7 Les tableaux

7.1 Les tableaux à une dimension

Représentation d'un produit cartésien $A \times A \times \ldots \times A = A^k$ de taille k (qui peut être grande) et « homogène » (chaque composant est élément du même ensemble A).
On peut faire des tableaux de n'importe quel type, `entier_nat`, `personne`, tableau d'`entier_nat`, etc.

Type abstrait des tableaux à une dimension de réels (à titre d'exemple).
Nom du type : `tableau`.
Types abstraits importés : `booléen`, `entier_nat`, `réel`.
Opérations primitives :
`nouveau` : $\longrightarrow$ `tableau`
`écrire` : `entier_nat` $\times$ `réel` $\times$ `tableau` $\longrightarrow$ `tableau`
`Erreur_tableau` : $\longrightarrow$ `tableau`
`lire` : `entier_nat` $\times$ `tableau` $\longrightarrow$ `réel`
Axiomes des opérations primitives :

[Ax-1] $\texttt{lire}(i, \texttt{nouveau}) = \texttt{Erreur\_réel}()$

[Ax-2] **si** $i \notin \{0, 1, \ldots n-1\}$ **alors** $\texttt{écrire}(i, x, T) = \texttt{Erreur\_tableau}()$

[Ax-3] **si** $i \notin \{0, 1, \ldots n-1\}$ **alors** $\texttt{lire}(i, T) = \texttt{Erreur\_réel}()$

[Ax-4] Pour $i \in \{0, 1, \ldots n-1\}$ et $j \in \{0, 1, \ldots n-1\}$
$\texttt{lire}(i, \texttt{écrire}(j, x, T))$ = **si** $(i = j)$ **alors** x **sinon** $\texttt{lire}(i, T)$ **fin-si**

Notation algorithmique :
– Au lieu de $\texttt{lire}(i, T)$, on écrit $T[i]$;
– Au lieu de $T := \texttt{écrire}(i, x, T)$, on écrit $T[i] := x$;
– Au lieu de $T :=$ `nouveau`, on écrit $T :=$ nouveau tableau de N réels.

Recherche dans un tableau. Étant donné un tableau T de (p. ex.) réels, de taille k et un réel x, on cherche à savoir si x est élément du tableau.
Variantes de cette question : voir exercices (par exemple, indice de x dans T).
Dans le cas général : parcours séquentiel du tableau (complexité : $O(k)$).
Cas particulier : le tableau est rangé dans l'ordre croissant ($T[i] \leqslant T[i+1]$ pour $0 \leqslant i \leqslant k-2$). On peut appliquer le principe de la *dichotomie* :
– Soit $m = \lfloor k/2 \rfloor$: l'indice du « milieu du tableau ».
– On « coupe » le tableau en deux parties : U et V (U : partie du tableau de l'indice 0 à l'indice m, V : partie du tableau de l'indice $m+1$ à l'indice $k-1$).
– Si $T[m] > x$, alors, x ne peut pas être dans V : on recommence la procédure en ne testant que la partie U du tableau.
– Si $T[m] < x$, alors, x ne peut pas être dans U : on recommence la procédure en ne testant que la partie V du tableau.
– Si $T[m] = x$, alors on a trouvé x à l'indice m.
(Complexité : $O(\log k)$).

Exercice 1 *Écrire un algorithme itératif qui cherche dans un tableau T de N entiers l'indice de la première occurrence de la valeur a (et retourne -1 si a n'est pas dans T). Autrement dit, l'algorithme doit retourner i tel que :*

$$\textbf{\textit{Si}}\ \forall k \in \{1, 2, \ldots, N\}, T[k] \neq a\ \textbf{\textit{alors}}\ i = -1\ \textbf{\textit{sinon}}\ i = \min\{k \in \{1, 2, \ldots, N\} \mid T[k] = a\}$$

Exercice 2 *Modifier l'algorithme de l'exercice 1 pour chercher l'indice de la dernière occurrence de a dans T.*

Exercice 3 *Modifier l'algorithme de l'exercice 1 pour chercher le nombre d'occurrences de a dans T (le cardinal de $\{k \in \{1, 2, \ldots, N\} \mid T[k] = a\}$).*

Exercice 4 *Écrire l'algorithme donnant le tableau trié dans l'ordre croissant des indices des occurrences de a dans T.*

Exercice 5 *Reprendre chacun des exercices précédents avec l'hypothèse supplémentaire « T est trié de façon croissante ». Utiliser le principe de la dichotomie.*

Tri d'un tableau. On transforme le tableau T en un tableau contenant les mêmes éléments, mais rangés dans l'ordre croissant ou décroissant.
Après un tri croissant (resp., décroissant) du tableau T, on a : si $i < j$ alors $T[i] \leqslant T[j]$ (resp. $T[i] \geqslant T[j]$).
Beaucoup d'algorithmes en $O(n^2)$. Quelques algorithmes en $O(n \log n)$ (on ne peut pas faire mieux, théoriquement, sans s'appuyer sur des hypothèses sur le tableau ou sur le type des éléments).
Un algorithme en $O(n \log n)$ est le tri fusion (*merge sort*), parfois appelé *tri par interclassement*. Il s'appuie sur le principe « diviser pour régner » : pour résoudre un problème complexe, on le divise en problèmes plus simples. En l'occurrence, le problème est de trier un tableau de n éléments. On ramène ce problème au tri de deux tableaux de $n/2$ éléments (si n est pair[1]). En effet, il est relativement facile de faire un tableau trié à partir de deux tableaux triés, c'est ce qu'on appelle l'opération de *fusion* de deux tableaux. Pour trier les deux tableaux de $n/2$ éléments, on applique le même principe et ainsi de suite. Un algorithme récursif est beaucoup plus simple à mettre en œuvre qu'un algorithme itératif, en l'occurrence...
Voici ce que donne, sur un exemple, le tri fusion :

```
[3      5      2      6      12     4      8      15     7      0      1      5]
[3      5      2      6      12     4]     [8     15     7      0      1      5]
[3      5      2]     [6     12     4]     [8     15     7]     [0     1      5]
[3]     [5     2]     [6]    [12    4]     [8]    [15    7]     [0]    [1     5]
        [5]    [2]           [12]   [4]           [15]   [7]           [1]    [5]
                  fin de la division, début de la fusion
        [2     5]           [4     12]           [7     15]           [1     5]
[2      3      5]     [4     6      12]     [7     8      15]     [0     1      5]
[2      3      4      5      6      12]     [0     1      5      7      8      15]
[0      1      2      3      4      5      5      6      7      8      12     15]
```

Exercice 6 *Écrire un algorithme pour trier un tableau d'entiers naturels par valeurs croissantes ($T[i] \leqslant T[i+1]$) et évaluer sa complexité.*

Exercice 7 *Modifier l'algorithme de l'exercice 6 pour un tri par valeurs décroissantes.*

Exercice 8 *On se donne le type* `Point2D` *des points du plan (cf. exercice du chapitre précédent). Modifier l'algorithme de l'exercice 7 pour un tri d'un ensemble de points du plan, par distance euclidienne croissante à l'origine.*

Exercice 9 *Écrire l'algorithme du tri fusion d'un tableau de N entiers naturels.*

Exercice 10 *On considère un tableau T de N entiers et a, un entier. Écrire un algorithme pour la recherche d'un couple (i, j) tels que $T[i] + T[j] = a$ (si un tel couple n'existe pas, retourner $(-1, -1)$).*
Évaluer la complexité de l'algorithme précédent. Montrer qu'il existe un algorithme dont la complexité est en $O(n \log n)$.

7.2 Les tableaux à plusieurs dimensions

On peut définir des tableaux à plusieurs dimensions : $T[i, j]$, $T[i, j, k]$, etc.
Le type abstrait est très proche.
En fait, on peut se ramener à des tableaux à une dimension : par exemple, si T est un tableau à deux dimensions de réels, on peut le représenter par un tableau U dont chaque élément est lui-même un tableau de réels : $T[i, j] = U[i][j]$.

Exercice 11 *On considère le type des matrices à m lignes et n colonnes (tableaux à deux dimensions de réels).*
Écrire un algorithme pour la somme de deux matrices. Rappel : si $A = [a_{ij}]_{ij}$ et $B = [b_{ij}]_{ij}$, alors $A + B = C = [c_{ij}]_{ij}$, avec $c_{ij} = a_{ij} + b_{ij}$.
Écrire un algorithme pour le produit de deux matrices carrées (i.e., on supposera $m = n$). Rappel : si $A = [a_{ij}]_{ij}$ et $B = [b_{ij}]_{ij}$, alors $A \cdot B = C = [c_{ij}]_{ij}$, avec $c_{ij} = \sum_k a_{ik} b_{kj}$.

7.3 Les chaînes de caractères

Une chaîne de caractères est un tableau (à une dimension) de caractères.

---

[1] Si n est impair, on s'intéressera aux tris de deux tableaux de tailles $\frac{n-1}{2}$ et $\frac{n+1}{2}$.

Exercice 12 *On suppose que $\leqslant$ dénote l'ordre sur les caractères. En particulier, `'a'` $<$ `'b'` $< \ldots <$ `'z'`. L'ordre lexicographique sur les chaînes de caractères est l'ordre utilisé pour classer deux mots d'un dictionnaire. Par exemple, les mots suivants sont dans l'ordre lexicographique croissant : `"abricot"`, `"arbre"`, `"ours"`, `"ourse"`, `"ourses"`.*
Écrire un algorithme qui teste l'ordre lexicographique de deux chaînes de caractères : cette fonction doit retourner un booléen qui vaut `vrai` ssi la première chaîne précède strictement la deuxième.

Exercice 13 *En vous appuyant sur l'algorithme décrit dans la question précédente, écrire un algorithme qui trie un tableau de chaînes de caractères selon l'ordre lexicographique croissant.*

7.4 Implantation des tableaux et des chaînes de caractères en C

7.4.1 Les tableaux à une dimension, en C

On veut manipuler le type des tableaux d'éléments de type `double`. Le type de ces tableaux est `double[n]` où `n` est la taille du tableau. L'exemple ci-dessous montre une manipulation de tableaux :

```
/*****************************************/
/* exemple_tableaux.c                    */
/* Exemple de manipulation de tableaux en C */
/* auteur : Jean Lieber                  */
/* version : 2                           */
/* date : 07/05/11                       */
/*****************************************/

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

/* Affichage d'un tableau tab de taille éléments de type double */
void afficher_tableau (size_t taille, double tab[taille])
{
  size_t i ;
  printf("[") ;
  for (i = 0 ; i < taille ; i = i +1)
    {
      printf ("%f%s", tab[i],
              (i < taille - 1) ? " " : "") ;
    }
  printf ("]") ;
}

/* Nombre d'éléments > 0 du tableau tab contenant taille doubles */
size_t nb_positifs (size_t taille, double tab[taille])
{
  size_t i, cpt ;
  cpt = 0 ;
  for (i = 0 ; i < taille ; i = i + 1)
    {
      if (tab[i] > 0)
        {
          cpt = cpt + 1 ;
        }
    }
  return cpt ;
}

/* tab est un tableau de taille doubles contenant taillePos éléments
   strictement positifs. tabPos est un tableau de taillePos éléments.
   L'appel à cette procédure fait que taillePos contiendra le tableau
   des éléments > 0 de tab, dans l'ordre de leurs indices croissants */
void positifs (size_t taille, double tab[taille],
               size_t taillePos, double tabPos[taillePos])
{
  size_t i, iP ;
  iP = 0 ;
```

```
  for (i = 0 ; i < taille ; i = i + 1)
    {
      if (tab[i] > 0)
        {
          tabPos[iP] = tab[i] ;
          iP = iP + 1 ;
        }
    }
}

void tests ()
{
  size_t n, np ;
  n = 6 ;
  double t[6] ;
  t[0] = 3. ; t[1] = -4. ; t[2] = 0. ;
  t[3] = 8. ; t[4] = -6. ; t[5] = 4.1 ;
  afficher_tableau (n, t) ;
  np = nb_positifs (n, t) ;
  printf ("\n%u éléments strictement positifs :\n", np) ;
  double tP[np] ;
  positifs (n, t, np, tP) ;
  afficher_tableau (np, tP) ;
  printf ("\n") ;
}

int main ()
{
  tests () ;
}
```

Remarquons que, quand on fait passer en paramètre un tableau, on indique préalablement sa taille (le nombre d'éléments qu'il contient). En effet, il n'y a pas de moyen, étant donné un tableau `t`, de connaître cette taille...
`size_t` est un type d'entiers naturels qui permet de représenter la taille d'un tableau (on l'utilisera, de préférence à `unsigned int`).

7.4.2 Les tableaux à plusieurs dimensions n'existent pas en C

Les tableaux à plusieurs dimensions n'existent pas en C, mais on s'appuie sur la remarque de la section 7.2 : un tableau à deux dimensions de réels se représente par un tableau de tableaux de réels. Le nom du type correspondant est donc `double[m][n]` (où `m` et `n` sont les nombres de lignes et de colonnes du tableau à deux dimensions). L'exemple ci-dessous est un exemple de manipulation de tels tableaux de tableaux de réels (qu'on assimile donc à des matrices de réels).

```
/*****************************************/
/* exemple_tableaux_2dimensions.c        */
/* Exemple de manipulation de matrices en C */
/* auteur : Jean Lieber                  */
/* version : 2                           */
/* date : 07/05/11                       */
/*****************************************/

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

void afficher_matrice (size_t nbLignes, size_t nbColonnes, double M[nbLignes][nbColonnes])
{
  size_t i, j ;
  for (i = 0 ; i < nbLignes ; i = i + 1)
    {
      for (j = 0 ; j < nbColonnes ; j = j + 1)
        printf ("%f  ", M[i][j]) ;
      printf ("\n") ;
    }
}

/* L'effet de cette procédure est de mettre dans C la somme des matrices
```

```
   A et B, à condition que les contraintes de nombres de lignes et de
   colonnes soient respectées */
void somme_matrices(size_t nbLignesA, size_t nbColonnesA, double A[nbLignesA][nbColonnesA],
                    size_t nbLignesB, size_t nbColonnesB, double B[nbLignesB][nbColonnesB],
                    size_t nbLignesC, size_t nbColonnesC, double C[nbLignesC][nbColonnesC])

{
  size_t i, j ;
  if (nbLignesA != nbLignesB || nbColonnesA != nbColonnesB ||
      nbLignesA != nbLignesC || nbColonnesA != nbColonnesC)
    {
      printf ("Erreur dans somme_matrices !\n") ;
      exit(EXIT_FAILURE) ;
    }
  for (i = 0 ; i < nbLignesA ; i = i + 1)
    {
      for (j = 0 ; j < nbColonnesA ; j = j + 1)
        {
          C[i][j] = A[i][j] + B[i][j] ;
        }
    }
}

void tests ()
{
  size_t i, j ;
  size_t m, n ;
  m = 2 ; n = 3 ;
  double A[m][n], B[m][n], C[m][n] ;
  for (i = 0 ; i < m ; i = i + 1)
    {
      for (j = 0 ; j < n ; j = j + 1)
        {
          A[i][j] = i + j ;
          B[i][j] = 10 * (i + j) ;
        }
    }
  somme_matrices (m, n, A, m, n, B, m, n, C) ;
  printf ("A : \n") ; afficher_matrice (m, n, A) ;
  printf ("B : \n") ; afficher_matrice (m, n, B) ;
  printf ("A + B : \n") ; afficher_matrice (m, n, C) ;
}

int main ()
{
  tests () ;
}
```

Exercice 14 *Ajouter au programme ci-dessus une fonction qui calcule le produit de deux matrices.*
Tester cette fonction.

Exercice 15 *On s'intéresse aux matrices* 2×2*. Compléter le programme ci-dessus par les fonctions et la procédure suivantes (dont vous devrez donner les signatures) :*

- `creer_matrice_2x2` *qui permet de créer une nouvelle matrice* 2×2*;*
- `liberer_matrice_2x2` *qui permet de désallouer la mémoire d'une matrice* 2×2*;*
- `somme_matrices_2x2` *qui permet de faire la somme de deux matrices* 2×2*;*
- `produit_matrices_2x2` *qui permet de faire le produit de deux matrices* 2×2*;*
- `determinant_matrice_2x2` *qui permet de calculer le déterminant de deux matrices* 2×2*. On rappelle que le déterminant de la matrice* $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ *est* $ad - bc$*.*
- `afficher_matrice_2x2` *qui permet de faire l'affichage d'une matrice* 2×2*.*

Remarque : *Vous vous appuierez évidemment sur les fonctions et la procédure déjà définies.*
Tester ces fonctions et cette procédure.

7.4.3 Les chaînes de caractères

Une chaîne de caractère en C est un tableau dont les éléments sont des caractères, i.e., sont de type `char`. Autrement dit, le type des chaînes de caractères est `char*`. En fait, on ne considère d'une chaîne de caractères que les éléments situés avant le caractère qui est noté `'\0'`.
Pour construire une chaîne, on peut utiliser la construction `"..."`. Ainsi, `"abc"` est un tableau de 4 caractères : `'a'`, `'b'`, `'c'` et `'\0'`, aux indices respectifs 0, 1, 2 et 3.
Ainsi, si `ch` est une variable de type `char*`, les instructions

```
ch = (char*)malloc(5 * sizeof(char)) ;
ch[0] = 'c' ; ch[1] = 'h' ; ch[2] = 'a' ; ch[3] = 't' ; ch[4] = '\0' ;
```

équivalent à

```
ch = "chat" ;
```

La librairie `string.h` contient plusieurs fonctions classiques sur les chaînes de caractères, en particulier :
- `strlen(ch)` donne la *longueur* de la chaîne de caractères, c'est-à-dire, le nombre de caractères avant le caractère `'\0'`. À noter que cette longueur est toujours inférieure à la taille du tableau `ch`, lequel contient le caractère `'\0'` et éventuellement d'autres caractères après.
- `strcmp` permet de comparer deux chaînes de caractères pour l'ordre lexicographique (l'ordre des mots du dictionnaire). Cette fonction prend en paramètres deux `char* ch1` et `ch2` et retourne un `int` qui est
  - Négatif si `ch1` précède `ch2` (par exemple, `strcmp ("chat", "lapin")` est négatif) ;
  - Positif si `ch1` suit `ch2` (par exemple, `strcmp ("fourmi", "four")` est positif) ;
  - Nul si les chaînes de caractères sont égales (par exemple, `strcmp ("un ours", "un ours")` est nul).
- `strcpy` effectue une copie (un clone) de la chaîne en deuxième argument dans la chaîne en premier argument (qui doit être suffisamment grande pour accueillir les caractères de la deuxième chaîne).
- `strcat` effectue une concaténation. Ainsi, si `ch` représente la chaîne `"abc"` et a une taille d'au moins 7 éléments, alors l'instruction `strcat(ch, "def")` aura pour conséquence que `ch` représentera la chaîne de caractères `"abcdef"`.

Exercice 16 *Proposer une (ré-)implantation des fonctions* `strlen`*,* `strcmp`*,* `strcpy` *et* `strcat`*.*
