TP1 : Multiplication d'entiers, programmation C

L'objectif de cette séance est de s'exercer aux compétences suivantes :

- lire un algorithme (itératif ou récursif) en pseudo-code ;
- exécuter à la main un algorithme (itératif ou récursif) sur un exemple jouet ;
- évaluer de manière intuitive l'efficacité d'un algorithme ;
- écrire des programmes simples en C.

1 Configuration de l'environnement de travail

Nous commencerons par préparer l'environnement de travail : création d'un dossier dédié au cours, mise en place d'un dépôt Git (facultatif) et vérification de l'installation et du bon fonctionnement du compilateur.

1.1 Dossier de travail et dépôt Git

Il est recommandé de sauvegarder votre travail dans un dépôt Git. Cela vous permettra de mieux maîtriser cette technologie (qui pourrait vous être utile dans votre vie professionnelle) et de conserver l'historique de votre travail.

Toutefois, si vous ne souhaitez pas utiliser Git, vous devez créer un dossier sur votre machine dans lequel vous placerez tous les fichiers du cours. Nous désignerons ce dossier par la suite comme votre **dossier de travail**. Pour chaque TP, des fichiers vous seront mis à disposition, que vous devrez extraire dans le dossier de travail, en créant un sous-dossier distinct pour chaque TP.

Un résumé des étapes à suivre est présenté ci-dessous. Consultez le guide Git sur Blackboard pour plus d'informations.

Commencez par vérifier que Git est installé et que l'accès à GitLab ESIEE est correctement configuré, en suivant les instructions du guide. Créez ensuite un dépôt sur GitLab ESIEE.

Vous pouvez maintenant créer le dossier de travail sur votre machine. Ouvrez un terminal dans le répertoire où vous souhaitez placer votre dépôt (par exemple, dans votre dossier Documents), puis exécutez la commande suivante, en remplaçant `<username>` par votre nom d'utilisateur (sans les caractères < et >).

```
git clone git@git.esiee.fr:<username>/e3-algo-2026.git
```

Un dossier nommé `e3-algo-2026` devrait alors apparaître dans le répertoire courant. Vous pouvez le renommer si vous le souhaitez. Ce dossier nouvellement créé sera votre **dossier de travail**.

Pour enregistrer votre travail dans Git **à la fin de la séance**, vous devez ouvrir un terminal dans votre dossier de travail et exécuter les commandes `add`, `commit` et `push` :

```
git add .
git commit -m "Un message explicatif"
git push
```

N'oubliez pas le point après `add` (séparé par un espace) et de remplacer le message entre guillemets par une description de votre travail (par exemple, `"Exercices TP1"` pour ce TP).

1.2 Compilateur

Dans ce cours, nous programmerons en langage C sous un système Linux. Si vous utilisez Windows ou macOS, vous devez d'abord vous assurer que tous les outils nécessaires sont disponibles sur votre machine.

Plusieurs possibilités s'offrent à vous :

1. **Sous Windows :** installer WSL2 (comme dans le cours Programmation Java) ;
2. **Sous macOS :** macOS dispose déjà d'un compilateur accessible avec la commande `gcc`, qui est légèrement différent. Vous *pouvez* choisir d'installer le compilateur utilisé sous Linux, appelé GNU gcc (comme dans le cours Programmation Java) ;
3. **Sous Windows, macOS ou Linux :** utiliser un conteneur Docker (comme dans le cours Programmation Python).

Si vous utilisez WSL2, Docker ou Linux, assurez-vous d'installer le paquet `build-essential`. Ouvrez un terminal et exécutez les commandes suivantes :

```
sudo apt update
sudo apt install build-essential
```

1.3 Exécution du programme de test

Une fois l'environnement configuré, créez un sous-dossier `TP1` dans votre dossier de travail, puis extrayez le contenu du fichier `algo-tp01-code.zip` dans ce nouveau dossier. Ouvrez ensuite un terminal dans le dossier `TP1`.

Compilez le programme de test à l'aide de la commande suivante (ou d'une autre commande de votre choix) :

```
gcc -Wall test.c -o test
```

Enfin, exécutez le programme avec la commande suivante :

```
./test
```

Un message devrait vous indiquer que le test a réussi.

2 Multiplication d'entiers

L'objectif de cette partie du TP est de comparer différents algorithmes de multiplication d'entiers afin d'évaluer leur efficacité. Pour étudier cette tâche d'un point de vue algorithmique, nous devons restreindre les opérations que nous pouvons utiliser.

Problème : multiplier deux entiers donnés en n'utilisant que les opérations suivantes :

1. la multiplication de deux chiffres (tables de multiplication) ;
2. l'addition '+' ;
3. la multiplication par 10 (et ses puissances 100, 1000, ...) ;
4. la division entière par 10 : quotient '//' et reste '%'.

Exercice 1 : Algorithme Naïf. Considérez l'algorithme suivant :

Algorithme 1 MultiplicationNaïve

Données: deux entiers positifs `a` et `b`
Résultat: le produit de `a` par `b`

```
1: acc ← 0
2: for i de 0 à b − 1 do
3:     acc ← acc + a
4: end for
5: return acc
```

1. Exécutez à la main MULTIPLICATIONNAIVE pour les valeurs 32 et 13 et présentez le résultat de cette exécution de la manière la plus claire possible.
   Pour cela, vous pouvez vous aider du tableau suivant (à remplir) pour représenter l'état des variables au cours des itérations de l'algorithme.

i :		0	1	2	3	4	5	6	7	8	9	10	11	12
acc :														

2. Est-il possible (raisonnable) d'exécuter à la main MULTIPLICATIONNAIVE pour des grandes valeurs de a et b, par exemple $a = 2342$ et $b = 3224$?
   Combien de fois l'opération d'addition située à l'intérieur de la boucle serait-elle répétée ?

Exercice 2 : Comme à l'école primaire. Par la suite, si a est un entier, on désigne par a_i le $(i+1)$-ème chiffre de a en partant de la droite de sorte que si a s'écrit avec n chiffres on a :

$$a = \sum_{i=0}^{n-1} a_i 10^i$$

Par exemple si $a = 2342$, on a $a_3 = 2$, $a_2 = 3$, $a_1 = 4$, $a_0 = 2$. Plus graphiquement, on peut dessiner le tableau suivant pour représenter les $\{a_i\}$:

i	3	2	1	0
a_i	2	3	4	2

Considérez l'algorithme suivant :

Algorithme 2 MultiplicationEcolePrimaire

Données: deux entiers positifs `a` et `b`, comportant (au plus) n chiffres
Résultat: le produit de `a` par `b`

1: `p` $\leftarrow 0$
2: **for** i de 0 à $n-1$ **do**
3: `acc` $\leftarrow 0$, `r` $\leftarrow 0$
4: **for** j de 0 à $n-1$ **do**
5: `x` $\leftarrow$ `b`$_i \cdot$ `a`$_j +$ `r` // *multiplication de 2 chiffres + retenue précédente*
6: `r` $\leftarrow$ `x`$//10$ // *mémoriser la retenue pour l'itération suivante*
7: `acc` $\leftarrow$ `acc` $+ (x\%10) \cdot 10^j$
8: **end for**
9: `acc` $\leftarrow$ `acc` $+$ `r` $\cdot 10^n$
10: `p` $\leftarrow$ `p` $+$ `acc` $\cdot 10^i$
11: **end for**
12: **return** `p`

1. Exécutez à la main MULTIPLICATIONECOLEPRIMAIRE pour les valeurs de 2342 et 3224 et présentez le résultat de cette exécution de la manière la plus claire possible.
   Pour cela, vous pouvez vous aider du tableau suivant pour représenter l'état des variables au cours des itérations de l'algorithme.

i	3	2	1	0
a_i	2	3	4	2

i	3	2	1	0
b_i				

	init					reset					
i		0	0	0	0		1	1	1	1	
j		0	1	2	3		0	1	2	3	
x											
`r`	0										
`acc`	0										
`p`	0					9368					

	reset					reset					fin
i		2	2	2	2		3	3	3	3	
j		0	1	2	3		0	1	2	3	
x											
`r`											
`acc`											
`p`	56208					524608					7550608

2. Combien de fois l'opération de multiplication élémentaire de 2 chiffres située à l'intérieur de la double boucle est-elle répétée (lorsque $a = 2342$ et $b = 3224$) ? Comparez avec votre réponse à l'Exercice 2.
   Est-il possible (raisonnable) d'exécuter à la main MULTIPLICATIONECOLEPRIMAIRE pour des grandes valeurs de a et b, par exemple avec des nombres comportant $n = 16$ chiffres ?

3 Programmation en C

Exercice 3 : Affichage d'un tableau (*). Compilez le fichier `affichage.c` et exécutez le programme, puis observez le résultat. Examinez ensuite le code et assurez-vous d'en comprendre le contenu, notamment :

1. Que fait la fonction `affiche_tab` ?
2. Comment la fonction atteint-elle son objectif ? Quels sont les rôles de la boucle `for` et de la condition `if` ?
3. Comment modifier la fonction pour n'afficher que 2 chiffres après la virgule ? Et si vous souhaitez afficher chaque nombre sur une ligne distincte ?
4. La structure du fichier vous semble-t-elle cohérente ? Pourquoi la fonction `affiche_tab` est-elle placée avant la fonction `main` ?
5. Essayez d'exécuter le même programme avec des tableaux plus longs et plus courts. Quel est le résultat ? Quelles modifications ont été nécessaires ?

Exercice 4 : Recherche d'éléments sur critère (*). Écrivez une fonction qui retourne l'indice du premier élément strictement négatif parmi les n premiers éléments d'un tableau de réels (-1 si aucun élément n'est négatif).

Prototype: `int indice_premier_negatif (double t[], int n);`

Testez votre fonction avec au moins trois tableaux de tailles différentes.

Exercice 5 : Maximum d'un tableau (*). Écrivez une fonction qui retourne la valeur du plus grand des n premiers éléments d'un tableau de réels.

Prototype: `double valeur_plus_grand (double t[], int n);`

Testez votre fonction avec au moins trois tableaux de tailles différentes.

Exercice 6 : Un tableau est-il trié ? (*). Écrivez la fonction qui retourne 1 si les n premiers éléments du tableau sont en ordre croissant (au sens large) et 0 sinon.

Prototype: `int est_trie (double t[], int n);`

Exercice 7 : Trouver la moyenne. Complétez le programme du fichier `moyenne.c` afin qu'il demande à l'utilisateur de saisir une liste d'entiers, puis affiche la moyenne des nombres saisis.

La première ligne de l'entrée doit indiquer la longueur n du tableau, et les n lignes suivantes contiennent un élément du tableau chacune. Le programme doit afficher la moyenne des n valeurs saisis avec une précision de deux chiffres après la virgule.

Par exemple, pour l'entrée suivante :

```
4
1
3
3
5
```

le programme doit afficher `3.00`, la moyenne du tableau de longueur 4 et valeurs [1,3,5,5].

Testez votre programme avec au moins trois tableaux de tailles différentes.

4 Multiplication avec l'algorithme de Karatsuba (facultatif)

Exercice 8 : Vers l'algorithme de Karatsuba. Il est possible d'obtenir des algorithmes encore plus efficaces pour multiplier deux nombres. Ces algorithmes sont ceux qui sont implémentés dans nos ordinateurs.

Nous allons voir l'algorithme de Karatsuba[1] découvert en 1960. Il repose sur deux observations que nous exploiterons successivement pour améliorer l'efficacité de la multiplication.

Observation 1 Si a et b sont deux entiers comportant n chiffres, chacun peut être décomposé en deux parties de taille à peu près égale :

1. la partie droite comporte les $n//2$ chiffres situés le plus à droite du nombre ;
2. la partie gauche comporte les chiffres suivants.

La partie droite de a est désignée par a_d et sa partie gauche par a_g. La partie droite de b est désignée par b_d et sa partie gauche par b_g. Par exemple pour $a = 2342$ et $b = 3224$, nous avons :

$$a_g = 23, \quad a_d = 42, \quad b_g = 32, \quad b_d = 24.$$

On peut donc écrire :

$$a \cdot b = (a_g \cdot 10^{n//2} + a_d) \cdot (b_g \cdot 10^{n//2} + b_d)$$

En développant la partie droite de l'équation, on obtient :

$$a \cdot b = a_g \cdot b_g \cdot 10^{2 \cdot n//2} + a_g \cdot b_d \cdot 10^{n//2} + a_d \cdot b_g \cdot 10^{n//2} + a_d \cdot b_d$$

On observe dans la partie droite de l'équation que chacun des 4 termes est formé par :

- la multiplication de deux facteurs de taille inférieur à $n//2 + 1$ (par définition des décompositions de a et b) ;
- dont le résultat est multiplié par une puissance de 10 (et la multiplication par des puissance de 10 est facile car il s'agit d'un simple décalage de chiffres).

En factorisant, on peut simplifier l'équation précédente pour économiser une multiplication par une puissance de 10 :

$$a \cdot b = a_g \cdot b_g \cdot 10^{2 \cdot n//2} + (a_g \cdot b_d + a_d \cdot b_g) \cdot 10^{n//2} + a_d \cdot b_d$$

D'après cette équation on peut donc formuler l'algorithme récursif suivant pour effectuer le produit de a par b.

1. voir https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba

Algorithme 3 Mult1

Données: deux entiers positifs `a` et `b`, comportant (au plus) `n` chiffres
Résultat: retourne le produit de `a` par `b`

1: **if** n=1 **then**
2: **return** `a` · `b` // *Multiplication de deux chiffres*
3: **end if**
4: $\texttt{a}_d \leftarrow$ le nombre composé des n//2 chiffres les plus à droite dans `a`
5: $\texttt{a}_g \leftarrow$ le nombre composé des chiffres suivants de `a`
6: $\texttt{b}_d \leftarrow$ le nombre composé des n//2 chiffres les plus à droite dans `b`
7: $\texttt{b}_g \leftarrow$ le nombre composé des chiffres suivants de `b`
8: $p_1 \leftarrow \text{MULT1}(a_g, b_g, n//2 + n\%2)$
9: $p_2 \leftarrow \text{MULT1}(a_g, b_d, n//2 + n\%2)$
10: $p_3 \leftarrow \text{MULT1}(a_d, b_g, n//2 + n\%2)$
11: $p_4 \leftarrow \text{MULT1}(a_d, b_d, n//2)$
12: **return** $p_1 \cdot 10^{2 \cdot n//2} + (p_2 + p_3) \cdot 10^{n//2} + p_4$

1. Exécutez à la main MULT1 pour les valeurs de $a = 2342$ et $b = 3224$ et présentez le résultat de cette exécution de la manière la plus claire possible.
2. Combien de fois l'opération de multiplication élémentaire de 2 chiffres située à l'intérieur de la conditionnelle Si est-elle répétée (lorsque $a = 2342$ et $b = 3224$) ? Comparez avec votre réponse à l'Exercice 2.

Exercice 9 : Algorithme de Karatsuba. La version efficace de l'algorithme de Karatsuba repose sur une seconde observation fondamentale.

Observation 2 Si a_d, a_g, b_g, b_d sont quatre nombres entiers quelconques, l'égalité suivante est vraie :

$$a_g \cdot b_g + a_d \cdot b_d - (a_g - a_d) \cdot (b_g - b_d) = a_g \cdot b_d + a_d \cdot b_g$$

D'après cette égalité, nous pouvons réécrire l'équation surlignée dans la partie précédente :

$$a \cdot b = a_g \cdot b_g \cdot 10^{2 \cdot n//2} + \big(a_g \cdot b_g + a_d \cdot b_d - (a_g - a_d) \cdot (b_g - b_d)\big) \cdot 10^{n//2} + a_d \cdot b_d$$

Cette nouvelle équation ne fait intervenir que 3 multiplications différentes au lieu de 4 précédemment. Cette différence peut sembler négligeable mais c'est bien elle qui conduit à une amélioration très significative de l'efficacité de la méthode.

Algorithme 4 Karatsuba

Données: deux entiers positifs `a` et `b`, comportant (au plus) `n` chiffres
Résultat: retourne le produit de `a` par `b`

if `a` et `b` sont négatifs **then**
    return $\text{KARATSUBA}(|a|, |b|, n)$
else if un des deux nombres est négatif **then**
    return $-\text{KARATSUBA}(|a|, |b|, n)$
end if
if n=1 **then**
    return `a` $\cdot$ `b`
end if
$\texttt{a}_d \leftarrow$ le nombre composé des n//2 chiffres les plus à droite dans `a`
$\texttt{a}_g \leftarrow$ le nombre composé des chiffres suivants de `a`
$\texttt{b}_d \leftarrow$ le nombre composé des n//2 chiffres les plus à droite dans `b`
$\texttt{b}_g \leftarrow$ le nombre composé des chiffres suivants de `b`
$p_1 \leftarrow \text{KARATSUBA}(a_g, b_g, n//2 + n\%2)$
$p_2 \leftarrow \text{KARATSUBA}(a_g - a_d, b_g - b_d, n//2 + n\%2)$
$p_3 \leftarrow \text{KARATSUBA}(a_d, b_d, n//2)$
return $p_1 \cdot 10^{2 \cdot n//2} + (p_1 + p_3 - p_2) \cdot 10^{n//2} + p_3$

1. Exécutez à la main KARATSUBA pour les valeurs de $a = 2342$ et $b = 3224$ et présentez le résultat de cette exécution de la manière la plus claire possible.
2. Combien de fois l'opération de multiplication élémentaire de 2 chiffres située à l'intérieur de la conditionnelle est-elle répétée (lorsque $a = 2342$ et $b = 3224$) ? Comparez avec votre réponse votre réponse à la Question 3.2.
3. Est-il possible (raisonnable) d'exécuter à la main KARATSUBA pour des grandes valeurs de a et b, par exemple avec des nombres comportant $n = 16$ chiffres ? Combien de multiplications élémentaires seraient nécessaires dans ce cas ?
4. Pour aller encore un peu plus loin, essayez d'évaluer le nombre d'opérations nécessaires si a et b comportent chacun 2^k chiffres.
5. (Bonus) Implémentez un programme (en C) pour multiplier deux nombres en utilisant l'algorithme de Karatsuba. Cette fonction doit prendre en entrée deux entiers donnés sous forme de chaînes de caractères, et retourner une chaîne représentant leur multiplication.