Structure de données et algorithmes

Projet 1: Algorithmes de tri

24 février 2023

L'objectif du projet est d'implémenter, de comparer et d'analyser plusieurs algorithmes de tri, dont une variante adaptative du tri par fusion.

Un tri par fusion adaptatif

Bien qu'optimaux, un inconvénient des algorithmes de tri par fusion et par tas est qu'ils ne sont pas adaptifs, c'est-à-dire qu'ils ne sont pas capables d'exploiter le fait que certaines parties, ou l'entiéreté, du tableau peuvent déjà être triées pour obtenir une meilleure complexité. Leur meilleur cas est en effet $\Theta(N \log N)$. On se propose dans ce projet d'étudier une variante adaptative du tri par fusion. L'idée générale de cet algorithme est de partitionner le tableau de départ en sous-tableaux triés les plus grands possible et d'ensuite fusionner deux à deux ces sous-tableaux comme on le ferait dans le cadre du tri par fusion classique. Ces deux étapes, le découpage et la fusion, sont décrites de manière détaillée ci-dessous.

Découpage en sous-tableaux triés. Le découpage en sous-tableaux triés passera par une fonction FINDRUN$(A, start, minSize)$ qui devra renvoyer une valeur entière end ($\geq start + minSize - 1$)) telle que le sous-tableau $A[start\,..\,end]$ soit trié. L'idée de cette fonction est de déterminer d'abord la position i ($\geq start$) maximale telle que $A[start\,..\,i]$ soit trié. Si la longueur de ce sous-tableau, c'est-à-dire $i - start + 1$, est inférieure à $minSize$, on étendra cette partie triée en insérant les $start + minSize - i$ valeurs suivantes dans le tableau une à une à leur position dans le sous-tableau $A[start\,..\,start + minSize - 1]$ comme le ferait le tri par insertion. Au cas où il resterait moins de $minSize$ éléments dans le tableau, on s'arrêtera évidemment à la position N, qui sera la valeur renvoyée par la fonction.

Par exemple, pour le sous-tableau suivant:

$$A = [3, 4, 5, 7, 8, 3, 2, 1, 9],$$

un appel[1] à FINDRUN$(A, 1, 1)$ renverra 5 parce que le sous-tableau $A[1\,..\,5]$ est le plus grand sous-tableau trié commençant à 1 et de taille au moins 1. Un appel à FINDRUN$(A, 3, 4)$ renverra 6 et modifiera le tableau A:

$$A = [3, 4, 3, 5, 7, 8, 2, 1, 9],$$

en insérant 3 à la bonne position dans le sous-tableau $A[3\,..\,6]$ pour satisfaire la contrainte de l'argument $minSize = 4$.

L'algorithme de découpage tel que décrit ne pourrait pas bénéficier du fait que des sous-tableaux pourraient être triés en ordre inverse, alors que ces sous-tableaux pourraient être rendus triés par un simple retournement en $\Theta(n)$. Pour prendre ça en compte, lorsque $A[start]$ est plus grand que

[1] Dans l'énoncé, on suit la convention du cours théorique en faisant commencer l'indexation des tableaux à 1. Dans le code c, la même fonction devra faire commencer l'indexation à 0.

$A[start+1]$, on cherchera la valeur de i maximale telle que $A[start\,..\,i]$ soit trié en ordre inverse. Le sous-tableau $A[start\,..\,i]$ sera alors retourné, en place, et le sous-tableau sera étendu comme dans le cas précédent avec un tri par insertion. Dans l'exemple précédent, un appel à FINDRUN$(A, 5, 2)$ renverra 8 et modifiera le tableau en inversant le sous-tableau $A[5\,..\,8]$:

$$A = [3, 4, 5, 7, 1, 2, 3, 8, 9]$$

Pour partitionner le tableau entier, on pourra appliquer la fonction FINDRUN en partant de la position 1 et en utilisant la position $end + 1$, où end est la valeur renvoyée par l'appel précédent, pour obtenir chacun des sous-tableaux. Le sous-tableau de l'exemple ci-dessous avec $minSize = 3$ serait découpé de la manière suivante en trois sous-tableaux:

$$A = [3, 4, 5, 7, 8, 1, 2, 3, 9]$$

Fusions. Une manière inefficace de fusionner les sous-tableaux obtenus par les appels successifs de la fonction FINDRUN seraient de fusionner chaque nouveau sous-tableau commençant à la position $start$ avec le sous-tableau $A[1\,..\,start-1]$. Une meilleure manière d'ordonner les fusions consiste à procéder comme suit:

(a) Chaque nouveau sous-tableau récupéré par la fonction FINDRUN (ses positions de début et de fin) est stocké sur une pile.

(b) Soit A, B, et C, les trois sous-tableaux au sommet de la pile (C étant le sous-tableau au sommet de la pile) et $|A|$, $|B|$, et $|C|$ leurs longueurs respectives. A chaque nouveau sous-tableau stocké sur la pile, on répète les deux étapes ci-dessous jusqu'à ce que la pile ne soit plus modifiée:

- Si $|A| \leq |B| + |C|$, on fusionne B avec le plus petit des deux sous-tableaux A et C. La taille de B sera mise à jour sur la pile et le sous-tableau avec lequel il sera fusionné sera retiré de la pile.
- Sinon, si $|B| \leq |C|$, on fusionne B avec C, en mettant à jour la taille de B et en retirant C de la pile.

Seule la deuxième vérification est faite si la pile ne contient pas trois sous-tableaux et on ne fait rien si la pile ne contient qu'un seul sous-tableau.

(c) Une fois que l'entièreté du tableau a été lu, on fusionne les deux sous-tableaux au sommet de la pile en les remplaçant par le sous-tableau résultant de leur fusion, en s'arrêtant lorsque la pile ne contient plus qu'un sous-tableau.

L'idée de l'étape (c) est de rétablir l'invariant suivant sur la pile à chaque nouvel appel de FINDRUN:

$$|A| > |B| + |C| \quad (1)$$
$$|B| > |C| \quad (2)$$

où A, B, et C sont les sous-tableaux sur le sommet de cette pile, ce qui a pour effet d'équilibrer les fusions.

Pour donner un exemple, si les sous-tableaux obtenus pour les appels successifs à FINDRUN sont notés de gauche à droite, avec leur taille en exposant:

$$S_1^5, S_2^4, S_3^3, S_4^3, S_5^4, S_6^2,$$

les fusions seront réalisées comme indiqué par le parenthésage suivant, avec les opérations de fusion notées $+$ et numérotées en exposant dans l'ordre de leur exécution:

$$((S_1^5 +^2 (S_2^4 +^1 S_3^3)) +^5 ((S_4^3 +^3 S_5^4) +^4 S_6^2)).$$

Analyse théorique

Dans le rapport, on vous demande de répondre aux questions suivantes

1.1. Donnez le pseudo-code de la fonction FINDRUN($A, start, minSize$).

1.2. Est-ce que cet algorithme de tri peut être implémenté de manière stable ? Est-il en place ? Justifiez votre réponse.

1.3. Etudiez la complexité en temps et en espace de cet algorithme dans le pire et dans le meilleur cas en fonction de la taille du tableau N.
Suggestion: Pour simplifier l'analyse, vous pouvez supposer que dans le pire cas au niveau temps, aucune partie de tableau n'est triée et donc que tous les appels à FINDRUN *renvoie un sous-tableau de taille exactement minSize. Vous pouvez supposer également que la taille du tableau N peut s'écrire* $2^k minSize$*, où k est un entier* ≥ 0.

Analyse empirique

Dans cette partie du projet, vous allez comparer les performances du tri par fusion adaptatif avec celles des algorithmes vus au cours, c'est-à-dire le tri par insertion, le tri rapide, le tri par fusion et le tri par tas. Après avoir implémenté ces algorithmes (voir les contraintes d'implémentation ci-dessous), répondez aux questions suivantes dans votre rapport:

2.1. Calculez empiriquement le temps d'exécution moyen et le nombre de comparaison de ces cinq algorithmes sur des tableaux de tailles croissantes (de 10.000, 100.000 et 1.000.000 éléments) lorsque ces tableaux sont aléatoires, ordonnés de manière croissante, décroissantes, ou presque triés. Pour ces expériences, vous pouvez fixer $minSize$ à 32. Reportez ces résultats dans une table au format donné ci-dessous[2].

Type de tableau	aléatoire			croissant			décroissant			Presque trié		
Taille	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
INSERTIONSORT												
QUICKSORT												
MERGESORT												
HEAPSORT												
ADAPTATIVEMERGESORT												

2.2. Commentez ces résultats. Pour chaque type de tableau:

- comparer l'évolution des temps de calcul en fonction de la taille du tableau aux complexités théoriques
- commentez l'ordre relatif des différents algorithmes en reliant vos observations aux complexités théoriques.

2.3. Discutez de l'intérêt du tri par fusion adaptatif par rapport aux autres algorithmes.

Remarques:

- *Les fonctions pour générer les tableaux vous sont fournies dans le fichier* `Array.c`*. Un tableau presque trié est obtenu en faisant* $0.01 * N$ *permutations de valeurs prises au hasard dans un tableau croissant de taille* N.
- *Les valeurs reportées dans la table doivent être des valeurs moyennes établies sur base d'au moins 10 expériences.*

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

Implémentation et soumission

On vous fournit une implémentation du tri par insertion et du tri par fusion dans les fichiers `InsertionSort.c` et `MergeSort.c`. Vous devez implémenter les trois autres algorithmes dans les fichiers `QuikSort.c`, `HeapSort.c` et `AdaptiveMergeSort.c`. Dans le cas du tri par fusion adaptatif, vous devez rendre accessible (c'est-à-dire non statique) la fonction FINDRUN de manière à ce qu'on puisse la tester. Si l'algorithme peut être implémenter de manière stable, on vous demande de le faire. De manière à pouvoir comptabiliser le nombre de comparaisons faites par chaque algorithme, toutes les comparaisons entre éléments du tableau à trier (et seulement ces comparaisons) doivent passer par la fonction `intCmp` définie dans le fichier `Array.c`.

Deadline et soumission

Le projet est à réaliser ***en binôme*** pour le ***24 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) Un fichier `QuickSort.c`.

(c) Un fichier `HeapSort.c`.

(d) Un fichier `AdaptiveMergeSort.c`.

Respectez bien les extensions de fichiers ainsi que les noms des fichier `*.c` (en ce compris la casse). Les fichiers suivants vous sont fournis:

- `Array.h` et `Array.c`: une petite bibliothèque pour générer différents types des tableaux.
- `Sort.h`: le header contenant le prototype de la fonction de tri.
- `InsertionSort.c`, `MergeSort.c`: des implémentations des algorithmes de tri par insertion et par fusion.
- `main.c`: un fichier de test générant un exécutable prenant comme argument une taille de tableau et un nombre de répétitions et renvoyant le temps moyen et le nombre moyen de comparaisons obtenus sur les quatre types de tableaux.
- `Makefile`: un Makefile qui compile un exécutable par algorithme de tri.

Vos fichiers seront compilés avec la commande suivante[3]:

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

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).

[3]En substituant adéquatement `InsertionSort.c` par l'implémentation de l'algorithme que nous voulons tester.

Un projet non rendu à temps recevra une cote globale nulle. En cas de plagiat[4] 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 !

[4]Des tests anti-plagiat seront réalisés