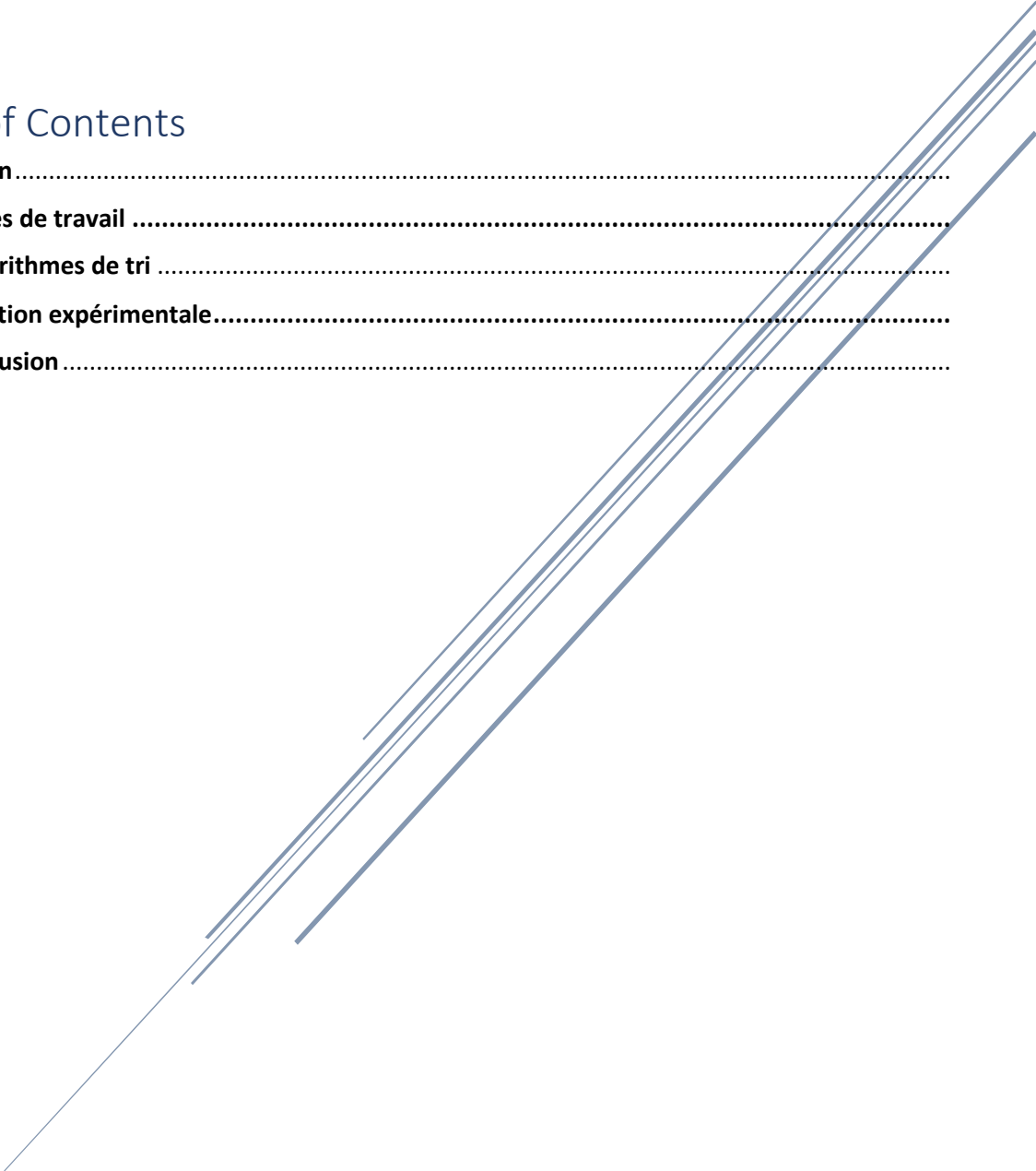


RAPPORT : Les TRIS

Table of Contents

introduction	
objectives de travail	
Les algorithmes de tri	
Évaluation expérimentale	
conclusion	



RAPPORT : Les TRIS

. INTRODUCTION

Les algorithmes de tri sont des outils essentiels en informatique, permettant d'organiser des données de manière ordonnée. Ils sont largement utilisés dans divers domaines, comme la recherche, les bases de données, et les opérations sur les données. Ce rapport explore plusieurs algorithmes de tri, leurs complexités théoriques, et évalue leur performance sur différentes tailles de vecteurs.

Dans un monde où les volumes de données augmentent exponentiellement, la performance des méthodes de tri devient un enjeu crucial pour l'efficacité des systèmes informatiques. La problématique que nous abordons est la suivante : **Comment choisir et optimiser un algorithme de tri ?** Chaque algorithme de tri possède ses propres caractéristiques, avantages et limites. Le choix du bon algorithme dépend de plusieurs paramètres essentiels : le volume et la nature des données, les contraintes de performance, la stabilité du tri, et la complexité algorithmique.

Nous allons étudier et comparer plusieurs méthodes de tri, notamment le tri par sélection, le tri à bulles, le tri par insertion, le tri rapide, le tri fusion et le tri peigne. Ces algorithmes seront évalués en fonction de leurs performances théoriques et expérimentales, avec un accent particulier sur les enjeux de l'analyse des complexités algorithmiques.

Ce rapport explore plusieurs algorithmes de tri, leurs complexités théoriques, et évalue leur performance sur différentes tailles de vecteurs.

OBJECTIFS DU TRAVAIL

Objectif principal : Implémenter et comparer différents algorithmes de tri.

OBJECTIFS SPÉCIFIQUES :

- Comprendre les mécanismes internes des algorithmes de tri
- Analyser les performances théoriques et expérimentales
- Développer des compétences en programmation en langage C
- Maîtriser l'analyse de complexité algorithmique.

.C. DESCRIPTION DÉTAILLÉE DES ALGORITHMES

TRI PAR SÉLECTION :

Le tri par sélection sélectionne l'élément minimum de la liste et le place à sa position correcte à chaque itération.

- . Parcourir la liste et trouver l'élément le plus petit.
- . Échanger cet élément avec le premier élément non trié.
- . Répéter le processus pour les éléments restants.

Pseudo-code :

```
Pour i = 0 à n-1 faire
  min_index = i
  Pour j = i+1 à n-1 faire
    Si A[j] < A[min_index] alors
      min_index = j
  Fin Si
Fin Pour
échanger A[i] et A[min_index]
Fin Pour
```

Complexités :

- Meilleur cas : $O(n^2)$
- Cas moyen : $O(n^2)$
- Pire cas : $O(n^2)$

Avantages et inconvénients :

- Simple à comprendre et à implémenter
- Inefficace pour les grandes listes

Cas d'utilisation recommandés : Listes petites à moyennes, démonstrations éducatives

TRI À BULLES :

Principe de l'algorithme : Le tri à bulles compare et échange des éléments adjacents si nécessaire pour les placer dans le bon ordre.

.comparer les paires consécutives d'éléments

. Échanger les éléments par paire tels ce qui est plus petit est le premier

. Une fois arrivé en fin de liste, recommencer à nouveau

. Arrêter quand il n'y a plus d'échanges ont été faites

Pseudo-code :

```
Pour i = 0 à n-1 faire
  Pour j = 0 à n-i-1 faire
    Si A[j] > A[j+1] alors
      échanger A[j] et A[j+1]
    Fin Si
  Fin Pour
Fin Pour
```

Complexités :

- Meilleur cas : $O(n)$
- Cas moyen : $O(n^2)$
- Pire cas : $O(n^2)$

Avantages et inconvénients :

- **Avantages** : Simple et facile à implémenter.
- **Inconvénients** : Très inefficace pour les grandes listes.

Cas d'utilisation recommandés : Utilisé principalement à des fins éducatives.

TRI PAR INSERTION :

Principe de l'algorithme : Construire la liste triée un élément à la fois en insérant chaque nouvel élément dans sa position correcte.

1. Parcourir la liste et sélectionner un élément.
2. Insérer cet élément dans sa position correcte dans la partie triée de la liste.
3. Répéter le processus pour tous les éléments.
4. **Pseudo-code** :

```
Pour i = 1 à n-1 faire
  clé = A[i]
  j = i - 1
  Tant que j >= 0 et A[j] > clé faire
    A[j + 1] = A[j]
    j = j - 1
  Fin Tant que
  A[j + 1] = clé
Fin Pour
```

Complexités :

- Meilleur cas : $O(n)$
- Cas moyen : $O(n^2)$
- Pire cas : $O(n^2)$

Avantages et inconvénients :

- **Avantages** : Efficace pour les petites listes ou les listes presque triées.
- **Inconvénients** : Inefficace pour les grandes listes non triées.

Cas d'utilisation recommandés : Particulièrement utile pour les listes petites ou presque triées.

TRI RAPIDE (QUICK SORT) :

Principe de l'algorithme : Utiliser un pivot pour partitionner la liste en sous-listes, puis trier chaque sous-liste de manière récursive.

Algorithme pas à pas :

1. Choisir un pivot.
2. Réorganiser les éléments pour que ceux qui sont plus petits que le pivot soient à gauche et ceux qui sont plus grands soient à droite.

3. Appliquer récursivement le tri aux sous-listes.

Pseudo-code :

```
Fonction QuickSort(A, bas, haut) :  
  Si bas < haut alors  
    pivot_index = Partition(A, bas, haut)  
    QuickSort(A, bas, pivot_index - 1)  
    QuickSort(A, pivot_index + 1, haut)  
  Fin Si  
Fin Fonction  
  
Fonction Partition(A, bas, haut) :  
  pivot = A[haut]  
  i = bas - 1  
  Pour j = bas à haut - 1 faire  
    Si A[j] <= pivot alors  
      i = i + 1  
      échanger A[i] et A[j]  
    Fin Si  
  Fin Pour  
  échanger A[i + 1] et A[haut]  
  retourner i + 1  
Fin Fonction
```

Complexités :

- Meilleur cas : $O(n \log n)$
- Cas moyen : $O(n \log n)$
- Pire cas : $O(n^2)$

Avantages et inconvénients :

- **Avantages** : Très rapide en moyenne et efficace pour les grandes listes.
- **Inconvénients** : Peut-être inefficace dans le pire des cas sans optimisations.

Cas d'utilisation recommandés : Utilisé couramment en pratique pour sa rapidité.

TRI FUSION :

Principe de l'algorithme : Diviser la liste en sous-listes, trier chaque sous-liste, puis fusionner les sous-listes triées pour obtenir une liste triée.

Algorithme pas à pas :

1. Diviser la liste en deux moitiés.
2. Appliquer récursivement le tri fusion aux moitiés.

3. Fusionner les moitiés triées.

Pseudo-code :

Fonction MergeSort(A, l, r) :

Si $l < r$ alors

milieu = $(l + r) / 2$

MergeSort(A, l, milieu)

MergeSort(A, milieu + 1, r)

Fusionner(A, l, milieu, r)

Fin Si

Fin Fonction

Fonction Fusionner(A, l, m, r) :

$n1 = m - l + 1$

$n2 = r - m$

L = nouvelle liste de taille $n1$

R = nouvelle liste de taille $n2$

Pour $i = 0$ à $n1 - 1$ faire

$L[i] = A[l + i]$

Fin Pour

Pour $j = 0$ à $n2 - 1$ faire

$R[j] = A[m + 1 + j]$

Fin Pour

$i = 0$

$j = 0$

$k = l$

Tant que $i < n1$ et $j < n2$ faire

Si $L[i] \leq R[j]$ alors

$A[k] = L[i]$

$i = i + 1$

Sinon

$A[k] = R[j]$

$j = j + 1$

Fin Si

$k = k + 1$

Fin Tant que

Tant que $i < n1$ faire

$A[k] = L[i]$

$i = i + 1$

$k = k + 1$

Fin Tant que

Tant que $j < n2$ faire

$A[k] = R[j]$

$j = j + 1$

$k = k + 1$

Fin Tant que

Complexités

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

Avantages et inconvénients

- **Avantages** :
 - Complexité temporelle garantie de $O(n \log n)$
 - Algorithme stable.
 - Bien adapté aux grandes listes.
- **Inconvénients** :
 - Utilise de la mémoire supplémentaire ($O(n)$) pour les listes temporaires.
 - Moins efficace que certains algorithmes de tri en place pour les petites listes.

Cas d'utilisation recommandés

Le tri fusion est recommandé pour les applications nécessitant une complexité temporelle garantie et stable, ainsi que pour les grandes listes de données. Il est également utile lorsque la stabilité du tri est importante.

TRI PEIGNE (COMB SORT)

Principe de l'algorithme

Le tri peigne (Comb Sort) est une amélioration du tri à bulles visant à éliminer les tortues, c'est-à-dire les petits éléments situés en fin de liste qui ralentissent le tri. L'idée principale est de comparer et échanger des éléments situés à une certaine distance (gap) les uns des autres, cette distance diminuant progressivement.

Algorithme pas à pas

1. Initialiser le gap (distance) avec la longueur de la liste.
2. Réduire le gap selon un facteur de réduction (généralement 1.3).
3. Comparer et échanger les éléments situés à gap distance.
4. Répéter jusqu'à ce que le gap soit réduit à 1 et que la liste soit triée.

Pseudo-code :

```
Fonction CombSort(A) :  
  n = longueur(A)  
  gap = n  
  shrink = 1.3  
  sorted = faux  
  
  Tant que sorted = faux faire  
    gap = gap / shrink  
    Si gap < 1 alors  
      gap = 1  
  
    sorted = vrai  
    Pour i = 0 à n - gap faire  
      Si A[i] > A[i + gap] alors  
        échanger A[i] et A[i + gap]  
        sorted = faux  
      Fin Si  
    Fin Pour  
  Fin Tant que  
Fin Fonction
```

Complexités

- Meilleur cas : $O(n \log n)$
- Cas moyen : $O(n^2)$

- **Pire cas** : $O(n^2)$

Avantages et inconvénients

- **Avantages** :
 - Plus rapide que le tri à bulles en moyenne.
 - Facile à implémenter.
- **Inconvénients** :
 - Pas stable.
 - Complexité pire des cas similaire à celle du tri à bulles ($O(n^2)$).

Cas d'utilisation recommandés

Le tri peigne est recommandé pour les listes où les avantages du tri à bulles sont désirés mais avec une performance améliorée. Il est utile pour des applications où un algorithme simple et efficace est nécessaire, et où la stabilité n'est pas une contrainte.

..D. ÉVALUATION EXPÉRIMENTALE

Pour évaluer les performances des algorithmes de tri, nous avons sélectionné trois algorithmes de complexités différentes :

1. **Tri à Bulles (Bubble Sort)**
2. **Tri Rapide (Quick Sort)**
3. **Tri par insertion (insertion Sort)**

Tableau Comparatif des Performances :

Tri a Bulles				
Size n	Nb permutations	Nb comparisons	Nb iterations	Temps(s)
10	18	35	5	6.20
20	87	180	15	6.542
50	619	1197	42	9.138
100	2609	4779	81	7.859
200	9990	19795	185	12.230
500	59776	124399	473	36.640
1000	248594	498939	966	96.759

Tri rapide				
Size n	Nb permutations	Nb comparisons	Nb iterations	Temps
10	25	25	7	7.776
20	43	65	13	6.443
50	190	253	25	8.683
100	336	586	68	8.116
200	938	1879	151	7.058
500	2118	4837	555	17.027
1000	9503	21364	1455	49.714

Tri par Insertion				
Size n	Nb permutations	Nb comparisons	Nb iterations	temps
10	27	27	9	10.635
20	106	106	19	9.657
50	668	668	49	10.842
100	2708	2708	99	15.283
200	10189	10189	199	17.309
500	60275	60275	499	40.381
1000	249593	249593	999	113.943

CONCLUSION

Cette étude comparative des algorithmes de tri nous a permis de mettre en évidence :

- Les différences significatives de performances selon la taille et l'état initial du tableau
- L'importance du choix de l'algorithme en fonction du contexte
- Les compromis entre complexité théorique et performances pratiques

Le tri rapide s'avère généralement le plus performant pour les grandes collections, tandis que les tris par insertion et à bulles restent efficaces pour de petits tableaux.