README TP5-6

Auteurs : Julien Da Costa, Gabin Chognot

Table des matières

Brute force

On a 3 hash au format suivant :

Login	Hash
Alice	8FC92036B2963C604DC38B2DDB305148
Bob	367F3AC1129CC92DCBB8C4B9EA4EE55A
Clara	38251B4C8C210841C60CDE0B7E4C7A87

Les 3 mots de passe sont composés de 8 chiffres, on teste donc une résolution par bruteforce en hachant successivement de `00000000` à `99999999`.

Pour obtenir le temps de traitement, exécuter le fichier `Bruteforce.java`. Output :

```
All found :
- Clara : 17441945
- Alice : 45239142
- Bob : 39175522
Temps de recherche :
- Clara : 219s
- Alice : 575s
- Bob : 500s
```

On a un temps de recherche de 8 minutes et 20 secondes, pour trois mots de passe de 8 chiffres -> c'est particulièrement long.

Reduction

On implémente dans `Reduction.java` une fonction de réduction et la réduction successive 1000 fois dans la fonction `CalculChaine`. On teste la chaîne `1`, on a le bon output :

```
Chaine réduite obtenue à partir de la chaine "1" : 43183201
```

Génération de la rainbow

On implémente dans `Rainbow.java` la génération de la rainbow table. Elle output tous les 10000 éléments (tous les 1%) le couple (`px`,`p999`) dans la console pour s'assurer périodiquement de la conformité (en terme de format) des réductions :

```
1%. Noeud actuel : 13133487,90587456
2%. Noeud actuel : 29450375,71304037
3%. Noeud actuel : 710203,29317610
...
97%. Noeud actuel : 37941068,93695194
98%. Noeud actuel : 67761709,4607164
99%. Noeud actuel : 48195747,42706111
Temps de génération : 807s, environ 13min
```

On souhaitait générer une table de 1000003 entrées :

- D'une taille première, pour éviter que la fonction de réduction (qui fonctionne avec des modulos) crée de trop nombreuses collisions à cause des diviseurs communs.
- Sans ajouter de doublons, on a donc au final une table de 165901 entrées.

Recherche dans la rainbow

`Cracking.java` implémente la recherche dans la rainbow table selon le mécanisme suivant :

- L'utilisateur rentre un hash
- On calcule la réduction de ce hash
- On recherche dans la rainbow table si un `p999` correspond
- Si on n'a aucune correspondance, on hash puis réduit à nouveau notre première réduction et on recommence la recherche
- On fait ça jusqu'à trouver un `p999` correspondant ou jusqu'à avoir fait 1000 itérations
- Si on a trouvé un `p999` correspondant, on hash puis réduit à partir du `px` correspondant, jusqu'à tomber sur la réduction qui, une fois hashée, correspond au hash de départ.

On peut mesurer l'efficacité de cette méthode en comparant le temps de recherche avec le temps de bruteforce :

- **Clara :**

  ```
  ########## CHARGEMENT DE LA TABLE DEPUIS LE FICHIER : table2.txt ##########
  Entrer un hash MD5 :
  38251B4C8C210841C60CDE0B7E4C7A87
  ```

```
Pass is : 3169710 ?
Pass is : 78013224 ?
Pass is : 95667626 ?
...
Pass is : 78773739 ?
Pass is : 32538545 ?
Pass is : 17441945 ?
Pass cracked : 17441945
Inital hash : 38251B4C8C210841C60CDE0B7E4C7A87
Temps d'exécution : 1012ms
```

C'est 219 fois plus rapide que le bruteforce.

- **Alice :** Temps de recherche de 2979ms (~3s), presque 200 fois plus rapide que le bruteforce.
- **Bob :** Le hash de Bob n'est pas dans la rainbow table, et on reçoit donc l'exception `PASS NOT FOUND`. Ce n'est pas particulièrement suprenant, une rainbow table n'est jamais exhaustive. Une solution serait de générer une nouvelle rainbow table, plus grande. Au final, il s'agit toujours d'un compris complexité/exhaustivité.
- **Bonus :** On peut aussi démontrer que notre fonction de cracking fonctionne avec d'autres hash comme :

  ```
  Pass cracked : 12345678
  Inital hash : 25D55AD283AA400AF464C76D713C07AD
  Temps d'exécution : 2s
  ```

  ```
  Pass cracked : 88888888
  Inital hash : 8DDCFF3A80F4189CA1C9D4D902C3C909
  Temps d'exécution : 3s
  ```

La conclusion est sans appel, la rainbow table est beaucoup plus efficace que le bruteforce, même si elle n'est pas exhaustive.