TP 4 : Tris

lionel.rieg@ens-lyon.fr

Dans ce TP, nous allons nous intéresser aux divers algorithmes de tri. Pour simplifier, on ne va travailler qu'avec des entiers qu'on veut trier par ordre croissant. La structure de données dans laquelle seront stockés les entiers à trier sera un tableau (mais on pourrait aussi bien utiliser des listes, le type `vector<int>` de C++). Dans la mesure du possible, on s'efforcera de faire des tris *en place*, c.-à-d. des tris qui n'utilisent pas de mémoire en plus de celle du tableau à trier.

1 Tri bulle

Ce tri est de loin le plus simple des tris. Sa simplicité se paie néanmoins dans sa complexité en temps. Son principe est le suivant : tant que deux éléments consécutifs ne sont pas rangés dans l'ordre croissant, on les échange.

1. Comment s'assurer que l'on considère toutes les paires d'éléments consécutifs ?
2. Implémenter cet algorithme.
3. Après une passe complète de l'algorithme, que peut-on dire sur l'élément maximum ?
4. En déduire une borne de complexité en pire cas.

2 Test des tris

Pour vérifier que les algorithmes fonctionnent, on va avoir besoin d'un algorithme de test.
Pour cela, on utilise une fonction `main` qui :

1. crée un tableau dont les valeurs sont aléatoires,
2. trie ce tableau avec l'algorithme de tri qu'on veut tester
3. vérifie que le tableau est trié (attention aux éléments en plusieurs exemplaires)

Écrire une telle fonction.

3 Tri par insertion

Ce tri est aussi couramment appelé « tri du joueur de carte » car c'est celui qu'on utilise pour trier une main de cartes. Son principe est le suivant : on construit le tableau trié en ajoutant incrémentalement des éléments à un sous-tableau trié. En pratique, le tableau à trier est séparé en deux parties, la première est trié et la seconde contient les éléments à ajouter. On augmente la taille de la première partie en ajoutant le premier élément de la seconde à la bonne place.

1. Implémenter cet algorithme.
2. Est-ce qu'une recherche dichotomique de la position à laquelle insérer le nouvel élément est utile ?
3. Trouver un exemple pour lequel sa complexité est quadratique.

4 Tri fusion

Ce tri est très utilisé pour les listes car sa nature récursive correspond bien à celle des listes. Son principe est le suivant : on découpe le tableau en deux parties de tailles égales (à 1 près), on trie récursivement chaque partie, on les fusionne en un seul tableau.

1. De combien de mémoire supplémentaire a-t-on besoin ?

2. Implémenter cet algorithme.
3. Quelle est la complexité en pire cas de cet algorithme ?

5 Tri rapide

Ce tri est assez semblable au tri fusion dans sa nature récursive qui découpe le tableau en deux. Il est tout aussi rapide en moyenne mais il ne nécessite pas de mémoire supplémentaire. Son principe est le suivant : on prend un élément du tableau, le *pivot*, et on sépare les éléments du tableau en deux catégories suivant qu'ils sont plus grands ou plus petits que le pivot (on range les éléments égaux au pivot où l'on veut). On obtient donc deux sous-tableaux qu'on peut trier récursivement. Il ne reste plus alors qu'à mettre le pivot au milieu, entre les deux sous-tableaux triés pour obtenir un tableau trié. En général, on prend pour pivot le premier élément du tableau.

1. Implémenter cet algorithme.
2. En quoi le choix du pivot est-il crucial ? Trouver un tableau pour lequel la complexité du tri avec premier élément comme pivot est quadratique.
3. Quel meilleur choix de pivot peut-on faire ?