Chapitre 6 (Algorithmique/C)

Les tableaux

Objectif

Maîtriser la manipulation des tableaux à une ou à plusieurs dimensions (en particulier le cas de deux dimensions).

I. Introduction

Supposons que nous avons à déterminer à partir de 30 notes fournies en entrée, le nombre d'étudiants qui ont une note supérieure à la moyenne de la classe.

Pour parvenir à un tel résultat, nous devons :

1. Lire les 30 notes
2. Déterminer la moyenne de la classe : m
3. Compter combien parmi les 30 notes sont supérieures à la moyenne m.

Il faut donc conserver toutes les notes en mémoires afin qu'elles soient accessibles durant l'exécution du programme.

Solution 1 : Utiliser 30 variables réelles nommées x1, x2, …, x30

Cette façon de faire présente deux inconvénients :

- Il faut trouver un nom de variables par note ;
- Il n'existe aucun lien entre ces différentes valeurs. Or, dans plusieurs cas, on est appelé à appliquer le même traitement à l'ensemble ou à une partie de ces valeurs.

Solution 2 : Utiliser la notion de **tableau** qui consiste à :

- Attribuer un seul nom à l'ensemble des 30 notes, par exemple **Tnote**.
- Repérer chaque note par ce nom suivi, entre crochets, d'un numéro entre 1 et 30 : Tnote[1], Tnote[2], ..Tnote[30].

II. Tableaux unidimensionnels

Un tableau à une dimension ou encore unidimensionnel, appelé aussi vecteur, est une suite de « cases » de même taille contenant des éléments d'un type donné et

directement accessibles par leurs indices ou index. Chaque *élément* ou *composant du tableau* est accessible en lecture ou écriture.

Un tableau contenant des entiers peut se représenter de la façon suivante :

entier	entier	entier	…	entier	entier

II.1 Déclaration d'un tableau

Algorithmique

Pour définir une variable de type tableau, il faut préciser :

- le nom (identifiant du tableau)

- l'indice (généralement de type entier)

- le type des éléments (entier, réel, caractère, etc.)

On note :

Variables

Nom_Tab : **Tableau** [**PremInd..DernInd**] de **Type_éléments**

Où **PremInd** est le premier indice (généralement 1) et **DernInd** est le dernier indice (qui indique la taille réelle du tableau). Ainsi, un tableau est caractérisé par sa dimension c'est-à-dire le nombre de cases qu'il contient.

Exemple :

Tnote : Tableau [1..30] de Réel

Schématiquement, ce tableau peut être représenté comme suit : Tnote

10.5	8	….	15
1	2		30

Remarque :

Il est également possible de définir un type tableau comme dans l'exemple suivant :

Constantes

NMAX = 5

Types

Tab=Tableau[1..NMAX] de Entier

Variables

T : Tab

La solution du problème précédent est la suivante :

```
ALGORITHME étudiant
Var  Tnote : tableau [1..30] de réels
```

```
    i : entier
Début
    pour i de 1 à 30 faire
        Lire (Tnote [i])
    finpour
    Ecrire (" Voici les notes des étudiants")
    pour i de 1 à 30 faire
        Ecrire (Tnote[i])
    finpour
Fin
```

Langage C

Syntaxe : type_éléments nom_tableau[nb_cases];

type_éléments : indique le type des éléments contenus dans le tableau. Toutes les données doivent être de même type.

nom_tableau : le nom du tableau.

nb_cases : le nombre de cases dans le tableau (la dimension).

La numérotation des cases s'effectue de 0 à nb_cases-1.

Exemple : int tab [10] ;

float A [100] ;

Remarque : Un élément du tableau est repéré par son indice. En langage C, les indices commencent à partir de 0 et non de 1 comme en algorithmique.

Exemples :

```
compteur[2] = 5;

nombre[i] = 6.789;

printf("%d",compteur[i]);

scanf("%f",&nombre[i]);
```

Mémorisation

Le nom d'un tableau est le représentant d'une adresse du premier élément d'un tableau. Les adresses des autres composants sont calculées automatiquement relativement à cette adresse.

Si un tableau est formé de N composants et si un composant a besoin de M octets alors le tableau occupera N*M octets.

Initialisation et réservation automatique

On peut initialiser un tableau en indiquant la liste des valeurs entre accolades {}.

Syntaxe : type nom_tableau[N] = { val1, val2, ..., valN } ;

➔ N désigne la taille réelle du tableau nom_tableau. La taille réservée pour ce tableau est de **N* sizeof (type).** Le tableau peut contenir des cases remplies et des cases vides. ***Le nombre des cases remplies désigne la taille effective du tableau.***

En langage C, on peut initialiser les tableaux au moment de leur déclaration. Toutefois, en algorithmique, il faudra initialiser le tableau élément par élément.

Exemple : int A[5] = {20, 12, 12, 32, -1}

Dans le cas où la taille de la liste des valeurs dépasse la dimension indiquée pour le tableau, on aura une erreur.

Exemple : int B[3] = {10, 20, 30, 40} ➔ erreur

On peut ne pas indiquer la taille du tableau mais définir les valeurs inclues.

Exemple : int A [] = {10, 20, 30}

➔ La taille réservée sera 3 * sizeof (int)

Remarque : Attention aux bornes du tableau, l'écriture doit se faire entre 0 et nb_cases-1. Si on déclare un tableau de 7 cases (numérotées de 0 à 6) et qu'on tente d'écrire dans la case 9, on obtiendra le message : "Erreur de segmentation" (ou *segmentation fault*).

II.2 Affectation et Accès aux composants

Algorithmique

Un élément dans un tableau est identifié de la façon suivante :

NomTab[position de l'élément]

➔ Cela traduit bien l'accès direct aux éléments du tableau.

Ainsi, **Tnote [3]** désigne la note du **3éme** étudiant et d'une façon générale **T[i]** désigne le **ième** élément du tableau T.

Remplissage d'un tableau :

Un tableau peut être rempli élément par élément à l'aide d'une série d'affectations :

T[1]← Valeur 1

T[2]← Valeur 2

….

T[n]← Valeur n

n désigne la taille effective du tableau.

Il est également possible de lire les éléments du tableau à partir du clavier comme dans la procédure suivante :

Procédure remplir(Var T : tab, n : entier)

Variables

i : entier

Début

Pour i de 1 à n **Faire**

Ecrire(‘’ Entre un entier : ‘’)

Lire(T[i])

FinPour

Fin

Application 1 : Insertion d'un élément dans un tableau à la case d'indice p

Soit le type **tab** suivant :

Tab : Tableau[1..100] de entier

On vous demande de remplir n case du tableau A ensuite d'ajouter un élément à la position d'indice p.

Procédure insertion_élément(Var A : Tab, Var n : entier)

Variables

i,x : entier

p : entier

Début

Répéter

Ecrire (“Donner la taille du tableau“)

Lire(n**)**

Jusqu'à ((n>0) ET (n<=100))

Pour i **de** 1 **à** n **Faire**
Ecrire(“Donner A[“,i,“]“)
Lire(A[i])
FinPour

Si (n<100) Alors
Ecrire (“ Donner l’élément à insérer**“)**
Lire(x**)**
Répéter
Lire(p)
Jusqu’à (p<=n+1)
Si (p=n+1) Alors
n←n+1
A[n]←x
Sinon
Pour i de n à p Pas -1 Faire
A[i+1]←A[i]
FinPour
A[p]←x
FinSi
Sinon
Ecrire(“Tableau déjà rempli !“)
FinSi

Fin

Application 2 : Suppression d’un élément dans un tableau à la case d’indice p

Soit le type **tab** suivant :

Tab : Tableau[1..100] de entier

On vous demande de remplir n case du tableau A ensuite de supprimer un élément à la position d’indice p.

Procédure suppression_élément(Var A : Tab, Var n : entier)

Variables

i **: entier**

```
        p : entier
Début
    Répéter
        Ecrire (“Donner la taille du tableau“)
        Lire(n)
    Jusqu’à ((n>0) ET (n<=100))
    Pour i de 1 à n Faire
        Ecrire(“Donner A[“,i,“]“)
        Lire(A[i])
    FinPour
    Répéter
                    Lire(p)
    Jusqu’à (p<=n)
    Si (p=n) Alors
        n←n-1
    Sinon
        Pour i de p à n-1 Faire
          A[i]←A[i+1]
        FinPour
       n←n-1
    Finsi
 Fin
```

Affichage des éléments d’un tableau :

L’affichage des éléments d’un tableau se fait également élément par élément. Seulement, le tableau constitue ici un paramètre donné et non pas un résultat comme dans la procédure de remplissage.

```
    Procédure afficher (T : tab, n : entier)
    Variables :
        i : Entier
    Début
        Pour i de 1 à n Faire
                Ecrire(T[i])
        FinPour
```

Fin

Exercice :

Soit T un tableau contenant de taille n éléments de type entier. Ecrire une fonction MinTab qui retourne le plus petit élément de ce tableau.

Langage C :

L'accès à la valeur d'une case d'un tableau se fait par :

Syntaxe : nom_tableau[numero_case] ;

Pour affecter des valeurs dans des cases, on utilise la syntaxe suivante :

Syntaxe : nom_tableau[numero_case] = valeur;

Exemple : tab[9] = 15 ;

La valeur de tab est l'adresse du premier élément du tableau. Autrement dit, tab a pour valeur &tab[0]. On peut, donc, utiliser un pointeur initialisé à tab pour parcourir les éléments du tableau.

L'accès à la valeur de la case numéro i peut se faire en calculant l'adresse tab + (i-1)*sizeof(tab[0]).

L'élément i du tableau s'obtient aussi en calculant *(tab + i) donc tab[i] = *(tab + i).

➔ Le passage d'un tableau dans une fonction ou une procédure se fait par adresse puisque le tableau tab désigne toujours une adresse.

```
//Affichage des éléments d'un tableau

void afficher(int * tab, int n)

{
 int i;
 int *p;
 for (i = 0,p=tab; i < n; i++,p++)
   {
```

```
    printf(" %d \n",*p);
  }
}
```

Ou encore

```
void afficher(int tab[], int n)
{
 int i;
 for (i = 0; i < n; i++)
  {
   printf(" %d \n",tab[i]);
  }
}
```

Dans les deux cas l'appel à la fonction « afficher » se fait comme suit : afficher(t, n) (n est égal à la taille du tableau). La variable t est toujours passé par adresse.

Applications :

1. Ecrire une fonction qui permet de calculer la somme des éléments d'un tableau.

2. Ecrire une procédure qui permet d'inverser un tableau.

II.3 Algorithmes de recherche

On a souvent besoin de chercher, dans un grand tableau, la position d'un élément donné. Il ne faut pas oublier de traiter :

- ✓ le cas où l'élément recherché n'est pas dans le tableau
- ✓ le cas d'éléments identiques (doit-il donner le premier, le dernier, tous ?)

Il existe deux types de recherche :

- ✓ Recherche séquentielle

- ✓ Recherche dichotomique

a- **Recherche séquentielle**

Il suffit de lire le tableau progressivement du début vers la fin. Si le tableau n'est pas trié, l'arrivée à la fin du tableau signifie que l'élément n'existe pas. Dans un tableau trié, le premier élément trouvé supérieur à l'élément recherché permet d'arrêter la recherche.
Une recherche dans un tableau trié nécessitera en moyenne N/2 lectures, mais on se rapprochera de N pour un tableau non trié avec beaucoup de recherche d'éléments inexistants.

Soit T un tableau contenant n éléments de type entier, on veut écrire une procédure dont l'entête sera

Procédure recherche(T : tab, x : Entier)

Cette procédure affiche :

1) l'indice de la première occurrence de x dans T si x est dans T

2) le message « élément introuvable… » si x n'est pas dans T.

Principe : *comparer x aux différents éléments du tableau jusqu'à trouver x ou atteindre la fin du tableau.*

Exemple :

```
Type  Typ_tab = Tableau [1.. 20] de entier
var   trouve : Booléen
      i : entier
début
   i ← 1
   Trouve ← Faux
   Tantque (i<=n) et (Non Trouve) faire
      si T[i] = val  Alors   Trouve ← Vrai
      Sinon          i ← i+1
      finsi
   finTantque
Fin
```

b- Recherche dichotomique

Dans le cas d'un tableau trié, on peut limiter le nombre de lectures, en cherchant à limiter l'espace de recherche. On compare la valeur recherchée à l'élément central du tableau. Si ce

n'est pas la bonne, un test permet de trouver dans quelle moitié du tableau on trouvera la valeur. On continue récursivement jusqu'à un sous tableau de taille 1.

Soit T un tableau contenant n éléments **triés** dans le sens croissants :

T

1	3	5	5	8
1	2	3	4	5

→ Quelque soit i dans [1, n-1], T[i] ← T[i+1]

On veut écrire une procédure dont l'entête est de la forme

Procédure Rechdicho(T : tab, x : Entier)

Principe :

Le but de la recherche dichotomique est de diviser l'intervalle de recherche par 2 à chaque itération. Pour cela, on procède de la façon suivante :

Soient premier et dernier les extrémités gauche te droite de l'intervalle dans lequel on cherche la valeur x, on calcule m, l'indice de l'élément médian :

→ m=(premier + dernier) div 2

Il y a 3 cas possibles

- x<T[m] : l'élément x, s'il existe, se trouve dans l'intervalle [premier..m-1]
- x>T[m] : l'élément x, s'il existe, se trouve dans l'intervalle [m+1..dernier].
- x=T[m] : l'élément de valeur x est trouvé, la recherche est terminée.

La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou que l'intervalle de recherche soit vide.

Exemple :

```
var    trouve : Booléen
       g,m,d : entier  /* gauche, milieu, droite */
début
  g ← 1,  d ← n
  Trouve ← Faux
  Tantque (g<=d) et (Non Trouve) faire
       m ← (g+d)/2          /* division entière */
       si val<T[m]  Alors   d ← m-1
       Sinon
            si val>T[m]  Alors   g ← m+1
           Sinon      Trouve ← Vrai
```

```
            finsi
        finsi
    finTantque
Fin
```

II.4 Algorithmes de tri

Dans certains cas, la recherche d'un élément dans un tableau non ordonné est difficile et très lente. Pour cela, pour faciliter la recherche et minimiser le temps d'accès à un élément dans un tableau, on doit le trier c'est à dire l'ordonner, le classer : on donne un ordre bien déterminé (croissant ou décroissant) à ses éléments.

Les trois formes de tri les plus utilisées sont :

* Tri par sélection
* Tri à bulles
* Tri par insertion

a- Tri par sélection

Le principe est de parcourir un tableau à fin de chercher le plus petit élément qui est ensuite permuté avec le premier élément. Le même traitement s'effectue avec le tableau ayant le premier élément en moins. Cette opération est répétée jusqu'à ce que tous les éléments soient en places.

```
Algorithme T-Selection
Type
    tab = tableau[1..n] de entier
Var
     i, j, n, valmin, indmin : entier
     T : tab
Début
    lire(n)
    pour i de 1 à n faire
         lire(T[i])
    fin pour
    pour i de 1 à n-1 faire
          indmin    ←    i
          valmin    ←    T[i]
          pour j  de i + 1 à n  faire    /* Recherche de la valeur min */
```

```
            si (T[j] < valmin)  alors
                valmin ←      T[j]
                indmin ←      j
             fin si
        finpour
        T[indmin] ←      T[i]
        T[i]      ←      valmin
    finpour
    pour i de 1 à n  faire
        Ecrire(T[i])
    finpour
Fin
```

b- Tri à bulles

La méthode de tri à bulles consiste à répéter le traitement suivant : comparer le premier élément avec le deuxième et les échanger si le premier est plus grand que le second. On compare ensuite le second avec le troisième (le second peut ainsi contenir l'ancien premier) et on les échange si le second est plus grand que le troisième. On continue jusqu'à la fin de la première passe. À la fin de cette passe, le plus grand est nécessairement au bout du tableau. On recommence le même processus mais cette fois, on se rendra jusqu'à l'indice N-1 ce qui aura pour effet de repousser (faire remonter comme une bulle) le prochain plus grand à sa place.

```
pour i = n à 1 pas –1 faire
        pour j = 2 à i faire
                si T[j-1] > T[j] Alors
                        tampon ← T[j-1]
                        T[j-1]← T[j]
                        T[j]← tampon
                finsi
        finpour
finpour
```

c- Tri par insertion

Le tri par insertion est aussi simple que le tri par sélection mais il est sans doute plus flexible. La méthode s'apparente à celle que beaucoup de joueurs utilisent pour trier leurs mains, après la donne, au jeu de cartes.

Considérer que le premier élément est à sa place. Prendre le second élément et l'insérer dans la partie triée (celle-ci est de longueur 1 pour l'instant) en déplaçant les éléments triés au besoin pour faire de la place. Prendre le troisième élément et l'insérer dans la partie triée (celle-ci est de longueur 2 maintenant) en déplaçant les éléments triés au besoin pour faire de la place. Prendre le quatrième élément et l'insérer dans la partie triée (celle-ci est de longueur 3 maintenant) en déplaçant les éléments triés au besoin pour faire de la place. On continue jusqu'à l'élément N.

```
pour i = 2 à n faire
    elem ← T[i]
    j ← i
    Tantque (j>1) et  (T[j-1]>elem)  faire
        T[j] ← T[j-1]
        j ← j-1
    finTantque
    T[j]← elem
finpour
```

Application :

Ecrire un algorithme qui permet de créer deux tableaux de tailles 10 : l'un contenant les noms des étudiants et l'autre leurs moyennes (moyenne est comprise entre 0 et 20). Les données seront saisies à partir du clavier. Le programme devra afficher à la fin du traitement le nom et la moyenne des étudiants moyennant le format « Nom : Moyenne ». Par ailleurs, il doit aussi donner :

a. Le nom et la moyenne du meilleur étudiant (moyenne maximale)
b. Le nom et la moyenne du plus mauvais étudiant

Proposez deux solutions en utilisant deux algorithmes de recherches différents.

III. Tableaux à plusieurs dimensions

Les tableaux multidimensionnels, et en particuliers ceux à deux dimensions, sont des extensions des tableaux à un seul indice, ils servent à représenter des objets mathématiques plus complexes comme les matrices.

Un tableau à N dimensions est en fait un tableau unidimensionnel de tableaux à N-1 dimensions.

Déclaration en algorithmique :

```
Type
   Type_tableau = tableau [B1i .. B1s, B2i .. B2s, …, Bni .. Bns] de type_éléments
```

```
Var
   Nom_tableau : type_tableau
```

Tout ce qui a été présenté auparavant reste donc valable. La décision d'utiliser des tableaux multidimensionnels doit être bien réfléchie : ces tableaux nécessitent l'utilisation de beaucoup de mémoire.

Dans la suite de ce chapitre, nous nous focalisons principalement sur les tableaux à deux dimensions.

III.1 Déclaration d'un tableau à deux dimensions

Algorithmique

Un tableau à deux dimensions est interprété comme un tableau unidimensionnel de dimension L dont chaque composante est un tableau unidimensionnel de dimension C. On dit qu'un tableau à deux dimensions est carré si L=C.

Cette matrice peut être définie de la façon suivante :

Types

Mat : Tableau [1..3,1..4] de Réel

Variables

Matrice : Mat

Chaque élément de la matrice est repéré par deux indices :

- Le premier indique le numéro de la ligne.
- Le second indique le numéro de la colonne

Ainsi, Matrice [2,4] désigne l'élément situé à la 2ème ligne et la 4ème colonne.

Remarque

Cette représentation est arbitraire, on a pu considérer le premier indice pour la colonne et le second désigne la ligne. Dans ce cas l'élément Matrice [2,4] devient inexistant.

Application : Ecrire un algorithme qui permet de définir, créer et remplir un tableau à deux dimensions comme le suivant :

A	**B**
C	**D**
E	**F**

```
Type
```

```
    Matrice = tableau [1..3, 1..2] de caractères
Var
    M : Matrice
Début
        M[1, 1]  ← 'A'
        M[1, 2] ←  'B'
        M[2, 1] ←  'C'
        M[2, 2] ←  'D'
        M[3, 1] ←  'E'
        M[3, 2] ←  'F'
Fin
```

Langage C :

Déclaration d'un tableau

Syntaxe : type_éléments nom_tableau[taille_dim1][taille_dim2] ;

type_éléments : indique le type des éléments contenus dans le tableau. Toutes les données doivent être de même type.

nom_tableau : le nom du tableau.

Taille_dim1 : la taille des lignes du tableau.

Taille_dim2 : la taille des colonnes du tableau.

Un tableau T est présenté comme suit :

T [0] [0]	T [0] [1]	T [0] [2]
T [1] [0]	T [1] [1]	T [1] [2]
T [2] [0]	T [2] [1]	T [2] [2]

Exemples :

```
int compteur[4][5];
float nombre[2][10];
```

Appel :

```
nom[indice1][indice2]
```

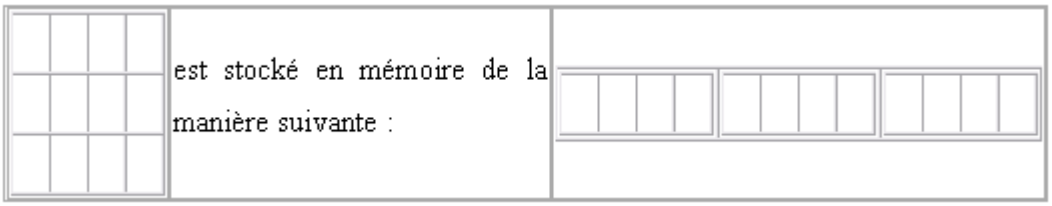
Exemples :

```
compteur[2][4] = 5;
nombre[i][j] = 6.789;
printf("%d",compteur[i][j]);
scanf("%f",&nombre[i][j]);
```

Mémorisation

Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux. Ils sont stockés en mémoire comme des tableaux unidimensionnels (ligne par ligne).

Par exemple le tableau bidimensionnel (3 lignes, 4 colonnes) suivant, est en fait un tableau comportant 3 éléments, chacun d'entre eux étant un tableau de 4 éléments :



Initialisation et réservation automatique

La taille réservée pour un tableau ayant L ligne(s) et C colonne(s) et de type « type » est L*C*sizeof (type).

Différentes manières existantes pour initialiser les valeurs d'un tableau. On peut initialiser un tableau en indiquant la liste des valeurs entre accolades {}, c'est-à-dire dès sa déclaration.

Syntaxe : type nom_tableau[N][P] = { {valN1_1, valN1_2, ..., valN1_P}, {valN2_1, valN2_2, ..., valN2_P} ..., {valN_1, valN_2, ..., valN_P}} ;

➔ La taille réservée pour ce tableau est de **N*P* sizeof (type)**

Exemple 1 : int A [3][2] = {{1, 2}, {3, 4}, {5, 6}} ;

Exemple 2 : int A[][3] = {{1, 2, 3}, {10, 20, 30}, {100, 200, 300}}

Si la taille du tableau dépasse le nombre de valeurs définies, ces valeurs sont remplacées par 0 sinon une erreur est générée.

Exemple : int C [4][2] = {{1,2}}

1	2
0	0
0	0
0	0

Exemple : C [4][2] = {{1,2,3}} ➔ erreur.

III.2 Remplissage d'un tableau à deux dimensions

Le remplissage d'un tableau bidimensionnel à n lignes et m colonnes se fait à peu près de la même façon qu'un tableau unidimensionnel. Seulement, il est nécessaire d'utiliser deux boucles imbriquées correspondant chacune à l'indice d'une dimension :

Procédure remplir(Var matrice : Mat)

Variables

i,j : Entier

Début

Pour i de 1 à n **Faire**

Pour j de 1 à m **Faire**

Ecrire(``Entrer un entier : '')

Lire(matrice[i,j])

FinPour

FinPour

Fin

L'accès à la valeur d'une case d'un tableau bidimensionnel se fait par :

Syntaxe : nom_tableau[numero_case_ligne][numero_case_colonne] ;

Pour affecter des valeurs dans des cases, on utilise la syntaxe suivante :

Syntaxe : nom_tableau[numero_case_ligne][numero_case_colonne] = valeur;

Exemple : tab[2][1] = 12 ;

tab a une valeur constante égale à l'adresse du premier élément du tableau, &tab[0][0]. Pour i de 0 à M-1 (pour un tableau dont le nombre de lignes est M), l'élément tab[i] a donc une valeur constante qui est égale à &tab[i][0].

Applications :

1. Ecrire un programme C qui permet de remplir un tableau de taille maximale 50*50.

2. Ecrire un programme C qui permet de sommer les lignes et les colonnes d'un tableau à deux dimensions.

3. Ecrire un programme C qui permet de rendre un tableau à deux dimensions en un tableau à une dimension.

Application 1 : Transposition d'une matrice carrée

Une matrice carrée est une matrice à n ligne et n colonnes. L'opération de transposition consiste à inverser les lignes et les colonnes en effectuant une symétrie par rapport à la diagonale principale de la matrice.

Exemple

La matrice

1	2	3
4	5	6
7	8	9

Devient

1	4	7
2	5	8
3	6	9

```
Procédure Transpose(Var matrice : Mat)
Variables
        i,j,x : Entier
Début
        Pour i de 1 à n Faire
                Pour j de (i+1) à n Faire
                        x← matice[i,j]
                        matice[i,j] ← matice[j,i]
                        matice[j,i] ←x
                FinPour
         FinPour
Fin
```

Application 2 : Somme de deux matrices

Soient M1 et M2 deux matrices à n lignes et m colonnes, on veut écrire une procédure qui calcule les éléments de la matrice M3=M1+M2

Exemple

M1

1	2	3
4	5	6

M2

2	5	3
3	0	1

M3

3	7	6
7	5	7

Procédure SomMat(M1,M2 : Mat ;Var M3 : Mat)

Variables

i,j : Entier

Début

Pour i de 1 à n **Faire**

Pour j de 1 à m **Faire**

M3[i,j] ← M1[i,j]+M2[i,j]

FinPour

FinPour

Fin

Application 3 : Produit de deux matrices

Soient M1 une matrice ayant n lignes et m colonnes

M2 une matrice ayant m lignes et p colonnes

On veut écrire une procédure qui calcule les éléments de la matrice M3=M1*M2. Notons d'abord que le nombre de colonnes doit être égal au nombre de lignes de M2.

Le produit M3=M1*M2 est définit comme étant une matrice ayant n lignes et p colonnes et dont les éléments sont calculés par la formule :

M3[i,j]=M1[i,1]M2[1,j]+M1[i,2]M2[2,j]+…+M1[i,m]M2[m,j]

Exemple

M1

1	2	3
4	0	5

M2

2	1
3	0
1	4

M3

11	13
13	24

Procédure ProdMat(M1 : Mat1 ; M2 : Mat2 ;Var M3 : Mat)

Variables

i,j,k : Entier

Début

```
Pour i de 1 à n Faire
    Pour j de 1 à p Faire
        M3[i,j]←0
        Pour k de 1 à m Faire
          M3[i,j] ← M3[i,j]+ M1[i,k]*M2[k,j]
        FinPour
    FinPour
 FinPour
Fin
```