Exemples d'algorithmes de tri

I Tris par substitution

Dans tout ce chapitre, on presentera les tri sur des tableaux

Théorème : Pour tout algorithme de tri déterministe de tri A, le nombre de comparaison de A sur un tableau de taille n domine $n \log n$ pour le pire des cas.

I.1) tri par insertion

Idée: Pour chaque element dans l'ordre des indices, on l'échange avec son prédecesseur tant que le prédecesseur est plus grand que l'élement.

complexité:

pire des cas: $O(n^2)$

Remarque : tri en place ayant une implémentation simple

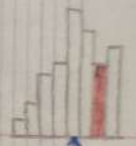
I.2) tri bulle

Idée: On echange chaque element et son predecesseur si l'element est inférieur à son prédecesseur, On répète jusqu'à ce qu'il n'y ait pas d'interversion.

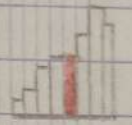
complexité: pire des cas: $O(n^2)$

Remarque: tri en place ayant une implémentation simple

I.3) tri par selection



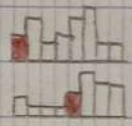
Idée: pour chaque indice i pris par ordre croissant, on cherche l'indice k du plus petit element du tableau à partir de i. ensuite on echange les elements aux i et k



complexité: toujours $O(n^2)$

Remarque : Tri en place

I.4) tri rapide (en place) / tri par pivot



Idée: On choisit un element appelé le pivot: On echange le premier element et le pivot. Ensuite on échange le 1er element plus grand que le pivot et le dernier plus petit que le pivot tant que l'indice du premier est inférieur à celui du second. Ensuite, on echange le pivot et le dernier element inférieur à celui-ci.

Enfin on tri par pivot le sous tableau des elements precedent le pivot et le sous-tableau des éléments suivant le pivot.

Complexité: pire des cas $O(n^2)$

Si le pivot est choisit aleatoirement dans le tableau, la complexité moyenne est $O(n \log n)$

Remarque: il existe une implementation classique par répartition mais l'implémentation en place est très utilisé en pratique.
Ce tri est souvent considéré comme le plus rapide en général.

I. 5) Shellsort (Dev 1)

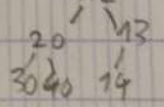
I. 6) Tri de Radix
Ce tri sert à trier un tableau dont les valeurs sont comprises entre k et $k+p$
Idée: on fait comme le tri par pivot avec $k+\frac{p}{2}$ à la place du pivot. Ensuite on fait le tri de radix des elements inferieur à $k+\frac{p}{2}$ avec pour borne k et $k+\frac{p}{2}-1$ et le tri de Radix des elements supérieur ou égaux à $k+\frac{p}{2}$ avec pour borne $k+\frac{p}{2}$ et $k+p$.
Complexité: $O(n \log(p))$

II Autres tris
II. 1) Tri fusion

Idée: On trie un tableau de longueur ≥ 2 en le coupant en 2. Ensuite, on tri par fusion les 2 sous-listes. Enfin, on fusionne les 2 listes trié en insérant le plus petit élément pas encore insérée.

Complexité: $O(n \log n)$

II. 2) tri par tas
Def: un tas est un arbre binaire complet par couche tel que tout noeud interne a une étiquette inférieure à ses fils.
Prop: On peut ajouter un element ou supprimer le plus petit element d'un tas contenant n element en temps $O(n \log(n))$
Idée: On ajoute chaque element du tableau dans un tas. Ensuite on enlève le minimum du tas tant qu'il n'est pas vide et on l'insère à la fin d'un second tableau initialement vide.
complexité: pire des cas $O(n \log n)$



III Structures triés:
Operations pour le maintien d'une structure triée:
recherche, insertion, suppression

III.1) Tableau trié

Recherche: Recherche dichotomique $O(\log n)$
Insertion: ajout au bout du tableau puis tri par insertion $O(n)$
Suppression: on substitue l'element à supprimer avec son successeur jusqu'à ce qu'il soit au bout du tableau $O(n)$

III.2) (ABR) Arbre binaire de recherche

Def: Un arbre binaire est dit de recherche si l'étiquette de chaque noeud est supérieure ou egale à tout etiquette de son sous-arbre gauche et inferieure ou égale à tout étiquette de son sous-arbre gauche.

Recherche(x, t): - si t est vide, on ne trouve pas
- si la racine de t est x: on a trouvé.
- Si elle est inferieure, on renvoit recherche(x, fg) avec fd le sous arbre droit de la racine. Sinon, on renvoit recherche(x, fg) où fg est le fils gauche de la racine
Insertion(x, t): on fait comme la recherche jusqu'à trouver un arbre vide que l'on remplace par la feuille x.

Suppression(x, t): on cherche x. Lorsque l'on le trouve, si x n'est pas une feuille, on remplace son etiquette par celle du maximum de son sous-arbre gauche ou le minimum de son sous-arbre gauche que l'on supprime.
recherche, insertion et suppression ont une complexité dominé par la hauteur de l'arbre.
défaut: a priori, la hauteur d'un ABR est de l'ordre de n au pire.

III.2.1) AVL

Def: Un AVL est un ABR tel que la hauteur du sous-arbre gauche et du sous-arbre droit d'un noeud diffère d'au plus 1.
Pour maintenir la propriété on ajoute à chaque noeud une balance = hauteur(sous arbre gauche) - hauteur(sous arbre droit).
Lors des insertions et des suppressions, on met à jour les balances et si une balance de passe -2 ou 2, on fait 1 à 2 rotations pour rééquilibrer l'arbre en dessous.

III.2.2) Arbres rouge noir

Def: Un ABR dont les noeuds sont colorés en rouge ou noir est un arbre rouge-noir si:
- la racine et les feuilles sont noires
- le nombre de noeud noir le long d'une branche ne dépend pas de la branche
- pas de deux noeud rouge et noir.

On peut adapter l'insertion et la suppression pour maintenir les arbres rouge-noir au travers de rotations et de recoloration le long de la branche modifiée.

Théorème: Les arbres rouge-noir et les AVL ont une hauteur dominée par $\log(n)$ et les opérations d'insertions et de suppressions sont toujours dominées par la hauteur.

III.2.3) Splay tree (Dev 2)