PROG
TP#5

2021-10-20

Instructions de rendu

Ce TP fait l'objet d'un rendu *en binôme*. Vous devez envoyer votre travail sous la forme d'une archive à :
pierre.karpman@univ-grenoble-alpes.fr (pour le groupe 1) ;
hippolyte.signargout@univ-grenoble-alpes.fr (pour le groupe 2) ;
au plus tard le 2021-10-29 à 18 :00. Celle-ci doit contenir :

- Votre programme répondant aux questions ci-dessous. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
- Un Makefile et les instructions permettant de l'utiliser.
- Un court compte-rendu en format PDF qui détaille vos réponses ***ainsi que vos choix de conception***.

*Vous ferez particulièrement attention aux allocation et désallocations mémoire dans l'ensemble de vos fonctions. Réfléchissez à vos besoins, et déterminez dans quelles fonctions l'allocation est la plus judicieuse ; il peut aussi être intéressant d'*aliaser *un certain nombre de vos arguments. Réfélchissez également aux fonctions auxiliaires (par exemple d'addition, de soustraction...) dont vous avez besoin et à leur possibles spécificités (par exemple déterminer quand le résultat doit être retourné dans une nouvelle variable ou quand il peut être calculé en place). Pensez à utiliser* `valgrind` *et* `assert()` *pour vérifier l'absence de fuites mémoires et de bugs.*

Multiplications de polynômes avec l'algorithme de Karatsuba

Le but de cet exercice est d'implémenter l'algorithme de Karatsuba pour multiplier deux polynômes de $\mathbb{Z}/2^{32}\mathbb{Z}[X]$ ***de même degré***, dont les coefficients sont stockés sur 32 bits. On rappelle que cet algorithme utilise la décomposition récursive suivante : soit P_0, P_1, Q_0, Q_1 quatre polynômes de degré $n-1$, on souhaite calculer le produit $R := (P_1X^n + P_0) \times (Q_1X^n + Q_0)$; on pose :

- $A := P_0 \times Q_0$;
- $B := (P_0 + P_1) \times (Q_0 + Q_1)$;
- $C := P_1 \times Q_1$;

ce qui permet de calculer $R = CX^{2n} + (B - (A + C))X^n + A$

Q.1 :

1. Définissez et implémentez une structure `struct poly_u` permettant de représenter un polynôme via ses coefficients stockés sur des entiers non signés de 32 bits et son degré.
2. Écrivez des fonctions d'allocation et de désallocation de ces structures, de prototypes `struct poly_u *alloc_poly_u(??? deg)` et `void free_poly_u(struct poly_u *p)` respectivement.
3. Écrivez une fonction d'affichage d'un polynôme.

Q.2 :

1. Écrivez une fonction `mulpu` qui calcule naïvement le produit de deux polynômes, et testez-la sur de petits exemples.
2. Testez les performances et évaluez le comportement asymptotique de `mulpu` sur des polynômes de degrés variables bien choisis. Testez également l'impact du compilateur et de ses optimisations.

Q.3 :

1. Écrivez une fonction `mulpuk1` qui calcule le produit de deux polynômes de même degré en appliquant *une fois* la décomposition ci-dessus (c'est à dire en utilisant `mulpu` pour calculer les termes A, B et C). Testez votre fonction en comparant ses résultats avec `mulpu`.
2. Testez les performances de `mulpuk1` sur des polynômes de degrés variables bien choisis, et comparez les résultats avec `mulpu`.

Q.4 :

1. Écrivez une fonction récursive `mulpukr` qui calcule le produit de deux polynômes de même degré en appliquant la décomposition ci-dessus tant que le degré du résultat est supérieur à un certain seuil `DEG_THRESHOLD` que vous définirez. Testez votre fonction en comparant ses résultats avec `mulpu`.
2. Testez les performances de `mulpukr` sur des polynômes de degrés variables bien choisis, ainsi qu'avec des valeurs variables pour `DEG_THRESHOLD`.
3. Quelle valeur de seuil vous donne les meilleurs résultats ? Celle-ci dépend-elle du degré initial de vos entrés ?

N.B. Vous devriez être capable de multiplier des polynômes de degré supérieur à 1 000 000 en un temps raisonnable.