Rapport de projet sur l'Analyse des algorithmes de tris

mis en page par
EMAM Mohamed El Mamy
DIALLO Boubacar Sadio
DIARE Youssouf 22008756
OLANGASSICKA Franck 22112035
L3 info,
Groupe 2
Université de Caen Normandie
2024–2025

28 mars 2025

Chargé de TP : **ZANUTTINI Bruno**

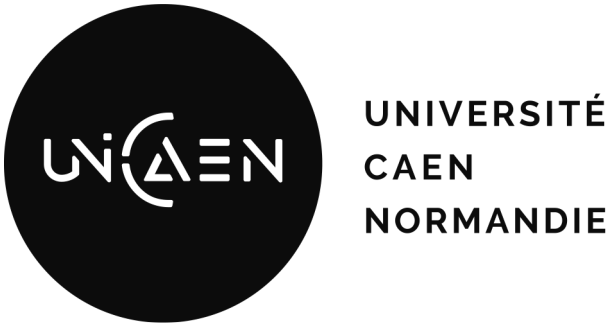


Table des matières

1 Introduction **3**

2 Objectifs du projet **3**
2.1 Problématique du projet . . . 3
2.2 Description des points-clés et des grandes étapes . . . 3

3 Fonctionnalités implémentées **4**
3.1 Description des fonctionnalités . . . 4
3.1.1 Graphique . . . 4
3.2 Technique . . . 5
3.3 Organisation du projet . . . 5
3.3.1 Contributions Individuelles par tache : . . . 5
3.4 Package `controler` . . . 6
3.5 Package **model** . . . 6
3.5.1 model.algo . . . 6
3.5.2 model.generateur . . . 6
3.6 Packages **experience** . . . 7
3.6.1 Les classes et fichiers principaux . . . 7
3.7 Package **vue** . . . 7
3.7.1 Classe TriGui . . . 8
3.7.2 Classe VueTri . . . 8
3.7.3 Classe Demo . . . 8

4 Éléments techniques **8**
4.1 Description des algorithmes de tri non triviaux . . . 8
4.1.1 QuickSort . . . 9
4.1.2 MergeSort . . . 9
4.1.3 HeapSort . . . 9
4.1.4 RadixSort . . . 10
4.1.5 TimSort . . . 10
4.2 Description des générateurs . . . 11
4.2.1 Générateur Croissant . . . 11
4.2.2 Générateur Décroissant . . . 11
4.2.3 Générateur Swap . . . 12
4.2.4 Générateur Kendall-Tau . . . 12
4.3 Description de nos types de désordre . . . 12
4.3.1 Désordre partielle . . . 12
4.3.2 Désordre Totale . . . 13
4.4 Description des description des sondes . . . 13
4.4.1 Nombre de Comparaison : . . . 13
4.4.2 Nombre de permutation ou assignations : . . . 13
4.4.3 Nombre d'accès aux données . . . 14

5 Expérimentations **14**
5.1 Méthodologie . . . 14
5.1.1 Génération des Graphiques . . . 14
5.1.2 Plan des Expériences . . . 15

6 Analyse des résultats des Expérimentations **15**
6.1 InsertionSort . . . 15
6.1.1 Désordre 25% . . . 15
6.1.2 Désordre 50% . . . 16
6.1.3 Désordre 75% . . . 17
6.1.4 Désordre 100% . . . 18
6.2 QuickSort . . . 18
6.2.1 Désordre 25% . . . 18
6.2.2 Désordre 50% . . . 19
6.2.3 Désordre 75% . . . 20
6.2.4 Désordre 100% . . . 20
6.3 RadixSort . . . 21
6.3.1 Désordre 25% . . . 22
6.3.2 Désordre 50% . . . 22
6.3.3 Désordre 75% . . . 23
6.3.4 Désordre 100% . . . 23
6.4 TimSort . . . 24
6.4.1 Désordre 25% . . . 24
6.4.2 Désordre 50% . . . 24
6.4.3 Désordre 75% . . . 25
6.4.4 Désordre 100% . . . 25
6.5 BubbleSort . . . 26
6.5.1 Désordre 25% . . . 26
6.5.2 Désordre 50% . . . 27
6.5.3 Désordre 75% . . . 28
6.5.4 Désordre 100% . . . 28
6.6 HeapSort . . . 29
6.6.1 Désordre 25% . . . 29
6.6.2 Désordre 50% . . . 30
6.6.3 Désordre 75% . . . 31
6.6.4 Désordre 100% . . . 31
6.7 MergeSort . . . 32
6.7.1 Désordre 25% . . . 32
6.7.2 Désordre 50% . . . 33
6.7.3 Désordre 75% . . . 34
6.7.4 Désordre 100% . . . 34

7 Comparaison générale des algorithmes de tri **35**
7.1 Comparaison en fonction du temps d'exécution . . . 35
7.2 Comparaison selon le nombre de comparaisons . . . 36
7.3 Comparaison selon les accès mémoire . . . 36

8 Conclusion **36**
8.1 Influence de la quantité de désordre sur l'efficacité des algorithmes de tri . . . 36
8.2 Influence de la répartition du désordre . . . 37

1 Introduction

Aujourd'hui, nous sommes confrontés à des volumes de données de plus en plus importants, qu'il s'agisse de données numériques, textuelles ou structurées. Dans de nombreux domaines, du commerce en ligne à la recherche scientifique en passant par les systèmes d'exploitation, trier efficacement ces données est devenu une nécessité incontournable. C'est pourquoi l'analyse des algorithmes de tri reste un sujet central en informatique.

Chaque algorithme possède ses propres caractéristiques, points forts et faiblesses. Mais leur efficacité dépend aussi largement du type de désordre présent dans les données à trier. Dans ce projet, nous avons étudié plusieurs algorithmes de tri afin de comparer leur comportement en fonction de différents types de désordre (aléatoire, inversé, partiellement trié, etc.).

2 Objectifs du projet

L'objectif est de mieux comprendre comment adapter le choix d'un algorithme de tri à la nature des données pour optimiser les performances.

2.1 Problématique du projet

Comment l'efficacité des algorithmes de tri évolue-t-elle en fonction du niveau de désordre des données (quantité et répartition) ?

2.2 Description des points-clés et des grandes étapes

Afin de bien mener la réalisation du projet, il a été essentiel de le répartir en plusieurs tâches importantes.

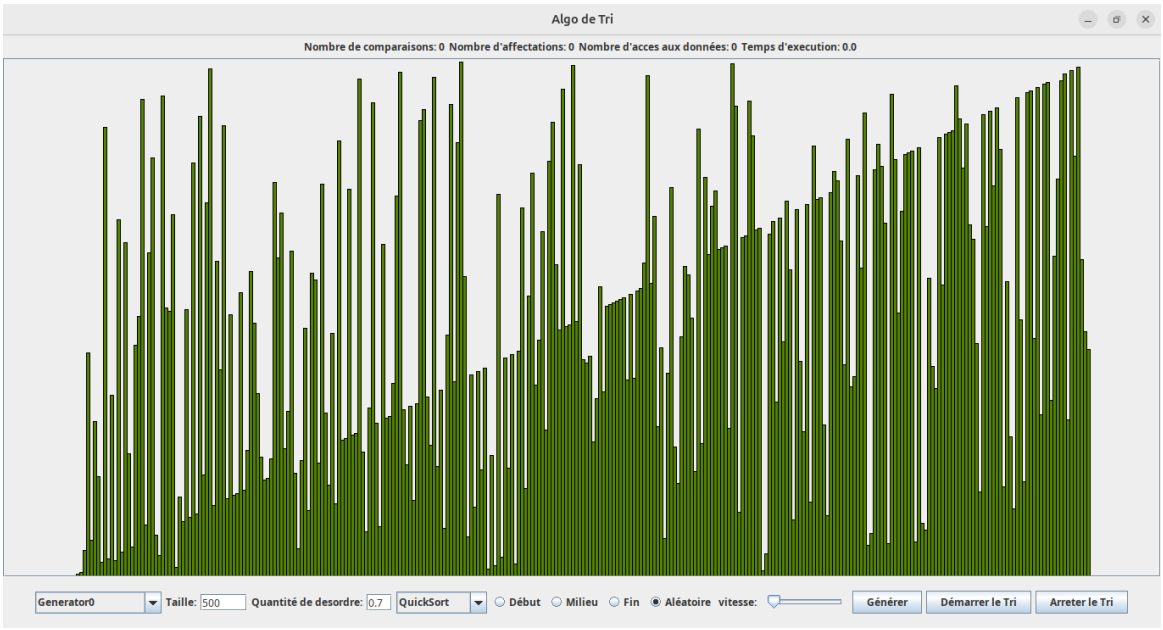
- **Phase de recherche :** Cette première étape nous a permis d'explorer le sujet, de nous familiariser avec les différents algorithmes de tri, et d'envisager les types de générateurs de désordre que nous allions développer.
- **Choix des méthodes de conception :** Nous avons ensuite défini l'architecture générale du projet ainsi que les patterns de conception les plus adaptés pour structurer efficacement notre code.
- **Phase d'implémentation :** Nous avons commencé par développer séparément les algorithmes de tri, les générateurs de désordre et l'interface graphique, sans encore les intégrer les uns aux autres.
- **Mise en communication des composants :** À cette étape, nous avons connecté les différents modules du projet afin qu'ils puissent interagir entre eux de manière cohérente.
- **Mise en place d'une structure pour l'expérimentation :** Nous avons conçu une structure adaptée pour réaliser nos expériences de manière systématique et reproductible.
- **Analyse des résultats :** Enfin, nous avons analysé les performances des algorithmes en fonction de différents paramètres, afin de comprendre ce qui influence leurs résultats et leur efficacité.

3 Fonctionnalités implémentées

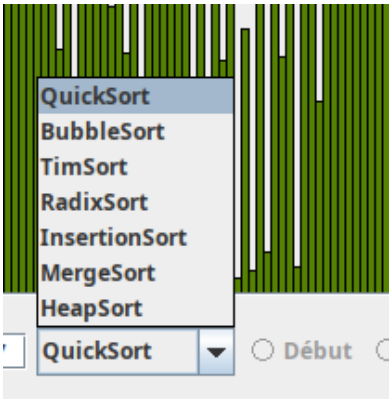
3.1 Description des fonctionnalités

3.1.1 Graphique

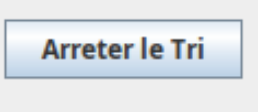
Ce projet nous permet de générer une liste désordonnée de nombres, selon un **niveau de désordre** (quantité) et une **répartition** (emplacement des désordres). Chaque nombre est représenté graphiquement par une barre dont la hauteur dépend de sa valeur.



Une fois la liste générée, nous pouvons sélectionner un algorithme de tri pour l'appliquer à cette liste.

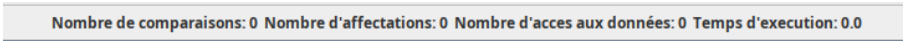


Conscients que certains algorithmes peuvent prendre un temps considérable à trier une liste, nous avons ajouté un bouton permettant d'interrompre un tri en cours.



À l'inverse, certains algorithmes sont si rapides que l'animation devient presque invisible à l'œil nu. Pour y remédier, nous avons intégré une barre coulissante permettant de **contrôler la vitesse du tri**.

Enfin, pour suivre l'avancement du tri en temps réel, nous avons ajouté un panneau d'informations affichant dynamiquement le **nombre de comparaisons**, le **nombre d'accès aux données** ainsi que le **temps d'exécution**.



3.2 Technique

Nous avons rendu possible le fait de pouvoir expérimenter et de comparer les différents algorithmes selon notre choix, et de pouvoir visualiser, à travers des courbes et des histogrammes, leur efficacité.

3.3 Organisation du projet

Afin de mener à bien ce projet, une répartition claire des tâches a été mise en place entre les membres du groupe. Chaque étape, de la conception à l'analyse des résultats, a été pensée de manière collaborative tout en attribuant à chacun des responsabilités spécifiques. Cette organisation nous a permis de travailler efficacement, de respecter les délais et de valoriser les compétences de chacun. La section suivante détaille les contributions individuelles au sein du projet.

3.3.1 Contributions Individuelles par tache :

Tâche	**Membres concernés**
Implémentation des algorithmes de tri	Youssouf Diare, Boubacar Sadio Diallo
Implémentation des générateurs de désordre	Franck Loick
Développement de l'interface graphique	Franck Loick ,El mamy, youssouf Diare
Connexion entre générateurs et algorithmes	Youssouf Diare, Boubacar Sadio Diallo
Connexion de l'interface avec les algorithmes et les générateurs	Youssouf Diare
Mise en place de la structure d'expérimentation	Boubacar Sadio Diallo, Franck Loick
Analyse des résultats et génération des courbes/histogrammes	Youssouf Diare, Franck Loick
Génération du fichiers XML	Boubacar Sadio Diallo
Rédaction des classes de test	El Mamy, Franck Loick

TABLE 1 – Répartition des contributions par tâche

Le projet est structuré en plusieurs packages clairs et cohérents, chacun jouant un rôle spécifique dans l'architecture logicielle. Cette section présente une description détaillée de chaque package ainsi que les patrons de conception utilisés.

3.4 Package controler

Ce package contient les classes liées à la gestion du modèle et à son écoute, ces classes principales sont :

— EcouteurModele.java

— ModeleEcoutable.java

— AbstractModeleEcoutable.java

ModeleEcoutable et **EcouteurModele** implémentent une architecture où les vues sont notifiées dès les changements du modèle. **AbstractModeleEcoutable** joue le rôle d'observable central.

3.5 Package model

Ce package contient deux sous-packages en plus du modèle principal de l'application.

3.5.1 model.algo

C'est dans ce package que nous recensons tous nos algos de tri et nous les implémentons à travers le design pattern stratégie qui nous permet de les interchanger dynamiquement selon le besoin via StrategieTri et ContexteTri.

Algorithmes classiques :

— QuickSort,

— MergeSort,

— HeapSort,

— TriBulle,

— InsertionSort, RadixSort,

— TimSort

3.5.2 model.generateur

Ce package se charge de l'implementation de nos générateur de desordre à travers le pattern strategie qui nous permet d'interchanger dynamiquement nos generateur.

Composants généraux :

— AbstractGenerator : cette classe sert de facteur commun à tous nos generateur il nous permet d'évité la redondance de code en définissant une seule fois toute les méthodes comunne à nos generateurs. StrategieGeneration, ContexteGeneration

Générateurs spécifiques :

— GenerateurBaseCroissant,

— GenerateurBaseDecroissant,

— GeneratorSwaps,

— GeneratorKendallTau

3.6 Packages experience

Le rôle de ce package est de s'occuper de tous ce qui concerne les experimentation , en passant par leurs automatisations , leurs stokage et leurs analyse . Ces sous packages sont :

— `experience.automatisation`.

— `experience.analyse`

— `experience.stockage`

3.6.1 Les classes et fichiers principaux

Main – Package experience : La classe Main, située dans le package experience, a pour objectif principal de permettre l'exécution automatique d'un tri dans un contexte expérimental. Lorsqu'elle est lancée, elle attend quatre paramètres en ligne de commande : le type de désordre à appliquer à la liste de données, le nom de l'algorithme de tri à utiliser, la taille de la liste, et enfin, la quantité de désordre à introduire. Ces paramètres permettent de simuler différentes configurations et de tester les performances des algorithmes dans des contextes variés.

Une fois les arguments fournis, la classe vérifie leur validité. Elle sélectionne ensuite un générateur de désordre adapté à partir du package **model.generateur**. Ce générateur crée une liste de nombres avec un niveau de désordre contrôlé selon les paramètres spécifiés. Ensuite, Main instancie le bon algorithme de tri à l'aide du contexte **ContexteTri**, présent dans le package **model.algo**, ce qui permet d'appliquer dynamiquement la stratégie de tri choisie.

L'algorithme est ensuite exécuté sur la liste générée, et durant ce processus, le système mesure plusieurs indicateurs de performance, tels que le temps d'exécution, le nombre de comparaisons effectuées et le nombre d'accès aux données. Une fois le tri terminé, ces résultats sont transmis à la classe Resultats, qui se charge de les enregistrer de manière structurée pour une analyse ultérieure. Cette approche permet de répéter facilement des expériences variées, dans un cadre rigoureux et reproductible, sans intervention manuelle.

Resultats - package experience : La classe Resultats, située dans le package experience.automatisation, est dédiée à la gestion de l'enregistrement des données issues des expérimentations. Elle intervient à la fin de chaque exécution lancée par la classe Main, afin de stocker les résultats de performance dans un fichier CSV. Lors de son initialisation, elle reçoit plusieurs informations essentielles : le type de désordre utilisé, le nom de l'algorithme de tri exécuté, la taille du tableau de données, et la quantité de désordre appliquée.

Son fonctionnement débute par une vérification de l'existence du répertoire de stockage. Si celui-ci n'existe pas encore (typiquement experience/stockage), la classe le crée automatiquement. Ensuite, elle ouvre — ou crée si besoin — le fichier resultats.csv dans ce répertoire. Ce fichier agit comme une base de données expérimentale, dans laquelle chaque ligne représente une exécution unique, accompagnée de toutes les métriques associées.

Chaque ligne du fichier contient les informations suivantes : le type de désordre, le nom de l'algorithme, la taille de la liste, la quantité de désordre, ainsi que les valeurs numériques mesurées pour le temps d'exécution, le nombre de comparaisons effectuées et le nombre d'accès aux données. Grâce à cette structure, les données enregistrées sont faciles à exploiter notamment pour produire des graphiques, faire des comparaisons entre algorithmes, ou analyser l'influence du désordre sur les performances.

3.7 Package vue

Le package vue regroupe l'ensemble des classes responsables de l'interface graphique du projet. Il représente la couche "Vue" dans l'architecture **MVC** (Modèle-Vue-Contrôleur). Son

objectif principal est de permettre à l'utilisateur d'interagir facilement avec les fonctionnalités offertes par l'application : génération d'une liste désordonnée, choix de l'algorithme de tri, visualisation du tri en temps réel, contrôle de la vitesse ou arrêt du processus.

Ce package s'appuie fortement sur la communication avec le modèle (via le contrôleur) pour se tenir informé de chaque changement d'état. Lorsqu'une opération est déclenchée ou qu'un tri est en cours, la vue est automatiquement mise à jour grâce au patron de conception **Observer**, ce qui assure une cohérence entre ce que voit l'utilisateur et ce qui se passe réellement en arrière-plan.

Le package contient trois classes principales : **TriGui, VueTri et Demo**, qui coopèrent pour proposer une interface complète, fonctionnelle et réactive.

3.7.1 Classe TriGui

TriGui est la classe principale de l'interface graphique. Elle construit l'ensemble de la fenêtre utilisateur et organise tous les éléments graphiques : boutons pour lancer ou arrêter le tri, menus déroulants pour choisir l'algorithme et le générateur de désordre, curseur pour régler la vitesse d'animation, et panneaux d'informations pour afficher en direct les statistiques comme le nombre de comparaisons, le temps d'exécution ou les accès mémoire.

Elle initialise également la communication avec le modèle (ModelTri) et configure les actions des différents boutons, assurant ainsi le lien entre l'utilisateur et la logique métier. C'est cette classe qui centralise l'ensemble des interactions de l'utilisateur.

3.7.2 Classe VueTri

VueTri est responsable de l'affichage visuel de la liste à trier. Chaque élément de la liste est représenté par une barre verticale dont la hauteur est proportionnelle à la valeur numérique correspondante. Lorsqu'un tri est lancé, cette classe met à jour l'affichage en temps réel à chaque modification du tableau, permettant ainsi de suivre visuellement le déroulement de l'algorithme.

Elle agit comme un écouteur du modèle (grâce au système d'observateurs) et redessine automatiquement l'état de la liste dès que le modèle change. Cela permet une visualisation claire et dynamique.

3.7.3 Classe Demo

La classe Demo sert de point d'entrée simplifié pour l'interface graphique. Elle instancie TriGui et initialise l'IHM sans nécessiter d'arguments externes. C'est la classe à utiliser lorsqu'on veut tester ou présenter rapidement l'application.

Même si son rôle est limité, elle est essentielle pour lancer le projet en mode graphique, notamment lors des démonstrations ou tests manuels.

4 Éléments techniques

4.1 Description des algorithmes de tri non triviaux

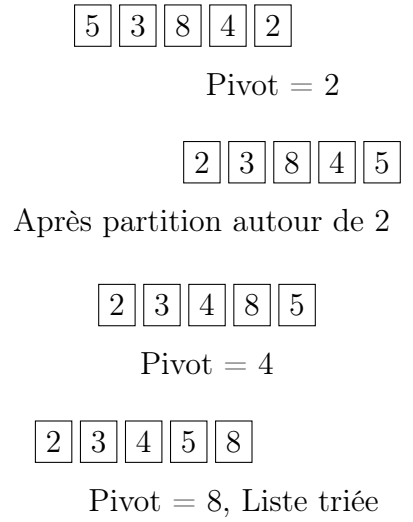
Avant de comparer les performances des différents algorithmes de tri, il est essentiel de bien comprendre leur fonctionnement interne. Parmi ces algorithmes, certains sont dits "non triviaux" car leur logique de tri repose sur des structures ou des stratégies avancées. Contrairement aux méthodes simples comme le tri à bulles ou le tri par insertion, ces algorithmes offrent des performances bien supérieures, en particulier sur des jeux de données volumineux ou variés. Cette section présente en détail cinq algorithmes de tri non triviaux largement utilisés :

QuickSort, MergeSort, HeapSort, RadixSort et TimSort. Pour chacun d'eux, nous illustrons leur comportement à l'aide d'exemples concrets et de schémas explicatifs .

4.1.1 QuickSort

Le **QuickSort** est un algorithme de type *divide and conquer*. Il sélectionne un pivot, puis réorganise les éléments du tableau de manière à ce que ceux qui sont inférieurs soient à gauche, et les supérieurs à droite. Ensuite, il applique récursivement le même processus aux sous-tableaux.

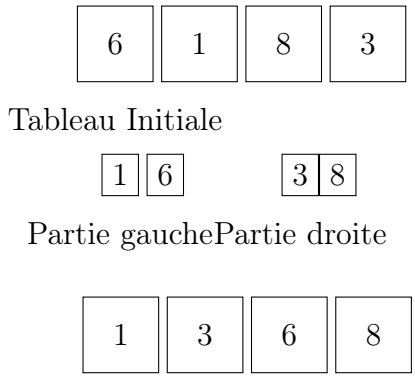
Principe illustré :



4.1.2 MergeSort

Le **MergeSort** divise le tableau en deux parties, les trie indépendamment, puis les fusionne.

Exemple de fusion :

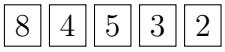


4.1.3 HeapSort

Le **HeapSort** construit un *tas binaire* à partir du tableau, puis extrait l'élément maximal à chaque étape.

Exemple d'extraction :

Liste initiale



Tas max

Liste triée

4.1.4 RadixSort

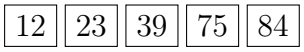
RadixSort est un algorithme de tri non comparatif qui trie les entiers en les examinant chiffre par chiffre, en partant du chiffre de poids faible (unités) jusqu'au chiffre de poids fort (dizaines, centaines, etc.). À chaque passe, un tri stable est effectué pour conserver l'ordre relatif des valeurs.

Exemple avec des entiers à deux chiffres :
Soit la liste suivante : `[75, 23, 84, 12, 39]`

- **1re passage (unités) :** On trie selon le chiffre des unités :
  75 (5), 23 (3), 84 (4), 12 (2), 39 (9) ⇒ `[12, 23, 84, 75, 39]`
- **2e passage (dizaines) :** On trie maintenant selon les dizaines tout en conservant l'ordre actuel pour les unités similaires :
  12 (1), 23 (2), 84 (8), 75 (7), 39 (3) ⇒ `[12, 23, 39, 75, 84]`
- **Résultat final :** La liste est maintenant triée selon les valeurs entières.

Unités (1re passe)

Dizaines (2e passe)



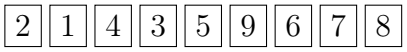
Liste triée finale

4.1.5 TimSort

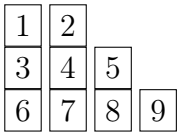
Le **TimSort** est un algorithme hybride combinant *Insertion Sort* et *MergeSort*. Il est optimisé pour des tableaux partiellement triés.

Il découpe le tableau en **runs** (sous-tableaux), trie chaque run par insertion, puis fusionne les runs entre eux efficacement.

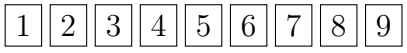
Illustration simplifiée :



Liste initiale



Runs identifiés et triés localement



Fusion finale des runs

4.2 Description des générateurs

Afin d'évaluer objectivement les performances des algorithmes de tri, il ne suffit pas d'observer leur comportement sur des listes aléatoires. Il est également nécessaire de contrôler la structure du désordre initial, c'est-à-dire la manière dont les éléments sont répartis ou perturbés. Pour cela, nous avons mis en place différents générateurs de désordre, chacun simulant un type de perturbation spécifique : inversions localisées, entropie contrôlée, liste partiellement trié, permutations par swaps, etc. Cette section présente ces générateurs et leur fonctionnement.

4.2.1 Générateur Croissant

Ce générateur permet d'introduire du désordre dans une liste initialement triée de manière croissante, en contrôlant à la fois la quantité et la répartition de ce désordre. Nous avons veillé à ce qu'il soit possible de spécifier précisément la zone où le désordre est appliqué, car sa localisation peut avoir un impact significatif sur l'efficacité des algorithmes de tri, un aspect que nous analyserons dans les sections suivantes.

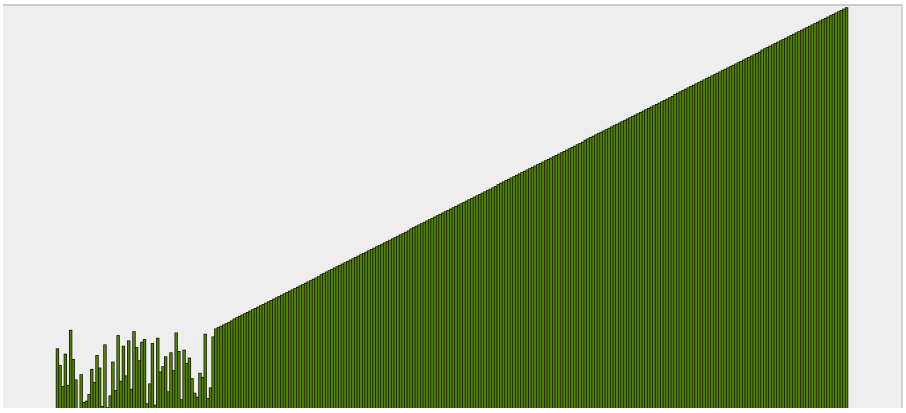


FIGURE 1 – générateur croissant - désordoné au début `vue`

4.2.2 Générateur Décroissant

Ce générateur produit une liste triée dans l'ordre décroissant, sur laquelle un certain niveau de désordre peut être introduit. Comme pour le générateur croissant, il est possible de contrôler la quantité et la localisation du désordre. Cette capacité à cibler une zone spécifique du tableau est essentielle, car elle permet d'observer comment les algorithmes de tri réagissent à des perturbations localisées dans une structure initialement inversée.

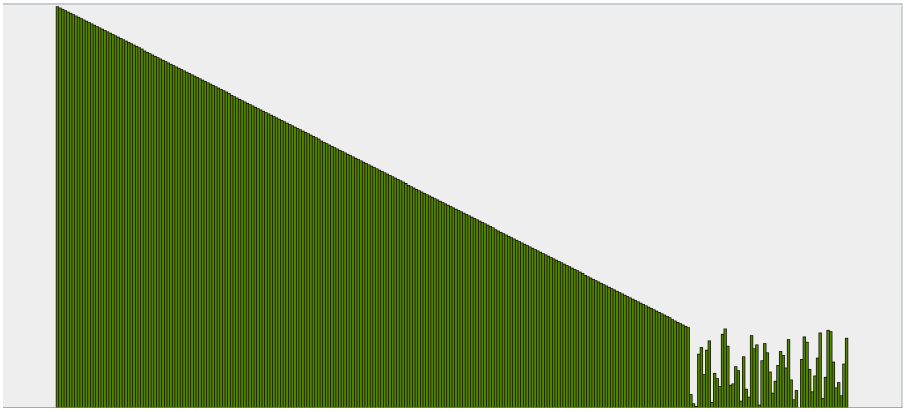


FIGURE 2 – générateur décroisant - désordoné à la fin `vue`

4.2.3 Générateur Swap

Ce générateur produit une liste triée dans l'ordre croissant, sur laquelle un certain niveau de désordre peut être introduit. Comme pour le générateur croissant ou décroissant, il est possible de contrôler la quantité et la localisation du désordre. Dans ce générateur , on effectuer un certains nombre de swaps selon la quantité de désordre rentré en paramètre.

4.2.4 Générateur Kendall-Tau

A l'image des trois autres générateurs, il suit la même méthode de conception. La différence ne se situe qu'au niveau du calcul de la génération de Désordre. Kendall-Tau est une mesure qui permet de comparer de façon efficace les algos de tri en surveillant le nombre de permutations. La logique de est de donnée un pourcentage de désordre qui nous donne un tau de Kendall de départ pour générer le désordre et qu'on comparera ensuite avec le nombre de swaps.

4.3 Description de nos types de désordre

Dans le but de mener des expériences pertinentes, nous avons choisi de tester différents types de désordre, que nous avons regroupés en deux catégories : les désordres partiels, où seule une partie de la liste est perturbée, et les désordres totaux, où le désordre concerne tous les éléments de la liste . Cette distinction nous permet d'évaluer plus finement nos algorithmes.

4.3.1 Désordre partielle

Nous avons défini huit types de désordres partiels. Leur spécificité principale réside dans la localisation du désordre : celui-ci peut être situé au début, au milieu ou à la fin de la liste. De plus, l'intensité du désordre est ajustable grâce à un paramètre de quantité qui varie entre 0 et 1, où 0 correspond à une liste totalement triée et 1 à une portion totalement désordonnée.

— **partielleDébut** : désigne une liste de nombres où le désordre est concentré au début, selon la quantité spécifiée.

— **partielleMilieu** : désigne une liste de nombres où le désordre est localisé au milieu, en fonction de la quantité choisie.

— **partielleFin** : désigne une liste de nombres où le désordre est appliqué à la fin, proportionnellement à la quantité donnée.

— **partielleAléa** : désigne une liste où le désordre est réparti de manière uniforme sur l'ensemble de la liste, selon la quantité.

Nous retrouvons les mêmes types de désordre que précédemment, mais cette fois à partir d'une liste initialement triée dans l'ordre décroissant. Ces désordres, appliqués sur une base inversée, permettent d'évaluer les performances des algorithmes dans des contextes où les données sont à l'opposé de l'ordre souhaité. Les types générés sont les suivants :

— **partielleDébutInverse** : le désordre est introduit au début d'une liste décroissante, en fonction de la quantité spécifiée.
— **partielleMilieuInverse** : le désordre est localisé au centre de la liste inversée, selon une proportion définie.
— **partielleFinInverse** : le désordre est appliqué à la fin de la liste décroissante.
— **partielleAléaInverse** : le désordre est réparti aléatoirement sur toute la liste inversée, proportionnellement à la quantité.

4.3.2 Désordre Totale

Ce type de désordre correspond à une situation où la quantité de désordre est maximale. Autrement dit, tous les éléments de la liste sont désorganisés et aucun n'est à sa position initiale

4.4 Description des description des sondes

Pour analyser finement le comportement des algorithmes de tri, il est nécessaire d'aller au-delà de la simple mesure du temps d'exécution. C'est pourquoi nous avons mis en place un système de sondes. Ces sondes sont des outils de mesure intégrés au cœur des algorithmes, permettant de compter avec précision le nombre de comparaisons effectuées, le nombre d'accès en lecture ou écriture sur les données, le nombre de permutation, ainsi que le temps réel d'exécution. Elles constituent un élément clé de notre démarche expérimentale, car elles offrent une vision détaillée de la complexité opérationnelle des algorithmes dans différents contextes de désordre.

4.4.1 Nombre de Comparaison :

Parmi les algorithmes que nous avons implémentés, certains réalisent le tri essentiellement en comparant les valeurs entre elles avant de procéder à des permutations. Dans notre projet, le nombre de comparaisons représente le nombre de fois où deux valeurs du tableau sont explicitement comparées dans le but de déterminer leur ordre. Il est important de noter que nous ne comptons que les comparaisons portant sur les valeurs elles-mêmes, et non celles qui pourraient concerner des index.

4.4.2 Nombre de permutation ou assignations :

Lors du tri d'une liste, nos algorithmes procèdent à des permutations entre les éléments afin de les réorganiser dans le bon ordre. Cependant, certains algorithmes effectuent un nombre de permutations nettement plus élevé que d'autres. Il est également important de souligner que toutes les permutations ne sont pas définitives : certains éléments peuvent être déplacés plusieurs fois avant d'atteindre leur position finale dans le tableau trié.

La sonde dédiée au comptage des permutations ou assignations permet de mesurer avec précision le nombre total de mouvements effectués sur les données. Elle offre ainsi une autre perspective sur l'efficacité d'un algorithme, notamment en termes d'impact sur la mémoire et le cache processeur. Ce type de mesure est particulièrement utile pour comparer les performances réelles entre des algorithmes qui, à complexité égale, peuvent avoir un comportement très différent selon la structure de la donnée d'entrée.

4.4.3 Nombre d'accès aux données

Le nombre d'accès aux données désigne le nombre de fois où une valeur de la liste à trier est lue ou modifiée, typiquement lors d'expressions comme **array[i]**. Il est important de noter que, dans notre convention de mesure, lorsqu'une opération implique à la fois une lecture et une écriture sur une même case du tableau (comme dans une affectation simple ou un échange), nous n'incrémentons le compteur qu'une seule fois. Ce choix permet de ne pas surévaluer artificiellement les accès dans les situations où lecture et écriture sont indissociables.
Exemple

```
public void swap(int[] array, int i, int j) {
    if (i != j && array[i] != array[j]) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

Listing 1 – Méthode `swap` pour échanger deux éléments d'un tableau

Avec la méthode **swap** que nous avons implémentée, le nombre d'accès aux données est incrémenté de deux à chaque appel. En effet, cette opération implique la lecture de deux valeurs distinctes du tableau : `array[i]` et `array[j]`. Même si ces accès sont suivis de modifications (écritures), nous avons choisi de ne comptabiliser ici que les lectures, conformément à notre convention de mesure.

5 Expérimentations

L'objectif principal de ces expérimentations est d'analyser la performance de plusieurs algorithmes de tri en fonction de différentes tailles de tableaux et de pourcentages de désordre. En particulier, nous testerons les algorithmes sur des tableaux dont le désordre varie en termes d'intensité (25 %, 50 %, 75 %, 100 %) et de localisation (désordre partiel ou total), pour mieux comprendre comment chaque algorithme réagit face à ces différentes situations.

5.1 Méthodologie

Pour chaque algorithme, les temps d'exécution ont été mesurés en millisecondes. Ces temps sont collectés pour chaque combinaison de taille de tableau, de pourcentage de désordre et de type de désordre. Les tests ont été réalisés sur des machines avec des configurations similaires pour garantir la reproductibilité des résultats. La mesure des performances est effectuée à l'aide de la méthode `System.nanoTime()` en Java, pour obtenir une estimation précise du temps d'exécution.

5.1.1 Génération des Graphiques

Les résultats sont représentés graphiquement sous forme de courbes, avec les axes définis comme suit :

— **Axe X (horizontal)** : Représente la **taille du tableau**. Les valeurs sont : 100, 5000, 100000, 1000000.

— **Axe Y (vertical)** : Représente le **temps d'exécution** en millisecondes.

— **Courbes** : Chaque courbe représente un type de désordre appliqué sur un tableau d'une taille donnée. Les types de désordre sont : **PartielleDébut**, **PartielleMilieu**, **PartielleFin**, **PartielleAléa**, etc. Chaque algorithme aura des graphes distincts pour chaque pourcentage de désordre (25 %, 50 %, 75 %, 100 %).

5.1.2 Plan des Expériences

Les expérimentations seront menées de manière suivante :

1. **Initialisation des Tests** : Les tableaux seront générés avec des tailles et des désordres spécifiés.
2. **Exécution des Algorithmes** : Chaque algorithme sera exécuté pour chaque type de désordre et pour chaque taille de tableau.
3. **Collecte des Résultats** : Les temps d'exécution seront collectés pour chaque combinaison d'algorithme, de taille et de désordre.
4. **Génération des Graphiques** : Les résultats seront représentés graphiquement pour chaque algorithme et pour chaque pourcentage de désordre. Les courbes seront tracées pour comparer les performances des algorithmes sous différents types de désordre et tailles de tableau.

6 Analyse des résultats des Expérimentations

Dans cette section, nous présentons les résultats expérimentaux pour les différents algorithmes de tri testés. Les performances sont mesurées en fonction de plusieurs niveaux de désordre (25%, 50%, 75% et 100%) pour chaque algorithme.

6.1 InsertionSort

6.1.1 Désordre 25%

Sur 25% de désordre, on constate que pour les différents types de désordres les performances sont plutôt positifs sauf sur le désordre *partielleFinInverse* pour lequel la courbe croit de façon exponentielle. Le temps d'exécution pour ce dernier augmente au fur et à mesure que la taille des données augmente. Ce qui rend cet algorithme inefficace sur ce type de désordre. Le temps reste en dessous de 1000s pour les autres courbes mais est presque multiplié par 5 pour un désordre aléatoire inverse en fin de données.

L'inefficacité de cet algorithme sur ce type de désordre s'explique par le fait que, lorsque le désordre est concentré à la fin du tableau, l'algorithme doit parcourir toute la liste jusqu'à rencontrer le premier élément mal placé, ce qui augmente inutilement le nombre d'itérations.

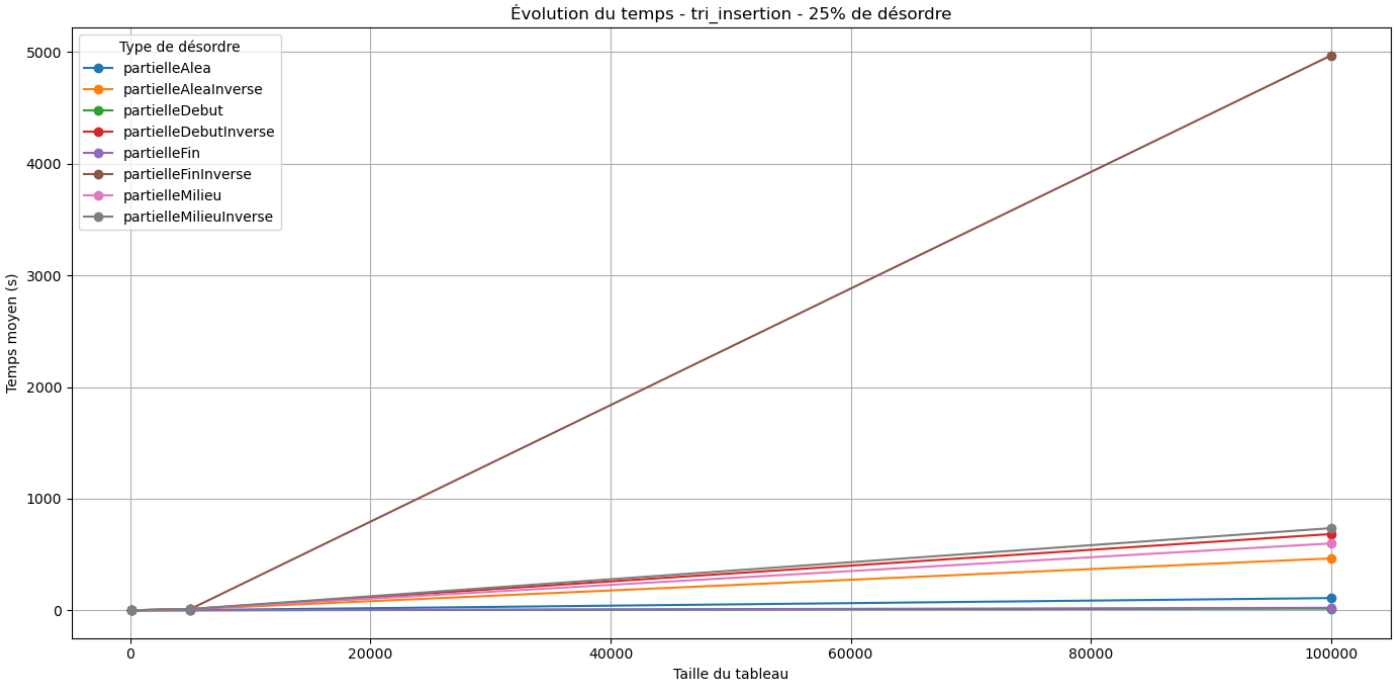


FIGURE 3 – Résultat sur un quantité de désordre de 25% sur l'algo Insertion

6.1.2 Désordre 50%

Pour une quantité de 50% sur des données, le temps d'exécution suit une croissance quasi linéaire, reflétant la complexité $O(n^2)$ du tri par insertion dans le pire des cas, mais pouvant être optimisé sur des séquences partiellement triées.

Les types de désordre influencent fortement les performances : le désordre aléatoires (*partielleAlea*) affichent le temp les plus faible, tandis que les désordres structurés en fin de tableau (*partielleFinInverse*), allongent considérablement le temps d'exécution.

Les cas de désordre en milieu de tableau (*partielleMilieu* et *partielleMilieuInverse*) restent également coûteux, bien que légèrement inférieurs aux désordres de fin. Ces résultats confirment que le tri par insertion est particulièrement inefficace lorsque le désordre est concentré en fin de tableau, alors qu'il reste performant sur des données partiellement triées.

Le graphique ci-dessous illustre l'évolution du temps d'exécution du tri par insertion en fonction de la taille du tableau avec un désordre de 50%.

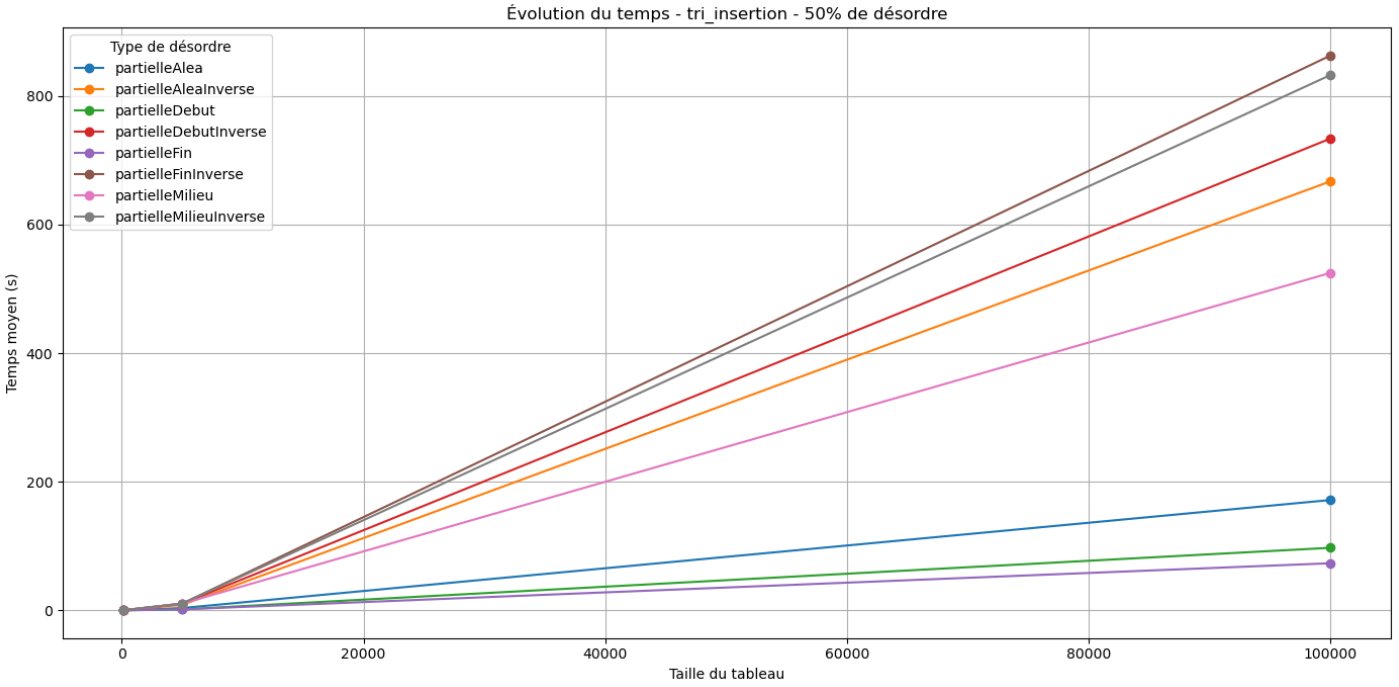


FIGURE 4 – Résultat sur un quantité de désordre de 50% sur l'algo Insertion

6.1.3 Désordre 75%

La quantité de désordre de 75% est à l'opposé du précédent à quelques exceptions près. On remarque une optimisation de performance sur certains types de désordres structurés qui connaissent une baisse de temps d'exécution à l'image du désordre *partielleFinInverse*) qui connaît une optimisation considérable. Faisant chemin inverse le désordre *partielleAlea* connaît lui une augmentation de temps. Il devient difficile de gérer ce type désordre surtout avec de grandes tailles.

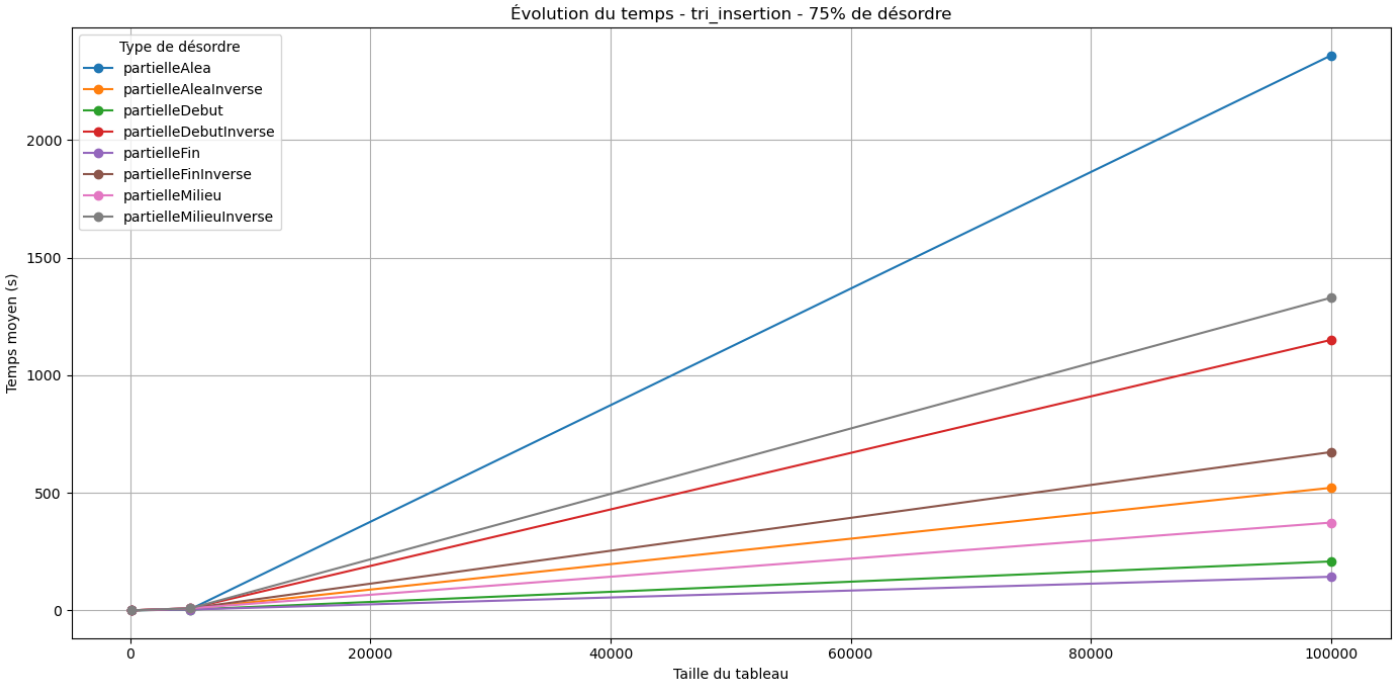


FIGURE 5 – Résultat sur un quantité de désordre de 75% sur l'algo Insertion

6.1.4 Désordre 100%

Ce graphique ci dessous représente l'évolution du temps d'exécution du tri par insertion en fonction de la taille du tableau, lorsque les données sont totalement désordonnées (100% de désordre). On observe une croissance exponentielle du temps d'exécution avec l'augmentation de la taille du tableau, ce qui est conforme à la complexité théorique du tri par insertion en $O(n^2)$ dans le pire des cas. Pour de petites tailles de tableaux, le temps de tri reste relativement faible, mais dès que la taille atteint plusieurs dizaines de milliers d'éléments, le temps de calcul devient prohibitif, dépassant plusieurs centaines voire milliers de secondes pour les plus grands jeux de données. Cette analyse confirme que le tri par insertion est inefficace pour trier de grands ensembles de données fortement désordonnés.

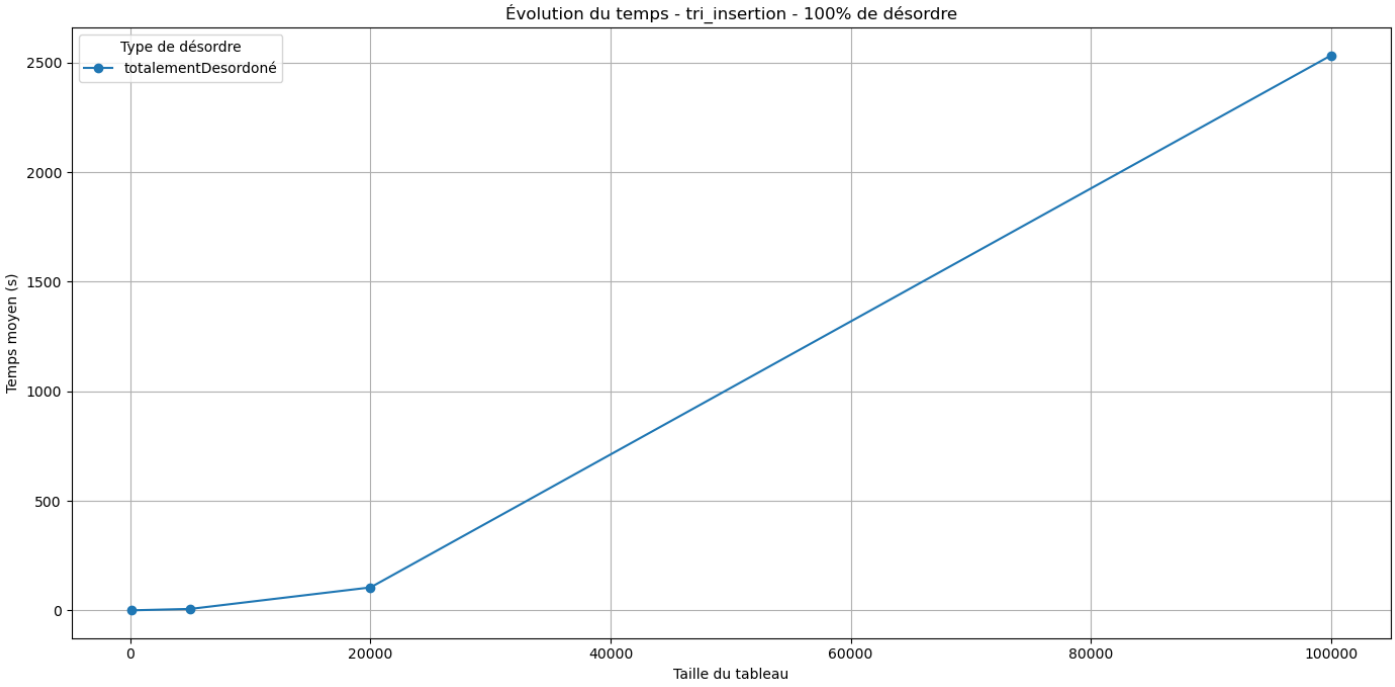


FIGURE 6 – Résultat sur un quantité de désordre de 100% sur l'algo Insertion

6.2 QuickSort

6.2.1 Désordre 25%

On constate dans un premier temps que l'algorithme QuickSort est efficace pour les désordres de type *partielleAlea* et *partielleAleaInverse*. Cela peut s'expliquer par le fait que ces types de désordre permettent une meilleure répartition des éléments autour du pivot, favorisant ainsi des découpages équilibrés lors des appels récursifs.

Dans notre implémentation du tri rapide, le pivot est initialement positionné à l'extrémité droite du tableau et se déplace vers la gauche. Plus les éléments en désordre sont éloignés de cette position (donc vers le début du tableau), plus l'algorithme est susceptible de devoir parcourir de longues séquences triées, ce qui nuit à son efficacité.

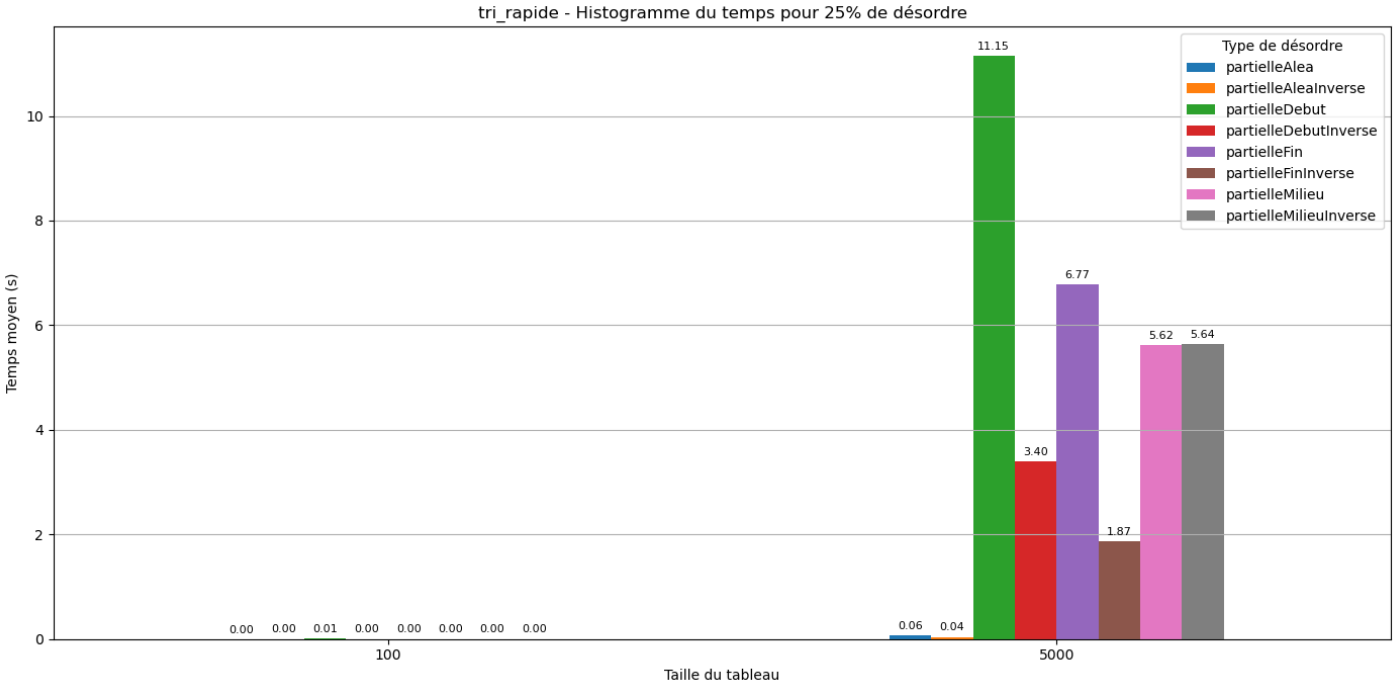


FIGURE 7 – Résultats pour un taux de désordre de 25% avec l'algorithme QuickSort

6.2.2 Désordre 50%

À 50% de désordre, QuickSort continue de montrer de bonnes performances. Le désordre est suffisamment réparti pour permettre un découpage efficace des sous-tableaux, même si l'on observe une légère augmentation du temps d'exécution pour certains types de désordre (notamment *partielleAleaInverse*).

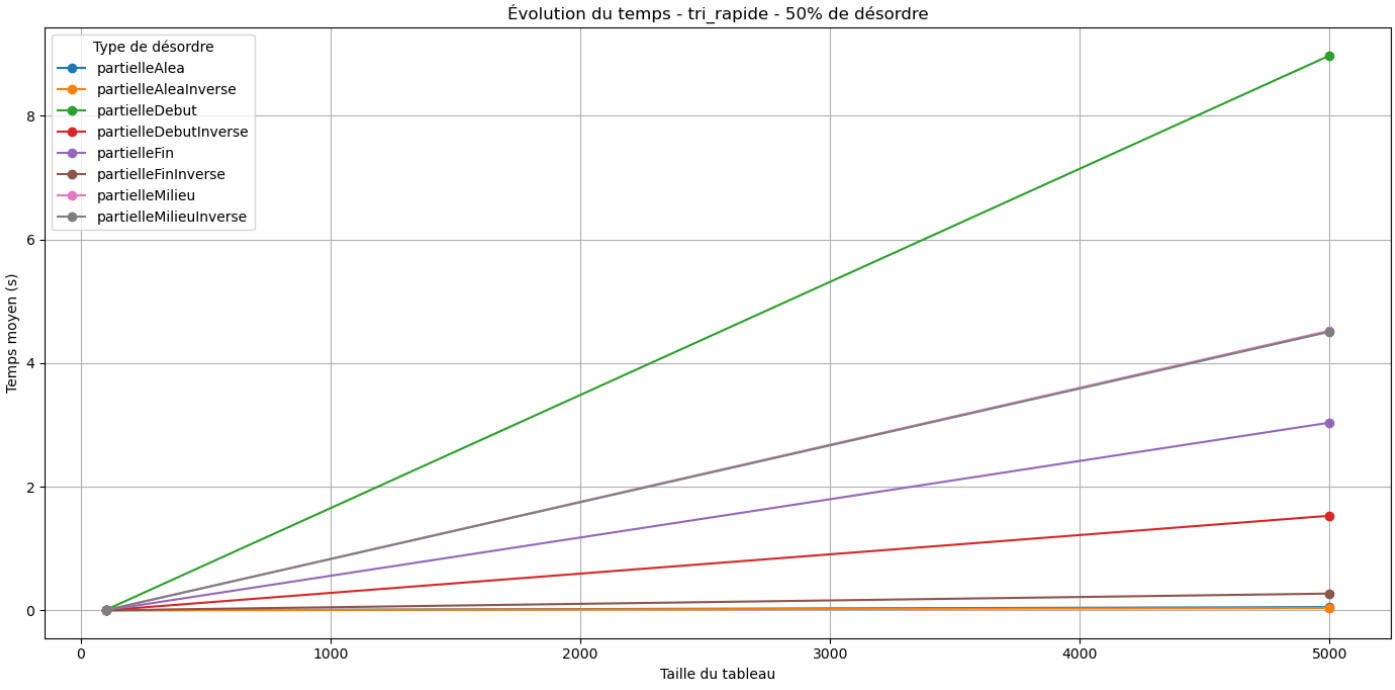


FIGURE 8 – Résultats pour un taux de désordre de 50% avec l'algorithme QuickSort

6.2.3 Désordre 75%

Lorsque le désordre atteint 75%, QuickSort reste relativement stable en termes de performances, en particulier pour le désordre *partielleAlea*, qui favorise toujours un bon choix du pivot. On note cependant que les performances sont plus sensibles au type de désordre structuré ce qui peut ralentir significativement l'algorithme.

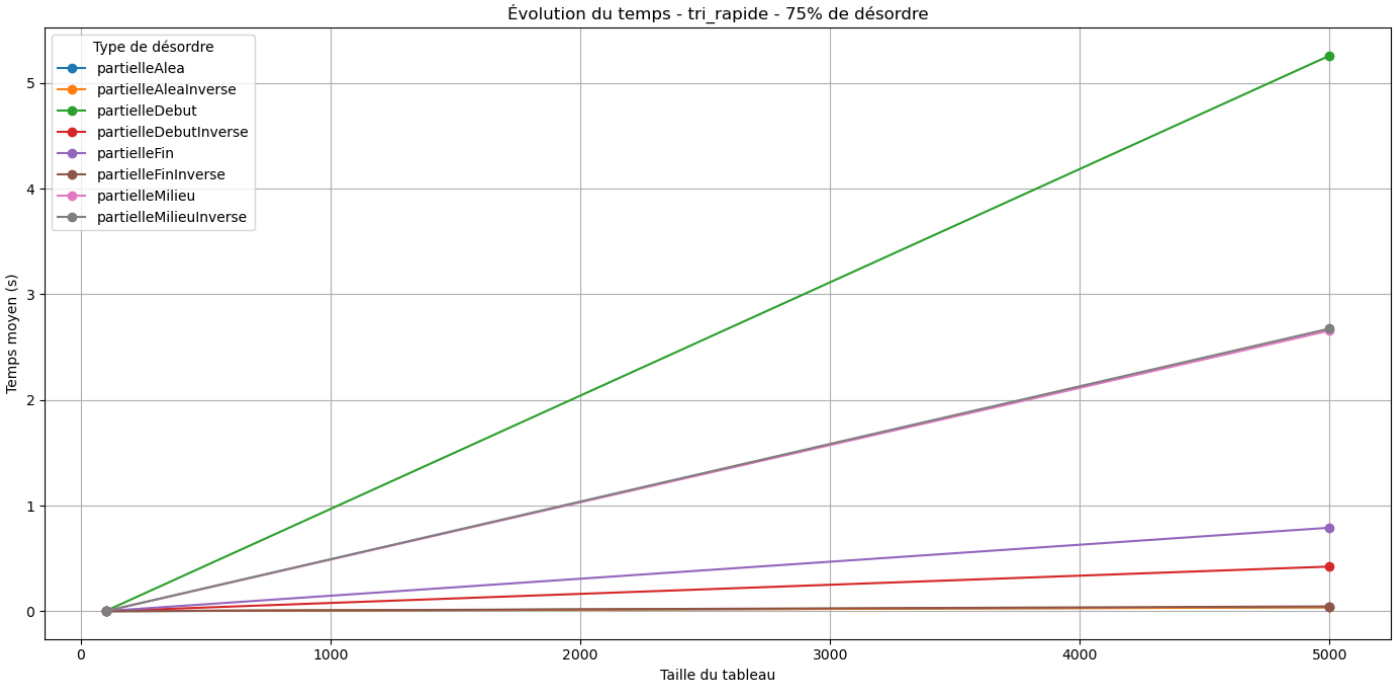


FIGURE 9 – Résultats pour un taux de désordre de 75% avec l'algorithme QuickSort

6.2.4 Désordre 100%

Avec un désordre total (100%), le tri rapide donne de très bons résultats. Le caractère complètement aléatoire du tableau garantit généralement un découpage équilibré à chaque étape, ce qui correspond au cas moyen optimal de complexité $O(n \log n)$. QuickSort tire donc pleinement profit de la nature chaotique des données dans ce scénario.

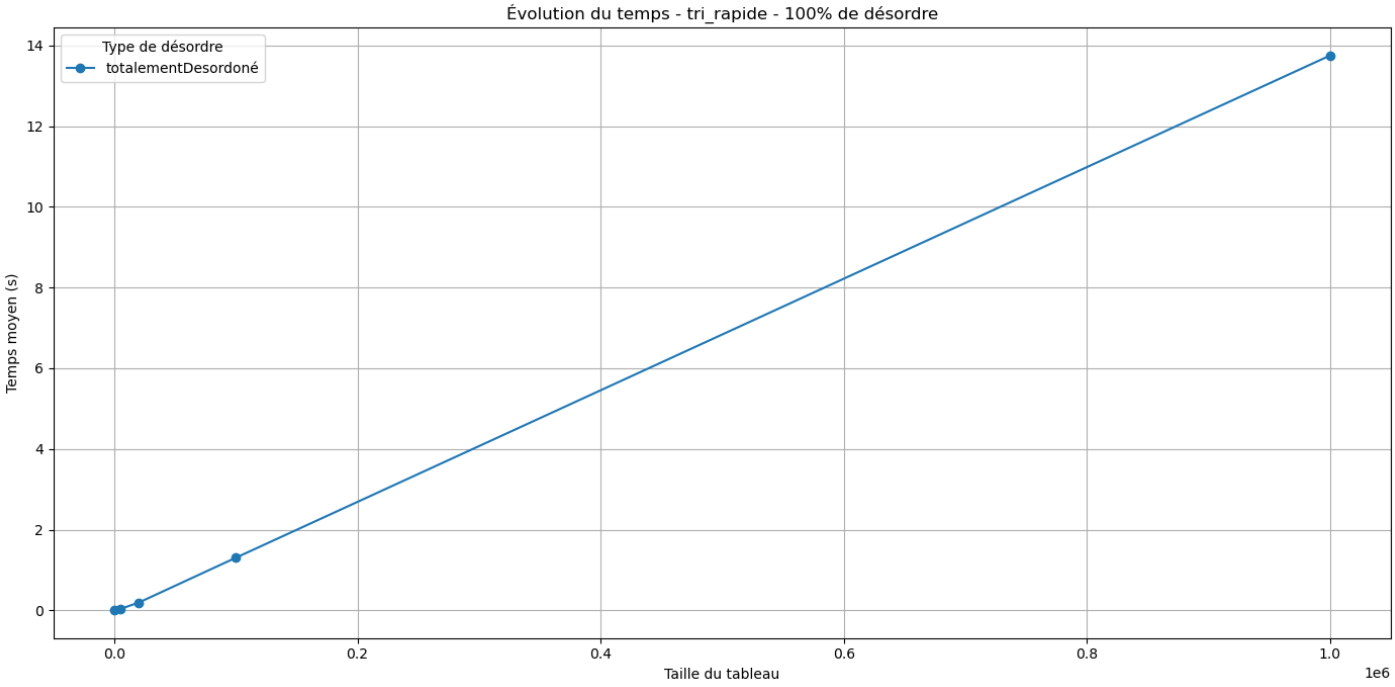


FIGURE 10 – Résultats pour un taux de désordre de 100% avec l'algorithme QuickSort

6.3 RadixSort

RadixSort, également appelé tri par base, est un algorithme non comparatif particulièrement efficace pour trier des entiers. Il trie les éléments chiffre par chiffre, dans notre cas il tri les unité en suite les dizaine ,ainsi de suite. Sa complexité théorique est de $O(n \cdot k)$ où k est le nombre de chiffres (ou de bits), ce qui le rend très performant pour des entiers de taille limitée.

Une de ses principales caractéristiques est que ses performances sont indépendantes du désordre. Contrairement aux algorithmes comparatifs comme QuickSort ou InsertionSort, RadixSort ne dépend pas des comparaisons mais plutôt du nombre de chiffres utilisés dans les entiers.

6.3.1 Désordre 25%

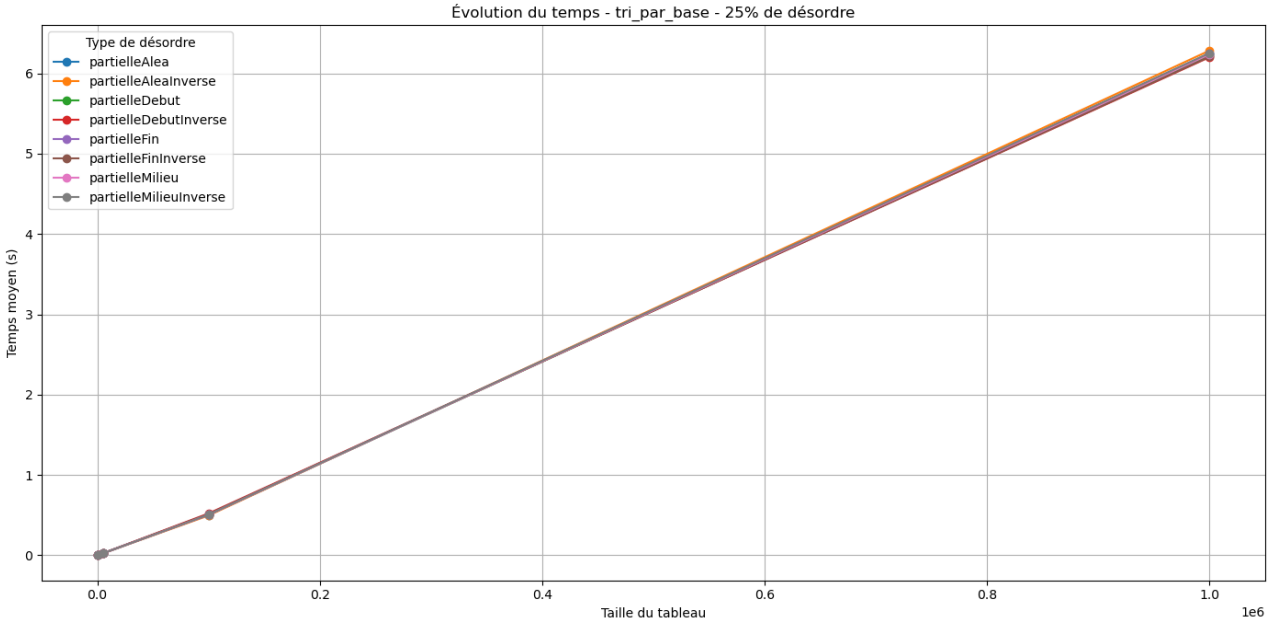


FIGURE 11 – Résultats pour un taux de désordre de 25% avec l'algorithme RadixSort

6.3.2 Désordre 50%

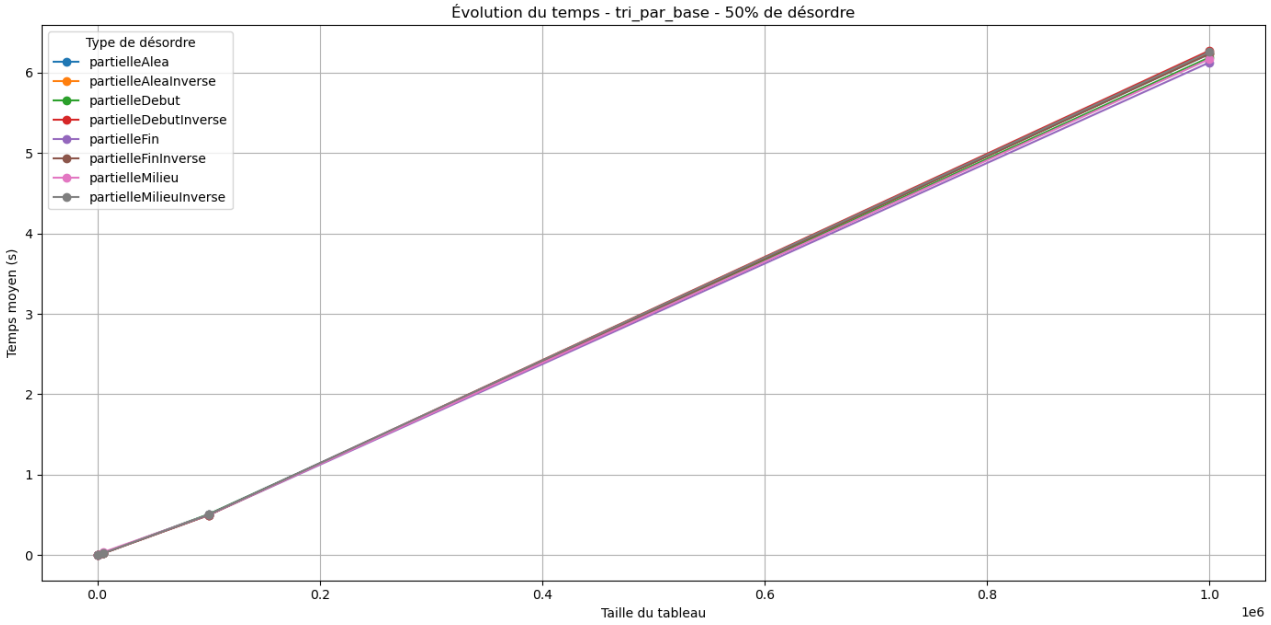


FIGURE 12 – Résultats pour un taux de désordre de 50% avec l'algorithme RadixSort

6.3.3 Désordre 75%

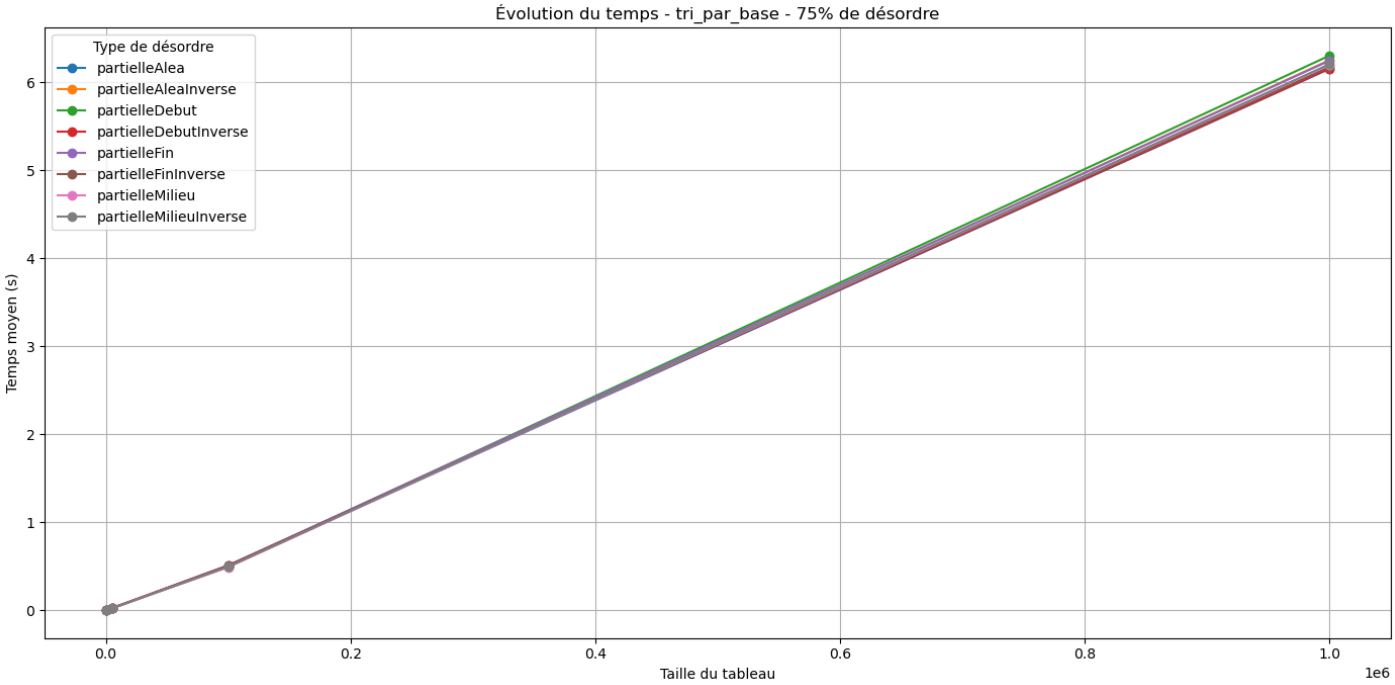


FIGURE 13 – Résultats pour un taux de désordre de 75% avec l'algorithme RadixSort

6.3.4 Désordre 100%

Même en cas de désordre total (100%), RadixSort reste très performant. Les écarts de temps entre les différents types de désordre sont très faibles, confirmant que l'algorithme est peu sensible à l'état initial des données.

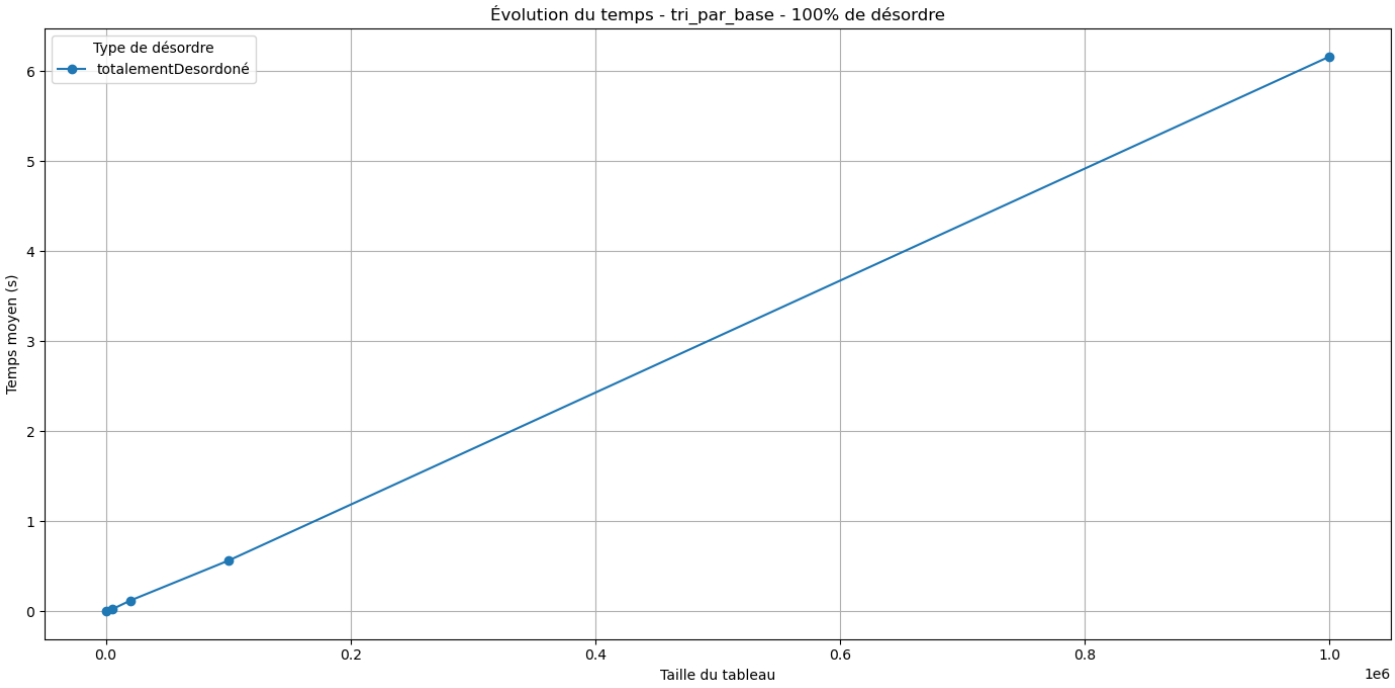


FIGURE 14 – Résultats pour un taux de désordre de 100% avec l'algorithme RadixSort

6.4 TimSort

TimSort est un algorithme hybride,le tri par insertion et le tri fusion.Il exploite les séquences déjà triées (appelées “runs”).

6.4.1 Désordre 25%

Pour un taux de désordre de 25%, TimSort affiche d’excellentes performances pour le désordre *partielleAlea*, grâce à la présence de nombreux sous-tableaux déjà triés qu’il peut exploiter directement. En revanche, le désordre *partielleAleaInverse* augmente significativement le temps d’exécution, car les séquences inversées sont plus coûteuses à traiter.

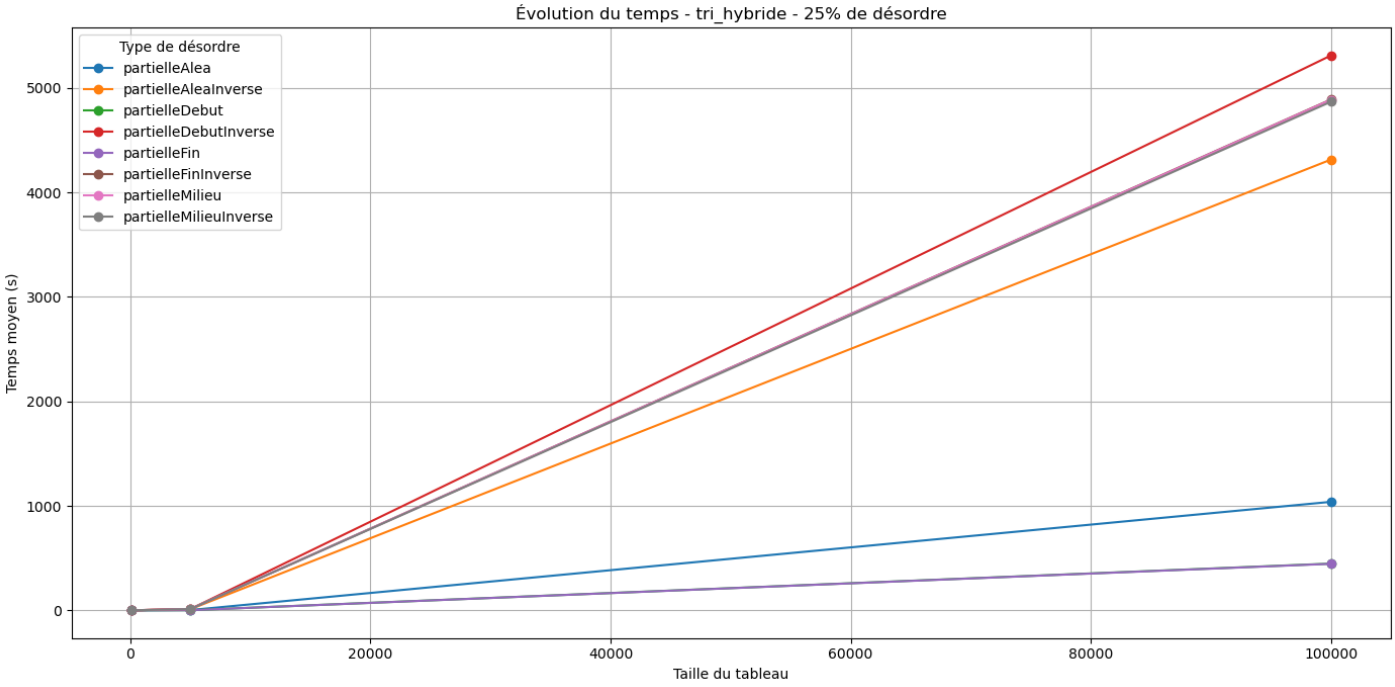


FIGURE 15 – Résultats pour un taux de désordre de 25% avec l’algorithme TimSort

6.4.2 Désordre 50%

À 50% de désordre, la tendance se confirme : les données aléatoirement désordonnées restent relativement efficaces à traiter pour TimSort, tandis que les désordres inversés continuent d’augmenter la charge de tri. TimSort reste largement plus rapide que les algorithmes simples comme BubbleSort.

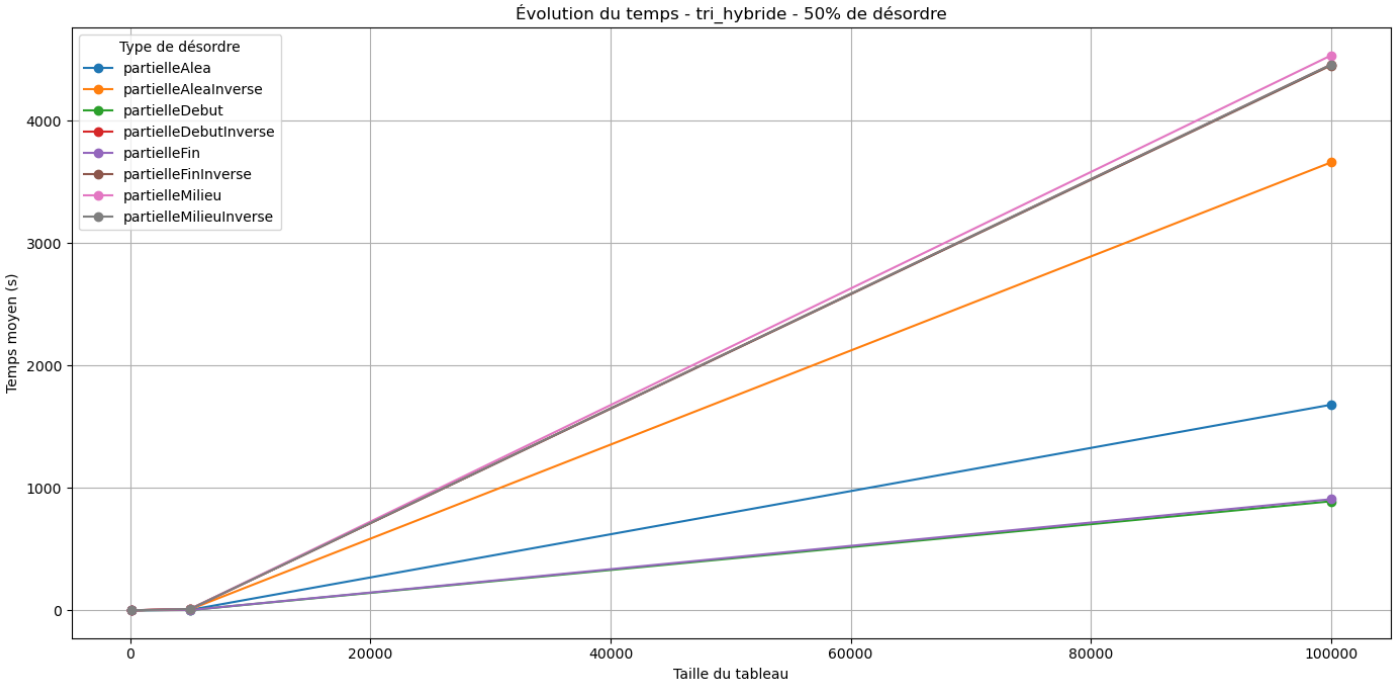


FIGURE 16 – Résultats pour un taux de désordre de 50% avec l'algorithme TimSort

6.4.3 Désordre 75%

On remarque que le Tim Sort continue à êtres inéfficasse pour des listes inverses.

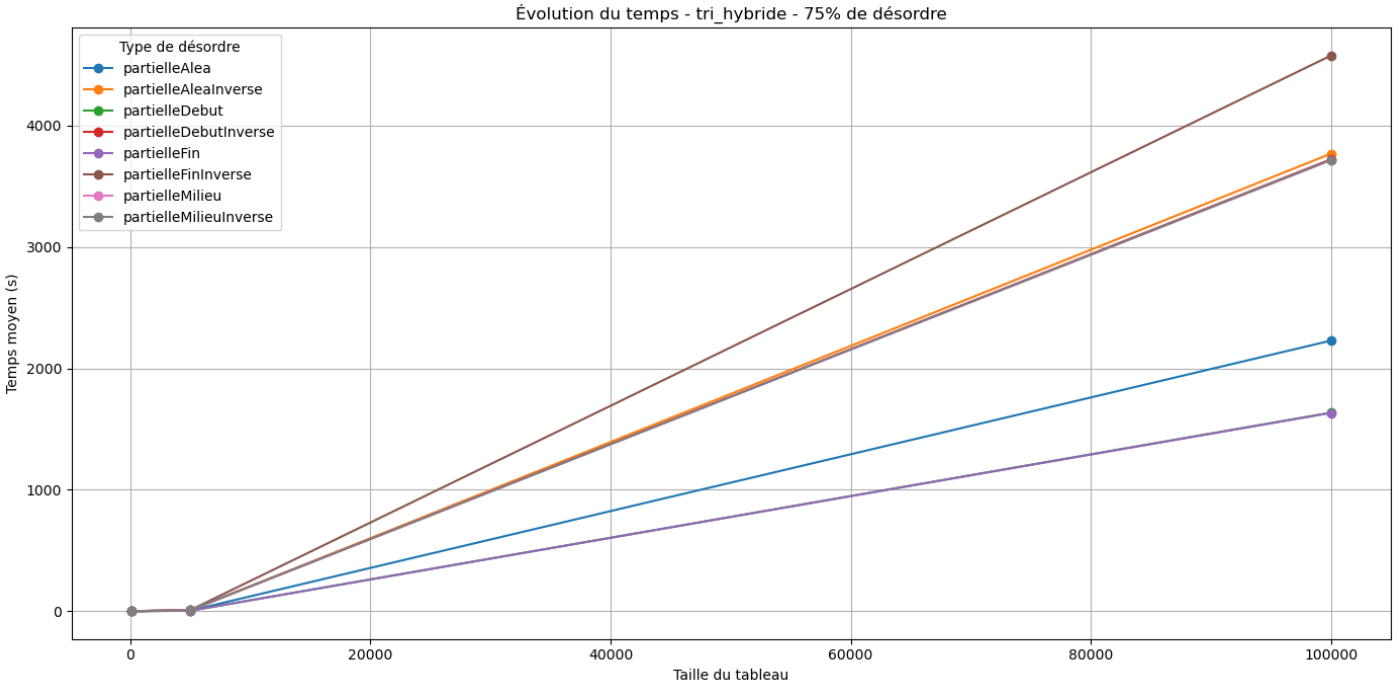


FIGURE 17 – Résultats pour un taux de désordre de 75% avec l'algorithme TimSort

6.4.4 Désordre 100%

En cas de désordre total, TimSort se comporte comme un tri fusion classique. Il n'exploite plus les séquences triées, mais continue à garantir une bonne performance grâce à son comporte-

ment en $O(n \log n)$. Il reste donc l'un des plus efficaces même dans des conditions défavorables.

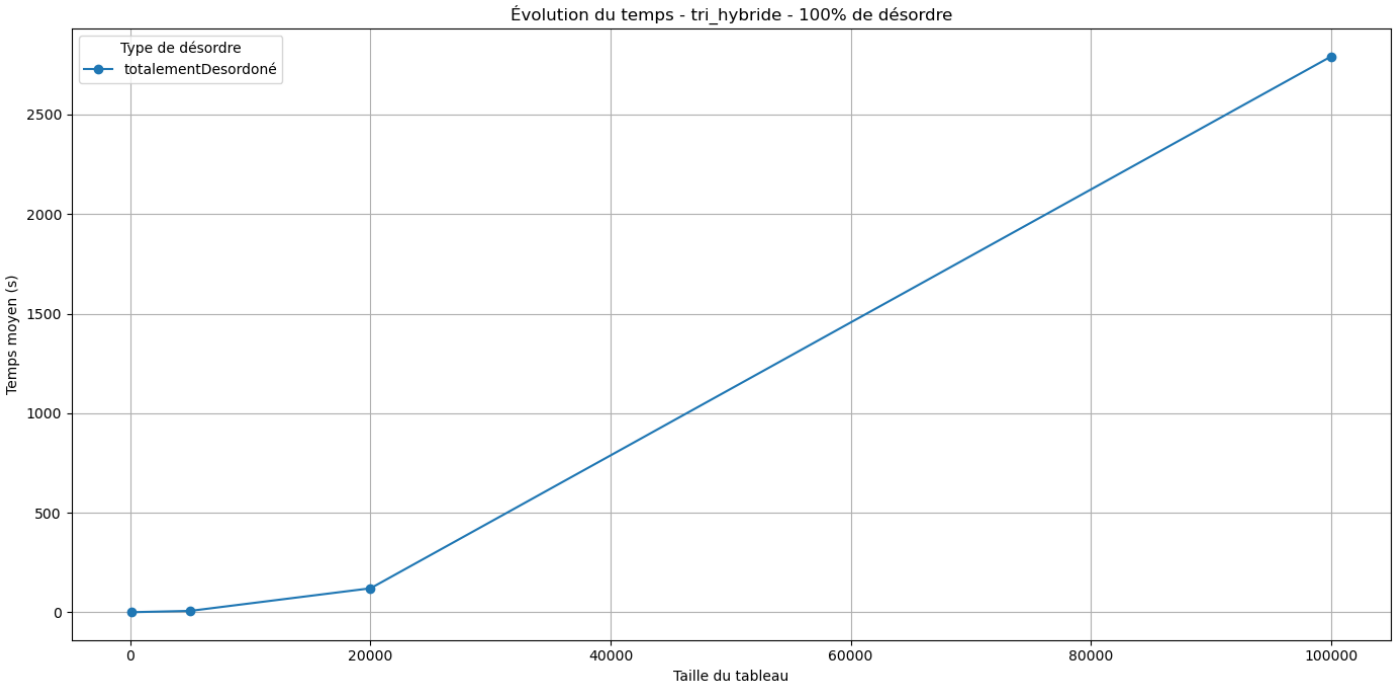


FIGURE 18 – Résultats pour un taux de désordre de 100% avec l'algorithme TimSort

6.5 BubbleSort

BubbleSort compare et échange de proche en proche les éléments adjacents dans un tableau, jusqu'à ce que le tableau soit trié. Sa complexité est de $O(n^2)$ dans la majorité des cas, sauf si le tableau est déjà trié.

6.5.1 Désordre 25%

Avec un désordre léger (25%), BubbleSort profite partiellement de l'ordre déjà existant dans le tableau. On observe de bonnes performances pour le désordre *partielleDebut*, mais une nette augmentation du temps pour les désordres inverses, qui perturbent fortement le tri. Cela s'explique par le fait que les éléments les plus éloignés de leur position finale nécessitent de nombreuses permutation pour remonter.

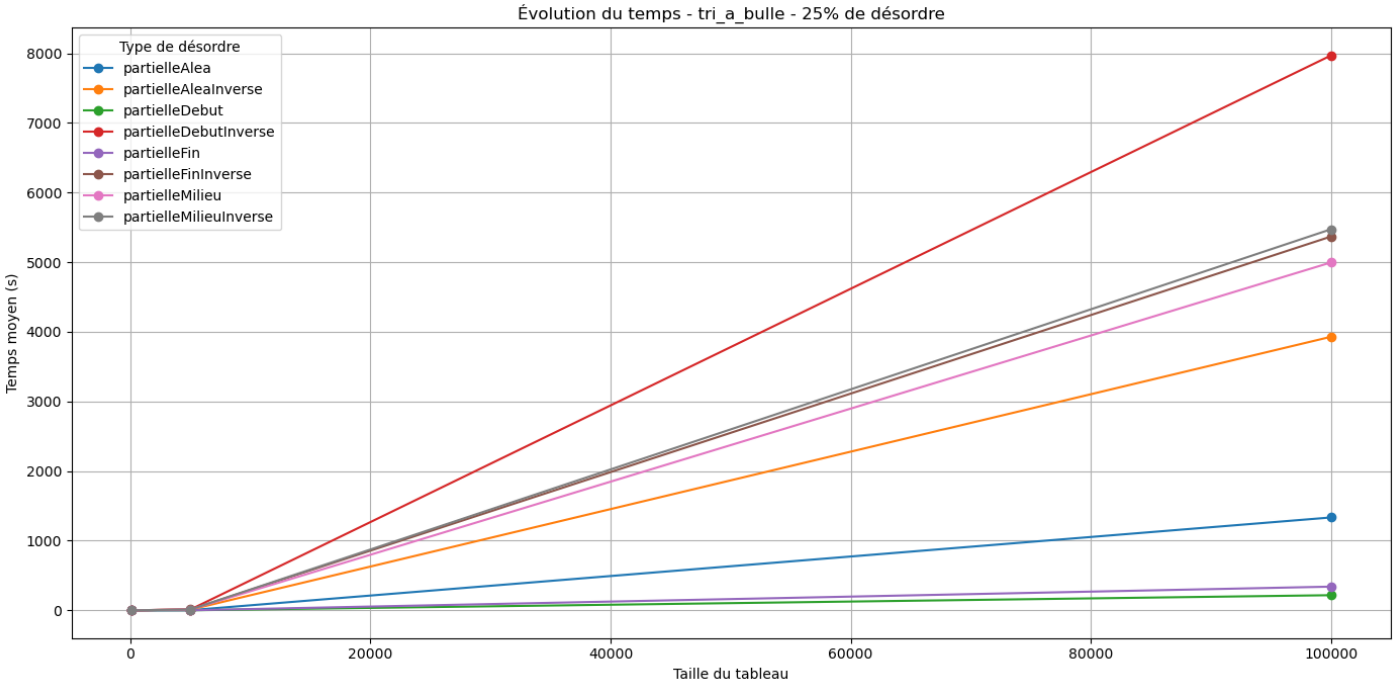


FIGURE 19 – Résultats pour un taux de désordre de 25% avec l'algorithme BubbleSort

6.5.2 Désordre 50%

À 50% de désordre, le comportement de BubbleSort reste cohérent : les performances chutent lentement à mesure que le désordre augmente. Les différences entre les types de désordre deviennent moins prononcées, mais l'algorithme reste globalement plus lent sur des désordres inverse.

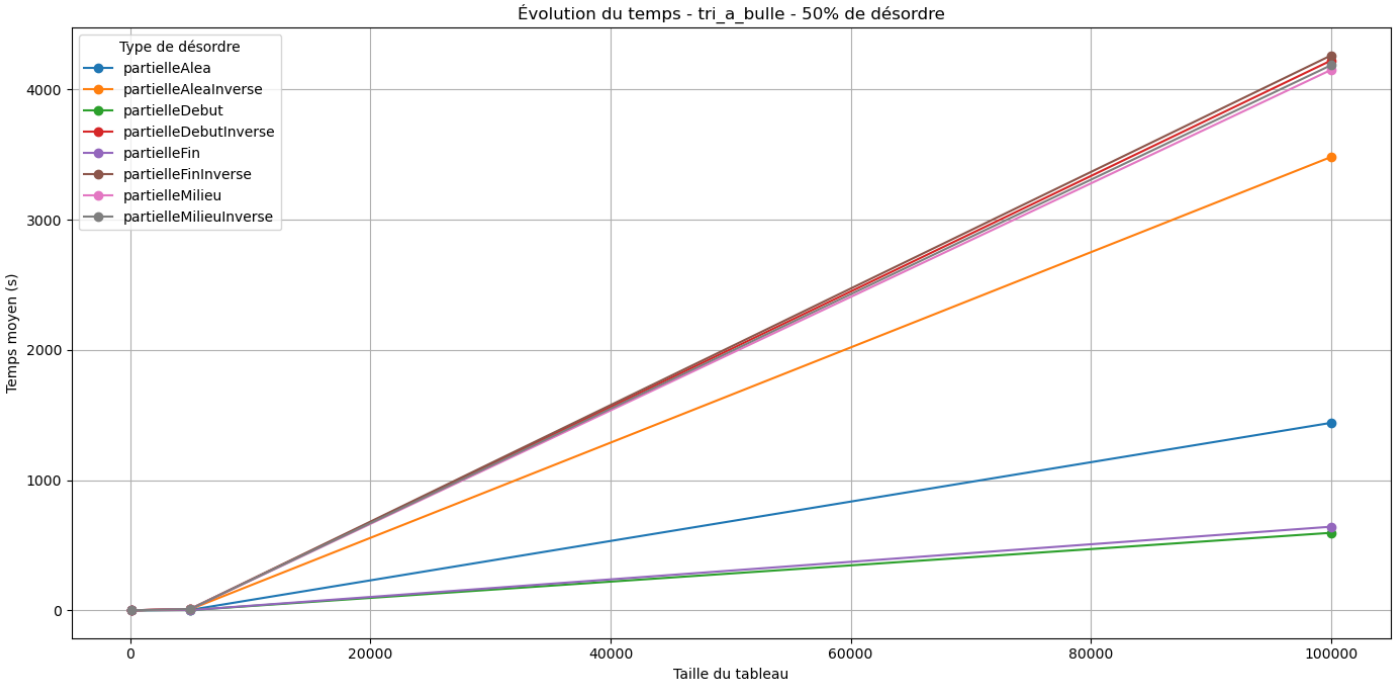


FIGURE 20 – Résultats pour un taux de désordre de 50% avec l'algorithme BubbleSort

6.5.3 Désordre 75%

Lorsque le désordre atteint 75%, BubbleSort montre ses limites : le temps d'exécution continue d'augmenter, cependant il reste toujours plus éfficasse sur des désordre partielle croissant que sur des désordres inverses.

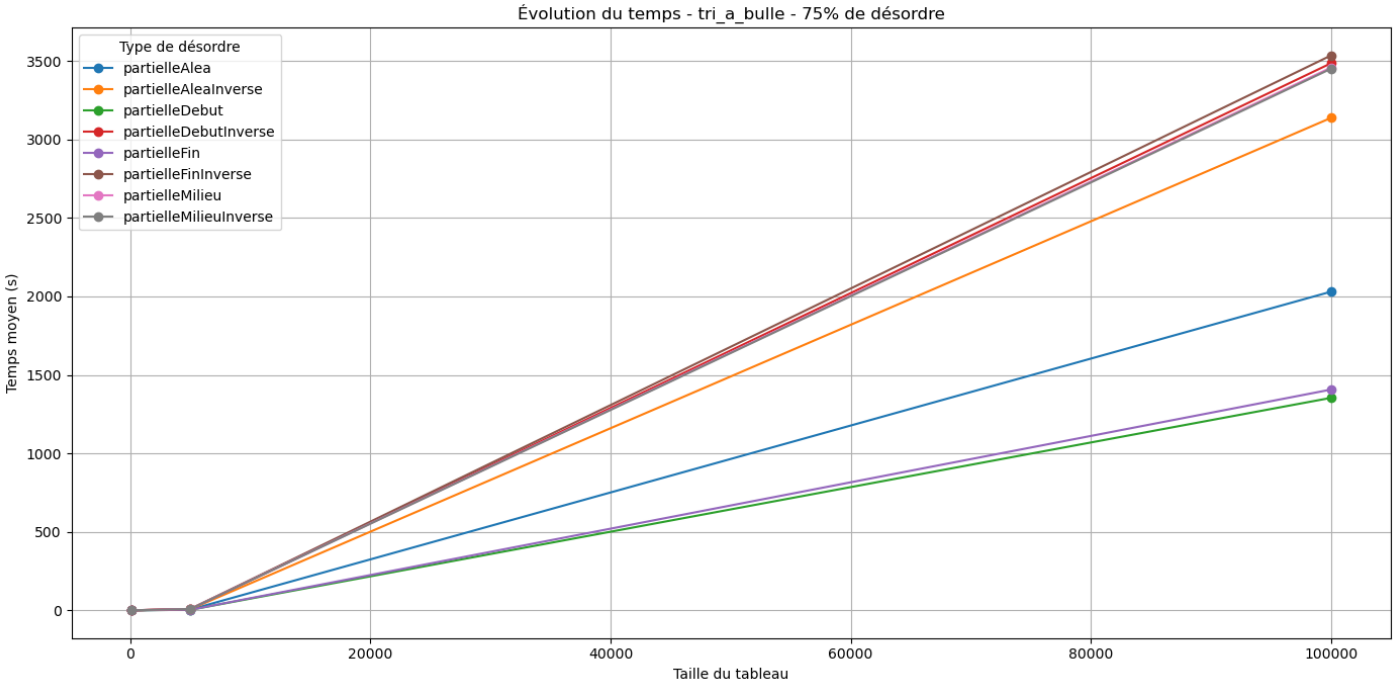


FIGURE 21 – Résultats pour un taux de désordre de 75% avec l'algorithme BubbleSort

6.5.4 Désordre 100%

Avec un désordre total, BubbleSort atteint ses pires performances. Tous les éléments doivent être échangés plusieurs fois, entraînant un nombre de comparaisons et de permutations maximal. Cet algorithme devient rapidement inadapté dès que la taille du tableau ou le désordre augmente.

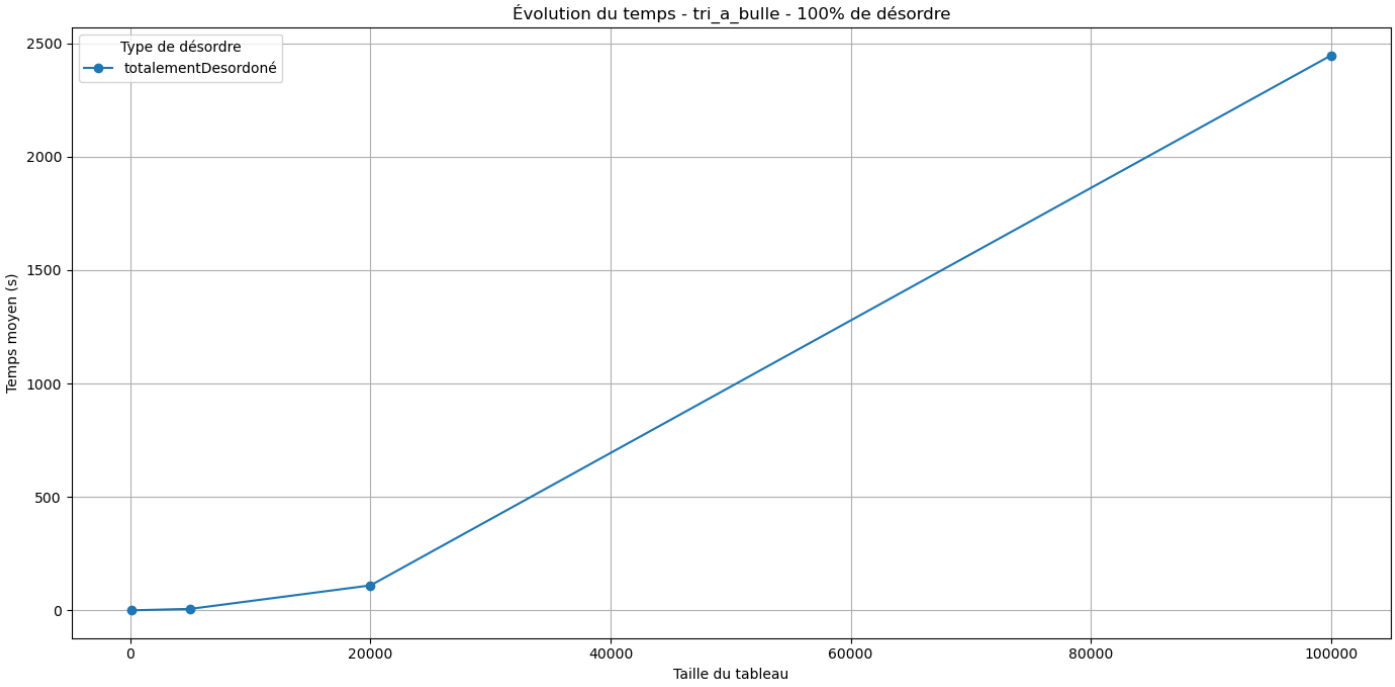


FIGURE 22 – Résultats pour un taux de désordre de 100% avec l'algorithme BubbleSort

6.6 HeapSort

HeapSort est un algorithme de tri fondé sur la structure d'un tas binaire (heap), dans lequel les éléments sont organisés de manière à permettre une extraction rapide de l'élément maximal. Il garantit une complexité temporelle en $O(n \log n)$, quelle que soit la configuration initiale des données. Cette stabilité en fait un excellent choix lorsque la robustesse de performance est recherchée.

6.6.1 Désordre 25%

Avec un taux de désordre de 25%, HeapSort démontre une excellente efficacité. Que ce soit pour un désordre aléatoire ou inversé, les performances restent stables et très proches. Cela s'explique par le fait que l'algorithme commence par reconstruire l'ensemble du tableau sous forme de tas, rendant l'ordre initial presque insignifiant. Les légères variations observées sont donc négligeables, confirmant la résilience de HeapSort aux perturbations de faible intensité.

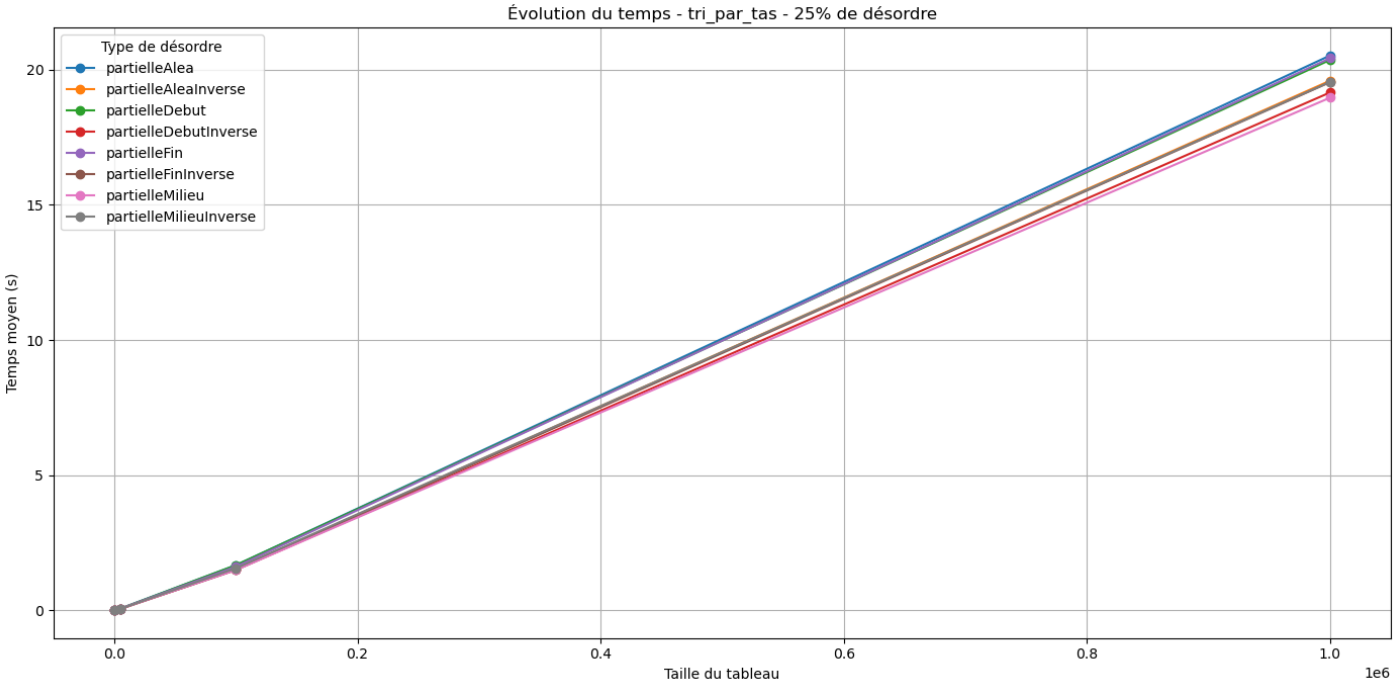


FIGURE 23 – Résultats pour un taux de désordre de 25% avec l'algorithme HeapSort

6.6.2 Désordre 50%

À 50% de désordre, HeapSort conserve des performances constantes et fiables. Les différences entre les types de désordre restent minimes. Cela confirme que la phase de construction du tas neutralise efficacement l'impact du désordre initial. Contrairement à des algorithmes sensibles au positionnement des éléments (comme QuickSort ou InsertionSort), HeapSort conserve une régularité d'exécution qui le distingue.

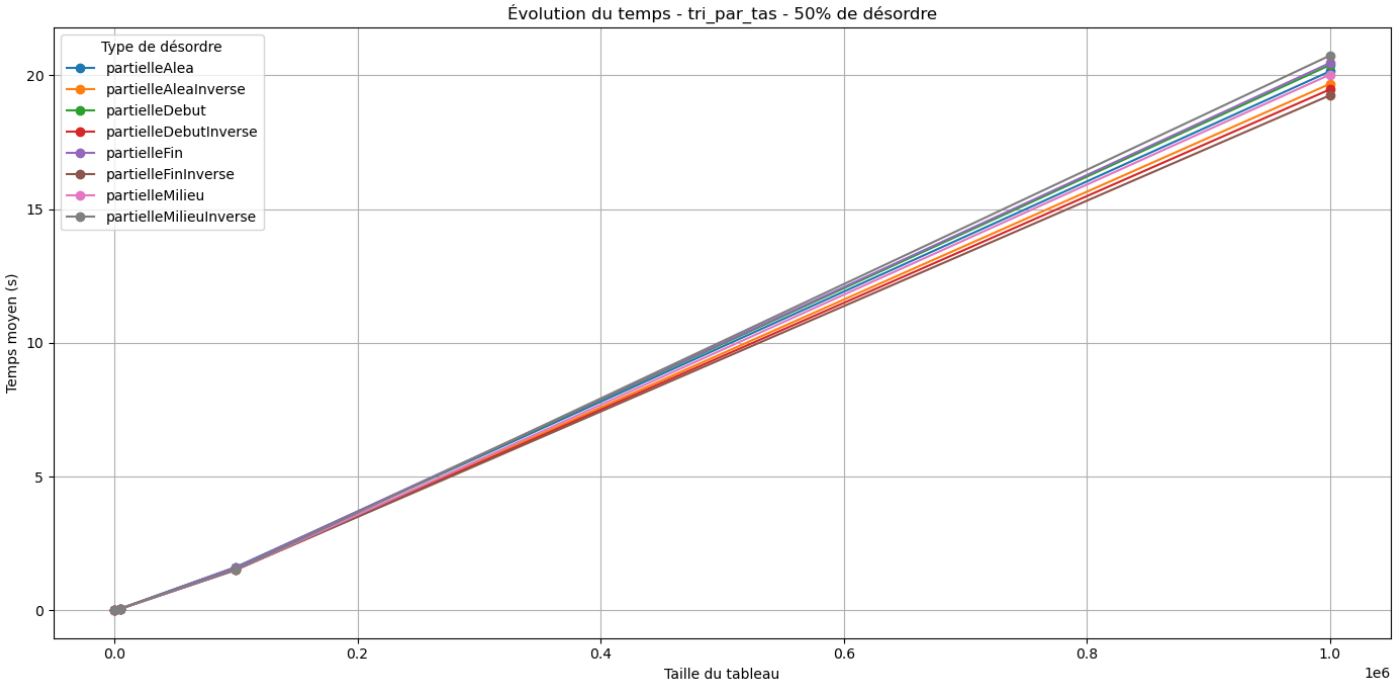


FIGURE 24 – Résultats pour un taux de désordre de 50% avec l'algorithme HeapSort

6.6.3 Désordre 75%

Lorsque le taux de désordre atteint 75%, HeapSort ne subit quasiment aucune dégradation en termes de temps d'exécution. Que les éléments soient partiellement inversés ou simplement mélangés, la nature du désordre n'affecte pas la logique du tri par tas. Le coût en temps reste dominé par les opérations fixes de création du tas et d'extraction, toutes deux réalisées en complexité logarithmique.

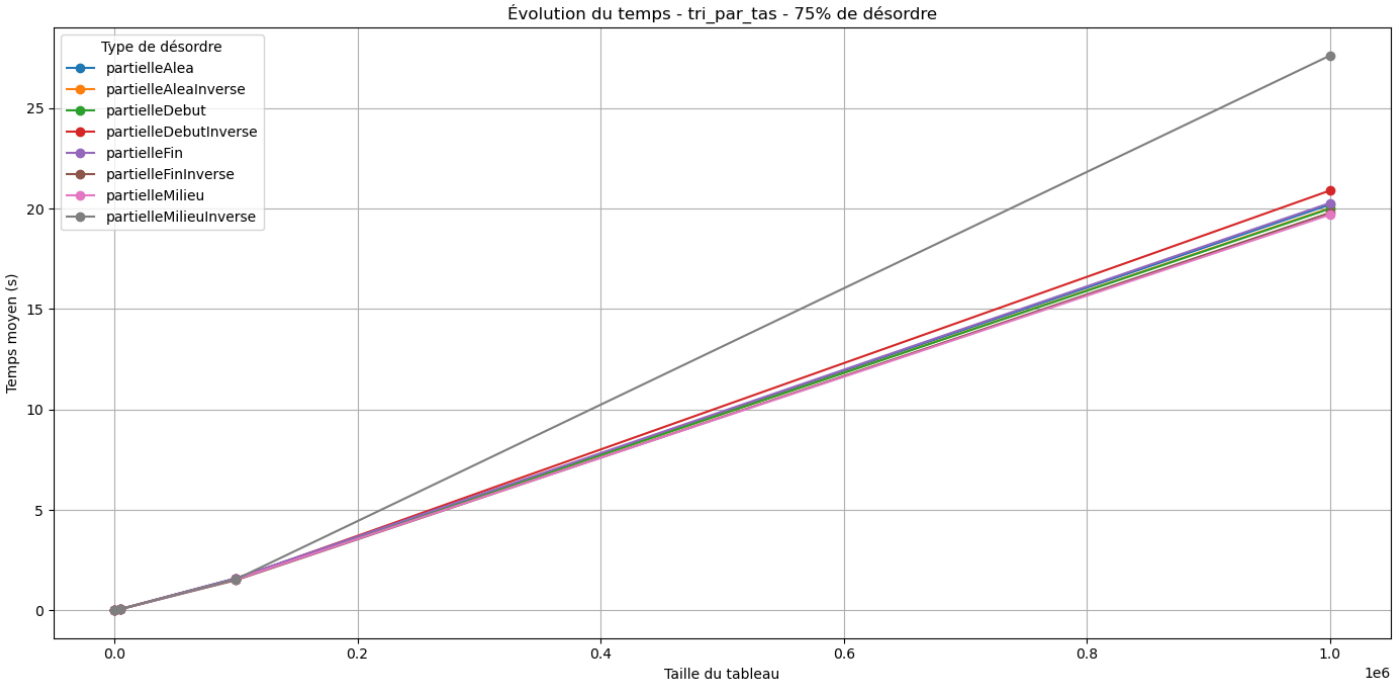


FIGURE 25 – Résultats pour un taux de désordre de 75% avec l'algorithme HeapSort

6.6.4 Désordre 100%

En cas de désordre total, HeapSort brille par sa stabilité. Il maintient une performance quasiment identique aux cas précédents. Comme il ne repose pas sur des comparaisons et que le désordre le l'influence pas, l'algorithme offre une constance impressionnante, même dans les pires conditions. Cette propriété en fait un excellent choix pour les cas où l'on ne connaît pas l'état initial des données.

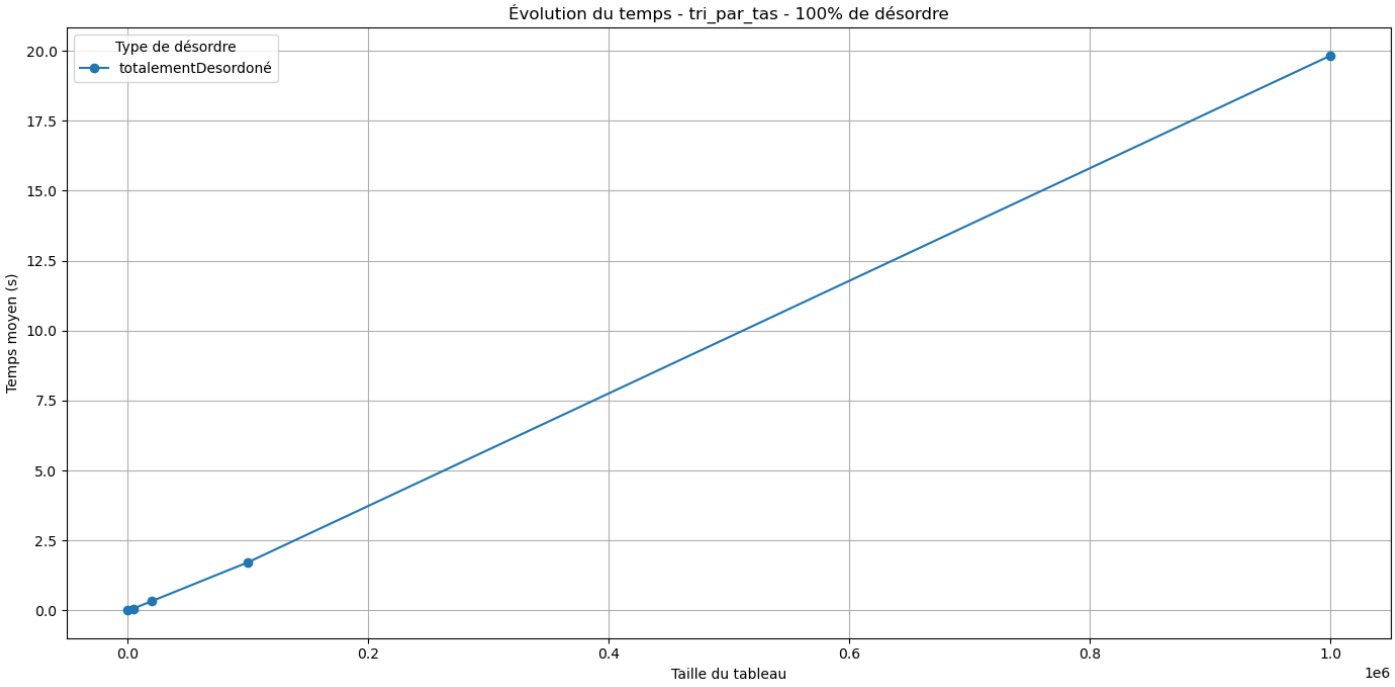


FIGURE 26 – Résultats pour un taux de désordre de 100% avec l'algorithme HeapSort

6.7 MergeSort

MergeSort (ou tri fusion) est un algorithme de tri basé sur la stratégie **diviser pour régner**. Il divise récursivement le tableau en sous-tableaux jusqu'à obtenir des segments unitaires, puis fusionne ces segments de manière ordonnée. Sa complexité est de $O(n \log n)$ dans tous les cas (meilleur, moyen et pire), ce qui le rend très stable et prévisible.Peu import la quantité de désordre il effectue les mêmes opérations selon les tailles

6.7.1 Désordre 25%

Avec 25% de désordre, MergeSort fonctionne très efficacement, et est très stable pour tous nos types de désordre.

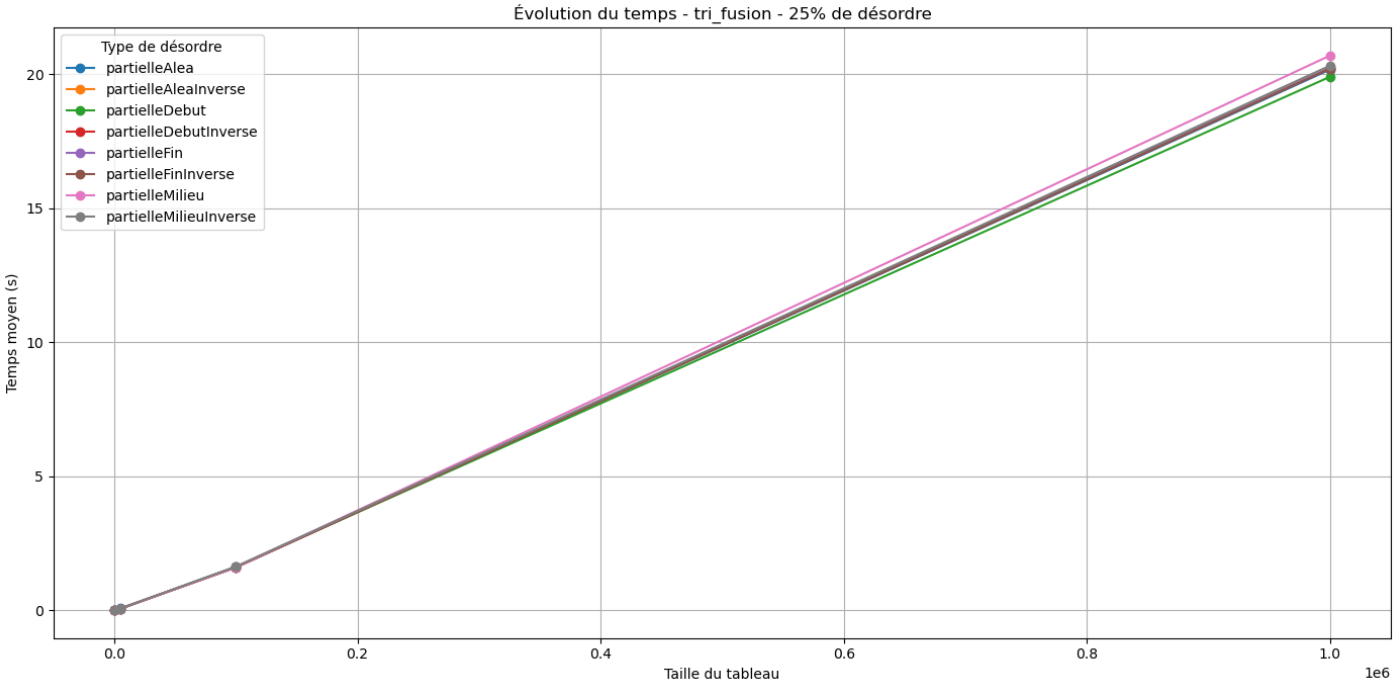


FIGURE 27 – Résultats pour un taux de désordre de 25% avec l'algorithme MergeSort

6.7.2 Désordre 50%

À 50% de désordre, MergeSort conserve sa régularité. La différence entre les types de désordre tend à s'atténuer. L'algorithme divise les segments sans prêter attention à l'ordre initial, et effectue des fusions structurées qui assurent un comportement stable. Le temps d'exécution reste dans le mêmes marges pour tous nos types de désordre.

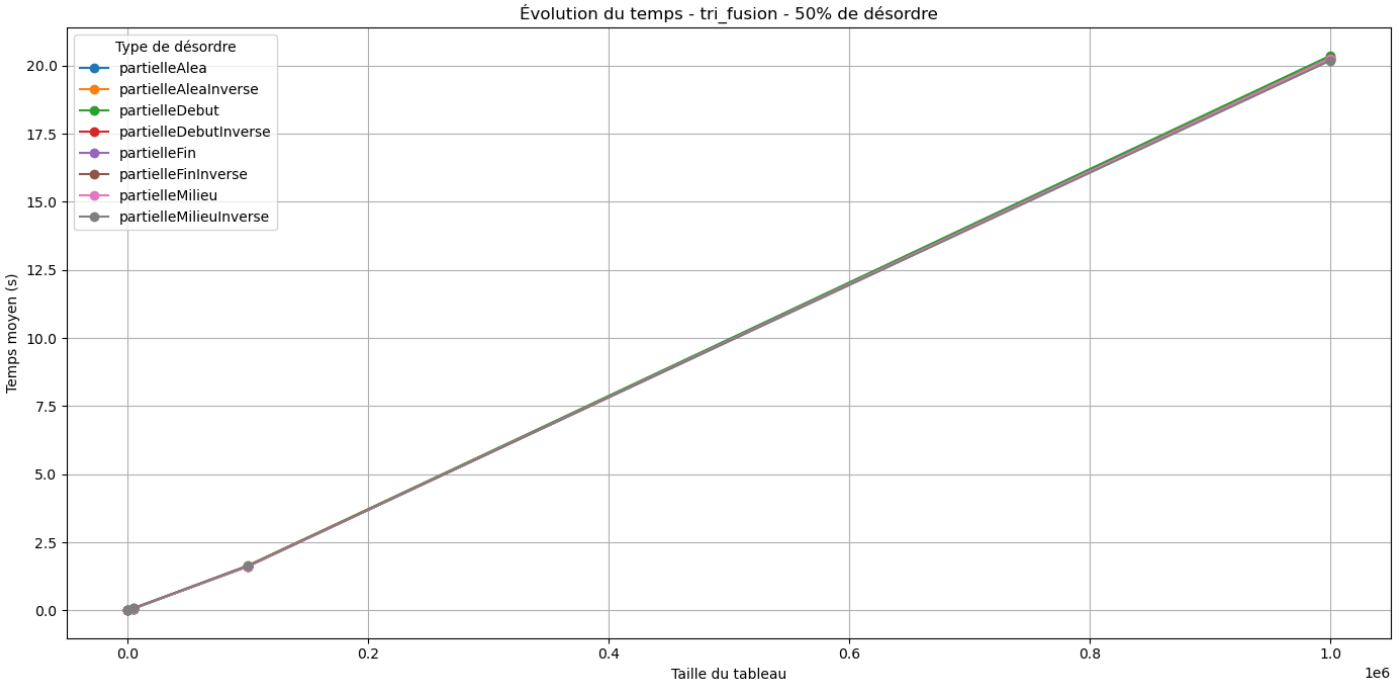


FIGURE 28 – Résultats pour un taux de désordre de 50% avec l'algorithme MergeSort

6.7.3 Désordre 75%

Avec un désordre de 75%, on à les mêmes constats que pour les désordre précédents

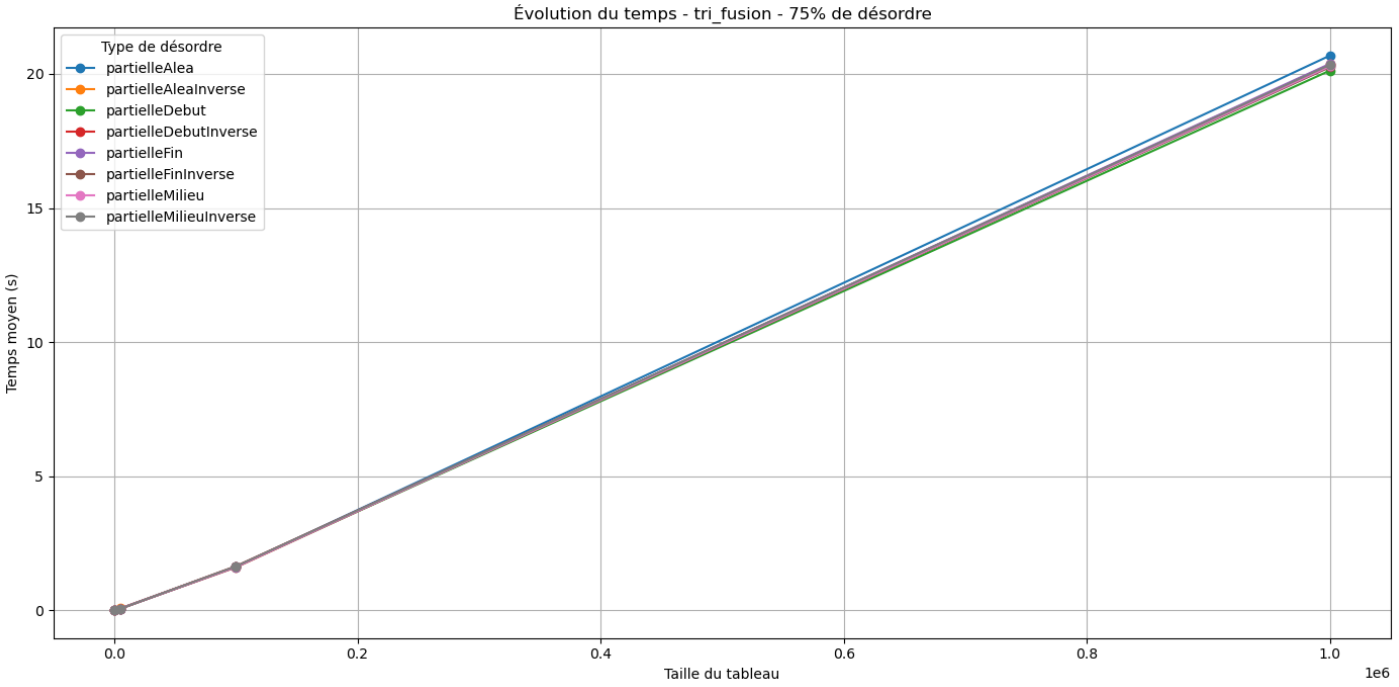


FIGURE 29 – Résultats pour un taux de désordre de 75% avec l'algorithme MergeSort

6.7.4 Désordre 100%

Avec un désordre total, MergeSort conserve une sa stabilité et nous confirme que la quantité de désordre et sa répartition n'influence pas son fonctionnement .

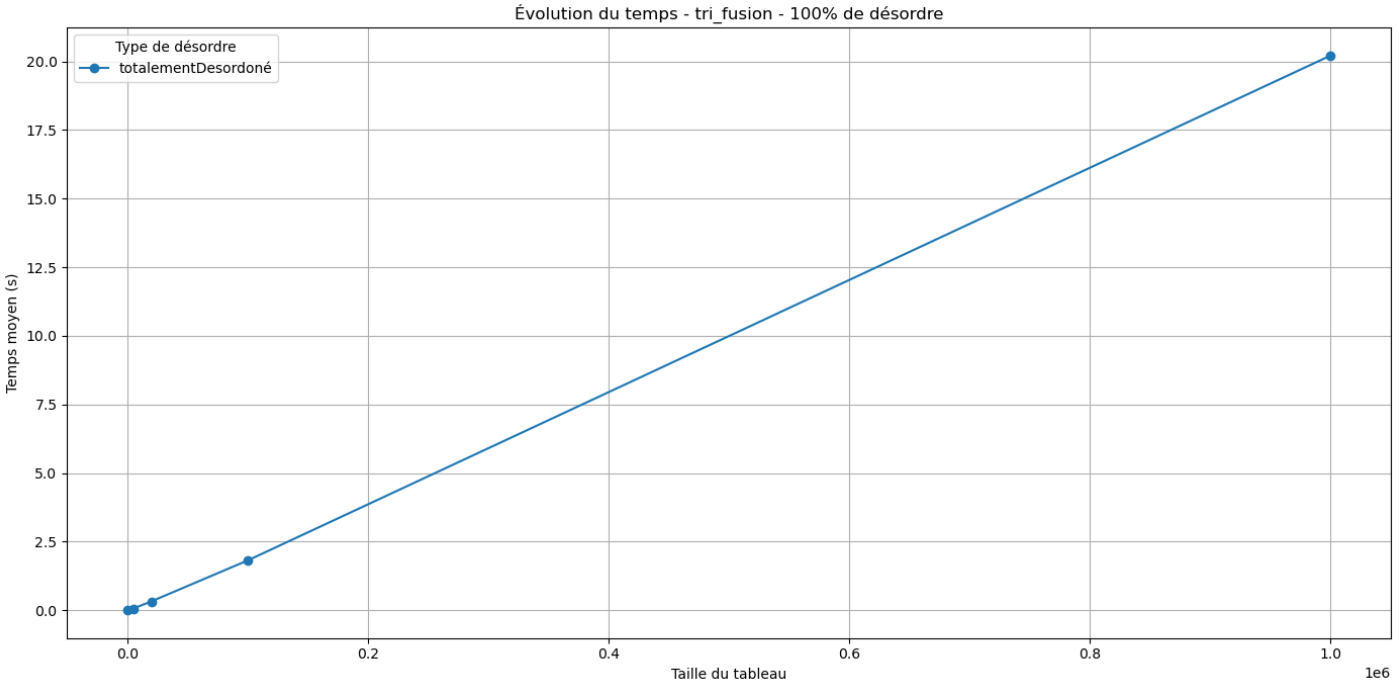


FIGURE 30 – Résultats pour un taux de désordre de 100% avec l'algorithme MergeSort

7 Comparaison générale des algorithmes de tri

Dans cette partie, nous comparerons les différents algorithmes étudiés selon des critères communs : le temps d'exécution, le nombre de comparaisons effectuées, ainsi que le nombre d'accès aux données. L'objectif est d'évaluer leur efficacité relative selon le type et le taux de désordre initial.

7.1 Comparaison en fonction du temps d'exécution

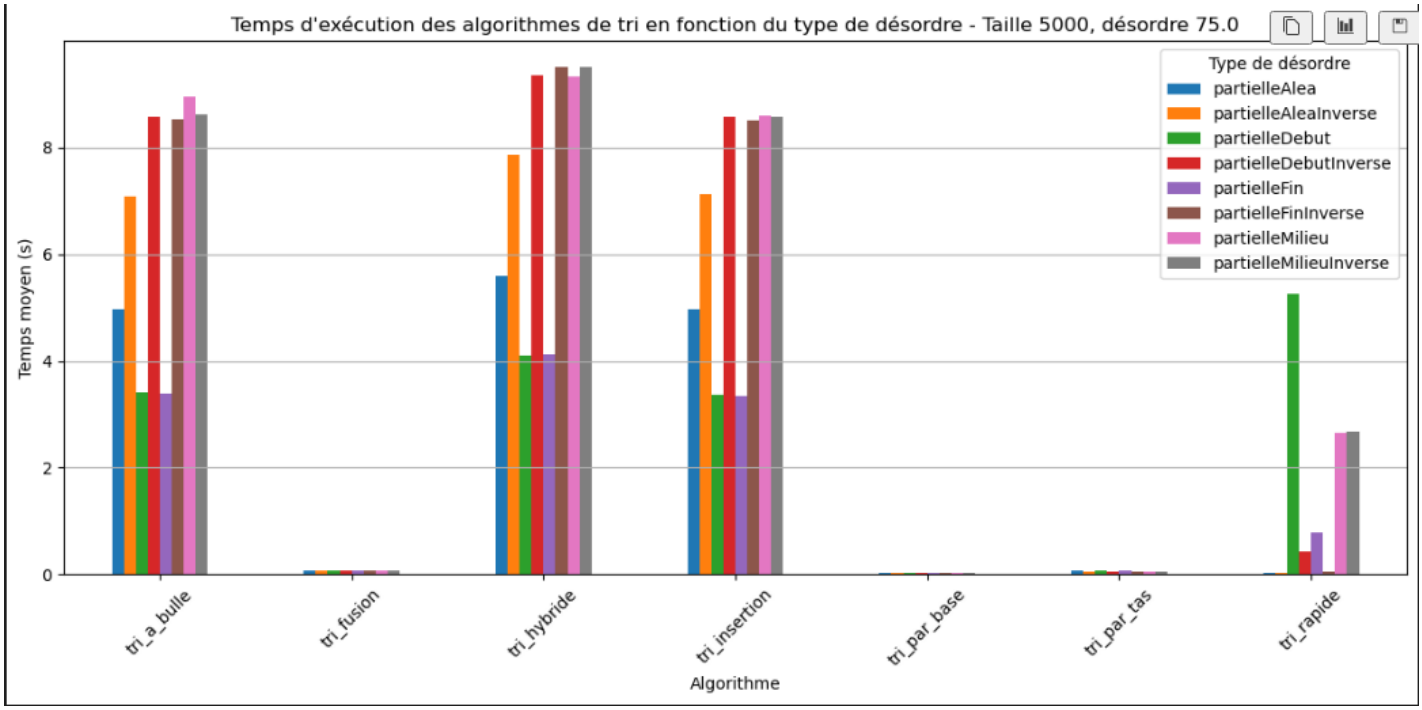


FIGURE 31 – Résultats pour un taux de désordre de 75% et une taille 50000

Sur l'ensemble des tests réalisés, on observe que les algorithmes **TimSort**, **MergeSort** et **HeapSort** offrent les meilleures performances en termes de temps d'exécution, indépendamment du niveau de désordre. En particulier :

— **TimSort** est extrêmement performant pour des tableaux partiellement triés (*partielleAlea*), car il exploite efficacement les séquences déjà ordonnées.

— **HeapSort** reste stable pour tous les types de désordre, même les plus structurés (*partielleAleaInverse*) ou complètement aléatoires.

— **RadixSort** est très rapide, mais sa performance dépend de la nature des données (entiers positifs). Il n'est pas affecté par le désordre.

À l'opposé :

— **BubbleSort** et **InsertionSort** se montrent très lents dès que le taux de désordre dépasse 25%. Ils sont fortement pénalisés par des séquences inversées.

— **QuickSort**, bien que performant dans de nombreux cas, peut être ralenti si le pivot est mal choisi dans une structure de désordre inversée.

7.2 Comparaison selon le nombre de comparaisons

TABLE 2 – Nombre moyen de comparaisons par algorithme et type de désordre

Algorithme	partielleAlea	partielleAleaInverse	partielleDebut	partielleDebutInverse	partielleFin	partielleFinInverse	partielleMilieu	partielleMilieuInverse	totalementDesordoné
tri_a_bulle	1670817483	1670817483	1670817483	1670817483	1670817483	1670817483	1670817483	1670817483	1303110612
tri_fusion	5015114	5000993	3833248	3828477	3856319	3805431	3817044	3816980	4105355
tri_hybride	475148330	1195435846	243795316	1426774159	244034053	1427053140	1427431888	1427476176	650459054
tri_insertion	477347828	1195400752	244268005	1427643548	243533331	1427619600	1427172059	1427289320	650755070
tri_par_base	276274	276274	276274	276274	276274	276274	276274	276274	225019
tri_par_tas	7396518	7074432	7460989	7002348	7398023	6908982	6975427	6975202	5844435
tri_rapide	44022	39911	4444184	1853470	1851656	753189	4445437	4443616	5752872

Le nombre de comparaisons varie fortement selon la stratégie de tri :

— **BubbleSort** et **InsertionSort** effectuent un grand nombre de comparaisons, surtout en présence de désordre inversé.

— **QuickSort** et **MergeSort** ont une complexité en $O(n \log n)$ en moyenne, avec un nombre modéré de comparaisons, sauf en pire cas pour QuickSort.

— **HeapSort** maintient un nombre de comparaisons relativement constant grâce à sa structure de tas.

— **RadixSort**, étant non comparatif, se démarque en n'effectuant aucune comparaison directe entre les éléments.

7.3 Comparaison selon les accès mémoire

Concernant les accès mémoire :

— **MergeSort** nécessite une mémoire auxiliaire pour les phases de fusion, ce qui augmente ses accès.

— **HeapSort** effectue de nombreux accès pour maintenir la propriété de tas, mais sans allocation supplémentaire.

— **TimSort** utilise également une mémoire intermédiaire pour les runs fusionnés, mais son efficacité sur les données partiellement triées compense cela.

— **BubbleSort** et **InsertionSort** effectuent beaucoup d'échanges en place, ce qui multiplie les accès, surtout en cas de désordre élevé.

8 Conclusion

L'efficacité des algorithmes de tri dépend fortement du niveau de désordre des données, que ce soit en termes de quantité (pourcentage de désordre) ou de répartition (position des éléments désordonnés : début, milieu, fin, etc.). Certains algorithmes sont très sensibles à ces facteurs, tandis que d'autres restent relativement stables.

8.1 Influence de la quantité de désordre sur l'efficacité des algorithmes de tri

Algorithmes sensibles (comme *BubbleSort* ou *InsertionSort*) deviennent très lents lorsque la quantité de désordre augmente.

— **Exemple :** *BubbleSort* est rapide si le tableau est trié ou presque trié, mais devient extrêmement inefficace avec 75% ou 100% de désordre.

Algorithmes robustes (comme *HeapSort*, *MergeSort*, *RadixSort*) maintiennent des performances stables, quelle que soit la quantité de désordre.

— **Exemple :** *HeapSort* effectue toujours les mêmes opérations, peu importe si les données sont triées, inversées ou mélangées.

QuickSort est performant dans la majorité des cas, mais peut devenir lent si le pivot est mal choisi dans un tableau presque trié ou totalement inversé.

— **Exemple :** *QuickSort* peut passer de $\mathcal{O}(n \log n)$ à $\mathcal{O}(n^2)$ si les données sont triées et que le pivot est toujours choisi comme le dernier élément.

8.2 Influence de la répartition du désordre

Les **algorithmes adaptatifs** comme *TimSort* ou *InsertionSort* peuvent tirer avantage d'un désordre localisé (par exemple uniquement en fin de tableau).

— **Exemple :** *TimSort* exploite les *runs* (séquences déjà triées) et est donc très rapide si le désordre est concentré dans une petite partie du tableau.

Le **désordre inversé** (`partielleAleaInverse`, `partielleFinInverse`, etc.) est souvent le pire cas pour les algorithmes naïfs, car les éléments doivent être entièrement déplacés.

— **Exemple :** *InsertionSort* est efficace avec des valeurs presque triées, mais devient très lent si les grands éléments sont placés au début et doivent être "descendus" dans le tableau.

Les algorithmes comme *RadixSort* ou *HeapSort* sont **peu affectés par la localisation** du désordre, car ils ne reposent pas sur des comparaisons entre éléments adjacents mais sur des structures indépendantes de l'ordre local.

Alors que le taux et la structure du désordre dégradent les performances des algorithmes de tri naïfs, certains algorithmes plus avancés y sont peu sensibles. C'est pourquoi bien choisir l'algorithme de tri en fonction de la nature des données peut considérablement améliorer les performances, notamment dans les applications en temps réel ou sur de grands volumes de données.