

TP4 : Diviser pour régner

Le but de ce TP est d'implanter des algorithmes de type « diviser pour régner » vus en cours. On fournit une trame de code pour les deux premiers exercices dans les fichiers TP4Exo1.cc et TP4Exo2.cc. Des fonctions de base d'initialisation et de gestion des tableaux sont fournies dans les fichiers TP4.cc et TP4.h. Ces deux fichiers ainsi que le Makefile, à ne pas modifier, sont communs aux deux exercices.

Le troisième exercice, difficile, et en bonus. Aucune trame de code n'est fournie.

Exercice 1.

Tri fusion

Le but de cet exercice est d'implanter le tri fusion, comme étudié en cours. À chaque question, testez impérativement ce que vous venez de faire sur le tableau fixé proposé (choix 1) et vérifiez que le résultat est correct !

1. Commencer à compléter la fonction `trifusion(int n, int T[])` : remplir les tableaux T1 et T2.
2. Compléter la fonction `fusion(int n1, int n2, int T1[], int T2[], int T[])` qui fusionne les tableaux T1 et T2 de tailles respectives n1 et n2 dans le tableau T.
3. Finir de compléter la fonction `trifusion(int n, int T[])`.
4. Implémenter la fonction `void tribulles(int n, int T[])`, qui trie le tableau T à l'aide d'un tri à bulles. On pourra faire appel à la fonction `swap(a, b)` de la librairie standard qui échange le contenu des variables *a* et *b* en temps $O(1)$.
5. Comparer les temps mis par le tri fusion et le tri à bulle. Augmenter la taille des tableaux à traiter pour observer une différence importante.
6. (bonus) Programmer une nouvelle version de l'algorithme TRIFUSION qui permet de trier sans garder en mémoire des tableaux (à part le tableau de départ) à chaque appel récursif. Pour cela il faut préciser aux différentes fonctions sur quel intervalle du tableau *T* elle doivent travailler. Ainsi, par exemple la signature de la fonction de tri pourra être : `void trifusion(int n, int a, int b, int T[])` signifiant que lors de l'appel de cette fonction, on cherchera à trier les éléments rangés entre les indices *a* et *b* (compris) du tableau *T*. L'appel initial sera `trifusion(n, 0, n-1, T)` ; Il faudra modifier de même la fonction `fusion`, en s'autorisant ici à créer temporairement un tableau intermédiaire permettant d'effectuer la fusion, ce tableau devant être libéré à la fin de la fonction `fusion`.

Exercice 2.

Calcul de rang

Le but de cet exercice est d'implémenter un calcul de rang linéaire, comme étudié en cours. Tester vos programmes sur l'exemple fixe proposé avant de les faire tourner sur des tableaux de tailles plus importantes.

1. On propose deux choix possibles pour `choixPivot` : un pivot fixe (`T[0]`) et un pivot aléatoire. Compléter la fonction `int choixPivot(int n, int T[], pivot P)` pour implanter le choix de pivot aléatoire.
2. La tâche principale est l'implantation de la fonction `int rang(int k, int n, int T[], pivot P)` permettant de trouver le $k^{\text{ème}}$ rang du tableau *T* de taille *n*.
 - (a) Compléter le calcul de `ninf` et `neq`.
 - (b) Compléter le cas 1.
 - (c) Compléter le cas 2.
 - (d) Compléter le cas 3.
3. Comparer les temps de calcul et les nombres de choix de pivots effectués : voit-on une différence sur un tableau aléatoire ? sur un tableau initialement trié (ordre croissant ou décroissant) ?

4. (*bonus*) Pour aller plus loin : écrire une fonction void `rang-par-tri(int k, int n, int T[])` qui calcule le $k^{\text{ème}}$ rang du tableau T en le triant (avec `tri-fusion` par exemple) et affiche sa valeur. Comparer les temps d'exécution de `rang` et de `rang-par-tri`.

Exercice 3.

Multiplication d'entiers (bonus)

L'objectif de cet exercice, pour lequel aucune trame de code n'est fournie, est d'implanter la multiplication d'entiers par l'algorithme naïf et celui de Karatsuba et de comparer leurs performances.

Pour cela, on représentera un entier comme un tableau de chiffres, chaque chiffre étant un entier non signé de 32 bits. Ainsi, un tableau de n chiffres représentera un entier écrit en base 2^{32} . Formellement, on définit :

```
typedef uint32_t chiffre;  
typedef entier chiffre*;
```

Il vous faudra implanter (au minimum) les fonctions suivantes :

1. `int somme(int nX, int nY, entier X, entier Y, entier Z)` qui effectue la somme des entiers X (de taille nX) et Y (de taille nY) et stocke le résultat dans Z. La valeur renvoyée est la taille du résultat.
2. `void mul_chiffres(chiffre x, chiffre y, entier z)` qui prend en entrée deux chiffres x et y et renvoie leur produit, sous forme d'un entier à deux chiffres. *On pourra transformer les chiffres en `uint64_t`, faire leur produit, et enfin reconstruire l'entier de deux chiffres correspondant.*
3. `int produit_naif(int nX, int nY, entier X, entier Y, entier Z)` qui effectue le produit des entiers X et Y via l'algorithme naïf et stocke le résultat dans Z. La valeur renvoyée est encore la taille du résultat.
4. `int produit_kara(int nX, int nY, entier X, entier Y, entier Z)` qui effectue le produit des entiers X et Y via l'algorithme de Karatsuba et stocke le résultat dans Z. La valeur renvoyée est toujours la taille du résultat.