver la valeur minimale dans une variable dite temporaire « TempMin ».

Au départ, on suppose que la température maximale est la première du tableau.

Var TempMin : réel
TempMin = Temp (1)

Pour i ← 2 à 12

Si Temp (i) < TempMin alors TempMin ← Temp (i)

Fin Pour i
A la fin du processus, la variable TEMPMin contiendra la valeur recherchée.

c. Recherche dichotomique

Lorsqu'on veut trier un élément dans l'ordre croissant et retourner l'indice d'une occurrence de l'élément cherché, on procède à la recherche dichotomique. Une telle recherche peut être réalisée de manière séquentielle ou dichotomique. On développe ici la version dichotomique qui est la plus efficace en temps d'exécution.

Principe : On effectue la comparaison de l'élément cherché par rapport à celui qui se trouve au milieu du tableau. Si l'élément cherché est plus petit, on continue la recherche dans la première moitié du tableau sinon dans la seconde. On recommence ce processus sur la moitié. On s'arrête lorsqu'on a trouvé ou lorsque l'intervalle de recherche est nul.

Exemple

Recherche dans le tableau d'entiers suivant défini sur l'intervalle [1..8] :

T	5	13	18	23	46	53	89	97

Recherche de 46 :

Etape 1 : comparaison de 46 avec t(4) (4=(8+1)÷2), t(4)<46 => recherche dans [5..8]
Etape 2 : comparaison de 46 avec t(6), (6=(8+5)÷2), t(6)>46 => recherche dans [5..5]
Etape 3 : comparaison de 46 avec t(5), t(5)=46 => élément cherché est trouvé à l'indice 5

Recherche de 10 :

Etape 1 : comparaison de 10 avec t(4), t(4)>10 => recherche dans [1..3]
Etape 2 : comparaison de 10 avec t(2), t(2)>10 => recherche dans [1..1]
Etape 3 : comparaison de 10 avec t(1), t(1)<10 => recherche dans [2..1], Borne inférieure devient supérieure à la borne supérieure, donc on met fin à l'algorithme et l'élément cherché n'a pas été trouvé !

Nous avons l'algorithme suivant :

```
Début

Var e, n : entiers

Var Tableau t(n) : entier

        D ← 1
        F ← n
        trouve ←  faux
        tant que (D <= F) et (trouve=faux) faire
            i ← (D + F)÷2
            Si t(i) = e alors
                trouve ← vrai
                sinon si t(i) > e alors
                    F ← i -1
                    Sinon
               D ←  i +1
                Fin si
            Fin si
        Fin faire
        Si trouve=vrai alors
            indice ←  i
            afficher " L'élément se trouve à la position" ; indice
        sinon
        indice ← -1
          afficher " L'élément n'existe pas dans le tableau"
        Fin si
        Fin
```

Légende :

- e désigne l'élément recherché
- n : taille du tableau
- t : tableau trié par ordre croissant
- D : début de la zone de recherche
- F : fin de la zone de recherche
- trouve : booléen, faux tant que l'élément cherché n'est pas trouvé
- i : indice de la case du milieu de la zone de recherche
- indice : indice de l'élément recherché ou -1 s'il n'est pas trouvé.

3. Tri d'un tableau

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par sélection, et le tri à bulles.

a) Tri par sélection

Si l'on veut trier un tableau de 12 éléments dans l'ordre croissant, la technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis on met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément, et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	12	122	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera:

3	12	21	45	122	78	64	53	89	28	84	46

Etc jusqu'à l'avant dernier.

Nous pourrions décrire le processus de la manière suivante :

- Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
- Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Cela s'écrit :

boucle principale : le point de départ se décale à chaque tour

```
Pour i ← 0 a 10
On considère provisoirement que t(i) est le plus petit élément
posmini ← i
on examine tous les éléments suivants
Pour j ← i + 1 a 11
Si t(j) < t(posmini) Alors
posmini ← j
Finsi
Fin pour j
```

A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'a effectuer la permutation.

```
temp ← t(posmini)
t(posmini) ← t(i)
t(i) ← temp
```

On a placé correctement l'élément numéro i, on passe a présent au suivant.

i suivant

b) Tri à bulles

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel **tout élément est plus petit que celui qui le suit**.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ».

En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés.

Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous utiliserons une variable booléenne Yapermute qui va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

```
Variable Yapermute en Booleen
Debut
…
TantQue Yapermute
…
FinTantQue
Fin
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire.

```
Variable Yapermute en Booleen
Debut
…
TantQue Yapermute
Pour i ← 0 a 10
Si t(i) > t(i+1) Alors
temp ← t(i)
t(i) ← t(i+1)
t(i+1) ← temp
Finsi
i suivant
Fin
```

Mais il ne faut pas oublier un détail capital : la gestion de la variable booléenne. L'idée, c'est que cette variable va signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai a Yapermute tout au départ de l'algorithme.

La solution complète donne donc :

```
Variable Yapermute en Booleen
Debut
…
Yapermut ← Vrai
TantQue Yapermut
Yapermut ← Faux
Pour i ← 0 a 10
Si t(i) > t(i+1) alors
temp ← t(i)
t(i) ← t(i+1)
t(i+1) ← temp
Yapermut ← Vrai
Finsi
Fin pour i
FinTantQue
Fin
```

2.2.3. Tableaux multidimensionnels

Un tableau multidimensionnel est un tableau dont les éléments sont respectivement repérés par deux ou trois indices. C'est-à-dire un tableau à plusieurs dimensions et il utilise souvent des boucles imbriquées lors de sa mise en place.

Un tableau multidimensionnel doit être déclaré comme tel également avant son utilisation, en précisant la taille (intervalle de définition selon chaque dimension) et le type de valeurs qu'il contiendra. La syntaxe retenue est :

Var Tableau NomTableau (taille du tableau) : type d'éléments

La programmation autorise à déclarer des tableaux de dimensions multiples. Par exemple, l'instruction suivante déclare un tableau bidimensionnel de 10 sur 10 à l'intérieur d'une procédure :

A(9, 9) : Réels

Vous pouvez déclarer l'une et/ou l'autre de ces deux dimensions avec des limites inférieures explicites :

A(1 à 10, 1 à 10) : Réels

Vous pouvez étendre cela à plus de deux dimensions. Par exemple :

Declarer MultiD(3, 1 à 10, 1 à 15)

Cette déclaration crée un tableau en trois dimensions de 4 sur 10 sur 15. Le nombre total d'éléments est le produit de ces trois dimensions, soit 600.

Manipulation de tableaux à l'aide de boucles

Vous pouvez traiter efficacement un tableau multidimensionnel à l'aide de boucles Pour imbriquées. Les instructions suivantes, par exemple, initialisent chaque élément de A à l'aide d'une valeur basée sur sa position dans le tableau :

```
Declarer I, J: Entiers
A(1 à 10, 1 à 10) : Réels
Pour I = 1 à 10
  Pour J = 1 à 10
    A(I, J) = I * 10 + J
  Fin pour J
FinPour I
```

Tableaux à deux dimensions

Un tableau A à deux dimensions m x n est un ensemble de m.n éléments d'information tels que chacun d'entre eux est spécifié par une paire d'entiers (tels que J,K), appelés indices et possédant la propriété $1 \leq J \leq m$ et $1 \leq K \leq m$. L'élément de A dont le premier indice est j et le second, k sera noté $A_{j,k}$ ou A[J,K].

Les tableaux à deux dimensions sont appelés matrices et sont faciles à se représenter comme une grille ayant un certain nombre de lignes (première dimension) et un certain nombre de colonnes (seconde dimension).

Considérons une matrice (tableau) M avec 6 colonnes et 4 lignes dont :

56	54	1	- 56	20	22
72	8	54	34	43	2
70	5	16	78	90	4
56	23	- 47	0	5	12

Pour accéder à un élément du tableau, il suffit de préciser entre parenthèses l'indice de la case contenant cet élément, et ce pour chacune des dimensions. Par exemple, pour accéder à l'élément 23 du tableau d'entiers ci-dessus, on écrit : M (4,2). L'instruction suivante affecte à la variable x la valeur du premier élément du tableau, c'est à dire 56.

x ← M (1,1)

L'élément désigné du tableau peut alors être utilisé comme n'importe quelle variable :

M (2,1) ← - 56

Cette instruction modifie le contenu de la case (2,1) du tableau M. (72 en – 56)

Exemple : l'algorithme qui calcule la somme des éléments dans ce tableau

```
Début
Var Somme : réel
Var Tableau m() : réels
Var li, co : entiers
Afficher "Combien y a-t-il de lignes à saisir ?"
Lire li
Afficher "Combien y a-t-il de colonnes à saisir ?"
Lire co
Redim m(li, co)
Pour i de 1 à li : Pour j de 1 à co : Lire m(i,j) : Fin pour j : Fin pour i
Somme ← 0
Pour i de 1 à li : Pour j de 1 à co : Somme ← Somme + m(i,j) : Fin pour j
Fin pour i
Afficher Somme
Fin
```

Exercices sur les Tableaux

Question 1 : Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

Question 2 : Ecrivez un algorithme permettant, toujours sur le même principe, à l'utilisateur de saisir un nombre déterminé de valeurs. Le programme, une fois la saisie terminée, renvoie la plus grande valeur en précisant quelle position elle occupe dans le tableau. On prendra soin d'effectuer la saisie dans un premier temps, et la recherche de la plus grande valeur du tableau dans un second temps.

Question 3 : Ecrivez un algorithme de délibération des étudiants d'une promotions. L'algorithme demande le nombre d'étudiant de la promotion et le nom de chacun, le nombre de cours vu et leurs intitulés. Et pour chaque étudiant, l'algorithme demande les cotes obtenues pour chacun de cours enregistrés. Après saisi, l'algorithme affiche pour chaque étudiant son Nom, son pourcentage, sa mention et le nombre d'échecs.

Chapitre 3 : Fichiers, Listes chainées et notions de complexité des algorithmes

3.1. Les Fichiers

Par défaut, lors de l'exécution d'un algorithmes les données sont lues via le clavier et les résultats sont affichées à la console d'exécution (i.e. l'écran). Le clavier (en lecture) et la console d'exécution (en écriture) sont les *interfaces d'entrées/sorties standards* de *LARP*.

Dans certains cas cependant, l'algorithme doit exploiter des données provenant d'autres sources que le clavier. Ces données sont généralement stockées dans un document externe à l'algorithme. Dans d'autres cas, l'algorithme doit conserver les résultats produits dans un document externe permanent, ce qui n'est pas le cas avec la console d'exécution où les résultats affichés sont perdus lorsque la console est fermée.

Un Fichiers est support de données généralement stocké sur le disque rigide de l'ordinateur (ou sur un autre ordinateur accessible via une connexion réseau). Le fichier est identifié par un nom unique au système de fichiers de l'ordinateur.

3.1.1. Canaux d'entrées/sorties

L'exploitation d'un document (un fichier) dans un algorithme est effectuée via un *canal d'entrées/sorties*. Un numéro unique est associé à chaque document lors de son ouverture. Par la suite toute instruction impliquant ce document exploite le numéro de canal d'entrées/sorties pour identifier le document visé.

Lorsqu'un canal d'entrées/sorties est associé à un document (lors de son ouverture), cette association doit être explicitement rompue (par la fermeture du document) avant d'associer le canal à un autre document.

Similairement, deux canaux d'entrées/sorties ne peuvent pas simultanément être associés à un même document. Toute violation à ces restrictions entraîne automatiquement une interruption de l'exécution de l'algorithme.

L'utilisation des canaux d'entrées/sorties est bien illustrée dans les exemples des prochaines sections.

3.1.2. Ouverture d'un Fichier

L'instruction d'ouverture d'un document est **OUVRIR**. Cette instruction permet :

- d'associer un canal d'entrées/sorties à un fichier ou un tampon d'entrées/sorties, et
- d'indiquer le mode d'accès au canal d'entrées/sorties.

Un fichier ne peut être associé à plus d'un canal simultanément, et il demeure associé au canal jusqu'à ce que celui-ci soit fermé.

Les instructions de gestion de canal (**LECTURE**, **ÉCRITURE**, **FERMER**, ...) de *LARP* utilisent le canal d'entrées/sorties comme référence au document ouvert.

```
\\ Ouvrir un fichier en mode Lecture
OUVRIR FICHIER "C:\\DONNEES.TXT" SUR 3 EN LECTURE
```

Le pseudo-code ci-dessus ouvre en mode lecture le fichier **DONNEES.TXT** localisé dans le répertoire **C:**. Le chemin du fichier dans la structure de répertoires doit être spécifié à l'aide d'*antéslashs* (****), où le double antéslash est la séquence d'échappement représentant l'antéslash simple (****).

Si aucun chemin de répertoire n'est spécifié avec le nom du fichier, *l'algorithme* ouvre le fichier dans le répertoire courant (habituellement le répertoire où est sauvegardé le projet en exécution). Cependant, puisqu'il peut y avoir des exceptions à cette règle, il est recommandé de toujours précéder le nom d'un fichier du chemin de répertoire complet où est situé le fichier dans l'unité de stockage.

L'algorithme peut être dans l'impossibilité d'ouvrir le fichier spécifié pour une ou l'autre des raisons suivantes :

- **Le fichier n'existe pas** : l'algorithme tente d'ouvrir en mode lecture un fichier inexistant.
- **Le fichier est déjà ouvert** : l'algorithme tente d'ouvrir un fichier qui est déjà ouvert.
- **Le canal d'entrées/sorties n'est pas disponible** : l'algorithme tente d'ouvrir un fichier sur un canal d'entrées/sorties invalide ou déjà associé à un autre fichier.
- **Le nom du fichier est invalide** : le nom de fichier spécifié est invalide (peut-être dû au répertoire inexistant, au nom du fichier contenant des caractères interdits par *Windows®* ou à l'unité de stockage défectueuse ou non disponible).

Lorsqu'il y a erreur à l'ouverture d'un fichier, *l'algorithme* interrompt l'exécution de l'algorithme et affiche un message d'erreur précisant la cause de l'erreur.

3.1.3. Mode d'accès

Lors de l'ouverture d'un fichier, un mode d'accès doit être spécifié :

Trois modes d'accès aux tampons d'entrées/sorties et fichiers sont supportés par *LARP* :

- **LECTURE** : permet de lire le contenu du document à l'aide de l'instruction **LIRE**. Si le document est inexistant, l'exécution de l'algorithme est interrompue.
- **ÉCRITURE** : permet d'écrire des résultats dans le document à l'aide de l'instruction **ÉCRIRE**. Le contenu antérieur à l'ouverture du document est effacé. Si le document est un fichier inexistant, celui-ci est créé. Si le document est un tampon d'entrées/sorties inexistant, l'exécution de l'algorithme est interrompue.
- **AJOUT** : permet d'écrire des résultats *à la fin* du document à l'aide de l'instruction **ÉCRIRE**. Le contenu antérieur à l'ouverture du document est conservé. Si le document est un fichier inexistant, celui-ci est créé. Si le document est un tampon d'entrées/sorties inexistant, l'exécution de l'algorithme est interrompue.

La principale distinction entre les modes d'accès **ÉCRITURE** et **AJOUT** est au niveau du contenu antérieur du document :

- L'ouverture d'un tampon d'entrées/sorties ou d'un fichier en mode **ÉCRITURE** efface automatiquement le contenu antérieur du document (i.e. si le document existait déjà, son contenu est effacé).

- L'ouverture d'un tampon d'entrées/sorties ou d'un fichier en mode **AJOUT** préserve le contenu existant, les résultats y étant écrits à la fin.

Seule l'instruction de lecture (**LIRE**) est autorisée sur un canal d'entrées/sorties associé à un document ouvert en mode **LECTURE**. Similairement, seule l'instruction d'écriture (**ÉCRIRE**) est autorisée sur un canal d'entrées/sorties associé à un document ouvert en mode **ÉCRITURE** ou **AJOUT**. Toute instruction de lecture ou d'écriture invalide sur un canal d'entrées/sorties cause automatiquement l'arrêt de l'exécution de l'algorithme et un message d'erreur approprié est affiché.

3.1.4. Fermeture d'un canal d'entrées/sorties

Tout tampon d'entrées/sorties ou fichier ouvert doit éventuellement être fermé. L'instruction de fermeture est appliquée au canal d'entrées/sorties associé au document qu'on désire fermer :

```
OUVRIR "DONNEES" SUR 3 EN LECTURE
FERMER 3
```

Voici les règles régissant la fermeture des documents (tampons d'entrées/sorties et fichiers) :

- Tout document ouvert doit être fermé. Si l'algorithme termine son exécution sans avoir fermé tous les tampons d'entrées/sorties et fichiers ouverts, un message d'avertissement est affiché dans le panneau de messages et les documents ouverts sont automatiquement fermés.
- L'instruction de fermeture (**FERMER**) ne fait aucune distinction entre les modes d'accès (i.e. les canaux ouverts en mode **LECTURE**, **ÉCRITURE** et **AJOUT** sont fermés de façon identique).
- Un document ouvert doit être fermé qu'une seule fois (une deuxième instruction de fermeture visant un même canal cause l'arrêt d'exécution de l'algorithme).
- Une instruction **FERMER** impliquant un canal d'entrées/sorties invalide (par exemple, un canal non associé à un tampon d'entrées/sorties ou un fichier) cause l'arrêt d'exécution de l'algorithme.

L'instruction **FERMER** permet de fermer plus d'un canal d'entrées/sorties.

3.1.5. Lecture via un canal d'entrées/sorties

La lecture de données à partir d'un fichier se fait via le canal d'entrées/sorties associé au document en question :

```
OUVRIR "DONNEES" SUR 3 EN LECTURE
LIRE Nom, Numéro, Salaire DE 3
```

La syntaxe d'une instruction **LIRE** visant un canal d'entrées/sorties est semblable à une instruction de lecture visant le clavier; il suffit de spécifier le canal d'entrées/sorties visé à l'aide du mot réservé **DE** dans un pseudo-code ou de spécifier le numéro de canal dans la *fenêtre d'édition d'instruction d'organigramme*

3.1.6. Écriture via un canal d'entrées/sorties

L'écriture de résultats dans un fichier se fait via le canal d'entrées/sorties associé au document visé

La syntaxe d'une instruction **ÉCRIRE** visant un canal d'entrées/sorties est semblable à une instruction d'écriture visant la console d'exécution; il suffit d'ajouter le canal d'entrées/sorties visé à l'aide du mot réservé **DANS** dans un pseudo-code ou de spécifier le numéro de canal dans la fenêtre d'édition d'instruction d'organigramme

3.1.7. Exemples d'applications

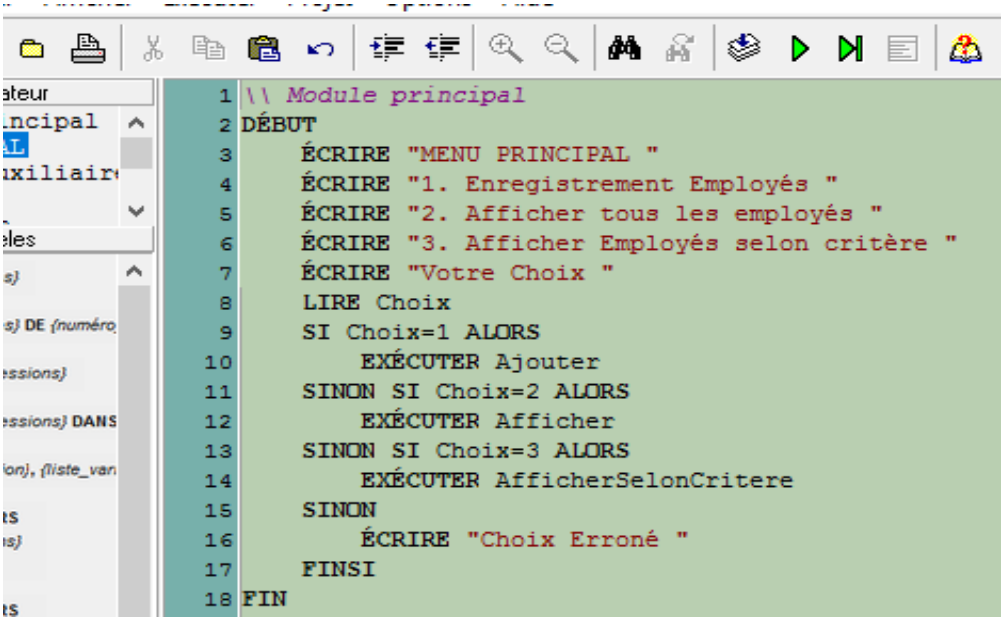
Question 1 : Soit un fichier "Personnel.txt" sur disque dont la structure est : Nom employé, N° matricule, sexe et Montant (le sexe est M pour les hommes et F pour les femmes). Concevoir un algorithme et un ordinogramme qui qui affiche le Menu suivant :

1. Enregistrement Employés
2. Afficher tous les Employés
3. Afficher Employés selon critère
   Votre Choix :

Le premier menu permet d'enregistrer les données dans le fichier, le deuxième permet d'afficher le contenu de tout le fichier et la troisième affiche uniquement les noms et matricules des femmes qui ont un montant inférieur à 25000 Fc.

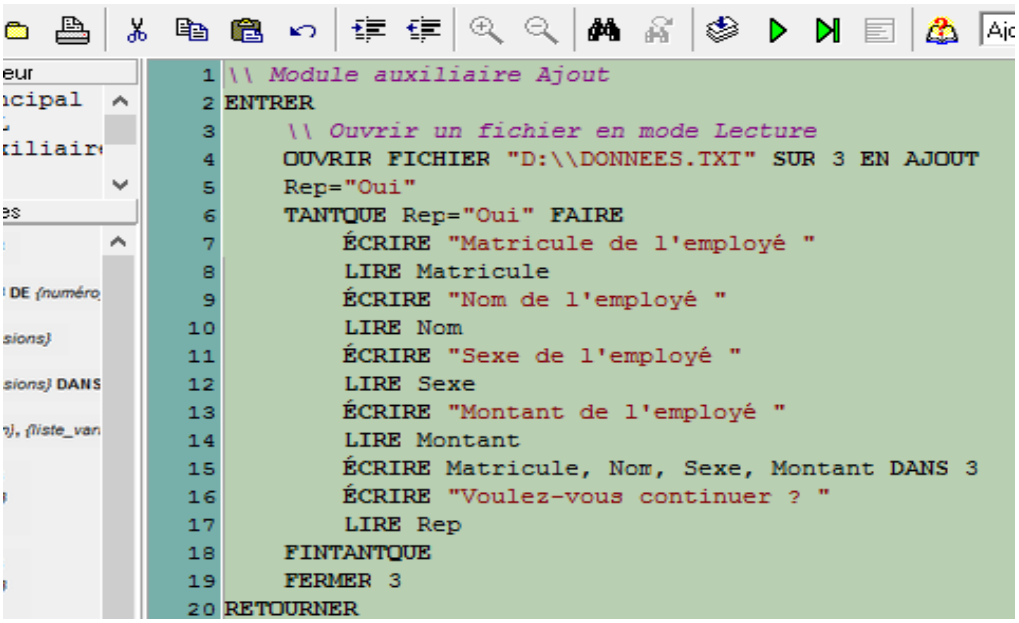
Solution

1. **Programme Principal**



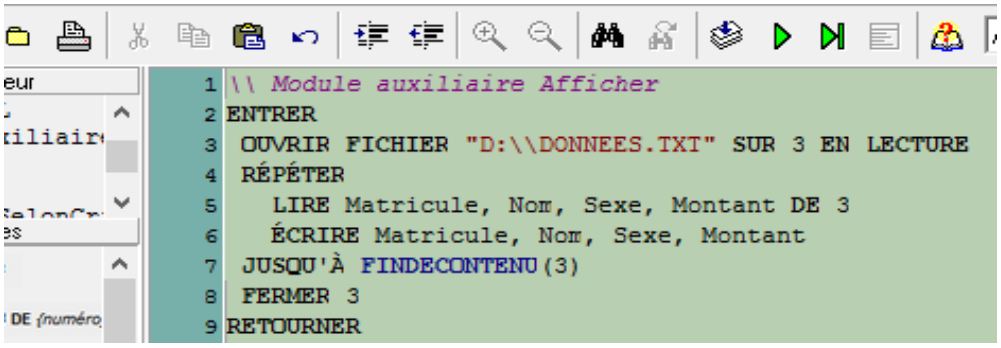
```
\\ Module principal
DÉBUT
    ÉCRIRE "MENU PRINCIPAL "
    ÉCRIRE "1. Enregistrement Employés "
    ÉCRIRE "2. Afficher tous les employés "
    ÉCRIRE "3. Afficher Employés selon critère "
    ÉCRIRE "Votre Choix "
    LIRE Choix
    SI Choix=1 ALORS
        EXÉCUTER Ajouter
    SINON SI Choix=2 ALORS
        EXÉCUTER Afficher
    SINON SI Choix=3 ALORS
        EXÉCUTER AfficherSelonCritere
    SINON
        ÉCRIRE "Choix Erroné "
    FINSI
FIN
```

2. Sous-Programme Ajouter



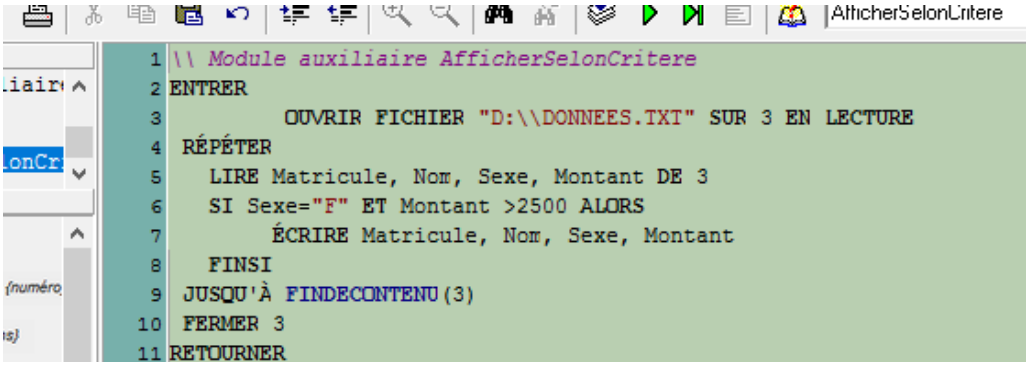
```
1  \\ Module auxiliaire Ajout
2  ENTRER
3      \\ Ouvrir un fichier en mode Lecture
4      OUVRIR FICHIER "D:\\DONNEES.TXT" SUR 3 EN AJOUT
5      Rep="Oui"
6      TANTQUE Rep="Oui" FAIRE
7          ÉCRIRE "Matricule de l'employé "
8          LIRE Matricule
9          ÉCRIRE "Nom de l'employé "
10         LIRE Nom
11         ÉCRIRE "Sexe de l'employé "
12         LIRE Sexe
13         ÉCRIRE "Montant de l'employé "
14         LIRE Montant
15         ÉCRIRE Matricule, Nom, Sexe, Montant DANS 3
16         ÉCRIRE "Voulez-vous continuer ? "
17         LIRE Rep
18     FINTANTQUE
19     FERMER 3
20 RETOURNER
```

3. Sous-Programme Afficher



```
1  \\ Module auxiliaire Afficher
2  ENTRER
3   OUVRIR FICHIER "D:\\DONNEES.TXT" SUR 3 EN LECTURE
4   RÉPÉTER
5     LIRE Matricule, Nom, Sexe, Montant DE 3
6     ÉCRIRE Matricule, Nom, Sexe, Montant
7   JUSQU'À FINDECONTENU(3)
8   FERMER 3
9  RETOURNER
```

4. Sous-Programme Afficher selon critère



```
1  \\ Module auxiliaire AfficherSelonCritere
2  ENTRER
3          OUVRIR FICHIER "D:\\DONNEES.TXT" SUR 3 EN LECTURE
4   RÉPÉTER
5     LIRE Matricule, Nom, Sexe, Montant DE 3
6     SI Sexe="F" ET Montant >2500 ALORS
7          ÉCRIRE Matricule, Nom, Sexe, Montant
8     FINSI
9   JUSQU'À FINDECONTENU(3)
10  FERMER 3
11 RETOURNER
```

Question 2 : Vous disposez d'un fichier, où chaque enregistrement contient les informations : (Nom, prix unitaire, quantité, adresse) ; On vous demande d'écrire un algorithme et un programme qui permet de donner pour chaque client, le nom, le prix unitaire, la quantité et le prix d'achat. Une remise de 10% est accordée, si le prix d'achat dépasse 100000 Fc.

3.2. Les Listes chainées

Parmi les structures de données linéaires il y a :

- ✓ les tableaux,
- ✓ les listes chaînées,
- ✓ les piles,
- ✓ les files.

Les structures de données linéaires induisent une notion de séquence entre les éléments les composant (1er, 2ème, 3ème, suivant, dernier…).

3.2.1. Définitions

Une **liste chaînée** est une structure linéaire qui n'a pas de dimension fixée à sa création. Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs. Sa dimension peut être modifiée selon la place disponible en mémoire. La liste est accessible uniquement par sa tête de liste c'est-à-dire son premier élément.

Un **élément** d'une liste est l'ensemble (ou structure) formé :

- ✓ D'une donnée ou information,
- ✓ D'un pointeur nommé *Suivant* indiquant la position de l'élément le suivant dans la liste.

A chaque élément est associée une adresse mémoire.

Les listes chaînées font appel à la notion de variable dynamique. Une variable dynamique :

- ✓ Est déclarée au début de l'exécution d'un programme,
- ✓ Elle y est créée, c'est-à-dire qu'on lui alloue un espace à occuper à une adresse de la mémoire,
- ✓ Elle peut y être détruite, c'est-à-dire que l'espace mémoire qu'elle occupait est libéré,
- ✓ L'accès à la valeur se fait à l'aide d'un pointeur.

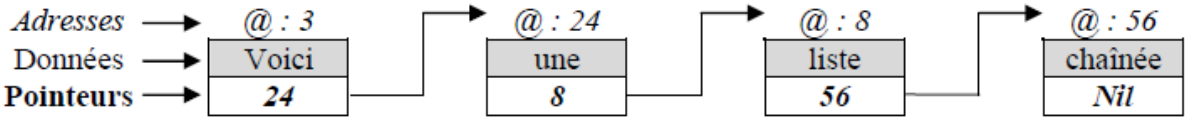
Un **pointeur** est une variable dont la valeur est une adresse mémoire. Un pointeur, noté P, pointe sur une variable dynamique notée P^.

Le **type de base** est le type de la variable pointée.
Le **type du pointeur** est l'ensemble des adresses des variables pointées du type de base. Il est représenté par le symbole ^ suivi de l'identificateur du type de base.

Pour les listes chaînées la séquence est mise en œuvre par le pointeur porté par chaque élément qui indique l'emplacement de l'élément suivant. Le dernier élément de la liste ne pointe sur rien (*Nul*).

On accède à un élément de la liste en parcourant les éléments grâce à leurs pointeurs.

Soit la liste chaînée suivante (@ indique que le nombre qui le suit représente une adresse) :



Pour accéder au troisième élément de la liste il faut toujours débuter la lecture de la liste par son premier élément dans le pointeur duquel est indiqué la position du deuxième élément. Dans le pointeur du deuxième élément de la liste on trouve la position du troisième élément…

Pour ajouter, supprimer ou déplacer un élément il suffit d'allouer une place en mémoire et de mettre à jour les pointeurs des éléments.

Il existe différents types de listes chaînées :

- ✓ **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
- ✓ **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la liste reste triée.
- ✓ **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. Ceci permet de lire la liste dans les deux sens, du premier vers le dernier élément ou inversement.
- ✓ **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.

Ces différents types peuvent être mixés selon les besoins.

On utilise une liste chaînée plutôt qu'un tableau lorsque l'on doit traiter des objets représentés par des suites sur lesquelles on doit effectuer de nombreuses suppressions et de nombreux ajouts. Les manipulations sont alors plus rapides qu'avec des tableaux.

Résumé

Structure	Dimension	Position d'une information	Accès à une information
Tableau	Fixe	Par son indice	Directement par l'indice
Liste chaînée	Evolue selon les actions	Par son adresse	Séquentiellement par le pointeur de chaque élément

3.2.2. Traitements de base d'utilisation d'une liste chaînée simple

Les traitements des listes sont les suivants :

- ✓ Créer une liste.
- ✓ Ajouter un élément.
- ✓ Supprimer un élément.
- ✓ Modifier un élément.
- ✓ Parcourir une liste.
- ✓ Rechercher une valeur dans une liste.

3.3. La notion de complexité

Quand un programmeur a besoin de résoudre un problème informatique, il écrit (généralement) un programme pour cela. Son programme contient une *implémentation*, c'est-à-dire si on veut une "transcription dans un langage informatique" d'un algorithme : l'algorithme, c'est juste une description des étapes à effectuer pour résoudre le problème, ça ne dépend pas du langage ou de l'environnement du programmeur ; de même, si on traduit une recette de cuisine dans une autre langue, ça reste la "même" recette.

3.3.1. Correction de l'algorithme

Que fait, ou que doit faire un programmeur qui implémente un algorithme ? Comme Haskell le fermier, il doit commencer par vérifier que son algorithme est *correct*, c'est-à-dire qu'il produit bien le résultat attendu, qu'il résout bien le problème demandé.
C'est très important (si l'algorithme ne fait pas ce qu'on veut, on n'a pas besoin de chercher à l'optimiser), et c'est parfois l'étape la plus compliquée.

Dans la pratique, la plupart des informaticiens "font confiance" à leurs algorithmes : avec un peu d'habitude et pour des problèmes abordables, un programmeur expérimenté peut se convaincre qu'un algorithme fonctionne correctement, ou au contraire trouver un problème s'il y en a ("et que se passe-t-il si tu as un nombre impair de grenouilles ?").

L'approche plus 'sérieuse' consiste à écrire une preuve que l'algorithme est correct. Il y a différents niveaux de preuves, mais ils sont tous un peu trop formels pour ce tutoriel, et nous n'aborderons sans doute pas (ou alors très brièvement) cet aspect.

Bien sûr, un algorithme correct ne veut pas dire un programme sans bug : une fois qu'on a un algorithme correct, on l'implémente (en écrivant un programme qui l'effectue), et on est alors exposé à toutes les petites erreurs, en partie spécifiques au langage de programmation utilisé, qui peuvent s'incruster pendant l'écriture du programme. Par exemple, l'algorithme ne décrit pas en général comment gérer la mémoire du programme, et la vérification des erreurs de segmentations et autres réjouissances de la sorte est laissée aux soins du programmeur.

3.3.2. Complexité

Une fois que le programmeur est convaincu que son algorithme est correct, il va essayer d'en évaluer l'efficacité. Il veut savoir par exemple, "est-ce que cet algorithme va vite ?".

On pourrait penser que la meilleure façon de savoir ce genre de choses est d'implémenter l'algorithme et de le tester sur son ordinateur. Curieusement, ce n'est généralement pas le cas. Par exemple, si deux programmeurs implémentent deux algorithmes différents et mesurent leur rapidité chacun sur son ordinateur, celui qui a l'ordinateur le plus puissant risque de penser qu'il a l'algorithme le plus rapide, même si ce n'est pas vrai.

De plus, cela demande d'implémenter l'algorithme avant d'avoir une idée de sa rapidité, ce qui est gênant (puisque la phase d'implémentation, d'écriture concrète du code, n'est pas facile), et même pas toujours possible : si le problème que l'on veut résoudre est lié à une centrale nucléaire, et demande d'utiliser les capteurs de la centrale pour gérer des informations, on peut difficilement se permettre d'implémenter pour tester en conditions réelles tous les algorithmes qui nous passent par la tête.

Les scientifiques de l'informatique ont créé pour cela un outil extrêmement pratique et puissant, que nous étudierons dans la suite de ce tutoriel : la **complexité algorithmique**.

Le terme de 'complexité' est un peu trompeur parce qu'on ne parle pas d'une difficulté de compréhension, mais d'efficacité : "complexe" ne veut pas dire "compliqué".