# 4. Les structures de données statiques

## 4.1 Tableaux à une dimension

### 4.1.1 Introduction

Imaginons que dans un programme, nous ayons besoin simultanément de 25 valeurs (par exemple, des notes pour calculer une moyenne). La solution consiste à déclarer 25 variables réelles, appelées par exemple $n_1, n_2, n_3, \ldots, n_{25}$ et la variable `moyenne` réelle.

`moyenne = (`$n_1+n_2+n_3+\ldots+n_{25}$`) / 25`

En programmation (exemple langage C) : l'ordinateur va réserver `25*4=100` octets pour les valeurs réelles des 25 variables et `25*4=100` octets pour les adresses de ces 25 variables.
La programmation nous permet de rassembler toutes ces variables en une seule : " la note numéro 1 ", " la note numéro 2 ", …, " la note numéro 25 ".
Un ensemble de valeurs portant ainsi le même nom de variable et repérées par un nombre, s'appelle un **tableau**, et le nombre qui sert à repérer chaque valeur s'appelle un **indice**.
Un tableau de taille `n` est une structure très simple constituée de `n` emplacements consécutifs en mémoire. Il est donc possible d'accéder à un élément d'un tableau en temps constant pourvu que l'on connaisse sa position (ou indice). Un tableau est donc une structure très simple et très efficace. Il n'est cependant pas possible de modifier la taille d'un tableau, ce qui est gênant pour un certain nombre d'algorithmes. On dit cette structure est statique. Le nombre d'éléments qu'elle contient ne peut pas varier.
Dans notre exemple, nous créerons donc un tableau appelé `Note[]`. Et chaque note individuelle sera désignée par : `Note[i]` (l'élément qui se trouve à la position `i`).
Pour déclarer un tableau il faut préciser le nombre et le type de valeurs qu'il contiendra.

```
Tableau Note[25] : réels
```

Cette déclaration réserve l'emplacement de 25 éléments de type `réels`. Chaque élément est repéré par son indice (position de l'élément dans le tableau). Dans la plus part des langages de programmation (en particulier en langage C), la première position porte le numéro `0`. Dans notre exemple, les indices vont de `0` à `24`. Le premier élément du tableau sera désigné par `Note[0]`, le deuxième par `Note[1]`, le dernier par `Note[24]`. L'utilisation de `Note[25]` déclanchera une erreur.

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera par exemple :

### 4.1.2 Exemple

Voici un programme qui comporte la déclaration d'un tableau de 25 réels (les notes d'une classe), on commence par effectuer la saisie des notes, et en suite on calcul la moyenne des 25 notes et on affiche la moyenne :

```
variables tableau Note[25],i,somme    : entier
                          moyenne     : réel
      début
            /* saisir les notes */
            pour i allant de 0 à 24 faire
                  ecrire("entrer une note :")
                  lire(Note[i])
            finpour
                  /* effectuer la moyenne des notes */
            somme ← 0
            pour i allant de 0 à 24 faire
                  somme ← somme + Note[i]
            finPour
            moyenne = somme / 25
                  /* affichage de la moyenne */
            ecrire("la moyenne des notes est :",moyenne)
      fin
```

**Exécution :**

```
entrer une note : 12
entrer une note : 10.5
entrer une note : 14
…     …     …     …
entrer une note : 08
la moyenne des notes est :11.75
```

**Une amélioration :**

```
Constante Max 200
variables tableau Note[Max],i,somme,n : entier
                              moyenne : réel
      début
            écrire("entrer le nombre de notes :")
            lire(n)
                  /* saisir les notes */
            ecrire("entrer les ",n," notes :")
            pour i allant de 0 à n-1 faire
                  lire(Note[i]
            finpour
                  /* effectuer la moyenne des notes */
            somme ← 0
            pour i allant de 0 à n-1 faire
                  somme ← somme + Note[i]
            finPour
            moyenne = somme / 25
                  /* affichage de la moyenne */
            ecrire("la moyenne des ",n," notes est :",moyenne)
      fin
```

**Remarque :**

- A la compilation la constante `Max` sera remplacée par le nombre `200`.
- Ici le nombre de note n'est pas fixé à `25`, mais il est seulement inférieur à `200`.

- Même si on donne à `n` la valeur `10`, l'ordinateur à réserver quand même la place pour `200` variables de types réels.
- La saisie des notes est aussi améliorée, puisque on rentre sur la même ligne toutes les notes. Attention : si `n=10`, il faut rentrer `10` valeurs réelles séparer par un espace, si vous rentrez moins de dix valeurs, l'ordinateur restera bloquer, et attend que vous rentrer les dix valeurs attendues. Si au contraire vous rentrez 11 valeurs au lieu de 10, l'ordinateur affectera les10 premières valeurs au tableau `Note` et gardera la 11-ième valeur pour une prochaine lecture, ce qui provoquera peut être une erreur !!

### 4.1.3 Les caractéristiques de l'indice d'un tableau

L'indice qui sert à parcourir les éléments d'un tableau peut être exprimé directement comme un nombre ou une variable.

La valeur d'un indice doit toujours :

- être égale au moins à 0 (dans le langage Pascal, le premier élément d'un tableau porte l'indice 1). Mais, nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en langage C. Donc attention, `Tab[2]` est le troisième élément du tableau `Tab` !
- être un nombre entier. Quel que soit le langage, l'élément `Tab[3,1416]` n'existe jamais.
- être inférieure ou égale au nombre d'éléments du tableau moins 1. Si le tableau Tab a été déclaré comme ayant 10 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Tab[10] déclenchera automatiquement une erreur.

**Remarques :**

1. La déclaration :
   ```
   tableau Tab[10] : entier
   ```
   créera un tableau `Tab` de 10 éléments, le plus petit indice étant 0 et le plus grand indice est 9.
2. Ne pas confondre l'**indice** d'un élément d'un tableau avec le **contenu** de cet élément. La première maison de la rue n'a pas forcément un habitant, et la dixième maison n'a pas dix habitants. En notation algorithmique, il n'y a aucun rapport entre `i` et `Tab[i]`.

## Exercice 21 :

1. Ecrivez un algorithme qui lit la taille `n` d'un tableau `T`, il saisi les `n` éléments du tableau `T`, il effectue la somme des `n` éléments du tableau et il affiche cette somme.

2. Ecrivez un algorithme qui lit la taille `n` de deux tableaux T1 et T2, il effectue la lecture de ces deux tableaux, ensuite il effectue la somme des tableaux T1 et T2 dans un tableau T et il affiche le tableau T.
   **Exemple** :
   pour `n=8` $T_1$`=[4,5,8,-2,5,6,0,-5]`, $T_2$`=[1,-5,-7,0,-1, 3,-8,9]`,
   le tableau T obtenu est : `T=[5,0,1,-2,4,9,-8,4]`,

## Exercice 22 :

1. Ecrivez un algorithme qui permet à l'utilisateur de saisir les notes d'une classe, ensuite il renvoie le nombre de ces notes supérieures à la moyenne de la classe.
2. Ecrivez un algorithme qui permet à l'utilisateur de saisir un tableau de taille n et d'afficher le plus grand et le plus petit élément du tableau.

## Exercice 23 :

Que produit l'algorithme suivant ?

```
Variable Tableau F[10], i  : entier
     début
          F[0]← 1
          F[1]← 1
          écrire(F[0],F[1])
          pour i allant de 2 à 10 faire
               F[i]← F[i-1]+F[i-2]
               écrire(F[i])
          finpour
     fin
```

### 4.1.4 Les tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments) pour être sûr que "ça rentre". Mais d'une part, on n'en sera jamais parfaitement sûr (si le nombre n des éléments du tableau dépasse 10 000, ça provoquera une erreur), d'autre part, en raison de l'immensité de la place mémoire réservée (la plupart du temps non utilisée), c'est un gâchis qui affectera la taille de notre algorithme, ainsi que sa rapidité.

Pour résoudre ce problème, on a la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments. Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction d'allocation : **`Allocation(nom,nombre,type)`**, Dans la quelle il faut préciser le **nom** du tableau, le **nombre** d'éléments et le **type** des éléments à allouer. Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**. Il ne faut pas oublier de libérer le tableau à la fin de son utilisation avec l'instruction : **`libère(nom).`**

**Exemple :** on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. L'algorithme sera :

```
variables tableau Note[], moyenne, somme   : réels
                                  i, n     : entiers
début
     écrire("entrer le nombre de notes à saisir : ")
     lire(n)
               /* allouer n nombres de types réels */
     allocation(Notes,n,réels)
               /* saisir les notes */
     écrire("entrer les ",n," notes :")
     pour i allant de 0 à n-1 faire
```

```
                lire(Note[i])
        finpour
                /* effectuer la moyenne des notes */
        somme ← 0
        pour i allant de 0 à n-1 faire
             somme ← somme + Note[i]
        finPour
        moyenne = somme / n
                /* affichage de la moyenne */
        écrire("la moyenne des ",n," notes est :",moyenne)
                /* libérer le tableau Notes */
        libère(Notes)
fin
```

**Exercice 24 :**
Refaire l'Exercice 21 précédent en utilisant des tableaux dynamiques.

## 4.2 Tableaux à deux dimensions

### 4.2.1 Introduction

Pour représenter par exemple les matrices dans un ordinateur, un tableau ne suffit pas, puisque chaque ligne de la matrice est en effet un tableau, donc une matrice n×k peut par exemple être représenter par n tableaux de k éléments chacun. Mais cette représentation sera difficile à gérer, surtout si on veut implémenter un algorithme qui effectue la multiplication ou l'addition de deux matrices. L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par un indice seule, mais par deux indices. C'est la solution à notre problème de représentation de matrice.
Un tel tableau se déclare ainsi :

```
tableau  matrice[10][10] : entier
```

Cette déclaration signifie : réserver un espace de mémoire pour 10 x 10 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices.
`matrice[i][j]` est l'élément de la matrice qui se trouve à l'intersection de la ligne `i` et la colonne `j`.

```
matrice[2][3]← 5
```

Cette instruction signifie : mettre à l'emplacement qui se trouve à l'intersection de la deuxième ligne avec la troisième colonne la valeur 5.
Dans la mémoire l'ordinateur représente un tableau matrice[10][10] par un seul tableau avec la taille 10×10.

### 4.2.2 Initialisation de matrice :

Pour initialiser une matrice on peut utiliser par exemple les instructions suivantes :

```
T1[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
T2[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
T3[4][4] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
T4[4][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Ces instructions initialisent quatre tableaux de la manière suivantes :

| $T_1$ | | | $T_2$ | | | $T_3$ | | | | $T_4$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 0 | 5 | 6 | 7 | 8 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 0 | 9 | 0 | 0 | 0 |
| | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Remarque** : Rappelons que ces représentations rectangulaires sont très conventionnelles. Dans la mémoire de l'ordinateur ces quatre tableaux sont plutôt arrangés en lignes :

$T_1$ → 1 2 3 4 5 6 7 8 9
$T_2$ → 1 2 3 4 5 6 7 8 9
$T_3$ → 1 2 3 0 4 5 6 0 7 8 9 0 0 0 0 0
$T_4$ → 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0

### 4.2.3 Lecture et écriture d'une matrice

**Lecture d'une matrice :**

```
variable Tableau matrice[10][10],i,j,n,k  : entiers
début
     écrire("donner le nombre de ligne de la matrice :")
     lire(n)
     écrire("donner le nombre de colonne de la matrice :")
     lire(k)
     pour i allant de 1 à n faire
          écrire("donner les éléments de la ",i," ligne:")
          pour j allant de 1 à k faire
               lire(matrice[i][j])
          finpour
     fipour
fin
```

**Ecriture d'une matrice :**

```
variables Tableau Mat[10][10],i,j,n,k : entier
     début
          pour i allant de 1 à n faire
               pour j allant de 1 à k faire
                    écrire(Mat[i][j],"  ")
               finpour
               écrire("\n")    /* retour à la ligne */
          finpour
     fin
```

### 4.2.4 Exemples d'utilisation d'une matrice

**Somme de deux matrices :**

```
constante N 20
variables Tableau A[N][N],B[N][N],C[N][N],i,j,n : entier
     début
          écrire("donner la taille des matrices(<20) :")
```

```
          lire(n)
                         /* lecture de la matrice A */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(A[i][j])
               finpour
          fipour
                         /* lecture de la matrice B */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(B[i][j])
               finpour
          fipour
                         /* la somme de C = A + B */
          pour i allant de 1 à n faire
               pour j allant de 1 à n faire
                    C[i][j] ← A[i][j]+B[i][j]
               finpour
          fipour
                         /* affichage de la matrice de C */
          pour i allant de 1 à n faire
               pour j allant de 1 à n faire
                    écrire(C[i][j],"  ")
               finpour
               écrire("\n")    /* retour à la ligne */
          finpour
     fin
```

**Produit de deux matrices :**

```
constante N 20
variables Tableau A[N][N],B[N][N],C[N][N],i,j,k,n,S : entier
     début
          écrire("donner la taille des matrices(<20) :")
          lire(n)
                         /* lecture de la matrice A */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(A[i][j])
               finpour
          fipour
                         /* lecture de la matrice B */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(B[i][j])
               finpour
          fipour
                         /* le produit de C = A * B */
```

```
        pour i allant de 1 à n faire
             pour j allant de 1 à n faire
                  S ← 0
                  pour k allant de 1 à n faire
                       S ← S + A[i][k]*B[k][j]
                  finpour
                  C[i][j] ← S
             finpour
        fipour
                       /* affichage de la matrice de C */
        pour i allant de 1 à n faire
             pour j allant de 1 à n faire
                  écrire(C[i][j],"  ")
             finpour
             écrire("\n")    /* retour à la ligne */
        finpour
fin
```

## Exercice 25 :

1. Ecrire un algorithme qui effectue la lecture d'une matrice carrée A ainsi que sa taille n et affiche la trace de A (pour une matrice $A(a_{i,j})$, $Trace(A)=\sum a_{i,i}$ la somme des éléments sur la diagonale).
2. Ecrire un algorithme qui effectue la lecture d'une matrice carrée A ainsi que sa taille n et affiche la matrice transposé tA de A (Pour une matrice $A(a_{i,j})$, $tA(a_{j,i})$ ).

## Exercice 26 :

1. Écrivez un algorithme qui effectue la lecture de :
   - n un entier.
   - vect[] un tableau de n nombre réels,
   - mat[][] une matrice carrée de n×n nombre réels,

   il calcule et affiche le produit de la matrice mat par le vecteur vect.
2. Écrivez un algorithme qui effectue la lecture de deux matrices allouées dynamiquement et affiche le produit de ces deux matrices.

# 5. Les fonctions

L'analyse descendante consiste à décomposer le problème donné en sous problèmes, et ainsi de suite, jusqu'à descendre au niveau des primitives. Les étapes successives donnent lieu à des **sous-programmes** qui peuvent être considérés comme les primitives de machine intermédiaires, c'est les **fonctions** en langage C.

Beaucoup de langages distinguent deux sortes de sous-programmes : les *fonctions* et les *procédures*. L'appel d'une fonction est une expression, tandis que l'appel d'une procédure est une instruction. Où, si on préfère, l'appel d'une fonction renvoie un résultat, alors que l'appel d'une procédure ne renvoie rien.
En C on retrouve ces deux manières d'appeler les sous-programmes, mais du point de la syntaxe le langage ne connaît que les fonctions. Autrement dit, un sous-programme est toujours supposé renvoyer une valeur, même lorsque celle-ci n'a pas de sens ou n'a pas été spécifiée (`void`).

La réalisation d'un programme passe par l'analyse descendante du problème : il faut réussir à trouver les actions élémentaires qui, en partant d'un environnement initial, nous conduisent à l'état final.

## 5.1 Déclaration des fonctions

Une fonction se définit par la construction :

```
fonction NomFonction(NomArg1 :TypeArg1, …, NomArgk:TypeArgk) : TypeRetour
déclarations des variables
début
     Bloc d'instructions
fin
```

Une fonction est connue dans le programme par son nom (`NomFonction` ) et il est nécessaire de connaître les éléments qui vont permettre de l'utiliser. Comme pour les variables il faut la déclarer.
Cette déclaration se fera au moyen :

- Du type du résultat (`TypeRetour`), éventuellement vide si la fonction ne rend pas de résultat.
- Du nom de la fonction (`NomFonction`), identificateur standard.
- Entre parenthèses, la liste des paramètres d'appel (`NomArg`$_i$) précédés de leur type (`TypeArg`$_i$) et séparés par une virgule. Si la liste est vide, on peut ne rien.
  Elle sera suivie des déclarations des variables et du corps de la fonction qui est constitué par une suite d'instructions, éventuellement une seule, mises entre début et fin.

**Exemple** :

```
fonction  somme( x : entier, y : entier) : entier
debut
     retourne(x+y)
fin
```

Cette fonction admet deux paramètres de type entiers et le résultat est de même type.
La fonction `somme()` peut être utilisée par une autre fonction de la façon suivante :

```
variables a, b, c, d : entiers
a ← 7
b ← -2
c = somme(a,b)
d = somme(a,somme(a,c))
écrire(" la somme est = ",somme(a,b))
```

Contrairement à d'autres langages, en C on ne peut pas définir une fonction à l'intérieur d'une autre : toutes les fonctions sont au même niveau, c'est-à-dire globales, on va adapter ici cette réstriction.

**Type de la fonction et des arguments :**
L'en-tête de la fonction définit le type des objets qu'elle renvoie. Ce peut être :
– tout type numérique ;
– tout type pointeur ;
– tout type struct ou union.

Si le type de la fonction n'est pas indiqué, le compilateur suppose qu'il s'agit d'une fonction qui ne retourne rien. Lorsqu'une fonction ne renvoie pas une valeur, c'est-à-dire lorsqu'elle correspond plus à une procédure qu'à une vraie fonction, il est prudent de la déclarer comme rendant un objet de type `vide`. Ce type est garanti *incompatible avec tous les autres types* : une tentative d'utilisation du résultat de la fonction provoquera donc une erreur à la compilation.

## 5.2 Exemple complet

Calculer la somme des puissances p-ième des entiers $S_p(n)$ :

$$S_p(n) = 1^p + 2^p + 3^p + \ldots + n^p$$

On sait que :

$$S_1(n) = 1 + 2 + 3 + \ldots + n = n(n+1)/2$$
$$S_2(n) = 1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n+1)(2n+1)/2$$
$$S_3(n) = 1^3 + 2^3 + 3^3 + \ldots + n^3 = n^2(n+1)^2/4$$

Ce calcul doit être effectué n fois pour des valeurs de p qui pourront varier : on va fabriquer un outil, une fonction `puissance(x,p)`, auquel l'on donnera deux entiers x et p et qui restituera le nombre $x^p$.

La fonction **`puissance(x,p)`** : $\mathbf{x^p}$

**Analyse** :

L'algorithme utilisé est simple et utilise la propriété de récurrence :

$$\forall p \in N^* \qquad x^p = x * x^{p-1}$$

**Conception** :

```
z est un entier
z ← 1
tant que p>0 faire
     z ← z*x
```

```
        p ← p-1
    fintantque
    retourne(z)
```

Ces instructions on les regroupant dans une fonction (`puissance()`) de la manière suivante :

```
    puissance( x : entier, p :entier) : entier
    variables z : entier
    debut
        z ← 1
        tant que (p>0) faire
            z ← z*x
            p ← p-1
        fintantque
        retourne(z)
    fin
```

La dernière instruction, `retourne(z)` est le moyen d'indiquer que le résultat de la fonction est `z`. On peut maintenant écrire le programme qui fait la somme des puissances `p`-ième des entiers compris entre `1` et `n` (le calcul de $S_p(n)$) :

```
                /* la fonction qui calcule x^p */
puissance( x : entier, p :entier) : entier
    variables z : entier
    debut
        z ← 1
        tant que (p>0) faire
            z ← z*x
            p ← p-1
        fintantque
        retourne(z)
    fin
fonction menu ()
variables p, n, s, i : entiers
début
    i ← 0
    écrire(" entrer la puissance p et l'entier n :")
    lire(p,n)
    pour i allant de 1 à n faire
        s = s + puissance(i,p)
                /* les variables i et p sont des paramètres réels, la fonction
                puissance(i,p)est appelée pour effectuer le calcul de i^p */
    finpour
    écrire(" La somme est : ",s)
fin
```

**Exécution**

```
    entrer la puissance p et l'entier n :1 10
    La somme est : 55
    ============================================
    entrer la puissance p et l'entier n :2 10
```

```
La somme est : 1155
==========================================
entrer la puissance p et l'entier n :3 10
La somme est : 3025
```

La fonction peut être mise n'importe où dans le programme mais, si l'on veut que le compilateur puisse vérifier la correspondance entre paramètres formels et paramètres réels (nombre et type en particulier), il faut que la fonction soit placée avant son utilisation.

**Prototype** : pour éviter cette restriction (et permettre aussi que deux fonctions puissent s'appeler mutuellement), on peut utiliser un prototype de fonction.
Un prototype est constitué par la partie déclaration de la fonction, on la place en général au début du programme.
On pourra écrire, par exemple :

```
                /* prototypes */
fonction f(entier,entier):entier
fonction g(chaîne de caractère, entier) ;

                /* corps de la fonction f */
fonction  f(x:entier, y:entier)
var message : chaîne de caractère
début
          … … …
     g(message,y) ;
          … … …
Fin
                /* corps de la fonction g */
fonction  g(m :chaîne de caractère, y :entier)
var a,b,c : entier
début
          … … …
          c ← f(a,b);
          … … …
fin
```

L'utilisation des prototypes n'est pas obligatoire mais est fortement recommandé pour un meilleur contrôle des paramètres et pour une meilleure lisibilité du programme : les outils utilisés sont listés au début du texte.

**Résultat d'une fonction**
Pour retourner le résultat d'une fonction on utilise l'instruction :

```
retourne(valeur) ;
```

où `valeur` doit être du type prévu dans l'entête de la fonction.

Soit `max()` une fonction qui donne le plus grand de deux nombres entiers `x` et `y`, elle peut s'écrire :

```
fonction max ( x : entier, y : entier) : entier
          /* maximum de x et y */
     si (x<y) retourne(y)
```

```
        sinon retourne(x)
    }
```

L'instruction **retourne** provoque la sortie immédiate de la fonction et le retour dans le programme appelant, les instructions qui suivent dans la fonction ne sont pas exécutées, donc on peut ne pas mettre "sinon".

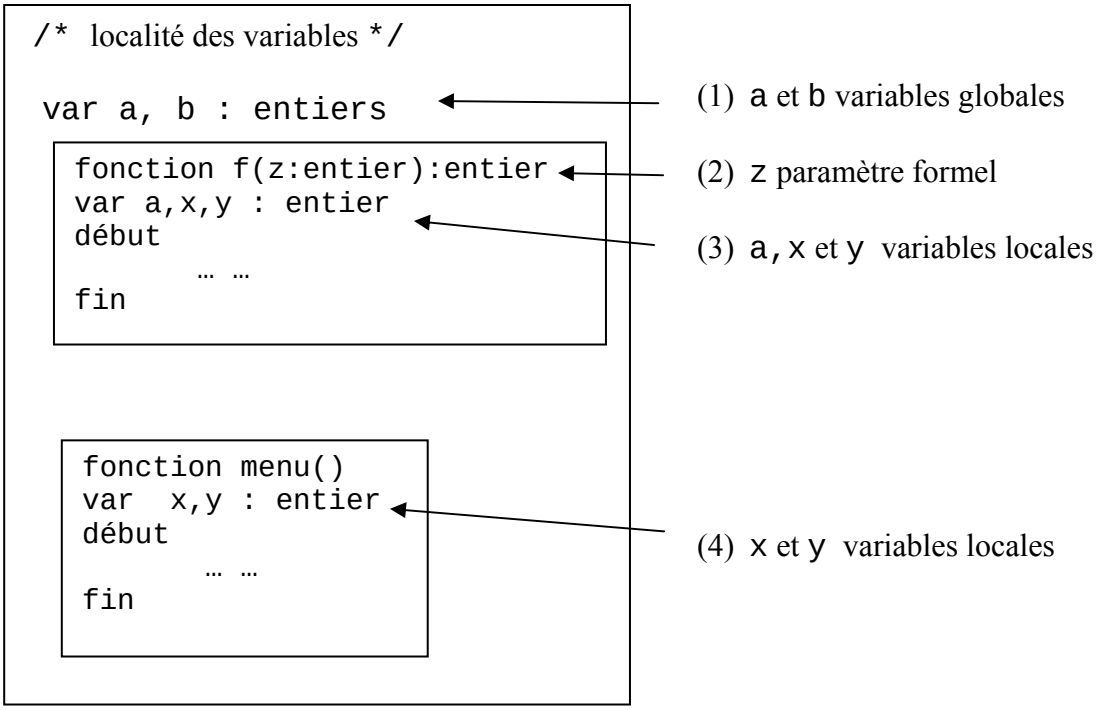
```
    fonction max ( x : entier, y : entier) : entier
              /* maximum de x et y */
        si (x<y) retourne(y)
        retourne(x)
    }
```

## 5.3 Variables locales et globales

Il peut y avoir différents niveaux de déclaration de variables.

- Les variables définies dans le programme hors des fonctions sont dites **variables globales**. Les variables globales sont connues de tout le programme.
- Les variables déclarées dans le corps d'une fonction sont dites **variables locales** ou **variables automatiques**. Les variables locales sont connues uniquement à l'intérieur de la fonction, c'est-à-dire qu'elles sont propres à la fonction et invisibles de l'extérieur. Cela permet d'utiliser des variables de même nom dans des fonctions différentes sans craintes d'interférences fâcheuses et "d'oublier" le corps de la fonction pour n'en retenir que l'essentiel, son mode d'emploi, qui figure dans sa déclaration : son nom, ses arguments et leur type, le type du résultat.

**Exemple** :



Les variables a et b, (1) sont des variables globales, elles seront connues et utilisables dans toutes les parties du programme.
La fonction menu() ne connaît pas le nom des fonctions et le nombre et le type de leurs paramètres d'appel. La fonction menu() ne pourra travailler que sur les variables globales a et b, et appeler la fonction f().
Une fonction connaît ses propres variables, les paramètres formels, les variables globales dont le nom n'est pas redéfini dans la fonction, toutes les fonctions définies avant elle et, enfin, elle se connaît elle-même.
La fonction f() pourra travailler avec les variables locales a, x, y, z et la variable globale b. La variable globale a est inaccessible de l'intérieur de f(), toute utilisation de a fait référence à la variable local. La fonction f() pourra s'appeler elle-même.
La fonction menu() pourra travailler sur les variables locales x, y et les variables globales a et b la fonction menu() pourra appeler f().

## 5.4 Les paramètres d'appels

Il y a deux types de passage de paramètres, le passage par **valeur** et le passage par **adresse**.

**Passage par valeur :**
Dans un appel par valeur le paramètre d'appel est considéré comme une variable locale, toutes les modifications se feront dans une case mémoire temporaire dans laquelle est rangée la valeur du paramètre d'appel. En particulier, on peut appeler la fonction avec une expression qui sera évaluée et dont la valeur sera le paramètre réel.

**Exemple** : Soit le programme suivant :

```
fonction menu ()
var a,b,c : entier          /* déclaration de a et b*/
début
     écrire("entrer les valeurs de a et b :")
     lire(a,b)
     écrire(" avant l'échange a = ",a," et b = ",b)
     c ← a
     a ← b           /* l'échange   */
     b ← c
     écrire(" après l'échange a = ",a," et b = ",b)
fin
```

L'exécution :

```
entrer les valeurs de a et b : 5  7
avant l'échange a=5 et b=7
après l'échange a=7 et b=5
```

Ce programme effectue l'échange des valeurs de a et b. On désire maintenant écrire une fonction echange() qui prend deux paramètres x et y et effectue l'échange des valeurs de x et y (elle ne retourne rien c'est une procédure).

**Passage par valeur** :

```
fonction echange(x:entier, y:entier)   /* x,y:paramètre formel */
var tmp :entier                        /* tmp variable locale */
début
     tmp ← x
     x ← y                /* l'échange  */
     y ← tmp
     écrire("dans la fonction echange(): x = ",x," et y = ",y)
fin

fonction menu()
var a,b : entier          /* déclaration de a et b*/
début
     écrire("entrer les valeurs de a et b :")
     lire(a,b)
     écrire(" avant l'échange a = ",a," et b = ",b)
     echange(a,b)         /* appel de echange()  */
     écrire(" après l'échange a = ",a," et b = ",b)
fin
```

**Exécution :**
```
entrer les valeurs de a et b : 5  7
avant l'échange a=5 et b=7
dans la fonction echange() : x=7 et y=5
après l'échange a=5 et b=7
```
**L'échange n'a pas eu lieu !**

**Passage par adresse**

Si on veut modifier une variable à l'intérieur d'une fonction et si l'on désire que cette modification soit effective dans la fonction appelante on fera un passage de paramètre par **adresse**.
En C tous les arguments des fonctions sont des appels par valeur. Le seul moyen de modifier une variable externe à une fonction est indirect : on transmet son adresse. L'argument de la fonction doit donc être une adresse c'est-à-dire un **pointeur**.

**Exemple** : passage par adresse

```
fonction echange( x:pointeur entier, y:pointeur entier)
var tmp :entier
début
     tmp ← contenu(x)
     contenu(x) ← contenu(y)              /* l'échange  */
     contenu(y) ← tmp
     écrire("dans la fonction echange(): x = ",contenu(x),"
          et y = ",contenu(y))
fin

fonction menu()
var a,b : entier              /* déclaration de a et b*/
début
     écrire("entrer les valeurs de a et b :")
```

```
        lire(a,b)
        écrire(" avant l'échange a = ",a," et b = ",b)
        echange(adr(a),adr(b)) /*on appel echange() avec les adresses de a et b*/
        écrire(" après l'échange a = ",a," et b = ",b)
fin
```

**Exécution:**
```
entrer les valeurs de a et b : 5  7
avant l'échange a=5 et b=7
dans la fonction echange() : x=7 et y=5
après l'échange a=7 et b=5
```
**C'est bien la solution souhaitée.**

**Commentaires sur le programme** :

Dans la fonction echange() : - les deux paramètres formels sont des pointeur sur des entiers (des adresses).
- le `contenu(x)` est la valeur qui se trouve à l'adresse x (puisque `x` est une adresse).
- l'instruction `tmp=contenu(x)` signifie mettre dans la variable `tmp` (de type `entier`) la valeur de la variable qui se trouve à l'adresse `x`.

Dans la fonction menu() : - l'instruction `echange(adr(a),adr(b))` signifie qu'on appelle la fonction `echange()` en lui transmettant les adresses des variables `x` et `y`.

## 5.5 Exemples

Tous les algorithmes vus dans les chapitres 3 et 4 peuvent être réécrit sous la forme de fonction :

1. La fonction suivante prend pour argument un entier `n` et retourne le nombre de chiffres qui compose `n` :
   Exemple : si `n=1452635` la fonction retourne la valeur `7`.

```
fonction NombreDeChiffres(n :entier) :entier
var i, nb : entier
début
      nb ← 0
      tant que (n<>0) faire
            n ← n/10
            nb ← nb+1
      fintantque
      retourne(nb)
fin
```

2. La fonction suivante prend pour argument un tableau `Note[]` (qui contient les notes d'une classe) et `n` la taille du tableau (le nombre des notes) et elle retourne la moyenne de la classe :

```
Constante Max 200
fonction MoyenneClasse(tableau Note[Max]:réel, n:entier) :réel
variables  i : entier
```

```
            somme, moyenne : réel
début
      somme ← 0
      pour i allant de 0 à n-1 faire
            somme ← somme + Note[i]
      finPour
      moyenne ← somme / n
      retourne(moyenne)
fin
```

**3.** La fonction suivante prend pour argument deux entiers a et b et retourne leur pgcd :

```
fonction pgcd(a :entier, b :entier) : entier
variables r : entier
début
      tantQue b>0 faire
            r ← a%b
            a ← b
            b ← r
      finTanQue
      retourne(a)
fin
```

**4.** On veut écrire une fonction qui prend pour argument un tableau `T[]` d'entiers, sa taille `n` et on veut qu'elle retourne le plus grand élément du tableau ainsi que sa position. Comme une fonction ne peut retourner qu'une valeur, alors on va passer les deux variables comme arguments de la fonction mais avec un passage par adresse pour récupérer les valeurs de deux variables après l'appelle de la fonction :

```
fonction PlusGrand(tableau T[Max]:entier,n:entier,
                        pg:pointeur entier,pos: pointeur entier)
var  i : entier
début
      contenu(pg) ← T[0]
      contenu(pos) ← 0
      pour i allant de 1 à n-1 faire
            si (T[i]>contenu(pos)) alors
                  contenu(pg) ← T[i]
                  contenu(pos) ← i
            finsi
      finpour
 fin
```

La fonction `PlusGrand()` peut être appelée de la façon suivante :

si `A` est un tableau d'entier de taille `n` et `max` et `ind` deux entiers

`PlusGrand(A,n,adr max, adr ind)`

les deux variables `max` et `ind` contiennent alors respectivement la plus grande valeur du tableau et sa position.

**Remarque :** dans cet exemple on pouvait faire retourner une variable par la fonction, mais la deuxième devrait obligatoirement passer par adresse :

```
fonction PlusGrand2(tableau T[Max]:entier,n:entier,
                            pos: pointeur entier) : entier
var  i, pg : entier
début
     pg ← T[0]
     contenu(pos) ← 0
     pour i allant de 1 à n-1 faire
          si (T[i]>contenu(pos)) alors
               pg ← T[i]
               contenu(pos) ← i
          finsi
     finpour
     retourne(pg)
 fin
```

La fonction `PlusGrand2()` peut être appelée de la façon suivante :

si `A` est un tableau d'entier de taille `n` et `max` et `ind` deux entiers

`max=PlusGrand2(A,n,adr ind)`

et les deux variables `max` et `ind` contiennent alors respectivement la plus grande valeur du tableau et sa position.

**Remarque** : On peut aussi déclarer les deux variables `max` et `ind` comme variables globales, c'est la plus mauvaise solution.

## 5.6 Fonctions récursives

### 5.6.1 Introduction

Il arrive, en mathématique, que des suites soient définies de la manière suivante :

$u_0$ = constante

$u_n = f(u_{n-1})$

Exemple : La suite factorielle : $n! = n \times (n-1)!$, pour $n \geq 1$ avec $0!=1$, peut s'écrire :

```
f(0)=1
f(n)=n*f(n-1)
```

Ce que l'on peut traduire par : f(n) = (si n=0 alors 1 sinon  n*f(n-1)).
Cela peut se traduire en algorithmique par :

```
fonction factorielle1(n :entier) : entier
début
     si (n=0) alors retourne(1)
     sinon retourne(n*factorielle1(n-1))
     finsi
fin
```

Dans la fonction `factorielle1()`, on constate que la fonction s'appelle elle-même. Ceci est possible, puisque la fonction `factorielle1()` est déclarée avant son utilisation (c'est l'en-

tête d'une fonction qui la déclare).
Que se passe-t-il lorsque on calcul `factorielle1(3)`?

```
factorielle1(3)     =  3 * factorielle1(2)
                    =  3 * (2*factorielle1(1)
                    =  3 * (2 * (1*factorielle1(0)))
                    =  3 * (2 * (1 * (1)))
                    =  3 * (2 * (1))
                    =  3 * (2)
                    =  6
```

## 5.6.2 Définition de la récursivité

La récursivité est un concept fondamental en mathématiques et en informatique. La définition la plus simple que l'on puisse en donner consiste à dire qu'un programme récursif est un programme qui s'appelle lui-même. Pourtant, il faut bien qu'un programme cesse de s'appeler lui-même si l'on veut éviter la boucle infinie. Un programme récursif doit contenir une condition de terminaison qui autorise le programme à ne plus faire appel à lui-même.
Les définitions récursives de fonctions sont fréquentes en mathématiques ; le type le plus simple, portant sur des arguments entiers, est la relation de récurrence. La fonction la plus familière de ce type est sans doute la fonction factorielle vue plus haut.
La fonction `factorielle1()` illustre les caractéristiques élémentaires de tout programme récursif (elle s'appelle elle-même, avec une valeur inférieure de l'argument et elle contient une condition de terminaison dans laquelle elle calcule directement le résultat), mais on ne peut cacher qu'il n'est rien d'autre qu'une boucle "`pour`". Cet exemple ne démontre pas vraiment la puissance d'un programme récursif.

```
fonction factorielle2(n : entier) : entier
variables fact, i : entier
début
     fact ← 1
     pour i allant de 2 à n faire
          fact ← fact * i
     finpour
     retourne(fact)
fin
```

## 5.6.3 La suite de Fibonacci

Une deuxième relation de récurrence bien connue est celle qui définit la suite de Fibonacci :

$F_n = F_{n-1} + F_{n-2}$, pour $n \geq 2$ avec $F_0=0$ et $F_1=1$. (1)

**1er algorithme de calcul de $F_n$** :

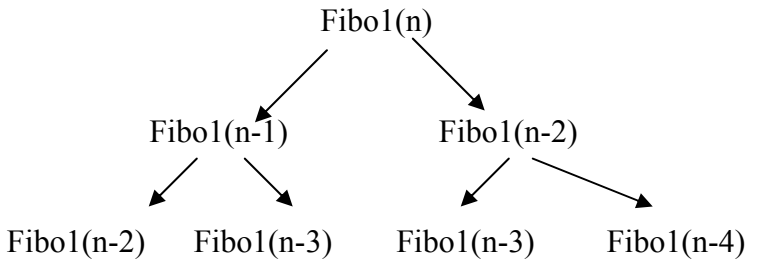
On a un programme récursif simple associé à cette relation :

```
fonction Fibo1(n entier) : entier
début
     si (n<=1) alors retourne(1)
```

```
        sinon retourne(Fibo1(n-1) + Fibo1(n-2))
        finsi
fin
```

Ce programme est récursif puisqu'il s'appelle lui-même, avec des valeurs inférieures de l'argument et il contient une condition de terminaison.

Il s'agit d'un mauvais exemple d'utilisation de la récursivité, puisque pour calculer `Fibo1(n)`, on a besoin de calculer `Fibo1(n-1)` et `Fibo1(n-2)` ; voir l'arbre suivant :




… … … … ... … … …

Le nombre d'appels nécessaires au calcul de $F_n$ est égal au nombre d'appels nécessaires au calcul de $F_{n-1}$ plus celui relatif au calcul de $F_{n-2}$, ceci correspond bien à la définition de la suite de Fibonacci : le nombre d'appels de la fonction `Fibo1()` pour calculer $F_n$ est exactement le nombre $F_n$. Or $F_n$ est **exponentielle en fonction de n**.
La solution de la suite récurrente (1) est :

$$F_n = \frac{\alpha^n - \overline{\alpha^n}}{\sqrt{5}} \quad avec \quad \alpha = \frac{1+\sqrt{5}}{2} \quad et \quad \overline{\alpha} = \frac{1-\sqrt{5}}{2} \qquad (2)$$

où $\alpha$ = le nombre d'or $\cong 1.618\ldots$
On a aussi $F_{n+1} \,/\, F_n \rightarrow \alpha$

Donc pour calculer `Fibo1(n)` on fait $F_n$ appelle à la fonction `Fibo1(n)`, ce n'est pas pratique du tout (inutilisable).

**2ème algorithme de calcul de $F_n$** :

En revanche, il est très facile de calculer $F_n$ en utilisant un tableau :

```
constante : N = 26
fonction Fibo2(entier :n) : entier
variable i: entier ; tableau F[N] : entier
début
      F[0] ← 1
      F[1] ← 1
      pour i allant de 2 à n faire
            F[i] ← F[i-1] + F[i-2]
      finpour
      retourne(F[n])
fin
```

Après exécution pour `n=25`, on regroupe dans le tableau suivant les 26 premier termes de la suite de Fibonacci :

| *n* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_n$ | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 | 987 | 1597 |

| *n* | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|
| $F_n$ | 987 | 1597 | 2584 | 4181 | 6765 | 10946 |

| *n* | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|
| $F_n$ | 2584 | 4181 | 6765 | 10946 | 17711 | 28657 | 46368 | 75025 | 121393 |

$F_{50}$ = 20365011074

$F_{100}$ = 354224848179262000000

$F_{500}$ = un nombre de 104 chiffres

Dans ce cas, pour calculer $F_n$ on stocke tous les termes $F_0$, $F_1$, …, $F_{n-2}$ et $F_{n-1}$. Ce qui est inutile puisque pour calculer $F_n$ on a besoin que des deux derniers termes $F_{n-2}$ et $F_{n-1}$. Donc l'algorithme peut être amélioré encore !

## 5.7 Exercices

### Exercice 27 :

1. Écrivez une fonction qui prend pour argument un tableau d'entiers T, sa taille n, une valeur x, et il indique ensuite si l'élément x appartient ou non au tableau T.

2. Écrivez une fonction qui prend pour argument un tableau d'entiers T et sa taille n, et nous informe si les éléments d'un tableau d'entiers sont tous consécutifs ou non. (Par exemple, si le tableau est : 7; 8; 9; 10, ses éléments sont tous consécutifs. Si le tableau est : 7; 9 ; 10; 11, ses éléments ne sont pas tous consécutifs).

### Exercice 28 :

Écrivez une fonction qui prend pour arguments un tableau de réels T et sa taille n, et nous donne la plus grande et la plus petite valeur du tableau T.

### Exercice 29 (un nombre parfait) :

Un nombre **parfait** est un entier positif supérieur à 1, égal à la somme de ses diviseurs ; on compte 1 comme diviseur, mais on ne compte pas comme diviseur le nombre lui-même.
Exemple : 6 est un nombre parfait puisque : 6 = 3 + 2 + 1.

1. Donner un nombre parfait différent de 6.
2. Donner la conception d'un algorithme qui effectue la lecture d'un entier `n` et affiche si `n` est parfait ou non.
3. Ecrire une fonction qui prend pour argument un nombre entier `n` et retourne si `n` est parfait ou non.

## Exercice 30 :

Soit la fonction mystere() suivante :

```
Fonction mystere(x :entier, p :entier) :entier
variable z :entier
début
     z ← 1
     Tant que  p>0  faire
          si (p%2==0) alors
               x ← x*x
               p ←  p/2
          sinon
               z ← z*x
               p ← p-1
          finsi
     fin tantque
     retourne(z)
fin
```

1. Exécuter cette fonction avec :
   a. x=2 et p=16
   b. x=3 et p=31
   c. x=a et p=30
2. Que fait la fonction `mystere(x, p)` ? (justifier votre réponse).

## Exercice 31 :

On note par **`pgcd(a,b)`** le **p**lus **g**rand **c**ommun **d**iviseur de a et b.

Soit `a` et `b` deux entiers naturels, si `b` est non nul, on peut effectuer la division euclidienne de `a` par `b`. Il existe un couple unique d'entiers `(q,r)` tels que :

$a = bq + r$ avec $0 \leq r < b$

On a la formule suivante

$$pgcd(a,b) = pgcd(b,r)$$

A partir de cette formule déduisez une fonction `pgcdRec(a,b)` récursive qui calcul le pgcd de a et b.

## Exercice 32 :

1. Ecrire une procédure `afficher_matrice()` : qui prend pour arguments `n` et une matrice A, et elle affiche la matrice A.
2. Ecrire une procédure qui donne la transposée d'une matrice A.
3. Ecrire une procédure qui vérifie si une matrice est symétrique.
4. Ecrire une procédure qui effectue la somme de deux matrices.
5. Ecrire une procédure qui effectue le produit de deux matrices.
6. Ecrire une fonction qui donne le déterminant d'une matrice carrée d'ordre n=2 ou 3.
7. Ecrire une procédure qui donne l'inverse d'une matrice carrée d'ordre n=2 ou 3.

## Exercice 33 :

**I**. Recherche d'un élément dans un tableau.

1. Ecrire une procédure qui recherche un élément dans un tableau de façon séquentiel.
2. On se place dans le cas où le tableau est ordonnée et les éléments du tableau sont deux à deux distincts. Ecrire une procédure qui recherche un élément de façon dichotomique.

**II**. Insertion et suppression d'un élément d'un tableau.

1. Ecrire une procédure qui permet d'insérer un élément dans un tableau à une position donnée.
2. Ecrire une fonction qui permet d'insérer un élément dans un tableau déjà triée à une position donnée. .
3. Ecrire une fonction qui permet de supprimer un élément d'un tableau.

## Exercice 34 :

I. Algorithme d'Euclide.

1. Ecrire une fonction qui donne le pgcd de deux entiers positifs a et b.
2. Ecrire une fonction qui donne le ppcm de deux entiers positifs a et b.
3. Comparer ces deux fonctions avec les fonctions `igcd()` et `ilcm()` de Maple (utiliser `time()`).

II. Décomposition d'un entier en facteurs premiers.

1. Ecrire une fonction qui stocke dans une liste les nombres premiers inférieur ou égale à n. (on peut utiliser `isprime()`).
2. Ecrire une fonction qui construit la liste des facteurs premiers de l'entier n, (on peut utiliser `ifactor()`).
3. Faire des testes avec des nombres de 10, 15 et 20 de chiffres (utiliser `time()`).

III. Suite de Fibonacci .

La suite de Fibonacci est donnée par les équations :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \text{pour} \quad n \geq 2 \end{cases}$$

Ecrire une procédure qui affiche les n premiers termes de la suite de Fibonacci.

## Exercice 35 :

I. Implantation des algorithmes de tri simple.

1. Ecrire une fonction qui tri un tableau par la méthode d'insertion simple.
2. Ecrire une fonction qui tri un tableau par la méthode de sélection.
3. Ecrire une fonction qui tri un tableau par la méthode de tri à Bulles :

   **Principe** :

L'algorithme de permutation simple est basé sur le principe de la comparaison et de l'échange de couples d'éléments adjacents jusqu'à ce que tous les éléments soient triés.

- Parcourir le tableau en comparant deux à deux les éléments successifs, permuter s'ils ne sont pas dans le bon ordre.
- Répéter tant que des permutations sont effectuées.

II. Comparaison des algorithmes sur des exemples concrets. Comparer avec la fonction `sort()` de Maple (utiliser `time()`).

## Exercice 36:

La représentation d'un nombre entier positif en binaire.

1. Ecrire une fonction qui converti un entier décimal en binaire (utiliser la fonction `convert()` de Maple).
2. Ecrire une fonction qui converti un entier décimal en binaire (sans utiliser la fonction `convert()` de Maple).
3. Ecrire une fonction qui converti un entier binaire en décimal (utiliser la fonction `convert()` de Maple).
4. Ecrire une fonction qui converti un entier binaire en décimal (sans utiliser la fonction `convert()` de Maple).
5. Ecrire une fonction qui donne le nombre de bits nécessaire pour coder un entier positif n en binaire.
6. Ecrire une fonction qui donne le nombre de bits égaux à 1 dans sa décomposition en binaire.

# 6. Evaluation d'un algorithme

## 6.1 Introduction

Quand on tente de résoudre un problème, la question se pose souvent du choix d'un algorithme. Quels critères peuvent guider ce choix ? Deux besoins contradictoires sont fréquemment en présence : l'algorithme doit

1. être simple à comprendre, à mettre en œuvre et à mettre au point,
2. mettre intelligemment à contribution les ressources de l'ordinateur, et plus précisément, il doit s'exécuter le plus rapidement possible.

Si un algorithme ne doit servir qu'un petit nombre de fois, le premier critère est le plus important. Le temps employé à concevoir le programme dépassera vraisemblablement de beaucoup celui de son exécution, et le temps à optimiser est celui passé à écrire le programme. Si la solution du problème à traiter doit être employée assez souvent, le temps d'exécution risque d'être un facteur bien plus déterminant que le temps passé à écrire le programme. Dans ces conditions, il paraît rentable de mettre en œuvre un algorithme plus élaboré.

## 6.2 Mesure de la complexité algorithmique

Le temps d'exécution d'un algorithme dépend des facteurs suivants :

1. Les données entrant dans le programme,
2. La qualité du code généré par le compilateur pour la création du programme objet,
3. La nature et la vitesse d'exécution des instructions du microprocesseur utilisé pour l'exécution du programme.
4. La complexité du programme.

Le temps d'exécution d'un programme ne dépend pas de la nature précise des données mais de leur taille. On note $T(n)$ le **temps d'exécution** ou la **complexité algorithmique** d'un programme portant sur des données de taille n. Dans ce cours la complexité d'un algorithme désigne le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par l'algorithme. Elle s'exprime en fonction de la taille $n$ des données. On définit $T(n)$ comme la complexité dans le pire des cas, c'est-à-dire la mesure de la complexité maximum, sur tous les ensembles de données possible de taille $n$ pour un programme donné. Il est aussi possible de définir la complexité en moyenne, $T_{moy}(n)$.

## 6.3 La notation de Landau "O"

### Définition :

Une fonction $T(n)$ est de l'ordre de $f(n)$ ( ou $\mathbf{O(f(n))}$) s'il existe deux constantes $c$ et $n_0$ telles que : $T(n) < cf(n)$, pour tout $n>n_0$.

### Exemples :

**1.** Soit la fonction $T(n)=(n+1)^2$ pour $n\geq 0$, alors la fonction $T(n)$ est un $\mathbf{O(n^2)}$ pour $n_0=1$ et $c=4$. En effet, pour $n\geq 1$, on a $(n+1)^2 \leq 4n^2$ (il suffit d'étudier le signe de $(n+1)^2 - 4n^2$).

2. La fonction $T(n)= 3n^3+2n^2$ est $O(n^3)$ avec $n_0=0$ et $c=5$. En effet, $3n^3+2n^2-5n^3=2n^2(1-n)\leq 0$ pour $n\geq 0$ ; par conséquent $T(n)$ est **$O(n^3)$.**
3. La fonction $T(n)=3^n$ n'est pas un $O(2^n)$. En effet, par absurde, supposons que $T(n)$ est un $O(2^n)$, c'est-à-dire il existe deux constantes $n_0$ et $c$ telles que pour tout $n\geq n_0$, $3^n\leq c2^n$, donc on a $3^n/2^n\leq c$ pour tout $n\geq n_0$. Mais $(3/2)^n$ est une suite géométrique qui tend vers $+\infty$, ce qui est absurde. Conclusion $T(n)$ ne peut pas être un $O(2^n)$ .

## 6.4 Les grandes classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

- La plupart des instructions de la plupart des programmes sont exécutées une fois ou au plus un petit nombre de fois. Si toutes les instructions d'un programme ont cette propriété on dit qu'il a une complexité constante c'est-à-dire *O(1).*
- Les algorithmes sub-linéaires, dont la complexité est en général en *O(log(n))*. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal *n.*
- Les algorithmes linéaires en complexité *O(n)* ou en *O(n log(n))* sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de *n* symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k>3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en *n*) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

On a la classification suivante de ces fonctions :

$$\log n << n^{1/2} << n << n(\log n) << n^2 << n^3 << 2^n << e^n << n!.$$

Voici une comparaison numérique entre ces fonctions, on notera la croissance tres rapides des fonctions exponentielles $(2^n, e^n, n!)$. A titre de comparaison, on estime le nombre de particules dans l'univers à $10^{80}$.

| $\log(n)$ | 3.3 | 6.6 | 10 |
|---|---|---|---|
| $n^{1/2}$ | 3.1 | 10 | 32 |
| **$n$** | **10** | **100** | **1000** |
| $n(\log n)$ | 33 | 664 | $10^4$ |
| $n^2$ | 100 | $10^4$ | $10^6$ |
| $n^3$ | $10^3$ | $10^6$ | $10^9$ |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{300}$ |
| $e^n$ | $2\times 10^4$ | $10^{43}$ | $10^{434}$ |

| | | | |
|---|---|---|---|
| n! | $3.6\times 10^6$ | $10^{158}$ | $10^{2568}$ |

## 6.5 Evaluation d'un algorithme

Pour évaluer un algorithme, nous chercherons une borne supérieure, relativement à toutes les exécutions possibles, du nombre d'actions élémentaires contenues dans l'algorithme, en fonction de paramètres correspondant à la taille des données. Ceci nous permettra d'obtenir l'ordre de grandeur du temps d'exécution dans le pire des cas. En effet, supposons que l'exécution faisant intervenir le maximum d'actions élémentaires nécessite :

- $n_1$ actions élémentaires de genre 1 (par exemple affectations)
- $n_2$ actions élémentaires de genre 2 (par exemple additions)
- ….
- $n_k$ actions élémentaires de genre k

Chaque $n_i$ demandant un temps $t_i$.

Le temps total pour cette exécution sera :

$$t = \sum_i t_i n_i$$

Pour une machine donnée :

$$c_1 \sum_i n_i \le T(n) \le c_2 \sum_i n_i$$

avec $c_1 = \min_i t_i$, $c_2 = \max_i t_i$.

Donc $T(n) = O\left(\sum n_i\right)$ est l'ordre de grandeur du temps de l'algorithme dans le pire des cas, indépendamment de la machine utilisée.

## 6.6 Exemples

**La somme des entiers allant de 1 à n** (algorithme déjà vu)

```
variables n, i, somme : entiers  /* instructions élémentaires */
debut
  écrire("Donner n :")          /*  1   écriture */
  lire(n)                       /*  1   lecture */
  somme ← 0                     /*  1   affectation *
  pour i allant de 1 à n        /*  1   affectation, n  comparaison,
                                        n   additions, n affectation  */
     somme ← somme + i          /*  n   additions, n affectations */
  ecrire("la somme est :",somme)/* 1 écriture  */
fin
```

**Total** **5n+5** instructions élémentaires

Supposons que :

- l'affectation, la lecture, l'écriture et l'addition prennent chacune un temps $t_1$,
- la comparaison prend un temps $t_2$.

Le temps nécessaire pour la réalisation de cet algorithme est :

$$(5+4n)t_1 + nt_2 = n(4t_1+t_2)+ 5t_1.$$

D'ou la complexité $T(n)$ est en $O(n)$.

**Schéma de Hörner**

**Problème** .

On considère le polynôme en $x$ réel, à coefficients réels, de degré $n$ :

$$P(x)=a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$$

On veut calculer sa valeur $P(x_0)$ pour $x = x_0$ donné, en utilisant les seules opérations élémentaires : addition et multiplication.

**1ère méthode** :

On peut écrire un algorithme qui calcule $a_nx_0^n, a_{n-1}x_0^{n-1}, \dots, a_1x_0$ les unes après les autres et les additionne (ajouter $a_0$).

Calculons en fonction de $n$ le nombre d'opérations élémentaires qui seront effectuées lors de l'exécution de cet algorithme :

- nombre de multiplications pour un $a\ x^i$ : $i$
- nombre de multiplications pour tous : $1+2+\dots+(n-1) = n(n-1)/2$
- nombre d'additions : $n$
- Total : $n(n+3)/2$.

Donc cet algorithme est $O(n^2)$.

**2ème méthode** :

P(x) peut s'écrire (schéma de Hörner) :

$$P(x)=(...(((a_nx + a_{n-1})x + a_{n-2})x + a_{n-3})x +...+ a_1)x+a_0$$

Grâce à cette écriture, on peut écrire un algorithme qui calcule $P(x_0)$ de la manière suivante :

**Analyse** : itérer $n$ fois :

- multiplier $A$ par $x_0$ et additionner le coefficient suivant
- mettre le résultat obtenu dans $A$

commencer avec $A=a_n$

**Conception** :

```
Nom : Algorithme d'Hörner
variables n, i : entier
          A, B, x: réels
début
     écrire("donner n et x : ")
     lire(n,x)
     écrire("donner A: ")
     lire(A)
     pour i allant de 1 à n faire
          lire(B)
          A←A*x+B
     finpour
     écrire(A)
Fin
```

**Test** : Schema_Hörner

```
entrer les valeurs de n et X :2 2
entrer la valeur de A : 1
entrer la valeur de B : 1
entrer la valeur de B : 1
```
la valeur du polynôme en 2.0000 est : 7.0000
```
entrer les valeurs de n et X :5 2.5
entrer la valeur de A : -1
entrer la valeur de B : 1.5
entrer la valeur de B : 0
entrer la valeur de B : 3.2
entrer la valeur de B : 4
entrer la valeur de B : 1
```
la valeur du polynôme en 2.50000 est : 306.656250

**Complexité** :

Cet algorithme comporte *n* multiplications, *n* addition, *n+3* lectures, *n* affectations, 1 écriture. Il est en $O(n)$.
Il est donc meilleur que le précédent en ce sens que pour les grandes valeurs de n, le temps d'exécution du premier a pour ordre de grandeur le carré du deuxième.

Tableau de comparaison (on ce place dans le cas d'une machine qui effectue $10^6$ opérations par seconde) :

| N | Algorithme naïf | Algorithme d'Hörner |
|---|---|---|
| $10^3$ | 1 sec | $10^{-3}$ sec |
| $10^6$ | $10^6$ sec ≅11.6 jours | 1 sec |
| $10^{10}$ | $10^{14}$ sec ≅ $3\times10^6$ années | 2h 45min |

## Remarques :

1. On peut se demander si ça vaut le coup de se creuser la tête pour passer d'un algorithme à un autre algorithme. Regardons donc quelques chiffres pour se convaincre que oui (pour une machine pouvant effectuer $10^6$ opérations par seconde) :

| | Const | lg (N) | N | N lg(N) | $N^2$ | $N^3$ | $e^N$ |
|---|---|---|---|---|---|---|---|
| $10^2$ | 1µs | 6.6µs | 0.1ms | 0.6ms | 10ms | 1s | $10^{16}$ans |
| $10^3$ | 1µs | 9.9µs | 1ms | 9.9ms | 1s | 16.6min | ∞ |
| $10^4$ | 1µs | 13.3µs | 10ms | 0.1s | 100s | 11.5j | ∞ |
| $10^5$ | 1µs | 16.6µs | 0.1s | 1.6s | 2.7h | 31ans | ∞ |
| $10^6$ | 1µs | 19.9µs | 1s | 19.9s | 11.5j | $3\times10^4$ ans | ∞ |

2. Estimation de la taille maximale des données que l'on peut traiter par un algorithme de complexité donnée (pour une machine pouvant effectuer $10^6$ opérations par seconde) :

| | Const | lg (N) | N | N lg(N) | $N^2$ | $N^3$ | $e^N$ |
|---|---|---|---|---|---|---|---|
| 1µs | ∞ | ∞ | $10^6$ | $63\times10^3$ | $10^3$ | 100 | 19 |
| 1min | ∞ | ∞ | $6\times10^7$ | $28\times10^5$ | $7\times10^3$ | 390 | 25 |
| 1h | ∞ | ∞ | $36\times10^8$ | $13\times10^7$ | $60\times10^3$ | $15\times10^2$ | 31 |
| 1j | ∞ | ∞ | $86\times10^9$ | $276\times10^8$ | $29\times10^4$ | $44\times10^2$ | 36 |

# 6.7 Applications

## 6.7.1 Les algorithmes de recherche dans un tableau

**Problème** : recherche d'un élément dans un tableau.

### Recherche séquentielle

**Analyse** :

Parcours séquentiel du tableau
Arrêt lorsque la valeur est trouvée
    Retour l'indice correspondant
La valeur n'est pas trouvée
    Retour d'une valeur spéciale

**Conception** :

Une fonction qui prend comme paramètre `x` (la valeur cherchée), un tableau `Tab[]` (dans lequel on effectue la recherche, un entier `n` (la taille du tableau) et elle rend un entier.

```
fonction recherche(n:entier, Tab[]:entier, x:entier):entier
  variables i : entier
  début
       i ← 0
       Tant que ( i < n  et  Tab[i] != x) faire
            i ← i + 1
       fintantque
       si (i<n) alors retourne(i)
       sinon retourne(-1)
  fin
```

**Test** :

- Si on prend `n=10`, `Tab= 1 –9 2 0 3 5 2 4 11 -1` et `x=7` la fonction retourne `-1` (puisque `7` ne se trouve pas dans `Tab` !)

- Si on prend `n=10`, `Tab= 1 –9 2 0 3 5 2 4 11 -1` et `x=2` la fonction retourne `2` (puisque `Tab[2]=2` , la première position trouvée)

## **Complexité** :

On considère juste les comparaisons :

- le meilleur cas : 3 comparaison
- le pire cas : 2n+1 comparaisons
- en moyenne : (2+4+…+2n)/n=2(n+1)/2=n+1

Conclusion la complexité T(n) est O(n)

## Recherche dichotomique

On se place dans le cas où le tableau est **ordonné** et les éléments du tableau sont deux à deux distincts.

**Analyse** :

valRech : élément recherché,
Tab : tableau ordonné, sans duplication,
taille : la taille du tableau (le nombre d'éléments du tableau)
mil : indice du milieu du tableau,

```
si valRech = Tab[mil] alors retourne mil
sinon
     si valRech < Tab[mil] alors
          recherche dans la 1ère moitié du tableau (Tab[d … mil-1])
     sinon
          recherche dans la 2ème moitié du tableau (Tab[mil+1 … f])
     finsi
finsi
retourne(-1)
```

**Conception** :

```
     variables d, f, mil, entières
     début
          d ← 0
          f ← taille – 1
          Tantque (d<=f)
               mil ← (d + f)/2
               si (valRech = Tab[mil]) retourne mil
               si (valRech<Tab[mil])  f ← mil-1
               sinon
                    d ← mil + 1
          fintantque
          retourne(-1)
     fin
```

**Test** :

Soit le tableau T[1..9] (ordonné et sans répétition) suivant :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 8 | 11 | 17 | 18 | 22 | 45 | 102 |

On cherche l'élément **8**.

| **d** | 0 | 0 | 1 |
|---|---|---|---|
| **f** | 8 | 3 | 3 |
| **mil** | 4 | 1 | 2 |

Au début : `d=0`, `f=8` comme `d<f` on rentre dans la boucle "`Tantque`", `mil=(0+8)/2=4` et comme `valRech<Tab[mil]` (8 <17) on a `f=mil-1 (f=3)`, `d=0`, puisque `d<f` (`0<3`) on reste dans la boucle `Tantque` et on a `mil =(0+3)/2=1` et comme `valRech>Tab[mil]` (`8>6`), donc `d=mil+1=2` et `f=3` par conséquent `d<f (2<3)`, alors on reste dans la boucle `Tantque` et `mil=2`, la `valRech=Tab[mil]` (`8=Tab[2]`) alors la fonction retourne la valeur `2` qui est bien la position que occupe la `valRech=8`.

## **Complexité** :

A chaque étape :

- 1 addition, 1 division, entre 1 et 3 tests
- division par 2 de la taille du tableau, donc `lg(n)` étapes

**Conclusion** : la complexité `T(n) = O(lg(n)).`

**Comparaison** des deux méthodes de recherche (séquentielle et dichotomique) :

| Taille | Recherche séquentielle | Recherche dichotomique |
|---|---|---|
| 16 | 16 | 4 |
| 1024 | $1024=2^{10}$ | 10 |
| $2^{1024}$ | $2^{1024} \approx 10^{307}$ | 1024 |

**Exercice 32** : Soit le tableau T[0..10] (ordonné et sans répétition) suivant :

-2 3 5 7 10 17 19 23 50 62 70

1. Chercher l'élément 3 dans le tableau T, par dichotomie.
2. Chercher l'élément 19 dans le tableau T, par dichotomie.
3. Chercher l'élément 56 dans le tableau T, par dichotomie.

## 6.7.2 Les algorithmes de tri

**Problème** : réarranger les éléments d'un tableau dans l'ordre croissant ou décroissant pour rendre plus efficaces les opérations de recherche et, par conséquent, les opération d'insertion, et de suppression, etc.

## Tri par sélection

**Principe** :

Ranger le premier élément
Trier le reste du tableau.

**Analyse** :

Recherche le plus petit élément : ppe
Permuter avec le premier élément du tableau
Trier le tableau à partir de l'élément suivant.

- Données :
  - Entrées : `Tab` : tableau à trier, `taille` : la taille du tableau,
  - Sorties : tableau trié
  - Locales : le plus petit élément et son indice (`ppe` et `indppe`).
- Traitements :
  - Boucle : trouver `ppe` (le plus petit élément)
  - Permuter `ppe` et le premier élément,
  - Trier à partir de l'élément suivant.

**Conception** :
On présente deux versions une itérative et l'autre récursive :
1ère version :

```
fonction Tri_par_selection(Tab[]:entier, taille:entier)
     variables ppe, indppe, i, j, tmp : entiers
     début
          pour i allant de 0 à taille-2 faire
               ppe ← Tab[i]
               indppe ← i
               pour j allant de i+1 à taille-1 faire
                    si (Tab[j] < ppe)
                         ppe = Tab[j]
                         indppe = j
               finpour
               si (i != j)
                    permuter(Tab,indppe, i)
          finpour
     fin
```

Application : Exécuter l'algorithme précédent avec le tableau T suivant :

| 44 | 55 | 12 | 42 | 94 | 18 | **06** | 67 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

Dans le tableau : l'élément courant est souligné alors que le plus petit élément est en **gras**.

**Complexité :**
Dans la fonction Tri_par_selection () on comptabilise les opérations de comparaisons, on a deux boucles imbriquées, le total des comparaisons est 1+2+…+(n-1) = n(n-1)/2.
Conclusion : la complexité $T(n) = O(n^2)$.

## Tri par insertion simple

**Principe** :
Cette méthode est très utilisée lorsqu'on joue aux cartes. Les éléments (les cartes) sont divisés en une suite destination $a_1$ … $a_{i-1}$ et une suite source $a_i$ … $a_n$. A chaque étape, en partant de `i=2` et en augmentant `i` de `1`, on prend le $i^{ème}$ élément de la suite source et on l'insère à sa place dans la suite destination. Pour insérer l'élément couramment considéré, on déplace simplement les élément qui lui sont supérieurs un cran vers la droite et on l'insère dans la place laissée vacante.

**Analyse** :

Pour i allant de 2 à n faire
    x ← Tab[i]
    insérer x à la bonne place dans $a_1$ … $a_{i-1}$

**Conception** :

```
Tri_insertion_simple( Tab[] :entier, taille :entier)
     i, j, x : entiers
     début
          pour i allant de 2 à taille-1 faire
               x ← Tab[i]
               j ← i
               Tant que (x<Tab[j-1]) faire
                    Tab[j] ←Tab[j-1]
                    j ← j-1
               Tab[j] ← x
     fin
```

Application : Exécuter l'algorithme précédent avec le tableau suivant :

| | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
|---|---|---|---|---|---|---|---|---|
| **i=2** | | | | | | | | |
| **i=3** | | | | | | | | |
| **i=4** | | | | | | | | |
| **i=5** | | | | | | | | |
| **i=6** | | | | | | | | |
| **i=7** | | | | | | | | |
| **i=8** | | | | | | | | |

**Commentaires sur le programme** :

- Comme pour le tri par sélection, les éléments situés à gauche de l'indice i sont dans le bon ordre relatif pendant le tri, mais ne sont pas toujours dans leur position finale puisqu'ils peuvent être déplacés par la suite. Malgré tout, le tableau est entièrement trié lorsque l'indice atteint l'extrémité droite.
- Il y a un point important à remarquer : la fonction Tri_insertion_simple() est prise en défaut la plupart du temps ! La boucle *tantque* provoquera le dépassement de l'extrémité gauche lorsque x contient la valeur du plus petit élément du tableau. Pour éviter ceci, on place dans

Tab[0] une "sentinelles" inférieure ou égale à la plus petite valeur rencontrée. Si, le recours à une sentinelle n'est pas recommandé (parce qu'il est dur de définir la plus petite valeur du tableau), le test while( j>1 && x<Tab[j-1]) peut être utilisé.

**Complexité** :
Pour chaque i de 1 à n, il y a un échange et n-i comparaisons, soit un total de n-1 échanges et (n-1) + (n-2) + … + 2 + 1 = n(n-1)/2 comparaisons. Donc Le nombre de comparaisons effectuées par le tri par sélection est de l'ordre de $n^2/2$ et le nombre d'échanges est de l'ordre de n. La complexité est donc $T(n)=O(n^2)$.

## Tris par permutation simple (tri à bulles et tri shaker)

On va étudier une méthode dans laquelle la permutation de deux éléments est la caractéristique essentielle du processus de tri.

## Tri à bulles

**Principe** :
L'algorithme de permutation simple est basé sur le principe de la comparaison et de l'échange de couples d'éléments adjacents jusqu'à ce que tous les éléments soient triés.

1. Parcourir le tableau en comparant deux à deux les éléments successifs, permuter s'ils ne sont pas dans le bon ordre.
2. Répéter tant que des permutations sont effectuées.

**Exemple** :
On veut trier le tableau :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |

On effectue des passes successives sur le tableau en faisant glisser chaque fois le plus petit élément de l'ensemble restant vers l'extrémité gauche du tableau. Si, pour changer un peu, nous imaginons que le tableau est vertical et non horizontal, et que les éléments sont comme des bulles dans un réservoir d'eau, les bulles dont le poids est à trier, chaque passe sur le tableau conduit à l'ascension d'une bulle vers le niveau correspondant à son poids, voir le tableau suivant. C'est pourquoi cette méthode est très connue sous le nom de **tri à bulles**.

| **i = 1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| 44 | 06 | 06 | 06 | 06 | 06 | 06 | 06 |
| 55 | 44 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12 | 55 | 44 | 18 | 18 | 18 | 18 | 18 |
| 42 | 12 | 55 | 44 | 42 | 42 | 42 | 42 |
| 94 | 42 | 18 | 55 | 44 | 44 | 44 | 44 |
| 18 | 94 | 42 | 42 | 55 | 55 | 55 | 55 |
| 06 | 18 | 94 | 67 | 67 | 67 | 67 | 67 |
| 67 | 67 | 67 | 94 | 94 | 94 | 94 | 94 |

**Conception** :

```
fonction Tri_a_bulles ( Tab[] :entier, taille :entier)
     i, j, tmp : entiers
     début
          pour i allant de 2 à n faire
               pour j allant de n à i par –1 faire
                    Si Tab[j-1]>Tab[j] alors
                         tmp = Tab[j-1]
                         Tab[j-1] = Tab[j]
                         Tab[j] = tmp
                    finsi
               finpour
          finpour
     fin
```

**Raffinement** :

- Cet algorithme peut être facilement amélioré. Dans l'exemple précédent les trois dernières passes ne servent à rien puisque les éléments sont déjà triés. Une manière évidente d'améliorer cet algorithme est donc de se souvenir qu'il y a eu ou non une permutation pendant une passe ; une dernière passe sans permutation est cependant nécessaire pour savoir que l'algorithme peut être arrêté.
- On peut encore faire mieux en se rappelant non seulement du fait qu'il y a eu permutation, mais encore de la position (l'indice k) de la dernière permutation. Par exemple, il est clair que toutes les paires d'éléments adjacents en dessous de l'indice k sont dans l'ordre désiré. Les balayages suivants peuvent donc se terminer à k au lieu d'aller jusqu'à la limite prédéterminée i.
- On remarque aussi une certaine asymétrie : une bulle "légère" mal placée dans la partie la plus "dense" du tableau trouvera sa place en une seule passe, alors qu'une bulle plus lourde placée dans la partie la moins dense ne progressera que d'une position vers sa place définitive. C'est ainsi que le tableau :

  12 18 42 44 55 67 94 **06**

  est trié par un tri à bulles en une seule passe :

  06 12 18 42 44 55 67 94

  alors que le tableau :

  **94** 06 12 18 42 44 55 67

  nécessitera sept passes.

  06 94 12 18 42 44 55 67
  06 12 94 18 42 44 55 67
  06 12 18 94 42 44 55 67
  06 12 18 42 94 44 55 67
  06 12 18 42 44 94 55 67
  06 12 18 42 44 55 94 67
  06 12 18 42 44 55 67 94

Cette asymétrie non naturelle nous suggère une autre amélioration : changer de direction à chaque passe : c'est ce qu'on appelle le **tri shaker**.