CRO - TP - BigInt : Arithmétique en grande précision

Durée : 4 heures sur machine, 17 juillet 2024

Objectif pédagogique : L'objectif de ce TP est d'une part de vous faire prendre conscience que 32 bits pour des entiers, ce n'est pas beaucoup et d'autre part de vous sensibiliser à une méthode de programmation "propre", tout en abordant un problème algorithmique non trivial. Enfin ce TP fait travailler la manipulation de tableaux dynamique en C. Cette méthode consiste en les principes suivants :

- Coder proprement (noms de variables explicites, petites fonctions, peu de commentaires, conception préliminaire : algorithme et structure de données, méthodes d'accès aux structures de données "à la objet")
- Tester les fonctions dès que possible
- Utiliser une infrastructure de programmation adaptée
- Ne pas hésiter à restructurer...

Le langage C n'est pas le plus adapté pour ces objectifs, vous étudierez ces concepts plus en profondeur dans d'autre cours ou projets. Cependant, nous estimons qu'il est important que vous commenciez dès maintenant à suivre ces préceptes de programmation.

1 Déroulement du TP

On souhaite implémenter une bibliothèque de calcul sur des grands entiers (de l'ordre de 100 chiffres, mais cette taille sera paramétrable à la compilation). Le but du TP est de réaliser cette bibliothèque (addition, multiplication de grand entiers) de manière guidée afin de suivre *ce qui pourrait être* une bonne méthode de programmation.

Une partie des fonctions de base utiles vous est fournie sur Moodle. Des tests que doivent valider les fonctions que vous développerez vous sont aussi fournis. Dans un projet plus professionnel, vous seriez amenés à utiliser des bibliothèque dédiées à la mise en place de tests (par exemple `JUnit` pour Java). Une idée importante venant des technologies de développement moderne est qu'il faut *développer les tests avant ou en même temps que les fonctions.* Cela permet :

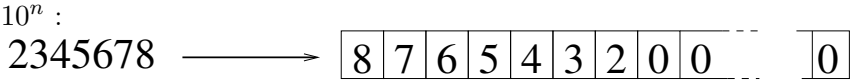
1. d'avoir un cycle de "écriture/test" des fonctions qui est court (quelques minutes à quelques heures) et donc de debugger des petites fonctions ;
2. (surtout) de pouvoir améliorer et *refactorer* le code de manière sûre, c'est-à-dire en s'assurant que les fonctionnalités sont conservées.

Le *refactoring* vient du fait que la première mouture du code n'est jamais la meilleure, il ne faut donc pas avoir peur de remettre en cause la structure ou la décomposition fonctionnelle d'un programme. Il faut donc avoir un système de test qui permet d'avoir confiance dans les modifications que l'on apporte à son programme.

Un environnement de développement tel que Eclipse peut aider pour un projet logiciel de grande envergure, cependant la prise en main de cet environnement nécessiterait plusieurs séances, nous travaillerons donc, comme d'habitude, en éditeur de texte avec un Makefile.

2 Analyse du problème

On doit donc implémenter une bibliothèque de calcul sur des grands entiers. On choisit de représenter un nombre entier par un tableau d'entiers dont chaque case correspondra à un chiffre. Par commodité, les chiffres d'un nombre seront stockés de la *gauche* vers la *droite* dans le tableau, donc dans l'ordre inverse de la notation habituelle des entiers (ca s'appelle l'*endianness* ou le boutisme – `https://en.wikipedia.org/wiki/Endianness` –, c'est pour faciliter les algorithmes avec propagation de retenue tels que l'addition par exemple). En résumé, la case n contient le coefficient de 10^n :



Les tableaux auront une taille égale à `TAILLEMAX=100` (mais bien sûr votre programme devra fonctionner si on change la valeur de `TAILLEMAX`) et on ne codera **que des entiers positifs**. Les grands entiers seront toujours stockés dans des tableaux de taille `TAILLEMAX` mais ces tableaux seront manipulés par l'intermédiaire d'un pointeur vers un entier. On utilisera donc le type suivant `bigInt` pour les grands entiers :

```
typedef int *bigInt;
```

Lors du démarrage d'un tel projet, il est important de choisir une convention pour l'allocation et la libération de la mémoire. Une convention possible est que, pour les fonction renvoyant comme résultat un `bigInt` à partir d'autre(s) `bigInt` (par exemple la fonction qui va additionner deux `bigInt`), le résultat est passé en paramètre de la fonction. Cela est possible puisqu'un `bigInt` est un pointeur sur un `int`, la fonction peut donc modifier son argument, la convention est donc **que le résultat d'une fonction aura été alloué par la fonction appelante**.

Une analyse rapide du problème fait apparaître des composants (fonctions) de base très souvent utilisés. La définition du nouveau type `bigInt` nécessite automatiquement des fonctions d'initialisation, de destruction et d'affichage des objets de ce type, ainsi que des fonctions de lecture et d'écriture dans des fichiers ou dans des chaînes de caractères. D'autre part, on va avoir besoin de fonctions de conversion entre le type `int` et le type `bigInt`, ainsi qu'une fonction donnant le nombre de chiffres d'un `int` ou d'un bigInt. Ces fonctions vous sont fournies (voir section suivante).

3 Prise en main du code

QUESTION 1 ▶ Telecharger la tarball `TP_GL.tar` sur moodle. Extrayez l'archive, faites `make`.

Dans l'archive disponible sous Moodle, un certain nombre de fonctions sont déjà programmées. le Makefile est donné ainsi qu'un mécanisme de test que les fonctions que vous devez implémenter doivent passer. Voici les fonctions déjà implémentées :

1. Manipulations basiques du type `bigInt` (fichier `util_bigInt.c`) :

```
bigInt initBigInt();              //allocation et mise à 0
void freeBigInt(bigInt bN);       //libération mémoire
void zeroBigInt(bigInt bN);       // mise à 0
int getNbChiffreInt(int N);       //renvoie le nombre de chiffres d'un entier
int getNbChiffreBigInt(bigInt bN) //renvoie le nombre de chiffres d'un bigInt
void printBigInt(bigInt bN);      // affichage à l'écran
char *sprintBigInt(bigInt bN);    //"impression" dans une chaine de caractères
void copyBigInt(bigInt src,bigInt dst); //copie src dans dst (déjà alloué)
```

```
int bigIntEqualQ(bigInt bN1,bigInt bN2); //teste l'égalité de deux bigInt
void intToBigInt(int N,bigInt bN);  //traduit l'entier N en bigInt bN (déjà alloué)
int bigIntToInt(bigInt bN);        // renvoie l'entier N corrrespondant au bigInt bN
void stringToBigInt(char *N_s,bigInt bN); //equivalent de atoi() pour les bigInt
```

2. Lecture/écriture dans des fichiers (fichier `io_bigInt`) :

```
int readBigInt(FILE *fich,bigInt res);
int writeBigInt(FILE *fich, bigInt bN);
```

3. enfin les prototypes des fonctions que vous devez programmer sont donnés :

```
int addBigInt(bigInt bN1,bigInt bN2,bigInt res);
int mulBigInt(bigInt bN1,bigInt bN2,bigInt res);
```

QUESTION 2 ▶ Passer rapidement en revue les fichiers `util_bigInt.c`, et `io_bigInt.c` pour voir ces fonctions. Un certain nombre de tests unitaires ont été effectués pour ces fonctions,

— faites `make testUtils` et étudiez les fichiers `testUtils.c` et `main_testUtils.c`.
— Étudier les fichiers `testBigInt` et `main_testBigInt` pour comprendre le mécanisme de test qui a été mis en place pour tester votre bibliothèque.

Ce système de test est primaire, il faut comprendre que l'écriture de test exhaustifs pour toutes les fonctions même les plus petites est assez fastidieux (écrire les `.h`, définir les règles de Makefile etc.). L'idée des environnements de test est que cette procédure est plus ou moins toujours la même et beaucoup d'opérations peuvent être automatisées. Il existe aujourd'hui des environnement permettant de gérer des tests de manière plus pérenne comme `JUnit` pour Java (`http://junit.sourceforge.net/`). Il existe un `CUnit` pour C (`http://cunit.sourceforge.net/`) mais son utilisation n'est pas triviale, nous ne l'utilisons pas ici.

QUESTION 3 ▶ Vous allez maintenant développer vos fonctions en travaillant, par exemple, avec le fichier `main.c` jusqu'à ce que vos fonctions passent les tests de `testBigInt`. Commencer par écrire l'algorithme d'addition de deux `bigInt`, puis implémenter la fonction `addBigInt` selon l'algorithme que vous avez conçu. Vérifier que la fonction valide tous les tests pour l'addition. Faire valider par l'enseignant.

QUESTION 4 ▶ On vous propose l'algorithme de multiplication de deux `bigInt` ci-dessous. Attention les opérations de cet algorithme (addition, multiplication par 10) sont à faire sur des **bigint**. D'autre part méfiez-vous du fait que votre fonction `addBigInt` n'a peut-être pas été prévue pour avoir deux arguments (un des opérande et le résultat par exemple) qui pointent vers le même tableau.

```
mulBigInt(bigInt bN1,bigInt bN2,bigInt res)
bigInt bNTemp ;
DEBUT
    overflow=0
    POUR i VARIANT de 0 à nbChiffre(bN1) FAIRE
        bNTemp=0 ;
        POUR j VARIANT de 0 à bN1[i]-1 FAIRE
            bNTemp = bN2 + bNTemp ;
        POUR j VARIANT de 0 à i-1 FAIRE
            bNTemp=10*bNTemp ;
        res = res + bNTemp ;

    FINPOUR
    liberer(bNTemp)
    return overflow // mettre overflow à jour a chaque operation
FIN
```

QUESTION 5 ▶ Vérifiez que vous comprenez bien l'algorithme ci-dessus, déroulez-le "à la main"pour une multiplication simple si vous n'êtes pas sûr. Implémentez la fonction `mulBigInt` selon cet algorithme (ou un que vous avez conçu). Validez cette fonction avec les fonctions de test données

Changez la taille maximum des `bigInt` (`TAILLEMAX` dans `type bigInt.h`, passez la à 1000). Attention, à partir de là, vérifiez bien que vous libérez les entiers que vous avez alloués quand vous n'en avez plus besoin, si vous avez des messages bizarres qui concernent `malloc`, c'est que vous avez saturé la mémoire de votre processeur, il faut redémarrer.

QUESTION 6 ▶ Programmer une factorielle de `BigInt`. Jusqu'à quel entier pouvez vous la tester (utiliser le fichier `dataTestFactorielle.txt`, par exemple en écrivant une fonction `testUnopFile`) ?

QUESTION 7 ▶ Avez-vous testé ce qu'il se passe lors d'un overflow ? Testez avec la factorielle en repassant `TAILLEMAX` à 100

QUESTION 8 ▶ Mesurez le temps de calcul pour 150 ! (avec `TAILLEMAX=1000`). Utiliser la commande `time` : `time ./main`, comparez avec votre voisin.

4 Si vous avez terminé

En admettant que le problème des entiers négatifs se résoud facilement, le gros problème de notre librairie est qu'elle présuppose que l'on connaisse la taille maximum d'un entier que l'on va manipuler. Les librairies existantes, dites multi-précision (GMP, `https://gmplib.org/` par exemple), ne présupposent pas de borne supérieure, hormis les limites physiques de la mémoire de l'ordinateur sur laquelle s'exécute votre programme bien sûr. Ces librairies utilisent l'allocation dynamique à base de listes chaînées.

QUESTION 9 ▶ Proposez un système d'arithmétique entière multi-précision à base de listes chaînées qui ne suppose pas de taille maximum des entiers manipulés, et implémentez les algorithmes d'addition et de multiplication sur ce nouveau type (on pourra se limiter aux entiers positifs à nouveau).