

# RAPPORT PROJET

PROG C

NEGOVANOVIC Adrien | BENTOUNSI TOUHAMMI Alexandre



# ARCHITECTURE

## **README.TXT**

Dans ce fichier texte, vous retrouverez toutes les indications pour lancer et compiler correctement els programmes.

## **GERERMEM.C**

C'est un fichier qui contient les fonctions de gestion mémoire personnalisées (myMalloc, myRealloc, myFree) permettant de mesurer précisément l'espace mémoire alloué et libéré durant l'exécution du programme.

## **GRAPH.PY**

Ce fichier python est utilisé pour analyser les fichiers de performances générés par le programme C et produire des graphiques comparant le temps d'exécution et la mémoire utilisée pour chaque algorithme.

## **PROJET.C**

Ce fichier regroupe les différents algorithmes de comptage des mots ainsi que le programme principal permettant de lancer les tests et les comparaisons.



# ALGORITHMES

## ALGORITHME 1

Cet algorithme est le plus simple. Il utilise un tableau non trié pour stocker les mots et leur nombre d'occurrences.

À chaque mot lu dans le fichier, le programme parcourt le tableau pour vérifier si le mot existe déjà.

S'il est trouvé, le compteur est augmenté, sinon le mot est ajouté au tableau.

Cette méthode entraîne une complexité en  $O(n^2)$  car le tableau peut être parcouru entièrement pour chaque nouveau mot.

En revanche, elle consomme peu de mémoire, car une seule structure de données est utilisée.

## ALGORITHME 2

Ce deuxième algorithme fonctionne avec un tableau trié.

Pour chaque mot, une recherche dichotomique est effectuée afin de savoir s'il est déjà présent dans le tableau.

La recherche est plus rapide que dans l'algorithme 1, mais l'insertion d'un nouveau mot peut nécessiter de décaler des éléments pour garder le tableau trié.

La complexité moyenne est donc de  $O(n \log n)$ .

La consommation mémoire reste celle du premier algorithme puisqu'on utilise toujours un tableau.

## ALGORITHME 3

Le troisième algorithme utilise une table de hachage pour compter les occurrences des mots.

Chaque mot est associé à une case de la table grâce à une fonction de hachage.

En cas de collision, les mots sont stockés dans une liste chaînée.

Cela permet de retrouver et de mettre à jour les mots rapidement.

Grâce à cette structure, le temps moyen d'exécution est en  $O(n)$ , ce qui rend cet algorithme plus efficace pour les fichiers de grande taille.

Cependant, cette méthode utilise plus de mémoire que les deux précédentes.



# PERFORMANCES

## ALGORITHME 1

Cet algorithme présente une complexité en  $O(n^2)$ , ce qui le rend peu performant lorsque la taille du fichier augmente.

Il peut tout de même être rapide sur des fichiers de petite taille, car le nombre de comparaisons reste limité.

Pour ce qui est de la mémoire, cet algorithme utilise uniquement un tableau pour stocker les mots et leurs occurrences.

## ALGORITHME 2

L'algorithme 2 possède une complexité moyenne en  $O(n \log n)$ , ce qui améliore nettement les performances par rapport à l'algorithme naïf.

La recherche dichotomique permet de retrouver plus rapidement les mots dans le tableau.

La consommation mémoire reste similaire à celle de l'algorithme 1, car la structure de données utilisée est toujours un tableau.

Cet algorithme représente donc un bon compromis entre le temps d'exécution et l'utilisation de la mémoire, mais ses performances restent limitées pour des fichiers très volumineux.

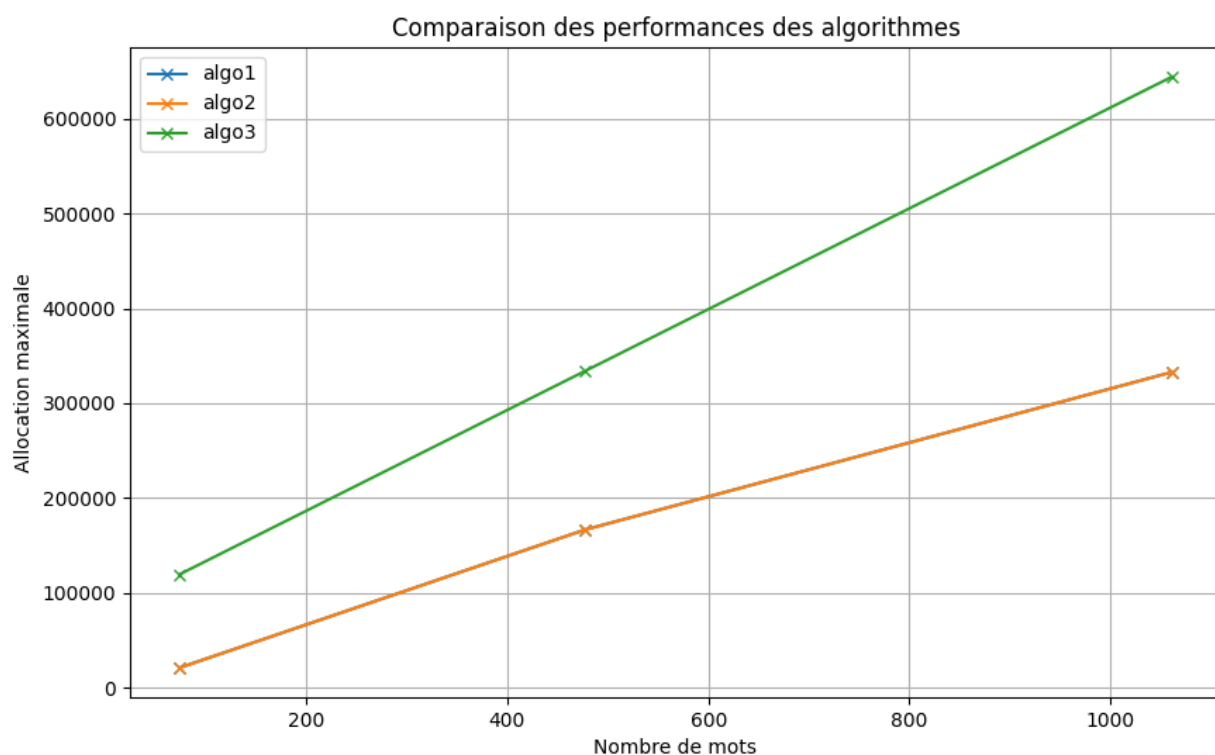
## ALGORITHME 3

Le troisième algorithme est le plus performant en termes de temps d'exécution.

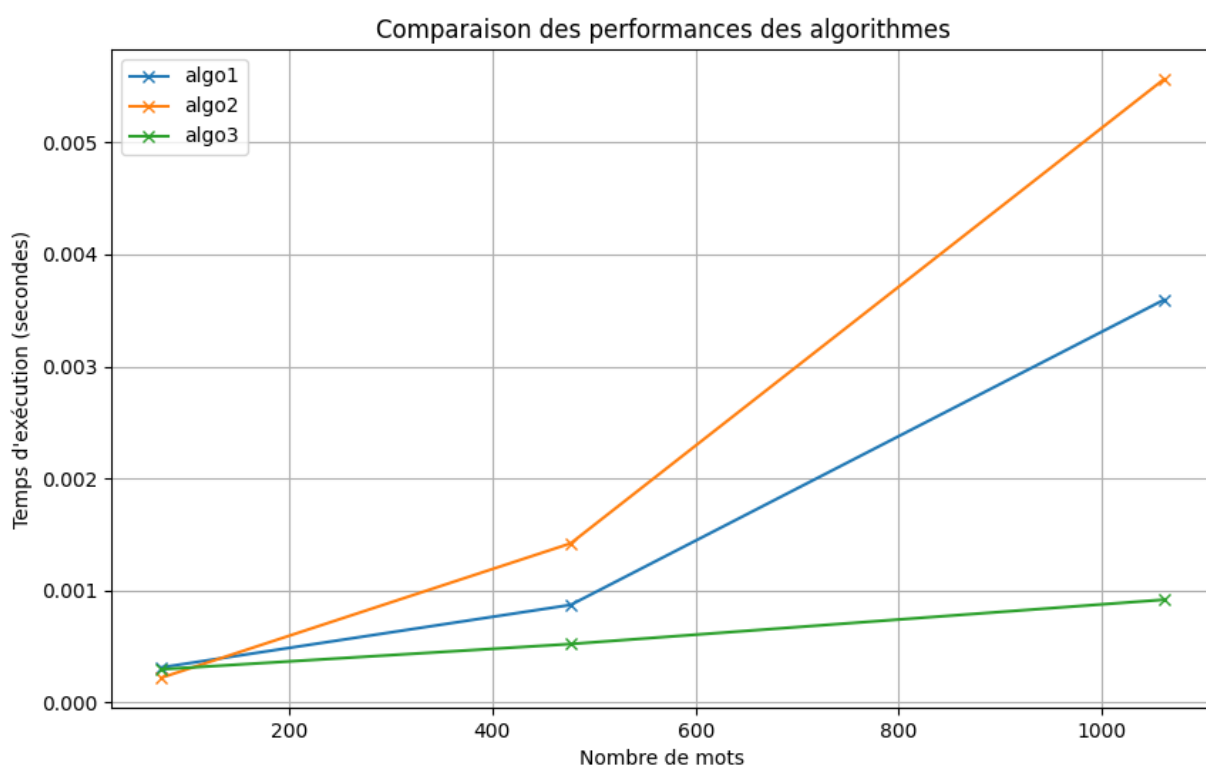
Grâce à l'utilisation d'une table de hachage, les opérations de recherche et d'insertion se font en temps moyen constant, ce qui donne une complexité globale en  $O(n)$ .

En contrepartie, cet algorithme consomme davantage de mémoire, en raison de l'utilisation de structures dynamiques supplémentaires pour gérer la table de hachage et les collisions.

Son avantage en temps devient particulièrement visible lorsque la taille des fichiers analysés augmente.



Sur ce graphique, on remarque que comme prévu, le nombre d'allocation entre l'Algo 1 et 2 sont similaire, d'où la superposition des deux courbes, et l'Algo 3 qui nécessite plus d'allocation.



Sur ce graphique, qui représente le temps d'exécution par rapport au nombre de mots, nous pouvons distinguer que l'Algo 3 est de loin le plus performant sur des nombres de mots plus importants.

```
algo=algo1 fichier=fichier1.txt nbmots=74 temps=0.000314 max_alloc=20800 cumul_alloc=23400 cumul_desalloc=2600
algo=algo1 fichier=fichier2.txt nbmots=477 temps=0.000872 max_alloc=166400 cumul_alloc=252200 cumul_desalloc=85800
algo=algo2 fichier=fichier1.txt nbmots=74 temps=0.000222 max_alloc=20800 cumul_alloc=23400 cumul_desalloc=2600
algo=algo2 fichier=fichier2.txt nbmots=477 temps=0.001421 max_alloc=166400 cumul_alloc=252200 cumul_desalloc=85800
algo=algo3 fichier=fichier1.txt nbmots=74 temps=0.000297 max_alloc=119424 cumul_alloc=122024 cumul_desalloc=2600
algo=algo3 fichier=fichier2.txt nbmots=477 temps=0.000523 max_alloc=333820 cumul_alloc=336420 cumul_desalloc=2600
algo=algo1 fichier=fichier3.txt nbmots=1061 temps=0.003594 max_alloc=332800 cumul_alloc=585000 cumul_desalloc=252200
algo=algo2 fichier=fichier3.txt nbmots=1061 temps=0.005562 max_alloc=332800 cumul_alloc=585000 cumul_desalloc=252200
algo=algo3 fichier=fichier3.txt nbmots=1061 temps=0.000918 max_alloc=644508 cumul_alloc=647108 cumul_desalloc=2600
```

Les résultats montrent que pour les petits fichiers, les trois algorithmes ont des temps d'exécution très proches. Lorsque la taille du fichier augmente, l'algorithme 1 devient plus lent, ce qui confirme sa complexité en  $O(n^2)$ . L'algorithme 2 offre de meilleures performances grâce à la recherche dichotomique. L'algorithme 3 est le plus rapide pour tous les fichiers, ce qui confirme l'efficacité de la table de hachage, au prix d'une consommation mémoire plus élevée.



# ORGANISATION

## **ALGORITHME 1**

Pour le premier algorithme, nous avons réfléchi ensemble à la solution. Nous l'avons ensuite développé chacun de notre côté, puis conservé une version finale commune.

## **ALGORITHME 2**

Le deuxième algorithme a été conçu et implémenté par Alexandre.

## **ALGORITHME 3**

Le deuxième algorithme a été conçu et implémenté par Adrien.



# CONTRIBUTION

Le projet a été réalisé en binôme avec une répartition équilibrée du travail. Les choix d'algorithmes ont été définis ensemble, puis chaque membre a développé un ou plusieurs algorithmes. La contribution est estimée à 50 % pour chaque membre. L'algorithme 1 est issu d'un travail commun, l'algorithme 2 a été conçu par Alexandre et l'algorithme 3 par Adrien.