Analyse des algorithmes de tris

Fouché Stanislas
Hill Tom
Leveziel Damien
Boulay Dimitri

Mars 2023

Table des matières

1 Objectifs du projet

1.1 Problématique du projet

Un algorithme de tri est un algorithme qui va permettre d'organiser une collection d'objets selon une relation. Il existe une multitude d'algorithmes de tri, au fonctionnement et à l'efficacité différentes. La plupart de ces algorithmes sont basés sur de la comparaison entre des éléments. Cependant, les éléments choisis pour ces comparaisons sont la différence fondamentale entre ces algorithmes. Certains de ces algorithmes sont peu efficaces, et la plupart dépendent de la structure des éléments à comparer. Nous nous demandons comment varie l'efficacité des ces algorithmes selon le type de tri en fonction du désordre des données. Au delà de comparer les temps d'exécution nous souhaitons aussi visualiser le nombre de comparaisons et l'accès aux données. Pour ce faire nous devons implémenter un certain nombre de ces algorithmes et les comparer sur des tableaux de données similaires pour constater leur efficacité.

1.2 Choix

On a choisit d'utiliser java parce qu'il était plus simple d'utiliser les adresses des tableaux en java, étant donné qu'on y était déjà tous habitués. De plus, la visualisation est plus simple à effectuer qu'en python, surtout si on veut faire une application qui trie des tableaux en boucle. Enfin, faire ce projet en java permettait d'approfondir notre connaissance au langage objet, ce qui faisait de java le langage qu'on préférait utiliser

1.3 Description des point clés du projet

Les points clés du projet sont dans un premier temps l'implémentation du générateur de tableau avec plusieurs de types de mélanges. Après avoir généré un tableau, le choix des algorithmes de tri ; il a fallu choisir des algorithmes pertinents à comparer avec des complexités différentes. Il a ensuite fallu implémenter ces algorithmes. Nous avons créé une interface graphique permettant de visualiser le fonctionnement de ces algorithmes en direct après les avoir fait. Et enfin, nous avons créé de quoi tester ces algorithmes et voir leur différences.

2 Fonctionnalités implémentées

2.1 Répartition du travail

Pour commencer, chacun de nous a implémenté de 3 à 4 algorithmes de tri. Ensuite il fallait disperser les taches restantes :

- Génération de tableau aléatoires : Damien et Dimitri
- Interface Graphique : Dimitri et Tom
- Mise en forme des résultats expérimentaux : Damien et Tom
- Tests et la mise en place d'un ant : Stanislas

3 Éléments techniques

3.1 Description du générateur de données

Il à été décidé de choisir comme données des tableaux d'entiers, ce sont des données simples et facilement manipulâmes. Le générateur va créer des tableaux de la taille désirée, avec des entiers allant de 1 à n (n étant la taille désirée). Ensuite selon le type de mélange choisi et l'entropie, le générateur va mélanger ce tableau. Il existe différents types de mélanges :

- Le mélange simple : Le générateur va faire X inversions aléatoires dans le tableau. (Avec X l'entropie)
- Le reverse : Le générateur va inverser le tableau. Généralement ce mélange permet de montrer la pire complexité des algorithmes.
- La fin mélangée : Le générateur va faire X inversions dans les 25 derniers % du tableau, le reste est trié dans l'ordre.

3.2 Description des algorithmes

Pour le bien de ce projet nous avons dû implémenter une dizaine d'algorithmes de tri aux complexités et au fonctionnement différents.

3.2.1 BubbleSort

Complexité : n^2
Cet algorithme est un algorithme basique qui va seulement inverser deux éléments si ils ne sont pas dans le bon ordre. Il est en théorie inefficace sur des grands tableaux de données.

3.2.2 InsertionSort

Complexité : n^2
Le tri par insertion est un tri basique. Il fonctionne de la manière suivante. L'algorithme va créer un tableau secondaire où à chaque passage dans le tableaux de données, il va la placer à son emplacement dans le tableau secondaire.

3.2.3 BogoSort

Complexité : $(n+1)!$
Cette algorithme est l'algorithme de tri le plus inefficace, il consiste seulement à mélanger aléatoirement le tableau à chaque itération et vérifier si les données sont triées. Il est utilisé généralement comme comparaison aux autres algorithmes de tri. Nous l'avions implémenté à la base pour tester notre génération de tableaux.

3.2.4 CombSort

Complexité : n^2
Le CombSort (ou tri à peigne) est une amélioration de l'algorithme BubbleSort 3.2.1. Au lieu de prendre chaque élément adjacent un par un, il prendra les éléments avec un espacement différent à chaque itération. L'espacement initial est égal à la taille de la liste, puis il est divisé par 1,3 à chaque itération. Le choix de 1,3 n'est pas anodin, il à été trouvé empiriquement sur beaucoup de listes pour savoir quelle valeur est optimale.

3.2.5 GnomeSort

Complexité : n^2
Le Gnome Sort est une variation du Tri par insertion (3.2.2) dont la principale différente est le fait qu'il ne passe pas par une deuxième tableau, il consiste en un parcours des données, et à chaque fois l'algorithme va prendre la valeur choisie et la suivante. Si la valeur suivante est inférieure il va échanger ces deux valeurs ensuite retourner à la valeur précédente du tableau. Si la valeur est suivante est supérieure, il va avancer jusqu'à la valeur suivante. Jusqu'à la fin du tableau.

3.2.6 QuickSort

Complexité : $n \log n$
Le Quick Sort est l'algorithme considéré comme le plus efficace pour trier un tableau. Il prend une valeur choisie, soit au hasard, à droite, à gauche ou au centre du tableau, et met cette valeur à sa place, avec toutes les autres valeurs inférieures à sa gauche, et supérieure à droite. Après avoir placé la valeur, on peut aller sur le tableau de gauche et de droite récursivement et faire la même chose pour faire des partitions qui ne sont pas loin d'être terminées, et qui auront une complexité de moins en moins grande.

3.2.7 PancakeSort

Complexité : n^2
Le pancake sort est un algorithme très peu , mais est un algorithme visuel et facile à comprendre. Pour ce faire, on prend la valeur la plus petite entre la partie du tableau non triée et la partie triée, puis on la met à la fin en renversant tout le tableau entre cette valeur et la dernière. Cela fait que la plus petite valeur est à la fin, puis on renverse tout le tableau non trié une nouvelle fois pour avoir la valeur nécessaire juste après la partie triée. Puis on incrémente de un la taille du tableau trié, et on recommence.

3.2.8 HeapSort

Complexité : $n \log n$
Le Heap sort est une amélioration du Tri par sélection, mais en utilisant la

structure d'un arbre binaire. Nous allons chercher l'élément minimal et le mettre au début. Et répéter jusqu'à la fin de l'exécution.

3.2.9 Radix

Complexité : n
Radix est un tri qui n'utilise pas le principe de comparaison. Le principe est de compter le nombre d'occurrences de chaque entier, et avec ceci on peut calculer un index pour chaque nombre dans la liste triée et recréer une liste en parcourant notre liste à trier et mettant chaque nombre à l'index calculé pour lui.

Il faut un astuce supplémentaire pour trier, car on ne peut pas faire un tableau d'indices de 2 milliards d'entrées. On faut le tri sur les 8 derniers bits, puis à partir de cette nouvelle liste on prends les 8 prochains bits, et après 4 itérations notre entier est triée.

3.2.10 ShakerSort

Complexité : n^2
Le Tri shaker est une variante du Tri à Bulles (3.2.1) bidirectionnel, qui va changer de direction à chaque passage le long de la liste à trier. Il est en théorie plus efficace que le tri à bulles car il va permettre de gérer les éléments de faible valeur qui vont rester en fin de liste en remontant lentement.

3.2.11 ShellSort

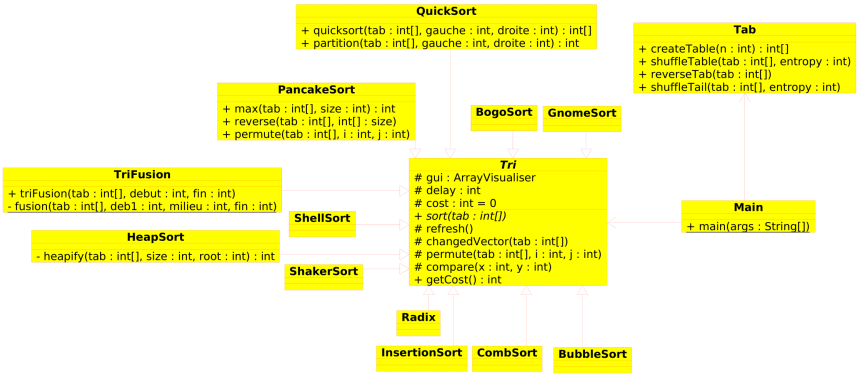
Complexité : $n^{4/3}$
Le tri de Shell est une amélioration du tri par insertion (3.2.2). Le tri de Shell trie les éléments séparés de n positions. L'algorithme diminue la valeur de n à chaque itération jusqu'à que n soit égal à 1. La valeur de diminution de n à été trouvée empiriquement et est 2.3.

3.2.12 MergeSort

Complexité : $n \log n$
Le tri fusion est un algorithme de tri par comparaison. Il va rassembler deux listes triées en une seule. La caractéristique principal de ce tri est qu'il utilise une zone de mémoire deux fois plus grande que le tableau de base car il doit en créer une copie.

4 Architecture du projet

4.1 Diagrammes des classes



La classe Tab créé le tableau, qu'on mélange juste après, que l'on va trier. La classe Main prend alors un Tri, quel qu'il soit, afin de trier ce tableau. Grâce à l'héritage, il est facile d'utiliser n'importe quel algorithme de trie que l'on a.

4.2 Ant

On a utilisé un ant pour gérer notre projet :

— `ant compile` : pour simplement compiler l'application
— `ant dist` : pour compiler l'application, créer le fichier `jar` et placer ce dernier dans le répertoire `/dist` du projet
— `ant run` ou `ant` : pour exécuter l'application
— `ant test` : pour exécuter les tests
— `ant javadoc` : pour générer la javadoc de l'application

Si le tests unitaires ne fonctionnent pas depuis le ant on peut utiliser éclipse pour voir que nos tests fonctionne. (problème de librairie)

Les tests cherchent juste à prouver que les algorithmes fonctionnent.

4.3 Lancement algorithme

Pour effectuer une simulation d'un des algorithmes, il faut aller dans le fichier src/Main.java et choisir les options à lancer :

— taille du tableau
— entropy
— reverse
— choisir l'algorithme (de préférence mettre sous commentaires les algo non utilisé)

il ne reste plus qu'a utilisé la commande : **ant** ou **ant run** (run est le lancement par défault)

```
//init tableau

int tailletab   =     100;
int entropy     =     10000;
boolean reverse =     false;
```

```
/* UTILISATION : choisir un des algo suivants et mettre les autres sous commentaires
on ne modifira que le délai, on touche pas au panel ni au sort.



/******* complexité moyenne nlogn **********/

new HeapSort(panel, 100).sort(tab);

//new QuickSort(panel,100).sort(tab);

//new TriFusion(panel,100).sort(tab);


/******* complexité moyenne n² **********/

//new BubbleSort(panel,20).sort(tab);

//new ShakerSort(panel,10).sort(tab);

//new InsertionSort(panel,10).sort(tab);

//new CombSort(panel,100).sort(tab);

//new GnomeSort(panel,100).sort(tab);

//new ShellSort(panel,100).sort(tab);

//new PancakeSort(panel, 10).sort(tab);

/****** autres ******/

//new BogoSort(panel,10).sort(tab);

//new Radix(panel,100).sort(tab);
```

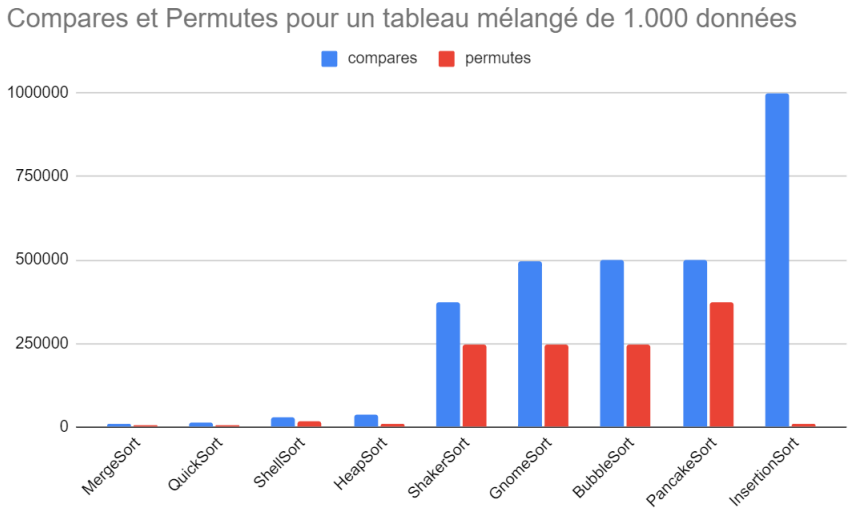
5 Expérimentations

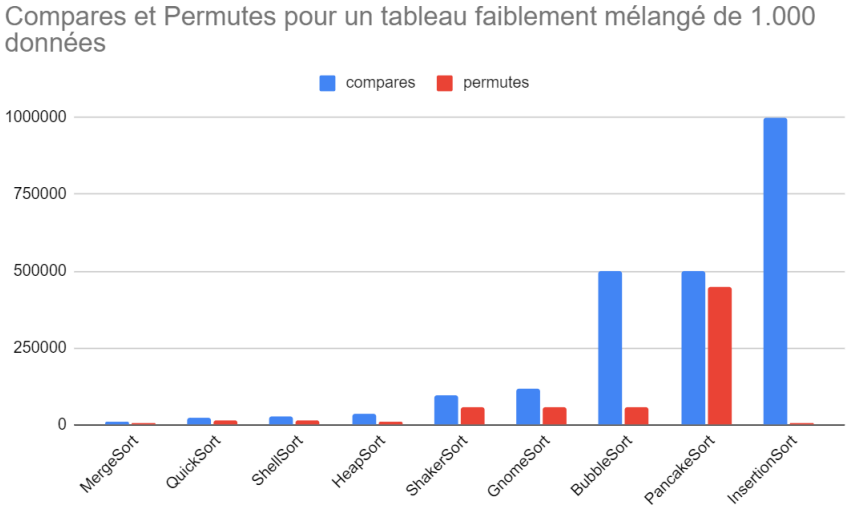
5.1 Descriptions des cas d'expérimentation

Pour l'analyse de nos données, nous avons distinguées 4 cas :

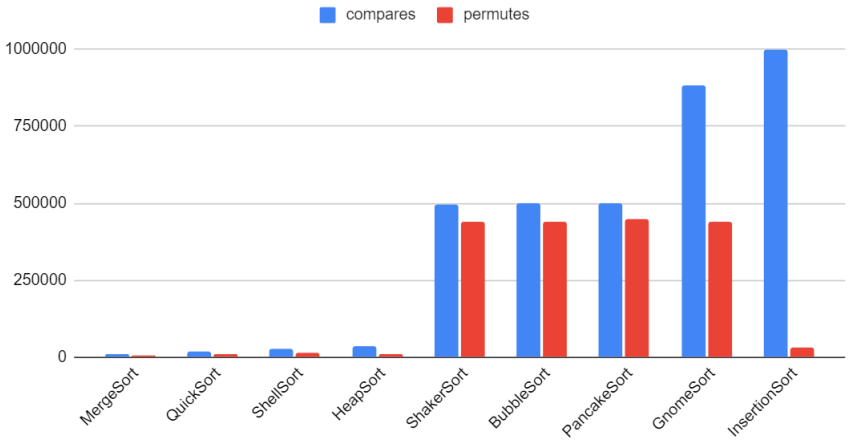
1. Un tableau trié aléatoirement, avec pour but de voire l'efficacité de l'algorithme sous un contexte couramment rencontré
2. Un tableau trié avec une très faible entropie, avec pour but de voire si avec peu de corrections à faire l'algorithme savait être plus rapide
3. Un tableau renversé et ensuite trié avec une faible entropie, avec pour but de voire si l'algorithme savait gérer un cas quasi pire efficacement
4. Un tableau avec uniquement la fin du tableau trié, avec pour but similaire que la faible entropie, mais pour les algorithmes nlogn qui utilisent souvent le principe de diviser pour régner on peut s'attendre à des différences

5.2 Résultats expérimentaux

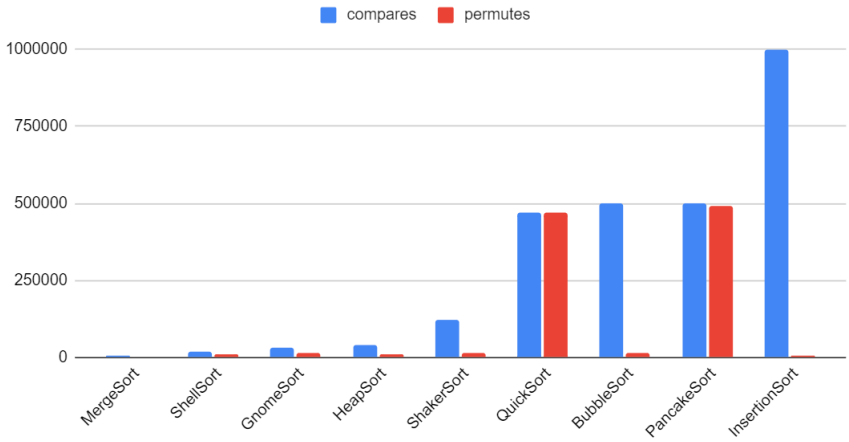


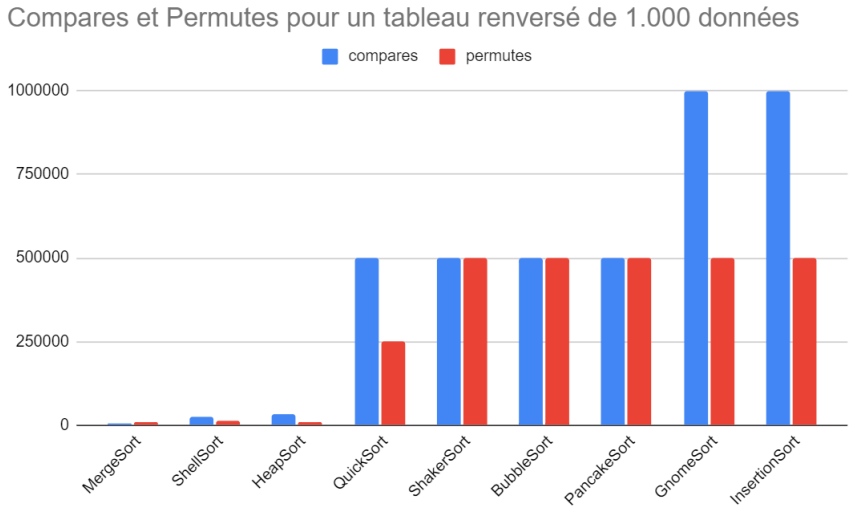


Compares et Permutes pour un tableau renversé et mélangé en faible entropie de 1.000 données



Compares et Permutes pour un tableau avec la fin du tableau mélangé en faible entropie de 1.000 données





5.3 Analyse des résultats

Dans le cas 1, voit une forte distinction entre nos algorithmes nlogn et nos algorithmes en n**2. Aucune surprise ici.

Dans le cas 2 on remarque que les algorithmes en nlogn ne changent pas beaucoup, cependant le GnomeSort et le ShakerSort savent efficacement trier le tableau, presque aussi efficacement que nos algos en nlogn

Dans le cas 3 on a des résultats similaires que dans le cas 1, mais avec la différence notable que les permutation et les comparaisons sont beaucoup plus proches. Ceci est attendu, avec un tableau inversé presque chaque comparaison devra donner lieu à une permutation

Dans le cas 3 il n'y a pas beaucoup de remarquable, les résultats sont similaires. Il est quand même très important de noter que QuickSort est très lent à faire le tri, et se retrouve a faire autant d'opérations que les tris en n**2

6 Conclusion

Chaque tri à des spécificités. Certains tris sont stables au niveau du nombre d'opérations, comme le mergesort et le bubble sort, alors que d'autres tris sont très efficaces dans certains cas spécifiques, mais souvent cette grande efficacité vient au coût d'une très grande inefficacité dans d'autres cas, comme le quicksort extrêmement lent à trier un tableau dont la fin est triée.
Dans le cas général, l'utilisation du mergesort semble être sur, mais avec certaines informations sur la répartition de nos données, il peut être utile de

connaitre d'autres tris qui peuvent être parfait pour nos besoins.
Il y a bien sur les tris plus ludiques comme le bogosort, que nous avons choisis de ne pas analyser afin d'obtenir des résultats avant la fin de l'univers

6.1 Récapitulatif de la problématique et de la réalisation

Notre but était de comparer des algorithmes de tri pour voir comment le désordre influait sur le fonctionnement de ceux-ci. Nous avons donc créé un générateur et des algorithmes de tri à tester.

6.2 Propositions d'améliorations

En guise d'améliorations nous avons plusieurs propositions :
— Une interface de démarrage d'algorithme où nous pouvons choisir l'algorithme, le type de mélange, l'entropie et la taille du tableau.
— Nous pourrions toujours rajouter des algorithmes de tri, nous en avons implémenté qu'une partie parmi tout ceux existants
— Dans l'idéal nous aurions pu essayer de créer notre propre algorithme de tri, même si c'est un projet très ambitieux.

7 Problèmes rencontrées

7.1 Traitement des données

Lors du développement de notre projet, nous avons réussi très rapidement à implémenter de nombreux algorithmes de tri ainsi qu'une visualisation de leur tri en action.
Malheureusement pour la partie analyse de leur efficacité nous avons eu de nombreuses approches qui n'ont pas fonctionné.
Dans un premier temps notre idée était de stocker les résultats dans un fichier sous format json, afin de pouvoir facilement lire, éditer et réécrire les données. Nous n'avons pas pu trouver de librairie qui permettait toute ces fonctionnalités, donc nous avons essayé de créé notre propre librairie à base d'ANTLR et les principes vu dans notre cours de Théorie des Langages et Compilation. Nous avons abandonnées cette approche avoir rencontré des problèmes au niveau de l'intégration de notre librarie dans le projet.
Finalement, par manque de temps, nous avons implémentees une approche rudimentaire sous format txt.

7.2 Radix

Le tri radix n'étant pas un tri par comparaison comme les autres tris, c'était difficile de choisir une métrique juste que l'on pouvait utiliser pour le comparer avec les autres.
De plus, le fait qu'il n'était pas par comparaison à fait que nous n'avons pas réussi à en faire une visualisation