# Algorithmique et programmation en 1AA
# Projet période 1 (2021)

Ce projet est à réaliser par équipes de 2 ou 3 étudiants, et à rendre (sur teide) pour le 21 octobre à 8h00. Vous rendrez une archive .tar.gz contenant votre compte-rendu (réponse aux questions) au format pdf, un fichier texte contenant la liste des mots de passe trouvés et vos programmes en langage C (accompagnés d'un Makefile). Veillez à respecter strictemment les spécifications imposées.

La charte contre la fraude proposée par l'Ensimag s'applique. Notamment tout partage ou copie (même partielle) de programmes ou de réponses avec une autre équipe est strictement interdit. Il vous est en conséquence interdit de travailler dans un répertoire ou un dépôt git accessible à tous en lecture.

Ce sujet est disponible sur `http://perso.crans.org/frenoy/projet_periode1.tar.gz`, et sera si nécessaire mis à jour pour répondre à d'éventuels signalements d'erreurs ou demandes de clarification.

## 1 Introduction

Dans un système informatique, 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[1] 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 est bien choisie, cette inversion se fait par force brute et requiert donc une puissance de calcul considérable (pour des mots de passe de complexité suffisante).

On s'intéresse ici à des techniques probabilistes permettant de trouver rapidement des antécédents de hashs en utilisant des structures de données pré-calculées.

La fonction de hachage attaquée, qu'on notera $h$ dans la suite de l'énoncé, est fournie dans `hash.h` : vous utiliserez impérativement cette implémentation pour l'écriture de vos programmes. Cette implémentation s'appuie sur la bibliothèque openssl, l'argument `-lcrypto` doit donc être donnée au compilateur.

**Question 1** — *Quel est le nombre maximal théorique de hashs différents que cette fonction de hachage peut produire ?*

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 2** — *Combien d'essais doit-on faire par force brute (c'est à dire en essayant tous les*

1. En bon français, on devrait écrire *valeur de hachage*

*mots de passe séquentiellement) pour trouver l'antécédent d'un hash donné, dans le pire des cas et en moyenne ?*

Si une grande partie des systèmes utilisent une même fonction de hachage standard, on peut imaginer calculer une seule fois et stocker les hashs 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.

**Question 3** — *Quelle structure de données vous semble adaptée pour stocker ces informations pré-calculées (hashs associés à tous les mots de passe possibles) ? Quelle sera la taille de cette structure de données ? Quelle sera la complexité temporelle de la recherche d'un mot de passe à partir de son hash ?*

**Question 4** — *Donnez ces valeurs (complexité de l'attaque d'un hash par force brute, taille de la base de données exhaustive, et complexité de la recherche dans cette base de données du mot de passe associé à un hash) pour M=6 et M=8. L'utilisation de ces techniques (force brute et pré-calcul) vous parait-elle alors réaliste ?*

## 2 Méthode des hashs chainés : 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 [2]. On peut ainsi composer une chaine de mots de passe et de hashs :

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

En stockant seulement le premier et le dernier mot de passe de la chaine ($pass_0$ et $pass_L$), on peut facilement tester si un hash donné correspond à un mot de passe dans cette chaine : si en appliquant la fonction de réduction une fois au hash donné on obtient $pass_L$, alors le mot de passe cherché est probablement [3] $pass_{L-1}$. 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 cherché est probablement $pass_{L-2}$. 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 cherché est probablement $pass_{L-i-1}$.

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

$$pass_{1,0} \xrightarrow{h} hash_{1,0} \xrightarrow{r} pass_{1,1} \xrightarrow{h} hash_{1,1} \xrightarrow{r} ... \xrightarrow{r} pass_{1,L-1} \xrightarrow{h} hash_{1,L-1} \xrightarrow{r} pass_{1,L}$$

$$pass_{2,0} \xrightarrow{h} hash_{2,0} \xrightarrow{r} pass_{2,1} \xrightarrow{h} hash_{2,1} \xrightarrow{r} ... \xrightarrow{r} pass_{2,L-1} \xrightarrow{h} hash_{2,L-1} \xrightarrow{r} pass_{2,L}$$

$$...$$

$$pass_{N,0} \xrightarrow{h} hash_{N,0} \xrightarrow{r} pass_{N,1} \xrightarrow{h} hash_{N,1} \xrightarrow{r} ... \xrightarrow{r} pass_{N,L-1} \xrightarrow{h} hash_{N,L-1} \xrightarrow{r} pass_{N,L}$$

2. La seule propriété attendue de cette fonction et qu'elle distribue ses sorties uniformément dans l'espace d'arrivée

3. On explicitera ce *probablement* dans les questions suivantes

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 5** — *Le candidat est-il nécessairement un antécédent du hash attaqué ? Pourquoi ?*

**Question 6** — *Dans le cas très favorable où il n'y a pas de redondance (les $pass_{x,i}$ sont uniques), combien de candidats (mots de passe potentiels) différents sont testés avec cette méthode ?*

**Question 7** — *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 ? Quelle sera alors la complexité temporelle de l'attaque d'un hash (choisissez et précisez vos hypothèses, et si vous traitez la complexité moyenne ou dans le pire des cas) ?*

On a pour l'instant supposé que les $pass_{x,i}$ sont uniques, mais ce ne sera probablement pas toujours le cas selon les valeurs de L et N choisies.

**Question 8** — *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 ?*

**Question 9** — *Combien il y a-t-il théoriquement de hashs différents (en utilisant notre fonction de hachage sur toutes chaines de caractères possibles de toute longueur, et pas seulement sur des mots de passe valides) ?*

**Question 10** — *Combien il y a-t-il de mots de passe possibles avec $M = 6$ et avec $M = 8$ ? Quelle est donc l'étape qui cause la majorité des collisions (réduction ou hachage) ?*

## 3 Méthode des rainbow tables

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

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

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

L'attaque d'un hash se fait de façon similaire[4] : on calcule l'image de ce hash par les différentes fonctions $\prod_{j=L-i}^{j=L-1}(r_j \circ h) \circ r_{L-i-1}$ $(\forall i \in [0, L-1])$, et cherche les résultats obtenus parmi les $pass_{x,L}$ $\forall x \in [1, N]$. Si on trouve un match, $\prod_{j=0}^{j=L-i-1} r_j \circ h(pass_{x,0})$ est le mot de passe candidat.

**Question 11** — *Est-ce alors toujours aussi problématique d'avoir $pass_{x,i} = pass_{y,j}$ avec $x \neq y$ ? Pourquoi ?*

**Question 12** — *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 $pass_{y,L}$ ?*

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

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

**Question 13** — *Si on choisit deux mots de passes initiaux aléatoires, obtenant les chaines* $(pass_{x,0}, pass_{x,L})$ *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* $pass_{x,L} = pass_{y,L}$*) ?*

**Question 14** — *Quel est alors l'ordre de grandeur du nombre de chaines qu'on peut facilement insérer dans une rainbow table sans créer de collisions entre les mots de passe finaux ?*

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.

On va donc créer plusieurs rainbow tables (notons R le nombre de tables), chacune contenant N chaines, 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 parait cependant judicieux de ne pas autoriser l'égalité de deux $pass_{x,0}$ même pour des tables différentes.

**Question 15** — ***Implémentez tout ça****, avec M=6, R=10, N=100000, L=1000.*

Plus précisément, on veut écrire :

- un programme `rainbow_create` qui écrit les R tables dans des fichiers dont les noms sont donnés en arguments (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 (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 hashs à attaquer (un par ligne, comme dans `crackme2021.txt`), et le nom du fichier où écrire les mots de passe 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 hashs à attaquer).
- un programme `hash_many` qui prend en argument le nom d'un fichier contenant une liste de mots de passe, et écrit leurs hashs dans un second fichier dont le nom est aussi donné en 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 d'école.

Quand tout fonctionne, attaquez la liste de hashs donnée dans le fichier `crackme2021.txt`.

Rendez avec vos sources et vos réponses un fichier `found.txt` contenant les mots de passe trouvés : chaque ligne de ce fichier contiendra un mot de passe suivi d'un espace suivi du hash correspondant.

**Question 16** — *Pour combien de hashs trouvez-vous un antécédent ? Cela correspond-t-il à vos attentes ? Commentez librement vos résultats. Vous pouvez essayer de les améliorer en changeant les valeurs de R, N et L dans les limites du raisonnable, en terme de temps de calcul.*

## Recherche documentaire

**Question 17** — *Quelles techniques (couramment implémentées de nos jours) permettent de se prémunir simplement de ce genre d'attaques ?*

**Question 18** — *Quels sont les principaux usages des fonctions de hachage ? Quel est le consensus sur le niveau de sûreté de* `md5` *pour chacun de ces usages ?*