# Rapport de projet d’Algorithmique

Nous soussignons BARBÉ Romain et RODRIGUEZ Maxime, déclarons sur l’honneur que ce projet est le résultat de notre travail personnel et que nous n'avons pas copié tout ou partie du code source d’autrui afin de le faire passer pour le nôtre.

## Introduction:

Notre projet se base sur l’étude d’une structure de données nommée le filtre de Bloom. Il s’agit d’une structure étant destinée à utiliser une faible consommation de mémoire au détriment d’une précision infaillible.

Au cours de ce rapport, nous allons vous expliciter nos travaux, nous chercherons à expérimenter les valeurs optimales à utiliser pour le filtre de bloom et, nous effectuerons des comparaison avec d'autres structures de données tel que les tables de hachage et les Arbres Binaires de Recherches.
Nos expériences vont être réalisées au travers de listes de mots de passe.

## Guide utilisateur :

Nous avons plusieurs fichiers au sein de notre projet. Il va se découper en deux parties :

- Une partie avec les listes de mots de passe dans le dossier passwords
- Une partie avec nos fichiers .c et .h

La seconde partie, qui contient nos fichiers main de tests, va se composer de plusieurs manières.
Comme nous avons un fichier de test pour chaque structure, il y a une base commune d’exécution des fichiers qui diffère selon les structures.

Voici la base commune (qui sera répétée pour chaque fichier):

```
./exe -fin <fichier d'entrée> -fout <fichier de sortie>
```

- fin: chemin vers le fichier d’entrée de mots de passe.
- fout: chemin vers le fichier de sortie (dans lequel seront écrits nos résultats).

Pour les fichiers de test “main_test_bloom_filter” et “test_false_positive”:

```
./exe -fin <fichier d'entrée> -fout <fichier de sortie> -m <Valeur
de m>
      -k <Valeur de k> -n <Valeur de n>
```

- -m : taille du bit array

- -k : nombre de fonction de hachage
- -n : nombre de clés (uniquement pour les statistiques)

Pour le fichier “main_test_bst “

```
./exe -fin <fichier d'entrée> -fout <fichier de sortie>
```

Pour le fichier “main_test_hash”

```
./exe -fin <fichier d'entrée> -fout <fichier de sortie> -m <Valeur de m>
```

- -m : taille de la table de hachage

Tests fournis par M. Thapper:

```
./test
```

Vous pouvez faire un make de chaque test soit:

- `main_test_bloom_filter`
- `main_test_hash`
- `main_test_bst`
- `main_false_positive`
- `test`

# Répartition des tâches :

Maxime :

- bitarray.c
- hashtable.c
- tree_word.c
- étude des faux positifs

Romain :

- makefile
- filter.c
- comparaisons en temps et en espace

Les 2 membres du binôme :

- Différents fichiers de test (sauf “test.c” fournit par M.Thapper)
- Ajustements après la soutenance
- Rédaction du rapport

# Le projet et les résultats obtenus :

## Ce qui fonctionne:

En terme d'implémentation du projet, tous nos fichiers fonctionnent sans warning avec nos flags "-Wall -ansi -g" et sans fuite mémoire sous valgrind.
Nous avons réalisé l'ensemble des questions de base de l'énoncé.

## Ce qui ne fonctionne pas:

Pas de bug connu.

## Étude des faux positifs :

Pour notre étude des faux positifs, l'objectif est de tester différentes valeurs moyennes pour k et m. L'objectif est de tester différentes valeurs afin d'observer les faits émis dans l'énoncé.
Il nous est spécifié une formule "(1 − e^(-kn/m))^k" que nous allons tenter d'approcher.

Pour cela, nous allons effectuer un test au sein du fichier test_false_positive.c.

L'idée est, depuis un fichier de mots de passe, prendre les mots de longueur 3 ou 4 que l'on stocke dans notre filtre de bloom ainsi que dans une structure secondaire. Nous avons choisi une table de hachage pour la structure secondaire.

Une fois les données stockées dans nos structures, nous relisons le même fichier pour déterminer le taux de faux positifs.

Pour nos tests, nous avons, pour chaque données, fait une moyenne sur 1000 essais avec les mêmes caractéristiques (nous aimerions le faire avec plus, mais nous n'avons pas de super-ordinateur, et nous avons besoin de les utiliser mine de rien).

Nous prenons d'abord une valeur m entre 5 à 10 fois supérieure à notre nombre de clés pour obtenir le k optimal.
Ensuite, nous testons si les moyennes obtenues sont cohérentes :

| k/m/n // tests | Valeur | Moyenne de |
|---|---|---|

| | nominale | test (10 000 échantillons) |
|---|---|---|
| k=6<br>m=8000<br>n=1000 | 0.021% | 0.018% |

Avec notre implémentation, on peut admettre suite à ce test que la moyenne de faux positifs obtenue se rapproche du taux de faux positifs donné par la formule précédente.
Encore une fois, il faudrait des tests bien plus conséquents pour être sûr à 100%.

Ensuite nous effectuons quelques tests pour vérifier l'énoncé qui indique le changement du taux de faux positif en fonction de m pour n = 1000.

Pour chaque résultat, j'affiche une moyenne obtenue avec notre programme puis environ le résultat que nous sommes censés avoir selon la formule.

| k // m -- ><br>\|<br>v | 8000 | 4000 | 2000 |
|---|---|---|---|
| 6 | 0.018%<br>0.021% | 0.28%<br>0.022% | 1,2%<br>0.73% |
| 3 | 0.31%<br>0.031% | 2.12%<br>0.15% | 5.75%<br>0.46 % |

La conclusion que nous faisons par rapport à ces données, est bien sûr limitée dû à nos faibles tests. Mais nous pouvons probablement supposer que plus nous baissons "m", plus la chance que le nombre de faux positifs augmente sensiblement comparé aux données de base. On peut observer que plus nous allons vers un m faible et plus la différence entre les données théoriques et pratiques augmente.

## **Comparaisons en temps et en espace :**

Pour comparer en temps, nous avons utilisé la méthode time sur nos programmes de test, nous avons donc un temps pour chaque test, ce qui nous permet d'avoir un temps d'exécution qui équivaut au temps passé à mettre en place les données dans la structure ainsi que du temps passé à récupérer ces données.

Pour comparer en espace, nous avons utilisé l'outil de valgrind "massif" qui nous permet d'avoir affiché après l'exécution du programme la taille de la mémoire de la pile qui a été utilisée.

Comme nous avons utilisé un tableau de char au lieu d'un véritable bit array, il nous semble logique de diviser la mémoire obtenue via le test du filtre de bloom par 8.

Pour la comparaison en temps, nous avons le tableau suivant pour les valeurs m = 8 * n et k = (m/n)*ln(2) :

| Nombre de mots testés (n) | Bloom Filter | HashTable | ABR / BST |
|---|---|---|---|
| 1 000 | 0,007 secondes | 0,004 secondes | 0,020 secondes |
| 10 000 | 0,030 secondes | 0,015 secondes | 0,210 secondes |

Pour la comparaison en espace, nous avons le tableau suivant pour les valeurs m = 8 * n et k = (m/n)*ln(2) :

| Nombre de mots testés (n) | Bloom Filter | HashTable | ABR / BST |
|---|---|---|---|
| 1 000 | 16,53 kB | 92,35 kB | 35,26 kB |
| 10 000 | 86,84 kB | 746,4 kB | 165,2 kB |

On peut donc voir ici avec ces comparaisons en temps et en espace que le filtre de bloom est une structure très efficace, en effet, cette structure est une bonne alternative si on veut avoir une structure rapide comme une table de hachage ou si on veut avoir une structure moins gourmande en terme de mémoire comme l'ABR.

# Conclusion :

En conclusion de ce TP, nous pouvons dire que nous avons beaucoup appris sur le filtre de Bloom, que ce soit, en faisant des recherches sur Internet, avec l'énoncé du projet ou en l'implémentant et en l'améliorant.

Son implémentation est intéressante car, grâce aux résultats que nous avons obtenus durant les différents tests, nous pouvons juger nous-même directement de l'efficacité en temps et en mémoire de la structure.

En plus de l'apercevoir grâce à la rapidité d'exécution, c'est avec une comparaison des autres structures que nous avons effectivement l'efficacité du filtre de bloom.

Une chose nous frappe très rapidement lors de nos tests sur les comparaisons entre les structures. Nous avions déjà fait des tests sur le temps que prennent les structures lors de différentes opérations. Les complexités de la recherche, insertion, suppression etc...
Mais jamais nous n'avions comparé l'espace occupé par ses structures, et c'est une chose à laquelle nous ne pensions plus énormément car les machines d'aujourd'hui nous permettent de contenir des données immenses. Cependant, ce fut un vrai choc de voir à quel point la table de hachage est gourmande en terme de stockage et l'avantage du filtre de bloom en comparaison.

En somme, ce fut un projet très intéressant dans son contenu car nous avons finalement appris beaucoup. Nous avons été curieux d'aller observer quelles sont les caractéristiques précises des structures que nous connaissons déjà, mais surtout d'apprendre l'existence de celles que l'on ne voit pas forcément en cours.