TP 01 – Algorithmes de tri par comparaison

Langage de programmation au choix (à valider avec votre chargé de TP).

Dans cette planche de TP, nous nous intéressons aux algorithmes de tri par comparaison et à leur complexité. Le problème consiste à trier par ordre croissant un tableau T de n entiers, dont les valeurs sont bornées (par exemple entre 0 et 999). La taille de l'entrée sera donc n.

Tout graphique doit être commenté.

Exercice 1. *Tri rapide*

Cet algorithme est le plus efficace en pratique, et celui qui est en général implémenté par les méthodes `sort()`. Il consiste à choisir une valeur du tableau, appelée *pivot*, puis à placer :

— au début du tableau tous les éléments inférieurs au pivot, et
— à la fin du tableau tous les éléments supérieurs au pivot, et
— le pivot entre ces deux parties.

Le pivot est choisi comme étant la première valeur du tableau. Pour ordonner tous les éléments autour du pivot, on réalise un parcours en partant du dernier élément : tant qu'il est plus grand que le pivot, on avance en direction du pivot, puis quand une valeur plus petite que le pivot est rencontrée, on l'échange avec le pivot, et on poursuit le parcours dans l'autre sens (donc toujours en direction du pivot). Quand le parcours rencontre le pivot, celui a trouvé sa position. L'algorithme se poursuit en appliquant récursivement la même procédure aux deux parties.

1. Implémenter l'algorithme du tri rapide.
2. Analyser sa complexité dans le pire cas, qui est celui où le tableau est déjà ordonné. Tester votre algorithme pour vérifier cette borne. Produire un graphique.
3. La complexité en moyenne du tri rapide est $\mathcal{O}(n \log n)$. Tester votre algorithme sur des tableaux remplis aléatoirement pour vérifier cette borne. Produire un graphique.
4. Quelle est la complexité dans le pire cas du tri rapide si le pivot est à chaque itération la valeur médiane de la partie considérée ?
5. On peut trouver la valeur médiane d'un tableau en temps linéaire. Implémenter ce choix de pivot, et comparer l'efficacité de cette variante sur des tableaux remplis aléatoirement.

Exercice 2. *Borne inférieure aux tris par comparaisons*

Dans cet exercice, nous allons déterminer une borne inférieure en $n \log n$ au nombre de comparaisons dans le pire cas de tout algorithme de tri basé sur des comparaisons entre les éléments du tableau à trier. Un algorithme de tri prend en entrée un tableau, et permute ses éléments de telle sorte que le tableau soit trié par ordre croissant. Un algorithme de tri par comparaisons base ses décisions sur les comparaisons d'éléments. À chaque comparaison réalisée entre deux valeurs du tableau, l'algorithme peut avoir deux comportements différents, suivant le résultat de la comparaison.

Pour une taille n donnée, un algorithme de tri par comparaisons peut être vu comme un arbre binaire dont la racine est le départ, et dont les noeuds internes représentent les comparaisons réalisées et leur deux différentes issues possibles. Chaque chemin de la racine à une feuille représente une exécution possible de l'algorithme.

1. Pour un tableau de taille n, combien de permutations différentes l'algorithme doit-il potentiellement être capable de réaliser ?
2. La quantité de quels éléments de l'arbre est bornée inférieurement par cette valeur ?
3. La hauteur de son arbre correspond à quel élément de la complexité d'un algorithme ?
4. Quelle relation existe-t-il entre la hauteur d'un arbre et son nombre de feuilles ?
5. En déduire une inégalité, reliant la taille n et la complexité dans le pire cas de tout algorithme correct de tri par comparaisons.
6. Pour conclure, borner inférieurement $n!$ par le produit de ses $\frac{n}{2}$ premiers termes qui sont tous au moins $\frac{n}{2}$, ou utiliser la formule de Stirling approximant $n!$ par $\sqrt{2\pi n}(\frac{n}{e})^n$.

Exercice 3. *Tri fusion*

Le tri fusion est un algorithme récursif qui fonctionne en trois temps :

(a) trier par un appel récursif la première moitié du tableau,

(b) trier par un appel récursif la seconde moitié du tableau,

(c) fusionner les deux parties triées.

Si le tableau est de taille inférieure ou égale à deux, alors on le trie directement.

1. Implémenter l'algorithme du tri fusion.
2. Analyser sa complexité dans le pire cas, à l'aide du Master théorème.
3. Tester votre algorithme sur des tableaux remplis aléatoirement. Produire un graphique.

Exercice 4. *Tri deux-tiers*

Le tri deux-tiers est un algorithme récursif qui fonctionne en trois temps :

(a) trier les premiers $2/3$ du tableau,

(b) trier les derniers $2/3$ du tableau,

(c) trier les premiers $2/3$ du tableau.

Si le tableau est de taille inférieure ou égale à deux, alors on le trie directement.

1. Implémenter l'algorithme du tri deux-tiers.
2. Expliquer pourquoi cet algorithme trie correctement tout tableau.
3. Analyser sa complexité dans le pire cas, à l'aide du Master théorème.
4. Tester votre algorithme sur des tableaux remplis aléatoirement. Produire un graphique.

Exercice 5. *Suppression de doublons*

Pour supprimer les doublons dans un tableau de n entiers dont l'ordre des éléments importe peu, nous allons comparer deux méthodes. Le résultat est un nouveau tableau de taille $\leq n$.

1. Un algorithme simple consiste à ajouter le premier élément au tableau résultat, puis à ajouter chacun des éléments suivants seulement s'il n'est pas déjà présent. Implémenter cette méthode. Quelle est sa complexité ?
2. Une seconde méthode consiste à trier le tableau, puis supprimer les doublons en un parcours. Implémenter cette méthode. Quelle est sa complexité ?
3. Comparer expérimentalement ces deux méthodes. Produire un graphique.