PROGRAMMATION C

TP 1 – 18 septembre 2025 – durée : 2h40

Matrices carrées

TP 1 de Programmation C

Ce TP reprend le 1er TP noté 2024–25, qui durait 1h20 (le barème a été laissé). Il porte sur les matrices carrées. Une partie a été ajoutée, pour calculer le déterminant d'une matrice en se servant uniquement de la définition.

1 Objectif du TP

On veut coder des algorithmes manipulant des matrices carrées d'entiers, c'est-à-dire des tableaux carrés d'entiers. On appelle **dimension** d'une telle matrice son nombre de lignes (ou de colonnes). Par exemple, la matrice suivante est de dimension 4.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

L'objectif du TP est de coder :

- Deux implémentations simples mais différentes d'opérations de base sur les matrices carrées.
- Un fichier de test, permettant de créer des matrices, d'en afficher et d'en parcourir selon un ordre précis.
- Un fichier implémentant le calcul du déterminant d'une matrice basé sur sa définition, rappelée plus loin.

Chacun de ces objectifs est détaillé plus bas, et dans les fichiers d'en-tête matrix.h, basic.h et determinant.h.

1.1 Ce qui est fourni

1.1.1 Fichiers sources C et fichiers d'en-tête

> ⚠ **Important** : ne modifiez que les fichiers .c.

Le fichier main.c est destiné aux tests. Vous pouvez le modifier (des tests y sont déjà écrits, les résultats attendus sont indiqués dans le fichier results.txt). Les fichiers à compléter sont **uniquement** matrix1.c, matrix2.c, basic.c, determinant.c. Les autres fichiers (en-têtes .h, fichier Makefile) **ne doivent pas être modifiés**.

1.1.2 Le fichier `Makefile`

Le Makefile est volontairement exigeant. Par défaut, les *warnings* sont traités comme des erreurs. Le code doit se compiler **sans avertissement** par `make -B CFLAGS="-DVERSION=1"` et `make -B CFLAGS="-DVERSION=2"`. Le Makefile construit l'exécutable `tp`, exécutable sous le shell par la commande `./tp`.

Vous devez coder **deux versions** d'une micro-bibliothèque, la première dans matrix1.c, la seconde dans matrix2.c. Le Makefile fourni permet de créer l'exécutable avec l'une ou l'autre des bibliothèques.

1. On peut construire l'exécutable `tp` en traitant les warnings comme des erreurs (l'option `-Werror` impose d'éviter tout warning) et avec des options `-O0 -g` permettant d'utiliser les débogueurs `gdb` (ou `ddd`) et `valgrind` :

```
# Pour compiler avec la version de matrix1.c
$ make -B  CFLAGS="-DVERSION=1"

# Pour compiler avec la version de matrix2.c
$ make -B CFLAGS="-DVERSION=2"
```

La macro `VERSION` détermine l'implémentation à utiliser. Par défaut, si on tape `make`, c'est matrix1.c qui est utilisée car la macro vaut initialement `1`. Vous pouvez modifier cela en changeant `#define VERSION 1` en `#define VERSION 2` dans matrix.h (c'est la seule exception permise à l'édition des fichiers .h).

2. On peut construire l'exécutable tp sans l'option `-Werror` et en mode optimisé `-O3`, sans utilisation possible de `gdb`. Dans ce mode, vous voyez un avertissement pour chaque paramètre de fonction non utilisé.

```
# Pour compiler avec la version de matrix1.c
$ make -B  CFLAGS="-DVERSION=1" no-error-on-warning

# Pour compiler avec la version de matrix2.c
$ make -B CFLAGS="-DVERSION=2" no-error-on-warning
```

3. On peut nettoyer les fichiers produits par la compilation :

```
$ make clean
```

2 Fonctions à écrire

Il y a 4 parties à écrire, chacune dans un fichier : matrix1.c, matrix2.c, basic.c et determinant.c (voir les Sections 2.1, 2.2, 2.3 et 2.4). Les deux premiers fichiers, matrix1.c et matrix2.c, doivent fournir deux implémentations différentes d'une même micro-bibliothèque. Leurs fonctions sont courtes (parfois une seule instruction). Vous pouvez écrire les fonctions de basic.c ou determinant.c dès qu'un des fichiers matrix1.c ou matrix2.c est complet.

ℹ Les descriptions précises des fonctions à écrire sont détaillées dans les fichiers .h.

⚠ **Important** : Ne perdez **pas** de temps à tester si les arguments passés à votre programme sont corrects. (cette tâche est laissée à la charge de l'utilisateur des fonctions).

2.1 Première implémentation de la micro-bibliothèque : `matrix1.c` (3 points)

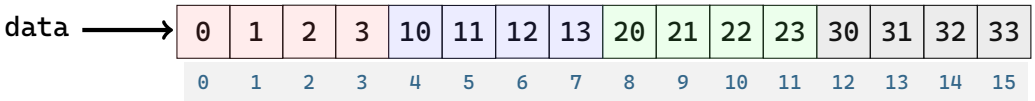
Cette implémentation est à réaliser dans le fichier matrix1.c. Une matrice est codée par le type suivant :

```
typedef struct matrix {
int dimension;
int *data;
} matrix;
```

Ici, `dimension` est la dimension de la matrice à représenter (c'est son nombre de lignes, ou de colonnes car ces nombres sont égaux). Le principe est que `data` permet de stocker les éléments de la matrice, les uns à la suite des autres en mémoire. Une fois alloué, `data` devra donc pointer sur la première case d'une zone de `(dimension * dimension)` entiers, contenant les lignes de la matrice, lues de haut en bas. Par exemple, pour la matrice suivante, `dimension` vaut 4.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{pmatrix}$$

Pour la représenter, il faudra allouer, en plus de la structure `matrix`, le tableau suivant pour le champ `data` :

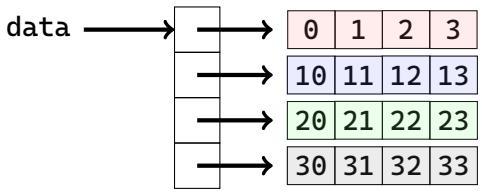


2.2 Seconde implémentation de la micro-bibliothèque : `matrix2.c` (3 points)

Cette implémentation est à réaliser dans le fichier `matrix2.c`. Une matrice est codée par le type suivant :

```
typedef struct matrix {
int dimension;
int **data;
} matrix;
```

À nouveau, `dimension` est la dimension de la matrice à représenter. Remarquez que le type de `data` est **différent** de celui de la première implémentation : il pointe sur la première case d'un tableau de `dimension` pointeurs sur des entiers. Le i-ème de ces pointeurs pointe sur la première case de la i-ème ligne de la matrice (en commençant à $i = 0$). Par exemple, pour allouer la mémoire représentant la même matrice que ci-dessus, il faudra allouer, en plus de la structure `matrix`, **plusieurs** tableaux. De même, la désallocation est un peu plus compliquée qu'avec l'implémentation précédente.



2.3 Fonctions de test de la bibliothèque : `basic.c` (14 points)

On demande d'écrire des fonctions de test des bibliothèques. Le travail demandé est décrit dans basic.h. Notez que les fonctions des deux bibliothèques ont les mêmes noms, et le fichier d'en-tête est le même, matrix.h. Cependant, l'une seulement des bibliothèques sera utilisée (laquelle dépend de la valeur de la macro `VERSION`). Les fonctions de basic.c doivent fonctionner **avec l'une comme l'autre des bibliothèques**. Autrement dit, vous ne devez **pas** utiliser directement la structure `matrix`, mais uniquement vos fonctions de bibliothèque. Par exemple, pour affecter 42 en ligne 2, colonne 5 d'une matrice `M`, on utilisera `set_matrix_value(M, 2, 5, 42);`.

2.4 Calcul de déterminant (hors TP noté de l'an dernier)

On veut calculer le déterminant d'une matrice carrée. Si n est la dimension d'une matrice M, les indices de ses lignes et colonnes varient entre 0 et $n-1$. On note $M_{i,j}$ la valeur de M en ligne i, colonne j. Le déterminant de M est alors :

$$\det(M) \stackrel{\text{def}}{=} \sum_{\sigma \in \mathfrak{S}_n} (-1)^{\text{nb\_inv}(\sigma)} M_{0,\sigma(0)} M_{1,\sigma(1)} \cdots M_{n-1,\sigma(n-1)}.$$

Cette formule utilise des notations standard mais mérite quelques explications :

- Ici, $\mathfrak{S}_n$ est l'ensemble des permutations de $\{0, 1, \ldots, n-1\}$, c'est-à-dire l'ensemble des bijections de $\{0, 1, \ldots, n-1\}$ dans lui-même. Dans determinant.c, une permutation est représentée par une structure contenant un tableau et sa taille. Ainsi, la permutation σ de $\mathfrak{S}_4$ définie par $\sigma(0) = 2, \sigma(1) = 0, \sigma(2) = 1$ et $\sigma(3) = 3$ est représentée par une structure ayant un champ `n` valant `4` et un tableau `T` de 4 entiers :

0	1	2	3
2	0	1	3

- Une inversion d'une permutation $\sigma \in \mathfrak{S}_n$ est un couple (i, j) avec $i < j$ et $\sigma(i) > \sigma(j)$. On définit alors $\text{nb\_inv}(\sigma)$ comme le nombre d'inversions de σ.

Le calcul de $\det(M)$ présente plusieurs étapes. Il faut générer une à une toutes les permutations de $\mathfrak{S}_n$, par la fonction `next_permutation()` : elle prend en entrée une permutation, et calcule la suivante dans l'ordre lexicographique. Cette fonction utilise elle-même trois fonctions auxiliaires appelées `pivot()`, `min_index()` et `reverse()` qui sont expliquées dans determinant.h. Pour écrire la fonction `determinant()` qui calcule le déterminant d'une matrice, on utilise aissi la fonction `product()`, qui calcule un seul terme $M_{0,\sigma(0)} M_{1,\sigma(1)} \cdots M_{n-1,\sigma(n-1)}$ de la somme ci-dessus.

Remarque. Le calcul propose de sommer $n!$ termes pour une matrice de dimension n, ce qui est inefficace. Il existe un algorithme en $O(n^3)$ pour calculer le déterminant d'une matrice de dimension n, mais il demande plus d'algèbre.