TD Maple (ou Caml) 4 : Tris

Michael Monerau

3 janvier 2011

1 Maple ou Caml

Cette séance est une séance d'algorithmie. Elle sert à la fois aux SI (épreuve d'algorithmie de l'X) et aux Info (à savoir).

En conséquence, le langage utilisé pour implémenter les algorithmes n'est pas important. Les option Info implémenteront donc en Caml, tandis que les SI feront du Maple.

Testez toujours vos algorithmes sur plusieurs exemples pour détecter les bugs évidents !

2 Généralités sur les tris

2.1 Objectif

Un algorithme de tri prend en entrée une série d'éléments ainsi qu'un pré-ordre complet sur ces éléments (ie. toute paire d'éléments est ordonnée). Il renvoie une série contenant les mêmes éléments, triés selon ce pré-ordre.

Pour rappel, un pré-ordre est une relation réflexive et transitive - un ordre est en plus anti-symétrique.

En **Maple**, on utilisera une liste pour représenter ces séries d'éléments. Dans ce TD, on triera des entiers avec un ordre passé en paramètre. L'ordre sera vu comme une fonction prenant deux paramètres (x, y) et retournant 1 si $x \sqsubseteq y$, 0 sinon.

En **Caml**, on choisira le type `'a vect` ou bien `'a list` selon l'implémentation la plus pratique pour le tri en présence. Chaque fonction de tri prendra en paramètre un prédicat de tri : une fonction `compare : 'a -> 'a -> bool` telle que `compare x y = true` ssi $x \sqsubseteq y$.

Notation : Dans la suite, on note $\preceq$ l'ordre passé à l'algorithme de tri en argument, $\langle x_1, \ldots, x_n \rangle$ la série d'éléments à trier, et $\langle z_1, \ldots, z_n \rangle$ la série d'éléments triée. En particulier, je note ici les indices de 1 à n.

Question 2.1. Décroissant

Supposons qu'on dispose d'une fonction de tri générique. Comment l'appeler pour qu'elle trie une liste d'entiers selon l'ordre usuel ?

Comment trier selon l'ordre décroissant ?

2.2 Propriétés

Il existe plusieurs propriétés que les algorithmes de tris se partagent entre eux.

On ne peut pas dire qu'il existe *un seul* algorithme de tri qui soit efficace dans toutes les situations. En pratique, c'est selon les besoins qu'on utilisera tel ou tel tri. Voir la discussion dans la dernière section pour plus de détails.

Voici deux propriétés importantes qu'un algorithme de tri peut ou non posséder :

Definition 2.1. *On dit qu'un algorithme de tri est* stable *si deux éléments égaux (pour $\sqsubseteq$) ne sont pas inversés dans le tableau par l'exécution du tri. Formellement :*

$$\exists i < j,\ x_i \equiv x_j \quad \Rightarrow \quad \exists i' < j',\ \begin{cases} z_{i'} = x_i \\ z_{j'} = x_j \end{cases}$$

où $x \equiv y$ est $x \preceq y \wedge y \preceq x$.

Question 2.2. Utilité

Donner une situation où la stabilité du tri à utiliser est essentielle.

Definition 2.2. *Un tri est dit* en place *s'il ne requiert que $O(1)$ mémoire pour s'exécuter (sans compter le tableau d'entrée évidemment).*

C'est une propriété essentielle pour trier de larges bases de données : si on trie un tableau de 10 millions d'entrées, on ne veut certainement pas le recopier, même partiellement, au cours du tri. Il faut que le tri soit *en place*.

La stabilité ou le caractère en place d'un tri peuvent dépendre de l'implémentation plus ou moins astucieuse qui en est faite.

2.3 Complexité

Sans hypothèse sur les éléments à trier, la complexité d'un tri (ie. le nombre de comparaisons en pire cas) est nécessairement au moins en $O(n \ln n)$. La preuve de ce fait utiliser un *arbre de décision*. L'exécution d'un algorithme de tri est représenté dans un arbre, où une exécution est un chemin de la racine à une feuille et où chaque noeud est une comparaison. On prouve que cet arbre est de hauteur au moins $n \ln n$, montrant ainsi que tout algorithme de tri, même prenant à chaque fois la bonne décision, ne peut s'exécuter en moins de $O(n \ln n)$ en pire cas, asymptotiquement.

3 Tris basiques

3.1 Tri par sélection

3.1.1 Principe

On parcourt le tableau de gauche à droite. Á l'étape $i = 1 \dots n$, on sélectionne l'élément le plus petit de $\langle x_i, \dots, x_n \rangle$, qu'on échange avec x_i.

Question 3.1. Correction
Montrer que le tableau est bien trié à la fin.

Question 3.2. Implémentation
Implémenter le tri.
Ce tri est-il en place ? Stable ?

3.1.2 Complexité

Question 3.3. Complexité
Quelle est la complexité de ce tri ?

3.2 Tri bulles

3.2.1 Principe

On parcourt le tableau de droite à gauche en échangeant deux à deux les éléments (x_i, x_{i+1}) lorsque $x_i \succ x_{i+1}$. On répète ceci tant qu'on ne fait pas un passage complet sans échanger deux éléments.

Question 3.4. Correction
Montrer que le tableau est bien trié à la fin.

Question 3.5. Implémentation
Implémenter le tri.
Ce tri est-il en place ? Stable ?

3.2.2 Complexité

Question 3.6. Complexité
Quelle est la complexité de ce tri ?

3.3 Tri par insertion

À l'étape $i = 2 \dots n$, on place l'élément i à sa place dans le tableau à sa gauche, ie. $\langle x_1, \dots, x_i \rangle$.

Question 3.7. Correction
Montrer que le tableau est bien trié à la fin.

Question 3.8. Implémentation
Implémenter le tri.
Ce tri est-il en place ? Stable ?

3.3.1 Complexité

Question 3.9. Complexité
Quelle est la complexité de ce tri ?

3.3.2 Shellsort

Shellsort est une variation du tri par insertion. L'idée est que les cas où l'algorithme de sélection est très peu efficace est quand les plus petits éléments sont loin dans le tableau. En effet dans ce cas, on trie le début, puis on refait beaucoup d'insertions par la suite.

Pour contrer ce phénomène, shellsort trie des sous-tableaux de plus en plus grands. On commence par se fixer une suite $\langle h_1, \dots, h_k \rangle$ finie et décroissante d'entiers telle que $h_k = 1$ et $h_1 < n$. Ensuite, on trie les sous-tableaux $\langle x_r, x_{h_i+r}, x_{2h_i+r}, \dots, x_{\lfloor n/h_i \rfloor + r} \rangle$ pour $r = 0 \dots h_i - 1$, puis pour $i = 1 \dots k$.

Le fait que $h_k = 1$ assure que shellsort est bien un algorithme correct pour trier. Les étapes pour $i < k$ sont là pour rendre cette dernière passe beaucoup plus rapide : les échanges auront lieu entre des éléments censés être plus proches que normal.

L'analyse mathématique de la complexité est extrêmement compliquée, et pas connue dans le cas général. Certaines suites (h_i) sont catastrophiques (comportement quadratique), mais certaines permettent d'avoir des complexités de $O(n^{4/3})$ par exemple.

En pratique shellsort a l'intérêt d'être facilement implémentable et d'avoir des performances correctes. Il n'est pas stable, mais il hérite du caractère en place du tri par insertion.

3.4 Discussion sur les trois algorithmes

Question 3.10. The winner is
Lequel de ces trois algorithmes vous semble préférable ? Quand ? Pourquoi ?

4 Tris avancés

4.1 Tri fusion

4.1.1 Principe

Le tri fusion est un algorithme de type *diviser pour régner* : on sépare le problème principal en deux sous-problèmes indépendants et strictement plus petits.

Plus précisément, l'algorithme s'organise comme suit :

- **Cas triviaux** : si il n'y a que trois éléments ou moins dans la série, le trier à la main. Sinon :
- **Étape de division** : diviser la série d'éléments en deux séries (s_1, s_2) de même taille (à un élément près)
- **Récursion** : trier récursivement par fusion s_1 et s_2
- **Étape de combinaison** : fusionner s_1 et s_2 (ie. construire s triée telle que $s = s_1 \cup s_2$)

Question 4.1. Implémentation
Implémenter le tri.
Ce tri est-il en place ? Stable ?

4.1.2 Complexité

Question 4.2. Complexité
Quelle est la complexité de ce tri ?

4.2 Tri rapide

4.2.1 Principe

Le tri rapide est également un algorithme de type *diviser pour régner*.

Pour simplifier la discussion, supposons que tous les éléments sont distincts ($x_i \not\equiv x_j$ pour $i \neq j$).

Pour trier un tableau non trivial, on choisit un élément dedans, le *pivot*. Puis on arrange les éléments tels que les éléments inférieurs ou égaux au pivot soient au début, puis le pivot, puis les éléments plus grands que lui.

Ensuite, on trie récursivement la partie du tableau $\prec$ pivot, puis la partie $\succ$ pivot.

Dans ce cas, l'étape de division est délicate alors que celle de combinaison est triviale (on remet simplement les résultats bout à bout).

Question 4.3. Implémentation
Implémenter le tri. On choisira le dernier élément du tableau comme pivot.
Ce tri est-il en place ? Stable ?

4.2.2 Complexité

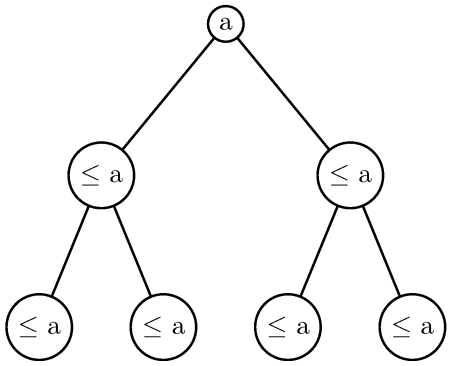
L'analyse de la complexité de l'algorithme quicksort est complexe. Citons cependant les résultats les plus importants. En pire cas, la complexité est quadratique (le pivot se trouve être toujours le plus grand élément, par exemple dans un tableau déjà trié). En moyenne (la notion n'est pas triviale à définir, mais restons-en à l'intuition), la complexité est en $O(n \ln n)$. En meilleur cas, c'est aussi du $O(n \ln n)$.

En pratique, c'est l'algorithme utilisé si le pire cas quadratique ne présente pas un problème majeur. On le combine souvent avec un tri par insertion à partir des appels récursifs sur des tableaux de moins de 10 éléments. On choisit aussi un pivot plus malin : par exemple le médian de 3 éléments du tableau.

4.3 Tri par tas

4.3.1 Principe

Definition 4.1. *Un* tas *(ou* heap*) est un arbre binaire tel que tout noeud a une étiquette supérieure ou égale à celle de toutes celles de ses sous-arbres droit et gauche.*



Le tri par tas procède en deux étapes (la 3 n'est qu'une répétition) :

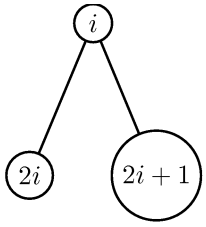
1. Placer les éléments dans un tas

2. Enlever le plus grand élément du tas (la racine), tout en conservant la structure de tas avec les éléments restant
3. Répéter l'étape 2 jusqu'à qu'il n'y ait plus qu'un élément

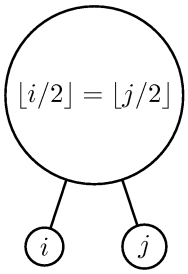
Il suffit donc de définir comment on crée un tas à partir d'un tableau, puis comment on supprime un élément en restant un tas.

4.3.2 La structure de tas

On représente un arbre binaire dans un tableau linéaire grâce à la correspondance suivante sur les indices :



Réciproquement,



4.3.3 La dynamique dans le tas

Modification bottom-up

Pour insérer un élément dans le tas, on procède de bas en haut. On le place tout à la fin du tableau (qui est déjà sous forme de tas). Puis tant qu'il est strictement supérieur à son père, on l'échange avec celui-ci.

Modification top-down

Pour supprimer la racine du tas, on prend le dernier élément du tas (un élément d'une feuille donc) qu'on place en racine. Alors, tant qu'il est strictement plus petit qu'au moins un de ses deux fils, on l'échange avec le plus grand des deux.

4.3.4 Implémentation

Question 4.4. Fonctions helpers
Implémenter ces petites fonctions qui aident à implémenter l'algorithme : la fonction qui calcule les indices des fils droit/gauche, du parent, qui insère un élément et qui supprime la racine (en retournant sa valeur).

Question 4.5. Implémentation
Implémenter alors le tri par tas.
Ce tri est-il en place ? Stable ?

4.3.5 Complexité

Question 4.6. Complexité
Quelle est la complexité de ce tri ?

5 Le café du commerce

Résumons à présent tout cela.

Les trois premiers algorithmes de tri (sélection, bulle, insertion) sont les plus basiques : ils sont quadratiques, mais à part le tri par sélection, ils sont assez efficaces sur des tableaux presque déjà triés.

Les trois algorithmes qui suivent (fusion, quicksort, tri par tas) sont plus délicats mais fournissent des performances largement meilleures.

Dans les applications réelles, on utilise essentiellement une version améliorée de quicksort (pour traiter efficacement les éléments égaux), avec un cutoff à ≈ 10 éléments comme précisé plus haut, pour finir avec un rapide tri par insertion (pour un petit tableau, le coup de la récursion fait dépasser le coût d'un rapide tri par insertion).

Si le scénario d'un pire cas quadratique est inacceptable pour l'application (cela peut même être une vulnérabilité donnant lieu à une attaque), on utilise le plus souvent un tri fusion. En effet, il présente souvent de meilleures performances qu'un tri par tas (grâce aux accès séquentiels qui se comportent bien avec les mécanismes de caches, et à la parallélisation aisée), il est stable et garantit du $O(n \ln n)$. Cependant, dans les cas où il faut un tri en place, le tri par tas reste le meilleur choix.