INFO2050 - Programmation avancée

Projet 1: Algorithmes de tri

Assistants: Jean-Michel Begon, Jean-François Grailet
Professeur: Pierre Geurts

09 octobre 2015

L'objectif du projet est d'implémenter, de comparer et d'analyser différents algorithmes de tri dans différentes conditions. L'analyse s'effectuera en deux temps. Premièrement, différents algorithmes de tri de tableau seront comparés entre eux. Ensuite, ces algorithmes devront être comparés à un algorithme efficace de tri d'une liste simplement liée.

1 Implémentation

On vous demande d'implémenter différents algorithmes de tri de tableau et un algorithme de tri de liste liée selon les instructions ci-dessous.

Tri de tableau

Nous vous fournissons comme exemple l'algorithme BubbleSort et nous vous demandons d'implémenter les algorithmes suivants:

(a) L'algorithme InsertionSort (à implémenter).

(b) L'algorithme QuickSort standard (à implémenter).

(c) L'algorithme QuickSort avec un pivot aléatoire (à implémenter).

(d) L'algorithme HeapSort (à implémenter).

Tri d'une liste liée

Rappel: Un nœud de liste est une structure possédant au moins deux champs:

`data`: La donnée contenue au nœud.
`next`: Un pointeur vers un autre nœud.

Une liste liée est une suite récursive de telles structures; la tête est un nœud et son champ `next` pointe vers une autre liste: la liste commençant au second élément. Le champ `next` du dernier maillons vaut NIL, indiquant la fin de la liste.

Dans la suite, on ne considérera que des listes "figées" (*frozen list*). C'est à dire des listes dont le champ `data` est immuable; une fois initialisé, il ne peut être changé.

Les algorithmes de tri du cours théorique étant adaptés aux tableaux, ils supposent que l'accès à n'importe quel élément se fait en temps constant. A contrario, l'accès est séquentiel dans le cas d'une liste liée: pour atteindre le 3e élément, il faut passer par le premier, puis par le second. Au vu de cette contrainte, il vous est demandé d'implémenter un algorithme de tri *efficace* (en moyenne) en temps et *en place*, c'est-à-dire $\Theta(1)$ en espace (en négligeant le coût des appels récursifs).

Remarque: La contrainte d'immuabilité couplée à celle d'espace impose notamment que

- On ne peut pas simplement permuter les données contenues dans le champ `data` de deux nœuds.
- On ne peut pas allouer un tableau, déverser le contenu de la liste dedans, le trier et reconstruire une nouvelle liste sur base de celui-ci.

Il s'agit donc de réorganiser les nœuds de la liste liée, en changeant les pointeurs `next`, afin que ses éléments soient triés. Le tout sans perdre de nœuds ni créer de cycles.

Fichiers fournis

Les fichiers suivants vous sont fournis:

- `Array.h` et `Array.c`: une petite bibliothèque pour générer différents types des tableaux.
- `SortArray.h`: le header contenant l'interface de la fonction de tri de tableaux.
- `BubbleSortArray.c`: une implémentation répondant à l'interface contenue dans `SortArray.h` utilisant l'algorithme BubbleSort.
- `mainArray.c`: une application prenant en entrée une taille de tableau et évaluant le temps de tri pour les trois conditions.
- `FrozenList.h` et `FrozenList.c`: l'interface et l'implémentation des listes liées d'entiers.
- `SortFrozenList.h`: le header contenant l'interface de la fonction de tri de listes liées.
- `mainFrozenList.c`: une application prenant en entrée une taille de liste et évaluant le temps de tri pour les trois conditions.

`FrozenList` Afin d'encapsuler la structure de liste liée, les champs des nœuds ne sont pas directement accessible. L'interface fournit donc des fonctions pour y accéder:

`getData`: renvoie la valeur du champ `data`. C'est la valeur à utiliser pour trier la liste.
`getTail`: renvoie la valeur du champ `next`.
`setTail`: modifie la valeur du champ `next`.

2 Analyse théorique

Dans votre rapport, nous vous demandons de répondre aux questions suivantes:

1.a. Donnez le pseudo-code de votre algorithme de tri pour listes liées.
1.b. Etudiez sa complexité en temps dans le meilleur et dans le pire cas en fonction de la taille N de la liste liée. Expliquez à quoi correspondent les meilleur et pire cas et justifiez la complexité que vous obtenez dans ces deux cas.
1.c. Est-ce que cet algorithme de tri est stable ? Justifiez brièvement votre réponse.

3 Analyse expérimentale

Dans votre rapport, nous vous demandons également de répondre aux questions suivantes:

2.a Reportez dans le tableau ci-dessous les temps de tri moyens des différents algorithmes dans les trois conditions suivantes:

- Les données sont toutes différentes et dans un ordre aléatoire.
- Les données sont toutes différentes et presque triées.
- Il y a peu d'éléments différents mais dans un ordre aléatoire.

Tableau	aléatoire	presque trié	peu d'elem. uniques
BubbleSort InsertionSort QuickSort (standard) QuickSort (pivot aléatoire) HeapSort			
ListSort			

2.b Commentez les performances des différents algorithmes dédiés aux tableaux.
2.c Commentez les performances de l'algorithme dédié aux listes par rapport aux autres.

Remarques:

- Vous devez choisir une taille de tableau pertinente pour l'analyse.
- Les fonctions pour générer les tableaux vous sont fournies dans le fichier `Array.c`.
  - Les paramètres `nbSwaps` et `nbUniques` peuvent être choisis comme valant la racine carrée du nombre d'éléments à trier.
- Les temps reportés doivent être des temps moyens établis sur base de 10 expériences.

4 Deadline et soumission

Le projet est à réaliser **individuellement** pour le **dimanche 25 octobre 2015** à **23h59** au plus tard. Le projet est à remettre via la plateforme *cicada* : http://cicada.run.montefiore.ulg.ac.be/. Il doit être rendu sous la forme d'une archive `tar.gz` contenant:

(a) Votre rapport (5 pages *maximum*) au format PDF. Soyez bref mais précis et respectez bien la numération des (sous-)questions.

(b) Le fichier `InsertionSortArray.c` répondant à l'interface de `SortArray.h` et implémentant le tri par l'algorithme du InsertionSort.

(c) Le fichier `QuickSortArray.c` répondant à l'interface de `SortArray.h` et implémentant le tri par l'algorithme du QuickSort standard.

(d) Le fichier `RandQuickSortArray.c` répondant à l'interface de `SortArray.h` et implémentant le tri par l'algorithme du QuickSort avec pivot aléatoire.

(e) Le fichier `HeapSortArray.c` répondant à l'interface de `SortArray.h` et implémentant le tri par l'algorithme du HeapSort.

(f) Le fichier `SortFrozenList.c` répondant à l'interface de `SortFrozenList.h` et implémentant le tri par un algorithme efficace en moyenne (*cf.* section 1).

Vos fichiers seront évalués sur les machines ms8xx avec les commandes:

```
gcc Array.c BubbleSortArray.c mainArray.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes
-lm -o array
```

```
gcc Array.c SortFrozenList.c FrozenList.c mainFrozenList.c --std=c99 --pedantic -Wall -Wextra
-Wmissing-prototypes -lm -o list
```

Où `BubbleSortArray.c` est à remplacer par l'algorithme adéquat.

Ceci implique que:

- Le projet doit être réalisé dans le standard C99.
- La présence de *warnings* impactera négativement la cote finale.
- Un projet qui ne compile pas avec cette commande sur ces machines recevra une cote nulle (pour la partie code du projet).

Un projet non rendu à temps recevra une cote globale nulle. En cas de plagiat avéré, l'étudiant se verra affecter une cote nulle à l'ensemble des projets.

Les critères de correction sont précisés sur la page web des projets.

Bon travail !