Les tableaux

Pour représenter un grand nombre de données devant toutes rester directement accessibles, on ne peut se servir d'une variable scalaire puisque celle-ci ne peut donner accès qu'à une seule valeur à la fois. On ne peut songer à attribuer une variable à chaque donnée car cela pourrait devenir vite fastidieux aussi bien pour l'écriture que pour l'exécution du programme.
Dans la plupart des langages de programmation, le type *tableau* joue ce rôle. Une variable pourra représenter plusieurs données à la fois qui pourront être manipulées collectivement ou individuellement.

Les tableaux à une dimension

Les *tableaux monodimentionnels* ou *tableaux à une dimension* ou encore à *un indice* (qui seront désignés en fin d'année sous le terme de vecteurs) sont constitués d'*éléments* et d'*indices*.
Il faut donc définir :

- le type de l'élément de base du tableau (ce que contient le tableau)
- le type de l'indice (ce qui nous permet de choisir l'élément qui nous intéresse).

Les indices sont des entiers naturels allant de 1 à *N* si le tableau comporte *N* éléments. Contrairement à certains langages compilés (comme le Turbo-Pascal), ils peuvent être créés dynamiquement (il n'est pas utile de préciser dès le début leur taille).

Création ; plusieurs modes sont envisageables :

• Tableau défini par la liste de ses éléments ; il faut placer cette liste d'éléments, séparés par des virgules, entre crochets :

```
-->v=[2,3,5,9]
 v  =
!  2.    3.    5.    9. !
```

• Tableau dont les éléments constituent une progression arithmétique ; il faut utiliser la notation *n*:*m* ou *n*:*p*:*m* (où *n* est la valeur initiale, *p* le pas d'incrémentation et *m* la valeur finale à ne pas dépasser)

```
-->3:8
 ans  =
!  3.    4.    5.    6.    7.    8. !

-->a=12:-3:-4
 a  =
!  12.    9.    6.    3.    0.  - 3. !
```

• tableaux particuliers

Tableau dont tous les éléments sont nuls : fonction *zeros*

```
-->v=zeros(1,7)          //permet de créer un tableau à une dimension de taille 7, d'éléments nuls
 v  =
!  0.    0.    0.    0.    0.    0.    0. !
```

Tableau vide : [] ; il est défini :
- en écrivant []
- par la commande 3:1 (les entiers à partir de 3 avec un pas de 1 et inférieurs à 1)
- zeros(1,0)

Accès aux éléments

Si temp est une variable de type tableau de 7 entiers correspondant aux températures journalières d'une semaine donnée, les indices possibles sont 1, 2, 3, 4, 5, 6, 7. Les éléments de ce tableau seront des entiers notés temp(1), temp(2), temp(3), temp(4), temp(5), temp(6), temp(7). On peut représenter cette variable sous la forme :

	1	2	3	4	5	6	7
temp	8	6	7	11	6	8	9

L'expression placée entre parenthèse peut être quelconque, à condition qu'elle fournisse un résultat entier compris entre 1 et la taille du vecteur. L'accès à un élément du tableau peut, si k est un entier, se faire par :

Si k est initialisé à 5 temp(k)=6
temp(k+1)=8
temp(k-1)=11

Le calcul de la moyenne des températures de la semaine est :

```
-->somme=0;
-->for k=1:7 do somme=somme+temp(k);
-->end
-->moyenne=somme/7
 moyenne  =
   7.8571429
```

Les fonctions vectorielles

La fonction length fournit le nombre d'éléments d'un tableau

```
-->length([1,2,3,5,8,12])
 ans  =
   6.
-->length(3:2:197)
 ans  =
   98.
```

La variable $ permet de désigner le dernier élément d'un tableau à une dimension :

```
-->v=[2,7,1,-1,2,9];
-->v($)
 ans  =
   9.
-->v(length(v))
 ans  =
   9.
```

La fonction isempty() appliquée à un tableau permet d'indiquer si celui-ci est vide ou non :

```
-->isempty(1:10)
 ans  =
  F

-->isempty([])
 ans  =
  T
-->isempty(2:-1:8)
 ans  =
  T
```

Tableaux à deux dimensions

La déclaration se fait comme pour les tableaux à une dimension ; les lignes sont séparées entre elles par des points virgules :

```
-->A=[1,5,6,7;5,9,8,4]
 A  =
!  1.    5.    6.    7. !
!  5.    9.    8.    4. !
```

L'opération la plus simple pour l'accès aux éléments d'un tableau à deux dimensions est *M*(*n*,*p*) : cette opération lit l'élément du tableau situé à la ligne *n* et à la colonne *p*.

```
-->M=rand(3,4)      //tableau de taille (3,4) dont les éléments sont des réels aléatoires de [0;1]
 M  =
!   .8833888    .9329616    .3616361    .4826472 !
!   .6525135    .2146008    .2922267    .3321719 !
!   .3076091    .312642     .5664249    .5935095 !
-->M(2,3)
 ans  =
    .2922267
```

La variable $ permet alors de repérer les éléments d'indice maximal en ligne ou en colonne ; en reprenant l'exemple ci-dessus :

```
-->M(1,$)                -->M($,3)                -->M($,$)
 ans  =                   ans  =                   ans  =
    .4826472                 .5664249                 .5935095
```

La fonction size() permet de fournir les dimensions d'un tableau donné en paramètre.

```
-->size([1,2,5,8,14,-9])
 ans  =
!  1.    6. !
-->size([1,1,2;4,-9,-8;2,3,5])
 ans  =
!  3.    3. !
```

L'extraction de sous tableau

L'instruction matrix(*A*,*n*,*p*) permet de restructurer le tableau *A* selon les dimensions *n* et *p*.

```
-->M=matrix(1:2:29,3,5)
 M  =
!  1.    7.    13.    19.    25. !
!  3.    9.    15.    21.    27. !
!  5.    11.    17.    23.    29. !
```

Nous pouvons alors extraire de ce tableau, par exemple les colonnes 1,4 et 5 des lignes 2 et 3 :

```
-->N=M([2,3],[1,4,5])
 N  =
!  3.    21.    27. !
!  5.    23.    29. !
```

Nous pouvons également extraire directement un sous-tableau :

```
-->P=M(2:3,3:5)
 P  =
!  15.    21.    27. !
!  17.    23.    29. !
```

Les fonctions vectorielles

Si vous utilisez les fonctions suivantes vous devez être capable de donner un algorithme permettant de les construire.

max	maximum
min	minimum
sort	tri par ordre croissant
gsort	tri, ordres particuliers
sum	somme
prod	produit
cumsum	sommes cumulées
cumprod	produits cumulés
mean	moyenne
median	médiane
st-deviation	écart type

Gestion des polynômes

Dans un programme de calculs et de gestion des polynômes, on peut effectuer une représentation par des tableaux (elle existe déjà mais est hors programme des deux années)

Ainsi, le polynôme $-2X^6 + 3X^4 - 5X^3 - 8X + 7$ peut être représenté par une variable de type polynôme de taille 20 (qui est de taille suffisante pour envisager la gestion de polynômes) qui sera le tableau de réels suivant (tous les coefficients supérieurs au degré du polynôme doivent être nuls dans ce tableau de taille 20) :

1	2	3	4	5	6	7	8	…	19	20
+7.0	-8.0	0.0	-5.0	+3.0	0.0	-2.0	0.0	…	0.0	0.0

1) Ecrire une fonction **[P]=SaisiePoly** permettant d'entrer au clavier les coefficients d'un polynôme *P*.
2) Ecrire une fonction **[n]=degre(P)** permettant de déterminer le degré du polynôme *P*.
3) Ecrire une fonction **VoirPoly(P)** affichant l'expression de $P(x)$, *P* étant un polynôme donné en paramètre.
Facultatif : Vous pourrez examiner tous les cas d'affichage :
certains coefficients nuls,
addition avec un réel négatif donc soustraction,
…
4) Ecrire une fonction **EvalPoly** calculant la valeur de $P(x)$, le réel *x* et le polynôme *P* étant passés en paramètres d'entrée.
5) Ecrire une fonction **[R]=AddPoly(P,Q)** calculant la somme de deux polynômes *P* et *Q*.
6) Ecrire une fonction **[R]=MultPoly(P,Q)** calculant le produit de deux polynômes *P* et *Q*.
7) Ecrire un programme permettant de tester les modules ci-dessus.

Tri par bulle

Principe : Le principe consiste à parcourir le tableau en comparant 2 à 2 des éléments voisins et en les permutant éventuellement. Après un balayage complet, le plus grand élément se trouve à l'indice n. On recommence l'opération sur les *n* - 1 éléments restants. On s'arrête lorsque la partie du tableau restant à trier est d'un seul élément ou lorsqu'un balayage complet du tableau n'a pas provoqué de permutation

Par exemple, pour trier le tableau	3 5 2 1 8 9 7 3	
on obtiendra successivement	3 2 1 5 8 7 3 9	5 et 9 "remontent"
	2 1 3 5 7 3 8 9	3 et 8 "remontent"
	1 2 3 5 3 7 8 9	2 et 7 "remontent"
	1 2 3 3 5 7 8 9	5 "remonte"

Ecrire en Scilab une fonction tri_bulle(t)

Le nom 'par bulle" vient du fait que la plus grande valeur remonte petit à petit vers le haut comme une bulle. Cette méthode est efficace pour un tableau presque trié.

Tri par sélection croissant

Principe: Le principe de ce tri consiste à réaliser un premier parcours complet pour rechercher le minimum parmi les *n* éléments d'un tableau, puis un deuxième pour rechercher le minimum parmi les *n* - 1 éléments restants et ainsi de suite. A chaque tour, il faut comparer l'élément de départ, appelé *élément courant, à* tous ses successeurs et effectuer un échange d'emplacement lorsque l'élément courant est supérieur à l'élément utilisé dans la comparaison.

Par exemple, pour trier le tableau	6 2 8 1 5 4	
on obtiendra successivement		
	1 2 8 **6** 5 4	car le minimum est 1
	1 **2** 8 6 5 4	car le minimum est 2
	1 2 **4** 6 5 **8**	car le minimum est 4
	1 2 4 **5** **6** 8	car le minimum est 5
	1 2 4 5 **6** 8	car le minimum est 6
	1 2 4 5 6 **8**	

Ecrire en Scilab la fonction tri-selection(t)

tri par insertion

Principe : étant donné un tableau T, on le trie par ordre croissant en procédant de la manière suivante : chaque élément est inséré dans la liste supposée triée des éléments précédents.

Par exemple, pour trier le tableau	6 2 8 1 5 4	
on obtiendra successivement	6	puis on insère 2
(en n'écrivant que la partie triée du tableau)	2 6	puis on insère 8
	2 6 8	puis on insère 1
	1 2 6 8	puis on insère 5
	1 2 5 6 8	puis on insère 4
	1 2 4 5 6 8	

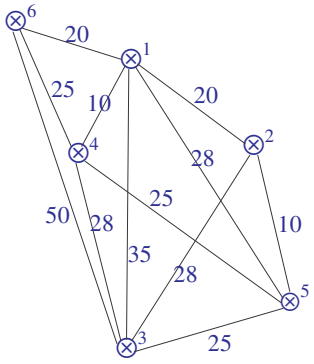
Description de l'algorithme :

- La variable i désignera l'élément t(i) à insérer dans la liste triée t(1)..t(i-1) (i varie de 2 à N)
- L'élément t(i) sera recopié dans une variable auxiliaire X pour éviter d'être perdu (ou "écrasé") au cours de l'insertion.
- Un deuxième compteur j parcourra l'intervalle [1..i-1] en décroissant pour localiser la place de X (tout élément t(j) supérieur à X sera recopié en t(j+1) puis j sera décrémenté)
- La recherche s'arrêtera dès que l'on aura soit j=0 soit t(j)<=X.
- X sera alors recopié en t(j+1)

Ecrire en Scilab la fonction tri_ins(t)

Problème du voyageur

Un voyageur doit se rendre dans plusieurs villes et souhaite trouver un parcours avantageux, permettant de passer dans toutes les villes en partant d'une ville de départ donnée. Les distances, deux à deux, entre les villes sont connues.
Nous supposerons qu'il utilise la stratégie suivante :
A partir de chaque ville, il choisit de se rendre à la ville la plus proche qu'il n'ait pas encore visitée.
L'ensemble des distances peut être donné sous forme d'un tableau carré symétrique ayant la diagonale principale constituée de 0 (la distance d'une ville à elle-même est nulle).
Exemple :
Soit le graphe suivant, représentant les connexions entre les six villes :



Le tableau des distances de l'exemple est :

	V1	V2	V3	V4	V5	V6
V1	0	20	35	10	28	20
V2	20	0	28	20	10	35
V3	35	28	0	25	25	50
V4	10	20	25	0	25	25
V5	28	10	25	25	0	45
V6	20	35	50	25	45	0

Il faut donc, en partant de la ville où l'on se trouve, rechercher le minimum de la ligne (si l'on fait la lecture par ligne) correspondante, en excluant les villes déjà visitées. Pour cela, il suffit de remplacer à chaque étape, toutes les distances d'une colonne correspondant à la ville où l'on se trouve, par la plus grande des distances possible, par exemple par la constante %inf.
1) Ecrire la fonction **[t]=Saisir** demandant à l'utilisateur d'initialiser le tableau des distances par une lecture au clavier.
2) Ecrire la fonction **[n]=min_ligne(i,t)** donnant l'indice de colonne du plus petit élément de la ligne i du tableau t.
3) Ecrire la fonction **[tab]=deja_fait(j,t)** remplaçant tous les éléments de la colonne j du tableau t par la constante %inf.
4) Ecrire un programme Scilab utilisant les fonctions précédentes et partant de la ville numérotée 1.
Vous utiliserez la variable globale distance donnant la distance parcourue et afficherez les étapes successives comme, en prenant la solution de l'exemple :

VILLE : 1 DISTANCE : 0
VILLE : 4 DISTANCE : 10
VILLE : 2 DISTANCE : 30
VILLE : 5 DISTANCE : 40
VILLE : 3 DISTANCE : 65
VILLE : 6 DISTANCE : 115

La suite de Fibonacci

Partie A

1) On considère la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$
et pour tout $n \geqslant 2$, $F_n = F_{n-1} + F_{n-2}$
Ecrire en Scilab une fonction qui, pour un entier n donné, calcule la valeur du terme F_n de la suite de Fibonacci : function [f]=Fibonacci(n);

Partie B

On désire pouvoir calculer exactement, pour $2 \leqslant n \leqslant 100$, la valeur d'un terme F_n de la suite de Fibonacci. La fonction précédente renvoie un résultat erroné à partir de $n = 79$.
Afin de calculer F_n, pour $79 \leqslant n \leqslant 100$, sans erreur de troncature ou d'arrondi, on définit l'algorithme suivant :
Cet entier est représenté par un tableau de taille 25 à raison d'un chiffre par élément. Si on note t une variable de type entier, alors $t(25)$ est le chiffre des unités de cet entier, $t(24)$ celui des dizaines, $t(23)$ celui des centaines, etc … Au delà du dernier chiffre de l'entier, les éléments du tableau sont nuls.
Ainsi F_{47} = 2971215073 est représenté par le tableau

0	0	0	0	…	0	2	9	7	1	2	1	5	0	7	3
1	2	3	4	…	15	16	17	18	19	20	21	22	23	24	25

Ce type permet donc de représenter tout entier naturel de l'intervalle $[0…(10^{26} - 1)]$.

2) Ecrire une fonction pour calculer la somme de deux nombres de type entier
function [f]=Somme(f1,f2)
Où la somme des entiers représenté par f1 et f2 est donné par la variable f3.
Exemple :

f1	0	0	0	0	0	…	0	8	1	7
+										
f2	0	0	0	0	0	…	1	4	6	4
=										
f	0	0	0	0	0	…	2	2	8	1

3) Ecrire une fonction [f]=Fibonacci2(n); qui construit le tableau t représentant le nombre de Fibonacci F_n.
4) Ecrire une fonction pour afficher à l'écran l'entier naturel représenté par un tableau t de type entier.
Par exemple, pour le tableau f de la question 2), cette procédure devra afficher 2281.

Partie C

5) Ecrire un programme permettant de saisir au clavier la valeur d'un entier n, si cet entier est inférieur à 79, d'utiliser la fonction Fibonacci de la partie A, si l'entier est entre 79 et 100, d'utiliser la fonction Fibonacci2 de la partie B et si l'entier est supérieur à 100, demander un autre entier.
Il suffira d'afficher la valeur du terme F_n ainsi obtenu.

Exercices sur les tableaux

I) Manipulations dans un tableau

On définit un tableau de taille 10, par exemple, contenant des entiers aléatoires entre 0 et 100.

1) Ecrire une fonction saisie fournissant un tel tableau.

2) Ecrire une fonction moyenne retournant la moyenne arithmétique des éléments du tableau.

3) Ecrire une fonction bornes retournant, lors d'une recherche simultanée, le plus petit et le plus grand des éléments.

4) Ecrire une fonction permuter effectuant une permutation circulaire de ses valeurs : chaque élément doit prendre la valeur du suivant, sauf le dernier qui doit prendre la valeur du premier.

5) Ecrire un programme qui, après avoir réalisée la procédure saisie, propose un menu permettant d'effectuer les modules 2) 3) ou 4) sur le tableau.

II) Tri par bulle

Principe : Le principe consiste à parcourir le tableau en comparant 2 à 2 des éléments voisins et en les permutant éventuellement. Après un balayage complet, le plus grand élément se trouve à l'indice n. On recommence l'opération sur les *n* - 1 éléments restants. On s'arrête lorsque la partie du tableau restant à trier est d'un seul élément ou lorsqu'un balayage complet du tableau n'a pas provoqué de permutation

Par exemple, pour trier le tableau	3 5 2 1 8 9 7 3	
on obtiendra successivement	3 2 1 5 8 7 3 9	5 et 9 "remontent"
	2 1 3 5 7 3 8 9	3 et 8 "remontent"
	1 2 3 5 3 7 8 9	2 et 7 "remontent"
	1 2 3 3 5 7 8 9	5 "remonte"

Ecrire en Scilab une fonction tri_bulle(t)

Le nom 'par bulle" vient du fait que la plus grande valeur remonte petit à petit vers le haut comme une bulle. Cette méthode est efficace pour un tableau presque trié.

III) Carré magique

On considère un tableau carré de n lignes et n colonnes. Un tel tableau est appelé **carré magique** lorsque les sommes des éléments sur une même ligne, une même colonne ou une même diagonale (principale) sont égales.

Test d'un tableau

1) Ecrire la fonction [t]=Saisie_Carre demandant la construction d'un tableau à l'utilisateur.

2) Ecrire la fonction [s]=Ligne(n,t) renvoyant la somme des éléments de la ligne n du carré t.

3) Ecrire de même une fonction Colonne et une fonction Diagonale (cette dernière doit avoir un paramètre supplémentaire pour déterminer laquelle des deux diagonales est concernée).

4) Ecrire une fonction booléenne Magique retournant *true* ou *false* selon que le carré t reçu en paramètre est magique ou non.

5) Ecrire un programme utilisant les modules précédents.

Création d'un carré magique

Voici un algorithme permettant la construction d'un carré magique d'ordre impair n = 2p + 1 (où le nombre d'éléments par côté est impair) et dans lequel les nombres entiers vont de 1 à n^2.

- L'élément juste au-dessous du centre est occupé par le chiffre 1 (t(p + 2,p + 1) := 1).
- Les éléments suivants : 2, 3, 4, ... n^2 sont placés dans les cases se trouvant à l'intersection de la ligne du dessous et de la colonne de droite.
- Arrivé au bord du carré, on poursuit l'opération à l'extrémité opposée en suivant la même règle. On identifie donc les colonnes n+k et k ainsi que les lignes n+k et k (k≠0)
- Si la case suivante est déjà remplie, le nombre est placé dans la même colonne, deux lignes en dessous.

Ex :

4	9	2
3	5	7
8	1	6

somme = 15

Ecrire un programme qui engendre puis affiche un carré magique au moyen de cet algorithme.