ASI3 Mardi 14 Janvier 2020

Algorithmique et Bases de la programmation

Durée : *1h55*
Documents autorisés : **AUCUN**

L'objectif de cet examen pratique est de développer une bibliothèque C de manipulation de matrices.

Informations

Avant de commencer l'examen

Connectez vous à la machine et suivez les instructions suivantes :

— copiez l'archive `C.tar.gz` qui se trouve dans `/opt/files` dans votre répertoire (commande `cp`) ;

— après l'avoir décompressée (commande `tar` avec les options `zxvf`), renommez le répertoire en lui donnant comme nom : 'EXAM-ALGO-' (les tirets sont ceux du signe moins) suivi de votre nom et prénom, sans espace, sans caractères accentués et en minuscule sauf pour les premières lettres du prénom et du nom. Par exemple, si « Paul Du Villaré » devait passer cet examen, il nommerai son répertoire EXAM-ALGO-DuvillarePaul.

À la fin de l'examen

Cinq minutes avant la fin de l'examen :

— assurez vous que les fichiers sont bien au format ISO-8859 (c'est par défaut le cas) ;

— **assurez vous qu'il n'y a aucun accent dans les commentaires que vous auriez pu ajouter ;**

— assurez vous que votre projet compile ;

— créez une archive de votre répertoire (commande `tar` avec les options `zcvf`), que vous nommerez comme le répertoire (avec le préfixe `.tar.gz`) ;

— déposez sur moodle votre archive (cours « algorithmique avancée et programmation C », section « examens pratiques »).

Attention :

— le lien de dépôt est actif uniquement 10 minutes, si vous dépassez le délai, votre note sera 0 ;

— on ne peut déposer qu'une seule fois le projet sur moodle.

Quelques conseils

Pour réussir votre examen suivez les conseils suivants :

— respectez bien les consignes données ci dessus. **Le fait de ne pas les respecter vous fera perdre 2 points sur la note finale** ;

— par défaut le projet compile et s'exécute. Après le développement de chaque fonction C, compilez votre projet (`make`) et ne passez à la fonction suivante que lorsque votre projet compile : **le fait de rendre un projet qui ne compile pas ou ne s'exécute pas (par exemple à cause d'un *segmentation fault*), divisera par deux votre note finale** ;

— la note finale sera fonction :

  — du nombre de tests unitaires qui seront valides ;

  — du nombre de *Warning* (compilation avec l'option `-Wall`) qui seront affichés ;

  — de la quantité (nombre de fonctions développées correctement), de la qualité (pas de copier/coller, fonctions algorithmiquement performantes) et de la lisibilité du code (indentation et identifiants significatifs).

1 La bibliothèque `libmatrice.a`

La bibliothèque `libmatrice.a` propose les fonctions suivantes :

— `M_Matrice M_matriceValeurs(unsigned int nbLignes, unsigned int nbColonnes, double valeurs[])` qui permet d'obtenir une matrice ayant *nbLignes* lignes, *nbColonnes* colonnes avec des valeurs initialisées par *valeurs*, telles que les valeurs de celles ci sont rangées dans ce tableau en ligne.

Ainsi pour obtenir la matrice suivante :

$$\begin{pmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{pmatrix} \tag{1}$$

on appellera cette fonction avec :

  — la valeur `2` comme paramètre effectif associé au paramètre formel `nbLignes` ;

  — la valeur `3` comme paramètre effectif associé au paramètre formel `nbColonnes` ;

  — le tableau `[1., 2., 3., 4., 5., 6.]` comme paramètre effectif associé au paramètre formel `valeurs[]`.

— `M_Matrice M_matriceValeur(unsigned int nbLignes, unsigned int nbColonnes, double valeur)` qui permet d'obtenir une matrice ayant *nbLignes* lignes, *nbColonnes* colonnes dont toutes les valeurs valent *valeur* ;

— `M_Matrice M_matriceZero(unsigned int nbLignes, unsigned int nbColonnes)` qui permet d'obtenir une matrice ayant *nbLignes* lignes, *nbColonnes* colonnes dont toutes les valeurs valent 0 ;

— `M_Matrice M_matriceId(unsigned int taille)` qui permet d'obtenir une matrice identité de taille *taille* ;

— `M_Matrice M_copier(M_Matrice m)` qui permet de copier une matrice m. Attention la matrice obtenue est égale à m mais pas identique ;

— `unsigned int M_nbLignes(M_Matrice m)` qui permet d'obtenir le nombre de lignes d'une matrice m ;

— `unsigned int M_nbColonnes(M_Matrice m)` qui permet d'obtenir le nombre de colonnes d'une matrice m ;

— `double M_obtenirValeur(M_Matrice m, unsigned int ligne, unsigned int colonne)` qui permet d'obtenir la valeur stockée à la position *(ligne,colonne)* d'une matrice m. Par exemple `M_obtenirValeur(m, 2, 1)` avec `m` représentant de la matrice (1) retournerait `4`.

— `void M_fixerValeur(M_Matrice *pm, unsigned int ligne, unsigned int colonne, double valeur)` qui permet de fixer à *valeur* la valeur de la position *(ligne,colonne)* d'une matrice référencée par le pointeur `pm` ;

— `int M_egale(M_Matrice m1, M_Matrice m2)` qui permet de savoir si deux matrices *m1* et *m2* sont égales (même nombre de lignes, même nombre de colonnes et même valeurs à `PRECISION` près [1]) ;

— `void M_transposer(M_Matrice *pm)` qui permet de transposer la matrice référencée par le pointeur *pm*. Pour rappel la transposée de la matrice (1) est la matrice :

$$\begin{pmatrix} 1. & 4. \\ 2. & 5. \\ 3. & 6. \end{pmatrix}$$

— `void M_liberer(M_Matrice *pm)` qui permet de libérer la mémoire allouée dynamiquement d'une matrice lorsqu'on ne l'utilise plus.

Le type `M_Matrice` est défini de la façon suivante :

```
typedef struct {
  unsigned int nbLignes;
  unsigned int nbColonnes;
  double *valeurs;
} M_Matrice;
```

Les deux premiers attributs permettent de stocker le nombre de lignes et le nombre de colonnes de la matrice. L'attribut `valeurs` pointe vers la zone mémoire allouée dynamiquement qui stocke linéairement les valeurs de la matrice (ligne par ligne). Ainsi la valeur de la matrice stockée aux coordonnées mathématiques i, j (avec $i \in 1..nbLignes$ et $j \in 1..nbColonnes$) est stockée à la position $(i-1) * nbColonnes + j - 1$ de cette zone mémoire. Ce calcul est réalisé par la fonction privée `M_position`.

Enfin, afin que tous les tests unitaires se déroulent par défaut convenablement (pas de *segmentation fault*), si nécessaire chaque fonction publique appelle la fonction privée `void M_init(M_Matrice *pm, unsigned int nbLignes, unsigned int nbColonnes)` qui initialise ces attributs avec les valeurs `0`, `0` et `NULL`.

Pour valider le développement de cette bibliothèque des suites de tests unitaires ont été développées dans le programme `matriceTU.c` (programme utilisant le *framework CUnit*). La première suite est une suite de tests unitaires en « boite noire » c'est-à-dire que les tests n'utilisent pas la façon dont sont représentées les matrices. La deuxième suite est une suite de tests unitaires en « boite blanche » c'est-à-dire que les tests vérifient que la structure de données est bien utilisée comme définie dans l'énoncé.

2 Travail à réaliser

Complétez le fichier `matrice.c` de façon à ce que le maximum de tests unitaires fonctionnent (`test/matriceTU`). Nous vous conseillons de développer les fonctions suivant leurs ordres d'apparition dans le programme C.

1. attention en C on considère que deux *float* ou *double* sont égaux si la valeur absolue de leur différence est plus petite qu'une certaine précision, voir la fonction C `fabs`.