On travaillera depuis la machine virtuelle ClefAgreg2019. Je vous fournis un squelette de fichier, à télécharger depuis `https://cahier-de-prepa.fr/mp2i-kleber/docs?rep=4`. Le fichier s'appelle `TP22_squelette.c` et il faut le compléter.

Vous pouvez glisser-déposer le fichier depuis Windows vers la machine virtuelle, ou l'ouvrir dans le "Bloc note" Windows et copier-coller son contenu vers un fichier `TP22.c` ouvert dans Emacs dans la machine virtuelle. *Si vous n'arrivez pas, vous pouvez travailler en ligne, avec* `https://www.onlinegdb.com/`.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal).

```
$ COMPILATEUR -O3 -Wall -Werror -Wextra -Wvla -fsanitize=address \
  -fsanitize=undefined -pedantic -std=c11 -o TP22.exe TP22.c
$ ./TP22.exe
```

Multiplications de matrices carrées par la méthode de Strassen

Comme en cours mardi 03/05/2022, on s'intéresse au problème de multiplications de matrices $n \times n$, pour $n \in \mathbb{N}$.

On va implémenter la méthode naïve ("i k j" en temps $\Theta(n^3)$), la méthode diviser pour (pas mieux) régner qui est encore cubique (en $\Theta(n^{\log_2(8)}) = \Theta(n^3)$) et la méthode diviser pour régner de Strassen, qui est asymptotiquement en $\Theta(n^{\log_2(7)})$.

Création et libération de matrices rectangles

Nous manipulerons des matrices, carrées ou rectangles, à valeurs entières (pour éviter les potentielles erreurs d'arrondis), soit des `A` et `B` de type `int**`.

Pourquoi pas un VLA ? Comme ils sont hors programme, nous éviterons d'utiliser des VLA (*Variable Length Arrays*) et des signatures écrites comme `int A[n][m]`, même si elles permettraient de mieux comprendre les signatures des fonctions.

On rappelle que pour créer une matrice rectangle `A` de taille $n \times m$, il faut allouer d'abord un tableau `int** mat = (int**) malloc(n * sizeof(int*))` de `n` pointeurs vers des `int*` (les lignes de la matrice), puis avec une boucle, allouer `n` fois des tableaux de taille `m` : `mat[i] = (int*) malloc(m * sizeof(int))`.

1. Écrire une fonction `int** MatConstante(int n, int m, int cste)` qui alloue une telle matrice, et la remplie entièrement de la valeur entière constante `cste`.
   Quelle est sa complexité temporelle ?

Pour libérer une telle matrice `mat`, il faut bien sûr libérer le pointeur `mat`, mais avant cela il faut libérer chaque ligne `mat[i]` (comme elle a été obtenue par un `malloc`, en question précédente).

2. Écrire une fonction (non récursive) `void liberer_matrice(int n, int** mat)` qui libère une matrice `mat` de taille $n \times m$.
   Pourquoi n'a-t-on pas besoin de fournir la deuxième dimension `m` ?

On pensera bien évidemment à appeler `liberer_matrice(n, mat)` partout où nécessaire dans la suite, pour éviter les fuites de mémoire. Ces fuites de mémoire sont normalement détectées lors de l'exécution du binaire obtenu après compilation, grâce à l'option de compilation `-fsanitize=address`.

Deux matrices utiles : `zéros(n,m)` et `identite(n,m)`

On va maintenant écrire deux fonctions pour générer des matrices rectangles utiles dans nos exemples en fin de fichier (dans le `main`), pour commencer à manipuler un peu ces matrices.

3. Écrire une fonction `int** zeros(int n, int m)` qui utilise `MatConstante` pour créer une matrice de taille $n \times m$, remplie de la valeur `0`.
4. Faire de même pour `int** identite(int n, int m)` qui crée une matrice identité, nulle partout sauf pour `i=j` (c'est-à-dire qu'il faut mettre `mat[i][i] = 1` pour $0 \leqslant i < \min(n, m)$).

La fonction `zeros(n, m)` sera utilisée partout après, mais `Inm = identite(n, m)` ne sera utile que dans les tests, pour vérifier que `A x Inm = A` (ils sont fournis dans le squelette).

Sommes et soustractions de matrices rectangles

Avant d'écrire les deux méthodes de produit en diviser pour régner (DPR), il nous faut écrire la somme de deux matrices de même taille. La méthode DPR n'utilise que des sommes (4 exactement), mais la méthode de Strassen utilise aussi des soustractions.

5. Écrire une fonction `int** somme(int n, int m, int** A, int** B)` qui commence par allouer une matrice nulle `C` de dimension $n \times m$, puis qui remplie ses cases comme il faut pour calculer `C = A + B` (somme matricielle).
   Quelle est sa complexité temporelle en fonction de `n` et `m` ?
6. Faire de même pour `int** soustraction(int n, int m, int** A, int** B)` qui calcule `C = A - B`.

Méthode naïve (cubique) pour le produit rectangle (méthode "i k j")

Pour vérifier que les deux méthodes DPR produisent des résultats corrects, mais aussi pour comparer leurs temps de calculs, et enfin pour les cas de bases des méthodes DPR (quand `n` est "trop petit"), nous avons besoin du produit naïf de deux matrices rectangles de dimensions compatibles.

On rappelle qu'un produit `C = A x B` est compatible si `A` est de taille $n \times m$ et `B` est de taille $m \times p$: si les deux matrices partagent respectivement leur seconde et première dimension.

7. Avec ces notations, quelles sont les dimensions de la matrice produit `C = A x B` ? Que vaut son coefficient `C[i][j]` en fonctions des coefficients de `A[i][k]` et `B[k][j]`.
8. En déduire une fonction `int** produit_naif(int n, int m, int** A, int p, int** B)` qui alloue `C` avec la fonction `zeros`, puis qui remplie ses cases avec la formule itérative de la question précédente.
   Quelle est sa complexité temporelle en fonction de `n`, `m` et `p` les trois dimensions ?

9. En déduire une fonction `int** produit_carre(int n, int** A, int** B)` qui calcule le produit de deux matrices carrées `A` et `B`, de dimensions $n \times n$.
   Quelle est sa complexité temporelle, en fonction de `n` uniquement ?
10. Pour tester nos autres fonctions de produit, on va avoir besoin d'une fonction qui permet de tester l'égalité de deux matrices `A` et `B` de mêmes dimensions. Écrire une fonction `bool sont_egaux(int n, int m, int** A, int** B)` qui vérifie l'égalité des deux matrices, et qui affiche un petit message d'erreur en cas d'erreur détectée, par exemple comme ça :

```
Oups : pour i = 0, j = 0, A[i][j] = 0 != B[i][j] = 1...
```

On est prêt à passer aux deux méthodes diviser pour régner !

Dans tout le reste du TP, on s'intéressera à des matrices carrées `A` et `B` de dimensions $n \times n$, en imposant que n soit une puissance exacte de 2 (c'est-à-dire qu'il existe un $i \in \mathbb{N}$ tel que $n = 2^i$). Cela permet de s'assurer que toutes les divisions par 2 sont des divisions exactes et pas des parties entières inférieure ou supérieure, et de simplifier considérablement le code.

Extraction de sous-matrices

Une composante commune aux deux méthodes DPR est leur besoin d'extraire des sous-matrices carrées de taille plus petite, en connaissant leurs indices `i1..i2` et `j1..j2` (avec la convention habituelle en C : les bornes à gauche sont incluses, celles à droite ne le sont pas).

11. Écrire une fonction `int** sous_matrice(int n, int m, int** A, int i1, int i2, int j1, int j2)` qui alloue une matrice nulle `ssA` de dimension `n12 = i2 - i1` et `m12 = j2 - j1`, et la remplit à partir des valeurs de `A`. Attention à ne pas faire d'erreurs de calcul sur les indices, entre ceux de `A` et ceux de `ssA`.

En pratique, on s'intéresse à découper une matrice carrée `A` en quatre sous-blocs, de dimensions $n/2 \times n/2$ (on rappelle que `n` est une puissance de 2, donc les divisions sont entières et chaque sous-blocs a les mêmes dimensions) :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ \\ B_{21} & B_{22} \end{pmatrix}$$

12. En déduire quatre fonctions `haut_gauche`, `haut_droite`, `bas_gauche` et `bas_droite` de signatures `int** f(int n, int** A)`, pour extraire les quatre sous-blocs carrés respectivement en haut à gauche (noté `A11`), en haut à droite (`A12`), en bas à gauche (`A21`) et en bas à droite (`A22`).
13. L'opération inverse est de reconstruire une matrice `C` à partir de ses quatre sous-blocs carrés `C11`, `C12`, `C21` et `C22`, de dimensions $n \times n$. Il faut allouer une matrice `C` nulle de dimensions $(2n) \times (2n)$, et la remplir correctement. Par exemple `C[i][j] = C11[i][j]` en haut à gauche, et `C[n+i][n+j] = C22[i][j]` en bas à droite (à faire dans une double boucle sur `i = 0..n` et `j = 0..n`).

Écrire une fonction `int** reconstruire(int n, int** C11, int** C12, int** C21, int** C22)` qui réalise cette opération.

14. Écrire une fonction récursive `bool est_puissance_de_deux(int n)` qui teste si `n` est bien une puissance de 2.
    - Relation de récurrence : il faut que `2 * (n/2) == n` et que `est_puissance_de_deux(n/2)` soit aussi vrai.
    - Cas de base : faux pour `n==0`, vrai pour `n==1`.
15. Avant les fonctions suivantes, choisir une valeur pour la constante entière `LEAF_SIZE`, par exemple `16`. Quand `n` sera plus petit ou égal à cette constante, les méthodes DPR de produit matriciel se contenteront d'appeler la méthode naïve "i k j". Cela permet d'accélérer grandement les calculs.

Méthode naïve diviser pour (pas mieux) régner pour le produit (cubique)

La méthode diviser pour régner naïve fonctionne comme suit :

- Si `n <= LEAF_SIZE`, appeler `produit_carre` naïf. Sinon :
- Découper `A` en quatre sous-blocs de dimensions $(n/2) \times (n/2)$ `A11`, `A12`, `A21` et `A22`. Idem pour `B`.
- Calculer les huit produits suivants : `p1 = A11 x B11`, `p2 = A12 x B21`, `p3 = A11 x B12`, `p4 = A12 x B22`, `p5 = A21 x B11`, `p6 = A22 x B21`, `p7 = A21 x B12`, `p8 = A22 x B22`, en utilisant des appels récursifs à la même fonction. Dès qu'ils sont calculés, il faut bien penser à libérer `A11, ..., A22` et `B11, ..., B22`.
- Calculer les quatre sous-blocs de `C = A x B` : `C11 = p1 + p2`, `C12 = p3 + p4`, `C21 = p5 + p6`, `C22 = p7 + p8` (les formules sont logiques et se retrouvent facilement). Il faut aussi penser à libérer `p1, ..., p8` ensuite.
- Reconstruire `C` à partir de `C11, ..., C22`, et les libérer ensuite.
- Renvoyer `C`.

16. Écrire une fonction récursive `int** produit_DPR(int n, int** A, int** B)`.
    Le squelette fournit une organisation à suivre, il faut remplir les trous.
17. Si on note $T(n)$ la complexité de cette fonction pour des matrices de dimensions $n \times n$, écrire une relation de récurrence de la forme $T(n) = aT(n/b) + O(n^k)$.

- Que valent les trois paramètres a, b et k ?
- Si on applique le "théorème maître", dans quel cas cas se trouve-t-on (i, ii ou iii) ? Et quelle forme obtient-on pour une majoration asymptotique de $T(n)$?
- Est-ce que cette méthode diviser pour régner naïve est asymptotiquement meilleure que la méthode naïve "i k j" ?

Méthode de Strassen : diviser pour (mieux) régner pour le produit (sous cubique)

La méthode de Strassen va être similaire, mais un peu plus compliquée. L'objectif est de réaliser seulement 7 multiplications de dimensions $(n/2) \times (n/2)$, au lieu de 8, quitte à réaliser plus de sommes de sous-blocs et aussi des soustractions. Ces sommes et soustractions sont toutes en temps $\Theta((n/2)^2) = \Theta(n^2)$, comme précédemment, donc on obtiendra une relation de récurrence de la forme $T(n) = 7T(n/2) + O(n^2)$ avec $a = 7$ au lieu de $a = 8$ dans la méthode DPR naïve.

La méthode diviser pour régner de Strassen fonctionne comme suit :

— Si `n <= LEAF_SIZE`, appeler `produit_carre` naïf. Sinon :

— Découper `A` en quatre sous-blocs de dimensions $(n/2) \times (n/2)$ `A11`, `A12`, `A21` et `A22`. Idem pour `B`.

— Calculer les $a = 7$ produits suivants : `p1 = (A11 + A22) x (B11 + B22)`, `p2 = (A21 + A22) x (B11)`, `p3 = (A11) x (B12 - B22)`, `p4 = (A22) x (B21 - B11)`, `p5 = (A11 + A12) x (B22)`, `p6 = (A21 - A11) x (B11 + B12)`, `p7 = (A12 - A22) x (B21 + B22)`, en utilisant des appels récursifs à la même fonction. Dès qu'ils sont calculés, il faut bien penser à libérer `A11, ..., A22` et `B11, ..., B22`.

— Attention, pour éviter les fuites de mémoire, il faut malheureusement nommer les résultats des appels intermédiaires à `somme` et `soustraction` (par exemple `p1x = A11 + A22` et `p1y = B11 + B22` avant de calculer `p1 = p1x x p1y`), pour les libérer dès qu'ils sont utilisés. Le squelette donne le code pour `p1`, `p2` et `p3`, il faut terminer le travail pour les autres.

— Calculer les quatre sous-blocs de `C = A x B`, avec des formules un peu plus compliquées que précédemment : `C11 = p1 + p4 - p5 + p7`, `C12 = p3 + p5`, `C21 = p2 + p4`, `C22 = p1 + p3 - p2 + p6`. Pour `C11` et `C22`, il faut de nouveau utiliser des variables temporaires (par exemple `C11x` et `C11y`) et les libérer après. Le squelette contient le code pour `C11`, il faut terminer pour les autres. Il faut aussi penser à libérer `p1, ..., p7` ensuite.

— Reconstruire `C` à partir de `C11, ..., C22`, et les libérer ensuite.

— Renvoyer `C`.

18. Écrire une fonction `int** produit_Strassen(int n, int** A, int** B)` qui implémente cette méthode. Le squelette donne une bonne partie du code, il faut le terminer.

Génération de matrices aléatoires

Afin de tester nos fonctions de produit, on va chercher à générer des matrices à valeurs entières, tirées uniformément dans un intervale, par exemple `[-100, 100]`.

Pour faire cela en C, on dispose d'une fonction `int rand()` qui renvoie un entier pseudo-aléatoire tiré dans un très grand intervale (la borne est `RAND_MAX = 2147483647`). Pour obtenir un entier dans un certain intervale `[lower, upper]` (bornes incluses), on fournit la fonction `int rand_range(int lower, int upper)`.

19. Regarder son code et comprendre la formule utilisée.

20. En déduire une fonction `int** rand_matrice(int n, int m, int lower, int upper)` qui alloue une matrice nulle de la bonne taille, et remplit chaque case d'un entier aléatoire donné par `rand_range(lower, upper)`.

Vérification des produits

Objectif : pas d'erreurs ni de fuites de mémoire ! Dans la fonction `main` fournie, vous pouvez décommenter les tests au fur et à mesure que vous avancez dans le TP. A la fin, il faut que tout soit décommenté, et que tous les tests passent sans erreurs lors de l'exécution du binaire (pour `n` assez grand, plus grand que `LEAF_SIZE` et une puissance de 2), et qu'il n'y ait aucune fuite de mémoire.

Arguments en ligne de commande (rappels) Dans la fonction `main` fournie, vous observerez que l'on utilise les arguments `argv[1]` et `argv[2]` fournis au binaire en ligne de commande (si présents), pour choisir `n` la taille des matrices (par défaut `n=256`) et `nbRepetitions` le nombre de répétitions de la boucle de tests.

Cela permet d'appeler le binaire comme suit :

```
$ ./TP22.exe 256 10
```

Exemple de sortie du binaire (à la fin) Appeler le binaire avec `n=512` et `nbRepetitions=1` affichera par exemple :

```
$ time ./TP22.exe 1024 1
Pour n = 1024, lower = -100, upper = 100, générations de matrices aléatoires :

Test 1 / 1...
Vérification que A x I = A (produit naif) pour I = Inn identité.
        Temps = 10 seconde(s)...
Vérification que A x I = A (produit DPR naif).
        Temps = 13 seconde(s)...
Vérification que A x I = A (produit Strassen).
        Temps = 9 seconde(s)...
Calcul du produit C = A x B naïf.
        Temps = 9 seconde(s)...
Vérification que A x B (produit naif) = A x B (produit DPR naif).
        Temps = 11 seconde(s)...
Vérification que A x B (produit naif) = A x B (produit Strassen).
        Temps = 9 seconde(s)...

real 61,049     user 60,696     sys 0,324       pcpu 99,95
```

Chronomètre de temps de calculs

Dans la fonction `main`, il y a quelques lignes de la forme suivante :

```
debut = time(NULL);
... calculs ...
printf("\tTemps = %li seconde(s)...\n", time(NULL) - debut);
```


Ils permettent de calculer (et d'afficher) le temps mis par les `... calculs ...`, en utilisant la fonction système (dans `stdlib.h`) `long int time(long int* timer)` (on ignore son argument, d'où l'utilisation de `NULL`). Cette fonction renvoie le temps machine en seconde, et donc utiliser `time(NULL) - debut` permet de faire un chronomètre.

Ouverture si vous voulez : généraliser à des tailles quelconques

Pour conclure ce TP, si vous avez le temps, vous pouvez essayer de généraliser les deux méthodes diviser pour régner à des matrices de dimensions quelconques, pas seulement des puissances exactes de 2. (cette partie n'est pas fournie dans le corrigé)

L'idée de base est la suivante : étant donné $n \in \mathbb{N}$, déterminer le plus petit $i \in \mathbb{N}$ tel que $n \leqslant 2^i$.

— Si $n = 2^i$, on sait déjà faire.

— Sinon :

  — On complète `A` et `B` de dimensions $n \times n$ en des matrices `A'` et `B'` de dimensions $2^i \times 2^i$, en les remplissant à droite et en bas par des valeurs nulles (`0`).

  — On calcule leur produit `C' = A' x B'`, qui sera lui aussi remplis de `0` dans les colonnes de droite et les lignes du bas.

  — Et enfin, pour obtenir `C` à partir de `C`, on a juste à regarder le sous-bloc en haut à droite, de dimensions $n \times n$, et le renvoyer.

21. Implémenter cette idée pour les deux méthodes DPR, dans deux fonctions :

— `int** produit_DPR_general(int n, int** A, int** B)`,

— et `int** produit_Strassen_general(int n, int** A, int** B)`.

22. Faire des tests pour des dimensions quelconques dans votre `main`.

On peut aussi généraliser à des produits de matrices non carrées, vous pouvez chercher de votre côté comment faire. (c'est la même idée !)

