Cette *vingt-sixième* colle vous fera travailler sur l'algorithme « Diviser pour Régner » de Gauss-Karatsuba pour la multiplication rapide de grands entiers, comme vu en cours vendredi 06/05 dernier, en C.

Exercice 0 : rappel de l'idée de algorithme

1. Rappeler au brouillon l'idée de l'algorithme de Gauss-Karatsuba, sans nécessairement redonner les formules exactes.
   - Spécifier le(s) cas de base,
   - Expliquer comment faire la séparation d'une instance E en un nombre fixé $a \geqslant 1$ d'instances $E_1, \dots, E_a$ de tailles plus petites, et spécifier a ainsi que le facteur $b \geqslant 1$ de division de la taille $|E_i| \leqslant \lceil \frac{|E|}{b} \rceil$,
   - Expliquer comment faire la fusion des résultats intermédiaires.

Exercice 1 : considérations théoriques sur l'algorithme

Illustration et exécution sur un exemple

Sur l'exemple de nombres : `x = 63` et `y = 57`, on trouve `x * y = 3591`.

- Avec la méthode diviser-pour-régner naïve, on trouve ce résultat en calculant `x = xg * 10 + xd` : `xg = 6`, `xd = 3`, et `y = yg * 10 + yd` : `yg = 5` et `yd = 7`, et donc `x * y = (xg * yg) * 100 + (xg * yd + xd * yg) * 10 + (xd * yd) = 6*5*100 + (6*7 + 3*5)*10 + 3*7 = 3000 + 570 + 21 = 3591`. On a fait 4 produits de chiffres, et deux "décalages vers la gauche", de respectivement deux chiffes (`*100`) et un chiffre (`*10`), et deux sommes.
- Avec la méthode diviser-pour-régner de Gauss-Karatsuba, ou trouve aussi ce même résultat, cette fois en calculant `x * y = u * 100 + (w - u - v) * 10 + v`, où `u` et `v` sont calculés comme précédemment : `w = (xg * yg) = 30` et `v = (xd * yd) = 21`, mais où `w = (xg + xd) * (yg + yd) = (6+3) * (5+7) = 9*12 = 108` et donc `(w - u - v) = 108 - 30 - 21 = 57`. On a fait seulement trois produits, encore deux décalages vers la gauche, et quatre sommes et deux soustractions.

Note : Le produit calculé pour le terme "du milieu" `w` peut être avec des valeurs plus grandes que des chiffres (ici `xg+xd = 6+3 = 9` reste un chiffre, mais `yg+yd = 5+7 = 12` dépasse), mais les chiffres des dizaines sont au plus 1 donc on a quand même simplifié le problème.

2. Sur l'exemple de nombres : `x = 79` et `y = 31`, faites de même : calculer le produit de la manière de votre choix, puis avec la méthode diviser-pour-régner naïve, et enfin la méthode de Gauss-Karatsuba.

Complexité temporelle

3. Si on note $T(n)$ la complexité temporelle de l'algorithme de Gauss-Karatsuba appelé sur deux grands entiers de même taille n, étant une puissance de 2, donner une relation de récurrence satisfaite par $T(n)$, sous la forme d'une inégalité de récurrence reliant $T(n)$ à

des valeurs de T plus petites et un terme en $\mathcal{O}(n^k)$, pour une certaine constante $k \in \mathbb{N}$ à spécifier.

4. Si on applique le « théorème maître » sur cette relation de récurrence, ou si on la résout à la main (avec la méthode de la somme télescopique vue vendredi), quel résultat obtient-on pour une domination asymptotique de $T(n)$ en fonction de n ? On ne demande *pas* de justification.
5. Est-ce que la méthode diviser-pour-régner naïve est plus efficace (asymptotiquement) que la méthode naïve de produit de deux grands entiers ?
6. Est-ce que la méthode diviser-pour-régner de Gauss-Karatsuba est plus efficace (asymptotiquement) que les deux méthodes précédentes ?

Implémentation en C

On s'intéresse désormais à implémenter en C cet algorithme de Gauss-Karatsuba, sur des grands entiers écrits en base $b = 10$. Je vous fournis un squelette de fichier, `Colles_26_squelette.c`, disponible sur `https://cahier-de-prepa.fr/mp2i-kleber/docs?rep=11`. Téléchargez le, et ouvrez le avec `https://www.onlinegdb.com/`.

On compilera avec les options de compilation ("Extra compiler flags") habituelles. Ou bien depuis la machine virtuelle ou votre ordinateur, et la ligne de commande suivante :

```
$ gcc -O3 -Wall -Wextra -Werror -Wvla -fsanitize=undefined -fsanitize=address
    -pedantic -std=c11 -o Colle_26.exe Colle_26.c && ./Colle_26.exe
```

Vous devrez remplir trois petites fonctions, et lire des morceaux du code fourni (qui est long, mais sans être hors de votre portée).

Structure pour représenter ces grands entiers

On propose la structure suivante :

```
struct grandentier {
    int longueur;
    int* chiffres;
};
typedef struct grandentier grandentier;
```

7. Que représente le champ `longueur` ? Et le champ `chiffres` ?
   Quelle relation y a-t-il entre eux ?
8. Remplir la fonction `libere_grandentier` dont le squelette fournit la signature.
9. Remplir la fonction `longueur`.
10. Remplir la fonction `chiffre`, pour qu'elle renvoie `x.chiffres[i]` si `i` est un indice valide pour ce tableau, ou la valeur `0` sinon.
11. **Lire** les codes des fonctions `normalise`, `Ajoute` (et `Soustrait`), `MultBase` et `MultChiffre` et pour **une** de ces fonctions, expliquer pourquoi elle s'exécute en temps linéaire en $\max(n_x, n_y)$ si elle a deux arguments x et y (des `grandentier`), ou en n_x si elle n'a qu'un argument x.
12. **Lire** le code de la méthode de multiplication naïve `MultNaive` et justifier pourquoi elle s'exécute en temps $O(n_x * n_y)$.
13. **Lire** le code de la méthode de multiplication diviser-pour-régner naïve `MultDPR`, et compter le nombre d'appels récursifs qui sont effectués dans le cas général.
    - En déduire une inégalité de récurrence satisfaite par $T'(n)$ la complexité temporelle de cette fonction, sur des entrées de tailles $n = 2^k$.
    - Si on la résout, par calcul direct ou par l'application du "théorème maître", quelle forme obtient-on pour $T'(n)$?
14. **Lire** le code de la méthode de multiplication diviser-pour-régner de Gauss-Karatsuba `MultKaratsuba`, et compter le nombre d'appels récursifs qui sont effectués dans le cas général.

Tests

Dans la fonction `main` sont fournis des tests, qui devraient marcher sans problème dès que vous aurez remplis les quelques fonctions de bases, demandées en questions 9, 10 et 11.

En exécutant le programme, il génère des "grands entiers" `x` et `y` de taille `n`, en base `b=10` pour un affichage agréable, dont les chiffres sont tirés aléatoirement entre 0 et 9. Le programme calcule ensuite différentes valeurs, par exemple `x+0`, `x+y`, `x*y` avec les trois différentes méthodes, et vérifient les résultats. Les temps de calculs des étapes importantes (les trois différentes méthodes de multiplication) sont affichés, en milli-secondes.

Voici un exemple de sortie :

```
Pour reproduire les résultats aléatoires suivants, utilisez seed = 1651958756847
Début de 1 tests :
Pour des grands entiers en décimal, avec n = 8 chiffres en base b = 10.

Test 1 / 1...
  x = 88971141
  y = 08947999
  zero_n = 00000000
    - x + zero_n = x : okay
    - zero_n + x = x : okay
  x_plus_y = 97919140
  y_plus_x = 97919140
    - x + y = y + x : okay
    - (x + y) - y = x : okay
    - (x + y) - x = y : okay
  x_fois_y = 796113680696859
  y_fois_x = 796113680696859
    - x * y = y * x : okay
        Temps = 0 milli-seconde(s)...
  x_fois_y_DPR = 796113680696859
  y_fois_x_DPR = 796113680696859
    - x * y = y * x : okay
        Temps = 0 milli-seconde(s)...
  x_fois_y_Karatsuba = 796113680696859
  y_fois_x_Karatsuba = 796113680696859
    - x * y = y * x : okay
        Temps = 0 milli-seconde(s)...
```

15. Expérimenter un peu avec le paramètre `n` (qui peut ne pas être une puissance exacte de 2), et trouver une taille assez grande, à partir de laquelle la méthode de Gauss-Karatsuba apparaît significativement plus rapide que les deux autres.

    Indice : pas besoin de dépasser `n=8192`, et on peut laisser `nbRepetitions = 1`.

16. Pouvez-vous vérifier que les deux méthodes les plus lentes semblent bien être en temps quadratique en `n` ? Si on multiplie `n` par 2, par combien est multiplié le temps de calcul des deux méthodes les plus lentes ?

