Projet SDA:

Rapport des algorithmes de tri de tableaux

1. InsertSort – tri par insertion

1.1. Théorie

Le **tri par insertion**, ou **InsertSort** pour les anglophones, est l'un des rares algorithmes de tri fonctionnant non seulement sur des tableaux statiques, mais aussi pour des listes. Le principe de cet algorithme est dit "*de comparaison au voisinage*": si le voisin de gauche de la valeur insérée est inférieur à celle-ci, un décalage est effectué. Tant que le tableau (ou la liste) n'a pas été trié.e, cette comparaison sera effectuée sur la partie restante à trier.

Sa complexité en pire cas est de $\Theta(n^2)$, et celle en moyenne, qui est admise, de $\Theta(n^2)$.

Cet algorithme est **en place** (puisqu'on ne travaille que dans un tableau ou une liste qui est celui/celle qui est donné.e en paramètre) et **stable** (on ne fait que des décalages et une insertion si la cellule I comprise entre 0 et n-1 (supposé un tableau de taille n) d'un tableau est inférieure à la valeur à insérer.

1.2. Pratique & analyse

Efficacité de ce tri: théoriquement ET pratiquement pas très efficace

Place mémoire de ce tri: efficient (O(1)), n'utilise qu'un seul tableau

2. MergeSort – tri fusion

2.1. Théorie

Le **tri fusion**, ou **MergeSort**, est un tri récursif, comme le tri suivant, et à l'inverse de l'algorithme précédent qui est itératif. Le processus de MergeSort découpe d'abord le tableau de manière dichotomique jusqu'à n'avoir que des "cellules-tableau" (des tableaux constitués d'une seule cellule), et les trie les uns aux autres, à chaque fois entre 2 tableaux, jusqu'à avoir un tableau trié unique.

Sa complexité en pire cas (et en moyenne) est de $\Theta(n \log_2 n)$.

Cet algorithme est **stable** (il garde l'ordre relatif des éléments égaux), mais il n'est **pas en place** dû à l'utilisation d'un tableau auxiliaire pour la fusion.

2.2. Pratique & analyse

Efficacité de ce tri: théoriquement efficace, pratiquement pas très efficace

Place mémoire de ce tri: gourmand (O(n)), utilise un tableau auxiliaire.

3. QuickSort – tri rapide

3.1. Théorie

Le **tri rapide,** ou plus communément **QuickSort**, est le tri le plus utilisé (donc le plus connu, pourtant peu s'intéressent à son algorithme) car le plus rapide. Il segmente le tableau à partir d'un pivot (souvent le premier élément du tableau non trié) et détermine où le placer pour le tri du tableau.

Sa complexité, qu'elle soit en pire cas ou en moyenne, est de $\Theta(n \log_2 n)$.

Cet algroithme est **en place** (il n'utilise que le tableau passé en argument), mais il est **instable**, vu qu'il ne prend pas en compte l'ordre relatif des éléments égaux lors de son exécution.

3.2. Pratique & analyse

Efficacité de ce tri: théoriquement et pratiquement efficace

Place mémoire de ce tri: peu gourmand (O(log n)), n'utilise que le tableau passé en argument

4. RadixSort – tri radix/par base

1.1. Théorie

La complexité en pire cas du tri par base est de: $n+m(n+b+2(n-1)) => \Theta(nm)$

Cet algorithme **n'est pas en place**, vu qu'il utilise un tableau auxiliaire, mais **stable**, gardant l'ordre relatif des éléments.

1.2. Pratique & analyse

Efficacité de ce tri: théoriquement pas très efficace, pratiquement efficace

Place mémoire de ce tri: très gourmand (O(n+k)), utilise un tableau auxiliaire en plus des différents indices dû à la base

5. BucketSort – tri par paquets

1.1. Théorie

La complexité en pire cas du tri par paquets est: ***Θ(n²), dû à l'InsertSort qui en est le tri auxiliaire.***

Cet algorithme **n'est, lui non plus, en place**, vu qu'il utilise un tableau auxiliaire, mais, tout comme le RadixSort, il est **stable**. Il garde l'ordre relatif puisqu'il utilise... InsertSort. (Après, ça dépend du tri auxiliaire utilisé.)

1.2. Pratique & analyse

Efficacité de ce tri: pas très efficace, tant théoriquement que pratiquement. *Après, c'est déjà mieux que le BogoSort.*

Place mémoire de ce tri: gourmand (O(n)), utilise un tableau auxiliaire

Bonus

On n'a pas pu prendre le temps de le faire...