Résumé

Ce rapport présente une étude des performances des algorithmes de tri en fonction de différents paramètres, tels que le niveau de désordre, et la taille des données,la distribution des valeurs. Nous débutons par une introduction générale du projet, puis nous abordons ses objectifs et sa problématique. Ensuite, nous décrivons les fonctionnalités implémentées, notamment la génération de jeux de données et l'implémentation de plusieurs algorithmes de tri afin de comparer leur efficacité. Nous analysons ensuite les performances des algorithmes à travers différentes expérimentations, en mesurant le temps d'exécution, le nombre de comparaisons effectuées et d'autres critères pertinents. Enfin, nous concluons en résumant les résultats obtenus et en proposant des améliorations possibles pour optimiser ces algorithmes.

Sommaire

1 Introduction

Le tri des données est une opération essentielle en informatique, utilisée dans des domaines variés comme les bases de données et l'intelligence artificielle. Face à la diversité des algorithmes de tri existants, il est crucial d'évaluer leurs performances dans différentes conditions. En effet, leur efficacité ne dépend pas uniquement de leur complexité théorique, mais aussi de la structure des données et des critères spécifiques influençant leur exécution.

Ce projet propose une étude approfondie des algorithmes de tri et de leurs performances dans différents contextes. À travers une approche expérimentale, nous chercherons à analyser les facteurs pouvant impacter leur efficacité et à comparer leurs comportements sur divers ensembles de données.

Présentation du plan du rapport

Ce rapport est structuré de la manière suivante :

- **Présentation des objectifs du projet** : Explication des buts de l'étude et des attentes.
- **Méthodologie** : Détail des étapes de mise en œuvre et des choix techniques.
- **Résultats expérimentaux et analyse** : Présentation des résultats obtenus et leur interprétation.
- **Conclusion** : Résumé des résultats et perspectives d'amélioration.

2 Objectifs du projet

2.1 Problématique du projet

Les algorithmes de tri varient considérablement en fonction des données qu'ils traitent. Des éléments tels que la distribution des valeurs, la taille de la liste, le niveau de désordre et l'entropie de Shannon jouent un rôle crucial dans l'efficacité de ces algorithmes. Ainsi, il devient essentiel de comprendre comment ces facteurs influencent des critères comme le temps d'exécution, le nombre de comparaisons et l'accès à la mémoire.

Comment la taille des données, le niveau de désordre et l'entropie de Shannon influencent-ils les performances des algorithmes de tri en termes de temps d'exécution, de nombre de comparaisons et d'accès en mémoire ?

2.2 Description des points-clés et des grandes étapes

Pour analyser les performances des algorithmes de tri en fonction de la distribution des données, de l'entropie, de la taille des données et du niveau de désordre, nous avons structuré notre projet en plusieurs étapes clés :

- Implémentation de générateurs de données ordonnées
- Implémentation de cinq fonctions de désordre
- Implémentation de cinq algorithmes de tri
- Analyse des performances

2.3 Description de travaux existants sur le même sujet

De nombreux travaux ont analysé les performances des algorithmes de tri en fonction de divers critères :

- **Analyse mathématique et algorithmique** : Donald Knuth, dans son ouvrage de référence *The Art of Computer Programming*, propose une étude approfondie des algorithmes de tri sous un prisme mathématique et algorithmique. Il examine leur complexité, leur efficacité en fonction de la structure des données et leur comportement dans divers scénarios. Son travail constitue une base incontournable pour comparer les performances des algorithmes et mesurer l'impact du désordre des données.
- **Optimisation en fonction de l'entropie** : Des études, comme celles de Brodnik et al., ont exploité l'entropie de Shannon pour optimiser les algorithmes de tri, en analysant leur efficacité face à des ensembles de données présentant différents niveaux de désordre.
- **Mesure du désordre des séquences** : La distance de Kendall a été utilisée dans plusieurs recherches pour quantifier le niveau de désordre d'un ensemble de données, permettant ainsi une meilleure évaluation des performances des algorithmes de tri en fonction de la structure des séquences à trier.

3 Fonctionnalités implémentées

3.1 Description des fonctionnalités

- **Génération de jeux de données variées :**
  - Création de jeux de données basés sur différentes distributions statistiques (uniforme, gaussienne, normale, exponentielle).
- **Application de fonctions de désordre :**
  - Application de fonctions spécifiques pour modifier le niveau de désordre des jeux de données générés.
  - Quantification de l'entropie de Shannon pour mesurer le degré de désorganisation des données.
- **Tri des données avec différents algorithmes :**
  - Implémentation de cinq algorithmes de tri aux comportements variés (tri à bulles, tri par sélection, tri fusion, tri par tas, et tri par seaux).
  - Comparaison des performances des algorithmes sur des jeux de données avec divers niveaux de désordre.
- **Faire des expérimentations :**
  - *Expérimentations sur différents générateurs de données :*
    - Génération de différents jeux de données selon des distributions variées.
    - Application des fonctions de désordre pour simuler des niveaux variés de désorganisation des données.
    - Test des algorithmes de tri sur ces données désordonnées.
  - *Analyse des performances :*
    - Mesure des performances des algorithmes à l'aide de trois indicateurs principaux:
      - Le temps d'exécution.
      - Le nombre de comparaisons.
      - Le nombre d'accès mémoire.
    - Enregistrement des résultats dans des tableaux comparatifs permettant d'analyser et de comparer directement les algorithmes sous différents niveaux de désordre des données.
  - *Visualisation des résultats :*
    - Génération de graphiques illustrant la variation des performances des algorithmes selon différents niveaux de désordre.
    - Création d'animations pour visualiser en temps réel le processus de tri et pour faciliter la compréhension de l'évolution des données pendant le tri.

]

4 Éléments techniques

4.1 Descriptions des algorithmes

Tri fusion (Merge Sort) :
Le tri fusion est un algorithme de tri basé sur la technique de **diviser pour régner**. Il divise récursivement le tableau en deux sous-tableaux, les trie séparément, puis fusionne ces sous-tableaux triés pour former un tableau trié final. Cette approche permet d'obtenir une performance stable, même dans le pire des cas.

Complexité :

- Meilleur cas et cas moyen : $O(n \log n)$
- Pire cas : $O(n \log n)$

Tri par tas (Heap Sort) :
Le tri par tas transforme le tableau en un tas binaire, une structure arborescente où chaque parent est supérieur (ou inférieur) à ses enfants. Il extrait successivement l'élément maximal (ou minimal) et réorganise le tas jusqu'à obtenir un tableau trié.

Complexité :

- Meilleur cas et cas moyen : $O(n \log n)$
- Pire cas : $O(n \log n)$

Cette méthode assure une complexité stable de $O(n \log n)$, indépendamment de l'ordre initial des données.

4.2 Descriptions des structures de données

Dans notre projet, nous utilisons principalement des tableaux comme structure de données pour stocker et manipuler les jeux de données générés. Un tableau est une structure de données linéaire où les éléments sont stockés de manière contiguë en mémoire et accessibles via un index.

Les tableaux sont la structure de données idéale pour notre projet car :

- **Accès rapide aux éléments :** L'accès à un élément via son index se fait en temps constant, $O(1)$.
- **Simplicité d'utilisation :** Ils sont faciles à manipuler et compatibles avec tous les algorithmes de tri.
- **Efficacité en mémoire :** Les tableaux sont stockés de manière continue en mémoire, ce qui réduit le gaspillage d'espace et facilite l'accès aux données.

4.3 Description des données

Les données numériques utilisées pour tester les algorithmes de tri sont générées sous forme de nombres selon différentes distributions statistiques et manipulées par des fonctions de désordre.

Distributions des données :

- **Uniforme :** Les valeurs sont réparties de manière égale dans une plage donnée.
- **gaussienne :** Les valeurs suivent une distribution en cloche, centrées autour de la moyenne.
- **Exponentielle :** Les valeurs sont générées selon une loi exponentielle, avec une probabilité décroissante pour les grandes valeurs.
- **normale :** Les valeurs sont générées sans suivre une distribution spécifique.

Fonctions de désordre appliquées :

- **Désordre par taux :** Un pourcentage défini d'éléments de la liste est échangé de manière aléatoire.

Algorithm 1 Apply_Disorder(Data, Disorder_Level)

```
Input: Data, Disorder_Level
Taille ← longueur(Data)
Nb_Swaps ← ⌊Disorder_Level × Taille / 2⌋
Tab ← liste des indices de 0 à Taille-1
for i = 1 to Nb_Swaps do
    if Longueur(Tab) < 2 then
        break
    end if
    (i, j) ← indices aléatoires distincts de Tab
    Échanger Data[i] et Data[j]
    Supprimer i et j de Tab
end for
Return Data
```

- **Désordre sur les derniers éléments :** Seuls les derniers éléments de la liste sont perturbés.

Algorithm 2 Apply_Disorder_Last(Data, Nb_Elements)

```
1:  Input: Data, Nb_Elements
2:  Taille ← longueur(Data)
3:  if Nb_Elements > Taille then
4:      Error: Nombre d'éléments trop grand
5:      Return
6:  end if
7:  Start_Index ← Taille - Nb_Elements
8:  Sub_Data ← Data[Start_Index:]
9:  Apply_Disorder(Sub_Data, 1)
10: Remplacer Data[Start_Index:] par Sub_Data
11: Return Data
```

- **Désordre par bloc :** La liste est divisée en plusieurs blocs, et chaque bloc est mélangé indépendamment8

Algorithm 3 Apply_Block_Sorted(Data, Num_Blocks)

Input: Data, Num_Blocks
Block_Size ← longueur(Data) // Num_Blocks
Reste ← longueur(Data) % Num_Blocks
Blocks ← []
Début ← 0
for $i = 0$ to Num_Blocks - 1 **do**
    Fin ← Début + Block_Size
    if $i < Reste$ **then**
        Fin ← Fin + 1
    end if
    Ajouter Data[Début:Fin] à Blocks
    Début ← Fin
end for
for chaque Block dans Blocks **do**
    Trier Block
end for
Mélanger(Blocks)
Résultat ← Concaténation de Blocks
Return Résultat

- **Désordre dans l'ordre inverse :** La liste est inversée, créant un désordre complet.

Algorithm 4 Apply_Inverse_Sorted(Data)

Input: Data
Return Data trié en ordre décroissant

- **Désordre selon l'entropie de Shannon :** .

Algorithm 5 Generate_List_Shannon(N, K, H_Target)

Input: N, K, H_Target
repeat
    Probs ← Générer une distribution aléatoire de K symboles
    Trier Probs en ordre décroissant
    Entropie ← $-\sum(p \times log_2(p))$ pour chaque $p > 0$
until |Entropie - H_Target| ≤ 0.01
Symboles ← Liste de 1 à K
Data ← Sélectionner N éléments parmi Symboles avec distribution Probs
Return Data, Entropie

5 Architecture du projet

5.1 Description des paquetages non standards utilisés

Dans ce projet, nous avons utilisé plusieurs bibliothèques non standards afin de faciliter la génération des jeux de données, l'implémentation des algorithmes de tri et l'analyse des performances.

- **NumPy** : utilisée principalement pour la génération des jeux de données en suivant différentes distributions statistiques (uniforme, normale, gaussienne, exponentielle) et pour la manipulation efficace des tableaux.
- **Matplotlib** : utilisée pour afficher des graphiques permettant de visualiser l'exécution des algorithmes de tri et de comparer leurs performances.
- **Pandas** : utilisée pour stocker, organiser et analyser les résultats des expérimentations sous forme de tableaux de données.

5.2 Diagrammes des modules et des classes

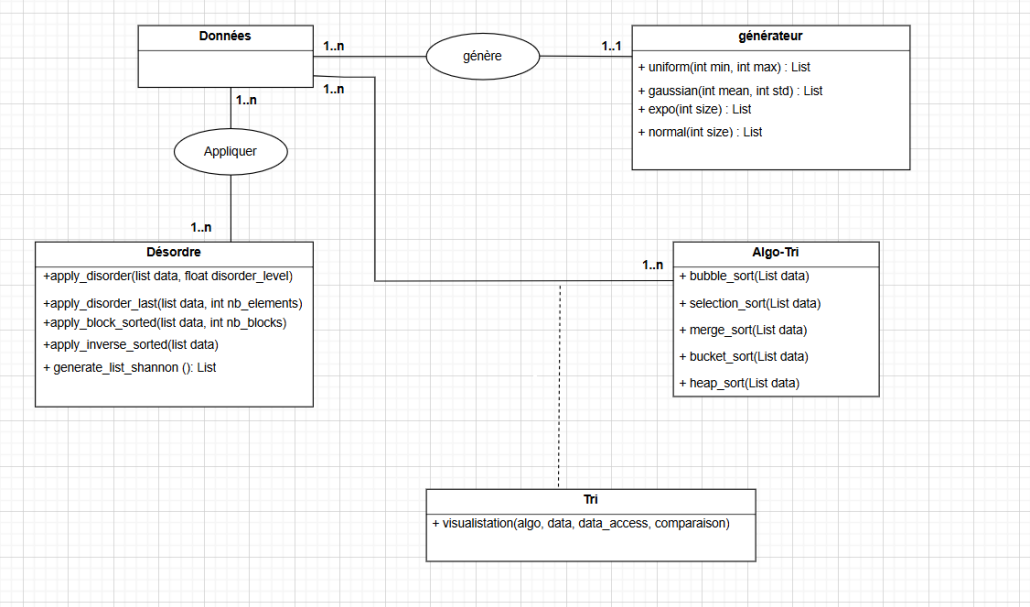

Figure 1: Diagramme de classe du projet

Le diagramme ci-dessus illustre l'architecture du projet en mettant en évidence les principales classes et leurs relations. Il montre comment les données sont générées, modifiées par les fonctions de désordre, triées à l'aide des algorithmes implémentés et visualisées pour analyser leurs performances.

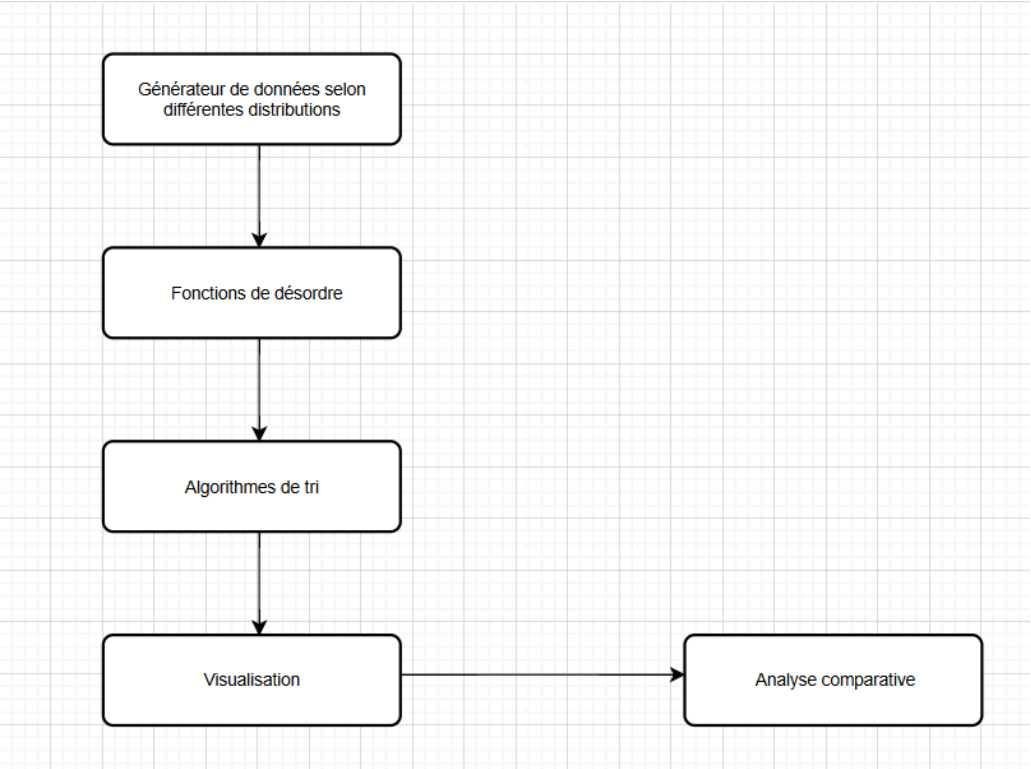

Figure 2: Diagramme des modules du projet

Le diagramme ci-dessus présente l'architecture du projet en mettant en évidence les différents modules et leurs interactions. Il montre comment chaque module, responsable d'une tâche spécifique, communique avec les autres pour exécuter le flux global du projet.

5.3 Chaînes de traitement (comment les classes interagissent et pourquoi)

- **Classe Générateur de données** : elle crée des ensembles de données ordonnées selon des distributions spécifiques (uniforme, normale, gaussienne, exponentielle). Ces données triées sont ensuite transmises à la classe de gestion du désordre.
- **Désordre** : cette classe prend cette liste de données générées, puis y introduit du désordre (par exemple, en les mélangeant aléatoirement). Les données désordonnées sont ensuite prêtes à être envoyées à la classe Algorithmes de Tri.
- **Classe Algorithmes de Tri** : une fois les données mélangées, la classe Algorithmes de Tri prend cette liste désordonnée et applique les différents algorithmes de tri. Elle renvoie ensuite les résultats triés.

Pourquoi ces interactions ?
Les interactions entre ces classes assurent une séparation claire des responsabilités, ce qui facilite l'organisation du code et sa maintenabilité.

6 Expérimentations

6.1 Analyse en fonction de la taille

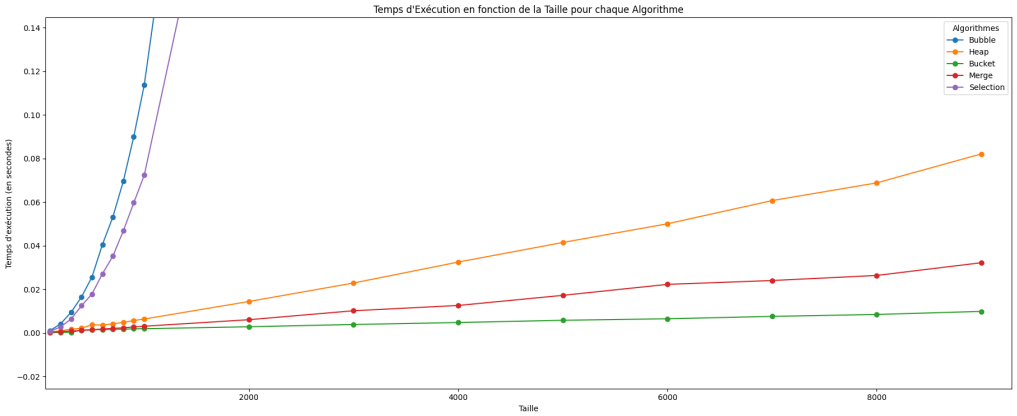

Figure 3: temps d 'éxecution en fonction de la taille pour chaque algorithme

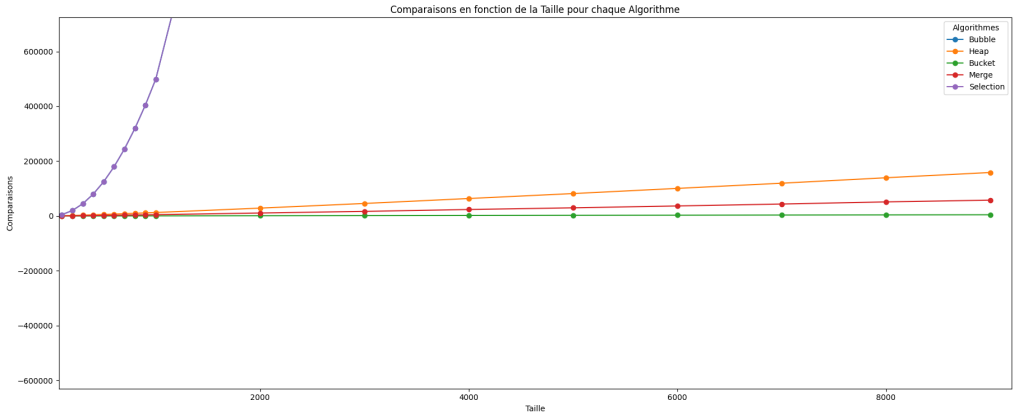

Figure 4: comparaison en fonction de la taille pour chaque algorithme

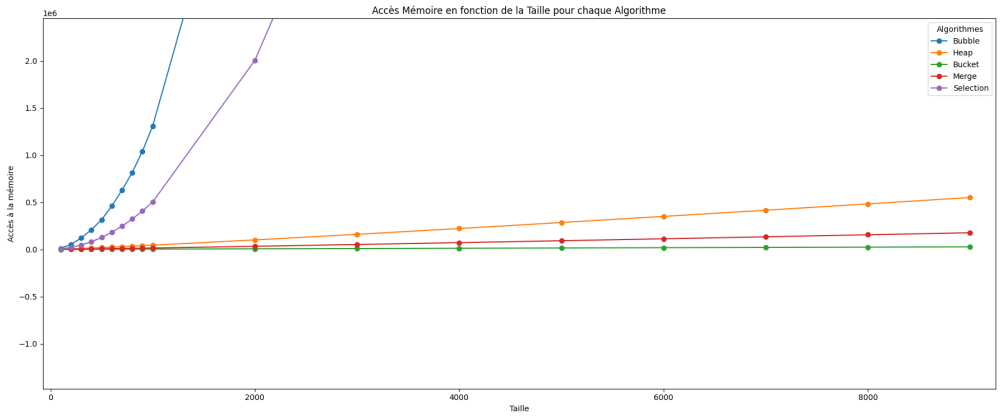

Figure 5: Acces Memoire en fonction de la taille pour chaque algorithme

- **Temps d'exécution en fonction – Figure 3**
  - **Bubble Sort et Selection Sort**

    Ces deux algorithmes montrent une augmentation très rapide du temps d'exécution lorsque la taille des données augmente.

    Dès que la taille dépasse environ 2000 éléments, on observe une explosion du temps de traitement, les rendant inadaptés pour des entrées plus grandes.

    Entre les deux, **Bubble Sort** est légèrement plus lent que **Selection Sort**, ce qui est conforme aux observations classiques.
  - **Heap Sort**

    La progression est bien plus modérée que pour **Bubble** et **Selection**.

    Même pour des tailles élevées, l'augmentation reste **progressive et linéaire**, ce qui le rend plus efficace sur des ensembles plus volumineux.
  - **Merge Sort**

    Son temps d'exécution est également assez **linéaire**, mais légèrement inférieur à **Heap Sort**.

    La croissance est **régulière et maîtrisée**, ce qui montre qu'il gère bien l'augmentation de la taille des données.

    Il reste performant même pour de très grandes tailles (plusieurs milliers d'éléments).
  - **Bucket Sort**

    C'est l'algorithme le plus **rapide** pour toutes les tailles observées.

    Sa courbe est **presque plate**, ce qui indique qu'il gère parfaitement l'augmentation de la taille des données.

    Son efficacité reste **constante** même pour les grandes tailles, contrairement aux autres méthodes.

Nombre de comparaisons – Figure 4

- **Selection Sort et Bubble Sort**

  Ces deux algorithmes montrent une forte augmentation du nombre de comparaisons lorsque la taille des données augmente.

  À partir de 2000 éléments, le nombre de comparaisons devient très élevé et continue d'exploser pour des tailles plus grandes.

  Bubble Sort effectue légèrement plus de comparaisons que Selection Sort, ce qui confirme son inefficacité relative.

- **Heap Sort** La croissance du nombre de comparaisons est plus modérée que pour Bubble et Selection.

  Même pour des tailles importantes, l'augmentation reste progressive, sans montée brutale.

  Cela montre qu'il reste beaucoup plus efficace que Bubble et Selection Sort pour les grandes tailles.

- **Merge Sort** Le nombre de comparaisons suit une progression régulière et bien contrôlée.

  Il est légèrement inférieur à Heap Sort pour toutes les tailles observées.

  Il reste performant même pour des tailles très grandes (plusieurs milliers d'éléments).

- **Bucket Sort**
  C'est l'algorithme le plus efficace en nombre de comparaisons.

  Sa courbe est presque plate, ce qui montre que l'augmentation de la taille des données n'a quasiment aucun impact sur le nombre de comparaisons.

  Contrairement aux autres méthodes, il maintient une efficacité constante, même sur les grandes tailles.

6.2 Analyse en fonction de taux de désordre

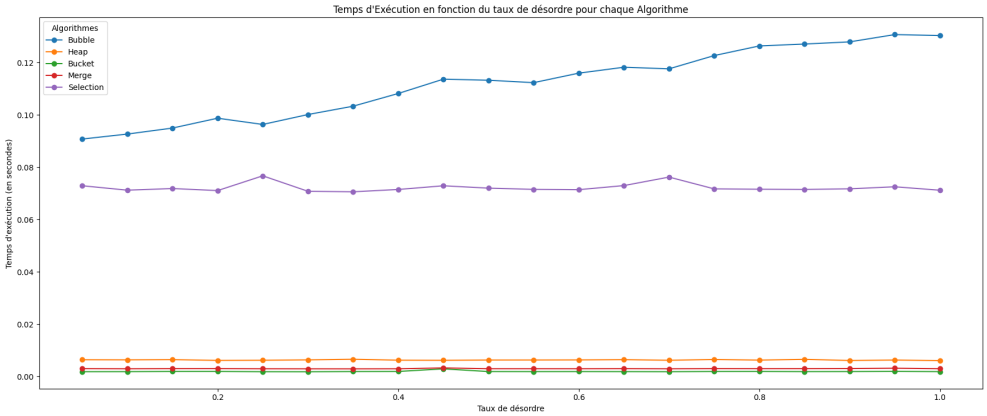

Figure 6: temps d'exécution en fonction du taux de désordre pour chaque algorithme

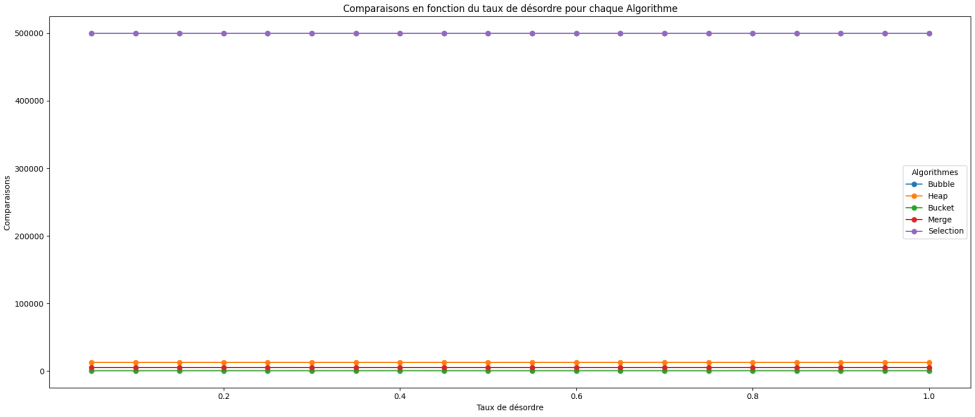

Figure 7: Comparaison en fonction du taux de désordre pour chaque algorithme

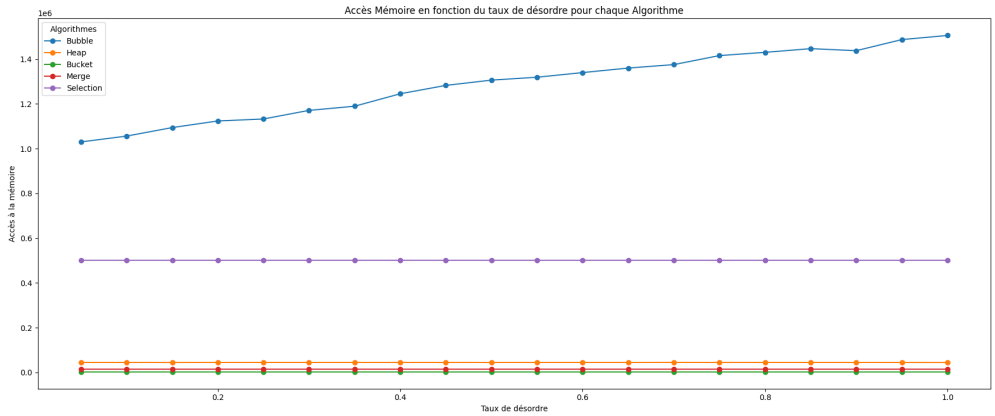

Figure 8: acces mémoire en fonction du taux de désordre pour chaque algorithme

- **Temps d'exécution – Figure 6**
  - **1. Faible désordre (0 - 0.3)**
    - **Bubble Sort** et **Selection Sort** ont un temps d'exécution relativement élevé dès le départ.
    - **Heap Sort**, **Merge Sort** et **Bucket Sort** sont bien plus rapides et stables.
  - **2. Désordre moyen (0.3 - 0.7)**
    - **Bubble Sort** et **Selection Sort** continuent d'avoir un temps d'exécution élevé et en augmentation.
    - **Heap Sort** et **Merge Sort** restent performants avec peu de variations.
    - **Bucket Sort** demeure le plus rapide, avec un temps d'exécution minimal.
  - **3. Fort désordre (0.7 - 1.0)**
    - **Bubble Sort** et **Selection Sort** ont le temps d'exécution le plus long, restant les moins efficaces.
    - **Heap Sort** et **Merge Sort** maintiennent un temps stable et compétitif.
    - **Bucket Sort** reste le plus performant, même avec un fort désordre.
  - En résumé, **Bubble Sort** et **Selection Sort** sont inefficaces quelle que soit la structure des données. **Heap Sort**, **Merge Sort** et surtout **Bucket Sort** offrent de meilleures performances en exécution.

Nombre de comparaisons – Figure 7

- **Bubble Sort et Selection Sort**
  - Leur courbe est plate, ce qui signifie que le nombre de comparaisons reste constant quel que soit le taux de désordre.
  - Ces algorithmes ne profitent pas d'un faible désordre : qu'un tableau soit presque trié ou totalement aléatoire, ils effectuent toujours un grand nombre de comparaisons.
  - Ils ne sont pas sensibles au désordre et sont peu efficaces en général.
- **Heap Sort**
  - Le nombre de comparaisons augmente légèrement avec le taux de désordre.

    * Heap Sort est modérément sensible au désordre, mais son efficacité reste assez stable.

  – **Merge Sort**

    * La courbe est presque plate, ce qui montre que le nombre de comparaisons est indépendant du taux de désordre.
    * Merge Sort est très fiable quel que soit l'état initial du tableau.

  – **Bucket Sort**

    * Son nombre de comparaisons est très bas et ne varie presque pas avec le taux de désordre.

- **Accès mémoire – Figure 8**

  – **Bubble Sort**

    * Le nombre d'accès à la mémoire est le plus élevé et augmente avec le taux de désordre.
    * Cet algorithme effectue de nombreuses opérations inutiles, ce qui explique sa faible efficacité.

  – **Selection Sort**

    * Son nombre d'accès à la mémoire reste constant quel que soit le taux de désordre.
    * L'algorithme suit toujours le même schéma de sélection, indépendamment de l'ordre initial des éléments.

  – **Heap Sort**

    * Il présente un faible nombre d'accès mémoire, stable sur toute la plage du taux de désordre.
    * Son fonctionnement basé sur un tas optimise les écritures et lectures mémoire.

  – **Merge Sort**

    * Le nombre d'accès mémoire est relativement faible et ne varie presque pas avec le taux de désordre.
    * Sa structure récursive divise efficacement les données, minimisant les accès mémoire redondants.

  – **Bucket Sort**

    * Il a le plus faible nombre d'accès mémoire.
    * L'algorithme repose principalement sur des insertions directes dans les seaux, limitant les lectures et écritures inutiles.

Bubble Sort est le moins performant en raison de ses nombreux accès mémoire, tandis que Bucket Sort est le plus efficace. Heap Sort et Merge Sort offrent un bon compromis avec une stabilité remarquable.

6.3 Analyse en fonction de l'entropie de Shannon

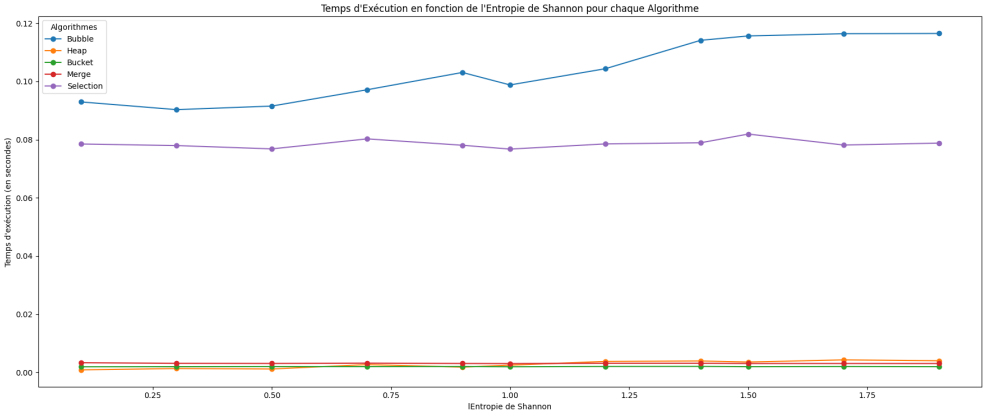

Figure 9: Temps d'exécution en fonction de l'entropie de Shannon pour chaque algorithme

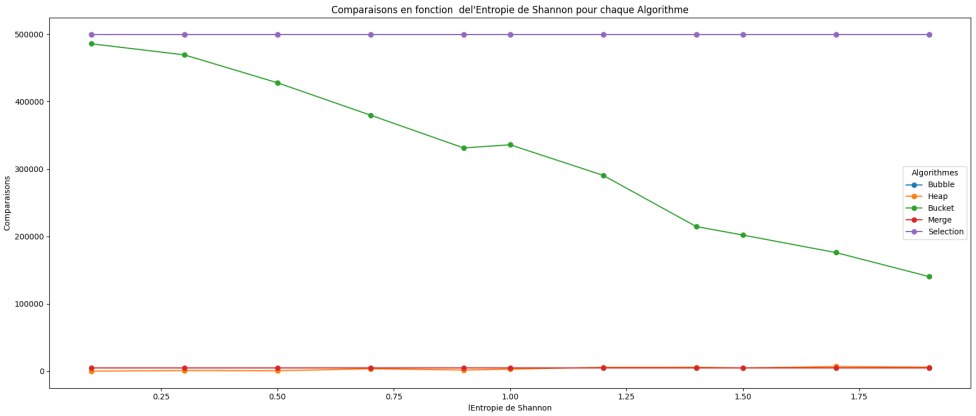

Figure 10: Nombre de comparaisons en fonction de l'entropie de Shannon pour chaque algorithme

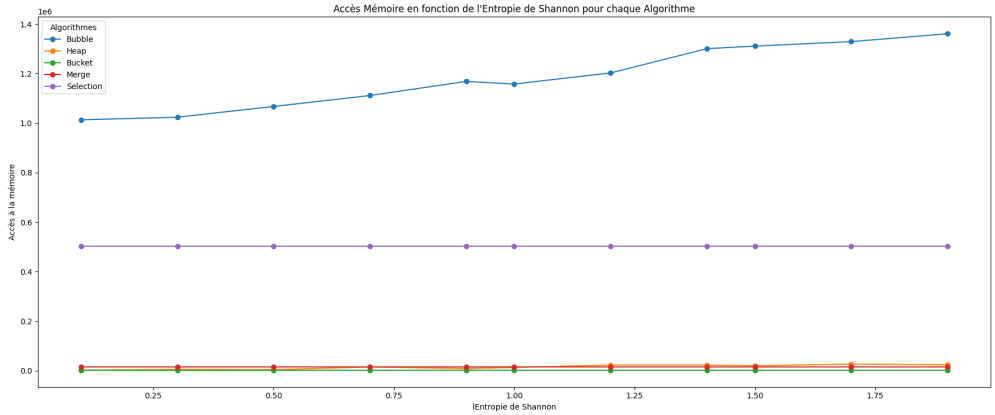

Figure 11: Accès mémoire en fonction de l'entropie de Shannon pour chaque algorithme

- **Temps d'exécution — Figure 9**
  - **Bubble Sort** :
    - Temps d'exécution le plus élevé parmi tous les algorithmes.
    - Son temps augmente avec l'entropie, ce qui indique une dépendance forte à l'ordre initial des données.
  - **Selection Sort** :
    - Deuxième algorithme le plus lent.
    - Son temps d'exécution reste relativement stable, ce qui est logique puisqu'il effectue toujours le même nombre de comparaisons, indépendamment de l'entropie.
  - **Merge Sort** et **Bucket Sort** :
    - Ces deux algorithmes sont nettement plus rapides que Bubble et Selection Sort.
    - Leurs courbes sont quasiment superposées, ce qui suggère qu'ils ont des performances similaires et peu impactées par l'entropie.
  - **Heap Sort**
    Contrairement aux autres algorithmes, Heap Sort montre des variations irrégulières en fonction de l'entropie.
- **Conclusion** :
  - Bubble Sort est clairement le moins efficace, surtout lorsque l'entropie augmente.
  - Selection Sort est également sous-optimal, bien que plus stable.
  - Heap Sort, Merge Sort et Bucket Sort sont les plus performants, et leur efficacité est peu affectée par l'entropie des données.

Nombre de comparaisons — Figure 10

- **Bubble Sort et Selection Sort**
  - Ces deux algorithmes effectuent un nombre de comparaisons **constant** quelle que soit l'entropie.

- Cela signifie qu'ils ne sont **pas sensibles au désordre des données** : qu'elles soient presque triées ou totalement aléatoires, le nombre de comparaisons reste le même.

- **Bucket Sort**

  - Le nombre de comparaisons **diminue fortement** à mesure que l'entropie augmente.
  - Cela indique que Bucket Sort **est très efficace lorsque l'entropie est elevée** (les données sont presque triées).
  - Cependant, lorsque l'entropie est élevée, son efficacité baisse, bien qu'il reste meilleur que Bubble et Selection Sort.

- **Merge Sort**

  - L'algorithme garde un nombre de comparaisons **quasi stable** quelle que soit l'entropie.
  - Cela signifie qu'il **n'est pas affecté par le degré de désordre des données**.
  - Il offre donc des performances homogènes, que les données soient déjà bien organisées ou totalement aléatoires.

- **Heap Sort**

  - Contrairement aux autres algorithmes, **Heap Sort montre des variations irrégulières** en fonction de l'entropie.
  - Son nombre de comparaisons fluctue, indiquant qu'il peut être influencé par le niveau de désordre, mais de manière **moins prévisible**.

- **conclusions**

  - **Bubble et Selection Sort sont insensibles à l'entropie** : Ils effectuent toujours le même nombre de comparaisons.
  - **Bucket Sort est le plus influencé par l'entropie** : Il est **très performant pour les faibles entropies** et voit son efficacité diminuer avec l'augmentation du désordre.
  - **Merge Sort reste stable** : Son comportement ne change pas en fonction de l'entropie.
  - **Heap Sort est imprévisible** : Il varie en fonction de l'entropie, mais sans tendance claire.

Accès mémoire — Figure 11

- **Bubble Sort**

  - Présente un nombre d'accès mémoire nettement plus élevé que les autres algorithmes.

- Augmente progressivement avec l'entropie, ce qui indique qu'il est particulièrement inefficace lorsque le désordre croît.
- Son comportement confirme sa complexité élevée en termes d'échanges de données.

- **Selection Sort**
  - Son nombre d'accès mémoire est constant, quelle que soit l'entropie.
  - Cela s'explique par son fonctionnement : il effectue toujours le même nombre d'opérations de sélection et d'échange.

- **Heap Sort, Merge Sort et Bucket Sort**
  - Ils ont des accès mémoire très faibles et quasi constants, quel que soit le niveau de désordre.
  - Cela confirme leur efficacité en termes de gestion de la mémoire, indépendamment de l'entropie.

- **Conclusion**
  - Bubble Sort est le plus inefficace en termes d'accès mémoire, surtout lorsque l'entropie est élevée.
  - Selection Sort est stable mais toujours sous-optimal en mémoire, indépendamment de l'entropie.

7 Conclusion

7.1 Récapitulatif de la problématique et de la réalisation

Cette étude a cherché à analyser les performances des algorithmes de tri en fonction de certaines caractéristiques des données. La problématique initiale visait à comprendre comment le niveau de désordre, la taille des données et l'entropie de Shannon influencent les critères de performance des algorithmes de tri, à savoir : le temps d'exécution, le nombre de comparaisons effectuées et les accès mémoire.

Pour répondre à cette problématique, une démarche expérimentale a été mise en place, structurée autour des étapes suivantes :

- Génération de jeux de données avec des tailles variables ;
- Application de différents niveaux de désordre sur les listes (listes triées, partiellement triées, ou totalement mélangées) ;
- Génération de listes avec des taux d'entropie de Shannon ;
- Implémentation et exécution de cinq algorithmes de tri : Bubble Sort, selection Sort, Heap Sort, Merge Sort, et Bucket Sort ;
- Évaluation des performances de chaque algorithme selon trois indicateurs : temps d'exécution, nombre de comparaisons, et nombre d'accès mémoire ;
- Visualisation et interprétation des résultats à l'aide de graphiques, permettant de comparer le comportement des algorithmes.

Cette approche a permis d'observer de manière concrète l'influence des facteurs structurels des données sur l'efficacité des algorithmes de tri, et d'identifier les contextes dans lesquels chaque algorithme est le plus adapté.

7.2 Récapitulatif des résultats

Algorithme	**Temps d'exécution**	**Nombre de comparaisons**	**Accès mémoire**
Bubble Sort	Le plus lent, temps d'exécution croît de manière quadratique avec la taille, le désordre et l'entropie	Très élevé, croît quadratiquement avec la taille des données, le désordre et l'entropie	Très élevé, augmente exponentiellement avec la taille données, le désordre et l'entropie
Selection Sort	Temps d'exécution lent, croît quadratiquement avec la taille des données	Très élevé, croît quadratiquement avec la taille des données	Relativement élevé, mais légèrement plus rapide que Bubble Sort
Merge Sort	Temps d'exécution stable et rapide, peu affecté par la taille ou l'entropie	Nombre de comparaisons faible et stable, peu influencé par le désordre	Relativement faible, stable même avec des tailles de données élevées
Heap Sort	Temps d'exécution très stable, efficace même avec de grandes tailles de données	Nombre de comparaisons faible, presque linéaire avec la taille des données, peu influencé par l'entropie	Très faible, très efficace, peu affecté par le désordre et l'entropie
Bucket Sort	Très bon temps d'exécution, surtout avec un fort désordre et une grande entropie	Nombre de comparaisons faible, très stable et indépendant du désordre	Très faible, reste stable même avec un grand nombre de données

Table 1: Synthèse des résultats selon les critères : temps, comparaisons, mémoire.

7.3 Propositions d'améliorations

Bien que l'expérimentation ait permis de comparer efficacement plusieurs algorithmes de tri sur différentes configurations de listes d'entiers, certaines améliorations pourraient enrichir cette étude.

- **Ajouter d'autres algorithmes** comme Radix ou Tim Sort pour compléter la comparaison.
- **Tester d'autres types de données** comme les nombres flottants ou les chaînes de caractères, pour analyser si le type de données influence les performances des algorithmes.
- **Étendre l'expérimentation** à des tailles de données beaucoup plus grandes pour mieux observer les limites de chaque algorithme.