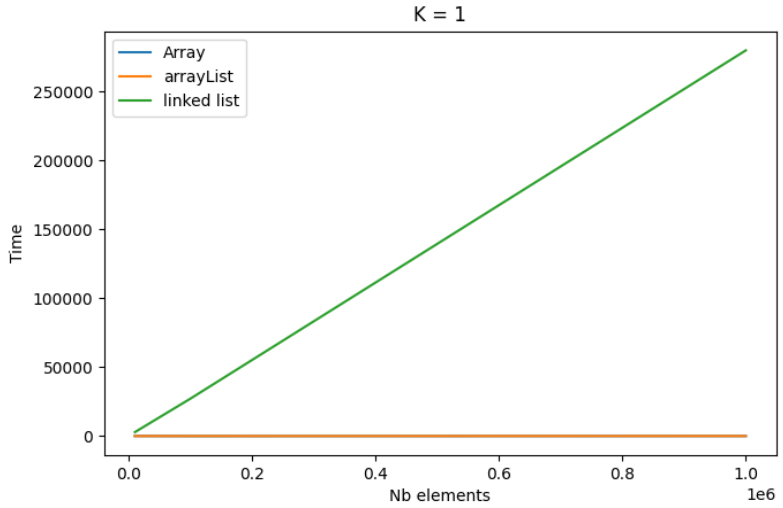
Rapport Bloom Filter

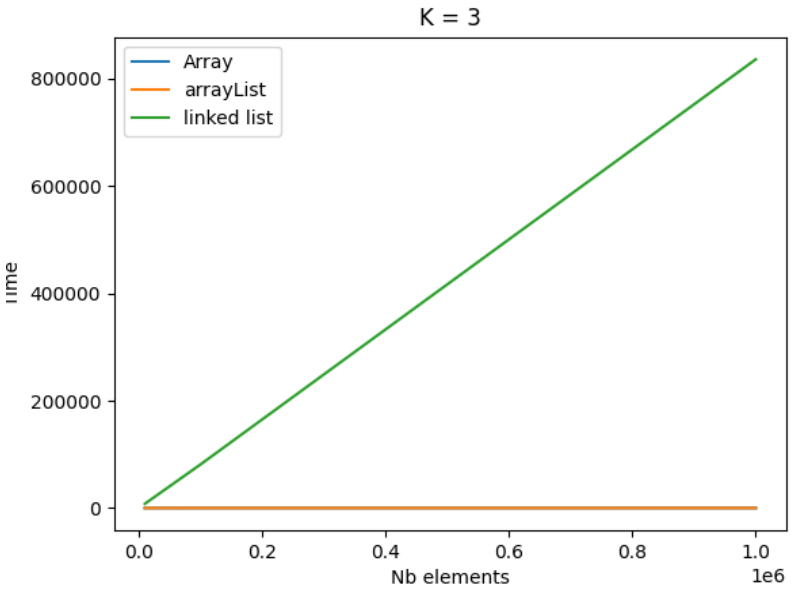
Tom Fourcaudot S3B’

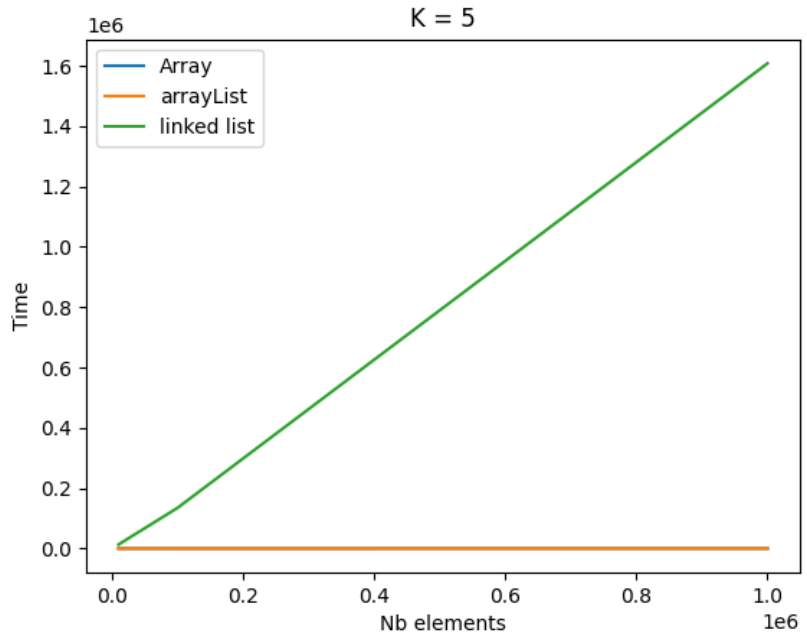
Temps d'exécution des différents filtres

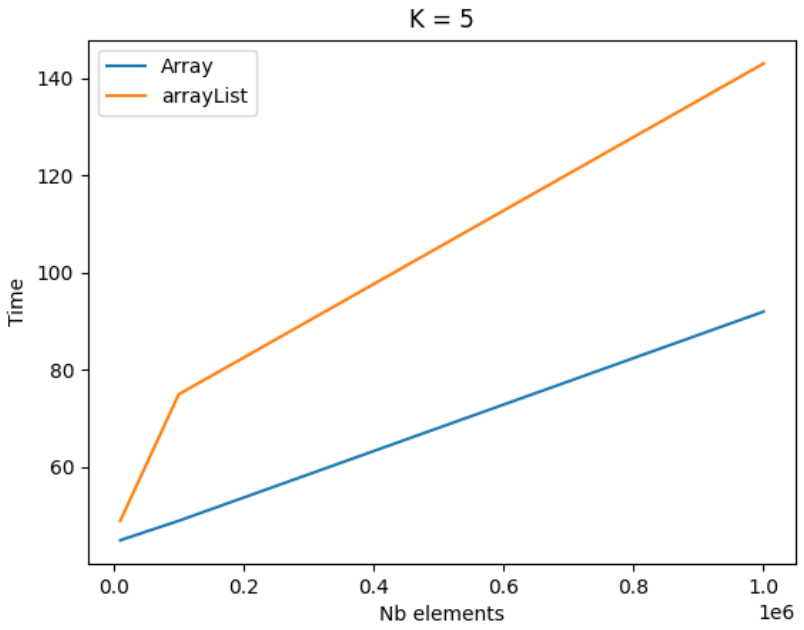
Dans cette section se trouveront des graphiques générés par les données de mon benchmark et python. Il représentent le temps d'exécution de la fonction contains, sur chacun des filtres, avec comme valeur fixe N (le nombre d'éléments ajoutés dans le filtre) = 10000.

Etudes du temps en fonction de la taille du filtre (M)









Zoom sur le graphique K=5

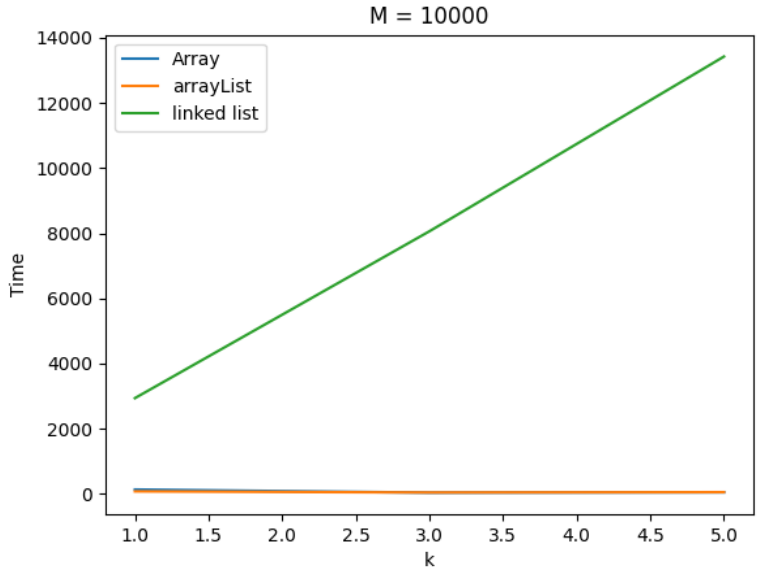
Les graphiques ci-dessus représentent le temps d'exécution de contains, avec un K (nb de fonctions de hachage) fixe. Nous ne faisons que varier la taille du filtre (10.000, 100.000 et 1.000.000).

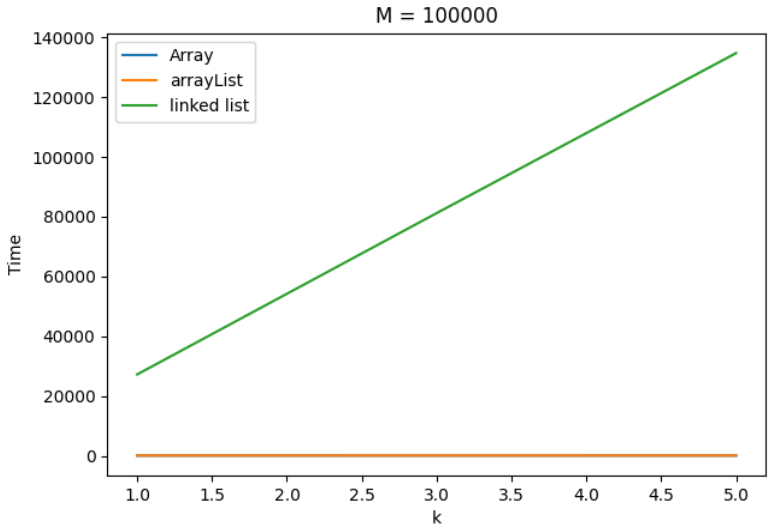
Nous pouvons déjà voir que plus le filtre est grand, plus le temps d'exécution est grand. Cela s'explique car le temps pour accéder à une case éloignée est grand. Dans les grands filtres, on peut supposer que nous devons accéder au cases éloignées un grand nombre de fois.

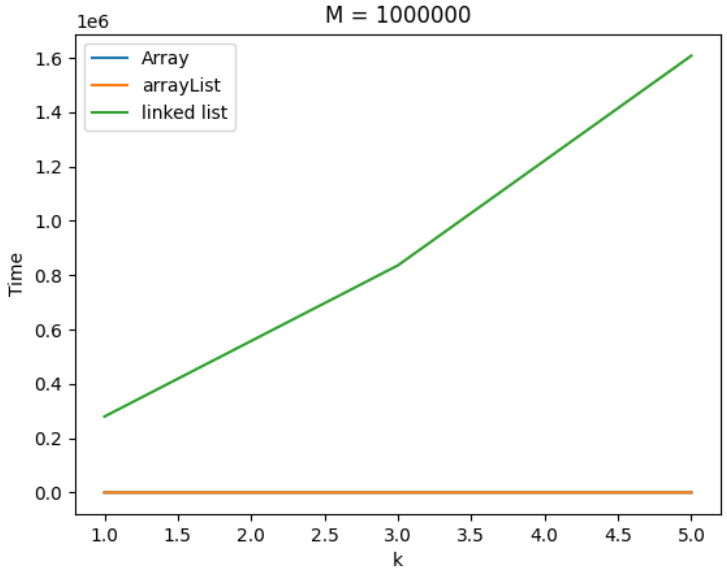
Ensuite, nous pouvons observer que le filtre de Bloom, implémenté avec une linkedlist, a un temps d'exécution beaucoup plus grand que la version Array et ArrayList. Cela s’explique par le fait que si nous voulons accéder à une case N, nous devons parcourir toutes les cases avant celle-ci. Dans le pire des cas, pour chacun des éléments à ajouter, nous devons accéder à M cases (M étant la taille du filtre). Plus le tableau sera grand, plus la version linked list prendra du temps.

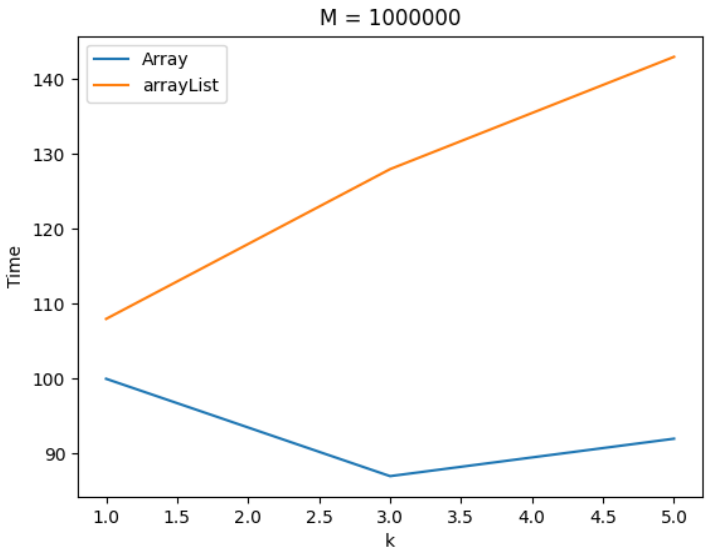
En revanche, la version ArrayList et Array ont un temps d'exécution très proche. Grâce au zoom du dernier graphique, nous pouvons voir que la version Array est un peu plus rapide que la version ArrayList, mais la différence reste légère. Cela est dû au fait qu’une ArrayList est une structure plus complexe qu’un simple tableau. Nous ne faisons que des actions simples sur toutes les structures de données, donc le tableau devient légèrement plus efficace que l’ArrayList.

Étude du temps d'exécution en fonction du nombre de hachage (K)









Zoom sur le graphique M = 1.000.000

Les graphiques ci-dessus représentent le temps d'exécution de contains, avec un M (nombre d'éléments ajouter et tester) fixe. Nous ne faisons que le nombre de fonctions de hachage (1, 3, 5).

De manière générale, nous pouvons voir que plus nous utilisons de fonctions de hachage, plus le temps d'exécution est long. Cela peut s'expliquer car si nous utilisons 1 fonction de hachage, un élément prendra 1 case du filtre. Il ne faudra accéder qu'une seule fois au filtre. En revanche, si nous utilisons k fonctions de hachage, il faudra accéder, dans le pire des cas, k fois au filtre, ce qui augmente le temps le temps d'exécution.

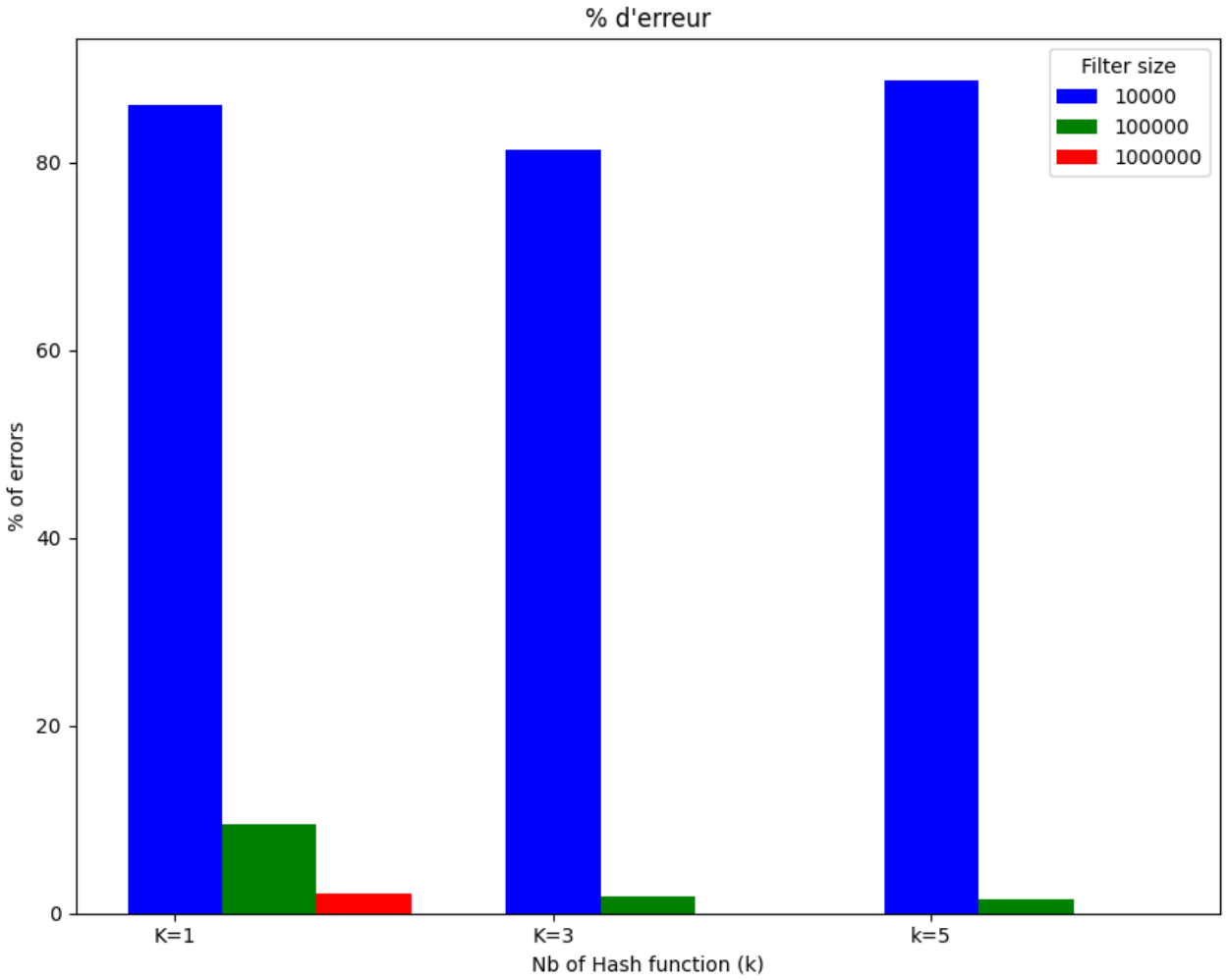
De nouveau, nous pouvons constater que la version linked list prend beaucoup plus de temps que les deux autres versions. En revanche, la version Array et ArrayList ont un temps d'exécution variant de quelques nanosecondes, mais la version Array reste plus rapide. Cela est dû à nouveau au fait que l'ArrayList est une structure plus complexe qu'un simple Array. L'Array devient à nouveau, légèrement plus efficace que l'ArrayList

Conclusion

Nous avons pu observer, de manière générale, que plus K ou M augmente, plus le temps d'exécution est long.
Nous avons aussi pu voir que la LinkedList n'est pas adaptée pour ce type d'algorithme. Elle devient significativement moins efficace si nous augmentons la taille du tableau ou le nombre de fonctions de hachages. L'Array reste meilleur que l'ArrayList, car sa structure est plus simple, et que les actions effectuées sont les plus basiques (accéder à une case ou écrire dans une case).

Taux d’erreur

Dans cette section, nous observerons le pourcentage d'erreurs, en fonction de K en de M. Le graphique a été généré grâce aux données de mon benchmark et python. Ici, j’utilise la version Array du filtre de Bloom,car elle est plus rapide. Le nombre d'éléments dans le filtre est fixé à 10.000. K prend les valeurs (1, 3, 5) et M (10.000, 100.000, 1.000.000). Pour ce test, nous ajoutons 10.000 élément aléatoire dans le filtre, puis nous appelons la fonction contained, sur tous les nombres non présents dans le filtre, de -1.000.000 à 1.000.000.



Au premier regard, nous pouvons voir que la taille du filtre influe grandement dans le taux d’erreur. Plus le tableau est grand, moins il y aura d'erreurs. Cela s’explique par le fait que plus le tableau est grand, plus nous réduisons le risque de collisions de la fonction de hachage. En effet, la fonction de hachage retourne un index entre 0 et M (taille du filtre). Plus M est grand, plus la fonction de hachage peut proposer des index différents, ce qui réduira les collisions, et donc les erreurs.

Nous pouvons aussi voir que le nombre de fonctions de hachage joue aussi un rôle dans le pourcentage d'erreurs. Plus nous utilisons de fonctions de hachage, plus le taux d’erreur est légèrement plus faible. Cela est dû au fait que si nous utilisons qu’une fonction de hachage, un élément sera rangé que dans une seule case du filtre. A la moindre collision de la

fonction de hachage, il y aura une erreur. En revanche, si nous utilisons K fonctions de hachage, il faudra K collisions pour produire une erreur.

Un cas intéressant se produit dans le graphique ci-dessus. Nous pouvons voir que le filtre avec le plus grand nombre de fonctions de hachage et la plus petite taille, a le plus grand pourcentage d'erreurs. Cela montre que si le filtre est trop petit, le nombre de fonctions de hachage peut avoir un effet négatif sur le taux d'erreur. Je suppose que dans un filtre de petite taille, si nous utilisons trop de fonctions de hachage, le filtre sera rempli trop rapidement, ne laissant que peux de case libres. Dans ce cas, le risque de collision est trop élevé.

Conclusion

Le filtre de bloom est un algorithme efficace pour stocker et tester la présence d'un élément. En revanche, les paramètres utilisés (K, M) doivent être choisis intelligemment. Si nous utilisons un N et un K trop grand, le taux d'erreur sera presque nul, mais l'algorithme prendra beaucoup de temps. En revanche, si nous utilisons des paramètres trop petits, l'algorithme sera très rapide, mais le taux d'erreur sera presque au maximum. Pour un filtre de Bloom efficace, le juste milieu entre rapidité et taux d'erreur.