TP 3 : Révisions sur les tris

Langage C (LC4)
Semaine du 15 février 2010

1 Introduction

Pour ce TP, il faut créer les fonctions de tri dans `sort_testbed.c`. La compilation se fait comme suit :
`gcc -W -Wall -Wextra -o sort_testbed sort_testbed.c`
L'exécution se fait de deux façons :
`./sort_testbed <array size>`
`./sort_testbed <array size> <seed>` pour tester sur différents tableaux aléatoires.

2 Tri à bulles

Nous allons commencer par un tri simple, le tri à bulles. L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est triée : l'algorithme peut s'arrêter.

Question 1. Écrire une fonction `void bubble_sort(int n, int table[])` qui trie le tableau `table`.

Question 2. Quelle est sa complexité : dans le meilleur des cas ? dans le pire des cas ? en mémoire utilisée ?

3 Tri rapide

Le tri rapide est un algorithme de tri très utilisé, car efficace et en place. La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Question 3. Écrire une fonction `void quick_sort(int n, int table[])` qui trie le tableau `table`.
Indication : on pourra utiliser une fonction auxiliaire `void quick_sort_internal(int table[], int index_start, int index_end)`, qui, s'il y a plus d'un élément dans l'intervalle délimité par `index_start` et `index_end`, itère sur le tableau et échange les éléments

de telle sorte que ceux avant le pivot soient plus petit que lui et ceux après soient plus grand. Cette fonction s'appelle ensuite récursivement sur les intervalles entre `index_start` et `pivot` et entre `pivot + 1` et `index_end`.

Question 4. Quelle est sa complexité : dans le meilleur des cas ? dans le pire des cas ? en mémoire utilisée ?
Remarque : en `C`, il existe une fonction `qsort` dans la librairie standard.

4 Tri par insertion

Ensuite, nous allons voir un tri efficace sur les petits tableaux, le tri par insertion. Il faut parcourir le tableau dans l'ordre ; à chaque étape, on insére un des éléments restant du tableau initial à la bonne place dans le tableau final, de telle sorte que le tableau final soit toujours trié.

Question 5. Écrire une fonction `void insertion_sort(int n, int table[])` qui trie le tableau `table` en place.

Question 6. Quelle est sa complexité : dans le meilleur des cas ? dans le pire des cas ? en mémoire utilisée ?

5 Tri fusion (pour les plus rapides)

Nous allons enfin voir un dernier tri, qui lui n'est pas en place, le tri fusion. L'algorithme s'effectue récursivement. On découpe le tableau en deux parties sensiblement égales, on trie ces deux parties, puis on les fusionne.
Ici, nous aurons besoin d'allouer la mémoire dynamiquement. Pour allouer un tableau de taille `n\`, il faut faire :

```
int array[] = malloc(n * sizeof(int))
```

Il faut ensuite penser à libérer l'espace alloué :

```
free(array)
```

Question 7. Écrire une fonction `void merge_sort(int n, int table[])` qui trie le tableau `table`.
Il sera utile d'utiliser une fonction auxiliaire `merge_arrays` qui fusionne deux tableaux.

Question 8. Quelle est sa complexité : dans le meilleur des cas ? dans le pire des cas ? en mémoire utilisée ?