UFR DES SCIENCES

Licence III
Informatique
Rapport

Analyse des algorithmes de tri

Étudiants :
Mariatou DIALLO
Samir MAOUDE
Patrice COTCHO
Aissatou DIALLO

Enseignants :
François RIOULT

28 mars 2025

Table des matières

1 Introduction **3**

2 Objectifs du projet **3**
2.1 Problématique du projet . . . 3

3 Fonctionnalités implémentées **5**
3.1 Description des fonctionnalités . . . 5
3.1.1 Expérimentation . . . 5
3.1.2 Visualisation du déroulé d'algorithme de tri . . . 5
3.2 Organisation du projet . . . 5

4 Éléments techniques **6**
4.1 Description des algorithmes . . . 6
4.1.1 Tri à bulles (Bubble Sort) : . . . 6
4.1.2 Tri par insertion (Insertion Sort) : . . . 6
4.1.3 Tri fusion (Merge Sort) : . . . 6
4.1.4 Timsort : . . . 6
4.1.5 Tri par tas (Heap Sort) : . . . 6
4.1.6 Tri rapide (Quick Sort) : . . . 6
4.2 Génération de tableau avec une valeur d'entropie . . . 7
4.2.1 Définition de l'Entropie . . . 7
4.2.2 Méthode de Génération de la Distribution de Probabilité . . . 7
4.2.3 Génération d'un Tableau avec une Entropie Donnée . . . 8
4.3 Architecture du projet . . . 8
4.3.1 Description des paquetages non standards utilisés . . . 8
4.3.2 Diagrammes des modules et des classes . . . 9
4.3.3 Chaînes de traitement . . . 10

5 Expérimentations **11**
5.1 Lancement du programme . . . 11
5.1.1 Expérimentation . . . 11
5.1.2 Visualisation . . . 12
5.2 Expérimentation . . . 12
5.3 Résultats et interprétations des expérimentations . . . 13
5.3.1 Évolution du nombre d'échange . . . 13
5.3.2 Évolution du nombre de comparaison . . . 15
5.3.3 Évolution du nombre d'accès . . . 17
5.4 Conclusion sur les interpretations . . . 18

6 Conclusion **19**

A Tri bulle **21**

B Tri insertion **21**

C Timsort **21**

D Tri fusion **23**

E Tri par tas **25**

F Tri rapide **25**

G Résultats sur les tableaux générés avec un pourcentage d'échange **27**
G.1 Évolution du nombre d'accès au tableau 27
G.2 Évolution du nombre de comparaison au tableau 28
G.3 Évolution du nombre d'échange effectué dans le tableau 29

1 Introduction

Le tri des données constitue une problématique fondamentale en informatique, sur laquelle repose de nombreuses applications et domaines d'étude. De ce fait, une multitude d'algorithmes de tri ont été conçus au fil du temps, chacun répondant à des objectifs spécifiques et présentant des performances variables en fonction des contextes d'utilisation. Si certains algorithmes, comme le tri à bulles, sont réputés pour leur simplicité mais leur faible efficacité, d'autres, tels que le tri rapide ou le tri par tas, se démarquent par leurs performances optimales dans des scénarios généraux. Toutefois, il est essentiel de souligner que l'efficacité d'un algorithme de tri ne dépend pas uniquement de sa complexité intrinsèque, mais également des caractéristiques des données à trier, notamment leur niveau de désordre et leur répartition.

Ce projet s'inscrit dans cette perspective d'analyse et vise à concevoir une chaîne de traitement complète dédiée à l'évaluation des performances d'algorithmes de tri en fonction des caractéristiques des données.

Ce rapport est structuré en quatre grandes parties. La première partie définit la problématique ainsi que les travaux à réaliser pour y répondre. La deuxième partie présente les fonctionnalités implémentées et l'organisation adoptée pour leur mise en œuvre. La troisième partie est consacrée aux aspects techniques. Enfin, une dernière section est dédiée au lancement du programme et à l'analyse des résultats des expérimentations.

2 Objectifs du projet

2.1 Problématique du projet

Dans quelle mesure les performances des algorithmes de tri (en termes de nombre de comparaisons, d'accès aux données et d'échange) dépendent-elles du niveau de désordre et de la répartition des données initiales ?

L'objectif principal de ce projet est de concevoir une chaîne complète de traitement permettant d'évaluer le comportement de différents algorithmes de tri face à des jeux de données caractérisés par des niveaux de désordre variés.

Pour ce faire, plusieurs étapes seront menées :

Conception d'un générateur de données non triées paramétrable : ce générateur permettra de produire des ensembles de données dont le niveau de désordre est contrôlable, en jouant sur des paramètres tels que la quantité de désordre, l'entropie, ou encore la répartition des données.

Implémentation de plusieurs algorithmes de tri : six algorithmes, choisis pour leur diversité en termes de comportement et de complexité, seront implémentés. Ces algorithmes incluront des approches aux caractéristiques variées, permettant une étude comparative approfondie.

Création d'une interface générique de visualisation : une interface interactive sera développée pour permettre la visualisation de l'exécution des algorithmes. Cet outil aidera à illustrer le déroulement des opérations de tri et à mieux observer les différences de fonctionnement entre les algorithmes.

Analyse comparative des performances : une évaluation méthodique sera réalisée en fonction de divers critères, tels que le nombre de comparaisons effectuées, les accès aux données et le nombre d'échanges. L'étude portera sur la corrélation entre ces métriques la taille et le niveau de désordre des données générées.

3 Fonctionnalités implémentées

3.1 Description des fonctionnalités

Dans le cadre de ce projet, nous avons mis en place deux principales fonctionnalités.

3.1.1 Expérimentation

Pour cette fonctionnalité, nous avons mis en place un fichier de configuration au format JSON, permettant à l'utilisateur de spécifier les paramètres de l'expérimentation. Deux générateurs sont utilisés à cet effet : l'un repose sur un nombre d'échanges à effectuer sur un tableau déjà trié, tandis que l'autre se base sur une mesure d'entropie. Une fois ce fichier rempli et le programme lancé, les résultats de l'expérimentation sont stockés dans un dossier Résultats. Ces résultats prennent la forme de fichiers image illustrant l'évolution des sondes (nombre d'échanges, nombre de comparaisons et nombre d'accès au tableau à trier) pour chaque algorithme de tri, appliqué à des jeux de données spécifiques.

3.1.2 Visualisation du déroulé d'algorithme de tri

Cette fonctionnalité a été développée dans le but d'observer le comportement de chacun des algorithmes, au cours de leur exécution. Pour ce faire, l'utilisateur peut, via une interface graphique, configurer le type de tableau à trier, sélectionner les algorithmes de tri souhaités, puis lancer la visualisation.

3.2 Organisation du projet

Pour mener à bien ce projet, nous avons adopté une méthodologie axée sur la collaboration et la spécialisation. Avant de commencer à coder, nous avons pris le temps, en équipe, de réfléchir au sujet, de discuter de la conception globale et de définir les grandes lignes de l'architecture logicielle. Cette phase collective a permis d'assurer une compréhension commune des objectifs et des contraintes du projet, tout en orientant nos choix techniques. Ensuite, nous avons réparti les tâches en fonction des compétences et des points forts de chacun afin de maximiser notre efficacité :

- **Samir** a pris en charge la mise en place des générateurs, ainsi que la structure du fichier de configuration des expérimentations et sa lecture.
- **Patrice** s'est concentré sur le développement des classes impliquées dans l'exécution des expérimentations et la production des résultats. Il a également travaillé sur l'interface graphique du projet.
- **Mariatou** a implémenté les algorithmes de tri utilisés dans l'expérimentation
- **Aissatou** s'est chargée de la production de ressources pour le rapport (notamment les diagrammes de classe) avec l'aide de **Mariatou**.

Parallèlement à nos tâches respectives, nous avons collaboré à la rédaction du rapport et à la mise en place des tests unitaires. Ce travail collectif nous a permis de garantir la robustesse de notre code en vérifiant son bon fonctionnement à chaque étape et en identifiant rapidement les éventuels problèmes.

4 Éléments techniques

4.1 Description des algorithmes

Étant donné la diversité des algorithmes de tri existants, nous avons sélectionné six d'entre eux, parmi les plus connus. Ces algorithmes ont été choisis dans l'optique d'avoir une diversité en terme de comportement, des structures de données qu'ils emploient, ainsi que de leurs complexités en termes de temps et d'espace.

4.1.1 Tri à bulles (Bubble Sort) :

Le pseudo-code est disponible à l'annexe A page 21.

— Complexité temporelle : $O(n^2)$ dans le pire des cas

4.1.2 Tri par insertion (Insertion Sort) :

Le pseudo-code est disponible à l'annexe B page 21.

— Complexité temporelle : $O(n^2)$ dans le pire des cas.

4.1.3 Tri fusion (Merge Sort) :

Le pseudo-code est disponible à l'annexe D page 23.

— Complexité temporelle : $O(n \log n)$.

4.1.4 Timsort :

Le pseudo-code est disponible à l'annexe C page 21.

— Complexité temporelle : $O(n \log n)$.

4.1.5 Tri par tas (Heap Sort) :

Le pseudo-code est disponible à l'annexe E page 25.

— Complexité temporelle : $O(n \log n)$.

4.1.6 Tri rapide (Quick Sort) :

Le pseudo-code est disponible à l'annexe F page 25.

— Complexité temporelle : $O(n \log n)$ en moyenne, $O(n^2)$ dans le pire cas.

Cette diversité permet d'évaluer comment chaque algorithme réagit face à des données de nature variée et à différents niveaux de désordre. L'analyse comparative des performances, en termes de nombre de comparaisons, d'accès aux données et de temps d'exécution, nous offrira une vision complète de l'efficacité de chaque algorithme dans des conditions spécifiques.

4.2 Génération de tableau avec une valeur d'entropie

L'objectif est de générer un ensemble de données de taille N avec une entropie cible spécifiée. L'entropie de Shannon est utilisée pour mesurer la diversité des valeurs dans l'ensemble généré.

On s'est inspiré de cet article [1] qui propose une méthode itérative basée sur la modification de paires de probabilités pour ajuster l'entropie d'une distribution.

4.2.1 Définition de l'Entropie

L'entropie d'une distribution discrète $P = \{p_1, p_2, \ldots, p_N\}$ est définie par :

$$H(P) = -\sum_{i=1}^{N} p_i \log_2 p_i. \tag{1}$$

L'entropie cible H_{cible} est normalisée par rapport à l'entropie maximale de la distribution, définie par $H_{max} = \log_2 N$. Ainsi, la valeur de H_{cible} varie dans l'intervalle $[0, 1]$.

4.2.2 Méthode de Génération de la Distribution de Probabilité

L'approche décrite dans [1] consiste à ajuster une distribution existante en modifiant deux probabilités (p_k, p_l) pour créer une nouvelle paire (p'_k, p'_l) tout en conservant la somme constante :

$$p_k + p_l = p'_k + p'_l = \sigma. \tag{2}$$

Les nouvelles valeurs sont définies par :

$$p'_k = \beta\sigma, \quad p'_l = (1 - \beta)\sigma. \tag{3}$$

Le changement d'entropie résultant est donné par :

$$\Delta H = \sigma\Big(g(\alpha) - g(\beta)\Big), \tag{4}$$

avec $g(x) = x \log x + (1 - x) \log(1 - x)$. La valeur de α est définie comme :

$$\alpha = \frac{p_k}{\sigma}. \tag{5}$$

Afin de garantir la convergence vers l'entropie cible H_{cible}, on sélectionne β de manière aléatoire dans un intervalle dépendant du signe de $H(P) - H_{cible}$.

— Si $H(P) > H_{cible}$, alors β est choisi dans l'intervalle $(0, \alpha)$.

— Si $H(P) < H_{cible}$, alors β est sélectionné dans l'intervalle $(\alpha, 0.5)$.

Cette sélection aléatoire permet d'introduire de la variabilité tout en assurant la convergence vers l'entropie souhaitée. L'algorithme 1 montre une implémentation de cette méthode.

Algorithm 1 Génération d'une Distribution de Probabilité avec Entropie Cible

Require: k (nombre de symboles uniques), H_{cible} (entropie cible), ϵ (tolérance)
Ensure: Distribution P telle que $|H(P) - H_{cible}| < \epsilon$
1: Initialiser P avec des valeurs aléatoires
2: Normaliser P pour que $\sum p_i = 1$
3: Calculer $H(P)$
4: **while** $|H(P) - H_{cible}| > \epsilon$ **do**
5: Sélectionner aléatoirement deux indices i, j
6: Ajuster p_i et p_j en respectant la somme constante
7: Mettre à jour $H(P)$
8: **end while**
9: **return** P

4.2.3 Génération d'un Tableau avec une Entropie Donnée

Une fois la distribution P obtenue, nous générons un tableau de taille N en échantillonnant selon cette distribution (voir l'algorithme 2).

Algorithm 2 Génération d'un Tableau avec Entropie Donnée

Require: N (taille du tableau), H_{cible} (entropie cible)
Ensure: Tableau T de taille N
1: Calculer k tel que $H_{max} = \log_2 k \approx H_{cible}$
2: Générer une distribution P avec entropie H_{cible}
3: Générer T en tirant N valeurs selon P
4: **return** T

Des tests ont été réalisés pour différentes valeurs de N et d'entropie cible afin de vérifier la convergence et la qualité des distributions générées.

4.3 Architecture du projet

4.3.1 Description des paquetages non standards utilisés

Nous avons utilisé la bibliothèque **com.github.sh0nk** pour générer les courbes illustrant les résultats de nos expérimentations. Elle fait référence à des bibliothèques développées par l'utilisateur GitHub "sh0nk", notamment matplotlib4j et solr-sudachi. Matplotlib4j permet aux développeurs Java d'exploiter la puissance de la bibliothèque Python Matplotlib pour la création de graphiques.

Pour implémenter la fonctionnalité de visualisation du déroulement des algorithmes, nous avons utilisé la bibliothèque **Processing Core**. Ce module autonome

fournit les fonctionnalités de base de Processing, notamment l'affichage graphique, l'interaction avec l'utilisateur (clavier, souris) et la gestion des animations. Plus précisément, nous avons exploité la classe *processing.core.PApplet* pour créer la fenêtre de visualisation de l'exécution des algorithmes.

4.3.2 Diagrammes des modules et des classes

Le projet contient en tout quatre packages regroupés par fonction. Le package *model* contient les classes de générateur et d'algorithme de tri. Le package *view* contient tout ce qui traite de l'interface graphique. Les packages *experience* et *visualisation* contiennent respectivement les classes intervenant dans la mise en place des fonctionnalités **expérimentation** et **visualisation**. Les figures 1 et 2 montrent les diagrammes décrivant la structure des classes de générateur et d'algorithme.

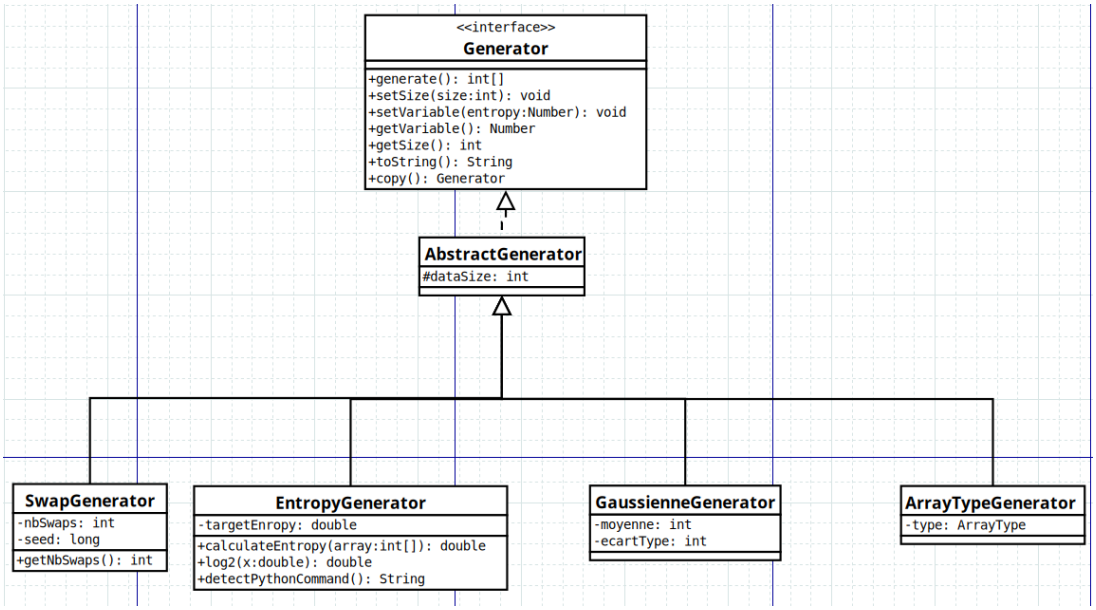

Figure 1 – Diagrammes des classes de générateur

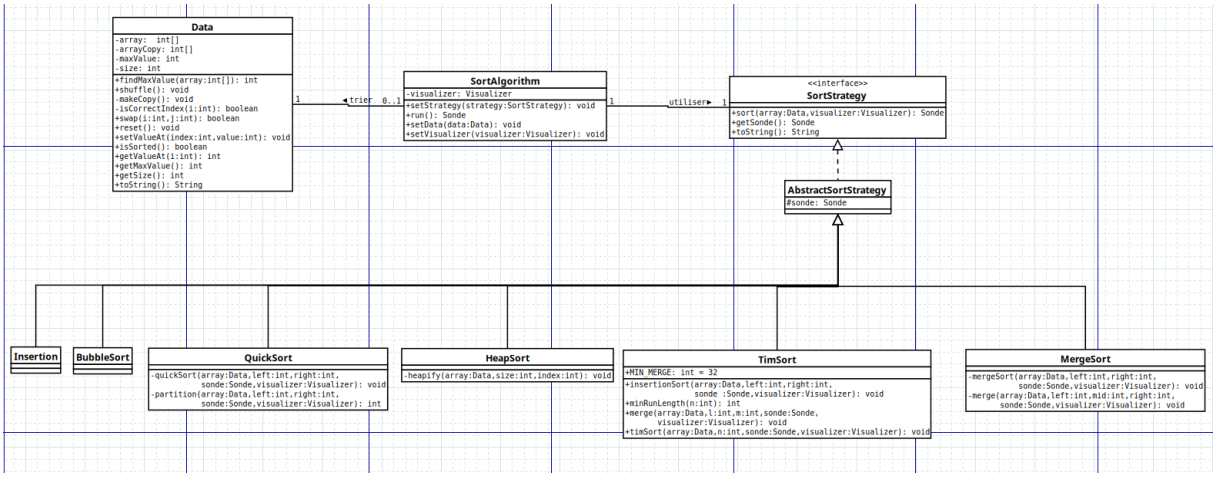

Figure 2 – Diagramme des classes des algorithmes

4.3.3 Chaînes de traitement

Pour mener une expérimentation, la classe *Experiment*, en s'appuyant sur la classe Configuration, est chargée de créer la bonne instance de générateur. Ce générateur produit le tableau à trier (représenté par la classe *Data*). Ce tableau est ensuite soumis aux différents algorithmes de tri pour son traitement et la mesure des performances. Le résultat du tri est représenté par la classe *Result*, qui contient une instance de la classe *Sonde*. Enfin, les résultats sont analysés afin de générer les graphiques à l'aide de la classe *GraphConstructor*.

Pour la visualisation, les classes du package view permettent à l'utilisateur de définir les paramètres du générateur et de choisir l'algorithme à visualiser. Une fois collectées, ces informations forment une instance de *DataSet*. Cette dernière crée ensuite une instance de *Runner* pour lancer la visualisation du déroulement du tri.

5 Expérimentations

5.1 Lancement du programme

Pour lancer le programme, il est nécessaire de remplir les pré-requis suivants :

— installer gradle, version 8.12 : **sudo apt install gradle -y**

— installer python 3 : **sudo apt install python3 python3-pip**

— installer Matplotlib et NumPy : **pip install matplotlib numpy**

5.1.1 Expérimentation

Pour lancer une expérimentation, il faut d'abord la configurer. Pour cela, dans le fichier config.json, il est nécessaire de renseigner les informations sur le tableau à trier ainsi que le type de générateur utilisé. Le code json 1 montre un extrait de son contenu.

Listing 1 – Exemple de configuration JSON

```
{
    "configurations":[
        {
            "size": 1000000,
            "generator": {
                "type": "SHANNON_ENTROPY",
                "xvalues": [0, 0.1, 0.2, 0.3,0.4,0.5,0.6,0.7,0.8, 0.9, 1.0]
            },
            "arraysNumber": 1
        },
        {
            "size": 1000000,
            "generator": {
                "type": "SWAPS",
                "xvalues": [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
            },
            "arraysNumber": 1
        }
    ]
}
```

Dans l'exemple présenté, on cherche à observer l'évolution des algorithmes de tri en fonction de l'augmentation de l'entropie dans un tableau de taille 1 000 000. L'attribut arraysNumber définit le nombre de tableau sur lequel tester une entropie donnée avant de calculer la moyenne des sondes pour cette entropie.

Dans une deuxième configuration, on utilise un générateur basé sur le nombre de swaps. Ici, il faut mentionner une liste de pourcentages d'échange, le nombre d'échanges étant compris entre 0 et taille/2. L'objectif est d'observer l'évolution des sondes sur un tableau de taille 1 000 000 en fonction du nombre d'échanges effectués dans le tableau.

Une fois le fichier rempli, il suffit de lancer le programme avec la commande : **gradle run**. Les résultats sont générés dans le dossier **Résultats** qui se trouve dans le dossier **app** du projet.

5.1.2 Visualisation

Pour visualiser comment se déroule le tri avec les différents algorithmes, il faut utiliser la commande **gradle run -PmainClass=view.Main**. L'interface graphique de configuration s'ouvrira et il faudra juste définir les réglages.

5.2 Expérimentation

Le json 2 montre notre fichier de configuration. Ici, nous cherchons à voir comment les sondes évoluent sur des tableaux de petite (100), moyenne (100000) et grande (1000000) taille, en fonction de l'entropie et du nombre d'échange.

Listing 2 – Plan d'expérimentation

```
{
    "configurations":[
        {
            "size": 100,
            "generator": {
                "type": "SHANNON_ENTROPY",
                "xvalues": [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
                    1.0]
            },
            "arraysNumber": 1
        },
        {
            "size": 100000,
            "generator": {
                "type": "SHANNON_ENTROPY",
                "xvalues": [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
                    1.0]
            },
            "arraysNumber": 1
        },
        {
            "size": 1000000,
            "generator": {
                "type": "SHANNON_ENTROPY",
                "xvalues": [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
                    1.0]
            },
            "arraysNumber": 1
        },
        {
            "size": 100,
```

```
            "generator": {
                "type": "SWAPS",
                "xvalues": [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
            },
            "arraysNumber": 1
        },
        {
            "size": 100000,
            "generator": {
                "type": "SWAPS",
                "xvalues": [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
            },
            "arraysNumber": 1
        },
        {
            "size": 1000000,
            "generator": {
                "type": "SWAPS",
                "xvalues": [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
            },
            "arraysNumber": 1
        }
    ]
}
```

5.3 Résultats et interprétations des expérimentations

L'évolution des courbes reste identique sur les tableaux de taille 100000 et 1000000. Par souci de simplicité, nous allons afficher que les graphiques montrant l'évolution des sondes sur des tableaux de tailles 100 et 1000000. Aussi, nous allons nous limiter aux résultats des tableaux générés avec entropie. Les autres résultats peuvent être consultés à l'annexe G

5.3.1 Évolution du nombre d'échange

Les figures 3 et 4 montrent les résultats de l'évolution du nombre d'échanges effectués par les algorithmes en fonction du désordre dans le tableau. On remarque que :

Graphique obtenu avec le générateur se basant sur l'entropie : InsertionSort, TimSort et MergeSort ne réalisent aucun échange. Plus l'entropie évolue, moins HeapSort est efficace comparé à QuickSort et BubbleSort, dans le cas des tableaux de petites tailles (100). Sur de grandes tailles, BubbleSort devient nettement moins performant. QuickSort maintient un nombre d'échanges constant, quelle que soit la taille du tableau ou le niveau de désordre.

En revanche, en l'absence de désordre, QuickSort effectue un nombre élevé d'échanges, le rendant moins efficace que HeapSort et BubbleSort dans ces

conditions.

Graphique obtenu avec le générateur se basant sur le pourcetage d'échange :
HeapSort est moins efficace que les autres sur des tableaux de petites tailles, quelle que soit la proportion d'échanges.

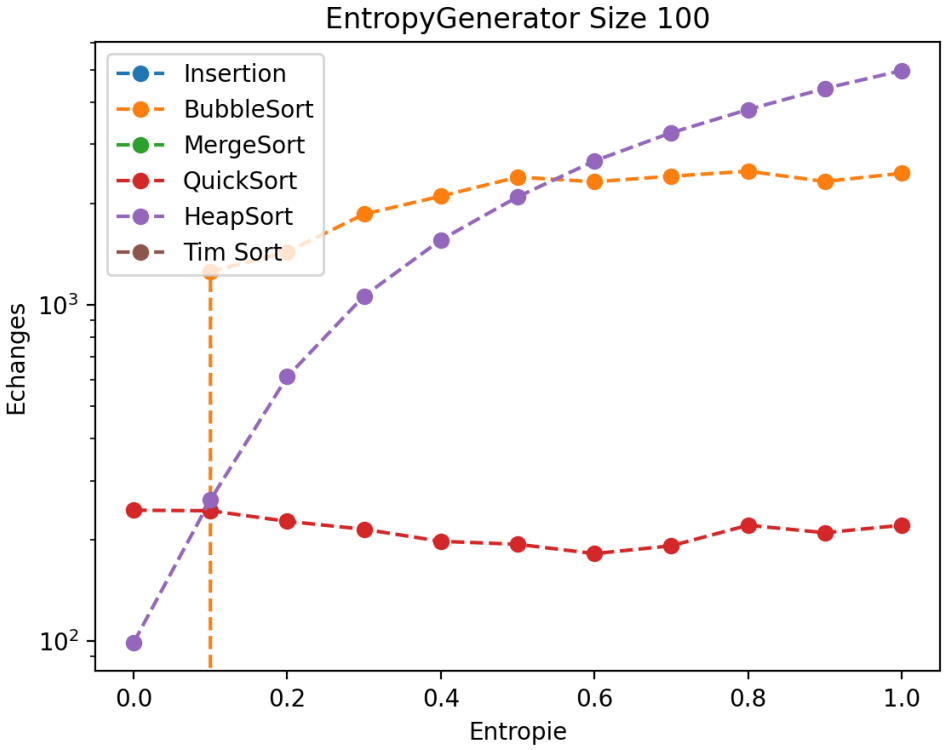

Figure 3 – Échange en fonction de l'entropie sur taille 100

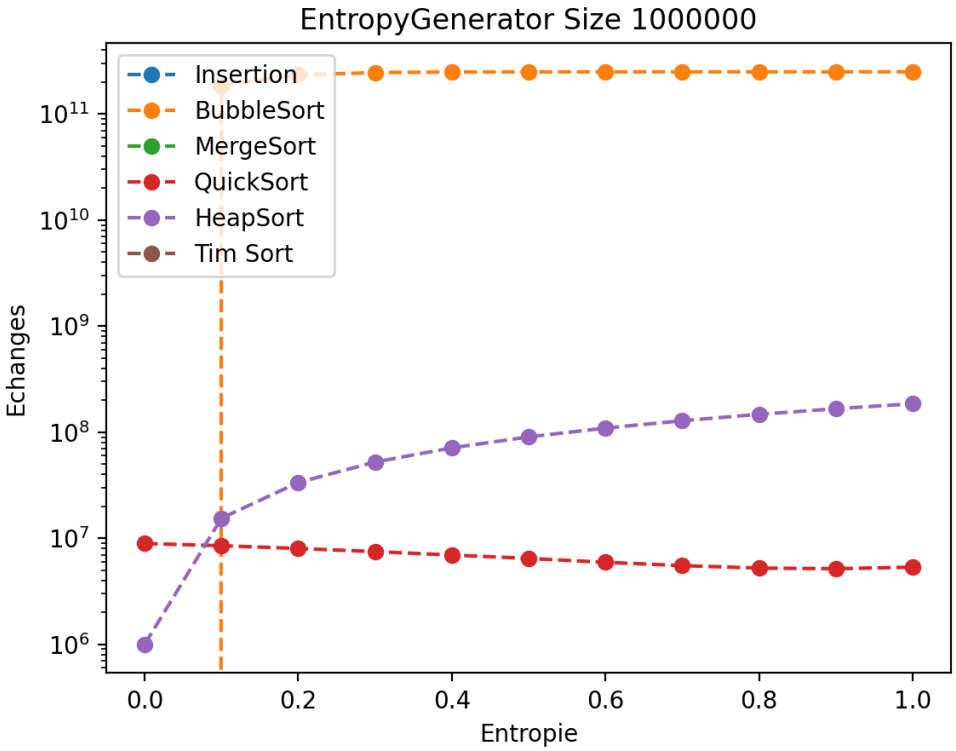

Figure 4 – Accès en fonction de l'entropie sur taille 1000000

5.3.2 Évolution du nombre de comparaison

Les figures 5 et 6 montrent les résultats de l'évolution du nombre de comparaisons effectuées par les algorithmes en fonction du désordre dans le tableau.

Graphique obtenu avec le générateur se basant sur l'entropie : Sur des tableaux sans désordre, InsertionSort et BubbleSort sont les plus efficaces. Toutefois, leur nombre de comparaisons augmente rapidement dès qu'un léger désordre apparaît. Lorsque le niveau de désordre augmente, le nombre de comparaisons de HeapSort croît jusqu'à dépasser InsertionSort et BubbleSort, dans le cas des tableaux de petites tailles.

QuickSort, MergeSort et TimSort restent relativement constants, avec un léger avantage de MergeSort sur QuickSort et de QuickSort sur TimSort.

Graphique obtenu avec le générateur se basant sur le pourcetage d'échange : Sur des tableaux de petites tailles, HeapSort est systématiquement moins performant que tous les autres en termes de comparaisons, quelle que soit la proportion= d'échanges.

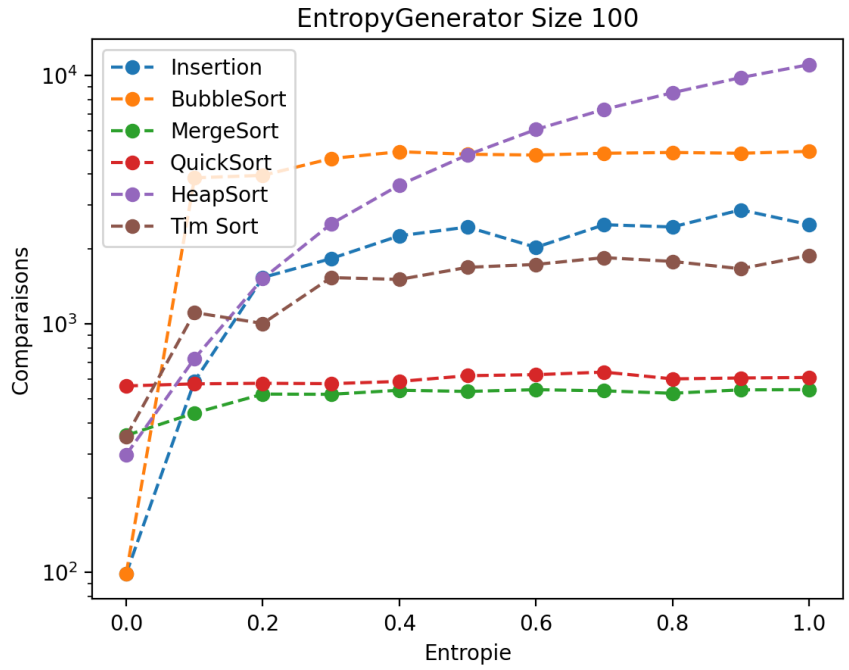

Figure 5 – Comparaison en fonction de l'entropie sur taille 100

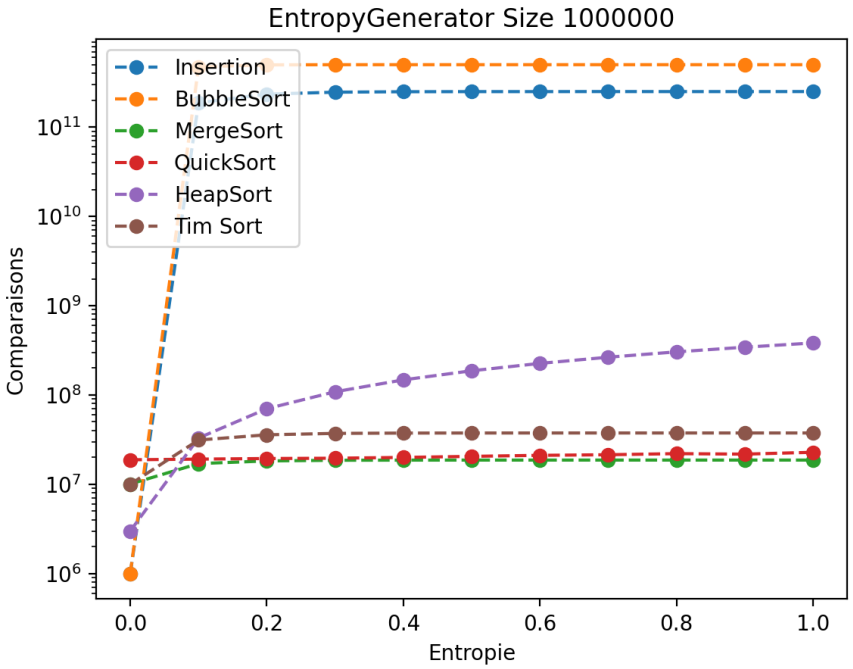

Figure 6 – Comparaison en fonction de l'entropie sur taille 1000000

5.3.3 Évolution du nombre d'accès

Les figures 7 et 8 montrent les résultats de l'évolution du nombre de comparaisons effectuées par les algorithmes en fonction du désordre dans le tableau.

Graphique obtenu avec le générateur se basant sur l'entropie : Sur des tableaux sans désordre, InsertionSort et BubbleSort sont les plus efficaces. Cependant, leur nombre d'accès explose dès l'apparition d'un faible niveau de désordre, BubbleSort étant systématiquement moins efficace qu'InsertionSort. À mesure que le désordre augmente, HeapSort voit son nombre d'accès croître jusqu'à dépasser InsertionSort et BubbleSort, dans le cas des tableaux de petites tailles. QuickSort, MergeSort et TimSort restent relativement constants, avec un avantage de QuickSort sur MergeSort et de MergeSort sur TimSort.

Graphique obtenu avec le générateur se basant sur le pourcetage d'échange : Sur des tableaux de petites tailles, HeapSort reste toujours moins performant en termes d'accès que tous les autres algorithmes, quel que soit le pourcentage d'échanges.

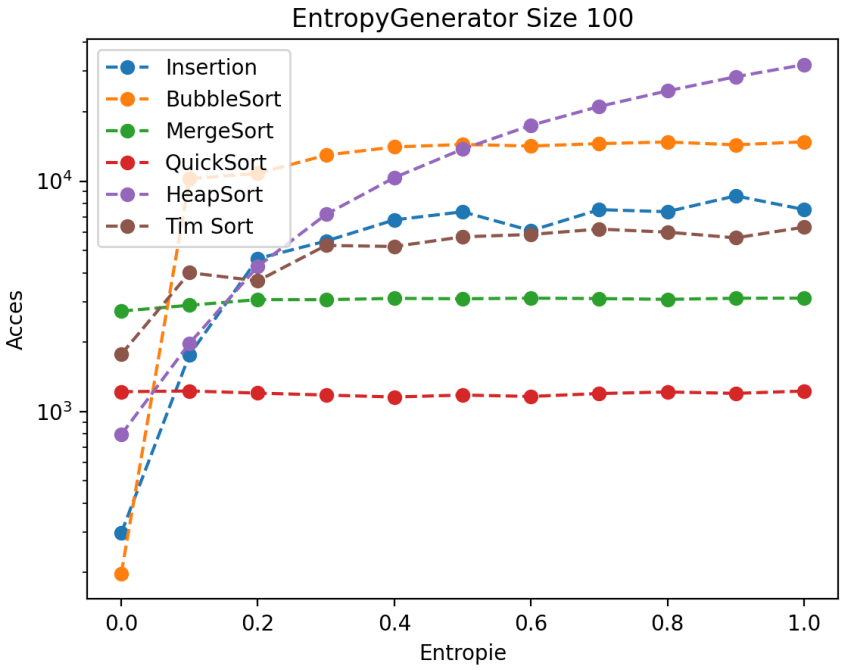

Figure 7 – Accès en fonction de l'entropie sur taille 100

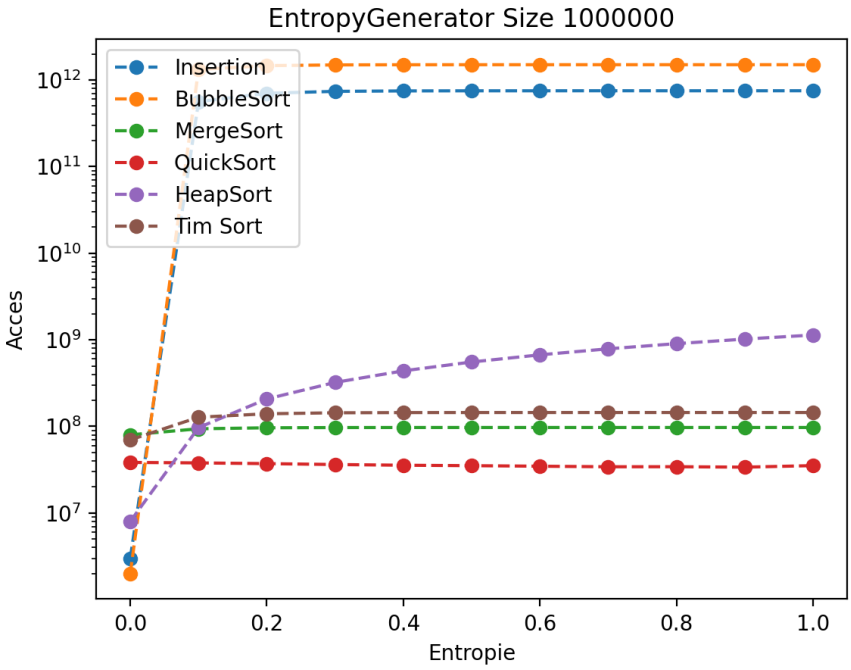

Figure 8 – Accès en fonction de l'entropie sur taille 1000000

5.4 Conclusion sur les interpretations

QuickSort se montre performant sur toutes les tailles de tableaux et à tous les niveaux de désordre. Pour des tableaux avec un désordre quasi nul, InsertionSort et BubbleSort sont les plus efficaces. Parmi les algorithmes de complexité quasi-linéaire $O(n \log n)$, MergeSort, QuickSort et TimSort présentent des performances relativement constantes, quelle que soit la taille du tableau ou le niveau de désordre. En revanche, la courbe de performance de HeapSort croit avec le niveau de désordre, ce qui le rend moins intéressant que les autres algorithmes de cette catégorie.

6 Conclusion

L'objectif de ce projet était de répondre à la question suivante : dans quelle mesure les performances des algorithmes de tri dépendent-elles du niveau de désordre et de la répartition des données initiales ?

Pour y parvenir, nous avons mené des expérimentations sur six algorithmes de tri en les appliquant à des jeux de données générés avec différentes propriétés. Par ailleurs, nous avons développé une fonctionnalité de visualisation permettant d'observer le comportement de chaque algorithme au cours de son exécution. Cette représentation graphique nous a offert une première intuition sur l'efficacité relative des algorithmes en fonction de la structure des données à trier.

Les résultats obtenus montrent que InsertionSort et BubbleSort sont efficaces sur des tableaux faiblement désordonnés, mais deviennent rapidement inadaptés lorsque le niveau de désordre augmente. Parmi les algorithmes de complexité quasi-linéaire $O(n \log n)$, MergeSort, QuickSort et TimSort offrent des performances relativement stables, quel que soit le niveau de désordre ou la taille du tableau. En revanche, HeapSort se révèle moins performant sur des ensembles de petite taille, sa courbe de performance se détériorant significativement avec le désordre, ce qui le rend moins intéressant que les autres algorithmes de sa catégorie.

Bien que ce projet ait atteint ses objectifs initiaux, plusieurs améliorations sont envisageables :

- Affiner l'analyse des performances en prenant en compte d'autres facteurs, tels que l'impact de la mémoire cache et l'optimisation des accès mémoire.
- Étudier d'autres types de données, notamment des tableaux partiellement triés ou suivant des distributions plus complexes (exponentielle, gaussienne, etc.).
- Élargir la comparaison en intégrant des algorithmes hybrides ou récents, comme IntroSort, qui combine les avantages de QuickSort et HeapSort.
- Optimiser l'outil de visualisation en ajoutant des métriques dynamiques, comme le nombre d'opérations effectuées en temps réel.

Références

[1] Kornilios Kourtis. *Notes on Generating Probability Distributions for a Given Entropy Value*. Lien vers l'article.

A Tri bulle

Algorithm 3 Tri à bulles

```
1: procedure tri_bulles(T)
2:     N ← taille de T
3:     for i = N − 1 à 1 do
4:         for j = 0 à i − 1 do
5:             if T[j] > T[j + 1] then
6:                 échanger T[j] et T[j + 1]
7:             end if
8:         end for
9:     end for
10: end procedure
```

B Tri insertion

Algorithm 4 Tri par insertion

```
1: procedure tri_insertion(T)
2:     for i = 1 à taille de T − 1 do
3:         cl ← T[i]
4:         j ← i − 1
5:         while j ≥ 0 et T[j] > cl do
6:             T[j + 1] ← T[j]
7:             j ← j − 1
8:         end while
9:         T[j + 1] ← cl
10:     end for
11: end procedure
```

C Timsort

Algorithm 5 Timsort

```
1: procedure LongueurMinRun(n)
2:     r ← 0
3:     while n ≥ MIN_MERGE do
4:         r ← r ∨ (n&1)
5:         n ← n ≫ 1
6:     end while
7:     return n + r
8: end procedure
9: procedure TriInsertion(tableau, gauche, droite)
10:    for i ← gauche + 1 à droite do
11:        temp ← tableau[i]
12:        j ← i − 1
13:        while j ≥ gauche et tableau[j] > temp do
14:            tableau[j + 1] ← tableau[j]
15:            j ← j − 1
16:        end while
17:        tableau[j + 1] ← temp
18:    end for
19: end procedure
20: procedure Timsort(tableau, n)
21:    minRun ← LongueurMinRun(n)
22:    for i ← 0 à n − 1 avec un pas de minRun do
23:        TriInsertion(tableau, i, min(i+ MIN_MERGE −1, n − 1))
24:    end for
25:    for taille ← minRun à n avec un pas de 2 × taille do
26:        for gauche ← 0 à n − 1 avec un pas de 2 × taille do
27:            milieu ← gauche + taille − 1
28:            droite ← min(gauche + 2 × taille − 1, n − 1)
29:            if milieu < droite then
30:                Fusion(tableau, gauche, milieu, droite)
31:            end if
32:        end for
33:    end for
34: end procedure
```

D Tri fusion

Algorithm 6 Tri fusion

```
1: procedure TriFusion(tableau, gauche, droite)
2:     if gauche < droite then
3:         milieu ← gauche + (droite − gauche)/2
4:         TriFusion(tableau, gauche, milieu)
5:         TriFusion(tableau, milieu + 1, droite)
6:         Fusion(tableau, gauche, milieu, droite)
7:     end if
8: end procedure
```

Algorithm 7 Fusion

```
1: procedure Fusion(tableau, gauche, milieu, droite)
2:     n1 ← milieu − gauche + 1
3:     n2 ← droite − milieu
4:     Créer les tableaux G[1...n1] et D[1...n2]
5:     for i ← 0 à n1 − 1 do
6:         G[i] ← tableau[gauche + i]
7:     end for
8:     for j ← 0 à n2 − 1 do
9:         D[j] ← tableau[milieu + 1 + j]
10:    end for
11:    i ← 0, j ← 0, k ← gauche
12:    while i < n1 et j < n2 do
13:        if G[i] ≤ D[j] then
14:            tableau[k] ← G[i]
15:            i ← i + 1
16:        else
17:            tableau[k] ← D[j]
18:            j ← j + 1
19:        end if
20:        k ← k + 1
21:    end while
22:    while i < n1 do
23:        tableau[k] ← G[i]
24:        i ← i + 1
25:        k ← k + 1
26:    end while
27:    while j < n2 do
28:        tableau[k] ← D[j]
29:        j ← j + 1
30:        k ← k + 1
31:    end while
32: end procedure
```

E Tri par tas

Algorithm 8 Tri par tas (Heap Sort)

```
1: procedure tri_tas(T)
2:     n ← taille de T
3:     for i = ⌊n/2⌋ − 1 à 0 do
4:         heapify(T, n, i)
5:     end for
6:     for i = n − 1 à 1 do
7:         échanger T[0] et T[i]
8:         heapify(T, i, 0)
9:     end for
10: end procedure
11: procedure heapify(T, n, i)
12:     gauche ← 2i + 1
13:     droit ← 2i + 2
14:     plus_grand ← i
15:     if gauche < n et T[gauche] > T[plus_grand] then
16:         plus_grand ← gauche
17:     end if
18:     if droit < n et T[droit] > T[plus_grand] then
19:         plus_grand ← droit
20:     end if
21:     if plus_grand ≠ i then
22:         échanger T[i] et T[plus_grand]
23:         heapify(T, n, plus_grand)
24:     end if
25: end procedure
```

F Tri rapide

Algorithm 9 Tri rapide (Quick Sort)

```
1: procedure tri_rapide(T, premier, dernier)
2:     if premier < dernier then
3:         pivote ← partition(T, premier, dernier)
4:         tri_rapide(T, premier, pivote - 1)
5:         tri_rapide(T, pivote + 1, dernier)
6:     end if
7: end procedure
8: procedure partition(T, premier, dernier)
9:     pivote ← T[dernier]
10:    i ← prem − 1
11:    for j = premier à dernier − 1 do
12:        if T[j] ≤ pivote then
13:            i ← i + 1
14:            échanger T[i] et T[j]
15:        end if
16:    end for
17:    échanger T[i + 1] et T[dernier]
18:    retourner i + 1
19: end procedure
```

G Résultats sur les tableaux générés avec un pourcentage d'échange

G.1 Évolution du nombre d'accès au tableau

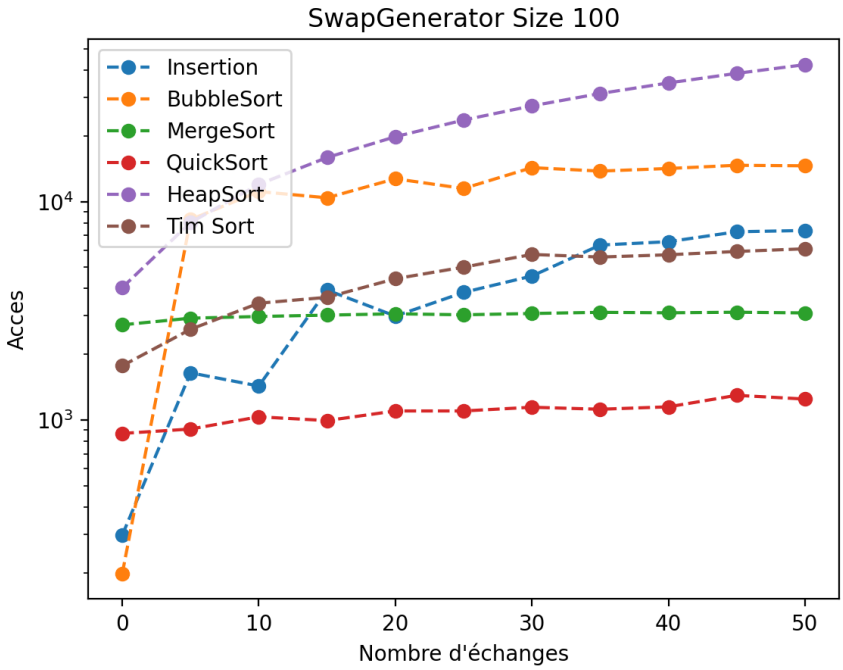

Figure 9 – Accès en fonction du % d'échange sur taille 100

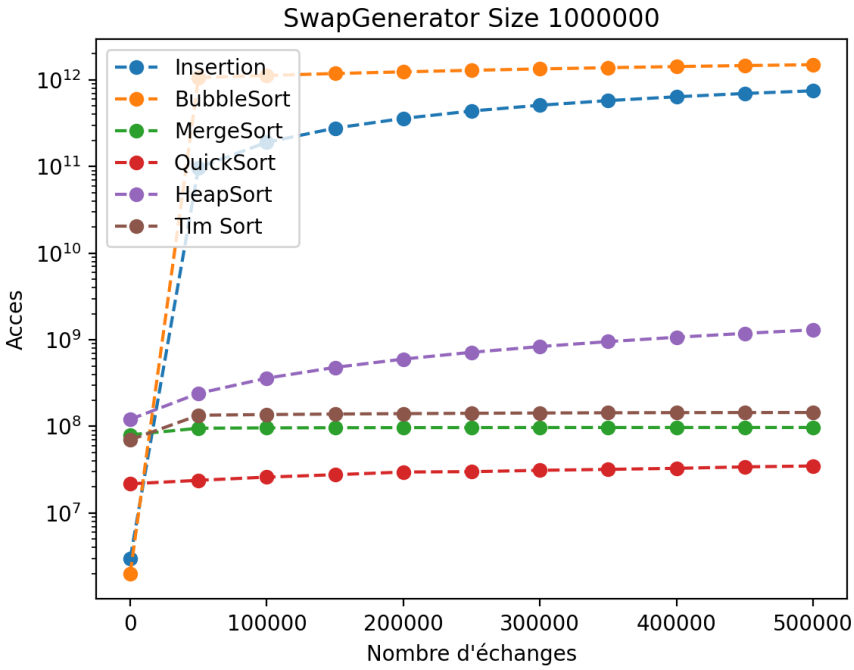

Figure 10 – Accès en fonction du % d'échange sur taille 1000000

G.2 Évolution du nombre de comparaison au tableau

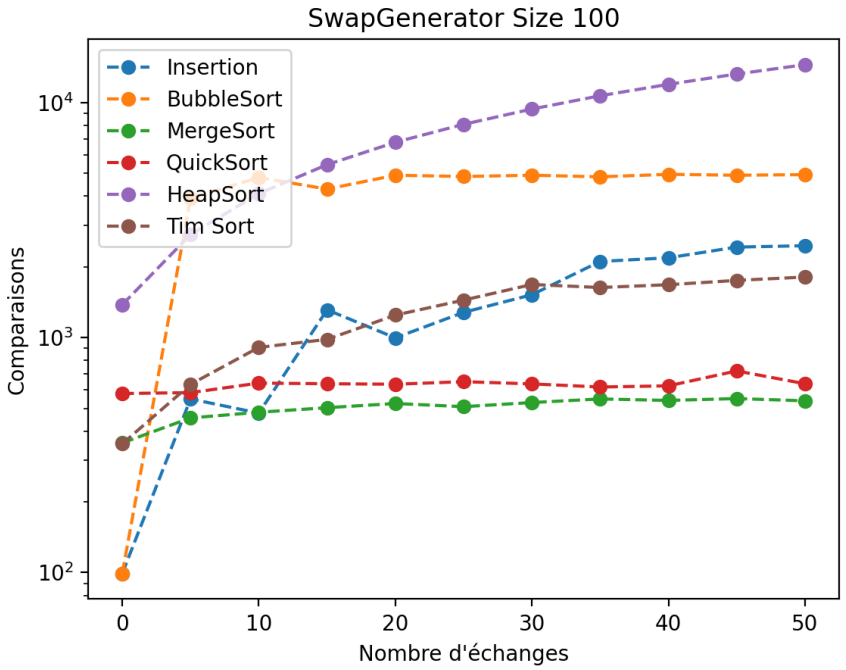

Figure 11 – Comparaison en fonction du % d'échange sur taille 100

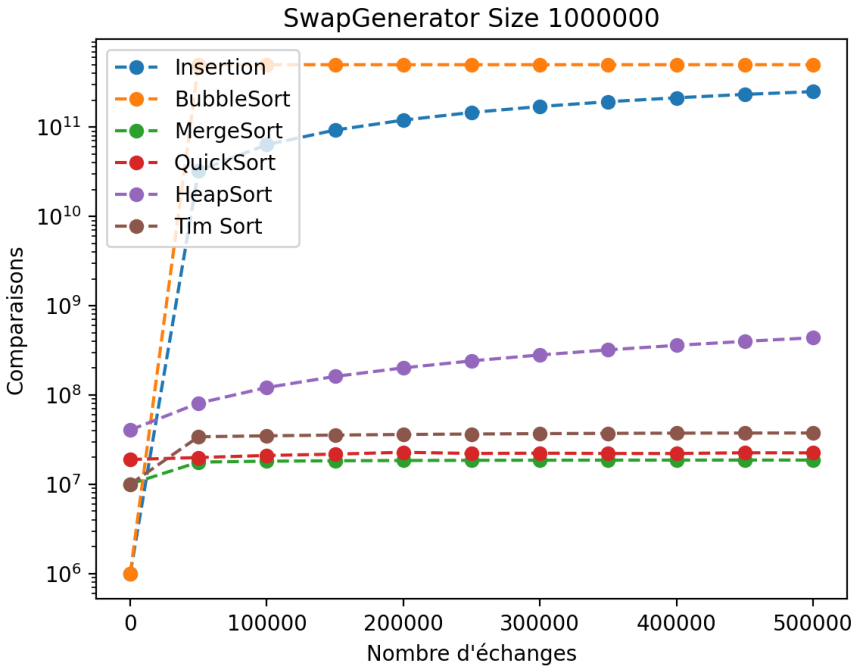

Figure 12 – Comparaison en fonction du % d'échange sur taille 1000000

G.3 Évolution du nombre d'échange effectué dans le tableau

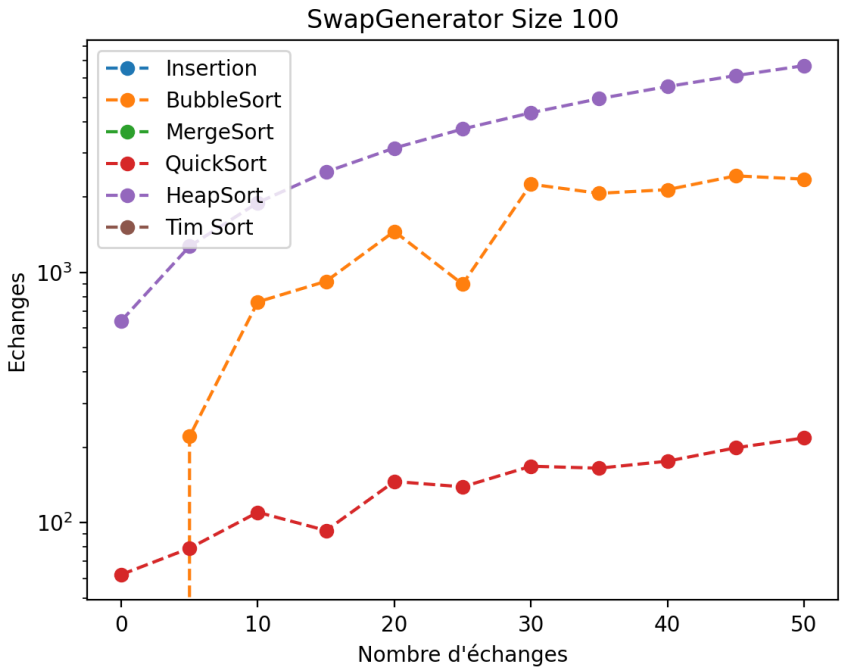

Figure 13 – Échange en fonction du % d'échange sur taille 100

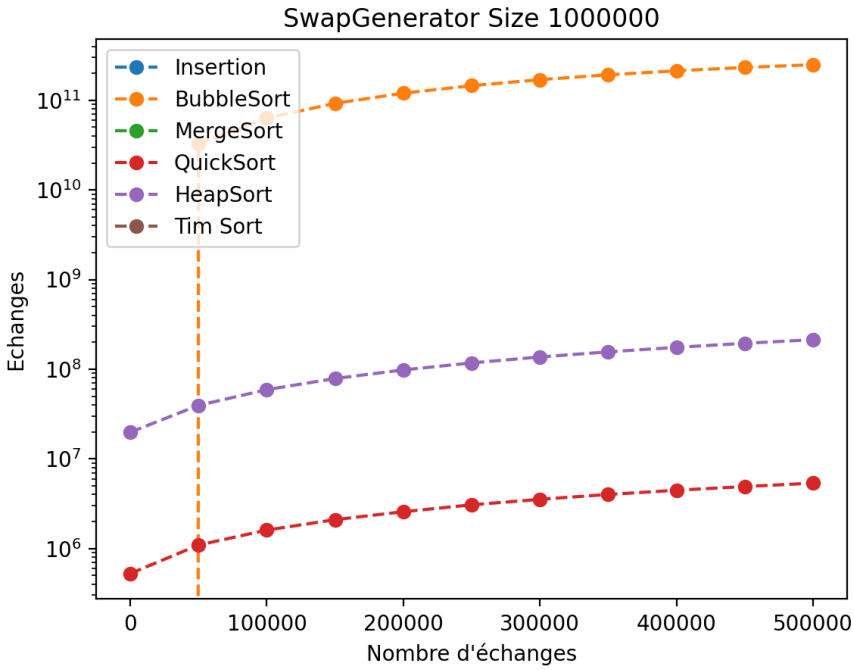

Figure 14 – Échange en fonction du % d'échange sur taille 1000000