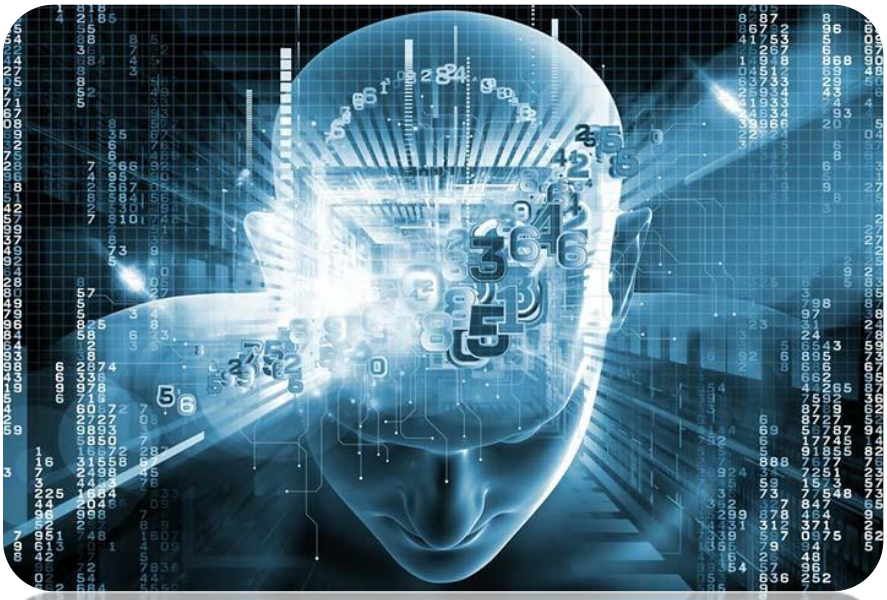
République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche
Université des Sciences et de la Technologie
d'OranFaculté des mathématiques et informatique
Département des Mathématiques

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي و البحث العلمي
جامعة وهران للعلوم والتكنولوجيا
كلية الرياضيات والإعلام الآلي
قسم الرياضيات



Algorithmique et Structures de Données

POLYCOPIE DE COURS + EXERCICES CORRIGES

Dr. Redouane TLEMSANI

TABLE DES MATIERES

INTRODUCTION GENERALE

L'algorithmique et les structures de données sont des sujets centraux en informatique et en sciences de l'information. L'algorithmique concerne l'étude de la conception et de l'analyse des algorithmes, qui sont des séquences d'instructions pour résoudre des problèmes. Les structures de données sont des moyens de stocker et d'organiser des données de manière efficace pour permettre un traitement rapide et une manipulation facile des informations.

Les algorithmes et les structures de données sont utilisés dans de nombreuses applications informatiques, de la conception de logiciels à la programmation de sites web en passant par l'analyse de données et la science des données. Ils sont également utilisés dans d'autres domaines, tels que les sciences sociales et naturelles, l'ingénierie, les mathématiques, la finance et la gestion.

Une connaissance solide de l'algorithmique et des structures de données est essentielle pour tout étudiant ou professionnel de l'informatique. Cela permet de résoudre des problèmes efficacement, de développer des logiciels robustes et d'améliorer les performances des applications.

Dans ce polycopié, on va entamer la notion des variables qui sont des éléments de base pour stocker des données en mémoire. Elles peuvent être de différents types, tels que des nombres, des chaînes de caractères, des tableaux ou des objets. Les variables permettent aux programmes de conserver des valeurs qui peuvent être utilisées pour effectuer des opérations et résoudre des problèmes. Les structures de contrôle vont été détaillées, telles que les boucles et les conditions, sont également des éléments clés de la programmation. Elles permettent de contrôler le flux d'exécution des instructions et de prendre des décisions en fonction des résultats des opérations.

L'utilisation de fonctions est un élément important essentiel traité dans ce document. Les fonctions sont des blocs de code qui peuvent être appelés à partir d'autres parties du programme pour effectuer une tâche spécifique. Les fonctions peuvent prendre des paramètres en entrée, effectuer des opérations et renvoyer des résultats. Les fonctions sont souvent utilisées pour réduire la duplication de code et améliorer la lisibilité et la maintenabilité du code.

Les trois objectifs majeurs de cette matière résume dans :

- Acquisition des techniques de base essentielles à la réalisation d'un programme informatique.
- Etude des structures de données et les algorithmes fondamentaux,
- Maîtrise les techniques algorithmiques de base et l'analyse de leur complexité.

En somme, l'étudiant devra avoir une connaissance approfondie des variables, des structures de contrôle et de l'utilisation des fonctions. Ces concepts sont fondamentaux pour concevoir des programmes efficaces, fiables et faciles à comprendre et à modifier. Les programmes bien conçus peuvent réduire les coûts, accélérer les opérations et améliorer la qualité des résultats.

1. LES VARIABLES

1.1 APERÇU

L'écriture d'un programme est une opération complexe qui requiert de nombreuses étapes. Le plus important est de comprendre l'objectif final et de le respecter. Pour cela, il est souvent préférable de décomposer le traitement souhaité en une succession d'opérations plus petites et plus simples.

1.2. STRUCTURE D'UN ALGORITHME :

Les algorithmes ont pour vocation de nous faire réfléchir, mais pas de d'exécuter sur un ordinateur : pour cela, il sera nécessaire de traduire l'algorithme dans un langage de programmation.

1.2.1 Définitions

Algorithme

Un algorithme est une suite d'opérations élémentaires permettant d'obtenir le résultat final déterminé à un problème.

La structure d'un algorithme est la suivante :

```
Algorithme  nom-de-l'algorithme      // partie entête
Debut                                // partie traitement
        Bloc d'instructions ;
Fin
```

Propriété d'un algorithme :

Un algorithme, dans des conditions d'exécution similaires (avec des données identiques) fournit toujours le même résultat.

Bloc d'instructions :

*Un bloc d'instructions est une partie de traitement d'un algorithme, constituée d'opérations élémentaires situées entre **Début** et **Fin** ou entre **accolades.***

- Chaque ligne comporte une seule instruction
- L'exécution de l'algorithme correspond à la réalisation de toutes les instructions, ligne après ligne, de la première à la dernière, dans cet ordre.

Commentaires :

Les commentaires sont des explications textuelles dans l'algorithme par le programmeur à la suite des deux caractères //.

Ils ne sont pas exécutés : ils sont invisibles de l'exécution de l'algorithme.

Ces commentaires seront utiles aux programmeurs qui veulent comprendre ou modifier l'algorithme.

```
Algorithme  algo_bonjour
    Debut
            Ecrire(«bonjour tout le monde ») ;
    Fin

Algorithme  algo_bonjour2
    Debut
            Ecrire(«bonjour») ;
            Ecrire(«tout le monde ») ;
    Fin
```

Même résultat mais le premier algorithme est le plus simple !!

1.3. LES DONNEES

1.3.1. Déclaration et utilisation des variables

Les valeurs, pour pouvoir être manipulées, sont stockées dans des variables.

Variable :

Une variable désigne un emplacement mémoire qui permet de stocker une valeur. Une variable est définie par :

- *un nom unique*
- *un type de définition unique*
- *une valeur attribuée et modifiée au cours du déroulement de l'algorithme*

1.3.2. La syntaxe

La structure d'un algorithme (déclarant une variable nommée indice et de type entier) est alors la suivante :

```
Algorithme  nom-de-l'algorithme        // partie entête
    Variables : indice : entier        //partie déclaration des variables
    Debut                              // partie traitement
            Bloc d'instructions ;
    Fin
```

Nom d'une variable- identifiant d'une variable :

Le nom d'une variable permet de l'identifier de manière unique au cours de l'algorithme.

1.3.3. Le nom – le type – la valeur

Pour faciliter la lecture des algorithmes, il convient de respecter des règles pour nommer les variables :

- Commence par une minuscule
- Ne comporte pas d'espace
- Si le nom de variable est composé de plusieurs mots, il faut faire commencer chacun d'eux par une majuscule et ne pas faire figurer de traits d'union. (exemple : laVitesse, valeurMaxOuMin)
- Il faut également faire attention à bien donner aux variables un nom explicite (proscrire i2, zz2,..)

Type- domaine de définition

Le type (appelé aussi domaine de définition) de la variable indique l'ensemble des valeurs que la variable peut prendre.

- Les variables peuvent appartenir à plusieurs domaines (entier, réel, caractère, booléen, etc,).
- Les variables changent de valeur grâce à l'opération d'affectation.

Affectation :

L'affectation est une opération qui fixe une nouvelle valeur à une variable. Le symbole de l'affectation est ←.

1.3.4. Détermination des variables

Soit le problème suivant : calculer et écrire le double d'un nombre réel donné.
La structure de l'algorithme est la suivante :

```
Algorithme  double
Variables : nombre, resulat : réel ;
    Debut
            nombre ←7 ;
            resulat ←nombre x 2 ;
            ecrire(resultat) ;
    Fin
```

L'algorithme se déroule de manière séquentielle :

Algorithme **double**	
Variables : **nombre, resulat :** réel ;	Déclaration des variables
Debut	nombre= ? resulat= ?
nombre ←7 ;	nombre= **7** resulat= ?
resulat ←nombre x 2 ;	nombre= **7** resulat= **14**
ecrire(resultat) ;	nombre= **7** resulat= **14**
Fin	Les variables n'existent plus

1.3.5. Les erreurs à éviter

Les erreurs les plus courantes concernant l'utilisation des variables sont :

- Une variables n'est déclarée qu'une seule fois dans un algorithme
- Une variable est déclarée au début de l'algorithme et non dans la partie réservée aux instructions de traitement.
- Avant de pouvoir utiliser une variable, il faut l'avoir déclarée dans le bloc des variables.
- Avant de pouvoir utiliser la valeur d'une variable, une valeur doit lui être attribué.

```
Algorithme variables-erreurs
Variables : nombre, resulat : réel ;
    Debut
            resulat ←nombre x 2 ;   //erreur : nombre n'a pas de valeur
            valeur←1 ;              //erreur : valeur n'a pas été définie
    Fin
```

1.3.2. Types des variables

1.3.2.1. Le type réel et le type entier

Le type réel – le type entier :

les variables de type numérique utilisées dans l'algorithme ont comme domaines usuels ceux fournis par les mathématiques : réel ou entier.

```
Algorithme type-réel
Variables : nombre1, nombre2, resulat : réel ;
            var1 : entier ;
    Debut
            nombre1←1.2;
            nombre2←15 ;
            var1←2 ;
            resulat ←nombre1/nombre2  x  var1
    Fin
```

1.3.2.2. Conversion

Convertir un entier en réel est naturel : cette opération n'entraine pas de perte d'information ; Par exemple, l'entier 15 deviendra 15.0.

Convertir un réel en entier entraine une perte d'information : les chiffres décimaux sont perdus. Par exemple, le réel 15.75 deviendra 15.

```
Algorithme conversion-numérique
Variables : nombre1: entier ;
            nombre2 : réel ;
    Debut
            nombre1←15;
            nombre2←nombre1;            //nombre2 vaut 15.0
            nombre1←nombre2 + 0.5;      //erreur : imposible
    Fin
```

1.3.2.3. Le type caractère

Le type caractère :

Il s'agit du domaine constitué des caractères alphabétiques, numériques et de ponctuation.

Il ne faut pas confondre entre le caractère '3' et l'entier 3 !!

Les seules opérations élémentaires pour les éléments de type caractère sont les opérations de comparaison :

$$> \quad < \quad \neq \quad = \quad \geq \quad \leq$$

En fait, à chaque caractère est associé une unique valeur numérique entière (le code ASCII établit cette correspondance : par exemple, la lettre 'A' correspond à la valeur 65).
Pour écrire un algorithme, nous ne devons pas connaitre par cœur les valeurs de la table ASCII. Mais nous utiliserons trois principes :

- Les entiers correspondant aux caractères 'A','B'…'Z' se suivent dans cet ordre.
- Les entiers correspondant aux caractères 'a','b'…'z' se suivent dans cet ordre.
- Les entiers correspondant aux caractères numérique'0' à '9' se suivent dans cet ordre.

1.3.2.3. Conversion

Alors pour convertir un caractère minuscule en majuscule, il suffit de luis ajouter la différence qui les sépare : 'c'+ ('A'-'a') vaut 'C'

La conversion de type caractère vers entier : pour convertir le caractère '3' en une valeur entière 3, il suffit de calculer la différence entre les deux caractères : '3'- '0', qui vaut 3.

La conversion de type entier vers caractère : pour convertir l'entier 3 en une valeur le caractère '3', il suffit de calculer la somme entre les deux caractères : 3+'0', qui vaut '3'.

Exemple simple :

```
Algorithme conversion-caractère-entier
Variables : nombre1: entier ;
            car : caractere ;
   Debut
            car←'3';
            nombre←'3' – '0';          //nombre vaut 3
            nombre←nombre + 2;         // nombre vaut 5
            car ← '0' + nombre ;       //car vaut '5'
   Fin
```

1.3.2.4. Le type logique booléen

Le type booléen :

Le domaine des booléens est l'ensemble formé des deux seules valeurs (vrai, faux).

Les opérations admissibles sur les éléments de ce domaine sont réalisées à l'aide de tous les connecteurs logiques, notés :

- ET : pour le « et logique »
- OU : pour le « ou logique »
- NON : pour le « non logique »

La table de vérité donne la réponse « Vrai » ou « Faux » des opérations logiques.

Opération ET	Faux	Vrai
Faux	Faux	Faux
Vrai	Faux	Vrai

Opération OU	Faux	Vrai
Faux	Faux	Vrai
Vrai	Vrai	Vrai

Exemples :

```
Algorithme type-booléen
Variables : booleen1, booleen2: booléen ;
    Debut
        booleen1←5<6;                  //booleen1 prend la valeur Vrai
        booleen2←NON booleen1;         //booleen2 prend la valeur Faux
        booleen2←(5<7) OU (3>8) ;      //booleen2 prend la valeur Vrai
        booleen1←vrai;                 //booleen1 prend la valeur Vrai
    Fin
```

1.3.2.5. Les erreurs à éviter

Lors d'une affectation, la valeur de la partie droite doit obligatoirement être du type de la variable dont la valeur est modifiée.

```
Algorithme type-erreur
Variables : car : caractere ;
        Debut
            car←1.56;                  //erreur : car n'est pas un réel
            car←5<8 ;                  // erreur : car n'est pas un booléen   Fin
```

1.4. FONCTIONS D'ENTREE-SORTIE

Les programmes utilisent fréquemment des instructions permettant l'affichage à l'écran et la saisie de valeurs au clavier par l'utilisateur. Nous allons nous munir de deux opérations analogues permettant de simuler :

- L'affichage d'une phrase avec l'instruction ecrire() ;
- La saisie d'une valeur par l'utilisateur avec l'instruction lire().

1.4.1. . La fonction lire

L'instruction de saisie de données par l'utilisateur est :

```
lire(nomDeLaVariable)
```

L'exécution de cette instruction consiste à :

1. Demander à l'utilisateur de saisir une valeur sur le périphérique d'entrée
2. Modifier la valeur de la variable passé entre parenthèses

1.4.2. La fonction ecrire

L'instruction d'affichage à l'écran (le périphérique de sortie) d'une expression est :

```
ecrire(expression)
```

Cette instruction réalise simplement l'affichage de l'expression passé entre parenthèses.

```
Algorithme exemple-lire-ecrire
Variables : nb : réel ;
    Debut
        lire(nb)                       //l'utilisateur saisie le nombre au clavier
        ecrire(''la valeur de nb '') ; //une phrase est affichée à l'écran
        ecrire(nb) ;                   // une valeur est affichée à l'écran
        ecrire(''la valeur de nb est : '', nb) ; //une phrase suivie de la valeur //sont
                                                  affichés à l'écran
    Fin
```

1.4.3. Exercice

Ecrire un algorithme qui demande à l'utilisateur de saisir au clavier trois nombres réels et qui affiche à l'écran la somme de ces trois nombres ;

```
Algorithme somme-trois-réels
Variables : nb1, nb2, nb3, somme : réel ;
    Debut
        lire(nb1) ; lire(nb2) ;lire(nb3) ;
        somme←nb1+nb2+nb3 ;
        ecrire(somme) ;
    Fin
```

1.5. LES TYPES OBJET : UNE BOITE A OUTILS

1.5.1. Les chaines de caractères

1.5.1.1. Présentation de la classe Chaine

Une chaine de caractères est composée de caractères alphanumériques formant un mot ou une phrase. Il est impossible de manipuler les chaines de caractères avec les opérations usuelles définies pour les réels ou les entiers : la classe ***Chaine*** nous fournit donc des opérations spécifiques.

L'interface utilisateur de la classe chaine(l'ensemble des opérations définies pour les manipuler) est décrite comme suit :

- Chaine() permet de créer une chaine en mémoire
- Ecrire() : vide permet d'écrire la chaine sur l'écran
- Lire() :vide permet à l'utilisateur de saisir le contenu de la chaine
- Longueur() :entier fournit le nombre de caractères de la chaine
- iemeCar(entier) :caractère fournit le caractère qui est à la position passée en paramètre (le premier est à la position 0)
- modifierIeme(entier, caractere) :vide remplace le caractère situé à la position donnée en paramètre (le premier caractère est à la position 0)
- concatener(Chaine) : vide modifie la chaine en lui juxtaposant la chaine passée en mémoire

1.5.1.2. Utilisation d'une chaine

Expliquons comment utiliser une chaine dans un algorithme à travers des exemples de déclaration et d'utilisation. La manipulation d'une chaine nécessite deux étapes :

1. Il faut créer la chaine. En effet, la simple déclaration dans le bloc variable ne suffit pas à la créer.
2. On peut alors utiliser la chaine grâce aux méthodes de l'interface utilisateur.

Déclaration

On déclare une variable de type Chaine de la manière suivante :

Variables : nomDeLaVariable : Chaine ;

Ensuite, on construit effectivement l'objet (appelé aussi instance) dans le corps du programme avec l'opération new :

nomDeLaVariable ← new Chaine() ;

On doit initialiser les chaines avant de les utiliser. Par exemple, les variables *nom, prenom* et *frere* dont initialisées par les trois méthodes suivantes :

Algorithme creation-de-la-chaine
variables : nom, prenom, frere : Chaine
Debut
nom← new Chaine(‘‘Tlemsani ‘‘) ;
//la variable nom est initialisée et contient ‘‘Tlemsani’’

frere ← new Chaine(nom) ;
//la variable frere contient aussi ‘‘Tlemsani’’
prenom ← new chaine() ;
//cette chaine est initialisée, mais vide **Fin**

Chaque instruction new déclenche dans la mémoire la création d’une zone réservée schématisée par une case contenant la chaine créée. Représentons l’état de la mémoire à la fin de l’exécution de l’algorithme précédent.

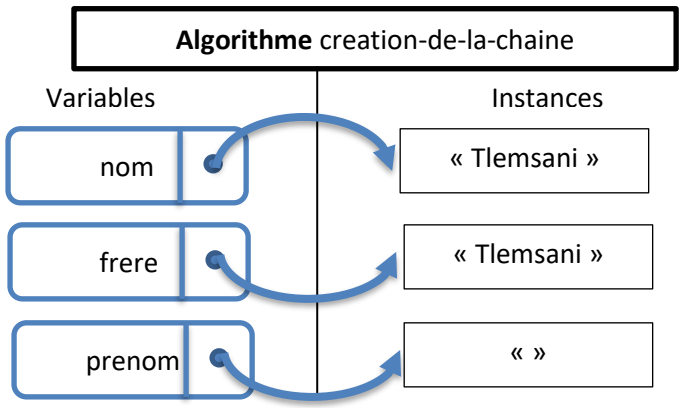


Figure 1 ***: Etat de la mémoire***

Utilisation

La méthode fonctionne en association avec une case spécifique (précisée lors de l’appel) du schéma mémoire.

Algorithme utilisation-de-la-chaine
variables : nom, prenom: Chaine ;
lg : entier ;
car : caractere ;
Debut
nom← new Chaine() ;
nom.lire() ; //l’utilisateur saisie ce qu’il veut
lg ← prenom.longueur() ; //lg contient la longueur du mot nom
car← nom.iemeCar(3) ; //car contient la 4ième lettre du mot nom //précisons que la première est à l’indice 0
nom.modifierIeme(5,’t’) ; //la 6ième lettre change
nom.ecrire() ; //on écrit le mot
Fin

Les erreurs à éviter :

- Ne pas oublier de créer la chaine avant de lui appliquer une opération
- Préciser sur quelle chaine est appliquée l’opération (ne pas oublier le pint «. » devant une méthode(instance).
- Toujours mettre des parenthèses
- Utiliser des paramètres convenables pour les opérations iemeCar et modifierIeme
- Pour lire et écrire des chaines, utiliser les méthodes lire et ecrire et non les fonctions d’entrée-sortie lire et ecrire.

Voici un algorithme avec cinq erreurs d’utilisation des méthodes :

```
Algorithme utilisation-de-la-chaine-avec-des erreurs
variables : nom: Chaine ;
            lg : entier ;
            car : caractere ;
Debut
    nom← new Chaine() ;
    lg ←prenom.longueur() ;        //la chaine prenom n'a pas été créée
lg ← longueur() ;            //la longueur de quoi ?
lg ← nom.longueur;                 //il manque les parenthèses
car← nom.iemeCar() ;       //il faut préciser le numéro de la lettre
    lire(nom) ;                    //il faut utiliser l'opération : nom.lire()
    ecrire(nom) ;                  //il afut utiliser l'opération : nom.ecrire() ;
Fin
```

1.5.2. LES SCHEMAS MEMOIRE

Il est primordial de connaitre l'état des variables en cours d'exécution d'un algorithme grâce à un schéma mémoire.

Schéma mémoire :

Le schéma mémoire d'un algorithme représente l'ensemble des variables et de leurs valeurs à une étape précise de son déroulement.

Un schéma mémoire délimite trois parties distinctes :

1. Le nom de l'algorithme
2. La partie des variables où toutes les variables définies seront représentées par :
   - Une valeur(ou par « ? » si la variable n'a pas encore de valeur) pour les variables de type primitifs ;
   - Une flèche(pointant sur une case dessinée dans la partie droite du schéma) pour les variables de type objet Chaine.
3. La partie des instances où chaque case aura été créée par l'utilisation de l'opérateur new : il y a autant de cases à représenter qu'il y a de new dans l'algorithme.

Certaines erreurs courantes sont à éviter :

- Oublier de représenter une variable ;
- Donner une mauvaise valeur à une variable ;
- Représenter les chines sans une case associée ;
- Oublier de préciser l'étape (au cours du déroulement de l'algorithme) représentée par le schéma mémoire.

1.6. LE TYPE TABLEAU

Le type tableau permet de stocker des valeurs de même type grâce à une seule variable.

1.6.1. Déclaration d'un tableau

Le type tableau :

Un tableau structure un ensemble de valeurs de même type accessibles par leur position.

Un tableau est défini en deux temps. Le tableau nommé *tab* est déclaré de la manière suivante, ainsi que le type de base de ses éléments. C'est une déclaration de variable :

Variable : tab : tableau[] de domaine ;

Puis l'instruction dans le corps du programme qui implémente effectivement le tableau , c'est-à-dire qui réserve de la place en mémoire, est construite avec un opérateur ***new*** agissant dans l'environnement d'exécution du programme :

tab← new domaine[10) ; // le tableau tab a une dimension de 10 éléments

L'exemple suivant représente un tableau tab de 10 entiers :

Dimension, type et indice d'un tableau :

Le nombre maximal d'éléments du tableau, qui est précisé à la définition, s'appelle sa dimension. Le type de ses éléments s'appelle le type du tableau. Pour accéder aux éléments d'un tableau, un indice indique le rang de l'élément.

tab=	**22**	**-7**	**56**	**12**	**0**	**23**	**-4**	**1**	**57**	**35**
indices	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A l'aide d'une seule variable tableau tab, il est possible de manipuler plusieurs valeurs différentes. Par exemple : **tab[0)** est vu comme une variable indépendante ayant la valeur **22**.

1.6.2. Utilisation d'un tableau

1.6.2.1. Tableau à une dimension

La manipulation des éléments du tableau tab est décrite dans l'exemple suivant : une seule variable permet de stocker 4 notes entières.

Algorithme utilisation tableau
Variables : notes : tableau[] d'entiers ;
Debut
notes←new entier[4] ;
notes[0]←12 ;
notes[1]←14 ;
notes[2]←10 ;
notes[3]←18 ;
Fin

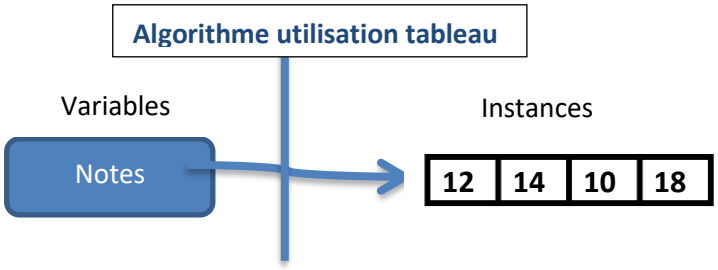


Figure 2: Schéma mémoire du tableau notes

1.6.1.2. Tableau à deux dimensions

Si nous désirons écrire un programme qui travaille avec un damier de 3 cases sur 3 contenant des entiers, nous introduirons une instance damier sous forme d'un tableau de 3 cases sur 3. Ecrivons l'algorithme modifiant trois éléments du tableau.

```
Algorithme damier
Variables : damier : tableau[][] d'entiers ;
Debut
    damier←new entier[3][3] ;
    damier[0][0)←0 ;
    damier[1][1)←1 ;
    damier[0][1)←1 ;
Fin
```

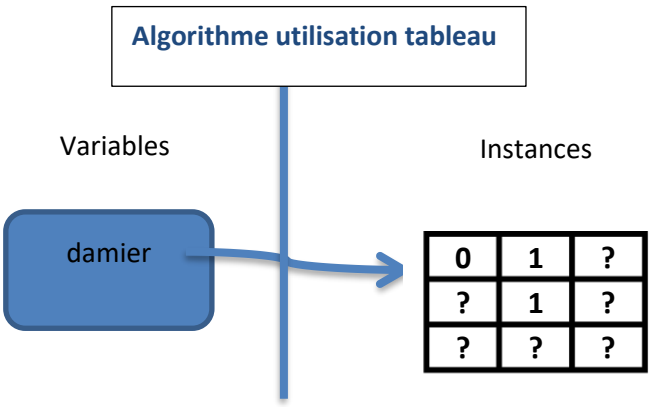


Figure 3: *Schéma mémoire du tableau damier*

1.6.3. Echanger deux variables

Savoir échanger le contenu de deux variables est une technique assez simple, dont la maitrise est nécessaire.

Imaginez que vous avez deux bouteilles : l'une contient de jus, l'autre de l'eau. Comment échanger leur contenu ? Tout simplement en passant par l'intermédiaire d'une troisième bouteille qui stockera temporairement le jus, le temps de transvaser l'eau.

Le principe est identique en algorithmique : il faut introduire une variable temporaire (du même type que les deux autres) qui stockera une valeur.

2. LES STRUCTURES DE CONTROLE

Nous allons introduire deux instructions extrêmement utilisées qui permettent de construire un algorithme au déroulement non linéaire. L'instruction conditionnelle permet d'exécuter ou non un bloc d'instructions. La boucle permet de revenir en arrière dans l'algorithme, pour réitérer un nombre de fois précis l'exécution d'un bloc.

Ces deux instructions reposent sur l'évaluation, par l'algorithme, d'une variable de type booléenne (vrai ou faux), qui conditionne la suite de son déroulement.

2.1. INSTRUCTION CONDITIONNELLE

2.1.1 La syntaxe

L'instruction conditionnelle nous autorise désormais à concevoir un algorithme qui n'exécutera pas certains blocs instructions.

La conditionnelle

L'instruction conditionnelle détermine si le bloc d'instructions suivant est exécuté ou non. La condition est une expression booléenne dont la valeur détermine le bloc d'instructions exécutées.

La syntaxe de cette instruction est :

```
Si (condition) alors
{
        Bloc d'instructions n°1 ;  //exécuté si condition égale Vrai
}
Sinon
{
        Bloc d'instructions n°2 ;  //exécuté si condition égale Faux
}
```

L'un des deux blocs est obligatoirement exécuté, l'autre ne le sera pas.
Ecrivons l'algorithme qui lit deux entiers et affiche le plus grand des deux.

```
Algorithme Max-de-deux-entiers
Variables : x,y, max : entier ;
Debut
    Lire(x) ;
    Lire(y) ;
    si (x>y) alors
    {
            max←x ;
    }
    sinon
    {
            max←y ;
    }
    ecrire(« le maximum est : », max) ;
Fin
```

Analysons le déroulement de l'algorithme ligne par ligne : prenons pour cela un exemple, supposons que l'utilisateur saisisse 5 pour x et 7 pour y.

	Les variables n'ont pas de valeur connue au début
Debut	x= ? y= ? max= ?
Lire(x) ;	x= 5 y= ? max= ?
Lire(y) ;	x= 5 y= 7 max= ?
si (x > y) alors	La condition est évaluée : (5>7) (Faux)
{	Ce bloc n'est pas exécuté
max←x ;	Ce bloc n'est pas exécuté
}	Ce bloc n'est pas exécuté
sinon	
{	Ce bloc est exécuté
max←y ;	x= 5 y= 7 max= 7
}	Fin du bloc conditionnel
ecrire(" maximum : ", max) ;	Affichage de maximum : 7
Fin	Les variables n'existent plus

2.1.2. Applications

2.1.2.1. La conditionnelle simple

Une version plus simple est utilisée si l'alternative n'a pas lieu. La syntaxe de cette instruction est alors :

Si (condition) **alors**
{
instructions ;
}

Ecrivons un algorithme qui lit un entier et affiche sa valeur positive.

Algorithme valeur-positive
Variables : valeur, positif : entier ;
Debut
lire(valeur) ;
positif←valeur ;
si (positif<0) alors
{
positif ← -1 x positive ;
}
ecrire(" la valeur positive est : ", positif) ;
Fin

Le même algorithme peut se passer de la variable positif contenant le résultat.

Algorithme valeur-positive
Variables : valeur: entier ;
Debut
lire(valeur) ;
si (valeur<0) alors
{
valeur ← -1 x valeur ;
}
ecrire(" la valeur positive est : ", valeur) ;
Fin

On peut simplifier l'écriture de l'instruction en omettant les accolades de délimitation de bloc, lorsqu'il n'y a pas d'ambiguïté (si le bloc ne se compose que d'une seule instruction).

```
Algorithme Max-de-deux-entiers
Variables : x,y, max : entier ;
Debut
    Lire(x) ;
    Lire(y) ;
    si (x>y) alors
    max←x ;
    sinon
    max←y ;
    ecrire(« le maximum est : », max) ;
Fin
```

2.1.2.2. La présentation

Les décalages dans l'écriture d'un algorithme (ou d'u programme) sont nécessaires à sa bonne lisibilité. Savoir présenter un algorithme, c'est montrer qu'on a compris son exécution.

```
Algorithme Max-de-deux-entiers
Variables : x,y, max : entier ;
Debut
Lire(x) ;
Lire(y) ;
si (x>y) alors
{
max←x ;
}
sinon
max←y ;
ecrire(« le maximum est : », max) ;
Fin
```

```
Algorithme Max-de-deux-entiers
Variables : x,y, max : entier ;
Debut
    Lire(x) ;
    Lire(y) ;
    si (x>y) alors
    {
        max←x ;
    }
    sinon
        max←y ;
    ecrire(« le maximum est : », max) ;
Fin
```

Figure 4*: Algorithme illisible et algorithme bien présenté*

2.2. CONDITIONNELLES IMBRIQUEES

2.2.1. L'usage

2.2.1.1. Exemple

Il est possible d'imbriquer des blocs de programme les uns dans les autres. Essayons de résoudre le problème consistant à faire lire à l'utilisateur une note et à afficher le commentaire associé à la note :

- note de 0 à 8 inclus : « Insuffisant » ;
- note de 8 à 12 inclus : « moyen » ;
- note de 12 à 16 inclus : « bi en » ;
- note de 16 à 20 inclus : « très bien » ;

La solution classique :

Algorithme commentaires-notes

```
Variables : note : entier ;
Debut
    Lire(note) ;
    si (note ≤ 8) alors
    ecrire(« insuffisant ») ;
    si (note > 8) ET (note ≤ 12)alors
    ecrire(« moyen ») ;
    si (note > 12) ET (note ≤ 16)alors
    ecrire(«bien ») ;
    si (note>16) alors
    ecrire(« très bien ») ;
Fin
```

Solution plus élégante :

Algorithme commentaires-notes-mieux

```
Variables : note : entier ;
Debut
    Lire(note) ;
    si (note ≤ 8) alors
            ecrire(« insuffisant ») ;
    sinon si  (note ≤ 12) alors
                    ecrire(« moyen ») ;
                    sinon si (note ≤ 16) alors
                            ecrire(«bien ») ;
                            sinon
                                    ecrire(« très bien ») ;
Fin
```

Il est possible de ne pas respecter la présentation avec les tabulations :

Algorithme commentaires-notes-mieux-bis

```
Variables : note : entier ;
Debut
    Lire(note) ;
    si (note ≤ 8) alors
            ecrire(« insuffisant ») ;
    sinon si  (note ≤ 12) alors
            ecrire(« moyen ») ;
    sinon si (note ≤ 16) alors
            ecrire(«bien ») ;
    sinon
            ecrire(« très bien ») ;
Fin
```

2.2.2. Erreur à éviter

Il est très fréquent chez les débutants d'oublier les instructions sinon intermédiaires.
Analysons l'algorithme suivant :

Algorithme commentaires-notes-faux
Variables : note : entier ;
Debut
Lire(note) ;
si (note ≤ 8) alors
ecrire(« insuffisant ») ;
si (note ≤ 12)alors
ecrire(« moyen ») ;
si (note ≤ 16)alors
ecrire(«bien ») ;
sinon
ecrire(« très bien ») ;
Fin

2.2.3. La présentation des conditionnelles

Il est capital d'écrire un algorithme aussi lisible et clair que possible.
Si (condition) **alors**
{
Bloc d'instructions n°1 ;
}
Sinon
{
Si (condition) **alors**
{
Bloc d'instructions n°2 ;
}
Sinon
{
Bloc d'instructions n°3 ; }
}

2.3. INSTRUCTION DE REPETITION

2.3.1. La boucle tant_que

2.3.1.1. Définition
2.3.1.2. Syntaxe :

La boucle

L'instruction de répétition, appelée boucle, permet d'exécuter plusieurs fois consécutives un même bloc d'instructions. La répétition s'effectue tant que la valeur de l'expression booléenne est «égale à Vrai.

tant_que (condition_de_poursuite) **faire**
{
Bloc d'instructions
}

2.3.1.3. Exemple

L'algorithme suivant affiche à l'écran les entiers de 1 à 5.

Algorithme affichage-des-5-premiers-entiers
Variables : compteur : entier ;
Debut

```
compteur←1 ;                          //initialisation
tant_que (compteur ≤ 5 ) faire        //condition de poursuite
{                                     //début du bloc
        ecrire( compteur) ;           //traitement
        compteur←compteur + 1 ;       //incrémentation du compteur
}                                     //fin du bloc
```

Fin

Représentons l'avancement des valeurs de la variable et de la condition booléenne.

Compteur	Condition : (compteur≤5)	Condition de continuité
1	Initialisation : avant de rentrer dans la boucle	
1	1 ≤ 5 : Vrai	Entrer dans la boucle
2	2 ≤ 5 : Vrai	Encore un tour
3	3 ≤ 5 : Vrai	Encore un tour
4	4 ≤ 5 : Vrai	Encore un tour
5	5 ≤ 5 : Vrai	Encore un tour
6	6 ≤ 5 : Faux	Sortir de la boucle

2.3.1.4. Plusieurs algorithmes équivalents

Les conditions suivantes permettent de sortir de la boucle précédente :

- Arrêt de la boucle quand compteur = 6, alors la condition tant_que(compteur ≠ 6) faire fonctionne.
- Arrêt de la boucle quand compteur ≥ 6, alors la condition tant_que(compteur < 6) faire fonctionne.
- Arrêt de la boucle quand compteur > 5, alors la condition tant_que(compteur ≤ 6) faire fonctionne.

```
compteur←1 ;
tant_que (compteur ≤ 5 ) faire
    {
    ecrire( compteur) ;
    compteur←compteur + 1 ;
    }
```

```
compteur←1 ;
tant_que (compteur ≠ 6 ) faire
    {
    ecrire( compteur) ;
    compteur←compteur + 1 ;
    }
```

```
compteur←1 ;
tant_que (compteur < 6 ) faire
    {
    ecrire( compteur) ;
    compteur←compteur + 1 ;
    }
```

```
compteur←0 ;
tant_que (compteur < 5 ) faire
    {
    ecrire( compteur) ;
    compteur←compteur + 1 ;
    }
```

2.3.1.5. La condition d'arrêt

Pour écrire une boucle, prenez l'habitude :

1. De chercher la condition d'arrêt ;
2. D'écrire sa négation à l'aide du tableau de correspondance des conditions d'arrêt qui suit (à connaitre).

Logique d'arrêt	=	≠	≥	<	>	≤	**ET**	**OU**
Logique de continuité	≠	=	<	≥	≤	>	**OU**	**ET**

Pour écrire une boucle, trois étapes sont obligatoires :

- L'initialisation des variables du compteur, et en général du bloc, avant d'entrer dans la boucle.
- La condition de poursuite. Il existe toujours différentes conditions de poursuite, qui sont toutes justes (équivalentes).
- La modification d'au moins une valeur dans la boucle(celle que l'on a initialisée précédemment) pour que la répétition exprime une évolution des calculs.

C'est une erreur grave de négliger l'un des points précédents → risque de ne pas sortir de la boucle (boucle infinie).

2.3.2. La syntaxe des autres boucles

2.3.2.1. La boucle pour-faire

La boucle pour-faire est utilisée très fréquemment en programmation pour réitérer une exécution un nombre de fois connu à l'avance.
Voyons comment écrire l'affichage des nombres de 1 à 5.

```
pour (compteur ← 1 jusqu'à  5) faire
{
        ecrire(compteur) ;
}
```

Voyons à travers un exemple comment passer d'une écriture pour-faire à une écriture tant_que-faire :

```
Algorithme boucle-tant-que-faire
Variables : compteur : entier ;
compteur←1 ;
tant_que (compteur ≤ 5 ) faire
    {
    ecrire( compteur) ;
    compteur←compteur + 1 ;
}
```

```
Algorithme boucle-pour-faire
Variables : compteur : entier ;
pour (compteur ← 1 jusqu'à  5) faire
{
    ecrire(compteur) ;
}
```

2.3.2.2. La boucle faire-tant_que

La boucle faire-tant_que effectue l'évaluation de la condition booléenne après avoir effectué le premier tour de boucle.

```
compteur←1 ;                          //initialisation
faire                                 //condition de poursuite
{
    ecrire( compteur) ;               //traitement
    compteur←compteur + 1 ;           //incrémentation du compteur
} tant_que(compteur ≤ 4 )             //attention à la condition
```

2.3.2.3. *Application en programmation*

Pour concrétiser l'utilisation de ces boucles, voyons comment elles sont implémentées dans quelques langages courants. Voici, en programmation C+, un exemple d'utilisation :

Boucle Tant-que-faire	Boucle pour-faire	Boucle faire-tant-que
Int i=1 ; **while (i<= 5){** **//l'opération itérée 5 fois** **I=i+1 ;** **}**	**Int i;** **for (i=1;i<= 5;i=i+1){** **//l'opération itérée 5 fois** **}**	**Int i=1 ;** **do{** **//l'opération itérée 5 fois** **I=i+1 ;** **} while (i<= 5);**

2.3.3. Applications

2.3.3.1. *Boucle et conditionnelle*

L'algorithme suivant faire lire à l'utilisateur cinq nombres entiers et affiche le plus grand à la fin.

Algorithme le-plus-grand-de-5-entiers
variables : compteur, valeur, max : entier ;
Debut

```
    Lire(valeur) ;
    max←valeur ;
    compteur ←1 ;
    tant_que (compteur <  5 ) faire
    {
            Lire(valeur) ;
            si (max< valeur) alors
            {       max←valeur ;
            }
            compteur←compteur + 1 ;
    }
    ecrire( «''max egale'', maxr) ;
```

Fin

Le déroulement de la boucle :

valeur	max	compteur	(compteur<valeur)	Condition de continuité
2	2	1	Les variables avant le test du tant_que	
2	2	1	1<5 : Vrai	Vrai, premier tour (1)
8	8	2	2<5 : Vrai	Vrai, encore un tour (2)
1	8	3	3<5 : Vrai	Vrai, encore un tour (3)
4	8	4	4<5 : Vrai	Vrai, encore un tour (4)
7	8	5	5<5 : Faux	Faux, sortie de la boucle

2.3.3.2. Boucle et tableau

L'algorithme suivant permet de saisir les éléments d'un tableau grâce à une boucle.

```
Algorithme boucle-et-tabeau
variables : tab : tableau[] d'entiers
            indice : entier ;
Debut
    tab←new entier[8] ;;
    indice ←0 ;
    tant_que (indice < 8 ) faire
    {
            lire(tab[indice]) ;
            indice←indice+ 1 ;
    }
Fin
```

2.4. LES BOUCLES IMBRIQUEES

2.4.1. L'usage

Il n'y a qu'un bloc d'instructions à répéter lors d'une boucle. Mais le bloc peut être lui-même composé d'une ou plusieurs boucles. On parle alors de *boucles imbriquées*.

Prenons exemple la saisie des notes, pour extraire la meilleure de toutes. Ajoutons comme contrainte supplémentaire qu'une note doit être comprise entre 0 et 20. Si ce n'est pas le cas l'algorithme doit prévenir l'utilisateur pour qu'il recommence la saisie.

```
Algorithme saisir-notes-entre-0-et-20
variables : note : entier ;
Debut
    ecrire( «''entrer une note :'') ;
    Lire(note) ;
    tant_que (note <  0 ) ET (note >20 ) faire
    {       ecrire( ''vous avez fait une erreur, essayer encore :''
            Lire(note) ;
    }
Fin
```

Intégrons ce bloc dans l'algorithme de saisie des 5 notes décrit précédemment.

```
Algorithme le-plus-grand-de-5-entiers
variables : compteur, note, max : entier ;
Debut
    ecrire( «''entrer une note :'') ;
    Lire(note) ;
    tant_que (note <  0 ) ET (note >20 ) faire
    {       ecrire( ''vous avez fait une erreur, essayer encore :'') ;
            Lire(note) ;
    }
      max ← note ;
      compteur←1 ;
tant_que (compteur <  5 ) faire
{
    ecrire( «''entrer une note :'') ;
    Lire(note) ;
    tant_que (note <  0 ) ET (note >20 ) faire
    {       ecrire( ''vous avez fait une erreur, essayer encore :'') ;
            Lire(note) ;
    }
      si (max< note) alors
      {     max←note;
      }
      compteur←compteur + 1 ;
}
ecrire( «la note la plus grande est », max) ;
Fin
```

2.4.2. Boucle et tableau à deux dimensions

Si nous désirons écrire un programme qui travaille avec un damier 10 cases sur 10 contenant des entiers, nous introduisons une instance damier sous forme une matrice (tableau à deux dimensions).

```
Algorithme mettre-a-zero-le-damier
variables :  damier : tableau[][]d'entiers
             indLigne, indColonne : entier ;
Debut
    damier←new entier[10][10] ;
    indLigne  ←0 ;
    tant_que (indLigne < 10 ) faire
    {       indColonne  ←0 ;
            tant_que (indColonne < 10 ) faire
            {       indColonne  ←0 ;
                    damier[indLigne](indColonne]←0 ;
                    indColonne←indColonne+ 1 ;
            }
            indLigne←indLigne+ 1 ;
    }
Fin
```

3. LES FONCTIONS

Nous avons déjà utilisé depuis le premier chapitre les fonctions lire et écrire pour saisir et afficher des valeurs. Une fonction fournit un service dans un algorithme isolé. Lorsque votre algorithme doit effectuer plusieurs fois la même tâche, il est judicieux d'isoler cette tâche dans une fonction et de l'appeler aux moments opportuns : votre algorithme n'en sera que plus facile à écrire et à modifier.

3.1. LES FONCTIONS SIMPLES

3.1.1. Définition

Fonction

Une fonction est un algorithme indépendant. L'appel (avec ou sans paramètre) de la fonction déclenche l'exécution de son bloc d'instructions. Une fonction se termine en retournant ou non une valeur.

La structure d'une fonction est la suivante :

fonction nomDeLaFonction***(liste des paramètres)*** : typeRetourne
Debut
Bloc d'instructions ;
Fin

Trois étapes sont toujours nécessaires à l'exécution d'une fonction :

1. Le programme appelant interrompt sont exécution.
2. La fonction appelée effectue son bloc d'instructions. Dés qu'une instruction retourne est exécutée, la fonction s'arrête.
3. Le programme appelant reprend alors son exécution.

Procédure

Une procédure est une fonction qui retourne vide : aucune valeur n'est retournée.

L'arrêt de la fonction

Une fonction s'arrête lorsque son exécution atteint la fin du bloc d'instructions, ou lorsque l'instruction retourne est exécutée (avec ou sans valeur)…

Le programmeur doit penser à concevoir et écrire des fonctions pour améliorer son programme ; Il y gagnera sur plusieurs points :

- Le code des algorithmes est plus simple, plus clair et plus court. Dans un algorithme, appeler une fonction se fait en une seule ligne et la fonction peut être appelée à plusieurs reprises.

- Une seule modification dans la fonction sera automatiquement répercutée sur tous les algorithmes qui utilisent cette fonction.
- L'utilisation de fonctions génériques dans des algorithmes différents permet de réaliser son travail et de gagner du temps.

3.1.2. Fonction sans valeur retournée

Apprenons à écrire et utiliser une fonction simple qui doit afficher «bonjour ». Cette fonction ne retourne pas de valeur : ceci est signalé en précisant qu'elle retourne vide.

fonction afficheBonjour***()*** : vide
Debut
ecrire(''bonjour'') ;
retourne ;
Fin

Une fonction se termine toujours par l'instruction retourne. Cette fonction effectuera les instructions situées entre Debut et Fin.

Ecrivons un algorithme qui appelle la fonction afficheBonjour().

Algorithme utilise-fonction
Debut
afficheBonjour() ;
Fin

Le schéma suivant montre la suite des instructions exécutées au cours du temps :

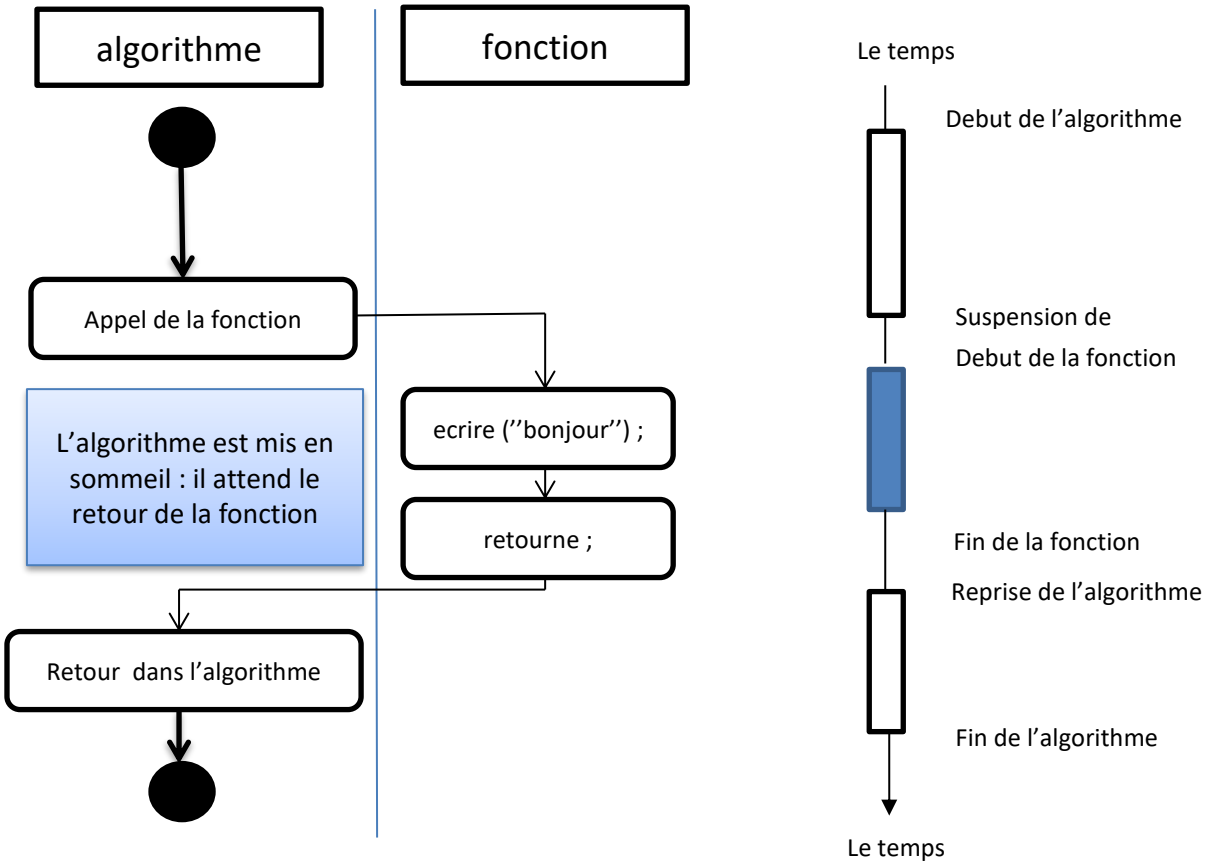


Figure 5 ***: Passage de l'algorithme à la fonction***

Imaginons un autre algorithme qui appelle 10 fois la fonction afficheBonjour().

```
Algorithme utilise-fonction-10
Variables : indice : entier ;
Debut
    indice ←0 ;
    tant_que (indice<10) faire
    {
            afficheBonjour() ;
            indice←indice +1 ;
    }
Fin
```

Pour faciliter la lecture des algorithmes, il convient de respecter des règles pour nommer les fonctions.

- Le nom d'une fonction commence par une minuscule.
- Le nom d'une fonction ne comporte pas d'espace.
- Si le nom de la fonction est composé de plusieurs mots, faire commencer chacun d'eux par une majuscule(exemple : sommeDeDeuxEntiers) et ne pas faire figurer de traits d'union.

3.1.3. Fonction avec une valeur retournée

La valeur de retour

Une fonction peut retourner une valeur au programme appelant. Cette valeur est unique. Le retour de la valeur signifie l'arrêt de la fonction.

Introduisons une autre fonction qui permet de lire une note en 0 et 20.

```
fonction lireNote() : entier
variables : note : entier ;
Debut
    ecrire( «''entrer une note :'') ;
    lire(note) ;
    tant_que (note <  0 ) ET (note >20 ) faire
    {       ecrire( ''vous avez fait une erreur, essayer encore :''
            lire(note) ;
    }
    retourne(note) ;
Fin
```

3.2. L'ENVIRONNEMENT DES DONNEES

3.2.1. Les paramètres

Le programme appelant doit donner à certaines fonctions des valeurs pour effectuer ses calculs. La fonction associe à ses valeurs des variables afin de les manipuler : ce sont les paramètres de la fonction.

Les paramètres

Un paramètre est une variable locale à une fonction. Il possède dès le début de la fonction la valeur passée par le programme appelant. .

3.2.2. Le passage des paramètres

Prenons un exemple d'une fonction maxDe2Valeurs qui retourene le maximum de deux valeurs passées en paramètre.

```
fonction maxDe2Valeurs(p1 : entier, p2 : entier) : entier
variables : resultat: entier ;
Debut
    si (p1 < p2) alors
    {
        resultat ← p2 ;
    } sinon {
        resultat ← p1 ;
    }
    retourne(resultat) ;
Fin
```

Soit un algorithme appelant :

```
Algorithme utilise-fonction-max
Variables : valeur1, max : entier ;
Debut
    Lire(valeur1) ;
    max← maxDe2 Valeurs(valeur1, 25) ;
    ecrire(max) ;
Fin
```

Déroulement après l'appel :

variables : resultat: entier ;			
Debut	p1=12	p2=25	resultat= ?
si (p1 < p2) alors	p1=12	p2=25	resultat= ?
resultat ← p2 ;	p1=12	p2=25	resultat= 25
sinon	Instruction non exécutée		
resultat ← p1 ;	Instruction non exécutée		
retourne(resultat) ;	Arrêt, la valeur 25 est retournée		
Fin	Variables et paramètres disparaissent à la fin.		

On peut écrire la fonction retourne au milieu de la fonction et on obtient une fonction plus lisible et plus simplifiée :

```
fonction maxDe2Valeurs(p1 : entier, p2 : entier) : entier
Debut
    si (p1 < p2) alors
    {
        retourne(p2) ;
    } sinon {
        retourne(p1) ;
    }
Fin
```

3.2.3. Les données d'une fonction

L'environnement de données

Un environnement de données, appelé aussi espace d'adressage, correspond à l'ensemble des variables associées exclusivement à un algorithme ou à une fonction.

Une variable définie dans un algorithme (respectivement dans une fonction), existe uniquement le temps limité de l'exécution de l'algorithme (respectivement de la fonction). Peu importe le nom des variables définies dans la fonction pour pouvoir l'utiliser.

Pour revenir à l'exemple précédent, on peut écrire la définition de la même fonction de différentes manières :

- **fonction** maxDe2Valeurs(p1 : entier, p2 : entier) : entier

ou

- **fonction** maxDe2Valeurs(valeur1 : entier, valeur2 : entier) : entier

ou

- **fonction** maxDe2Valeurs(entier, entier) : entier

Dans les trois cas, l'utilisation de la fonction est identique :

- leMax ← maxDe2Valeurs(455, 48) ;

Il est tout à fait possible que 2 variables, l'une déclarée dans le programme appelant et l'autre déclarée dans la fonction, portent le même nom. Elles peuvent être du même type ou non, peu importe, puisqu'elles sont utilisées de manière différente dans des environnements de données différents.

A travers le schéma mémoire (figure 2), nous visualisons que l'algorithme et la fonction sont dans des « boites indépendantes » représentant les espaces d'adressage indépendants.

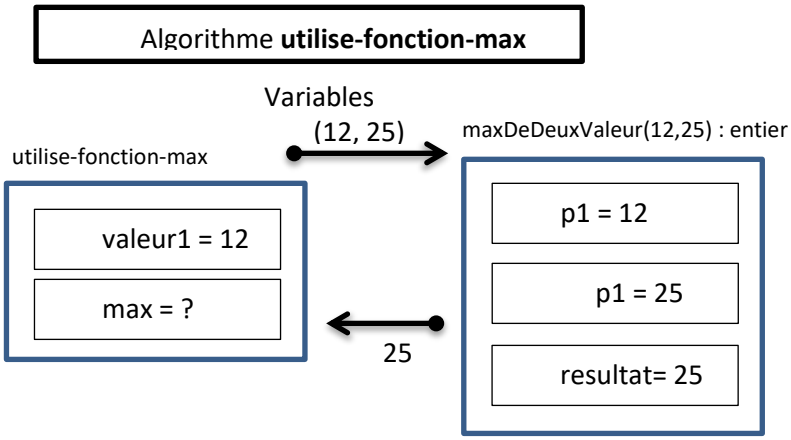


Figure 6*: Deux environnements de données distincts*

3.2.4. Les paramètres et les variables

La plupart du temps, l'exécution d'une fonction est paramétrable grâce à des valeurs qui lui sont passées. Les paramètres sont des variables de la fonction : il est donc faux de vouloir les redéfinir dans la zone de déclaration des variables.

Une fonction peut accéder à deux types de données :

- Les **paramètres**, dont les valeurs sont connues dès le début de la fonction. Les valeurs dont passées en paramètres. Il est inutile de nommer les paramètres avec le même nom que les variables utilisées lors de l'appel de la fonction.

- Les **variables,** appelée locales) définies dans le bloc de déclaration des variables.
  Au cours de l'exécution d'une fonction, les variables définies dans le programme appelant sont inconnues : aussi bien leur nom que leur valeur.

La valeur retournée est unique. Il est impossible pour une fonction de retourner plusieurs valeurs, mais également de modifier directement une variable du programme appelant.

3.2.5. Techniques

3.2.5.1. Définir une fonction

La signature d'une fonction

La signature d'une fonction décrit les éléments permettant de l'appeler correctement :

- *le nom de la fonction*
- *le type (et l'ordre) des paramètres ;*
- *le type de la valeur retournée*

Un programmeur qui souhaite utiliser une fonction n'a pas besoin de connaitre le corps de la fonction ; ni même le nom ou les types des variables internes à la fonction, mais seulement les caractéristiques nécessaires à son utilisation : sa signature. Il s'agit en fait de la carte d'identité de la fonction.

Quelques signatures de fonctions :

Signature de la fonction	Résultat de la fonction
afficheBonjour() : vide	Affiche « bonjour»
maxDe2Valeurs(entier,entier) : entier	Retourne une valeur entière, le maximum des deux paramètres
hasard(entier) : entier	Retourne une valeur entière aléatoire comprise entre 0 et la valeur passée en paramètre comprise
partieEntiere(reel) :entier	Retourne la valeur de l'entier égale à la partie entière du réel passé en paramètre
racineCarree(entier) :reel	Retourne le réel égale à la racine carré de la valeur positive passée en paramètre
valeurAbsolue(entier) : entier	Retourne la valeur absolue de la valeur entière passée en paramètre
valeurAbsolue(reel) : reel	Retourne la valeur absolue e la valeur réelle passée en paramètre

Le polymorphisme paramétrique

Deux fonctions peuvent avoir le même nom et des paramètres différents en nombre ou en type. Le polymorphisme paramétrique garantit automatiquement l'exécution de la bonne fonction associée au bon nombre de paramètres et à leurs types. En effet, les programmes identifient une fonction par sa signature (et pas uniquement par son nom).

3.2.5.2. Les erreurs fréquentes à éviter

Erreurs à éviter dans l'utilisation d'une fonction :

- Oublier les parenthèses
- Ne pas respecter le type de retour
- Ne pas respecter le type des paramètres
- Ne pas réécrire à chaque fois une fonction qui existe déjà ;
- Croire que la fonction peut modifier la variable du programme appelant.

Algorithme utilise-fonction-5-erreurs
Variables : valeur1, max : entier ;
Debut

```
afficheBonjour ;        //mettre le parenthèses afficheBonjour() ;
hasard(5) ;             //et la valeur retournée valeur1 ← hasard(5) ;
hasard(2.5) ;           //le type de paramètre
```

Fin

Erreurs à éviter dans l'écriture d'une fonction :

- Donner le même nom à un paramètre et à une variable.
- Placer plusieurs retourne consécutifs dans la fonction ;
- Vouloir retourner plusieurs valeurs ;
- Oublier de retourner la valeur ou retourner une valeur du mauvais type.
- Vouloir continuer un traitement après l'instruction retourne.
- Penser qu'en modifiant la valeur d'un paramètre, celui-ci sera modifié dans le programme appelant.

fonction fonctionErreur(p1 : entier) : entier
variables : resultat : réel ;

```
        p1 : entier ;                   // erreur : p1 est déjà un paramètre !
Debut
        retourne(resultat + p1) ;       //erreur : mauvais type de retour
        p1 ←2 ;                         //la variable passée à la fonction n'aura pas  //été
                                        modifiée
        retourne(2, resultat) ;         //erreur : on ne peut pas retourner plusieurs //valeurs
        resultat←p1 ;                   //erreur : cette opération ne sera jamais //exécutée
```

Fin

3.3. LES PARAMETRES INSTANCE

En utilisant les chaines dans des fonctions, nous allons étudier une manière de modifier directement une variable du programme appelant dans la fonction.

Imaginons une fonction qui convertit une chaine de caractères en minuscules.

3.3.1. Fonction qui retourne une instance

La première approche consiste à définir la donnée (la chaine à convertir) et le résultat (la chaine en minuscule).

La signature de la fonction serait alors :

```
fonction convertirEnMinuscule (c : Chaine) : Chaine
variables : car :caractère ;
            indice : entier ;
            resultat : Chaine ;
Debut
    resultat ← new Chaine(c) ;
    indice ←0 ;
Tant_que (indice< resultat.longueur()) faire
{
    Car ← resultat.iemeCar(indice) ;
    Si ((car ≥ 'A') ET (car ≤ 'Z')) alors
    {       car← car + ('a' – 'A') ;
            Resultat.modifierIeme(indice,car) ;
    }
    indice←indice+1 ;
}
Retourne resultat ;
Fin
```

Et son utilisation :

```
Algorithme utilise_fonction_minuscule
Variables : ch1, ch2 : Chaine ;
Debut
        ch1←Chaine(« TlEmSani ») ;
        ch2← convertirEnMinuscule(ch1) ;
        ch2.ecrire() ;
Fin
```

Représentant le schéma mémoire :

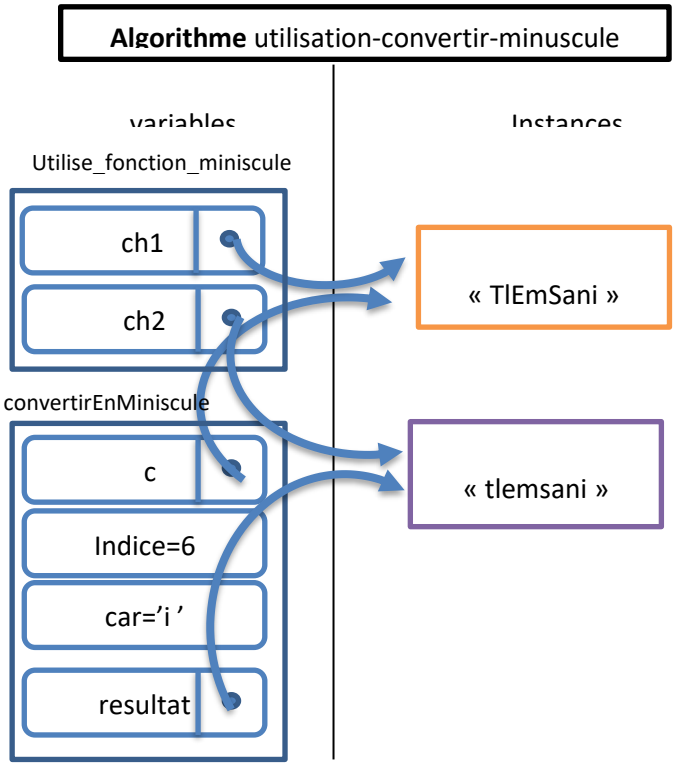


Figure 7*: état de la mémoire*

Ce schéma mémoire nous montre deux aspects importants des fonctions :

- Le programme appelant et la fonction sont dans des environnements de données différents, mais ils peuvent contenir des variables qui désignent la même instance (la même case).
- Le paramètre de la fonction est considéré comme une variable locale pour celle-ci.

3.3.1 Fonction qui modifie une instance paramètre

Une autre solution serait de convertir directement l'instance passée en paramètre. La signature de la fonction devient alors :

fonction convertirEnMinuscule (chaine) :vide
Cette fonction travaille directement sur l'instance indiquée par le programme appelant.

```
fonction convertirEnMinuscule (ch :Chaine) :vide
variables : car : caractere ;
            indice : entier ;
Debut
    Indice←0 ;
    tant_que (indice < ch. longueur()) faire
    {
            car← ch.iemeCar() ;
            si ((car ≥ 'A') ET (car ≤ 'Z')) alors
            {  car ← car + ('a' – 'A') ;
              ch.modifierIeme(indice, car) ;
            }
            indice ← indice + 1 ;
    }
    retourne ;
Fin
```

Et son utilisation :

```
Algorithme utilise-fonction-minuscule
variables : ch1 : Chaine;
Debut
    ch1 ← Chaine(« TleMsaNi ») ;
    convertirEnminuscule(ch1) ;
    ch1.ecrire() ;
Fin
```

Représentons le schéma mémoire de l'appel à la fonction en cours d'exécution (après trois tours de boucles) :

Cette technique est très utilisée : elle permet indirectement de partager des environnements de données.

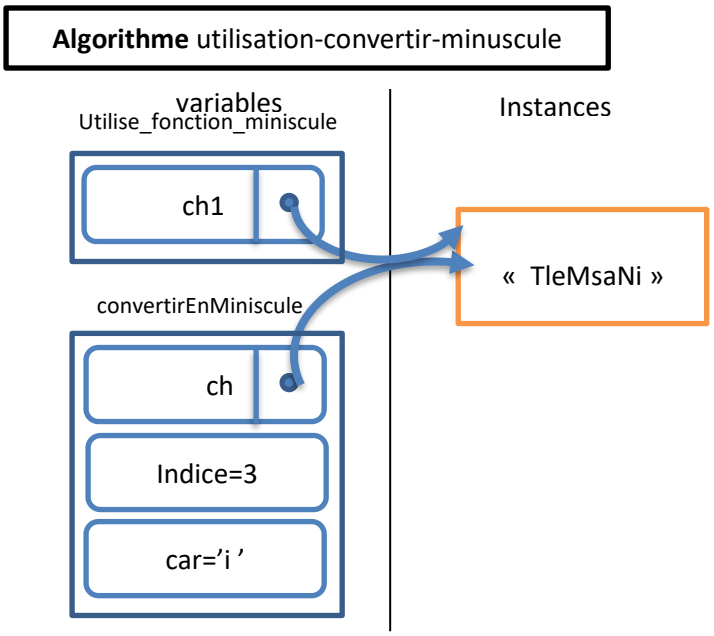


Figure 8*: état de la mémoire à l'itération indice=3*

3.4. LA RECURSIVITE

3.4.1. Définition

La notion de récursivité est assez naturelle mais pas toujours très simple à mettre en œuvre.

Fonction récursive

Une fonction est dite récursive si elle s'appelle elle-même.

Deux conditions sont nécessaires pour être en mesure d'utiliser la récursivité :

- Il faut pouvoir exprimer un algorithme sous forme d'une fonction de telle manière que sa valeur à un certain rang ne dépende que de sa valeur aux rangs inférieurs.
- Onj doit aussi connaitre la solution pour les rangs initiaux.

La technique de programmation est toujours la même : elle est assez déconcertante au début.

Technique pour écrire une fonction récursive

Il suffit d'utiliser la fonction que vous n'avez pas encore écrite en supposant qu'elle déjà un résultat.

Un algorithme récursif se compose de deux parties :

1. Au moins une condition d'arrêt des appels récursifs, où les valeurs à déterminer sont immédiatement connues.
2. Un appel récursif. La fonction s'appelle elle-même, dans un autre environnement.

Pour une fonction récursive qui retourne une valeur :

```
Fonction fonctionRecursive(liste des paramètres) : typeRetourne
Debut
        Si (condition d'arrêt) alors
         {
                Retourne(…) ;
        }
        Sinon
        {
                Retourne(fonctionRecursive(lise des nouveaux parametres)) ;
        }
Fin
```

3.4.2. La fonction factorielle

3.4.2.1. Définition

Dans les mathématiques, la fonction factorielle est définie par :
Factorielle(n)= n != 1 x 2 x … x (n-1) x n. Donc Factorielle(1)= 1, Factorielle(2)=2, Factorielle(3)= 1 x 2 x 3 = 6 et Factorielle(4)= 1 x2 x 3 x 4 = 24

On peut réécrire la fonction factorielle(n) d'une manière récurrente strictement équivalente à la précédente :

- Factorielle(1)=1 ;
- Factorielle(n)= n x Factorielle(n-1), pour n >0.

3.4.2.2. La fonction

```
Fonction factorielle(nb : entier) :entier
Variables : f : entier ;
Debut
    si (nb = 1) alors
    {
            f←1 ;
            retourne(f) ;
    }
    sinon {
        f← nb x factorielle(nb-1) ;
        retourne(f) ;
    }
Fin
```

Et l'algorithme d'utilisation :

```
Algorithme ManipulationDeFactorielle
Debut
Ecrire(factorielle(3)) ;
Fin
```

3.4.2.3. L'exécution :

Comment fonctionne cet algorithme ? Calculons factorielle(3)

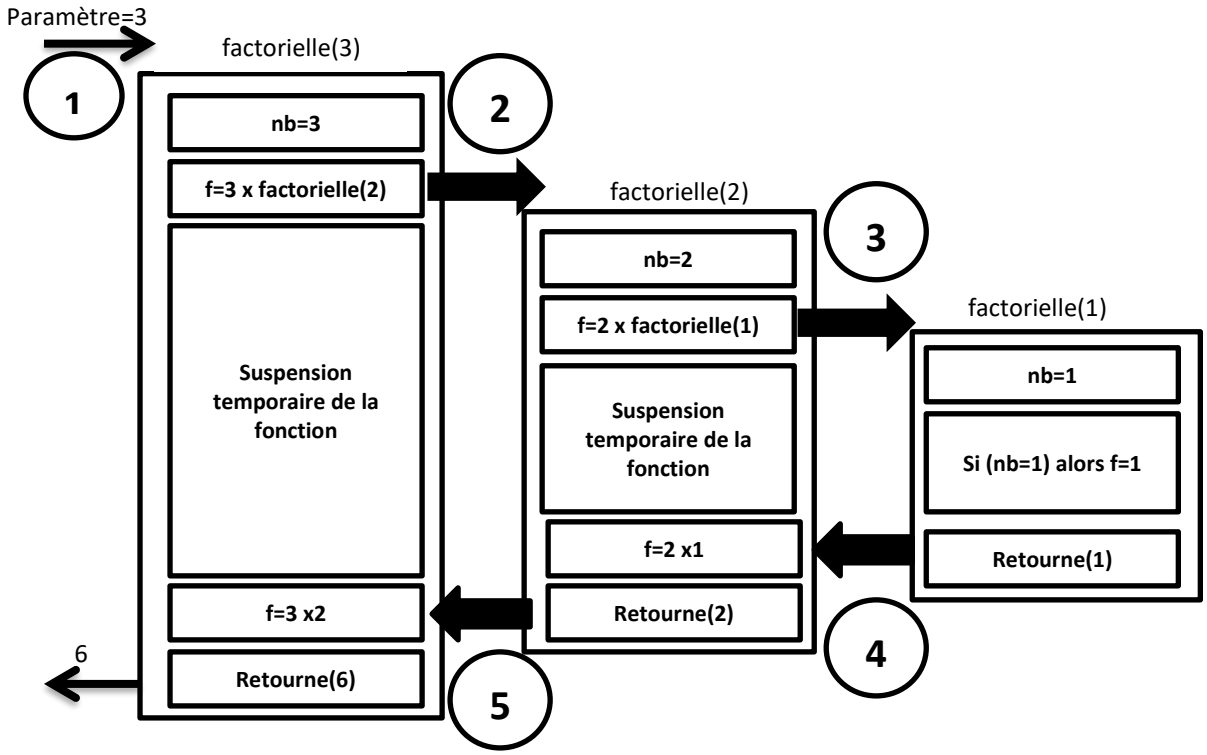


Figure 9*: calcul de factorielle 3*

Finalement, on a calculer :

factorielle(3) = 3 x factorielle(2)
= 3 x (2 x factorielle(1))
= 3 x (2 x (1))
= 6

La même méthode écrite de manière plus concise donne :

Fonction factorielle(nb : entier) :entier
Debut
si (nb = 1) alors
retourne 1 ;
retourne(nb x factorielle(nb-1) ;
Fin

3.4.3. Rechercher une valeur dans un tableau

Introduisons la fonction recherche : elle devra contenir comme paramètres le tableau étudié, sa taille et la valeur cherchée.

Function chercher(t :tableau[] d'entiers, taille :entier, valeur :entier) :entier

Pour chercher une valeur dans un tableau de manière récursive, il faut supposer que cette valeur a été trouvée (ou pas) au rang inférieur, donc dans tout le tableau sauf la 1ière case.

3	5	8	5	7	2	1	8	4

Je sais trouver la solution au rang inférieur

Il reste donc à tester le 1ier élément : s'il est égal à la valeur cherchée, on retourne sa place, sinon on retourne la solution trouvée au rang inférieur.

La condition d'arrêt a lieu simplement quand il ne reste pas de case à tester : la valeur n'a alors pas été trouvée.

Il faut donc introduire aussi le rang début de l'élément à tester.

```
Function chercher(t :tableau[] d'entiers, taille :entier, valeur :entier, debut :entier) :entier
Debut
    si (debut ≥ taille) alors                    //condition d'arrêt
        retourne (-1) ;
    si (t[debut]= valeur)                        //on a trouvé
    retourne debut ;
    retourne (chercher(t, taille, valeur, debut+1)) ;
Fin
```

3.4.4. La suite de Fibonacci

La suite de Fibonacci est une suite récurrente dont chaque terme dépend des deux précédents. Elle est définie par :
$U_0=0$ et $U_1=1$
$U_n=U_{n-1}+U_{n-2}$, pour tout entier n tel que $2 \leq n$
Par exemple, $U_2=U_1+U_0=1+0$, $U_3=U_2+U_1=1+1=2$, $U_4=U_3+U_2=3$.
On désire calculer tout terme de la suite de Fibonacci de rang n, pour tout entier n donné.
Deux méthodes s'offrent à nous :

- La méthode itérative, avec une boucle tant_que qui va permettre de calculer tous les termes du premier jusqu'au ni^{ème}.
- La méthode récursive. Elle sera comparable à celle utilisée pour le calcul récursif de factorielle et suit très exactement la définition de la suite de Fibonacci.

Pour écrire U(n), on suppose connues et justes les valeurs retournées par U(n-1) et U(n-2)
L'algorithme s'écrit dans le même esprit que celui du calcul de factorielle, en s'appuyant simplement sur la définition mathématique de la suite :

```
Fonction Fibonacci(n :entier) :entier
//Explication : calcul récursif à partir de la formule
Debut
        Si (n=0) alors          //première condition d'arrêt et de retour
                Retourne(0) ;   //cas où n=1
        Sinon
        {
                Si (n=0) alors          //seconde condition d'arrêt et de retour
                Retourne(0) ;           //cas où n=1
                Sinon
                Retourne(Fibonacci(n-1) + Fibonacci(n-2)) ;
        }
Fin
```

3.4.5. Les erreurs à ne pas commettre

L'utilisation de la récursivité semblera évidente pour certains, et demandera beaucoup plus de temps à d'autres. Il existe néanmoins certaines règles et certaines techniques qu'il faut garder à l'esprit :

- Ne pas mettre de boucle tant_que dans une fonction récursive (c'est faux dans 99% des cas).
- Ne pas oublier la condition d'arrêt.
- Ne pas hésiter à utiliser le résultat de la fonction que vous êtes en train d'écrire.
- Ne pas mettre une instruction retourne au milieu de votre fonction récursive : les instructions suivantes ne seraient pas exécutées.

3.4.6. La récursivité terminale

La récursivité terminale est une notion qui peut améliorer nettement les performances de vos

La récursivité terminale

Une fonction est récursive terminale si elle retourne sans autre calcul la valeur obtenue par son appel récursif.

algorithmes.
La dernière ligne d'une telle fonction sera :

retourne(fonction(paramètres)) ;

La fonction factorielle précedente utilise-t-elle la récursivité terminale ? La fonction factorielle se terminait par :

retourne(nb x factorielle(nb-1)) ;

Ce n'est pas une récursivité terminale, l'évaluation de l'appel récursif factorielle(nb-1) est suivie par la multiplication par nb avant le retourne. La version récursive terminale serait :

Fonction factorielle(nb : entier, resultat :entier) :entier
Debut
si (nb = 1) alors
retourne resultat ;
retourne(factorielle(nb-1, nb x resultat) ; **Fin**

Cette fonction est appelée en mettant initialement le résultat à 1 :
Fonction factorielle(nb : entier, resultat :entier) :entier
Debut retourne(factorielle(nb,1) ; **Fin**
Cette fonction factorielle terminale est souvent transformée par le compilateur en fonction itérative.

```
Fonction factorielle(nb : entier) :entier
Variables : resultat :entier ;
Debut
        resultat ← 1 ;
        tant_que (nb ≠1) alors {
                resultat ← nb x resultat ;
                nb ← nb – 1 ;
        }
        retourne(resultat) ; Fin
```

Concrètement en Java, voici les temps de calculs en millisecondes obtenus sur un calcul des factorielles en récursivité classique et terminale.

Récursivité	10 !	70 !	120 !	184 !
Classique	5	6	7	12
terminale	1	2	5	5

4. LES ALGORITHMES DE TRI & COMPLEXITE

4.1. LES ALGORITHMES DE TRI

Il existe plusieurs méthodes pour trier par ordre croissant les éléments d'un tableau.

4.1.1. Les Tri simples

4.1.1.1. Le tri par sélection

Le tri par sélection :

Le tri par sélection, aussi appelé aussi tri par le min, permet de trier un tableau. L'algorithme parcourt le tableau pour identifier le plus petit élément, positionne ce dernier au début du tableau, et recommence l'opération.

Un petit exemple permet de mieux comprendre l'évolution du tri. Soit le tableau **{7 ;16 ;5 ;10 ;2}.**
Chaque étape se fait en deux temps :

1. Déterminer le minimum de la partie non triée du tableau ;
2. Echanger le minimum avec la première case non triée.

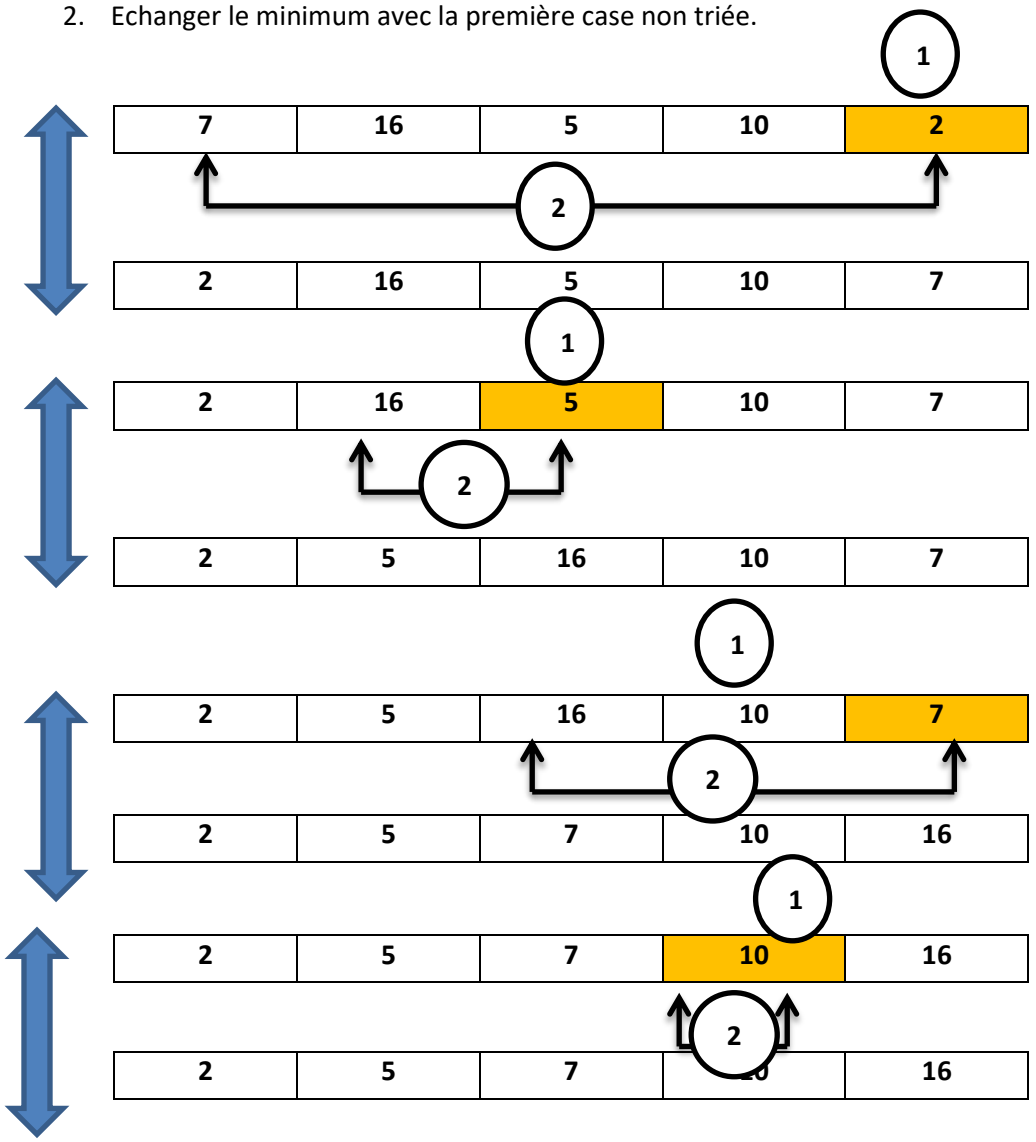


Figure 10 ***: différentes étapes du tri par sélection***

Ce tri parcourt tous les éléments de l'indice 0 au dernier. Pour chaque tour de la boucle principale, il y a deux opérations :

- Une boucle parcourt la partie non triée pour trouver le plus petit élément grâce à la variable **indiceNonTrie**.
- On procède à un échange entre le plus petit élément trouvé et le premier de la partie de tableau non triée.

```
Algorithme triSelection
Variables : indice, indiceNonTrie, posMinimum, minimum : entier ;
Debut
    indice ← 0 ;
    tant_que (indice < taille) faire
    {
            minimum ← tab[indice] ;
            posMinimum ← indice ;
            //boucle de recherche minimum
            indiceNonTrie ← indice ;
            tant_que (indiceNonTrie < taille) faire
            {
                Si (tab[indicenonTrie] < minimum) alors
                {
                   minimum ← tab[indiceNonTrie] ;
                   posMinimum ← indiceNontrie ;
                }
                indiceNontrie ← indiceNonTrie + 1 ;
            }
            echanger(posMinimum, indice) ;
            Indice ← indice + 1 ;
     }
Fin
```

4.1.1.2. Le tri par insertion

Le tri par insertion:

Le tri par insertion permet de trier un tableau. L'algorithme parcourt le tableau pour insérer chaque élément à la bonne place dans la partie triée du tableau.

Remarque :
Le premier élément constitue toujours le tableau trié de départ.

Ce tri parcourt tous les éléments de l'indice 1 au dernier. Il s'agit là de la boucle principale. On utilisera une variable *indice*.

Pour chaque élément de la boucle principale, l'insertion se fait en deux temps :

- Une boucle part de l'élément à insérer et décale tous les éléments plus grands d'une case vers la droite.

- Dès qu'on arrive à un élément plus petit ou au début du tableau, il suffit d'inscrire la valeur de l'élément à insérer. Cette valeur sera sauvegardée dans une variable *valeurAinserer*.

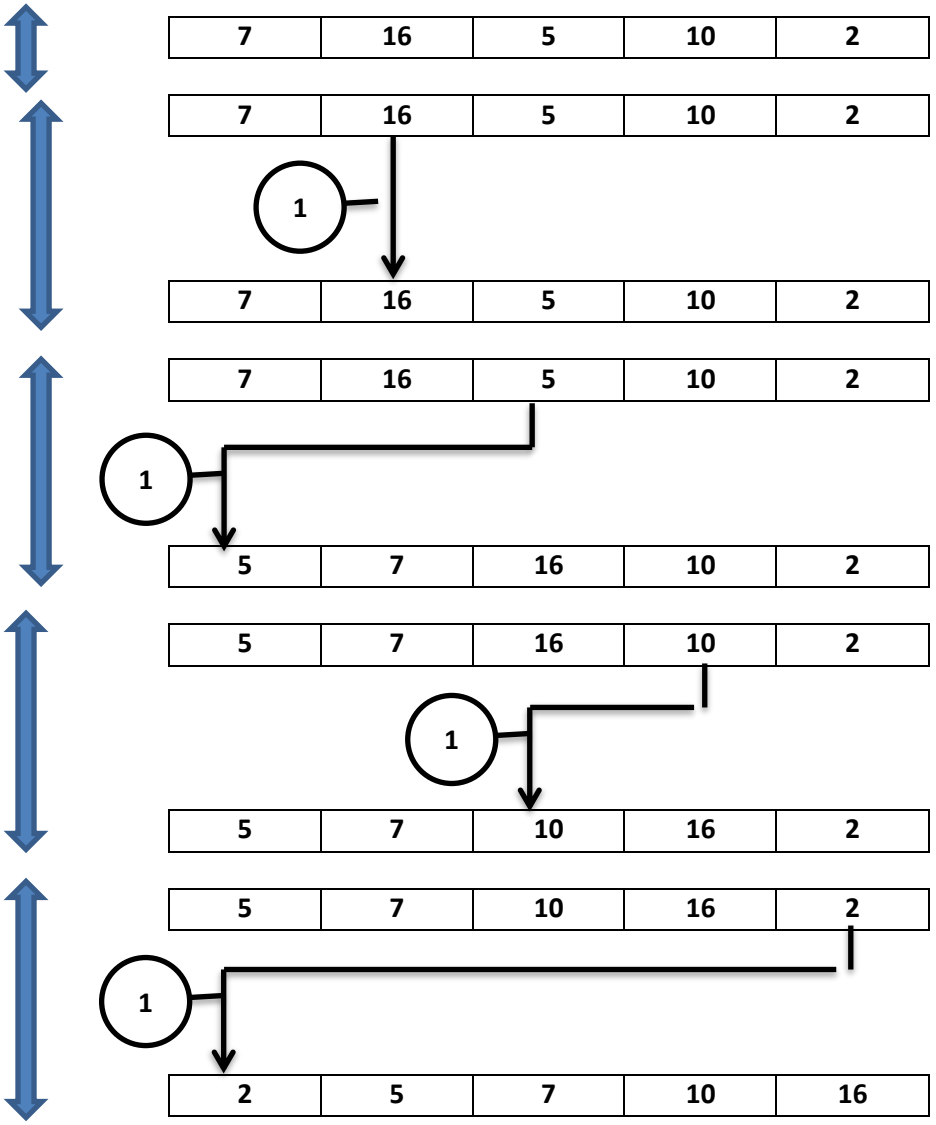


Figure 11 ***: différentes étapes du tri par insertion***

La partie la plus délicate est l'insertion : un exemple concret nous permettra de comprendre ce morceau d'algorithme avant d'écrire la méthode de tri par insertion entièrement. Trois étapes sont nécessaires :

1. Sauver la valeur à inserer dans une variables valeurAinserer
2. Décaler d'un rang vers la droite tous les éléments plus grands que l'élément à insérer ;
3. Placer la valeur à insérer (qui a été sauvée) à la place du dernier élément décalé.

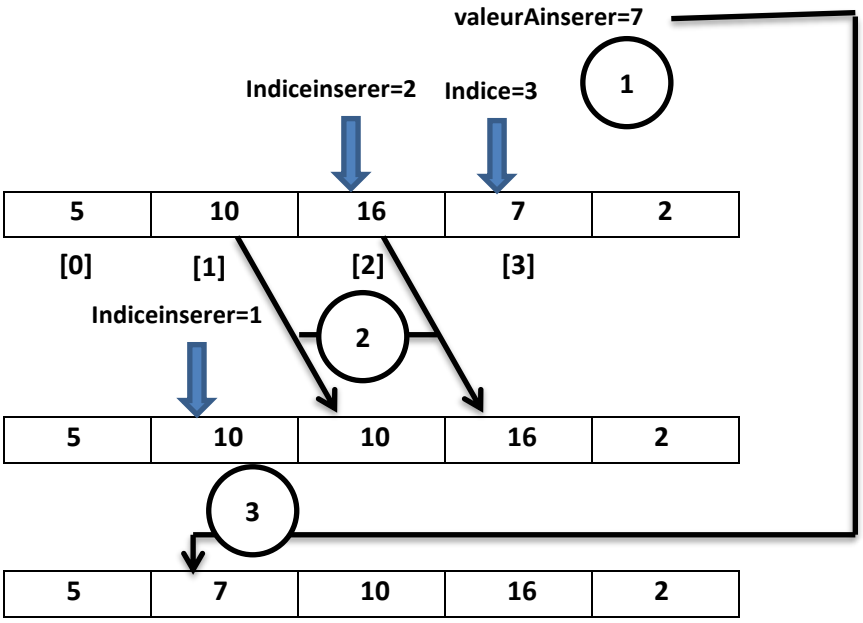


Figure 3 : insertion dans la partie déjà triée

```
Algorithme triInsertion
Variables : indice, indiceInserer, valeurAinserer : entier ;
Debut
    indice ← 1 ;
    tant_que (indice < taille) faire
    {
            valeurAinserer ← tab[indice] ;
             indiceInserer ← indice - 1 ;
            //boucle de recherche minimum
            tant_que (indiceInserer ≥ 0 ) ET ( valeurAinserer ≤ tab[indiceInserer]) faire
            {
                tab[indiceInserer+1]← tab[indiceInserer] ;
                indiceInserer ← indiceInserer  - 1 ;
            }
            tab[indiceInserer+1]← valeurAinserer ;
            Indice ← indice + 1 ;
     }
Fin
```

4.1.1.3. Le tri à bulle

Le tri à bulle:

Le tri à bulle, appelé aussi tri bulle ou bubble sort en anglais, permet de trier un tableau. L'algorithme parcourt le tableau pour comparer les éléments deux à deux afin de faire descendre les valeurs les plus lourdes en bas du tableau

Les éléments les plus légers (un peu comme les bulles d'air dans l'eau) vont remonter au début du tableau (à la surface).

Cette méthode présente l'avantage d'être très rapide si le tableau est presque trié (sauf quelques éléments), mais elle sera lente dans le cas contraire.

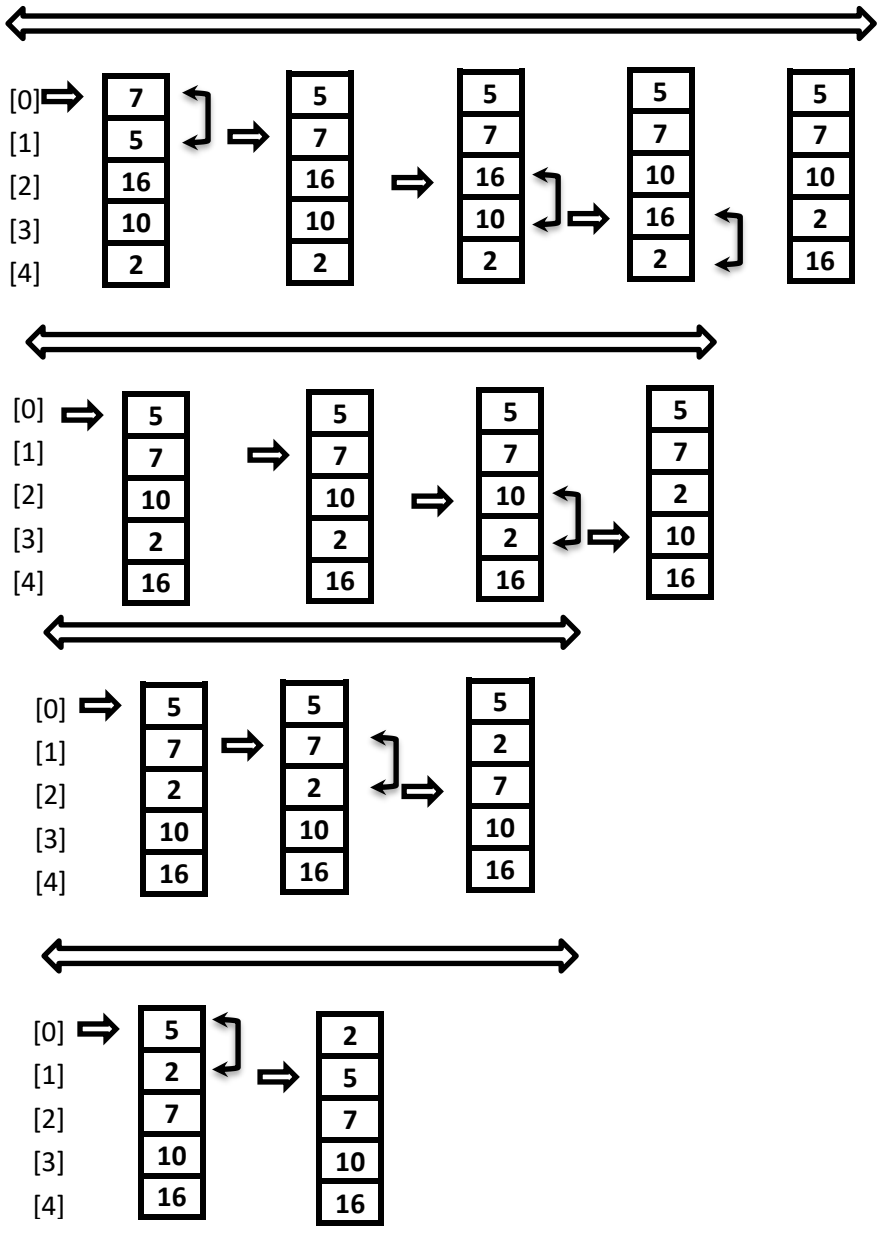


Figure 12*: différentes étapes du tri à bulle*

L'algorithme du tri à bulle est décrite comme suit :

```
Algorithme triBulle
Variables : indice, nbIteration : entier ;
            pasEncoreTrie : booléen ;
Debut
            pasEncoreTrie  ← vrai ;
            nbIteration ← taille – 1 ;
            tant_que (pasEncoreTrie=vrai) faire
            {
                 indice ← 0 ;
                pasEncoreTrie  ← faux;
                tant_que (indice <nbIteration) faire
                {
                        Si (tab[indice]≥tab[indice + 1]) alors
                        {
                        Echanger(indice, indice + 1) ;
                        pasEncoreTrie  ← vrai;
                        }
                Indice ← indice + 1 ;
                }
             }
Fin
```

4.1.2. La dichotomie

Le traitement par dichotomie:

La dichotomie consiste à subdiviser des données ou un problème en deux. Le traitement sur deux parties plus petites de moitié est en effet souvent plus simple.

La dichotomie, associée au concept « diviser pour régner », permet de répartir le traitement d'une grande quantité de données en deux traitements de deux quantités moins importantes.

4.1.2.1. La recherche dichotomique

La méthode de recherche retourne la position de l'élément ayant la valeur recherchée. Si l'élément n'est pas dans le vecteur, la méthode retourne -1. Si la valeur recherchée apparait plusieurs fois dans le vecteur la méthode retourne l'une des positions.

Pour utiliser la recherche dichotomique, le tableau doit être déjà trié.
La recherche dichotomique consiste à partir d'un tableau déjà trié :

1. Séparer le tableau en deux par un indice milieu (entre des indices gauche et droite) ;
2. Comparer la valeur recherchée et la valeur située au milieu du sous-tableau ;
3. Continuer la recherche dans un seul des deux sous-tableaux.

Il suffit de comparer la valeur recherchée et la valeur située au milieu du sous-tableau.

a. Recherche dichotomique itérative

```
fonction rechecheDicho(x :entier) :entier
Variables : gauche, milieu, droite : entier ;
            trouve : booléen ;
Debut
            gauche  ← 0;
            droite ← taille – 1 ;
            milieu← (gauche + droite) /2;          //division entière pour trouver le milieu
            trouve ← faux ;
            tant_que ((gauche ≤ droite) ET (NON trouve)) faire
            {
                Milieu← (gauche + droite) /2;     //recalculer le milieu(DIV)
                trouve  ← (tableau[milieu] = x); //on a trouvé l'élément ?
                si (x >tabelau[milieu]) alors
                        gauche ← milieu + 1 ;
                sinon                              //sinon, il est à gauche du milieu
                        droite ← milieu – 1 ;
              }
            Si (trouve = vrai) alors
                Retourne milieu                    //on sort
            Sinon
                Retourne (-1) ;                    //élément introuvable : on sort
Fin
```

b. Recherche dichotomique récursive

La méthode récursive cherche la valeur x dans la partie de tableu située entre les indices gauche et droite.

```
fonction rechecheDichoRecursif(x :entier, gauche : entier, droite :entier) :entier
Variables : milieu: entier ;
Debut
            milieu← (gauche + droite) /2;          //division entière pour trouver le milieu
            si (tableau[milieu] = x) alors         //deux conditions d'arrêt
                retourne (milieu) ;
            si (droite  ≤ gauche) alors
                retourne (-1) ;
            si (x  ≤ tableu(milieu]) alors         //appels récursifs
                retourne(rechercheDichoRecursif(x, gauche, milieu - 1)) ;
            sinon                                  //sinon il est à droite du milieu
                retourne(rechercheDichoRecursif(x, milieu + 1, droite)) ;
Fin
```

4.1.2.2. Le tri par fusion

Le tri par fusion:

Ce trie permet de trier un tableau avec un traitement récursif et dichotomique. Par récursivité, chaque tableau est diviser en deux sous-tableaux qui sont triés puis refusionnés dans le bon ordre grâce à un tableau intermédiaire.

Le tri demande à séparer le tableau en deux : il suffit d'introduire les indices *debut, milieu et fin* tels que *milieu = (debut + fin) / 2*. Grâce à la récursivité, ces deux sous-tableaux vont être triés. C'est la puissance de la récursivité, on doit supposer qu'ils ont été triés par la fonction :

```
fonction triFusion(debut :entier, fin : entier) : vide
Variables : milieu: entier ;
Debut
        Si (debut ≠ fin) alors
        {
        milieu← (gauche + droite) /2;
        triFusion(debut, milieu) ;
        triFusion(milieu + 1, fin)
        refusionner(debut, milieu, fin) ;
        }
Fin
```

Cette fonction sera appelée de 0 à taille-1 pour trier tout le tableau. Il faut alors refusionner les deux sous-tableaux triés comme le montre la figure suivante :

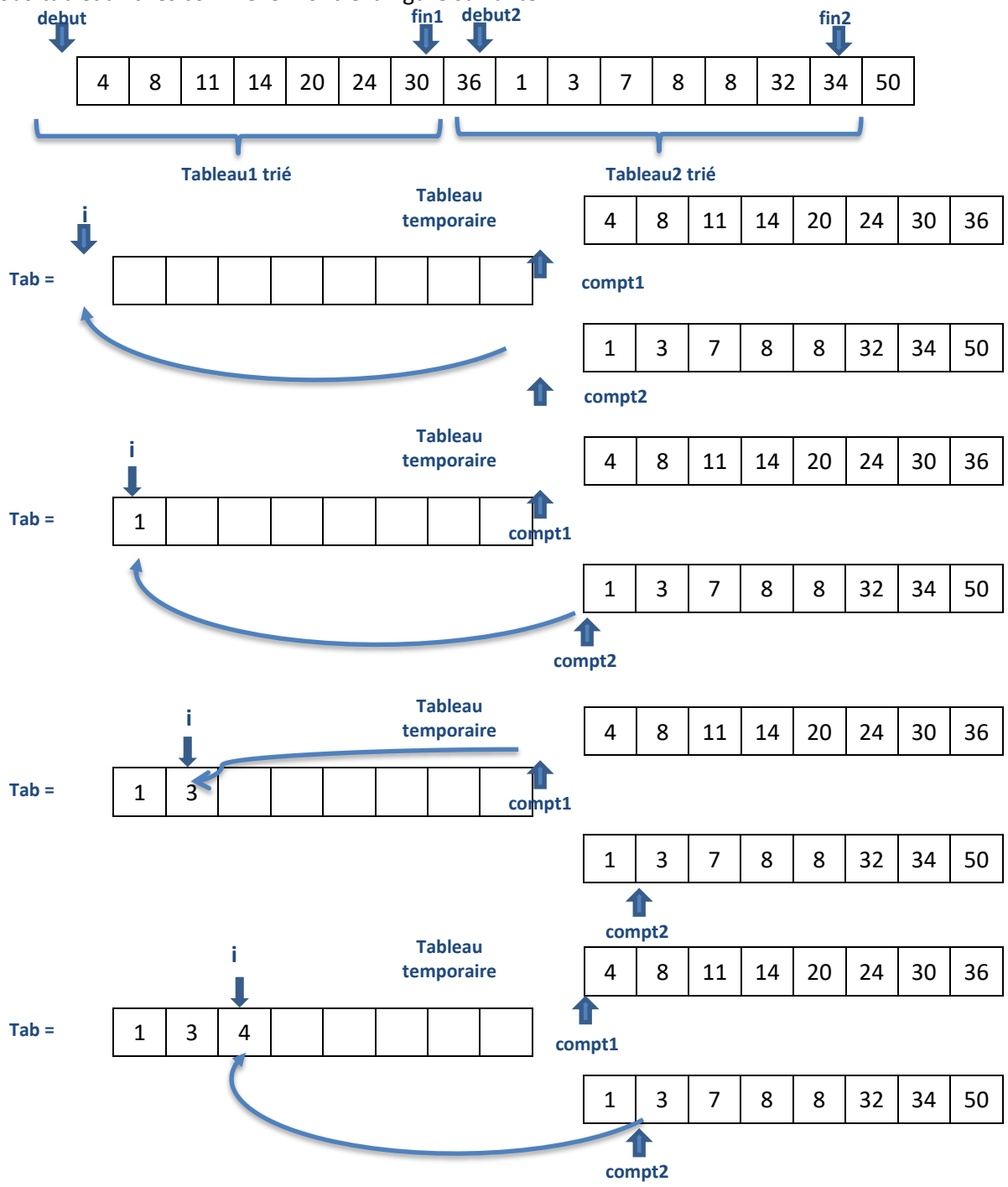


Figure 13*: La refusion de deux tableaux triés*

On commence à faire une copie du premier tableau dans le tableau temporaire. Les valeurs du 1ier tableau sont alors sans importance : les cases ont été vidées sur le schéma pour l'indiquer.
On introduit 3 indices : les indices *compt1* et *compt2* pour parcourir les éléments du 1ier et du 2ième tableau, et l'indice **i** pour préciser l'élément du tableau *tab* qui va être modifié.
On compare l'élément de 1ier tableau (grâce au tableau temporaire) avec celui du 1ième tableau :
Le plus petit élément est copié dans tab[i] et on passe cet élément en incrémentant *compt1* ou *compt2*.

```
fonction refusionner(debut :entier, fin1 : entier, fin2 : entier) : vide
Variables : debut2: entier ;
            tabTemp : entier[] ;
            compt1, compt2, i : entier ;
Debut
            tabTemp ← new entier[fin1-debut1 + 1) ;
            debut2 ← fin1 + 1 ;
            // on recopie les éléments du début du tableau
            i←0
            tant_que (i ≤ fin1) faire
            [
                tabTemp[i-debut1]←tab[i ];
                I ← i+ 1;
            }
            Compt1 ← debut1 ;
            Compt2 ← debut2 ;
            I← debut1 ;
            Tant_que ((i≤ fin2) ET (compt1 ≠ debut2)) faire
            {
                Si (compt2 = (fin2 + 1)) alors  //tous les éléments du 2ième tableau ont été placés
                {
                        tab[i] ← tabTemp[compt1-debut1] ;  //placer le reste du 1ier  tableau
                        compt1 ← compt1 + 1 ;
                }
                Sinon si (tabTemp[compt1-debut1] < tab[compt2]) alors
                {
                        tab[i] ← tabTemp[compt1-debut1] ;
                        //ajouter un élément du 1ier tableau
                        Compt1 ← compt1 + 1 ;
                }
                Sinon
                {
                        tab[i] ← tab[compt2] ;
                        compt2 ← compt2 + 1 ;
                }
            i ← i + 1 ;
            )

Fin
```

4.1.2.3. Le tri rapide : tri dichotomique récursif

Le tri rapide:

*Ce trie permet de trier un tableau avec un traitement récursif et dichotomique. Par récursivité, un élément appelé **pivot** est choisi. Le pivot est alors placé à sa place définitive dans le tableau avec les éléments plus petits avant et les plus grands après. La récursivité traite les deux sous-tableaux avant et après le pivot.*

Comme d'habitude, il est utile de dresser un schéma :

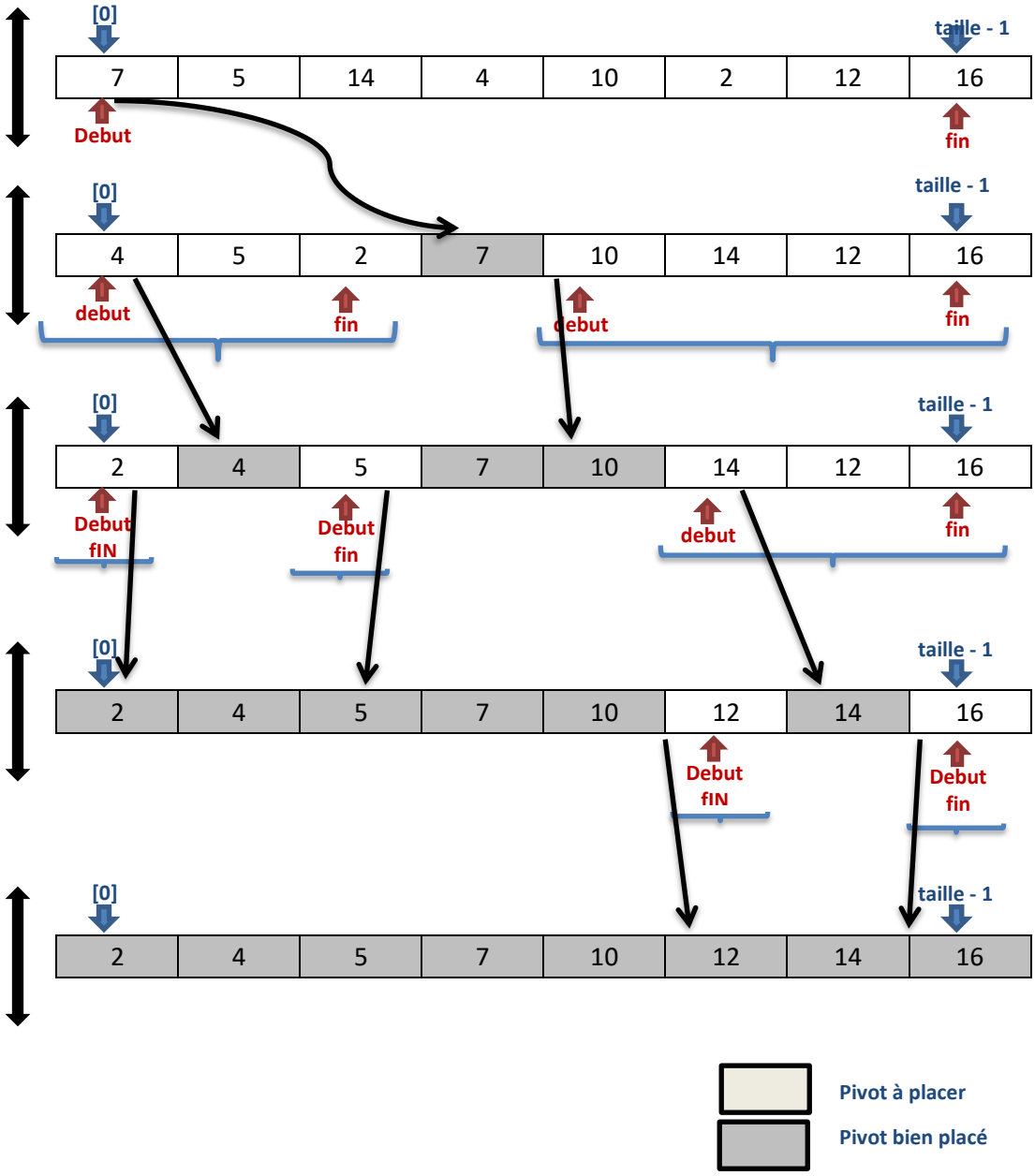


Figure 14*: Les différentes étapes du tri rapide*

```
fonction triRapide(debut :entier, fin: entier) : vide
Variables : pivot: entier ;
Debut
        Si (fin ≤ debut) alors
                Retourne ;                  //la condition d'arrêt
        Pivot ← placePivot(debut,fin) ;     //mettre le pivot à sa place
        triRapide(debut,pivot-1) ;          //appel récursif de la partie gauche de tab[]
        triRapide(pivot+1,fin) ;            //tri récursif de la partie droite de tab[]
Fin
```

Cette fonction sera appelée de 0 à taille-1 pour trier tout le tableau.

Le problème le plus épineux reste le placement du pivot à la bonne place. Utilisons pour cela deux variables *indiceGauche* et *indiceDroite*, qui vont laisser à gauche les éléments plus petits que le pivot, et à droite les éléments plus grands :

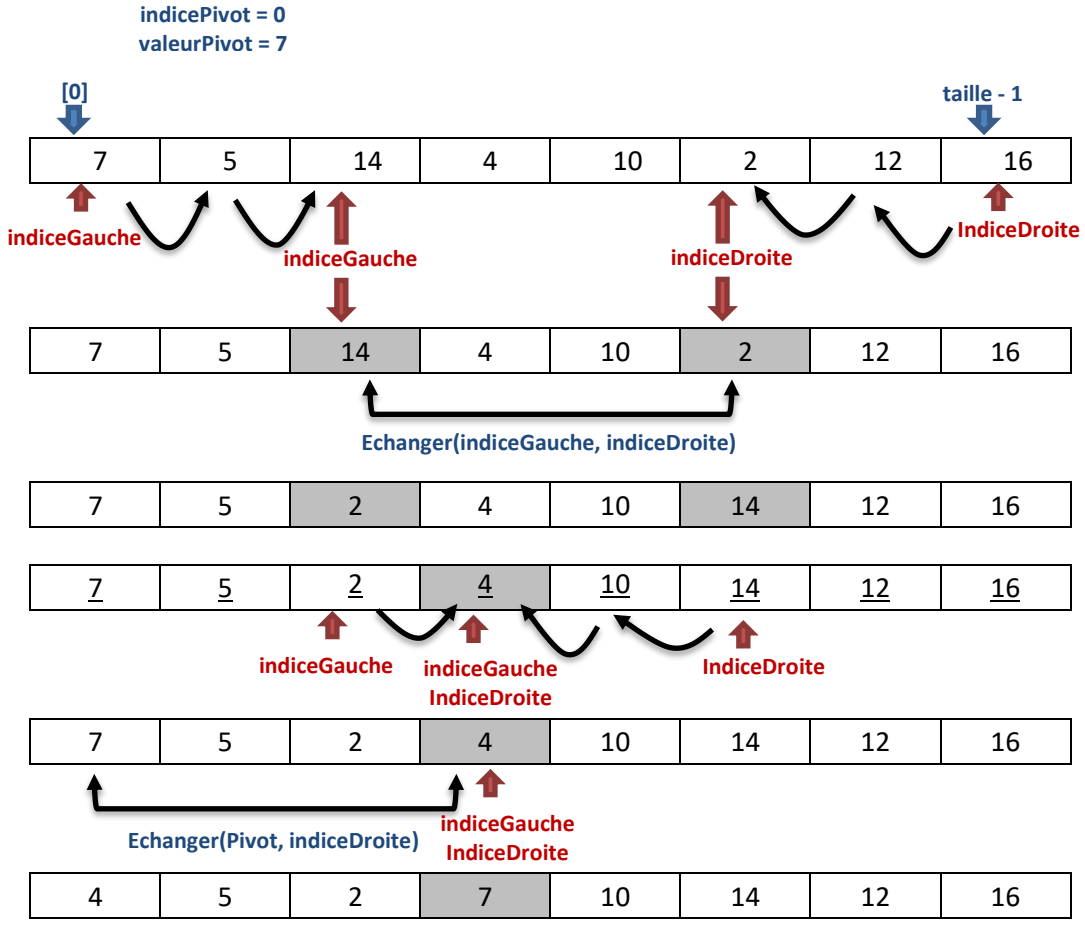


Figure 15*: Placer le pivot à sa place*

Voici la fonction correspondante à cette opération :

```
fonction placePivot(debut :entier, fin: entier) : entier
Variables : indicePivot, indiceGauche, indiceDroite, valeurPivot: entier ;
            pasPlace : booléen ;
Debut
    indciePivot ← debut ;
    valeurPivot ← tab[indicePivot] ;
    indiceGauche ← debut+1-1 ;  //on se place bien
    indiceDroite ← fin + 1 ;
    pasPlace ← vrai ;
    tant_que (pas Place) faire
    {
        indiceGauche ← indiceGauche +1 ;
        tant_que ((indiceGauche ≤ fin) ET (valeurPivot > tab[indiceGauche])) faire
        {
        indiceGauche ← indiceGauche +1 ;
        }
        indiceDroite ← indiceDroite -1 ;
        tant_que (valeurPivot < tab[indiceDroite]) faire
        {
        indiceDroite ← indiceDroite -1 ;
        }
        //en général indiceGauche et indiceDroite se croisent
        Si (indiceGauche ≤ indiceDroite) alors
        {
            Echanger(indiceGauche, indiceDroite) ;
        } sinon
        {
            pasPlace ← faux ;
        }
    }
    echanger(indicePivot, indiceDroite) ;
    Retourne indiceDroite ;
Fin
```

4.2. NOTION DE COMPLEXITE

4.2.1. Approche pratique

Un algorithme doit donner un résultat juste dans tous les cas, mais aussi s'effectuer de manière réaliste. La complexité représente l'évaluation du coût en mémoire utilisée et en temps de calcul d'un programme informatique. En effet, ces deux facteurs peuvent empêcher un algorithme, qui fonctionne sur le papier, de fournir un résultat, compte tenu des limites de vitesse et de capacité de stockage des ordinateurs.
De nos jours, la mémoire ne faisant pas défaut, le point sensible d'un programme reste son temps d'exécution : à quoi sert un programme qui fournira une réponse dan s200 ans ?

La complexité en temps:

La complexité d'un algorithme mesure le nombre d'opérations effectuées relativement au nombre N d'éléments traités.

Un algorithme peut avoir des résultats très différents en fonction des données initiales : le tri à bulle, par exemple, sera très rapide si les données sont déjà presque triées. Selon les cas favorables,

La complexité en moyenne:

La complexité en moyenne est la moyenne du nombre de traitements pour toutes les données possibles en entrée.

défavorables ou les cas intermédiaires, il y a encore plusieurs complexités à calculer.
Nous nous intéresserons uniquement à la complexité en moyenne, en choisissant quelques jeux de données au hasard.
Comparons les algorithmes de tris vus précédemment pour des tableaux identiques. Pour cela, calculons le nombre total de comparaisons effectuées pour chaque tri : il suffit d'ajouter la variable **nbTest** et de l'incrémenter au bon endroit.
Le tableau suivant présente le résultat obtenu par le programme informatique après quelques minutes (N représente le nombre d'éléments).

La complexité dans le pire (respectivement le meilleur) des cas:

Il s'agit de la complexité calculée lorsque les données demandent à l'algorithme le nombre maximum (respectivement le minimum) de traitements.

	Sélection	Insertion	Bulle	Fusion	Rapide
N=100	5.050	2.509	4.950	1.709	539
N=500	125.220	64.569	124.750	11.324	3.570
N=1000	500.500	247.462	499.500	25.165	7.827
N=2000	2.001.000	9947319	1.999.000	55.229	18.561
N=5000	12.502.500	6.277.463	12.497.500	156.382	48.558

Tableau 1 : Comparaison de la complexité des différents exemples vus précédemment

Nous constatons que le tri par fusion et le tri rapide sont effectivement les plus rapides (sur un tableau initialisé au hasard).
A l'aide d'une calculatrice, vous pouvez vérifier les résultats suivants(Log représente la fonction mathématique logarithme décimal, de base 10) :

	N x N	N x Log(N)
N=100	100 x 100 = 10.000	100 x log(1000) = 200
N=500	250.000	1.349,48
N=1000	1.000.000	3.000
N=2000	4.000.000	6.602,06
N=5000	25.000.000	18.494,85

Tableau 2 : Résultats

Nous constatons que les trois premiers tris sont à peu proportionnels à N x N comme montré dans le tableau suivant :

Sélection	Insertion	Bulle
-0,5 x N x N	-0,25 x N x N	-0,49 x N x N

Tableau 3 : Résultats

Par exemple, pour le tri opar insertion, pour N= 1 000, N x N = 1 000 000 et 0,25 x N x N = 250 000 : ce nombre est peu différent du nombre de comparaisons calculé par l'ordinateur : 247 462.
Le recours à une calculatrice est nécessaire pour déterminer le facteur de proportionnalité entre les tri rapide et fusion avec N x Lo(N) : 2,7 x 100 x log(100)=540, assez proche des 539 obtenus par l'expérience du tri rapide.

Fusion	Rapide
-8,5 x N x Log(N)	-2,7 x N x Log(N)

Tableau 4: Facteur de proportionnalité

La notion de l'efficacité d'un algorithme s'écrit sans tenir compte des valeurs de proportionnalité. Il suffit d'indiquer la croissance avec le nombre d'éléments N sous les formes suivantes :

- La complexité polynomiale de degré 2 est notée **$O(N^2)$** quand le traitement est proportionnel à N x N : c'est le cas pour les tris par sélection, par insertion et à bulle.
- La complexité linéaire notée **O(N)** intervient lorsque l'ensemble des éléments a été parcouru une seule fois. Il s'agit par exemple, de la complexité de l'algorithme de recherche linéaire d'une valeur dans un tableau.
- Un algorithme de complexité constante noté **O(1)** effectue toujours le même nombre d'opérations, quel que soit le nombre d'éléments N donnés.

4.2.2. Approche théorique

Calculons la complexité théorique du tri par sélection dans des cas simples ; Supposons pour cela que le tableau à trier possède N éléments. Déterminons le nombre de comparaisons effectuées en fonction de N.
Pour trouver l'élément le plus petit, il faut parcourir tout le tableau. Cette opération est effectuée N-1 fois, avec (au début) N éléments, puis N-2 fois avec N-1 éléments restants, ainsi de suite jusqu'à ce qu'il ne reste que 2 éléments (quand il n'en reste qu'un, il n'y a plus, de parcours à faire). Calculons le nombre de parcours du tableau :

Nombre de parcours = (N-1) + (N-2) + … + 3 + 2

~ N x (N-1)/2 : on retrouve le terme le plus grand **~0,5 x N x N,** trouvé par l'approche pratique.

5. LES POINTEURS

Chaque système de stockage présente des avantages et des inconvénients. Le problème des tableaux est leur taille fixe. Une autre structure, dynamique, permet de réserver uniquement en mémoire l'espace utilisé au fur et à mesure des besoins, mais au prix d'une plus grande complexité. Cette technique utilise une entité intermédiaire : la cellule.

5.1. LA CELLULE

5.1.1. Présentation

La cellule ne sera pas utilisée directement dans nos algorithmes : elle Sert juste à fabriquer

Cellule :

Une cellule est un composant qui contient un élément et qui référence la cellule suivante.

d'autres entités : les listes et les piles.

Figure 16 Une cellule qui contient la valeur 15.

5.1.2. Utilisation

Soit l'entité CelluleReel qui nous permettra de stocker un élément de type réel :

- CelluleReel(valeur :réel, suivant :CelluleReel)
- getValeur() :réel
- getSuivant() :celluleReel
- setValeur(valeur :réel) :vide
- setSuivant(suivant :Cellulereel) :vide

Ecrivons un petit algorithme permettant d'illustrer chaque méthode et le schéma mémoire associé.

Algorithme utilisation-CelluleReel
Variables : c1, c2 : CelluleReel ;
Debut
c1←new CelluleReel(15,null) ;
c2←new CelluleReel(3,c1) ;
//étape n°1

c1.setSuivant(c2) ;
c2.setValeur(99) ;
//étape n°2
Fin
Représentant le schéma mémoire à la fin de l'exécution de l'algorithme précédent.

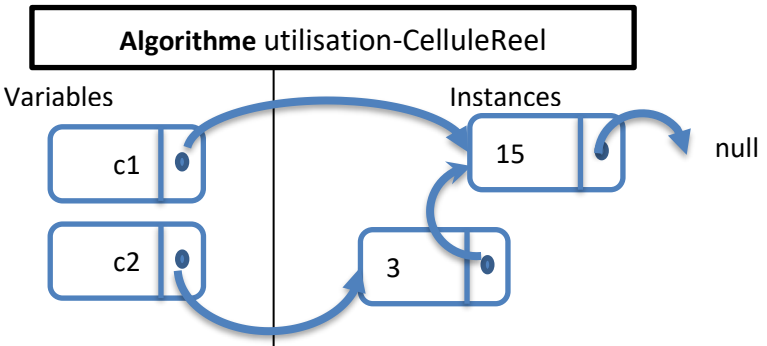


Figure 2. *Schéma mémoire à l'étape n°1*

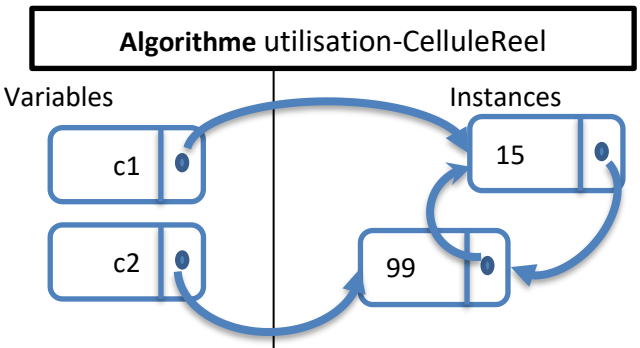


Figure 17 ***Schéma mémoire à l'étape n°21***

5.1.3. Les attributs

Les deux attributs définissant la cellule sont encapsulés dans CelluleReel. Ils définissent la valeur réelle contenue dans la cellule, et référence sur la cellule suivante :

- valeur : réel
- suivant : CelluleReel
- CelluleReel(valeur :réel, suivant : CelluleReel)
- getValeur() :réel
- getSuivant()CelluleReel ;
- setValeur(valeur :réel) :vide
- setSuivant(suivant :CelluleReel) : vide

L'attribut suivant référence une instance de CelluleReel, Si la cellule n'a pas de suivant, il vaut alors **null**.

```
Function CelluleReel()
Debut
    //Attributs
    valeur : réel
    suivant : CelluleReel
    //Constructeurs
    CelluleReel(valeur :réel, suivant : CelluleReel)
    Debut
        This.valeur←valeur ;
        This.suivant←suivant ;
    Fin
    //Méthodes
```

```
getValeur() :réel
Debut
    retourne(valeur) ;
Fin
getSuivant() :réel
Debut
    retourne(vsuivant) ;
Fin
setValeur(valeur :réel) :vide
Debut
    this.valeur←valeur ;
Fin
setSuivant(suivant :CelluleReel) :vide
Debut
    this.suivant←suivant ;
Fin
Fin
```

5.2. LA PILE

Employée parfois pour sa simplicité, une structure de données peut vous être utile. Il s'agit de la pile.

5.2.1. Présentation

Pile :

Une pile est une structure de stockage de données. Les éléments sont ajoutés les uns après les autres dans la pile. L'utilisateur peut accéder seulement au dernier élément stocké.

Soient les méthodes qui nous permettront de gérer des éléments de type entier :

- PileEntier()
- empiler(valeur :entier) :vide
- depiler() :entier
- estVide() :booléen

Détaillons l'utilisation de chaque méthode :

- **PileEntier() :** permet de créer une pile capable de contenir 7 éléments au maximum
- **PileEntier(n :entier) :** permet de créer une pile capable de contenir n éléments au maximum
- **empiler(n :entier) :** ajoute une nouvelle valeur n au sommet de la pile.
- **depiler() :entier :** retire le sommet de la pile et retourne sa valeur.
- **estVide() :booléen :** retourne Vrai si la pile n'a pas d'élément, Faux sinon.

Pour bien comprendre l'utilisation d'une pile d'entiers, écrivons un petit algorithme permettant d'illustrer chaque méthode et le schéma mémoire associé.

Algorithme utilisation-PileEntier
Variables : p : PileEntier ;
valeur : entier ;
Debut
P←new PileEntier() ;
//étape n°1
p.empiler(3) ;
//étape n°2
p.empiler(5) ;
p.empiler(7) ;
//étape n°3
valeur←p.depiler()
//étape n°4 **Fin**

Représentant l'évolution des valeurs de la pile : la pile est vide au début, puis les valeurs s'empilent (voir figure 4)

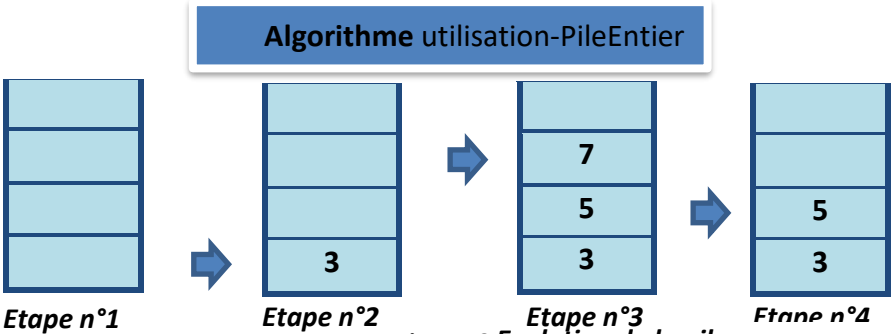


Figure 18 *Evolution de la pile*

5.2.2. Les attributs

La PileEntier peut être gérée par un tableau. Il faut définir un entier indiquant la position du sommet ou le nombre d'éléments déjà empilés. Choisissons la variable nbElement qui est égale à 0 quand la pile est vide, et qui augmente à chaque valeur empilée. (voir figure 5)

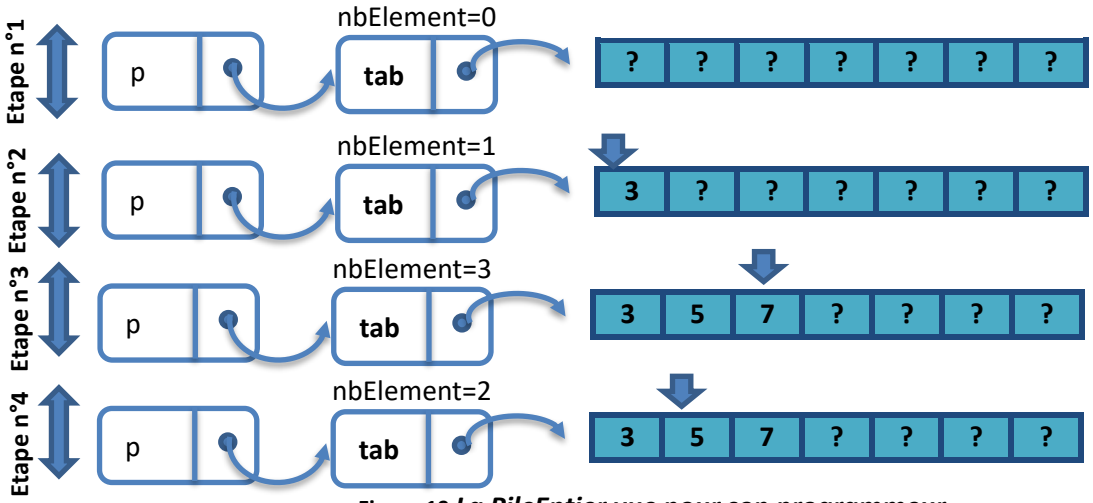


Figure 19 *La PileEntier vue pour son programmeur*

L'écriture des méthodes est plus simple que pour le cas de vecteur. Commençons par le constructeur qui initialise les deux attributs.

```
Function PileEntier()
Debut
    nbElement←0 ;              //aucune valeur n'a été empilée
    tab← new entier[7];        //L'attribut tab a été initialisé
Fin
Function empiler(nb :entier) :vide
Debut
    tab[nbElement]← nb;
    nbElement←nbElement + 1;
Fin
Function depiler() :entier
Debut
    nbElement←nbElement - 1;
    retourne(tab[nbElement]) ;
Fin
```

REMARQUE:

Il est inutile de modifier la valeur dépilée qui est dans le tableau : cette valeur est en effet inaccessible et sera écrasée au prochain empilement (disposer la valeur 7 ou 0, c'est la même chose…).

La méthode estVide() ne modifiant pas les attributs, elle s'écrit simplement :

```
Function estVide() :booléen
Debut
    retourne(nbElement=0) ;
Fin
```

5.2.3. Utilisation d'une pile de réel

La pile de réel que nous voulons créer est utilisée de la même manière qu'une pile d'entiers : d'ailleurs, les méthodes sont les mêmes, au type près.
On remplace PileEntier par PileReel. Pour pouvoir empiler, expliquons par la figure 6, la méthode empiler(99) est employée en 2 étapes :

1. On construit une nouvelle cellule qui contient la nouvelle valeur à empiler et dont l'élément suivant est l'instance pointée par l'attribut sommet.
2. Le sommet de la pile devient cette nouvelle cellule.

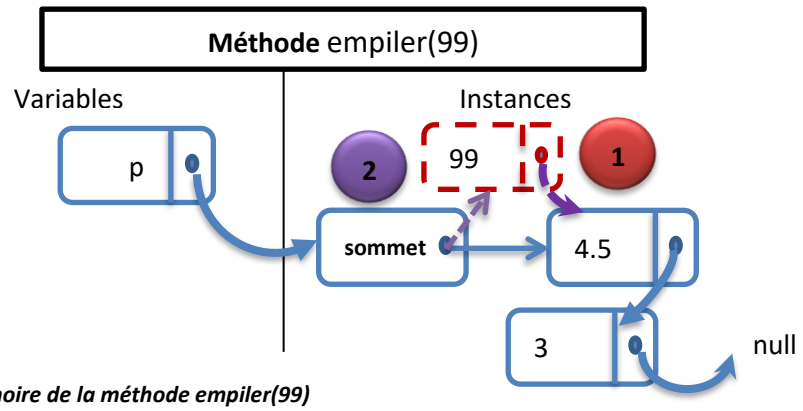


Figure 20 ***schéma mémoire de la méthode empiler(99)***

Pour la méthode dépiler(voir figure 7), il suffit de remarquer que la cellule pointée par l'attribut sommet possède un suivant : ce suivant est le prochain sommet(même s'il s'agit e la valeur null).

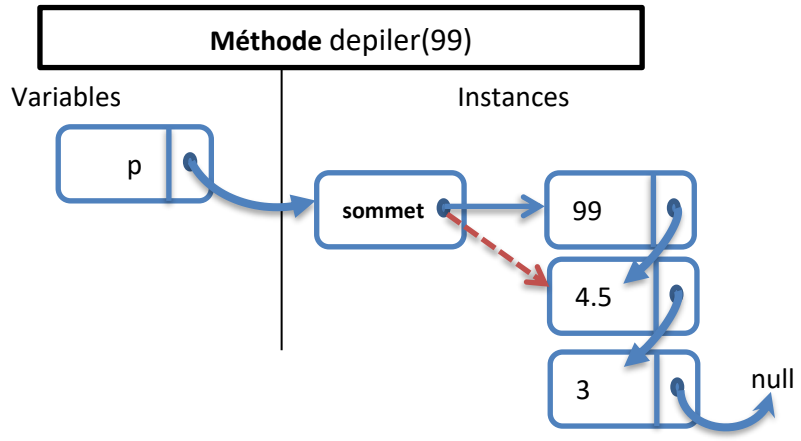


Figure 21 ***schémas mémoire de la méthode depiler(99)***

5.3. LA LISTE

5.3.1. Présentation

Liste :

Une liste appelée aussi liste chainée, est une structure dynamique de cellules. Les éléments sont atteints en parcourant chaque cellule depuis la première, appelée tête de la liste.

Voici le schéma d'une liste contenant les valeurs 99 , 4.5 et 3 :

Figure 22 ***Exemple d'une liste de 3 éléments.***

5.3.2. Utilisation

Soit l'entité ListeReel qui nous permettra de gérer un ensemble d'éléments de type réel :

- **ListeReel() :**permet de construire une liste vide
- **ajouterTete(valeur :réel) :vide** permet d'ajouter la valeur au début de la liste
- **ajouterQueue(valeur :réel) :vide** permet d'ajouter la valeur à la fin de la liste
- **supprimerTete() :vide** supprime la première cellule de la liste
- **supprimerValeur(valeur :réel) :vide** supprime la première valeur de la liste contenant la valeur
- **contient(valeur :réel) :booléen** retourne Vrai si la liste contient la valeur, sinon Faux
- **estvide() :booléen** retourne Vrai si la liste ne contient aucun élément, sinon Faux

5.3.3. Les attributs

Une liste de réels est définie par une suite de cellules de réels. Il est seulement nécessaire de connaitre la première cellule, appelée tête de la liste.
Avant d'écrire les méthodes, présentons deux listes : la première ne contenant aucun élément (la liste vide), et la deuxième où ont été ajoutées en queue les valeurs 99, 4.5 et 3.

Algorithme utilisation-ListeReel
Variables : liste : ListeReel ;
Debut
liste←new ListeReel() ;
//étape n°1

liste.ajouterQueue(99) ;
liste.ajouterQueue(4.5) ;
liste.ajouterQueue(3) ;
//étape n°2
Fin

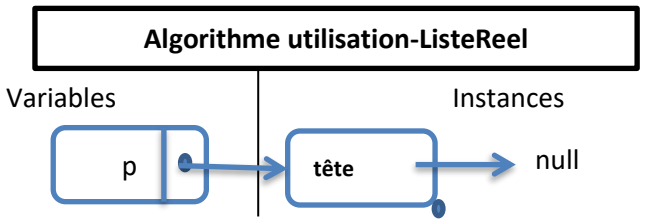


Figure 9 *liste vide (étape n°1)*

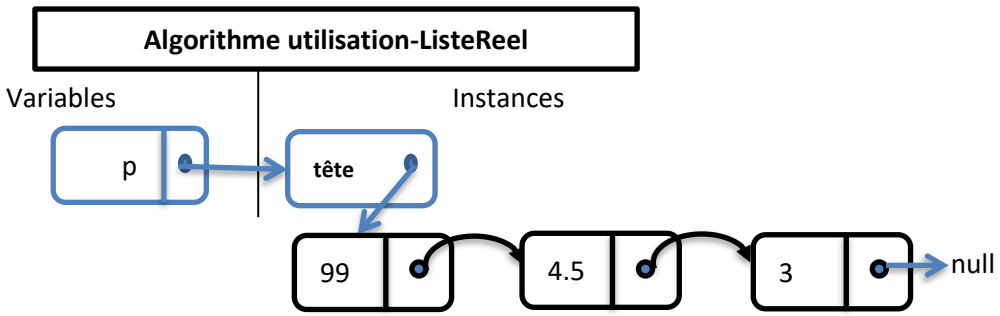


Figure 23 ***liste avec trois éléments (étape n°2)***

5.3.4. Les méthodes

Le constructeur de l'entité ListeReel doit initialiser les attributs ; Le constructeur initialise la tête : à la création, la liste est vide, aucune cellule ne compose la liste. La figure 9 nous en montre un exemple :

Function ListeReel()
Debut
tete←null ;
Fin

Ecrivons les autres méthodes, en commençant par celles qui permettent de travailler sur l'élément de tête. La méthode ajouterTete est identique à la méthode empile pour la pile.

Function ajouterTete(valeur:réel):vide
Variables: nouvelleCellule : CelluleReel
Debut
nouvelleCellule←*new* CelluleReel(valeur,tete);
tete←nouvelleCellule ;
Fin

La méthode supprimerTete est identique à la méthode depiler pour la pile.

Function supprimeTete():vide
Debut
tete←tete.getSuivant() ;
Fin
La méthode estVide est identique à celle de la pile.
Function estVide():booléen
Debut
Retourne(tete=null) ;
Fin
Pour travailler avec la cellule de queue(la dernière avant le null), c'est plus difficile : il faut parcourir la liste pour positionner la variable local(1) sur la dernière cellule, créer la nouvelle cellule(2) et faire en sorte que le suivant de queue désigne cette nouvelle cellule(3), voir figure 11.

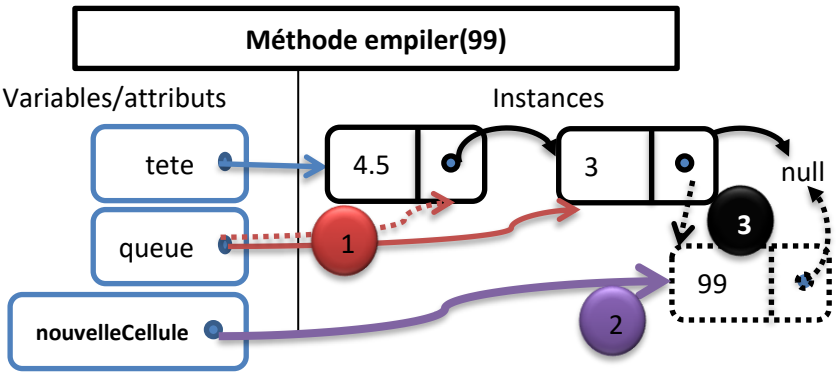


Figure 24 ***insertion en queue***

Function ajouterQueue(v :réel):vide
Variables : nouvelleCellule, queue : CelluleReel
Debut
//(1) on veut atteindre la dernière cellule
queue←tete ;
tant_que(queue≠null)faire
{
queue←queue.getsuivant() ;
}
//(2) création d'une nouvelle cellule
nouvelleCellule← new CelluleReel(v,null) ;
//(3) celle qui était dernière devient avant-dernière
queue.setsuivant(nouvellecellule) ;
Fin
Remarquons que cette méthode fonctionne aussi sur une liste vide.

La méthode *contient* permet de vérifier si une valeur appartient à la liste de réels : c'est assez simple, il suffit de parcourir la liste et d'arrêter dès que la valeur recherchée a été trouvée.
Function contient(valeur :réel):booléen
Variables : iterateur : CelluleReel
Debut
iterateur←tete ;
tant_que(iterateur≠null)faire
{
Si(iterateur.getElement()=valeur)alors
Retourne(Vrai) ;
iterateur←iterateur.getSuivant() ;
}

Retourne(Faux) ;

Fin

Examinons maintenant une méthode qui oblige à s'arrêter à un endroit de la liste : *supprimerValeur(valeur :réel).* Il faut parcourir les éléments à la recherche de la valeur, mais conserver la cellule précédente celle qui contient la valeur. On introduit donc deux variables : la première est un itérateur qui parcourt la liste depuis la deuxième cellule jusqu'à la dernière, la deuxième variable pointe toujours sur la cellule antérieure à l'itérateur(voir figure 12).

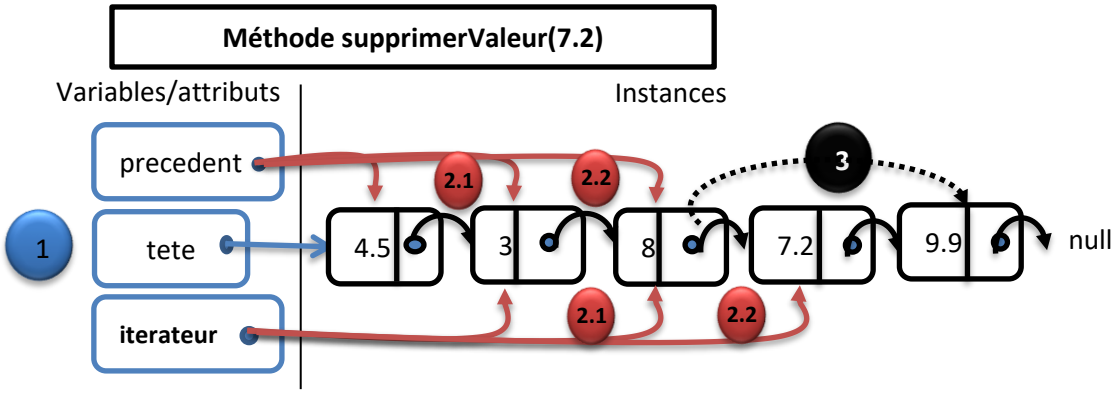


Figure 25 ***suppression d'une valeur au milieu***

Dans cet algorithme, les cas de la liste vide ou de la liste avec une seule cellule n'ont pas été pris en compte : il faut les vérifier dès le début.

Function supprimerValeur(valeur :réel):vide
Variables : iterateur, precedent : CelluleReel
Debut

```
si(estVide()) alors retourne ;
si(tete.getElement()=valeur) alors
        supprimerTete() ;
//(1)initialisation
        precedent←tete;
        iterateur←tete.getSuivant() ;
//(2) parcourt de la liste
        Tant_que(iterateur≠null)faire
        {
            si (iterateur.getElement()=valeur) alors
            {//(3) on supprime la cellule
                precedent← iterateur.getSuivant() ;
                iterateur←tete.getSuivant() ;
                retourne ;
            }
            precedent←iterateur ;
            iterateur←iterateur.getsuivant() ;
        }
        retourne ;
```

Fin

5.4. LA TABLE DE HACHAGE

La table de hachage présente deux avantages pour stocker des données :

- l'insertion est presque immédiate ;
- la recherche est très rapide.

5.4.1. Le principe

Dans le traitement de nombreuses données non triées, la difficulté est d'accéder le plus rapidement possible aux informations. Introduisons donc une technique d'algorithme qui mélange les tableaux et les listes chainées.
Rappelons que l'insertion dans un tableau trié est longue, et la recherche dans un tableau non trié aussi. Une solution très astucieuse consiste à stocker(de manière rapide) les données par blocs, puis, pour y accéder, à aller directement dans le bloc où se trouve la donnée. Le problème sera d'associer une donnée à un bloc de stockage. La table de hachage est la mise en pratique de cette idée.

La table de hachage :

Une table de hachage est une structure de données constituée d'un tableau de listes. Les données ne sont pas triées mais regroupées par blocs grâce à une fonction, appelée fonction de hachage.

Voici un schéma représentant une table de hachage constituée sur un tableau de N éléments (figure 13).

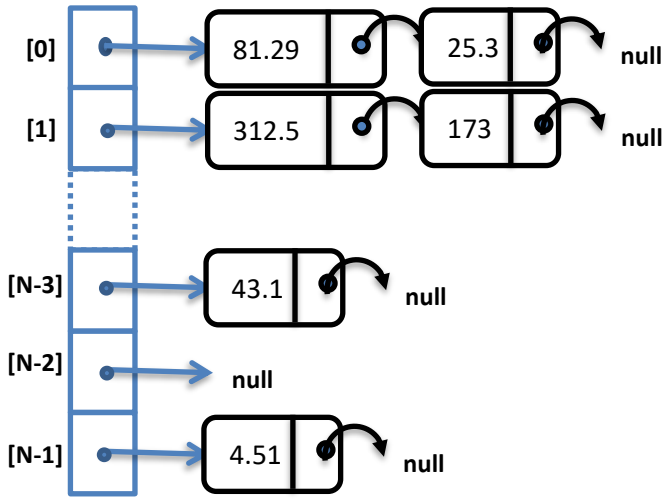


Figure 26 ***schéma d'une table de hachage de réels***

La fonction de hachage permet de séparer les données dans les différentes listes.

5.4.2. Fonction de hachage

La fonction de hachage :

La fonction de hachage permet de déterminer, à partir d'une donnée, l'indice du tableau(et donc la liste) dans lequel la donnée doit se trouver..

5.4.2.1. Exemple

Prenons un exemple pour illustrer l'utilisation de la fonction de hachage. Pour stocker 3 000 dates historiques du XXe siècle, avec un tableau de 100 listes chainées, l'idéal serait que chaque liste possède 30 dates.

- **Première fonction de hachage :** chaque liste représente une année.
  Fonction(date)=(année) MOD 100
  Chaque date est alors associée à un entier entre 0 et 99 grâce à l'opération modulo. 1900 est associée à la liste[0]. 1901 à la liste[1]. 1902 à la liste[2]..etc. Il semble probable que les listes des années riches en événements historiques (1914-1918 et 1939-1945) soient très remplies, et les autres beaucoup moins. Et rechercher une date dans une liste très longue, cela prend du temps.
- **Deuxième fonction de hachage :** utilisons le jour, le mois et l'année :
  Fonction(date)=(jour + mois + année) MOD 100
  On obtient encore un nombre compris entre 0 et 99. Chaque date est très précisément associée à un nombre. Les dates sont a priori réparties aléatoirement dans les différentes listes.

Les deux fonctions de hachage sont valides, mais la deuxième semble plus efficace que la première. Cela reste encore à vérifier concrètement par le programme en étudiant la bonne répartition du nombre de dates dans toutes les listes.

5.4.2.2. Conseils

La conception d'une table de hachage n'est pas théorique mais très pratique. Pour utiliser une table de hachage, les données doivent être réparties de manière homogènes dans les différents éléments du tableau.

L'insertion

Les données sont partiellement triées : pour une valeur donnée, on sait immédiatement, grâce à la fonction de hachage, à quelle liste chainée elle appartient. Il suffit d'insérer en tête cette valeur sur la liste associée(par la fonction de hachage).

La recherche :

Pour une valeur donnée, on sait immédiatement, grâce à la fonction de hachage, à quelle liste chainée elle appartient.
Par contre, la liste chainée n'est pas triée : il reste encore à la parcourir pour rechercher la valeur. D'où la nécessité de disposer de petites listes, et donc d'une fonction de hachage qui répartisse au maximum les données stockées dans toutes les listes.

Le nombre de listes

Le nombre d'éléments du tableau dépend du nombre de données à gérer. Des listes d'une centaine d'éléments semblent efficaces donc pour un dictionnaire de 30 000 mots, il suffira d'avoir 300 listes. En revanche, si vous n'avez que 200 valeurs à stocker avec 300 listes, cette structure est contreproductive.

Penser toujours à adapter le nombre de liste au nombre de données.

Changer le nombre de listes

Les inconvénients d'un programme proviennent souvent de l'évolution de son utilisation. Si vous aviez dimensionné votre table de hachage avec des listes de 50 éléments pour 5000 données, et qu'au cours du temps, le nombre de données est passé à 50000, votre programme se trouvera ralenti. Pour cela, il est judicieux de pouvoir augmenter dynamiquement le nombre de listes (et donc la dimension du tableau de listes). Cette opération demande du temps car il faut replacer tous les éléments dans la nouvelle liste.

6. ENONCES DES EXERCICES

6.1. Première série

Exercice 1 Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B en Entier
Début
A ← 1
B ← A + 3
A ← 3
Fin

Exercice 2 Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

Variables A, B, C en Entier
Début
A ← 5
B ← 3
C ← A + B
A ← 2
C ← B – A
Fin

Exercice 3 Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B en Entier
Début
A ← 5
B ← A + 4
A ← A + 1
B ← A – 4
Fin

Exercice 4 Que produit l'algorithme suivant ?

Variables A, B, C en Caractères
Début
A ← "423"
B ← "12"
C ← A + B
Fin

Exercice 5 Que produit l'algorithme suivant ?

Variables A, B, C en Caractères
Début
A ← "423"
B ← "12"
C ← A & B
Fin

Exercice 6 Donner le contenu de la variable X

Variable X : entier
Début
X ←3
X + 1 ←X mod 2
Fin

Exercice 7: Quel est le type et la valeur des expressions suivantes (NB: certaines peuvent être erronées) :

1 "VRAI" 17.86 2*2≠3+1 6 < 7 ou 8 < 5
1,5 4+5 19/1.9 6=5 non (6>7)

2,0	67	non VRAI	"12" > "2"	x≤0 ou x>0
"bonjour"	8*9	"2"+"3"	""=""	"bonjour" + "le monde"
"3"	11/10	"1+2"	2=2.0	VRAI et FAUX
" "	12 div 13	2 et 2	3=3.1	VRAI ou FAUX
""	14 mod 5	1 > 2	4/5=4 div 5	"VRAI et FAUX"
FAUX	15+16.0	3 ≤ 4	6 < 7 et 8 < 5	"bonjour" ≤ "le monde"

Exercice 8 Dans un algorithme complet, comme cela a été présenté en cours, cinq sections essentielles doivent apparaître. Donner la liste de ces sections, ainsi qu'une brève description de leur rôle.

Exercice 9 Écrire un algorithme permettant de calculer le volume d'un écrou hexagonal régulier de côté c et de hauteur h, évidé au centre d'un cylindre de rayon r.

Indication L'aire d'un polygone régulier à n côtés est :

$$\mathcal{A} = \frac{n \times t^2}{4 \times \tan(\pi/n)}$$

Où t est la longueur d'un côté.

Exercice 10 Ecrire un algorithme qui demande les coordonnées de deux points dans le plan, calcule et affiche à l'écran la distance entre ces deux points.
N.B. la distance entre deux points A(x1,y1) et B(x2,y2) est : AB= sqrt((x2-x1)^2 + (y2-y1)^2)
On donne la fonction sqrt(x) qui renvoie la racine carrée d'un nombre réel x.

Exercice 11 Elaborer un algorithme permettant de demander les valeurs de trois résistances r1,r2 et r3 et de calculer et afficher leurs résistances équivalente dans les deux cas suivants :
N.B. Lorsque ces résistances sont branchées en série :
Rser = r1+r2+r3
Lorsque ces résistances sont branchées en parallèle :
Rpar=(r1*r2*r3)/(r1*r2+r1*r3+r2*r3)

Exercice 12 Dans une école un étudiant passe quatre matières à l'examen :
1ère matière écrite : coefficient =3
2ème matière écrite : coefficient =2
1ère matière orale : coefficient =4
2ème matière orale : coefficient =5
Ecrire un algorithme permettant d'entrer toutes les notes de calculer et d'afficher la moyenne générale.

Exercice 13 Ecrire un algorithme qui lit trois nombres dans trois variables A, B et C, puis fait la permutation circulaire de ces trois nombres et affiche les nouveaux contenus des variables A,B et C .

Exercice 14 Ecrire un algorithme qui calcule le périmètre d'un cercle : p=2*π*R.

Exercice 15 Soit N un nombre entier. Proposer une opération avec laquelle nous pourrons conclure si le nombre N est pair ou impair.

Exercice 16 Ecrire un algorithme qui calcule la valeur absolue d'un nombre réel.
|x|= x si x>0
|x|= -x si x<0

Exercice 17 Ecrire un algorithme qui teste si une année est bissextile ou non.
N.B.
Une année est bissextile si elle est divisible par 4 et pas par 100 ou si elle est divisible par 400.

Exercice 18 Ecrire un algorithme permettant de résoudre une équation de deuxième degré : $ax^2+bx+c=0$

6.2. Deuxième série

Exercice 1 Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie

- "Poussin" de 6 à 7 ans
- "Pupille" de 8 à 9 ans
- "Minime" de 10 à 11 ans
- "Cadet" après 12 ans

Exercice 2 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 qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

Exercice 3 Les habitants d'une ville paient l'impôt selon les règles suivantes :

- les hommes de plus de 20 ans paient l'impôt
- les femmes paient l'impôt si elles ont entre 18 et 35 ans
- les autres ne paient pas d'impôt

Le programme demandera donc l'âge et le sexe des habitants, et se prononcera donc ensuite sur le fait que l'habitant est imposable.

Exercice 4 Une boulangerie est ouverte de 7 heures à 13 heures et de 16 heures à 20 heures, sauf le lundi après-midi et le mardi toute la journée. On suppose que l'heure h est un entier entre 0 et 23. Le jour j code 0 pour lundi, 1 pour mardi, etc.

Ecrire un programme qui demande le jour et l'heure, puis affiche si la boulangerie est ouverte.

Exercice 5 Ecrire un algorithme qui déclare et remplisse un tableau de 7 valeurs numériques en les mettant toutes à zéro.

Exercice 6 Ecrire un algorithme qui déclare et remplisse un tableau contenant les six voyelles de l'alphabet latin.

Exercice 7 Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Exercice 8 Ecrivez un algorithme constituant un tableau, à partir de deux tableaux de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.

Exercice 9 Toujours à partir de deux tableaux précédemment saisis, écrivez un algorithme qui calcule le **schtroumpf** des deux tableaux. Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau 1 par chaque élément du tableau 2, et additionner le tout. Par exemple si l'on a :

Tableau 1 :

4	*8*	*7*	*12*

Tableau 2 :

3	*6*

Le Schtroumpf sera :

3*4 + 3*8 + 3*7 + 3*12 + 6*4 + 6*8 + 6*7 + 6*12 = 279

Exercice 10 Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers »)

Exercice 11 Ecrivez un algorithme qui permette à l'utilisateur de supprimer une valeur d'un tableau préalablement saisi. L'utilisateur donnera l'indice de la valeur qu'il souhaite supprimer.

Attention, il ne s'agit pas de remettre une valeur à zéro, mais bel et bien de la supprimer du tableau lui-même ! Si le tableau de départ était :

12	*8*	*4*	*45*	*64*	*9*	*2*

Et que l'utilisateur souhaite supprimer la valeur d'indice 4, le nouveau tableau sera :

12	8	4	45	9	2

Exercice 12 Ecrire l'algorithme qui permet de diviser l'entier A (positif ou nul) sur l'entier B (positif) sans utiliser le DIV et le MOD. Le prog doit afficher le résultat et le reste de la division.

Exercice 13 Soit une matrice A(n,m) d'entier :

Ecrire l'algorithme qui permet de :

1. Compter le nombre de valeurs positives de chaque colonne et calculer leur somme.

2. Compter le nombre de valeurs négatives de chaque ligne et calculer leur produit.

Exercice 14 Soit A(n,n) une matrice carrée d'entiers (n<=50). Ecrire un programme qui permet de permuter les deux diagonales.

Exemple :

1 2 5 **6**	**6** 2 5 1
7 **9** **1** 4	7 **1** 9 4
8 **2** **7** 1	8 7 **2** 1
0 2 5 **3**	3 2 5 **0**

Exercice 15 Ecrire un algorithme "Palind" qui lit une chaîne de caractères et vérifie si cette chaine est un palindrome ou non.

Un palindrome est un mot qui peut être lu indifféremment de droite à gauche ou de gauche à droite (Exemple: "AZIZA", "LAVAL", "RADAR",…).

Exercice 16

Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces.

Exercice 17

Une société Privée Calcule la paye d'un salarié de la façon suivante ;

	Salaire<15000 DA	15000≤Salaire<30000 DA	Salaire≥30000 DA
Prime de Panier : par jour travaillé…	2% du salaire	1,5% du salaire	1% du salaire
Plus Prime Spéciale pour chaque Jour de repos travaillé	200DA/jour	500DA/jour	2000DA/jour

Pour simplifier on va dire qu'un mois se constitue toujours de 30 jours et que le nombre de jours repos dans un mois est de « 8 jours », le reste seront des jours normaux de travail. Le responsable de la paye aura besoins qu'on lui facilite la tâche par un programme qui, à partir du salaire et du nombre de jour travail et parmi les 30 jours du mois ainsi que la définition des primes, calcule la somme en argent qu'il doit verser à l'employer.

6.3. Troisième série

Exercice 1

Donner la fonction qui calcule le nombre de radiateurs dont on a besoin pour chauffer une pièce. On sait qu'un radiateur est capable de chauffer $8m^3$. L'utilisateur donnera comme paramètre d'entrée la longueur, la largeur et la hauteur de la pièce en mètres.

Exercice 2

Écrire une fonction pour convertir un nombre de secondes en un nombre d'heures, de minutes et de secondes. On utilisera les opérateurs modulo et division entière.

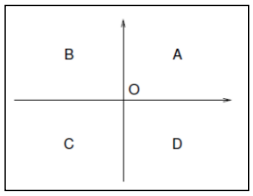
Exercice 3

On ne fait pas forcément des économies en achetant plus gros. Est-ce vrai quand on achète des pizzas ? Concevoir un algorithme qui lit le diamètre de deux pizzas, leur numéro et leur prix puis qui fait appel à une fonction qui imprime le numéro de celle qui a le meilleur rapport taille/prix.

Exercice 4

Écrire une fonction qui, étant donné les coordonnées x et y d'un point, détermine dans quelle partie (A, B, C ou D) du plan se trouve le point (voir figure1).

figure1 : Parties du plan



Exercice 5

Écrire une fonction qui demande en entrée un numéro de mois et indique en retour son nom et le nombre de jours dans ce mois.

Exercice 6

Ecrire une fonction MIN et une fonction MAX qui déterminent le minimum et le maximum de deux nombres réels.
Ecrire un programme se servant des fonctions MIN et MAX pour déterminer le minimum et le maximum de quatre nombres réels entrés au clavier.

Exercice 7

Ecrire un programme se servant d'une fonction F pour afficher la table de valeurs de la fonction définie par

$f(x) = \sin(x) + \ln(x) - \sqrt{x}$

Où x est un entier compris entre 1 et 10. On suppose que les fonctions mathématiques sin et ln sont prédéfinis.

Exercice 8

Ecrire un programme qui construit et affiche le triangle de Pascal en calculant les coefficients binomiaux:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
  . . .
```

On n'utilisera pas de tableau, c.-à-d. il faudra calculer les coefficients d'après la formule ci-dessous, tout en définissant et utilisant les fonctions adéquates.

$$C_p^q = \frac{p!}{q!(p-q)!}$$

Exercice 9

Ecrire la fonction MULTI_MATRICE qui effectue la multiplication de la matrice MAT1 par un entier X:
MAT1 = X * MAT1
Choisir les paramètres nécessaires et écrire un petit programme qui teste la fonction MULTI_MATRICE.

Exercice 10

Ecrire la fonction MULTI_2_MATRICES qui effectue la multiplication de deux matrices MAT1 (dimensions N et M) et
MAT2 (dimensions M et P) en une troisième matrice MAT3 (dimensions N et P): MAT3 = MAT1 * MAT2

Exercice 11

Quel résultat cet algorithme produit-il ?

```
Variable Truc en Caractère
Début
Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5)
LireFichier 5, Truc
Ecrire Truc
FinTantQue
Fermer 5
Fin
```

Exercice 12

On travaille avec le fichier du carnet d'adresses en champs de largeur fixe. Ecrivez un algorithme qui permet à l'utilisateur de saisir au clavier un nouvel individu qui sera ajouté à ce carnet d'adresses.

Exercice 13

Ecrire des procédures ou bien des fonctions qui :

1. Affiche la nature d'un entier (pair ou impair)
2. Calcul la moyenne des notes saisisses ;
3. Calcul le maximum entre 3 entiers ;
4. Calcul Le factoriel d'un nombre entier ; donner un concept ;
5. calcule la valeur approchée de e^x en faisant appel aux fonctions fact et puiss et en utilisant son développement limité.

$$e^x \simeq 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

6. qui calcule la valeur approchée de e^x en s'arrêtant lorsque le terme 1/x est plus petit que ξ. Faire un programme qui lit ξ réel, appelle calcpi puis affiche le résultat.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots$$

6.4. Quatrième série

Exercice 1

Créer un enregistrement nommé « Etudiant » qui est caractérisé par un identifiant, un nom et un prénom. On vous demande de saisir 10 étudiants, les ranger dans un tableau puis les afficher.

Exercice 2

Écrire un algorithme qui lit deux nombres complexes C1 et C2 et qui affiche en suite leur somme et leur produit.

On utilisera les formules de calcul suivantes:

(a+b**i**)+(c+d**i**)=(a+c)+(b+d)**i**

(a+b**i**) * (c+d**i**) = (ac – bd)+(ad + bc)**i**

Exercice 3

Soit la déclaration suivante :

type t_chanson = ENREGISTREMENT

titre: chaine de caractère;

artiste: chaine de caractère; album: chaine de caractère;

longueur: entier;

date: t_date;

FIN;

type t_tableau_chansons = tableau[1..n] de t_chanson;

var collection : t_tableau_chansons;

Trouver l'artiste d'une chanson donnée 'kaya'

Exercice 4

On souhaite mémoriser des noms des personnes dans un fichier nommé *« pers.dat ».* On vous demande alors de créer les sous-programmes qui suivent :

- Une procédure de création du fichier qui contient les noms des personnes.
- Une procédure d'affichage des noms de personnes.
- Une fonction qui permet de chercher un nom passé en argument et qui renvoie vrai si ce dernier est existant et faux sinon.

Écrire le programme principal faisant appel aux différents sous-programmes.

Exercice 5

Écrire une fonction récursive qui retourné la somme des chiffres d'un entier N donné. Exemple :(123 == > 1 + 2 + 3 = 6)

Exercice 6

Écrire une procédure récursive qui, étant donné un entier N en base 10, l'imprime en base 2.

Exercice 7

Écrire une procédure récursive qui permet d'inverser une chaine de caractères sans utiliser une chaine temporaire.

Exemple : information → noitamrofni

Exercice 8

Veuillez rappeler le principe de fonctionnement du partition-nement dans le tri rapide en vous basant sur l'exemple suivant :

4	1	2	8	5	4	3	8	9

Exercice 9

Écrire un algorithme itératif donnant l'indice de la première occurrence d'un élément minimal dans un tableau de N entiers (on suppose N >= 2). Évaluer sa complexité.

Exercice 10

On appelle MiniMax d'une matrice d'entiers la valeur minimale des maxima de chaque ligne.
Écrire un algorithme qui retourne la valeur MiniMax d'une matrice d'entiers n×m. On suppose N >=1 etM >= 1. Évaluer sa complexité.

Exercice 11

1. Donner une fonction récursive puissance(a,i) qui calcule a^i.
2. Calculer la complexité de l'algorithme précédent (puissance).
3. Donner une fonction récursive somme(a,n) qui calcule $\sum_{i=0}^{n} a^i$. 4. Calculer la complexité de l'algorithme précédent (somme).

Exercice 12

Que calcule la fonction suivante :

```
Fonction quid(x : entier, n : entier) : entier ;
Debut
        si (n=0)
                retourner 1;
        si (n=1)
                retourner x;
        sinon
                retourner (quid(x,n-1)*x);
Fin
```

Quelle est sa complexité ?

7. CORRIGES DES EXERCICES

7.1. Solutions première série

Exercice 1

Après	La valeur des variables est :
A ← 1	A = 1 B = ?
B ← A + 3	A = 1 B = 4
A ← 3	A = 3 B = 4

Exercice 2

Après	La valeur des variables est :
A ← 5	A = 5 B = ? C = ?
B ← 3	A = 5 B = 3 C = ?
C ← A + B	A = 5 B = 3 C = 8
A ← 2	A = 2 B = 3 C = 8
C ← B – A	A = 2 B = 3 C = 1

Exercice 3

Après	La valeur des variables est :
A ← 5	A = 5 B = ?
B ← A + 4	A = 5 B = 9
A ← A + 1	A = 6 B = 9
B ← A – 4	A = 6 B = 2

Exercice 4

Il ne peut produire qu'une erreur d'exécution, puisqu'on ne peut pas additionner des caractères.

Exercice 5

…En revanche, on peut les concaténer. A la fin de l'algorithme, C vaudra donc "42312".

Exercice 6

Il ne peut produire qu'une erreur d'exécution, puisque l'opérateur de gauche d'une affectation ne peut pas contenir une opération.

Exercice 8

Les cinq sections essentielles dans un algorithme sont :

Données description des données à fournir à l'algorithme
Résultat description du ou des résultats de l'algorithme
Idée de l'algorithme description, en langage naturel, des grandes étapes de la résolution du problème
Lexique des variables liste des variables utilisées par l'algorithme
Algorithme description précise, en langage algorithmique, des étapes de résolution du problème

Exercice 9

Données

Pour un écrou hexagonal, la longueur d'un côté, sa hauteur, et le rayon du cylindre évidé.

Résultat

Volume de l'écrou.

Idée

Calculer l'aire de la base en retranchant l'aire du disque (base du cylindre) à l'aire du polygone. Multiplier l'aire de
la base par la hauteur.

Lexique des constantes

N (entier) = 6 nombre de côtés de l'écrou
PI (réel) = 3,14 constante mathématique π

Lexique des variables

c (réel)	longueur d'un côté	DONNÉE
h (réel)	hauteur de l'écrou	DONNÉE
r (réel)	rayon du cylindre évidé	DONNÉE
aire (réel)	aire de la base	INTERMÉDIAIRE
volume (réel)	volume de l'écrou	RÉSULTAT

Algorithme

aire ← *N* × *c* × *c*/(4 × tan(*PI*/*N*))
aire ←*aire* − *PI* × *r* × *r*
volume ← *aire* × *h*

Exercice 10

```
Algorithme  calcul_distance;
Var
    X1,x2,y1,y2,s : réels ;
Debut
        Ecrire('entrer la valeur de x1 : ') ;
        Lire(x1) ;
        Ecrire('' entrer la valeur de y1 : '') ;
        Lire(y1) ;
Ecrire('' entrer la valeur de x2 : '') ;
        Lire(x2) ;
Ecrire('' entrer la valeur de y2 : '') ;
        Lire(y2) ;
        S ← sqrt((x2-x1)^2  + (y2-y1)^2) ;
        Ecrire('la distance entre A(' ,x1,',',y1,') et  B(',x2,',',y2,') est :  ',s) ;
Fin
```

Exercice 11

```
Algorithme  calcul_resistance;
Var
    r1,r2,r3,Rpar,Rser: réels ;
Debut
          Ecrire('entrer la valeur de r1 : ') ;
          Lire(r1) ;
          Ecrire(' entrer la valeur de r2 : ') ;
          Lire(r2) ;
Ecrire(' entrer la valeur de r3 :  ') ;
          Lire(r3) ;
          Rser ← r1+r2+r3;
          Rpar ←  (r1*r2*r3)/(r1*r2+r1*r3+r2*r3) ;
          Ecrire(' la résistance équivalente a r1 ,r2 et r3 en série est : ',Rser) ;
Ecrire(' la résistance équivalente a r1 ,r2 et r3 en parallèle  est : ',Rpar) ;
Fin
```

Exercice 12

```
Algorithme  calcul_note;
Var
        me1,me2,mo1,mo2,moy : réels ;
```

```
const
        cme1=3 ;
        cme2=2 ;
         cmo1=4 ;
        cmo2=5 ;
Debut
          Ecrire('entrer la  note du 1ère matiere écrite : ') ;
          Lire(me1) ;
          Ecrire('entrer la  note du 2ème matiere écrite : ') ;
          Lire(me2) ;
   Ecrire('entrer la  note du 1ère matiere orale : ') ;
          Lire(mo1) ;
   Ecrire('entrer la  note du 2ème matiere orale: ') ;
          Lire(mo2) ;
          moy<--  (me1*cme1+me2*cme2+mo1*cmo1+mo2*cmo2)/(cme1+cme2+cmo1+cmo2) ;
  Ecrire(' la moyenne generale est   :  ',moy) ;
Fin
```

Exercice 13

```
Algorithme  calcul_permutation;
Var
       A,b,c,aux : réels ;
Debut
          Ecrire('entrer la  de a : ') ;
          Lire(a) ;
          Ecrire('entrer la  de b : ') ;
          Lire(b) ;
   Ecrire('entrer la  de c : ') ;
          Lire(c) ;
   Ecrire('a = ',a,'     b= ',b,'        c=',c) ;
          aux← a ;
          a<--  b ;
          b<--  c ;
          c<--  aux ;
Ecrire('a = ',a,'     b= ',b,'        c=',c) ;
fin
```

Exercice 14

```
Algorithme  calcul_perimetre;
Const
       Pi=3.14 ;
Var
        R,p: réels ;
Debut
          Ecrire('entrer le rayon R : ') ;
          Lire(R) ;
          P<--  2*pi*R
Ecrire(' le périmètre du cercle R=',R,' est   :',p) ;
```

```
Fin
```

Exercice 15

```
Algorithme  parite;
Var
    N :entier;
Debut
          Ecrire('entrer  un entier : ') ;
          Lire(N) ;
          Si  N mod 2 =0 alors
     Ecrire('le nombre est pair);
  Else
      Ecrire('le nombre est impair .');
  Finsi
Fin
```

Exercice 16

```
Algorithme  valeur_absolue;
Var   X :réels;
Debut
          Ecrire('entrer  un nombre : ') ;
          Lire(X) ;
          Si  X>0 alors   Ecrire('la valeur absolue de X=',X);
          Sinon  Ecrire('la valeur absolue de X=',X);
          Finsi
Fin
```

Exercice 17

```
Algorithme  annne_bissextile;
Var
    annee :réels;
Debut
          Ecrire('entrer  l'année : ') ;
          Lire(annee) ;
  Si  ((annee mod 4 =0 et annee mod 100 <> 0) ou annee mod 400 =0 )  alors
     Ecrire('l'année que vous avez entrer est bissextile .');
  Sinon
      Ecrire('l'année que vous avez entrer n' est pas  bissextile .');
  Finsi
Fin
```

Exercice 18

```
Algorithme  eq2deg
Var
        A,b,c,d : réels ;
Debut
          Ecrire('entrer le coefficient a : ') ;
          Lire(a) ;
          Ecrire('entrer le coefficient  b : ') ;
          Lire(b) ;
```

```
Ecrire('entrer le coefficient  c : ') ;
        Lire(c) ;
        Si a=0 alors
           Si b=0 alors
              Si c=0 alors
                 Ecrire(' la solution est   : S = R') ;
               sinon
                  Ecrire(' l'equation n'a pas de solution ') ;
              Finsi
           sinon
              Ecrire('la solution est   : S = ',-c/b) ;
           Finsi
        sinon
           D<-- b*b-4+a*c ;
            Si d=0 alors
              Ecrire('la solution est   : S = ',-b/(2*a)) ;
            Sinon si d>0 alors
                    Ecrire('l'equation a deux solution:  S1= ',(-b- /(2*a) ,' et  S2 = ',(-b+ /(2*a)) ;
                Sinon
                   Ecrire('l'equation n'a pas de solution dans R ') ;
                Finsi
            Finsi
        Finsi
fin
```

7.2. Solutions deuxième série

Exercice 1

```
Variable age en Entier
Début
        Ecrire "Entrez l'âge de l'enfant : "
        Lire age
        Si age >= 12 Alors
                Ecrire "Catégorie Cadet"
        SinonSi age >= 10 Alors
                Ecrire "Catégorie Minime"
            SinonSi age >= 8 Alors
                Ecrire "Catégorie Pupille"
                SinonSi age >= 6 Alors
                        Ecrire "Catégorie Poussin"
        Finsi
Fin
```

On peut évidemment écrire cet algorithme de différentes façons, ne serait-ce qu'en commençant par la catégorie la plus jeune.

Exercice 2

```
Variables n, p en Numérique
Début
        Ecrire "Nombre de photocopies : "
        Lire n
        Si n <= 10 Alors
                p ← n * 0,1
        SinonSi n <= 30 Alors
        p ← 10 * 0,1 + (n – 10) * 0,09
             Sinon
                p ← 10 * 0,1 + 20 * 0,09 + (n – 30) * 0,08
        FinSi
Ecrire "Le prix total est: ", p
Fin
```

Exercice 3

```
Variable sex en Caractère
Variable age en Numérique
Variables C1, C2 en Booléen
Début
        Ecrire "Entrez le sexe (M/F) : "
        Lire sex
        Ecrire "Entrez l'âge: "
        Lire age
        C1 ← sex = "M" ET age > 20
        C2 ← sex = "F" ET (age > 18 ET age < 35)
        Si C1 ou C2 Alors
        Ecrire "Imposable"
        Sinon
        Ecrire "Non Imposable"
        FinSi
Fin
```

Exercice 4

```
Début
Ecrire "Entrez le jour :"
Lire j
Ecrire "Entrez l'heure :"
Lire h
b ← (h >= 7) and (h <= 13) and (j <> 1) or (h >= 16) and (h <= 20) and (j > 1);
si b alors  Ecrire "Boulangerie ouverte :"
sinon Ecrire " Boulangerie fermé"
Fin
```

Exercice 5

```
Tableau Truc(6) en Numérique
Variable i en Numérique
Debut
Pour i ← 0 à 6
Truc(i) ← 0
i Suivant
Fin
```

Exercice 6

```
Tableau Truc(5) en Caractère
Debut
Truc(0) ← "a"
Truc(1) ← "e"
Truc(2) ← "i"
Truc(3) ← "o"
Truc(4) ← "u"
Truc(5) ← "y"
Fin
```

Exercice 7

```
Tableau Notes(8) en Numérique
Variable i en Numérique
Pour i ← 0 à 8
Ecrire "Entrez la note numéro ", i + 1
Lire Notes(i)
i Suivant
Fin
```

Exercice 8

```
Variables i, N en Numérique
Tableaux T1(), T2(), T3() en Numérique
Debut
… (on suppose que T1 et T2 comptent N éléments, et qu'ils sont déjà saisis)
Redim T3(N-1)
…
Pour i ← 0 à N - 1
T3(i) ← T1(i) + T2(i)
i Suivant
Fin
```

Exercice 9

```
Variables i, j, N1, N2, S en Numérique
Tableaux T1(), T2() en Numérique
Debut
… On ne programme pas la saisie des tableaux T1 et T2.
On suppose que T1 possède N1 éléments, et que T2 en possède T2)
…
S ← 0
Pour i ← 0 à N1 – 1
Pour j ← 0 à N2 – 1
S ← S + T1(i) * T2(j)
j Suivant
i Suivant
Ecrire "Le schtroumpf est : ", S
Fin
```

Exercice 10

```
On suppose que n est le nombre d'éléments du tableau préalablement saisi
…
Pour i ← 0 à (N-1)/2
Temp ← T(i)
T(i) ← T(N-1-i)
T(N-1-i) ← Temp
i suivant
Fin
```

Exercice 11

```
Ecrire "Rang de la valeur à supprimer ?"
Lire S
Pour i ← S à N-2
T(i) ← T(i+1)
i suivant
Redim T(N–1)
Fin
```

Exercice 12

```
Programme Division
Var A,B,Q ,R :entier
Début
Si A>=0 et B>0
        Lire(A,B)
R←A
Q←0
Tanque (R>= B) faire
R←R-B
Q←Q+1
Fin tant que
Ecrire(R,Q)
Fin
```

Exercice 13

```
Programme Matrice
Var A : tableau[1..n,1..m] : entier
i, j, NBP, NBN, SP, PN: entire
Début
Pour j=1 à m
   NBP←0
   SP←0
   Pouri=1 à n
     Lire A[i,j]
     Si A[i,j]>0 alors NBP←NBP+1
     SP←SP+ A[i,j]
     FinSi
   Fin pour
   Ecrire (NBP, SP)
Fin pour
Pour j=1 à n
   NBN ←0
   PN ←1
   Pouri=1 à m
     Lire A[i,j]
     Si A[i,j]<0 alors
     NBP←NBP+1
     SP←SP* A[i,j]
     FinSi
   Fin pour
   Ecrire (NBN, PN)
Fin pour
```

Exercice 15

```
Algorithme Palind
Var  ch: Chaine  i, L : Entier  Pal : Booléen
```

```
Début
Ecrire("Entrer une chaîne non vide="),
Lire(ch)
L← long(ch)  (* longueur de la chaîne *)
Pal ←Vrai  (* on suppose initialement que la chaîne est palindrome *)
i ←1 (* initialisation du compteur i *)
 Tant que ( i <= L Div 2) ET (Pal) Faire  (* Parcours de la chaîne jusqu'à la moitié *)
          Si ( ch[i] = ch[L-i+1] ) Alors     (* s'il sont égaux alors on incrémente *)
 i ← i + 1  (* incrémentation *)
Sinon    Pal←  Faux (* s'il y' a deux lettres différentes alors on s'arrête  *)   Fin Si
 Fin Tant que
Si (Pal) Alors Ecrire(ch, " est un palindrome")
Sinon Ecrire(ch, " n'est pas un palindrome")
 Fin Si
Fin.
```

Exercice 16

```
Variable Bla en Caractère
Variables Nb, i en Entier
Debut
Ecrire "Entrez une phrase : "
Lire Bla
Nb ← 0
Pour i ← 1 à Len(Bla)
Si Bla(i) = " " Alors
Nb ← Nb + 1
FinSi
i suivant
Ecrire "Cette phrase compte ", Nb + 1, " mots"
Fin
```

7.3. Solutions troisième série

Exercice 1

```
Algorithme
Variables longueur, largeur, hauteur : réel ;
Debut
Lire(longueur) ;
Lire(largeur) ;
Lire(hauteur) ;
nbr ← calcul_nbrRad(longueur, largeur,hauteur) ;
écrire(« le nombre de radiateurs est », nbr) ;
Fin
Fonction calcul_nbrRad(a,b,c : réel) : entier
Variables volume :réel
Nombre : entier
Debut
```

Volume ?longueur × largeur × hauteur
Nombre ← ⌈volume DIV capacité⌉
retourner (nombre) ;
Fin

Exercice 2

Fonction convertirTemps(duree :entier)
Variables
heures : entier ;
minutes : entier ;
secondes : entier ;
Debut
heures ← duree DIV 3600
minutes ← (duree MOD 3600) DIV 60
secondes ← duree MOD 60
écrire(« votre durée en heure, minutes et secondes »,heures, minutes, secondes) ;
Fin

Exercice 3

Algorithme
Variables numéro1 :entier ;
numéro2 :entier ;
diamètre1 :réel ;
diamètre2 :réel ;
prix1 :réel ;
prix2 :réel ;
Debut
Ecrire("Entrer numéro, diamètre et prix de la première pizza") ;
lire(numéro1) ;
lire(diamètre1) ;
lire(prix1) ;
écrire ("Entrer numéro, diamètre et prix de la deuxième pizza") ;
lire(numéro2) ;
lire(diamètre2) ;
lire(prix2) ;
AfficheMQP(numéro1,numéro2,diamètre1,diamètre2,prix1,prix2)
Fonction AfficheMQP(n1,n2,d1,d2,p1,p2)
Contante PI :réel= 3,141 ;
Variables
rapport1 :réel ;
rapport2 :réel ;
debut
rapport1 ← d1 × d1 × PI/(4 × p1)
rapport2 ← d2 × d2 × PI/(4 × p2)
si rapport1 > rapport2 alors
´écrire "C'est la pizza", n1, "qui a le meilleur rapport taille/prix."
sinon si rapport1 < rapport2 alors
écrire "C'est la pizza", n2, "qui a le meilleur rapport taille/prix."

```
                    sinon
                        écrire "Les deux pizzas on le même rapport taille/prix."
        Fsi
Fin
```

Exercice 4

```
Fonction partiesPlan(x :réel, y :réel)
Debut
si x < 0 alors
        si y < 0 alors
                écrire "C"
        sinon si y > 0 alors
                écrire "B"
                sinon // y = 0
                écrire "BC"
        fsi
sinon si x > 0 alors
        si y < 0 alors
                écrire "D"
        sinon si y > 0 alors
                écrire "A"
              sinon // y = 0
                écrire "AD"
        fsi
sinon // x = 0
        si y < 0 alors
                écrire "CD"
        sinon si y > 0 alors
                écrire "AB"
               sinon // y = 0
                écrire "O (ABCD)"
        fsi
fsi
```

Exercice5

```
Fonction  MoisNomNbJours(mois :entier)
debut
selon que mois est
        cas 1 : écrire "janvier (31 jours)"
        cas 2 : écrire "février (28 ou 29 jours)"
        cas 3 : écrire "mars (31 jours)"
        cas 4 : écrire "avril (30 jours)"
        cas 5 : écrire "mai (31 jours)"
        cas 6 : écrire "juin (30 jours)"
        cas 7 : écrire "juillet (31 jours)"
        cas 8 : écrire "août (31 jours)"
        cas 9 : écrire "septembre (30 jours)"
```

```
        cas 10 : écrire "octobre (31 jours)"
        cas 11 : écrire "novembre (30 jours)"
        cas 12 : écrire "décembre (31 jours)"
        défaut : écrire "numéro invalide"
fselon
Fin
```

Exercice 6

```
Algorithme
Variables A,B,C,D : réel ;
Debut
écrire("Introduire 4 réels : ");
lire("A, B, C, D);
écrire("Le minimum des 4 réels est ",          MIN( MIN(A,B), MIN(C,D)) );
 écrire("Le maximum des 4 réels est ",
    MAX( MAX(A,B), MAX(C,D)) );
Fin

Fonction MIN(X :réel, Y :réel) :réel
début
 si (X<Y)
    retourner X;
sinon
    retourner Y;
Fsi
Fin
Fonction MAX(X :réel, Y :réel) :réel
début
 si (X>Y)
    retourner X;
sinon
    retourner Y;
Fsi
Fin
```

Exercice 7

```
Algorithme
Variables I:entier;
Debut
Pour I de 1 à 10 faire
écrire(I, F(I));
Fin
Fonction F(X :entier) : réel
{
 retourner sin(X)+log(X)-sqrt(X);
}
```

Exercice 8

Algorithme

```
Debut
Variables N :entier;
 debut
écrire("Introduire le nombre de lignes N : ");
 lire(N);
 TRIANGLE(N);
 Fin
Fonction TRIANGLE(LIGNES : entier)
Variables P :entier;
debut
Pour P de 0 à LIGNES faire
    LIGNEC(P);
FinPour
fin
fonction LIGNEC(P :entier)
Variables Q :entier;
debut
Pour Q de 0 à P faire
    écrire("        ", C(P,Q));
    écrire("\n"); //retour à la ligne
fin
Fonction C(P :entier, Q :entier) :entier
debut
retourner FACT(P)/(FACT(Q)*FACT(P-Q));
}
 FACT(N :entier) :entier
{
Retourner N*FACT(N-1) ;
}
```

Exercice 9

```
Algorithme
 Variables
 Tableaux d'entiers M[30][30];
L, C : entier
 X :entier ;
//LIRE MATRICE
écrire("Introduire le multiplicateur (entier) : ");
lire(X);
écrire("Matrice donnée : ");
 //ECRIRE_MATRICE
 MULTI_MATRICE (X,M,L,C,30);
//ECRIRE_MATRICE
}
Fonction MULTI_MATRICE(X :entier, MAT : tableaux d'entiers, L :entier, C :entier, CMAX :entier)
Variables I,J :entier;
debut
Pour I de 0 à L
```

```
    Pour J de 0 à C
          MAT[I,J]← MAT[I,J]*X;
     FPour
FPour
Fin
```

Exercice 10

```
Fonction MULTI_2_MATRICES (MAT1, MAT2 : tableaux d'entier, N :entier, K :entier, P :entier)
Variables I,J,K :entier;
debut
Pour I de 0 à N)
   Pour J de 0 à P      {
      MAT3[I,J]← 0;
      Pour K de 0 à M
         MAT3[I,J]← MAT1[I,k]* MAT2[k,J]
}
```

Exercice 11

Cet algorithme écrit l'intégralité du fichier "Exemple.txt" à l'écran

Exercice 12

```
Variables Nom, Prénom, Tel,
Mail, Lig : chaine de caractères
Début
Ecrire "Entrez le nom : "
Lire Nom
Ecrire "Entrez le prénom : "
Lire Prénom
Ecrire "Entrez le téléphone : "
Lire Tel
Ecrire "Entrez le mail : "
Lire Mail
Lig ← Nom & Prénom & Tel & Mail
Ouvrir "Adresse.txt" sur 1 pour Ajout
EcrireFichier 1, Lig
Fermer 1
Fin
```

7.4. Solutions quatrième série

Exercice 1

```
Fonction min(tab[N] : entier) : entier
Variables : i, index :entier
Debut
        // 1 première affectation de index
        index ← 0;
        // n comparaisons + n-1 affectations
        Pour i de 1 à N-1 faire
        {
```

```
            // n-1 comparisons
            // dans le pire des cas, n-1 affectations
            Si (tab[i] < tab[index])
            index ← i;
        }
        retourner index;
Fin
```

– complexité : T(n) = 1+n+(n−1)+2*(n−1) = 4n−1 ;
– ordre : O(n) ;

Exercice 2

```
MiniMax(int tab[N][M] : entier) : entier
Variables : i,j,max, minimax : entier ;
Debut
// (n-1) comparaisons // n affectations
Pour i de 1 à n-1
{
// n initialisations du max. pour chaque nouvelle ligne
        max ← tab[i][0];
// (n-1)*m comparaisons
        // (n-1)(m-1) affectations
        Pour j de 1 à m-1
        {
                // (n-1)(m-1) comparaisons
                // (n-1)(m-1) affectations
                Si (tab[i][j] < max)
                max ← tab[i][j];
        }
        // m comparaisons // 1 affectation
        Si(i=0)
                minimax ← max;
                // n comparaisons
                // (n-1) comparaisons
        Sinon si(minimax > max)
                minimax ← max;
}
retourner minimax;
Fin
```

– complexité :
T(n) =
(n−1)+n+n+(n−1)————————————————m+(n−1)(m−1)+(n−1)(m−1)+(n−1)(m−1)+m+1+n+(n−1) = 5nm+n−4m+3 ;
– ordre : O(nm), si m = n alors on est en $O(n^2)$;

Exercice 3

La fonction calcule x^n pour tout entier x et pour tout entier n ≥ 0.
Dans le pire des cas, on a :
T(quid(x,n)) = 3+T(quid(x,n−1))

= 3 + 3 + T(quid(x,n−2)
= 3 + 3 +...+ 3 + T(quid(x,1))
= 3 + 3 +...+3+2= 3n−1

n−1 fois

d'où la complexité est en O(n).

Exercice 4

Type : Tpoint=Entité (
abs : entier ;
ord : entier ;
)

Exercice5

```
Algorithme  SaisieGroupe (E : nb : entier, S : Fic : fichier  de Tetd)
Var : etd : Tetd i : entier
Début
OuvrirFichier (fic, écriture)
Si EtatFichier (fic)=succès alors
        Pour i de 1 à nb faire
                EtdÍSaisieEtd( )
                EcrireFichier (fic, etd)
                FermerFichier (fic)
Sinon écrire (« Erreur »)
Fin
Fonction Moyenne (fic : fichier de Tetd ) : réel
Var : Som : réel
Début
        Som←0
        Nb← 0
        OuvrirFichier (fic, lecture)
        Si EtatFichier (fic)=succès alors
                LireFichier (fic, etd)
                Tant que EtatFichier (fic)¹FdF) faire
                        Nb←nb+1
                        Som←Som + etd.moyenne
                        LireFichier(fic, etd)
                        Retourner (Som/nb)
                        FermerFichier (fic)
        Sinon écrire (« Erreur »)
Fin
```

8. TRAVAUX PRATIQUES

8.1. TP N° 1

8.1.1 Fonctions

Exercice 1:

Ecrire la fonction C qui permet de calculer la n-ième puissance d'une valeur x et leprogramme appelant.

Exercice 2 :

1) Ecrire la fonction C qui permet de calculer la somme des éléments d'une colonne d'une matrice.
2) Ecrire la fonction C qui permet de permuter deux colonnes d'une matrice.
3) Ecrire le programme C qui utilise les fonctions précédentes pour déterminer l'indice de la colonne dont la somme des éléments est minimale et celui dont la somme est maximale et permuter ces deux colonnes.

Exercice 3 :

Ecrire la fonction qui supprime les valeurs redondantes dans un vecteur ordonné.

Exercice 4: Ecrire la fonction qui permet de construire le mot miroir d'un mot.

8.1.2. Récursivité

Exercice 1 :

Ecrire un sous programme récursif qui calcule le pgcd de deux nombres a et b.

Exercice 2 :

A) Ecrire un sous programme récursif qui vérifie si une valeur val se trouve dans un vecteur.
B) Ecrire un sous programme récursif qui vérifie si un mot est palindrome.

Exercice 3 : Nombres de Fibonacci

Ecrire une fonction récursive permettant de calculer le n-ième nombre de Fibonacci.
f0 = f1 = 1,
fn+2 = fn + fn+1.

Exercice 4 : Le problème des tours de Hanoï

On dispose de trois plots et de 64 disques, tous de rayons différents, percés en leur centre de façon à passer à travers les plots.
Au départ les 64 disques sont sur le premier plot, rangés par taille, le plus grand tout en bas.Le but est de déplacer ces disques pour les amener sur le troisième plot en suivant les règles suivantes :
– on ne peut déplacer qu'un disque à la fois ;
– à chaque instant et sur chaque plot, un disque ne peut être placé qu'au-dessus d'un disque de rayon plus grand.

8.2. Solution TP N° 1

Exercice 1

```
///****  R. TLEMSANI  ****///

#include <stdio.h>
#include <stdlib.h>

float puissance(float x, int n)
 {
   int i;
   float p=1;

   for(i=1;i<=n;i++)
     {
        p=p*x;
     }
   return p;
 }


int main()
{
float x; int n;
   printf("x="); scanf("%f",&x);
     printf("n="); scanf("%d",&n);
     if (x==0)
        {
      if (n<=0) {printf(" \n erreur \n");}
                else {printf("\n %.2f^%d=%.2f \n",0,n,0.);}
                }
   else
     {
         if (n==0) {printf("\n %.2f^%d=%.2f \n",x,0,1.);}
         else
           if (n<0) {printf("\n %.2f^%d=%.2f \n",x,n,1/puissance(x,-n));}
        else { printf("\n %.2f^%d=%.2f \n",x,n,puissance(x,n));}
     }
system("pause");
return 0;
}
```

Exercice 2

```
///****  R. TLEMSANI  .  ****///
```

```
#include <stdio.h>
#include <stdlib.h>

int som_col(int tab[100][100], int n, int j)
 {  int i;
    int som=0;
    for(i=0;i<n;i++)
      {  som=som+tab[i][j];
      }
    return som;  }

void permute_col(int tab[100][100], int n, int col1, int col2)
 { int i; int t;
   for(i=0;i<n;i++)
 {  t=tab[i][col1];
    tab[i][col1]=tab[i][col2];
    tab[i][col2]=t;
 }
 return;  }

 int main()
{int somme,cmax,cmin;
int a[100][100];
int i,j,n,m,imax,imin;
   printf("entrer le nombre de lignes"); scanf("%d",&n);
   printf("entrer le nombre de colonnes"); scanf("%d",&m);
    for (i=0;i<n;i++)
   {  for (j=0;j<m;j++)
      { printf("A[%d][%d]=",i,j);
        scanf("%d",&a[i][j]);
      }
   }
printf("\n");
for (i=0;i<n;i++)
   { for (j=0;j<m;j++)
      { printf("%d\t   ",a[i][j]);
      }
      printf("\n");
   }
     j=0;
     somme=som_col(a,n,j);
     cmax=somme;
     imax=j;
     cmin=somme;
     imin=j;
     for (j=1;j<m;j++)
    {  somme=som_col(a,n,j);
       if (somme>cmax) { cmax=somme; imax=j; }
       else  if (somme<cmin) { cmin=somme; imin=j; }
     }
     permute_col(a,n,imin,imax);

     // affichage resultat
    printf("\n");
```

```
   for (i=0;i<n;i++)
   { for (j=0;j<m;j++)
      { printf("%d\t  ",a[i][j]); }
         printf("\n");
   }
system("pause");
return 0;}
```

Exercice 3

```
///****  R. TLEMSANI  ****///

#include <stdio.h>

int main() {
  int nbr, i, j, k;
  int tab[30];

  printf(" Entrez le nombre d'éléments dans le tableau: ");
  scanf("%d", &nbr);

  printf(" Entrez les éléments du tableau: ");
  for (i = 0; i < nbr; i++)
    scanf("%d", &tab[i]);

  // Supprimer les doublons
  for (i = 0; i < nbr; i++)
  {
    for (j = i + 1; j < nbr;j++)
      {
      if (tab[j] == tab[i])
                {
        for (k = j; k < nbr; k++)
                          {
          tab[k] = tab[k + 1];
        }
        nbr--;
      }
         }
  }

  printf(" Tableau sans doublons: ");
  for (i = 0; i < nbr; i++) {
    printf("%d ", tab[i]);
  }

  return 0;
}
```

Exercice 4

```
///****  R. TLEMSANI ****///
```

```
#include <stdio.h>
#include <string.h>

void miroir ( char chaine[100], int taille)
{
  int i;
  char c;
   for (i=0; i<taille/2; i++)
   {
    c=          chaine[i];
    chaine[i]=chaine[taille-i-1];
    chaine[taille-i-1]=c;
   }
}

 int main(void){
 char chaine[100];
int taille;
 /* Saisie de la chaîne */
 printf("Entrez la chaîne : ");
 gets(chaine);
 taille = strlen (chaine);
 printf("Taille de la chaine = %d\n", taille);
 miroir(chaine,taille);
printf("Chaîne convertie : %s\n", chaine);
return 0;
 }
```

Exercice 1 recursive

```
#include <stdio.h>
 int pgcd(int nbr1, int nbr2)
{
if (nbr2 != 0)
return pgcd(nbr2, nbr1%nbr2);
else
return nbr1;
}
int main()
{
int nbr1, nbr2;
 printf("Entrez deux entiers: ");
scanf("%d %d", &nbr1, &nbr2);
printf("PGCD de %d et %d = %d", nbr1, nbr2, pgcd(nbr1,nbr2));
 return 0;
 }
```

Exercice 2 recursive

```
#include <stdio.h>
int Rechercheoccurence(int tab[], int taille, int element,int position)
{
if(taille==position){
return -1;
```

```
}
if(tab[position] == element){
 return position;
 }
return Rechercheoccurence(tab,taille,element,position+1);
 }
int main()
{
int nbr,val,i;
int tab[30];
printf(" Entrez le nombre d'éléments dans le tableau: ");
 scanf("%d", &nbr);
printf(" Entrez les éléments du tableau: ");
for (i = 0; i < nbr; i++)
scanf("%d", &tab[i]);
printf(" Entrez la valeur : ");
scanf("%d", &val);
int trouve=Rechercheoccurence(tab,nbr,val,0);
if (trouve==-1) {printf("occurence %d non trouvee", val);}
else {printf("occurence %d trouvee", val);}
return 0;
}
```

8.3. TP N° 2

8.3.1 Fichiers

EXERCICE 1 :

Ecrire le programme C qui permet de compter le nombre de mots qui commencent par lalettre ‘A’ dans un fichier de caractères.

EXERCICE 2 :

Ecrire le programme C qui permet de construire un fichier G à partir des éléments d’unfichier de caractères f en remplaçant une séquence de blancs par un seul blanc.

EXERCICE 3 :

Soient F et G deux fichiers ordonnés par ordre croissant ; écrire le programme C quifusionne F et G en un fichier H ordonné.

EXERCICE 4 :

L’université désire faire la gestion de ses employés. Chaque employé est identifié par les informations suivantes :
(Num, Salaire, Nom, Prénom, Sexe, Département, Fonction « enseignant ou non »,)
Créer une structure correspondante. Ecrire un programme C de gestion de fichier avecmenu d'accueil: possibilité

1) de créer le fichier,
2) La saisie de 10 employés.
3) Afficher l’employé qui se trouve à la position 6 dans le fichier.
4) Compter le nombre d’enseignants de sexe féminin.

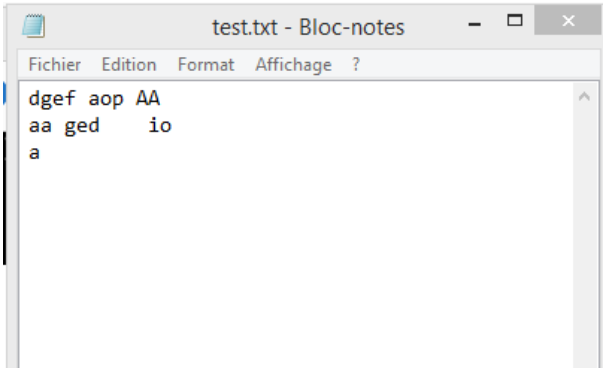
8.4. Solution TP N°2

EXERCICE 1 :

```
#include <stdio.h>
int main()
{
int n_mots=0 //nombre de mots dans le fichier mots.txt
char CHAINE[50];
FILE *fichier= fopen("test.txt","r"); //ouverture du fichier mots.txt en mode
lecture
if ( fichier == NULL )
      {
        printf ( "Could not open file test.txt" ) ;
        return 1;
      }
/* Compter le nombre de mots qui commencent par la lettre A*/
      while (!feof(fichier))              //feof permet de tester la fin d'un
fichier, on peu
       {
        fscanf(fichier, "%s\n", CHAINE);   // lit un texte formaté à partir
d'un fichier, les
        if (CHAINE[0]=='a' or CHAINE[0]=='A')
        {n_mots++; }
       }
       fclose(fichier);                     //fermeture du fichier à la fin du
traitement

       printf("Le nombre de mots qui commencent par la lettre (A/a) dans le
fichier est %d \n"

 }
```

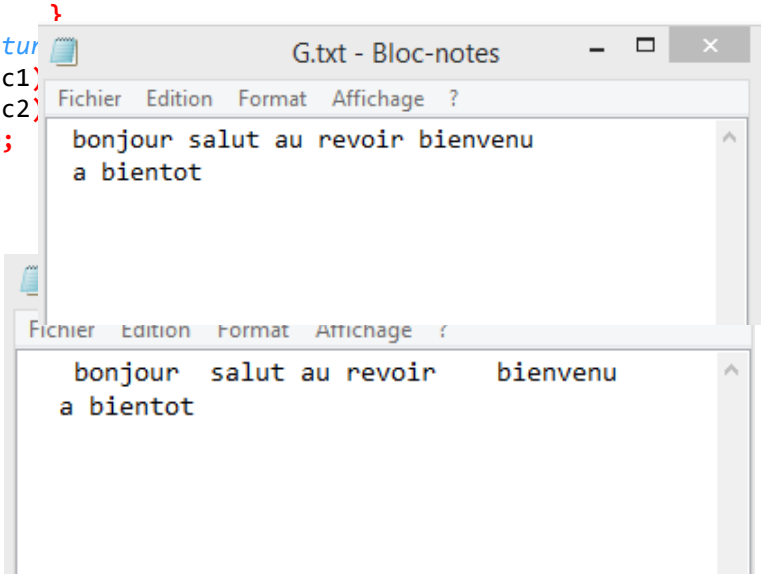



```
Le nombre de mots qui commencent par la lettre (A/a) dans le fichier est 4

--------------------------------
Process exited after 1.797 seconds with return value 0
Appuyez sur une touche pour continuer...
```

EXERCICE 2 :

```
#include <stdio.h>
int main()
 {
      char c; /* caractère lu dans le fichier */
      char n_space=0; /* Compteur des retours à la ligne consécutifs */
      FILE *fic1= fopen("F.txt","r");   //ouverture du fichier F.txt en mode lecture
       if ( fic1 == NULL )
       {
         printf ( "Could not open file F.txt" ) ;
         return 1;
       }
      FILE *fic2= fopen("G.txt","w");   //ouverture du fichier G.txt en mode ecriture
       if ( fic2 == NULL )
       {
         printf ( "Could not open file G.txt" ) ;
         return 1;
       }
       while((c=fgetc(fic1))!=EOF)    // je récupère dans lettre un caractére du
fichier1, EOF
        {
           if (c==' ')
           { n_space++;
             while((c=fgetc(fic1))==' ')
             { n_space++;
             }
             fputc(' ', fic2);
             fputc(c,fic2);
             n_space=0 ;
           }
           else {
fputc(c,fic2);}   }
       /* Fermetur
      fclose(fic1)
      fclose(fic2)
       return 0;
 }
```



EXERCICE 3 :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  // Ouvrir les deux fichiers à fusionner
  FILE *ff = fopen("F.txt", "r");
  FILE *fg = fopen("G.txt", "r");

  // Ouvrir le fichier pour stocker le résultat
  FILE *fh = fopen("H.txt", "w");
  char cf,cg;
   if (ff == NULL || fg == NULL || fh == NULL)
  {       puts("Impossible d'ouvrir les fichiers");
      exit(EXIT_FAILURE);   }
   // Copier le contenu du premier fichier dans file3.txt
  int pos_f=0,pos_g=0,pos_h=0;
  int nf,ng,nh;
 while (!feof(ff) && !feof(fg))
 { printf("position initiale pour le fichier F=%d\n",ftell(ff));
  //fseek(ff,pos_f,SEEK_SET); //dans le fichier F déplacer le curseur de pos_f position à partir du début
du fichier
   printf("position aprés fseek pour le fichier F=%d\n",ftell(ff));
   printf("position initiale pour le fichier G=%d\n",ftell(fg));
  // fseek(fg,pos_g,SEEK_SET); //dans le fichier G déplacer le curseur de pos_g position à partir du
début du fichier
   printf("position aprés fseek pour le fichier G=%d\n",ftell(fg));
     fscanf(ff, "%s", &nf); printf("nf=%d\n",nf);
   fscanf(fg, "%s", &ng); printf("ng=%d\n",ng);
   if (nf<ng) {fprintf(fh,"%d\n",nf);
            pos_f=ftell(ff); // retourner la position courante du curseur dans le fichier F
           }
        else  {fprintf(fh,"%d\n",ng);
             pos_g=ftell(fg); // retourner la position courante du curseur dans le fichier G
                 }  }
    /* Si F ou G sont arrivés à la fin du fichier */
  /* alors copier le reste de l'un des deux fichiers F ou G dans le fichier H  */
   if (feof(ff))
 {while (!feof(fg))
  {fscanf(fg, "%s",&ng);
   fprintf(fh,"%d\n",ng);
  } }
 else  if (feof(fg))
 {while (!feof(ff))
  {fscanf(ff, "%s",&nf);
   fprintf(fh,"%d\n",nf);
  } }
  fclose(ff);
  fclose(fg);
  fclose(fh);
  return 0; }
```

8.5. TP N° 3

EXERCICE 1 :

Soit une liste chaînée d'entiers

a) Ecrire la fonction qui permet de compter les valeurs positives, les valeurs nulles et les valeurs négatives.

b) Ecrire la fonction récursive C qui permet de chercher une valeur val.

c) Ecrire la fonction C qui permet calculer la somme des éléments d'une liste.

EXERCICE 2 :

Soit une liste chaînée de valeurs entières ordonnées. Ecrire la fonction quipermet d'insérer une valeur val dans cette liste.

EXERCICE 3 :

Soit une liste bidirectionnelle chaînée des réels, écrire une fonction C quipermet de supprimer toutes les valeurs négatives.

EXERCICE 4 :

Ecrire une fonction qui permet d'inverser une liste chaînée.

8.6. Solution TP N°3

Exercice 1

```
#include<stdio.h>
#include <conio.h>
#include <stdlib.h>

// définir un structure appelée boite qui va contenir l'élément et un poiteur qui pointe sur
l'élément suivant
typedef struct boite {
  int x;
  struct boite *suivant; //suivant est un poiteur qui va poiter sur une autre boite ou maillon
donc il est de type boite
}boite;

// définirune structure qui va contenir les valeurs positives, negatives et nulles
struct valeur{
  int pos;
  int neg;
  int nul;
};

//ou bien
//struct boite{
//  int x;
```

```
//  struct boite *suivant;
//};

//fonction initialiser la liste
boite *init()
{
  return NULL;
}
//fonction ajouter un elemement a la liste doit retourner le debut de liste et debut de liste est de type boite donc c'est un poiteur de type boite
boite *ajouter( boite *debut, int elem)
{
        boite *b; // il faut créer un pointeur appelé b pour réserver espace à ajouter car la résevation est dynamique
        b= (boite *) malloc(sizeof(boite));//appeler la focntion malloc qui va créer une boite (data+pointeur)
                                //malloc va retourner l'@ de la voite qu'il a crée (b: pointe sur l'élément crée)
        b->x=elem, //remplir la donnée de boite par elem
        b->suivant=debut;
        //return b; return b remplace les 2 instructions suivantes
        debut=b;// debut doit pointer sur b
        return debut; // retourner la nouvelle valeur de debut, on retourne tjrs le debut de liste car une liste est reconnue par son debut ou sa tete
}
//fonction afficher la liste
void afficher(boite *debut)
{
        boite *t; //t est un pointeur temporaire pour parcourir la liste
        t=debut; // initialisement on commence le parcours par le debut de liste
        if (debut == NULL)
           printf("La liste est vide");
        else
        {
                while (t!=NULL)
                { printf("%d-->",t->x); //afficher la donnée
                  t=t->suivant; // passer à l'éément suivant
                        }
                }
}
//somme des elements d'une liste
int somme(boite *debut)
{ int som = 0;
  boite *t;
  t=debut;
  while (t!=NULL)
  { som += t->x;
    t = t->suivant;
  }
  return som;
}
//les valeurs pos neg et nul
void valeurs(boite *debut,int *po,int *ne, int *nu)
{ *po=0;*ne=0;*nu=0;
  boite *t;
```

```
 t=debut;
 while (t!=NULL)
 {if (t->x>0)
     *po+=1;
     else if (t->x<0)
            *ne+=1;
           else  *nu+=1;
  t=t->suivant;
 }
}
//chercher une valeur
int chercher(boite *debut,int val)
{boite *t;
t=debut;
if (t==NULL) return -1;
else if (t->x==val) return 1;
    else return chercher(t->suivant,val);
}
//max d'une liste
int max(boite *debut)
{   boite *t;
   t=debut;
   if(t->suivant==NULL) return t->x;
   if(max(t->suivant) > t->x)
        return max (t->suivant);
   else  return t->x;
}
// fonction valeurs en utilisant structure
void values(boite *debut, valeur my_values)
{my_values.pos=0;
 my_values.neg=0;
 my_values.nul=0;
 boite *t;
 t=debut;
 while (t!=NULL)
 {if (t->x>0)
     my_values.pos+=1;
     else if (t->x<0)
            my_values.neg+=1;
           else  my_values.nul+=1;
  t=t->suivant;
 }

}
int main()
{
          //boite *liste=NULL;
          boite *liste; // je cree liste de type boite
          liste=init(); // initialiser la liste
          int n,i,nbr_elem,v;
          int pos,neg,nul;
          printf("Entrer le nombre elements liste: ");
          scanf("%d",&nbr_elem);
          for (i=0;i<nbr_elem;i++)
          { scanf("%d",&n);
```

```
            liste=ajouter(liste,n);
          }
          afficher(liste);
          getch();
          valeurs(liste,&pos,&neg,&nul);
          printf("\n Les valeurs positifs:%d\tLes valeurs negatives:%d\tLes valeurs
nulles:%d\t",pos,neg,nul);
          /* en utilisant une structure comme paramétre-->reste à résoudre
          valeur mes_valeurs;
          values(liste,mes_valeurs);
          printf("\n Les valeurs positifs:%d\tLes valeurs negatives:%d\tLes valeurs
nulles:%d\t",mes_valeurs.pos,mes_valeurs.neg,mes_valeurs.nul);*/
          getch();
          printf("\nEntrer la valeur a chercher:");
          scanf("%d",&v);
          if (chercher(liste,v)==1) printf("La valeur %d est trouvee",v);
             else printf("la valeur %d est non trouvee",v);
          getch();
          int s=somme(liste);
          printf("\nLa somme des elements de la liste = %d\n",s);
          //max d'une liste
          int mx=max(liste);
          printf("\nLe max des elements de la liste = %d\n",mx);
}
```

Exercice 2

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

// définir un structure appelée boite qui va contenir l'élément et un poiteur qui pointe sur
l'élément suivant
typedef struct boite {
  int x;
  struct boite *suivant; //suivant est un poiteur qui va poiter sur une autre boite ou maillon
donc il est de type boite
}boite;


//fonction initialiser la liste
boite *init()
{
  return NULL;
}
//fonction ajouter un elemement a la liste doit retourner le debut de liste et debut de liste est
de type boite donc c'est un poiteur de type boite
boite *ajouter( boite *debut, int elem)
{
          boite *b; // il faut créer un pointeur appelé b pour réserver espace à ajouter car la
résevation est dynamique
          b= (boite *) malloc(sizeof(boite));//appeler la foncion malloc qui va créer une boite
(data+pointeur)
                                    //malloc va retourner l'@ de la voite qu'il a crée (b: pointe sur
l'élément crée)
```

```
        b->x=elem, //remplir la donnée de boite par elem
        b->suivant=debut;
        //return b; return b remplace les 2 instructions suivantes
        debut=b;// debut doit pointer sur b
        return debut; // retourner la nouvelle valeur de debut, on retourne tjrs le debut de liste
car une liste est reconnue par son debut ou sa tete
}
//fonction inserer un element dans une liste ordonnee
boite *inserer(boite *debut, int v)
{
        boite *b, *t, *t_courant;
        b=(boite *)malloc(sizeof(boite));
        b->x=v;
        b->suivant=NULL;
        t=debut;
        if (t==NULL || t->x>v)   // inserer au debut de liste
          {b->suivant=t;
           debut=b;
          }
                else // inserer au milieur ou à la fin
        { while (t!=NULL && t->x<v)
          {t_courant=t;
           t=t->suivant;
          }
          b->suivant=t;
          t_courant->suivant=b;
   }
        return debut;
}
//fonction afficher la liste
void afficher(boite *debut)
{
        boite *t; //t est un pointeur temporaire pour parcourir la liste
        t=debut; // initialisement on commence le parcours par le debut de liste
        if (debut == NULL)
           printf("La liste est vide");
        else
        {
                while (t!=NULL)
                { printf("%d-->",t->x); //afficher la donnée
                  t=t->suivant; // passer à l'éément suivant
                        }
                }
}

int main()
{
        //boite *liste=NULL;
        boite *liste; // je cree liste de type boite
        liste=init(); // initialiser la liste
        int n,i,nbr_elem,val;
        printf("Entrer le nombre elements liste: ");
        scanf("%d",&nbr_elem);
        for (i=0;i<nbr_elem;i++)
        { scanf("%d",&n);
```

```
            liste=ajouter(liste,n);
           }
           afficher(liste);
           getch();
           printf("\nEntrer la valeur a inserer:");
           scanf("%d",&val);
           liste=inserer(liste,val);
           afficher(liste);
}
```

Exercice 3

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

// définir un structure appelée boite qui va contenir l'élément et un poiteur qui pointe sur l'élément
suivant
typedef struct boite {
  int x;
  struct boite *suivant; //suivant est un poiteur qui va poiter sur une autre boite ou maillon donc il est de
type boite
  struct boite *precedent;
}boite;

typedef boite *liste; // Liste est repérée par le pointeur début donc de type boite

//fonction initialiser la liste
boite *init()
{
  return NULL;
}

//fonction ajouter un elemement a la liste doublement chainée, le passage se fait par adresse
void ajouter( liste *debut, int elem)
{
           boite *b; // il faut créer un pointeur appelé b pour réserver espace à ajouter car la résevation est
dynamique
           b= (boite *) malloc(sizeof(boite));//appeler la fonction malloc qui va créer une boite
(data+pointeur)
                                    //malloc va retourner l'@ de la voite qu'il a crée (b: pointe sur l'élément
crée)
           b->suivant=b->precedent=NULL;//initialiser les pointeurs à NULL de préférence
           b->x=elem, //remplir la donnée de boite par elem
           b->suivant=*debut;   //return b; return b remplace les 2 instructions suivantes
             if (*debut!=NULL)   //si la liste n'est pas vide
                (*debut)->precedent=b;
           *debut=b;// debut doit pointer sur b
}
//fonction supprimer les valeurs négatives
void supprimer(liste *debut)
{
           boite *t,*temp;
           while (*debut!=NULL && (*debut)->x<0)   /*cas suppression au début*/
           {   t=*debut;
              if ((*debut)->suivant!=NULL)
                (*debut)->suivant->precedent=NULL;
              *debut=(*debut)->suivant;
              free(t);
   }
```

```
            t=*debut;
            while(t!=NULL)
              { if (t->x<0)
                 { t->precedent->suivant=t->suivant;
                  if (t->suivant!=NULL)      /*cas suppression au milieu*/
                    t->suivant->precedent=t->precedent;
                  temp=t;
                  t=t->suivant;
                  free(temp);
                        }
                      else t=t->suivant;
              }
}

//fonction afficher la liste doublement chaînée, elle reçoit comme paramétre le type liste
void afficher(liste debut)
{
        boite *t; //t est un pointeur temporaire pour parcourir la liste
        t=debut; // initialisement on commence le parcours par le debut de liste
        printf("Debut de liste: ");
        while (t!=NULL)
        { printf("%d-->",t->x); //afficher la donnée
          t=t->suivant; // passer à l'éément suivant
                  }
        printf(" :Fin de liste");
}
int main()
{       //boite *liste=NULL;
        boite *L; // je cree liste de type boite
        L=init(); // déclarer une liste vide au début
        int n,i,nbr_elem;
        printf("Entrer le nombre elements liste: ");
        scanf("%d",&nbr_elem);
        for (i=0;i<nbr_elem;i++)
        { scanf("%d",&n);
          ajouter(&L,n);
        }
        afficher(L);
        getch();
        printf("\n\nSuppression des elements negatifs en cours ....\n Veuillez taper une touche...\n\n");
        supprimer(&L);
        afficher(L);
}
```

9. BIBLIOGRAPHIE

Livres

[1] M. Divay, *Algorithmes et structures de données génériques*. Edition Dunod, 1999.
[2] J.-M. Steyaert, « Structure et complexité des algorithmes », 1984.
[3] S. Tollari et O. B. Fredj, « Algorithmique et structures de données », 2007.
[4] Amad Mourad, « Algorithmique et Structures de Données », 2016.

Web bibliographie

https://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9es
https://fr.wikibooks.org/wiki/Programmation_algorithmique