

# Algorithmique et programmation 1AA

## Projet période 1 (2025)

Ce projet est à réaliser par équipes de 2 à 5, et à rendre sur teide pour le **08 octobre à 23h59**. Chaque membre de l'équipe doit à la fin du projet être en mesure de comprendre l'ensemble du travail. Le fait de travailler en équipe ne doit pas conduire à un partage des tâches qui permet de ne pas se confronter à ses points faibles, et doit au contraire permettre de l'entraide pour que chacun puisse progresser sur la partie la plus difficile pour lui.

Vous rendrez une archive .tar.gz contenant votre compte-rendu (réponse aux questions) au format pdf (il n'est pas nécessaire d'être trop verbeux ni de passer trop de temps à la mise en forme), des fichiers textes contenant les mots de passe trouvés, et vos programmes en langage C accompagnés d'un Makefile, qui devront pouvoir être compilés (sans erreurs ni warnings) et exécutés sur les ordinateurs de l'ensimag. Veillez à respecter strictement les spécifications imposées.

La [charte contre la fraude](#) proposée par l'Ensimag s'applique. Notamment tout usage (même partiel) de matériel (code, réponses ou autres) que vous n'avez pas produit vous-même est strictement interdit, ainsi que plus généralement toute aide extérieure.

Ce sujet est disponible sur [http://bmasse.gitlab.io/ensimag\\_ap1aa/projet\\_periode1.tar.gz](http://bmasse.gitlab.io/ensimag_ap1aa/projet_periode1.tar.gz), et sera si nécessaire mis à jour pour répondre à d'éventuels signalements d'erreurs ou demandes de clarification. De plus, un document sur les tables de hachage est disponible sur [http://bmasse.gitlab.io/ensimag\\_ap1aa/hachage.pdf](http://bmasse.gitlab.io/ensimag_ap1aa/hachage.pdf).

## 1 Introduction

Dans un système informatique moderne, les mots de passe ne sont pas stockés en clair, mais sous forme hachée. Lorsque l'utilisateur fait une tentative de connexion, le mot de passe entré est haché et comparé au hash stocké. Ainsi, une personne ayant accès en lecture à la base de données de mots de passe ne peut pas l'utiliser pour se connecter frauduleusement, à moins d'inverser la fonction de hachage (c'est à dire de trouver un antécédent du hash).

Si cette fonction de hachage est bien choisie, cette recherche d'antécédent se fait par force brute (essai successif de chacun des mots de passe possibles) et requiert donc une puissance de calcul considérable (pour des mots de passe de complexité suffisante).

Si la majorité des systèmes utilisent une même fonction de hachage standard, on peut imaginer calculer une seule fois et stocker les hashes associés à tous les mots de passe possibles, et utiliser par la suite cette base de données pré-calculée pour attaquer n'importe quel hash sur n'importe quel système. Mais le stockage d'une telle base de données demandera un espace mémoire conséquent (pour des mots de passe suffisamment long et complexes).

Pour la suite de l'énoncé, on suppose que les mots de passe sont composés de  $M$  lettres minuscules exactement. Sauf précision contraire, les résultats doivent être donnés en fonction de  $M$ .

**Question 1** — *Donnez les complexités spatiales et temporelles de l'attaque d'un hash (et de l'étape de pré-calcul dans le second cas) de chacune de ces techniques (force brute et base de données pré-calculée). Jusqu'à quelles valeurs de  $M$  l'utilisation de ces techniques vous paraît-elle réaliste ?*

On s'intéresse dans la suite de l'énoncé à la compréhension et l'implémentation de techniques probabilistes permettant de trouver rapidement des antécédents de hashes en utilisant des structures

de données pré-calculées tout en limitant l'espace mémoire nécessaire.

La fonction de hachage attaquée, qu'on notera  $h$  dans la suite de l'énoncé, est donnée dans `hash.h`, et on fournit comme expliqué plus bas quelques hashes qu'on aimerait attaquer.

## 2 Méthode des hashes chaînés : compromis temps - mémoire

Cette technique est un compromis entre le calcul par force brute (pas de pré-calcul et stockage mais la recherche d'un mot de passe est longue) et la création d'une base de données exhaustive (recherche d'un mot de passe rapide une fois la base calculée mais espace nécessaire considérable).

On utilise une fonction de réduction  $r$  qui transforme un hash en un nouveau mot de passe. Les propriétés attendues de cette fonction sont les mêmes que pour une fonction de hachage : déterminisme (c'est une fonction au sens mathématique), sorties distribuées uniformément dans l'espace d'arrivée, et calcul efficace.

On peut ainsi composer une chaîne de mots de passe et de hashes :

$$pass_0 \xrightarrow{h} hash_0 \xrightarrow{r} pass_1 \xrightarrow{h} hash_1 \xrightarrow{r} \dots \xrightarrow{r} pass_{L-1} \xrightarrow{h} hash_{L-1} \xrightarrow{r} pass_L$$

En stockant seulement le premier et le dernier mot de passe de la chaîne ( $pass_0$  et  $pass_L$ ), on peut facilement tenter de tester si un hash donné correspond à un mot de passe dans cette chaîne : si en appliquant la fonction de réduction une fois au hash donné on obtient  $pass_L$ , alors le mot de passe candidat est  $pass_{L-1}$ . Si en appliquant la fonction de réduction puis la fonction de hash puis la fonction de réduction on obtient  $pass_L$ , alors le mot de passe candidat est  $pass_{L-2}$ , etc.

Plus généralement et plus formellement, on applique les  $(r \circ h)^i \circ r$  avec toutes les valeurs de  $i$  de 0 à  $L - 1$  au hash attaqué. Si on obtient  $pass_L$ , alors le mot de passe candidat est  $pass_{L-i-1}$ .

Supposons qu'on calcule et stocke au total  $N$  chaînes similaires à partir de mots de passes choisis aléatoirement :

$$\begin{aligned} & pass_{1,0} \xrightarrow{h} hash_{1,0} \xrightarrow{r} pass_{1,1} \xrightarrow{h} hash_{1,1} \xrightarrow{r} \dots \xrightarrow{r} pass_{1,L-1} \xrightarrow{h} hash_{1,L-1} \xrightarrow{r} pass_{1,L} \\ & pass_{2,0} \xrightarrow{h} hash_{2,0} \xrightarrow{r} pass_{2,1} \xrightarrow{h} hash_{2,1} \xrightarrow{r} \dots \xrightarrow{r} pass_{2,L-1} \xrightarrow{h} hash_{2,L-1} \xrightarrow{r} pass_{2,L} \\ & \dots \\ & pass_{N,0} \xrightarrow{h} hash_{N,0} \xrightarrow{r} pass_{N,1} \xrightarrow{h} hash_{N,1} \xrightarrow{r} \dots \xrightarrow{r} pass_{N,L-1} \xrightarrow{h} hash_{N,L-1} \xrightarrow{r} pass_{N,L} \end{aligned}$$

Pour trouver l'antécédent d'un hash à attaquer  $inputhash$ , on généralise le raisonnement précédent : on cherche si un des  $(r \circ h)^i \circ r(inputhash) \forall i \in [0, L - 1]$  est trouvé parmi les  $pass_{x,L} \forall x \in [1, N]$ . Si on trouve un match,  $(r \circ h)^{L-i-1}(pass_{x,0})$  est le mot de passe candidat.

**Question 2** — *Quelle structure de données peut-on utiliser pour stocker les couples  $(pass_{x,0}, pass_{x,L})$  de façon à rendre la recherche précédente relativement efficace ?*

**Question 3** — *Le candidat est-il nécessairement un antécédent du hash attaqué ? Expliquez précisément pourquoi, en vous appuyant sur un schéma si nécessaire.*

**Question 4** — *Le fait de ne pas trouver de candidat avec cette méthode implique-t-il nécessairement que le mot de passe cherché n'est pas présent dans la table ?*

**Question 5** — *Si on a  $pass_{x,i} = pass_{y,j}$  avec  $x \neq y$  (on appelle cette situation une collision), quelles sont les conséquences pour cette méthode ?*

### 3 Méthode des rainbow tables

Cette technique améliore la technique précédente en faisant varier la fonction de réduction utilisée à chaque étape. Chaque chaîne est ainsi de la forme

$$pass_{x,0} \xrightarrow{h} hash_{x,0} \xrightarrow{r_0} pass_{x,1} \xrightarrow{h} hash_{x,1} \xrightarrow{r_1} \dots \xrightarrow{r_{L-2}} pass_{x,L-1} \xrightarrow{h} hash_{x,L-1} \xrightarrow{r_{L-1}} pass_{x,L}$$

L'ensemble des N chaînes (toujours représentées par leurs premiers et derniers éléments) est appelé "rainbow table".

L'attaque d'un hash se fait de façon similaire<sup>1</sup> : on calcule l'image de ce hash par les différentes fonctions  $\left[ \prod_{j=1}^{j=i} (r_{L-j} \circ h) \right] \circ r_{L-i-1} \forall i \in [0, L-1]$ , et cherche les résultats obtenus parmi les  $pass_{x,L} \forall x \in [1, N]$ . Si on trouve un match,  $\left[ \prod_{j=i+2}^{j=L} (r_{L-j} \circ h) \right] (pass_{x,0})$  est le mot de passe candidat.

**Question 6** — *Est-ce alors toujours aussi problématique d'avoir  $pass_{x,i} = pass_{y,j}$  avec  $x \neq y$  et  $i \neq j$  ? Pourquoi ? Et si on a  $pass_{x,i} = pass_{y,i}$  avec  $x \neq y$ , que se passe-t-il au niveau des  $pass_{x,L}$  et  $pass_{y,L}$  ? Est-ce problématique ?*

Pour éviter cette situation, on veillera lors de la création de la table à rejeter toute chaîne terminant par un  $pass_{x,L}$  déjà présent et à recommencer avec un  $pass_{x,0}$  différent.

**Question 7** — *Si on choisit deux mots de passe initiaux aléatoires différents, obtenant les chaînes  $(pass_{x,0}, pass_{x,L})$  et  $(pass_{y,0}, pass_{y,L})$ , quelle est la probabilité d'obtenir une collision entre les mots de passe finaux (c'est à dire que  $pass_{x,L} = pass_{y,L}$ ) ? Quel est alors l'ordre de grandeur du nombre de chaînes qu'on peut facilement insérer dans une rainbow table en conservant l'unicité des mots de passe finaux ?<sup>2</sup>*

On comprend (et observe par l'expérience) qu'au delà de cet ordre de grandeur, le programme de création de la rainbow table aura de plus en plus de difficultés à trouver des valeurs de  $pass_{x,0}$  qui ne créent pas de collision.

Pour pallier ce problème, on va créer plusieurs rainbow tables (notons R le nombre de tables), chacune contenant N chaînes, toujours de longueur L. On continue d'imposer l'absence de collision (c'est à dire l'unicité des  $pass_{x,L}$ ) au sein de chaque table, mais pas entre les différentes tables. Il paraît cependant judicieux de ne pas autoriser l'égalité de deux  $pass_{x,0}$  même pour des tables différentes.

**Implémentez tout ça**, pour l'instant avec M=6, R=10, N=100000, L=1000. Ces valeurs sont définies comme constantes pré-processeurs dans le fichier `hash.h` fourni et pourront ainsi être modifiées ultérieurement.

Plus précisément, on veut écrire :

- un programme `rainbow_create` qui écrit les R tables dans des fichiers dont les noms sont donnés en arguments 1 à R (chaque table contient N lignes, chaque ligne contient un couple  $pass_{x,0} pass_{x,L}$  où les deux mots de passe sont séparés par un espace). Si un R+1ème nom de fichier est donné en argument, les  $pass_{x,0}$  seront lus depuis ce fichier (un par ligne), sinon (seulement R arguments donnés) ils seront générés aléatoirement.
- un programme `rainbow_attack` qui prend en arguments (dans cet ordre) les noms des fichiers contenant les R tables, le nom d'un fichier contenant une liste de hashes à attaquer (un par ligne, comme dans `crackme2025_6.txt`), et le nom du fichier où écrire les mots de

1. Ne laissez pas les indices compliqués vous troubler : essayez sur des exemples simples

2. À vous de décider ce qu'on entend par "facilement".

passé trouvés (un par ligne, ligne vide si pas d'antécédent trouvé, de sorte que ce fichier ait le même nombre de lignes que celui contenant les hashes à attaquer).

- un programme `hash_many` qui prend en premier argument le nom d'un fichier contenant une liste de mots de passe, et écrit leurs hashes (un par ligne) dans un second fichier dont le nom est donné en second argument. Cette fonction peut servir à tester les autres.

L'ensemble des exécutables demandés doit pouvoir être compilé par un simple appel à la commande `make` sans arguments sur les ordinateurs de l'école.

Quand tout fonctionne, attaquez la liste de hashes donnée dans le fichier `crackme2025_6.txt`. Rendez avec vos sources et vos réponses un fichier `found_6.txt` contenant les mots de passe trouvés. Ce fichier devra avoir le même nombre de lignes que `crackme2025_6.txt` : la ligne  $n$  contiendra un mot de passe (si vous avez trouvé un antécédent du hash présent à la ligne  $n$  de `crackme2025_6.txt`) ou sera dans le cas contraire (si vous n'avez pas trouvé d'antécédent) laissée vide pour garder la correspondance entre les deux fichiers.

**Question 8** — *Avec ces valeurs, pour combien de hashes trouvez-vous un antécédent ? Cela correspond-il à vos attentes ? Combien de temps vous a pris la création de la base de données, et combien de temps vous prend ensuite l'attaque de chaque hash sur un ordinateur de l'ensimag ? Commentez librement vos résultats.*

**Question 9** — (facultative) *Vous pouvez (si vous en avez le temps et l'envie) essayer d'améliorer vos résultats en changeant les valeurs de  $R$ ,  $N$  et  $L$ , dans les limites du raisonnable en terme de temps de calcul<sup>3</sup>.*

Si vous y parvenez (et là aussi si vous en avez le temps et l'envie), attaquez finalement la liste de hashes donnée dans le fichier `crackme2025_7.txt` qui correspondent maintenant à des mots de passe de  $M=7$  lettres, et rendez alors un fichier `found_7.txt` au même format que précédemment.

## Recherche documentaire

**Question 10** — *Quelles techniques couramment implémentées de nos jours permettent de se prémunir contre de ce genre d'attaques ? On parle ici de techniques qui permettraient de se prémunir spécifiquement contre une recherche d'antécédents de hashes par le biais d'une base de données pré-calculée.*

---

3. Vous pouvez utiliser les ordinateurs de l'ensimag, mais restez raisonnable, pensez à la planète...