DURÉE: 2h, AUCUN DOCUMENT AUTORISÉ

REMARQUES:
- Les exercices sont indépendants et peuvent être réalisés dans l'ordre voulu.
- Dans l'implémentation d'une méthode, on pourra utiliser n'importe quelle autre méthode définie auparavant même si celle-ci n'a pas été implémentée.
- Dans toutes les implémentations que vous écrivez, pensez à respecter le guide de syntaxe pour la programmation (règles de nommage, présentation des blocs, *etc.*).

EXERCICE 1: QUESTION DE COURS (4 POINTS)

1. Écrire une fonction `int* minDist( int** points, int nbPoints, int* minD )` qui à partir d'un tableau de points de taille $2\times$`nbPoints` calcule la plus petite distance `minD` entre2 points et renvoie les indices des 2 points correspondant à cette distance.
   Quelle est la complexité de cette fonction ?

Solution: (2,5 points) : 1,5 points pour la fonction, 1 point pour la compléxité ($\mathcal{O}(n^2)$).
Attention : l'allocation dynamique de `res` est obligatoire ! Sinon `res` est détruit à la fin de la fonction et on perd le résultat

```
int* minDist( int** points, int nbPoints, int* minD ) {
  *minD = abs( points[0][0] - points[0][1] ) + abs( points[1][0] - points[1][1]
     ↪  );
  int* res = calloc( 2, sizeof( int* ) );
  res[1] = 1;
  int i, j;
  for( i = 0; i < nbPoints - 1; i++ ) {
    for( j = i + 1; j < nbPoints; j++ ) {
      int dist = abs( points[0][i] - points[0][j] ) + abs( points[1][i] -
         ↪ points[1][j] );
      if( dist < *minD ) {
        *minD = dist;
        res[0] = i;
        res[1] = j;
      }
    }
  }
  return res;
}
```

2. Écrire le programme principal qui lit un nuage de points puis affiche les 2 points les plus proches dans le nuage.
   On supposera qu'une fonction `int** lireNuageDePoints( int* nbPoints )` existe et permet de lire un tableau alloué dynamiquement de $2\times$`nbPoints` points. Cette fonction pourra donc être utilisée sans être codée.

Solution: (1,5 points) : 1 point pour les free à la fin de la fonction.

```
int main( ) {
  int nbPoints, minD;
  int** points = lireNuageDePoints( &nbPoints );
  int* res = minDist( points, nbPoints, &minD );
  printf( "Distance minimale: %d entre les points (%d, %d) et (%d, %d)\n", minD
     ↪ , points[0][res[0]], points[1][res[0]], points[0][res[1]], points[1][
     ↪ res[1]] );

  free( points[0] );
  free( points[1] );
  free( points );
  free( res );

  return 0;
}
```

EXERCICE 2: EMISSIONS DE RADIO (5 POINTS)

Une station de radio souhaite informatiser son système de planning des émissions de radio. Chaque émission de radio dispose d'un ou plusieurs créneaux durant lesquels un ou plusieurs animateurs peuvent accueillir des invités. Certaines émissions étant enregistrées, la radio dispose aussi de plusieurs studios d'enregistrement sachant qu'une émission est toujours produite dans le même studio.

On représentera les personnes par leurs nom et prénom ainsi que leur statut (animateur ou invité). Un créneau correspondra à une heure de début et de fin d'émission ; une heure sera conservée sous la forme d'un tableau de 2 entiers. Le nom et l'étage où se situe le studio seront aussi conservés.

1. Donner la déclaration des structures permettant la résolution de ce problème.

Solution: (5 points) $\simeq$ 1 point par structure.

```c
typedef struct {
  char nom[30];
  char prenom[30];
  int statut;  // 0: animateur, 1: invite
} Personne;

typedef struct {
  int debut[2];
  int fin[2];
} Creneau;

typedef struct {
  char nom[30];
  int etage;
} Studio;

typedef struct {
  ListeSC personne;  // Animateurs et invites
  Creneau horaire;
  Studio lieu;
  int estLive;  // 0: enregistre, 1: live
} Emission;

typedef struct {
  ListeSC emissions;
} Planning;
```

- Comme on ne connait pas *a priori* le nombre d'animateurs et d'invités, il faut utiliser une liste (pour minimiser l'impact mémoire).
- Tolérer la non déclaration de la structure `Planning` (on pourrait juste avoir une variable de type liste).
- Attention à l'ordre dans lequel les structures sont déclarées : `Emission` en (avant-)dernier puisqu'elle utilise toutes les autres structures.

EXERCICE 3: MATRICE CREUSE (12 POINTS)

Une matrice creuse (ou sparse en anglais) est une matrice contenant beaucoup de zéros. Pour réduire la place occupée par ce type de matrice en mémoire, on propose de réaliser une librairie de manipulation et gestion de matrice creuse. Pour cela, on représentera chaque élément non nul de la matrice par ses coordonnées (x, y) ainsi que par sa valeur. La matrice creuse sera ensuite stockée dans une structure où l'on conservera les dimensions de la matrice ainsi qu'une liste de coefficients.

Pour réaliser ce projet, on dispose de 4 fichiers : `coefMatrice.h` et `coefMatrice.c` qui gèrent un coefficient de la matrice ainsi que `matSparse.h` et `matSparse.c` qui définissent la matrice creuse. Les structures ci-dessous sont définies dans les différents fichiers header.

```c
// Dans coefMatrice.h
typedef struct {
  int x, y;    // Coordonees
  double val;  // Valeur
} CoefMatrice;
```

```c
// Dans matSparse.h
typedef struct {
  int dimL, dimC;  // Dimensions de la matrice
  ListeSC* coef;   // Coefficients non nuls
} MatSparse;
```

On supposera qu'une librairie `matFull.h` et `matFull.c` permettant la manipulation de matrices usuelles (*i.e.* "pleine") existe et a déjà été codée. On pourra donc utiliser ses fonctions sans les réimplémenter. Le contenu de cette librairie est donné ci-dessous.

```c
typedef struct {
  int dimL, dimC;  // Dimensions de la matrice
  double** coef;   // Coefficients
} MatFull;

// Creation d'une matrice pleine (avec allocation dynamique)
MatFull createMatFull( int dimL, int dimC );
// Destruction d'une matrice pleine
void freeMatFull( MatFull* matS );
// Lecture d'une matrice pleine depuis un fichier texte
MatFull readMatFull( char* fName );
// Ecriture d'une matrice pleine dans un fichier texte
void writeMatFull( char* fName, MatFull matS );
```

1. **Coefficient de la matrice (`coefMatrice.c`)**
   (a) Donner l'implémentation de la fonction `CoefMatrice createCoefMatrice( int x, int y, double val )` qui créé et initialise un coefficient de la matrice.

   Solution: (0.5 point)
   ```c
   CoefMatrice createCoefMatrice( int x, int y, double val ) {
     CoefMatrice mC = {x, y, val};
     return mC;
   }
   ```

   (b) Une fonction de destruction est-elle nécessaire ? Si oui, donner son implémentation.

   Solution: (0.5 point)
   Comme il n'y a pas d'allocation dynamique, il n'est pas nécessaire de prévoir une fonction de destruction.

   (c) Écrire une fonction `void printCoefMatrice( CoefMatrice mC )` qui affiche un coefficient sous la forme : `(x, y) : val`.

   Solution: (0.5 point)
   ```c
   void printCoefMatrice( CoefMatrice mC ) {
     printf( "(%d, %d): %lf\n", mC.x, mC.y, mC.val );
   }
   ```

2. **Liste chainée (`listeSC.c`) :** Compléter les fonctions `void freeData(void *d)` et `void afficherData(void *d)` permettant la destruction et l'affichage d'un coefficient de matrice creuse.

   Solution: (1 point)
   ```c
   void freeData( void *d ) {
     // Rien a faire (pas d'allocation dynamique)
   }

   void afficherData( void *d ) {
     CoefMatrice* mC = (CoefMatrice*)d;
     printCoefMatrice( *mC );
   }
   ```

3. **Matrice creuse (`matSparse.c`) :**
   Initialisation et destruction :
   (a) Donner l'implémentation d'une fonction `MatSparse createMatSparse( int dimL, int dimC )` permettant de créer une matrice creuse. La liste de coefficients sera vide.

   Solution: (1 point)

```c
MatSparse creerMatSparse( int dimL, int dimC ) {
  MatSparse matS = {dimL, dimC};  // Creation de la structure
  matS.coef = creerListe( sizeof( CoefMatrice ) );
  return matS;
}
```

(b) Donner l'implémentation d'une fonction `void freeMatSparse( MatSparse* matS )` permettant de détruire une matrice creuse.

Solution: (1 point)

```c
void freeMatSparse( MatSparse* matS ) {
  freeListe( matS->coef );
  matS->dimL = 0;
  matS->dimC = 0;
}
```

Lecture et écriture : Les matrices creuses sont sauvegardées dans des fichiers textes. La 1ère ligne du fichier contient 2 entiers indiquant les dimensions de la matrice. Chaque ligne suivante contient un coefficient de la matrice sous la forme `x y val`. Un exemple de début de fichier est donné ci-contre

```
10 10
0 2 9.618981
0 6 1.066528
1 5 2.598704
...
```

Solution: 1 point pour la bonne gestion des fichiers dans les 2 questions suivantes (*i.e.* ouverture, test et fermeture)

(c) Donner l'implémentation d'une fonction `MatSparse readMatSparse( char* fName )` permettant de lire une matrice creuse dans un fichier texte.

Solution: (1 point)

```c
MatSparse readMatSparse( char* fName ) {
  MatSparse matS;
  // Ouverture en lecture d'un fichier texte
  FILE *fTxt = fopen( fName, "r" );
  if( fTxt != NULL ) {  // On s'assure que le fichier a bien ete ouvert
    int dimL, dimC;
    fscanf( fTxt, "%d %d\n", &dimL, &dimC );  // Lecture du nombre de lignes
       ↪ et de colonnes
    matS = creerMatSparse( dimL, dimC );  // Creation de la matrice
    // Lecture des coefficients de la matrice
    int x, y;    // Coordonnees
    double val;  // Valeur
    while( fscanf( fTxt, "%d %d %lf\n", &x, &y, &val ) == 3 ) {
      CoefMatrice mC = createCoefMatrice(x, y, val);    // Nouvel element
      ajout( matS.coef, &mC, 2 ); // Ajout a la fin
    }
    fclose( fTxt );    // Fermeture du fichier
  }
  return matS;
}
```

(d) Donner l'implémentation d'une fonction `void writeMatSparse( char* fName, MatSparse matS )` permettant d'écrire une matrice creuse dans un fichier texte.

Solution: (1 point)

```c
void writeMatSparse( char* fName, MatSparse matS ) {
  // Ouverture en ecriture d'un fichier texte
  FILE *fTxt = fopen( fName, "w" );
  if( fTxt != NULL ) {  // On s'assure que le fichier a bien ete ouvert
```

```
    fprintf( fTxt, "%d %d\n", matS.dimL, matS.dimC );  // Ecriture du nombre
       ↪ de lignes et de colonnes
    // Ecriture des coefficients de la matrice
    for( matS.coef->current = matS.coef->root; hasNext( matS.coef ); getNext
       ↪ ( matS.coef ) ) {
      CoefMatrice* mC = ( (CoefMatrice*) matS.coef->current->data );
      fprintf( fTxt, "%d %d %lf\n", mC->x, mC->y, mC->val );
    }
    fclose( fTxt );    // Fermeture du fichier
  }
}
```

Manipulations diverses :

(e) Écrire une fonction `void printMatSparse( MatSparse matS )` permettant l'affichage d'une matrice creuse à la console. On affichera les dimensions de la matrice, le nombre de coefficients non-nuls puis chaque coefficient. Un exemple d'affichage est donné ci-contre.

```
Matrice creuse (10x10)
13 coefficients non nuls
0: (0, 2): 9.618981
1: (0, 6): 1.066528
2: (1, 5): 2.598704
...
```

Solution: (0.5 point)

```
void printMatSparse( MatSparse matS ) {
  printf( "Matrice creuse (%d, %d)\n", matS.dimL, matS.dimC );
  printf( "%d coefficients non nuls\n", matS.coef->length );
  afficherListe( matS.coef );
}
```

(f) Écrire une fonction `double findCoef( MatSparse matS, int x, int y )` qui cherche si le coefficient en position (x, y) est présent dans la liste de coefficients. S'il est présent, la fonction renverra sa valeur ; sinon elle renverra 0.

Solution: (1 point)

```
double findCoef( MatSparse matS, int x, int y ) {
  double res = 0;
  matS.coef->current = matS.coef->root;
  while( hasNext( matS.coef ) && (res == 0) ) {
    CoefMatrice* mC = ( (CoefMatrice*)matS.coef->current->data );
    if( equalsCoefMatrice( *mC, x, y ) )
      res = mC->val;
    getNext( matS.coef );
  }
  return res;
}
```

(g) Écrire une fonction `MatSparse makeMatSparse( MatFull matF )` qui créé une matrice creuse à partir d'une matrice "pleine".

Solution: (1 point)

```
MatSparse makeMatSparse( MatFull matF ) {
  MatSparse matS = creerMatSparse( matF.dimL, matF.dimC );
  int i, j;
  for( i = 0; i < matF.dimL; i++ )
    for( j = 0; j < matF.dimC; j++ )
      if( matF.coef[i][j] != 0 ) {  // On cherche les coefficients non nuls
        CoefMatrice mC = createCoefMatrice( i, j, matF.coef[i][j] );
        ajout( matS.coef, &mC, 2 );  // On ajoute a la matrice
      }
  return matS;
}
```

4. **Fichier principal :** donner le code du programme principal. On déclarera en constante le nom des fichiers contenant la matrice pleine d'entrée `"file/matFull.txt"` et la matrice creuse de sortie `"file/matSparse.txt"`. Après avoir lu la matrice pleine, on la transformera en matrice creuse que l'on affichera et écrira dans le fichier de sortie.

Solution: (2 points) 0,5 point pour l'inclusion des bibliothèques, 0,5 point pour la définition de constante (`#define`), 0,5 point pour la destruction des matrices à la fin du programme.

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

#include "coefMatrice.h"
#include "listeSC.h"
#include "matSparse.h"
#include "matFull.h"

#define MAT_IN "file/matFull.txt"
#define MAT_OUT "file/matSparse.txt"

int main( ) {
  MatFull matF = readMatFull( MAT_IN );
  printMatFull( matF );

  MatSparse matS = makeMatSparse( matF );
  printMatSparse( matS );
  writeMatSparse( MAT_OUT, matS );

  freeMatFull( &matF );
  freeMatSparse( &matS );
  return 0;
}
```