Structure de données et algorithmes

Projet 1: Algorithmes de tri

23 février 2024

L'objectif du projet est d'analyser différents algorithmes de tri de manière empirique afin de les caractériser le plus finement possible et d'ensuite écrire une fonction permettant de rendre stable un algorithme de tri qui ne le serait pas nécessairement.

1 Qui est qui ?

Une librairie en c vous est fournie qui implémente 7 algorithmes de tri. Vous ne disposez cependant que d'une version précompilée de cette librairie et vous avez perdu également la documentation de cette dernière. Tout ce que vous savez de cette librairie, c'est qu'elle fournit une fonction au format suivant:

```
void sort(size_t algo, void *array, size_t length,
          int (*compare)(const void *array, size_t i, size_t j),
          void (*swap)(void *array, size_t i, size_t j))
```

où `algo` est un choix d'algorithme entre 0 et 6 inclus, `array` est un pointeur vers le début du tableau à trier, `length` est sa longueur, `compare` est une fonction renvoyant un entier respectivement plus petit, égal ou plus grand que 0 si l'élément à la position `i` dans `array` est plus petit, égal ou plus grand que l'élément à la position `j` et `swap` est une fonction permettant d'échanger les éléments du tableau aux positions `i` et `j`. Par exemple, on peut utiliser cette fonction pour trier un tableau d'entiers de la manière suivante:

```
void swap_int(void *array, int i, int j) {
   int temp = ((int*)array)[i];
   ((int*)array)[i] = ((int*)array)[j];
   ((int*)array)[j] = temp;
}

int compare_int(void *array, int i, int j) {
   return (((int*)array)[i] - ((int*)array)[j]);
}

int array[5] = {5, 4, 3, 2, 1};

sort(0, array, 5, compare_int, swap_int);
```

Parmi les 7 algorithmes accessibles depuis cette fonction, vous savez qu'il y a les suivants: le tri par sélection, le tri par insertion, le tri à bulles, le tri par fusion, le tri rapide[1], le tri par tas et un

[1] L'implémentation utilise la méthode "median-of-3" pour la détermination du pivot et la méthode de Hoare pour la détermination de la partition.

dernier algorithme de tri inconnu, non vu au cours. Votre tâche dans ce contexte est d'identifier quel algorithme se cache derrière chaque valeur de l'argument `algo` de la fonction sort.

Dans votre rapport, on vous demande de répondre aux questions suivantes:

(1) Calculez empiriquement le temps d'exécution et le nombre de comparaisons des 7 algorithmes sur des tableaux de tailles croissantes (de 1000, 10.000 et 100.000 éléments) lorsque ces tableaux sont aléatoires, ordonnés de manière croissante ou décroissante. Reportez ces résultats dans une table au format donné ci-dessous[2].

Type de tableau	aléatoire			croissant			décroissant		
`algo`	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
0									
1									
2									
3									
4									
5									
6									

(2) Remplissez ensuite la table suivante:

	Complexités					
`algo`	meilleur cas	pire cas	cas moyen	stable?	en place?	algorithme présumé
0						
1						
2						
3						
4						
5						
6						

Expliquez précisément votre démarche pour identifier chacun des algorithmes. Si vous avez du réaliser des expériences en plus de celles de la sous-question précédente, décrivez les de manière précise et rapportez en les résultats dans le rapport.

(3) Pour les algorithmes connus, montrez brièvement que les résultats empiriques de la table de la sous-question 1 sont en accord avec les complexités théoriques que vous avez données à la sous-question 2. Si certaines complexités théoriques ne peuvent pas être confirmées par les tests de la sous-question 1, soit réalisez une expérience supplémentaire confirmant ces cas, soit expliquez pourquoi réaliser une telle expérience serait complexe ou impossible.

(4) Pour l'algorithme inconnu, essayez d'être le plus précis dans votre caractérisation de ses complexités sur base de vos tests empiriques. Comparez les caractèristiques de cet algorithme aux autres algorithmes de la table et discutez de son intérêt pratique sur base de votre analyse.

Suggestions. La possibilité de fournir à la fonction de tri votre fonction de comparaison vous permet facilement de comptabiliser le nombre de comparaisons faites par l'algorithme de tri (en passant par un compteur global). Dans le cas d'un tableau aléatoire, il est important d'effectuer une moyenne des temps d'exécution et du nombre de comparaison sur plusieurs répétitions de l'expérience (de l'ordre de 5-10). Pour déterminer la stabilité des algorithmes, vous pouvez vous servir de la fonction `isSortStable` que vous devez écrire dans la deuxième partie du projet.

[2]Vous pouvez séparer cette table en plusieurs pour plus de lisibilité.

2 Stabilisation d'un algorithme de tri

Certains des algorithmes étudiés au point précédent ne sont pas stables. Tout algorithme de tri peut être néanmoins rendu stable en maintenant sa complexité en temps, dans tous les cas, inchangée, au prix d'une augmentation éventuelle de sa complexité en espace. On vous demande d'implémenter deux fonctions dans le fichier `Stable.c`:

`bool isSortStable(size_t algo, const int *array, size_t length)`: renvoyant `true` si le tri du tableau d'entiers donné en argument par l'algorithme `algo` de la fonction `sort` serait stable. Cette fonction ne doit pas modifier le contenu du tableau `array`.

```
void stableSort(size_t algo, void *array, size_t length,
                int (*compare)(const void *array, size_t i, size_t j),
                void (*swap)(void *array, size_t i, size_t j))
```

Cette fonction prend les mêmes arguments que la fonction `sort`. Elle doit réaliser le tri du tableau `array` avec l'algorithme `algo` mais de manière stable.

Répondez aux questions suivantes dans votre rapport:

(1) Expliquez brièvement le principe de votre implémentation de `stableSort`.

(2) Expliquez comment `stableSort` affecte les complexités en temps et en espace des algorithmes de tri.

(3) Testez empiriquement l'impact sur la complexité en temps de la stabilisation en reproduisant l'expérience de la sous-question 1.1 uniquement pour les algorithmes qui n'étaient pas stables. Comparez les nouveaux résultats obtenus avec les précédents.

Suggestions. Pour implémenter ces deux fonctions, vous aurez besoin d'appeler `sort` en lui donnant des fonctions `compare` et/ou `swap` spécifiques qui devront éventuellement avoir accès au tableau et aux fonctions `compare` et `swap` données en argument aux fonctions `isSortStable` et `stableSort`. Vous êtes autorisés pour cela à stocker ces arguments dans des variables globales (statiques) introduites dans le fichier `Stable.c`.

3 Fichiers fournis

Nous vous fournissons les fichiers suivants:

- `Sort.o`: la librairie (compilée) contenant les algorithmes de tri. Plusieurs versions de ce fichier vous sont fournies sur Ecampus pour différents processeurs et systèmes d'exploitation. Vous devez renommer le fichier correspondant à votre système en Sort.o pour pouvoir l'utiliser. Si vous souhaitez une version pour d'autres systèmes ou si les versions fournies ne fonctionnent pas sur votre ordinateur, contactez nous.
- `Sort.h`: le fichier d'entête pour utiliser cette librairie.
- `Stable.h/Stable.c`: les fichiers qui devront contenir les deux fonctions à implémenter. Vous ne devez modifier que le fichier Stable.c.
- `testsorting.c`: un fichier contenant une fonction `main` permettant de tester un algorithme de tri dans sa version originale ou stabilisée sur un tableau d'entiers aléatoire. Vous pouvez partir de ce fichier pour effectuer vos propres tests.
- `Makefile`: un fichier `Makefile` permettant de compiler testsorting.c

Remarque importante. pour que l'ordre des algorithmes soient différent d'un groupe à l'autre, nous vous demandons de définir une variable globale `studentcode` au début de tous vos fichiers de test à la somme des matricules des étudiants du groupe (au format complet). Par exemple, si les deux étudiants du groupe ont pour matricule 20221234 et 20214321, la ligne à faire apparaître dans votre fichier est:

```
const int studentcode = 20221234+20214321;
```

Ce code est utilisé dans la fonction `sort` pour déterminer quel algorithme est appelé en foncion de l'argument `algo` et nous l'utiliserons également lors de la correction pour vérifier que vous avez retrouvé la bonne association d'algorithme.

4 Plagiat

En soumettant votre projet, vous attestez avoir pris connaissance des règles en matière de plagiat en vigueur dans ce cours, qui sont précisées sur Ecampus (rubrique projet). En particulier, tout code et tout rapport remis doit être le résultat de votre travail personnel uniquement. Vous n'avez reçu d'aide d'aucune tierce personne, si ce n'est les encadrants du cours, et vous n'avez pas utilisé d'outil d'intelligence artificielle pour écrire les codes que vous avez remis. Vous n'avez pas envoyé votre code à d'autres étudiants, ni reçu de code d'autres étudiants. Comme le prévoit le règlement de l'université, toute infraction à ces règles sera sanctionnée d'un zéro pour l'entièreté du cours et mènera à l'ouverture d'un dossier auprès de la faculté.

5 Deadline et soumission

Le projet est à réaliser ***seul ou en binôme*** pour le ***vendredi 22 mars 2023 à 23h59*** au plus tard. Le projet est à remettre sur Gradescope (`http://gradescope.com`, voir code cours sur Ecampus).

Les fichiers suivants doivent être remis:

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

(b) Le fichier `Stable.c` contenant votre implémentation des fonctions `isSortStable` et `stableSort`.

Vos fichiers seront compilés avec `gcc` et les flags de compilation suivants (qui sont également utilisés dans le `Makefile`):

```
--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes
```

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 recevra une cote nulle (pour la partie code du projet).

Un projet non rendu à temps recevra une cote globale nulle. Les critères de correction sont précisés sur la page web des projets.

Bon travail !