Algorithme et Programmation

Certains voient, à tort, dans l'ordinateur une machine pensante et intelligente, capable de résoudre bien des problèmes. En fait, celui-ci ne serait capable de rien si quelqu'un (le programmeur en l'occurence) ne lui avait fourni la liste des actions à exécuter. Cette description doit être faite de manière non ambigüe car il ne faut pas s'attendre à la moindre interprétation des ordres fournis. Ils seront exécutés de manière purement mécanique. De plus, les opérations élémentaires que peut exécuter un ordinateur sont en nombre restreint et doivent être communiquées de façon précise dans un langage qu'il comprendra. Le problème principal du programmeur est donc de lui décrire la suite des actions élémentaires permettant d'obtenir, à partir des données fournies, les résultats escomptés. Cette description doit être précise, doit envisager le moindre détail et prévoir les diverses possibilités de données.
Cette marche à suivre porte le nom d'**algorithme** dont l'Encyclopaedia Universalis donne la définition suivante :

" Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données. "

En fait, la notion d'algorithme est très ancienne, et est indépendante de l'informatique et des machines : les marchands, par exemple, avaient recours à certains algorithmes pour calculer des intérêts etc ...
De même, au vu de la définition précédente, une recette de cuisine est un algorithme, tout comme une notice de montage d'un meuble en kit. Dans ces derniers cas, on voit bien la nécessité d'un algorithme clair (compréhensible par tous), le plus général possible (pour répondre au plus de cas possibles) et rapide (efficace).

Structure d'un algorithme

Structure d'une recette de cuisine : on commence par énumérer tous les ingrédients nécessaires ; puis on décrit les étapes successives à faire. Enfin, si l'algorithme est bon, un merveilleux gateau sort du four !

En fait, tout algorithme se décompose en 3 parties. La première partie concerne les **entrées** : on liste toutes les données nécessaires. La deuxième partie regroupe toutes les instructions ; et la troisième est la **sortie** ou résultat.

Programmation

Un programme est la traduction d'un algorithme en langage compréhensible par la machine.
Nous ne verrons que le langage Turbo-Pascal.

Structure générale d'un programme en Turbo-Pascal

1. en-tête du programme
2. déclaration éventuelle des constantes, des types et des variables
3. déclaration éventuelle des procédures et fonctions
4. corps du programme

Explications succintes :

1. syntaxe : `PROGRAM` nom_du_programme ;
2. il faut déclarer tous les objets utilisés dans le corps du programme : c'est une exigence du langage Turbo-Pascal.
   syntaxe :

```
CONST  a=2;
       x=3.5;
 VAR   i,k,n : INTEGER;          les entiers prennent leurs valeurs dans ⟦−32768, 32767⟧
       x,a,b : REAL;             x,a,b sont des réels
       test : BOOLEAN;           variable de type booléenne : sa valeur est true ou false.
       T : ARRAY [1..10] of integer ;   tableau de 10 cases à coefficients entiers
       mat : ARRAY[1..3,1..3] of real ;  tableau à 3 lignes 3 colonnes (matrice carrée d'ordre 3) à coefficients réels.
```

La déclaration des types sera vue ultérieurement.

3. les fonctions et procédures seront vues au second semestre : ces sous-programmes permettent de structurer davantage un programme.

4. c'est le corps du programme : on liste les instructions à exécuter.
   Cette partie est donc la traduction de l'algorithme proprement dit.

   syntaxe : `BEGIN`
   instructions ;
   ⋮
   `END.`

→ Sauf mention contraire, chaque ligne de commande se finit par un ;.

→ On peut ajouter des commentaires au programme pour améliorer la lisibilité en expliquant certaines instructions ou noms de variable. Ces commentaires peuvent être insérées n'importe où du moment qu'ils sont placés entre accolades : { blablabla }

1 Variables et actions élémentaires

1.1 Les variables

En informatique, une variable est une "case" mémoire, réservée pour stocker la future valeur de cette variable (valeur qui peut être modifiée au cours du programme).
Une variable doit avoir un nom (ou identificateur), ainsi qu'un type (entier, réel etc...), qui définit la taille de la case mémoire. Ces deux attributs sont déclarés simultanément dans la partie 2. :
syntaxe : `VAR` i :`INTEGER` ;
note :`REAL` ;

Remarque sur les identificateurs : vous pouvez choisir n'importe quel mot (non réservé bien sûr), du moment qu'il ne contient ni espace, ni accents, ni apostrophes. Conseil : choisir un mot ou une lettre évocatrice !

Puis dans le corps du programme, une variable doit être **initialisée**. Deux moyens sont possibles :

- **par lecture**
  ex : `writeln('écrire les valeurs de a et b');`
  quand le programme s'éxécute, on voit apparaître à l'écran : écrire les valeurs de a et b.
  `readln(a,b);`
  les deux valeurs alors écrites par l'utilisateur (séparées par un espace ou un retour à la ligne) sont affectées aux variables a et b.

- **par affectation**
  ex : x :=1 ; {x reçoit la valeur 1}
  ou si a et b ont été initialisées comme précédemment, x :=(a+b)/2 affecte à x la valeur (a+b)/2.
  Attention :
  1. si l'on écrit `a:=b`, il faut que les variables a et b soient de même type.
  2. Ne pas confondre := avec le = réservé aux données constantes et au symbole de comparaison.

1.2 Opérations élémentaires

Ces opérations dépendent du type des objets (variables ou constantes). Attention, les opérations sont à effectuer entre des objets de même type !

Opérateurs et fonctions arithmétiques

opérateurs	entrée(s)	sortie	commentaires
+ - *	réel/entier	réel /entier	opérations élémentaires
/	réel /entier	réel	le type de sortie peut donc être différent du type de l'entrée
div mod	entier	entier	quotient et reste de la division euclidienne : `37 div 5` donne 7 et `37 mod 5` donne 2.
exp, ln, sqrt	réel/entier	réel	sqrt est la racine carrée
sqr	réel/entier	réel/entier	carré
trunc	réel	entier	partie entière
abs	réel/entier	réel/entier	valeur absolue

Opérateurs relationnels
`=`, `<>` pour $\neq$, `<`, `>`, `<=` pour $\leq$, `>=` pour $\geq$.
Les variables d'entrées sont de type entier/réel ; la variable de sortie est de type booléen.
Ces opérations sont principalement utilisés dans les instructions conditionnelles (que l'on reverra ultérieurement) :

if	opérateur	**then**	instructions	(**else** instructions)
while	opérateur	**do**	instructions	
repeat	instructions	**until**	operateur	

Opérateurs logiques
Il y en a 3 : `not, or` et `and` ; qui ne seront utilisés que dans les instructions conditionnelles ci-dessus.
Par exemple : $(x < -1)$ `AND` $(x > 1)$ ou encore $(x > 0)$ `OR` $(y = 0)$, peuvent être une condition du **if**.

2 Instructions conditionnelles

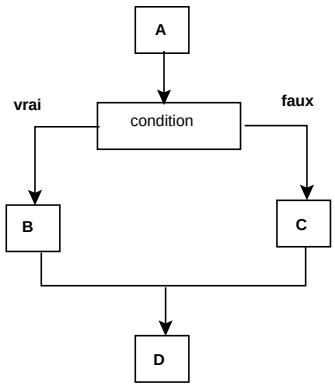
2.1 instructions

Une instruction est dite simple si elle se réduit à une seule commande : `writeln('bonjour')` ou `s:=0;`
Sinon, l'instruction est dite composée : on peut aussi parler de groupe d'instructions.

2.2 if then else

Cette instruction répond à l'attente :
Si une condition est vraie (par exemple $x \neq 0$), alors on veut effectuer une certaine instruction (par exemple, diviser par x) et sinon, on en effectue une autre.
La syntaxe est la suivante :

instructions A ;
IF condition **THEN**
  `BEGIN`
    instructions B ;
  `END`
ELSE
  `BEGIN`
    instructions C ;
  `END;`
instructions D ;



Remarques :

i. Si les instructions B ou C sont simples, le `BEGIN END;` correspondant est inutile.

ii. Le **ELSE** est facultatif ; mais il ne doit pas être précédé immédiatement d'un ';'.

Exemples de condition : $x = 0$ ou $x >= 10$ où x est une variable définie précédemment.
Les conditions peuvent être *composées* : par exemple $(0 < x)$ `AND` $(x <= 1)$ (qui signifie $0 < x \leq 1$).

Attention : ne pas confondre $x := 3$ et $x = 3$ lorsque x est une variable.
$x := 3$ est une affectation : elle correspond à l'instruction "donner à x la valeur 3".
$x = 3$ est une condition booléenne (proposition qui est vraie au fausse) :
elle correspond à la question "x est-il égal à 3 ?". Ces conditions booléennes ne seront présentes que dans les instructions conditionnelles (if then else) ou les boucles conditionnelles (while et repeat until).

3 Boucles itératives

3.1 Boucle FOR

Cette instruction s'emploie pour répéter une suite d'instructions n fois, lorsque n est connu à l'avance.
La syntaxe est la suivante :

instructions A ;
FOR $i := n_1$ **TO** n_2 **DO**
`BEGIN`
instructions B ;
`END;`
instructions C ;

Remarques :

i. Le groupe d'instructions B est effectué une première fois avec $i = n_1$, une deuxième avec $i = n_1 + 1$,...., puis une dernière avec $i = n_2$. Attention la variable compteur i doit être déclarée en 2 !

ii. n_1 et n_2 sont deux nombres entiers ou alors variables de type entier, déclarées et initialisées.

iii. Si l'instruction B est simple le `BEGIN END;` correspondant est inutile.

iv. Si $n_1 > n_2$, le groupe d'instructions B n'est pas exécuté.

Variante :
Si l'on veut aller de n_2 à n_1 dans l'ordre décroissant, la syntaxe devient :
FOR $i := n_2$ **DOWNTO** n_1 **DO** .

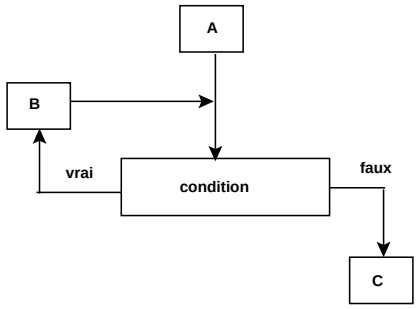
3.2 Boucles conditionnelles

Par opposition à l'instruction répétitive `FOR`, les instructions `WHILE` et `REPEAT` s'emploient dès que l'on ne connaît pas à l'avance le nombre d'itérations à effectuer.

3.2.1 Boucle WHILE

On répéte une suite d'instructions tant qu'une certaine condition est vraie.
La syntaxe est la suivante :

instructions A ;
WHILE condition **DO**
`BEGIN`
instructions B ;
`END;`
instructions C ;



Remarques :

i. Les instructions B peuvent ne pas être exécutées du tout (si dès le départ la condition est fausse)

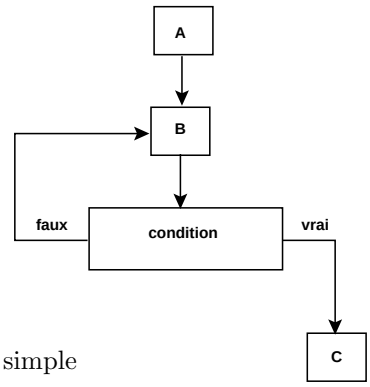
ii. Si l'instruction B est simple, le BEGIN END ; correspondant est inutile

iii. Attention de bien vous assurer, avant de lancer le programme, que la condition devient fausse au bout d'un certain temps. Sinon, le programme ne s'arrêtera jamais.

3.2.2 Boucle REPEAT

On répéte un groupe d'instructions jusqu'à ce qu'une condition soit vraie.
La seule différence avec la boucle `WHILE` est que dans la boucle `REPEAT` les instructions B sont exécutées **avant** de tester la condition donc sont exécutées au moins une fois.
La syntaxe devient :

instructions A ;
REPEAT
instructions B ;
UNTIL condition ;
instructions C ;



Remarques :

i. **REPEAT UNTIL** permet de délimiter l'instruction B, donc qu'elle soit simple ou composée, il n'y a pas besoin de `BEGIN END;`.

ii. Attention de bien vous assurer, avant de lancer le programme, que la condition devient vraie au bout d'un certain temps : sinon le programme ne s'arrêtera jamais.

4 Fonctions et Procédures

4.1 Notion de sous-programme

Il peut arriver que l'on doive utiliser une même séquence d'instructions à différents endroits d'un programme. Il est alors judicieux de créer un sous-programme (fonction ou procédure) dont le code sera défini une fois pour toutes dans l'étape 3, et que l'on appellera dans le corps du programme aux différents endroits souhaités.

Avantage : le corps du programme est plus court et plus lisible puisque l'on évite ainsi des répétitions de code.

Exemple : le calcul de $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ nécessite le calcul de 3 factorielles ; on pourra écrire une fonction factorielle en amont, et ensuite l'appeler 3 fois.

Exemple bis : dans la dichotomie, on fait appel à une fonction à chaque étape. Dans notre exemple c'est la fonction $f(x) = e^x - 2$ qui est facile à écrire à chaque fois. Mais si la fonction plus complexe, il est nécessaire de créer une fonction turbo-pascal en amont.

Un sous-programme a la même structure qu'un programme mais le END du corps du sous-programme est suivi d'un **;** et non d'un **.** Il faut déclarer ces sous-programmes dans **l'étape 3.**, juste avant le corps du programme. Toutes les variables utilisées dans un sous-programme doivent être définies soit comme des paramètres d'entrée soit comme des variables locales (les variables du programme sont appelées variables globales). Mais l'appel du sous-programme se fait dans le CORPS du programme (**étape 4**).
Il y a deux sortes de sous-programmes : les fonctions et les procédures.

4.2 Les fonctions

Une fonction est un sous-programme qui fournit **un résultat numérique** à partir des données qu'on lui apporte (les paramètres d'entrées) : la notion de fonction en Turbo-Pascal est assez proche de la notion de fonction en mathématiques.
Une fonction se déclare comme suit :

FUNCTION Nom-de-la-fonction (Nom-du-paramètre : Type-du-paramètre) : Type-sortie ;
VAR Déclaration des éventuelles variables LOCALES (ou constantes via CONST) ;
BEGIN
Corps de la fonction (liste des instructions) ;
Nom-de-la-fonction := Valeur-sortie ;
END ;

Remarques :
- Une fonction ne peut avoir en sortie qu'un réel ou un entier.
- Il peut y avoir plusieurs paramètres d'entrées dans une fonction. Ils seront alors séparés par un " ; " dans la déclaration mais par une " , "lors de l'appel de la fonction dans le corps du programme.
  exemple : `function nom(n:integer ; x:real):real;` ou...

Exemple 1 :
```
PROGRAM fonction;

FUNCTION ffff(x : REAL):REAL ;
BEGIN
ffff := 3*sqr(x)+2;
END;

BEGIN
write(ffff(3));
END.
```

Exemple 2 :
a) écrire la fonction Turbo-pascal factorielle, qui à un entier positif n associe $n!$.
b) l'intégrer dans un programme qui demande deux entiers n et $k \leq n$ à l'utilisateur, et qui renvoie $\binom{n}{k}$

```
PROGRAM ex2;
VAR k,n:INTEGER;
FUNCTION factorielle (N : INTEGER) : INTEGER ;
VAR i,F : INTEGER ;
BEGIN
  F:=1;
```

```
  FOR i:=1 TO N DO
  F:=F*i;
  factorielle :=F;
END;
BEGIN
write('Entrer deux entiers n et k');readln(n,k);
write(factorielle(n)/(factorielle(k)*factorielle(n-k));
END.
```

4.3 Les procédures

Contrairement aux fonctions, la procédure ne fournit pas un résultat mais crée une action ou une suite d'actions (instructions) : on utilisera les procédures principalement lorsqu'on manipulera des tableaux (matrices ou autres). En effet, une fonction ne permet pas de modifier un tableau puisque son but est de renvoyer un nombre. Sa syntaxe est analogue à la syntaxe d'une fonction :

PROCEDURE Nom-de-la-procédure (VAR Nom-du-paramètre : Type-du-paramètre) ;
VAR Déclaration des éventuelles variables LOCALES (ou constantes via CONST) ;
BEGIN
Corps de la procédure (liste des instructions) ;
END ;

Remarque : il existe une variante dans l'en-tête de la procédure
PROCEDURE Nom-de-la-procédure (Noms-paramètres : Type-paramètres) ;

Cette variante sans VAR sera utilisée lorsqu'on ne voudra pas que le paramètre d'entrée puisse être modifié par la procédure. On dit alors que le paramètre d'entrée passe par valeur (et non par variable) : ce paramètre d'entrée peut être juste un nombre (et non nécessairement une variable).

Exemple :
a) Ecrire une procédure qui échange le contenu de 2 variables.
b) L'intégrer dans un programme qui demande deux réels $x1$ et $x2$ à l'utilisateur et qui les trie dans l'ordre croissant.

```
PROGRAM exemple;
VAR x1,x2:REAL;
PROCEDURE echange(VAR a,b : REAL);
VAR aux : REAL;
BEGIN
 aux := a;
 a:=b;
 b:=aux;
END;
BEGIN
write('donner deux réels'); readln(x1,x2);
IF x1 > x2 then echange(x1,x2);
END.
```

5 Simulation d'expériences aléatoires

Introduction :

Supposons que l'on ait une pièce truquée dont on veut déterminer la probabilité p d'obtenir face.
Le moyen expérimental à notre disposition est de lancer un très grand nombre de fois la pièce, par exemple $n = 1000$, et de compter le nombre d'occurences "face" : n_f. Alors, le rapport $\frac{n_f}{n}$ appelé fréquence d'apparition de face, est une approximation de p, d'autant meilleure que n est grand.

Plus précisément, si l'on réalise n répétitions indépendantes d'une même expérience aléatoire de départ, la fréquence de réussite d'un certain événement tend vers la probabilité de cet événement lorsque $n \to +\infty$.
Donc pour avoir une valeur approchée de cette probabilité, il faut effectuer un grand nombre de fois la même expérience aléatoire, ce que permet l'outil informatique en très peu de temps ! ! Il suffit que le langage utilisé puisse créer du hasard : pour celà, il doit posséder un générateur de nombres aléatoires.

Syntaxe :
Pour activer le générateur de nombres pseudo-aléatoires de Turbo-Pascal, il faut insérer dans le début du corps du programme, la commande `Randomize ;`.
Ensuite, deux fonctions sont prédéfinies dans Turbo-Pascal :

i. `random ;` sans argument retourne un réel compris entre 0 et 1.

ii. `random(`n`);` où $n \in \mathbb{N}^*$, retourne un entier (parmi les n) compris entre 0 et $n-1$ avec équiprobabilité. Autrement dit, cette fonction `random` (n) permet de simuler la loi uniforme sur $\{0, 1, 2, \ldots, n-1\}$.

6 Les tableaux

Utilité des tableaux :

⟶ Pour stocker des données : dans la feuille de TP 8, on stockera les différents résultats d'une expérience aléatoire dans un tableau, avant de les analyser.
⟶ Pour travailler sur des matrices ou des polynômes.

6.1 Déclarer un tableau

6.1.1 Tableau à une entrée

Avant de pourvoir utiliser un tableau, il faut le déclarer comme variable :
`VAR tableau : ARRAY[deb..fin] OF (type) ;`

Dans cette syntaxe, `deb` et `fin` sont de type `integer`, sous la contrainte `deb` $\leq$ `fin`.
La variable `tableau` est alors un tableau composé de colonnes numérotées de `deb` à `fin`.
Ce tableau comportera donc `fin-deb+1` cases, remplies par des éléments de type `type`.
Chaque case du tableau est alors une **variable**, identifiée par `tableau[i]`. Au départ, le tableau est **vide** : il faudra donc penser à initialiser toutes les cases du tableau avant de l'utiliser !

Exemple : Les coordonnées d'un point A du plan peuvent être représentées par le tableau suivant :
`VAR coordA : ARRAY[1..2] OF REAL ;`
coordA[1] donne l'abscisse du point A et coordA[2] l'ordonnée.

Exemple : le polynôme $P(x) = 1 + 2x + 4x^3 + 5x^6 - 3x^9$, en tant que polynôme de degré inférieur ou égal à 10, peut être représenté par le tableau

1	2	0	4	0	0	5	0	0	-3	0

Il sera donc déclaré par : `VAR P : ARRAY[0..10] OF INTEGER ;`

6.1.2 Tableau à double entrées

Exemple : Une matrice A de $\mathcal{M}_{2,3}(\mathbb{R})$ sera déclarée par la syntaxe `VAR A : ARRAY[1..2,1..3] OF REAL ;`

tandis qu'une matrice B de $\mathcal{M}_{8,5}(\mathbb{R})$ à coefficients entiers le sera par `VAR B : ARRAY[1..8,1..5] OF INTEGER ;`

6.2 Créer un type "tableaux"

Rappel : la syntaxe d'une fonction comme d'une procédure demande la connaissance du type du paramètre d'entrée. Par exemple, `FUNCTION f (x : REAL) : INTEGER ;`

Pour pouvoir appliquer des fonctions ou des procédures à un tableau, il faut donc **déclarer le type** de ce tableau en amont. La déclaration d'un nouveau type se fait juste avant la déclaration des variables (étape 2 dans la structure générale d'un programme).

déclarations des types : `TYPE polynome = ARRAY[0..10] OF real ;`

Ensuite, se fait la **déclaration des variables** : `VAR P,Q,R : polynome ;`

Dès que l'on a défini un tel type `polynome`, on peut appliquer une fonction ou une procédure à un `polynome`. Attention de garder en mémoire qu'une fonction ne pourra jamais renvoyer un polynôme : sa sortie ne peut être qu'un réel (ou un entier). Mais une procédure pourra, par son action, changer la variable polynôme d'entrée.

Exemple : Lorsque l'on souhaitera travailler commodément avec plusieurs matrices de $\mathcal{M}_3(\mathbb{R})$, et créer des fonctions et procédures s'y rapportant, on procédera en déclarant :
`TYPE matrice = ARRAY[1..3,1..3] OF real ;`
`VAR A, B, C : matrice ;`

6.3 Opérations sur les tableaux

La seule opération globale possible sur les tableaux est l'affectation. Si `R` et `A` sont deux tableaux de même dimension et type de contenu, la commande `R:=A;` remplace le contenu du tableau `R` par celui du tableau `A`.

Toutes les autres opérations portant sur les tableaux se font **case par case**, donc en utilisant des boucles. Ce défaut engendre souvent des temps de traitement importants, en particulier pour les tableaux de grande taille.

Rappel : toutes les cases d'un tableau sont des variables donc doivent être initialisées par lecture ou affectation.

Tableaux à 1 dimension

• Déclarer le tableau :
`VAR T : ARRAY[1..50] of REAL;` (autre exemple : `VAR T : ARRAY[2..8] of INTEGER;`)

• Déclarer le type puis une variable de ce type :
```
TYPE tab = ARRAY[1..50] of REAL;
VAR T:tab;
```

• Initialiser la 15-ième case de `T` à 0 :
`T[15]:=0 ;`

• Initialiser la 18-ième case de `T` à une valeur entrée par l'utilisateur :
`WRITELN('Entrer la valeur de la case 18'); READLN(T[18]);`

• Afficher la valeur de la case 25 du tableau :
`WRITELN(T[25]);`

• Initialiser tout le tableau `T` à 0
`FOR k:=1 TO 50 DO T[k]:=0;`

• Initialiser tout le tableau `T` à des valeurs entrées par l'utilisateur :
```
FOR k:=1 TO 50 DO  BEGIN
                   WRITELN('Entrer la valeur de la case ',k);
                   READLN(T[k]);
                   END;
```

• Afficher tout le tableau :
`FOR k:=1 TO 50 DO WRITELN(T[k]);`

Tableaux à 2 dimensions

• Déclarer le tableau :
`VAR U : ARRAY[1..50,1..40] of REAL;`

• Initialiser la case située à la 11-ième ligne et 23-ième colonne de `U` à 0 :
`U[11,23]:=0 ;`

• Initialiser la case située à la 8-ième ligne et 27-ième colonne de `U` à une valeur entrée par l'utilisateur :
`WRITELN('Entrer la valeur de la case située ligne 8 colonne 27'); READLN(U[8,27]);`

• Afficher la valeur de la case située à la ligne 25, colonne 17 du tableau :
`WRITELN(U[25,17]);`

• Initialiser tout le tableau `U` à 0 :
```
FOR  i:=1 TO 50 DO
     FOR j:=1 TO 40 DO U[i,j]:=0;
```

• Initialiser tout le tableau `U` à des valeurs entrées par l'utilisateur :
```
FOR  i:=1 TO 50 DO
     FOR j:=1 TO 40 DO  BEGIN
                        WRITELN('Entrer la valeur ligne ',i,' colonne ',j);
                        READLN(U[i,j]);
                        END;
```

• Afficher tout le tableau :
```
FOR i:=1 TO 50 DO  BEGIN
                   FOR j:=1 TO 40 DO WRITELN(U[i,j]);
                   WRITELN; END;
```