**Rapport sur les Algorithmes de Tri S1.02**

***Yassine MABCHOUR***

**Une image contenant texte, Police, Graphique, logo

Description générée automatiquement**

Table des matières

[**Introduction** 2](#_Toc186626344)

[**Objectif** 2](#_Toc186626345)

[**Méthodologie** 2](#_Toc186626346)

[**Implémentation des algorithmes** 2](#_Toc186626347)

[**Initialisation des données de test** 2](#_Toc186626348)

[**Mesure des performances des différents algorithmes** 3](#_Toc186626349)

[**Visualisation des résultats des tests** 3](#_Toc186626350)

[**Analyse des résultats** 3](#_Toc186626351)

[**Conclusion** 4](#_Toc186626352)

# **Introduction**

Dans le cadre de ma formation informatique, il m’a été demandé d’effectuer une comparaison entre des algorithmes de tri afin de rendre mes connaissances sur leur fonctionnement, leurs performances, leurs avantages et leurs inconvénients au cas d’usage concerné. Le projet SAE S1.02, c’est-à-dire le projet sur lequel je travaille, m’offre l’occasion de consolider mes compétences en algorithmique tout en m’exerçant à programmer et à mesurer les performances de mon code. Ce rapport expose les méthodes employées, les résultats de mes expériences et une analyse plus fine de mes résultats.

# **Objectif**

Dans le cadre de ma formation, j’ai choisi d’orienter mes recherches vers la comparaison des algorithmes de tri afin de satisfaire aux exigences de ma SAE. En effet il s’agissait de mieux appréhender les différentes méthodes permettant de ranger des données dans un ordre précis, par exemple trier des numéros dans l’ordre croissant ou ranger des informations selon une règle édictée. Concrètement plusieurs façons de trier plusieurs séries d’éléments, appelées vecteurs, ont été analysées dites dans différentes configurations, dans un premier temps j’ai fait appel à une liste de nombres complètement mélangés, dans un deuxième temps j’ai eu recours à une liste dont la moitié des valeurs était déjà rangée, l’autre s’avérant encore aléatoire, pour finir enfin j’ai examiné les possibilités offertes par un tableau constitué d’au moins la moitié de ses valeurs rangées à l’envers de l’ordre croissant, et complété avec des nombres aléatoires.

# **Méthodologie**

## **Implémentation des algorithmes**

**Pour l’acheminement de mon projet, j’ai décidé de comparer plusieurs algorithmes de tri en utilisant les implémentations suivantes, chacune ayant ses particularités :**

**std ::sort :**

**C'est une méthode standard optimisée, que fournit la bibliothèque standard C++. Elle est réputée pour sa rapidité et sa performance grâce à la combinaison de plusieurs algorithmes internes (QuickSort, HeapSort, et InsertionSort)​.**

**std ::stable\_sort :**

**Cet algorithme a été retenu car il garantit que l’ordre relatif des éléments égaux est conservé. Cela correspond à des cas où la stabilité est requise, bien que son coût de performance soit légèrement supérieur à celui de std::sort​**

**Qsort :**

**Cet algorithme, qui fait partie de la bibliothèque standard C, constitue une version générique de QuickSort pour la comparaison des performances des solutions des bibliothèques standard.**

**SelectionSort :**

**C’est un algorithme très simple, qui parcourt la liste jusqu’à ce qu’il ait trouvé le plus petit élément à chaque itération et le place à sa position finale. Sa complexité le rend peu adapté à des listes très longues, il est majoritairement utilisé pour montrer les bases des algorithmes de tri.**

**InsertionSort :**

**Ce tri consiste à insérer un par un chaque élément à sa place dans une autre partie de la liste déjà triée. Très efficace pour des petites listes ou presque triées​.**

**BubbleSort :**

**Bien qu’on lui reproche bien souvent son inefficacité pour de grandes listes, j’ai choisi de l’inclure pour sa simplicité et son intérêt pédagogique. Il montre à la fois des limites très claires des méthodes naïves​.**

**QuickSort :**

**C’est une version classique d’un QuickSort avec un pivot (le premier élément de la liste) fixe. Il est rapide pour des données non ordonnées, mais peut stagner avec un mauvais choix du pivot.**

**V2QuickSort :**

**Un de mes premiers choix a été d’implémenter l’algorithme de QuickSort de manière modifiée par un choix aléatoire du pivot. Cette version améliore la performance pour une entrée moyenne en réduisant les partitions trop déséquilibrées qu’on pourrait rencontrer dans la version classique.**

**En combinant la diversité de ces algorithmes, je souhaitais réaliser un éventail allant des plus simples, pédagogiques, aux plus compliqués, optimisés pour commencer à comprendre les différences de performances, et les solutions adaptées pour chaque type de taille et de nature des données à trier.**

**Voici comment j’ai structuré mon code :**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**bin : Ce dossier est destiné à stocker les fichiers exécutables que je génère après la compilation. C’est ici que je mets mes programmes compilés.**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**data : Je l’utilise pour ranger les fichiers de données nécessaires à mon projet, comme des fichiers d’entrée, des configurations ou des résultats intermédiaires mais aussi les graphiques que j’ai généré.**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**data copy : Ce dossier est une copie du répertoire data, qui conservé pour tester mon code ou garder un backup temporaire.**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**src : C’est ici que je mets le code source de mon projet, comme mes fichiers .cpp, .hpp ou mon plot.sage.py.**

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

**.gitignore : C’est mon fichier qui collecte toutes les règles qui me permettent d’ignorer certains fichiers ou dossiers lors de mon usage de Github (afin de pouvoir toujours avoir le suivi de mon code sur n’importe quelle machine).**

**Une image contenant texte, Police, capture d’écran, nombre

Description générée automatiquement**

**build.sh : C’est mon script Bash qui me permet d’automatiser la compilation et la génération de mon projet.**

****

## **Initialisation des données de test**

Trois types de scénarios de tests d’évaluation ont été mis en place pour simuler différentes configurations de données ainsi que pour pouvoir tester les performances des divers algorithmes de tri dans des cas pratiques :

Données aléatoires :

C’est un vecteur généré de façon entièrement aléatoire (aucun ordre n’est respecté). Ce scénario représente les cas où l’on fait appel à des données brutes comme certaines données issues de mesures (ex. : mesures météo ou électriques, etc.).

Données semi triées :

C’est un vecteur dont une partie des éléments est déjà triée (par exemple, la moitié est en ordre croissant), l’autre partie est rangée dans le désordre. Ce cas ressemble à des situations où des données ont déjà été organisées partiellement (c’est souvent le cas pour le tri progressif ou par lots).

Données inversées :

C’est un vecteur constitué principalement d’éléments en tous points inversés, c’est-à-dire dans l’ordre décroissant, avec quelques ajouts de valeurs aléatoires. Ce cas permet de vérifier que certains algorithmes peuvent gérer des cas où l’ordre initial de traitement est contraire à celui souhaité de manière à obtenir le résultat.

Ces vecteurs ont été générés avec la bibliothèque random\_device utilisant des distributions uniformes pour produire les valeurs aléatoires. Ils sont initialisés dynamiquement dans le programme principal grâce à trois fonctions spécialisées correspondant chacune à l’un des types de scénario, assurant ainsi la répartition conforme des données pour les tests.

Ces scénarios permettront de mettre les algorithmes évalués dans des contextes divers, correspondant à des cas d’utilisation courants en informatique, mettant ainsi clairement en évidence leurs atouts et leurs faiblesses respectives.

Ces données serviront à évaluer les performances de chaque algorithme de tri dans différentes conditions. Chaque situation aura sa forte et sa faible partie, en fonction du type de données utilisées, qu’elles soient totalement aléatoires, partiellement triées ou inversées. Pour une analyse pertinente du comportement des algorithmes selon la nature des données traitées.

Les fonctions init\_vector\_test1, init\_vector\_test2 ainsi que init\_vector\_test3 ont été conçues afin d’initialiser un vecteur d’entiers avec des valeurs numériques précises en fonction de trois cas de figure différents pour tester les performances de divers algorithmes de tri. Le détail des fonctions est le suivant :

init\_vector\_test1 :

cette fonction init\_vector\_test1, génère un vecteur comportant des éléments représentant des valeurs aléatoires comprises entre un min et un max, la génération des nombres aléatoires étant faite grâce à la bibliothèque standard C++ (std::random\_device) et (std::mt19937).

On simule ici, un test aléatoire pour mesurer les performances d’un algorithme sans aucune « préinformation ».

init\_vector\_test2 :

La première moitié du vecteur est remplie avec des valeurs qu’on suppose croissantes,

La seconde moitié est occupée par une valeur aléatoire elle-même constituée grâce au même biais que ci-dessus,

On a donc ici un premier jeu de cas de mélange entre des données aléatoires et des données partiellement « triées ».

Init\_vector\_test3 :

La première moitié sont des valeurs décroissantes, la deuxième partie des valeurs aléatoires comme dans le cas précédent, il s’occupe de faire le test pour ce cas-là.

Ces 3 fonctions sont essentielles pour constituer un jeu de données d’entrées de caractéristiques différentes, qui permet de mesurer l’efficacité des algorithmes sur différents cas possible.

## **Mesure des performances des différents algorithmes**

Une image contenant capture d’écran, texte, ligne, Police

Description générée automatiquement

La durée d’exécution des algorithmes a été mesurée en microsecondes, à l’aide de la bibliothèque chrono. Chaque algorithme a été testé sur des vecteurs de tailles différentes, afin d’observer son comportement en fonction du volume de données :

La bibliothèque chrono du langage C++ fournit les classes permettant de capturer précisément le temps d’exécution par des microsecondes, ce qui est très utile pour des opération rapides.

Chaque algorithme fait l’objet d’un test indépendant des autres.  
Les tests consistent à exécuter l’algorithme sur des vecteurs de tailles différentes pour analyser comment varie son temps d’exécution en fonction de la taille des données fournies à l’algorithme.

## **Visualisation des résultats des tests**

Voici les algorithmes rapides :

Une image contenant texte, diagramme, ligne, Tracé

Description générée automatiquement

Voici les algorithmes lents :

Une image contenant texte, diagramme, ligne, capture d’écran

Description générée automatiquement

## **Analyse des résultats**

Algorithmes rapides : j’ai remarqué que dans les algorithmes dit « rapides » il y’a 2 agissements différents ou quicksort et V2quicksort sont moins performants que stable\_sort et stdsort dans le cas de données aléatoires (vecteur random).

Algorithmes lents : on remarque que quicksort a capoté (timeout) lors du tri des données croissantes (init\_vector\_test2) et décroissantes (init\_vector\_test3) parce que ça prend plus de 10 minutes. Bubble\_sort est un algorithme moyennement lent tandis que les algorithmes Qsort, Selection\_sort et Insertion\_sort sont les plus lents du a mon analyse.

# **Conclusion**

L’étude des algorithmes de tri a permis de mettre à jour des différences de performances selon le type des données et leur configuration. Les résultats démontrent que des algorithmes comme std::sort ou stable\_sort sont optimisés alors que d’autres comme bubble\_sort sont limités à un usage pédagogique. Ce projet avait comme but d’apprendre au-delà des différences des approches mais également à affiner une méthodologie d’évaluation algorithmique. Ces connaissances forment une plate-forme solide pour mes projets futurs.

***BONUS :***

Voici d’autres graphiques et leur commentaire que j’ai pu faire :

Bubble\_sort : Pour les trois tests du problème, le temps d’exécution croît de façon exponentielle par rapport à la taille des vecteurs. Comme en témoigne le premier test (en rouge), dont le temps d’exécution est nettement le plus élevé des trois. Le second test (en bleu) et le troisième test (en jaune) sont légèrement plus performants, néanmoins ils ne bénéficient pas d’un rapport tant en termes d’efficacité de l’algorithme qui les met en œuvre.

Une image contenant texte, diagramme, Tracé, ligne

Description générée automatiquement

Insertion\_sort : La courbe superposée quant au temps augmente également relativement vite mais un peu moins que le Bubble Sort. Le test3 (jaune) est le plus lent, à cause de données initialement inversées. Le test2 (bleu) est le plus efficace, ce qui suggère qu’il est plus adéquatement ajusté à des données partiellement triées.

Une image contenant texte, ligne, diagramme, Tracé

Description générée automatiquement

Qsort : Les temps d’exécution, pour les très grandes données, progressent beaucoup plus modérément, ce qui traduit une beaucoup meilleure gestion des grandes données. Les différences entre les tests sont moins significatives puisque Qsort est performant dans plusieurs types de distribution (comme vu dans les tests).

Une image contenant texte, Police, capture d’écran, Tracé

Description générée automatiquement

Quicksort : On note une variation significative des temps d’exécution pour de très grandes tailles. Les tests 2 et 3, (bleu et jaune), semblent montrer un comportement plus capricieux.

Une image contenant texte, diagramme, ligne, Tracé

Description générée automatiquement

Selection\_sort : Les performances sont du même ordre que le Bubble Sort avec une forte montée des temps. Les trois tests montrent des temps d’exécution similaires.

Une image contenant diagramme, texte, Tracé, ligne

Description générée automatiquement

Stable\_sort : Une performance acceptable obtenue pour les plus petites données, mais des temps croissants avec des fluctuations. Les tests 1 et 2 (rouge et bleu) apparaissent légèrement plus stables que test3.

Une image contenant texte, capture d’écran, Police, Tracé

Description générée automatiquement

Std ::sort : Des temps d’exécution courts bien linéaires pour toutes les tailles. Les trois tests montrent peu de différence en termes de performance. Des temps d’exécution courts bien linéaires pour toutes les tailles.

Une image contenant texte, capture d’écran, Tracé, diagramme

Description générée automatiquement

V2quicksort : Une amélioration significative par rapport à l’implémentation standard QuickSort. Les performances sont comparables entre les trois tests, avec une courbe quasiment linéaire.

Une image contenant texte, capture d’écran, Tracé, ligne

Description générée automatiquement