Structures de données et algorithmes
Projet 1: Algorithmes de tri

Pierre GEURTS – Jean-Michel BEGON

26 février 2016

L'objectif du projet est d'implémenter, de comparer et d'analyser empiriquement plusieurs algorithmes de tri.

1 Blocs pré-triés

Dans beaucoup d'applications réelles, les tableaux qu'on a à trier sont constitués de sous-tableaux déjà triés. Il serait donc intéressant de mettre au point un algorithme de tri aussi efficace que possible tirant parti de cette propriété.

Sur base de cette idée, on se propose d'étudier deux nouveaux algorithmes de tri itératif inspirés du MERGESORT.

1.1 NEWSORT1

Le principe est suivant. On commence par chercher l'indice i maximal tel que le sous-tableau $A[1\,..\,i]$ est trié. Ensuite on itère comme suit:

(a) On recherche l'indice j maximal tel que le sous-tableau $A[i+1\,..\,j]$ soit trié.

(b) On fusionne $A[1\,..\,i]$ et $A[i+1\,..\,j]$ (comme le fait la fonction MERGE vue au cours théorique).

(c) Tant que le tableau n'est pas complètement trié, on recommence en (b) en prenant $i = j$.

Exemple: Soit le tableau suivant: $[1, 5, 2, 6, 4, 3, 9]$. Les étapes de fusion seront les suivantes (la partie verte est fusionnée avec la partie rouge pour donner la partie bleue):

- $[1, 5, 2, 6, 4, 3, 9] \Rightarrow [1, 2, 5, 6, 4, 3, 9]$
- $[1, 2, 5, 6, 4, 3, 9] \Rightarrow [1, 2, 4, 5, 6, 3, 9]$
- $[1, 2, 4, 5, 6, 3, 9] \Rightarrow [1, 2, 3, 4, 5, 6, 9]$

1.2 NEWSORT2

Le principe est le suivant. On parcourt d'abord le tableau en partant de la gauche et se faisant, on le découpe en blocs consécutifs constitués chacun d'éléments triés. Ensuite, on itère comme suit:

(a) On fusionne chaque paire de blocs consécutifs triés pour former un nouveau bloc trié (au moyen de la fonction MERGE). S'il y a un nombre impair de blocs, le dernier reste inchangé.

(b) On recommence tant qu'il y a plus qu'un seul bloc.

Exemple: Soit le tableau suivant: $[1, 5, 2, 6, 4, 3, 9]$. La première étape identifie 4 blocs consécutifs triés: $[1, 5]$,$[2, 6]$, $[4]$, et $[3, 9]$. Les étapes de fusion seront ensuite les suivantes (le rouge fusionne avec le bleu pour donner le magenta, le vert fusionne avec le cyan pour donner le bleu clair):

- $[1, 5, 2, 6, 4, 3, 9] \Rightarrow [1, 2, 5, 6, 3, 4, 9]$
- $[1, 2, 5, 6, 3, 4, 9] \Rightarrow [1, 2, 3, 4, 5, 6, 9]$

2 Analyse théorique

(a) Écrivez en pseudo-code une fonction NewSort2 implémentant l'algorithme décrit ci-dessus. Vous pouvez supposer que la fonction Merge telle que définie au cours théorique est connue et vous pouvez également utiliser toutes les structures élémentaires vues au cours sans les redéfinir[1].

(b) Étudiez les complexités en *temps* et en *espace* des algorithmes NewSort1 et NewSort2 au pire et au meilleur cas. Expliquez précisément à quoi correspondent les pire et meilleur cas.

(c) Sur base de l'analyse de complexité, justifiez le choix de se focaliser sur l'algorithme NewSort2 dans l'analyse expérimentale.

(d) Est-ce que ces algorithmes de tri sont stables ? Justifiez brièvement votre réponse.

(e) Quelle est la complexité au pire cas de NewSort2 si le tableau de taille n est constitué de $k(\leq n)$ blocs pré-triés de *taille identique* ? Par exemple, le tableau suivant est constitué de 4 blocs pré-triés de taille 3:

$$[5, 6, 7, 1, 8, 9, 2, 3, 11, 12, 15, 16]$$

3 Analyse expérimentale

3.1 Implémentation

(a) Implémenter l'algorithme NewSort2 dans un fichier `NewSort.c`.

(b) Implémenter l'algorithme MergeSort dans un fichier `MergeSort.c`.

(c) Implémenter l'algorithme QuickSort avec pivot aléatoire dans un fichier `QuickSort.c`.

(d) Implémenter l'algorithme HeapSort dans un fichier `HeapSort.c`.

Les quatre implémentations doivent respecter l'interface de tri décrite dans le fichier `Sort.h`. Chaque tri doit être implémenté dans un fichier qui lui est propre.

Exemple avec le InsertionSort

Afin de vous aider à prendre rapidement en main le code mis en place, nous avons implémenté à titre d'exemple l'algorithme InsertionSort dans le fichier `InsertionSort.c` ainsi qu'un petit programme d'essai `main.c`. Nous vous fournissons également les fichiers `Array.c` et `Array.h` afin de générer les tableaux d'entiers.

Pour compiler le programme, vous pouvez utiliser la commande suivante:

```
gcc main.c Array.c InsertionSort.c --std=c99 -o test
```

Ce programme ne prend aucun argument: `./test`

[1] Vous devez cependant redéfinir ces fonctions si les arguments et les valeurs retournées ne sont pas strictement les mêmes que pour les fonctions vues au cours.

3.2 Temps d'exécution sur des tableaux aléatoires

(a) Soit n le nombre de données à trier dans un tableau. Calculez empiriquement le temps d'exécution moyen des différents algorithmes pour différentes valeurs de n (10, 100, 1.000, 10.000, 100.000 et 1.000.000) lorsque les tableaux sont générés de manière complètement aléatoire. La moyenne doit être obtenue sur un ensemble de 20 expériences. Reportez ces résultats dans une table au format donné ci-dessous.

n	InsertionSort	MergeSort	QuickSort	HeapSort	NewSort2
10					
100					
1.000					
10.000					
100.000					
1.000.000					

(b) Commentez ces résultats en comparant les algorithmes:

- les uns par rapport aux autres;
- par rapport à leur complexité théorique.

Notes:

- Une fonction `createRandomArray` vous est fournie pour générer un tableau aléatoire de taille n.
- Le temps d'exécution est une valeur peu précise qui dépend fortement des capacités de l'ordinateur mais également de l'état d'utilisation de celui-ci au moment des expériences. Pour limiter cet effet, il vous est conseillé de réaliser toutes vos mesures de manière séquentielle sur la même machine.

3.3 Temps d'exécution sur des tableaux de blocs pré-triés

(a) Pour une taille de tableau $n = 5000$, calculez empiriquement le temps d'exécution moyen des algorithmes lorsque le tableau à trier contient k blocs pré-triés, pour des valeurs de k croissantes. Comme pour le tableau précédent, on reportera dans le tableau ci-dessus les temps moyens sur 20 expériences.

k	InsertionSort	MergeSort	QuickSort	HeapSort	NewSort2
1					
20					
100					
500					
1000					
5000					

Note: Une fonction `createRandomBlockArray` vous est fournie pour générer un tableau aléatoire de blocs pré-triés.

(b) Commentez ces résultats en comparant les algorithmes les uns par rapport aux autres.

(c) Concluez sur l'intérêt ou non de NewSort2 par rapport aux autres algorithmes de tri.

4 Deadline et soumission

Le projet est à réaliser **individuellement** pour le ~~**21 mars 2016**~~ **25 mars 2016** à **23h59** au plus tard. Le projet est à remettre via la plateforme *cicada* : http://submit.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) Un fichier `NewSort.c` contenant l'implémentation de l'algorithme du même nom.

(c) Un fichier `MergeSort.c` contenant l'implémentation de l'algorithme du même nom.

(d) Un fichier `QuickSort.c` contenant l'implémentation de l'algorithme du même nom.

(e) Un fichier `HeapSort.c` contenant l'implémentation de l'algorithme du même nom.

Respectez bien les extensions de fichiers ainsi que les noms des fichier `*.c` (en ce compris la casse). Seule la dernière archive soumise sera prise en compte.

Vos fichiers seront évalués sur les machines ms8xx avec la commande:

```
gcc main.c Array.c InsertionSort.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -o test
./test
```

En substituant adéquatement l'algorithme de tri. 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 recevra une cote nulle.

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

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

Bon travail !