IUT de Montpellier **SAE2 : découverte des tables de hachages**[1]
M3103 Algorithmique avancée

Consignes avant de commencer :

- récupérer les fichiers "LeRougeEtLeNoir.txt", "randomWords", "randomWordsPetit", et "ListeBigI.java"
- date de rendu sur moodle : le dimanche 16 janvier 23h59

1 Contexte

Les fonctions de hachage sont un outil classique de l'algorithmique[2], dont une des principales application est la suivante. On souhaite créer une structure de donnée T (pour table) pour y stocker un ensemble d'objets $S \subseteq U$, dans un contexte où $|S|$ est très petit devant $|U|$. Typiquement:

- (Exemple 1) S représente l'ensemble des mots utilisés dans un texte donné, et U est l'ensemble des chaînes d'au plus 30 caractères (en considérant que tous les mots utilisés dans le texte sont de longueur au plus 30).
- (Exemple 2) S représente un ensemble de positions de sudoku 9x9 déjà rencontrées par un algorithme d'exploration de type "brute force" (chaque élément de S est une matrice de 9×9 remplie d'entiers entre 1 et 9), et U est l'ensemble de toutes les matrices 9×9 à coefficients entre 1 et 9.
- (Exemple 3) S représente l'ensemble des adresses IP des personnes ayant visité un site web donné, et U représente toutes les adresses IP possibles.

On aimerait définir T pour avoir les deux propriétés suivantes :

- (Propriété p_1) on voudrait que la taille totale utilisée par la table $|T|$ ne soit pas trop grande par rapport au nombre d'éléments $|S|$ à stocker : idéalement $|T| \leq c|S|$, ou même $|T| \leq c|S|^2$, avec c une petite constante, mais surtout pas $|T|$ proche de $|U|$
- (Propriété p_2) on voudrait que les opérations de base (insertion, suppression, recherche) soient très rapides, au sens où elles n'exécutent que très peu d'opérations (par exemple un nombre constant, c'est-à-dire borné par une certaine constante, ou logarithmique en $|T|$).

On considérera dans ce sujet l'exemple 1, qui est un cas dit *statique* dans lequel on connaît S à l'avance. L'objectif est donc d'écrire un algorithme qui, étant donné un fichier de texte en entrée, construit une table T ayant si possible les propriétés p_1 et p_2. Cette table sera alors soumise à un très grand nombre de recherches, l'objectif étant de minimiser le temps total de ces recherches (grâce à la propriété p_2), tout en garantissant un temps de construction et un espace mémoire utilisé raisonnable (grâce à la propriété p_1).

Question 1.
Ici, $|S|$ est donc le nombre de mots (différents) du texte, et U l'ensemble de toutes les chaînes d'au plus 30 caractères. Sachant qu'il y a 256 caractères possibles, combien vaut $|U|$? Remarque : on pourrait certes majorer plus finement $|U|$ en ne comptant que les caractères utilisés dans les mots (et donc pas $, *, par exemple), mais même avec au plus 32 caractères (les 26 lettres et quelques lettres avec accents), combien vaudrait $|U|$?

[1] Merci à Bruno Grenet pour les discussions sur les hashtables!
[2] Voir par exemple `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf`

2 Principe général d'une table de hachage

Les tables de hachage sont des structures de données typiquement utilisées pour obtenir ces deux propriétés. Comme c'est souvent le cas dans la littérature, nous allons d'abord nous intéresser au cas où $U \subseteq \mathbb{N}$. Autrement dit, on s'intéresse au problème de construire une structure de données pour stocker un sous ensemble d'entiers $S \subseteq U$, tout en garantissant les propriétés p_1 et p_2. Nous verrons ensuite comment ramener notre problème de stockage de mots d'un texte au problème de stockage des entiers, en convertissant chaque mot en un "grand" entier.

Question 2.
Supposons que $U = [0, x]$. Quelle structure de donnée très simple permet d'avoir la propriété p_2 ? (en s'autorisant une structure de taille $|U|$ entiers .. et donc ne respectant pas p_1). Notez que, lorsque x est petit (disons $x = 10000$), cette structure simpliste fait très bien l'affaire!

Voici maintenant le principe d'une table de hachage avec résolution dite *par liste (ou par chaînage)*. On commence par choisir une taille de tableau notée m, et une fonction $h : U \to [0, m-1]$ appelée *fonction de hachage* (nous verrons comment choisir h après). On définit alors T comme un tableau de m listes (cf Figure 1).

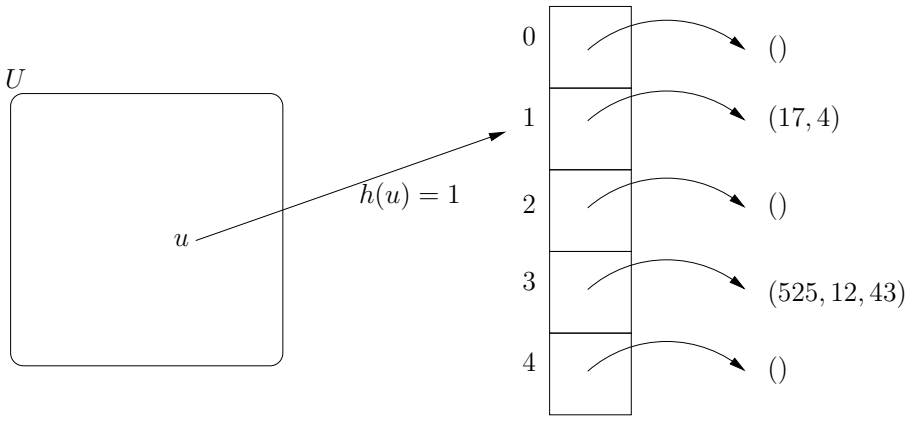

Figure 1: Exemple d'une table de hachage avec résolution par chaînage pour $m = 5$. Cette table contient les éléments $\{17, 4, 525, 12, 43\}$. L'élément u sera rajouté dans la liste $T[1]$.

Ensuite,

- pour ajouter un élément $u \in U$ (qui n'est pas déjà dans T), on ajoute u dans la liste $T[h(u)]$
- pour rechercher un élément $u \in U$, on se contente de chercher seulement dans la liste $T[h(u)]$

Observez que pour tout $i \in [0, \ldots, m-1]$, $T[i]$ contient donc la liste de tous les entiers de la table $u \in U$ tels que $h(u) = i$. On peut faire l'analogie suivante entre T et un dictionnaire. Imaginez que U contienne des chaînes de caractères composées uniquement de lettres minuscules, et que h soit à valeur dans $\{'a', \ldots, 'z'\}$. Supposons que, étant donnée une chaîne $s \in U$, on définisse $h(s)$ comme la première lettre de s. Alors, pour ajouter un nouveau mot $s \in U$ dans le dictionnaire, on regarde la première lettre $h(s)$ et on rajoute s à la liste des mots commen⊂ant par $h(s)$; et pour rechercher un mot $s \in U$, on se contente de chercher seulement dans la liste des mots débutant par la lettre $h(s)$.
On dit que deux éléments $u_1 \neq u_2$ de U sont en *collision* si $h(u_1) = h(u_2)$. Notez que deux éléments distincts ($u_1 \neq u_2$) peuvent tout à fait être en collision, et que tous les éléments d'une liste $T[i]$ sont d'ailleurs en collision. En revenant à l'analogie du dictionnaire, on constate que, par exemple, tous les mots commen⊂ant par la lettre 'a' sont en collision. A noter enfin qu'il existe d'autres techniques que les listes pour gérer les collisions, mais que pour commencer nous allons coder ce type de tables de hachage.
On aimerait une fonction h qui

- soit rapide à calculer (un nombre constant d'opérations typiquement), sinon la propriété p_2 serait déjà compromise,

- qui "répartisse bien" les éléments, au sens où, après avoir ajouté tous les éléments de S, on aimerait que chaque liste $T[i]$ contienne environ $\frac{|S|}{m}$ éléments.

Ainsi, en choisissant $m = |S|$ par exemple, on espère que chaque liste soit de taille constante, c'est-à-dire bornée par une certaine constante, et ainsi on aura les propriétés p_1 et p_2 [3].

3 Codage d'une première version naïve d'une table de hachage avec résolution par liste

Dans cette section nous allons écrire une classe `HTNaive` définie ainsi. L'utilisateur choisira le nombre de listes m (et donc la taille du tableau), et on définit h à valeur dans $\{0, \ldots, m-1\}$ par $h(u) = u\%m$. Attention, c'est une mauvaise pratique, car les "vraies" données ne sont souvent pas aléatoires, et dans le pire cas on pourrait imaginer que de très nombreux éléments se retrouvent en collision. Il faudrait au minimum choisir m aléatoirement (alors qu'ici on laisse à l'utilisateur la responsabilité de le faire), et même cela est insuffisant (cf Section bonus à venir). Discutons à présent la question du choix du type pour stocker nos entiers. On vous rappelle que les types `int` et `long` de Java sont définis sur 4 et 8 octets respectivement.

Question 3.
Etant donné $|U|$ calculé précédemment (avec 256 caractères possibles et pas 32), combien d'octets seraient nécessaires pour stocker un élément de $u \in U$?

La réponse à la question précédente nous montre que les types `int` et `long` ne sont pas suffisants. Il nous faut donc utiliser un type dit *multiprécision*, dont la taille n'est pas bornée et s'adapte à la valeur à stocker. Java nous fournit pour cela le type `BigInteger`, qui se manipule ainsi.

```
import java.math.BigInteger;
import java.util.Random;


BigInteger b1 = BigInteger.valueOf(12);
BigInteger b2 = BigInteger.valueOf(5);
BigInteger s = b1.add(b2); //s = 17
s = s.multiply(b2); // s = 17*5=85
int x = b1.compareTo(b2); //x vaut -1, 0 ou 1 selon que
                          //que b1 est <, = ou > à b2
BigInteger i = b1.remainder(b2)); //i = b1%b2 = 2
int i2= i.intValue();//conversion en int .. qui est effectuée sans
                     //perte seulement quand i tenait sur 4 octets!
```

On va donc maintenant écrire classe `HTNaive` correspondant à une table de hachage avec résolution par chaînage, et stockant des `BigInteger`. La table a comme propriété (invariant) qu'aucune des listes ne contient de doublons. Vous avez à votre disposition la classe `ListeBigI` qui correspond à classe Liste vue en TD, mais où l'on a remplacé `int` par `BigInteger`.

Question 4.
Déclarer une classe `HTNaive`, choisir l'attribut (un seul normalement), et fournir le constructeur `public HTNaive(int m)` qui construit une table de hachage avec m listes vides.

Question 5.
Ajoutez la méthode suivante (utile pour les tests automatiques)

[3] Pour la culture, on connaît des propriétés permettant ce bon comportement de répartition, les plus classiques étant *l'uniformité* et *l'universalité*. Le lecteur minutieux de `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf` constatera que ce ne sont pas des propriétés définies pour une fonction h, mais pour une famille de fonctions.

```
  public ListeBigI getListe(int i){
        return t[i];
    }
}
```

Question 6.
Définir la méthode `private int h(BigInteger u)` définie par $h(u) = u\%m$.

Question 7.
Définir la méthode `public boolean contient(BigInteger u)` qui retourne vrai ssi u est dans la table.

Question 8.
Définir la méthode `public boolean ajout(BigInteger u)` qui, si u était déjà présent dans la table alors ne fait rien et retourne faux, et sinon ajoute u et retourne vrai.
Définir également `public void ajoutListe(ListeBigI L)` qui ajoute à la table les éléments de L qui n'y sont pas déjà. Attention, L ne doit pas être modifiée.

Question 9.
Définir la méthode `public ListeBigI getElements()` qui retourne la liste de tous les éléments de la table.

Question 10.
Définir la méthode `public String toString()` qui retourne une chaine de la forme:
t[0] : ... (éléments de la liste 0)
t[1] : ... (éléments de la liste 1)
etc

Question 11.
Faites vos premiers tests!

.

Ajoutons maintenant quelques méthodes qui nous serviront à évaluer l'état de la table.

Question 12.
Définir les méthodes `public int getNbListes()` qui retourne le nombre de listes (vides ou non) de la table,
`public int getCardinal()` qui retourne le nombre d'éléments stockés dans la table, et
`public int getMaxSize()` qui retourne la taille de la liste la plus longue de this.
Notez que l'on aimerait que getMaxSize() reste petit, idéalement proche de getCardinal()/getNbListes(). Notez également que, si l'on souhaitait rendre instantané le calcul de ces deux dernières méthodes, alors on choisirait d'ajouter des attributs `card` et `maxSize`, que l'on maintiendrait à jour à chaque opération (mais ne le faites pas, car on s'autorise à ce que ces méthodes ne soient pas très rapides, puisque l'on est pas supposé les appeler souvent).

Question 13.
Définir la méthode `public String toStringV2()` qui par exemple pour une table où seules les listes t[3], t[7] et t[10] sont non vides, et de longueurs respectivement 5, 8, et 3 retourne une chaine de la forme:
t[3] : *****
t[7] : ********
t[10] : ***

Pour terminer, on va ajouter deux autres constructeurs.

Question 14.
Définir un constructeur `public HTNaive(ListeBigI l, int m)` qui construit une table ayant m listes contenant les entiers de l.

Question 15.
Définir un constructeur `public HTNaive(ListeBigI l, double f)` qui, étant donné $f > 0$ construit une table ayant $m = |l| \times f$ listes, où $|l|$ est le nombre d'éléments différents dans la liste l, et contenant les éléments de l. Attention, $|l|$ peut être différent de $l.longueur()$.
Remarques :

- Une solution pour calculer $|l|$ est de créer une instance $temp$ de la classe HTNaive (avec m' arbitraire, égale à 1000 par exemple), d'y ajouter les éléments de la liste l, et de récupérer le nombre d'éléments stockés dans $temp$.
- Pour une liste l de $n = 500000$ entiers, on prendra par exemple $f \in \{1, \frac{1}{2}, \frac{1}{10}, \frac{1}{100}\}$. Plus f est grand, plus la table est grande, mais plus sa taille maximum de liste $getMaxSize()$ a de chances d'être petite, puisque idéalement $getMaxSize() = \frac{|l|}{m} = \frac{1}{f}$.

On notera au passage que l'astuce pour calculer $|l|$ est en fait un grand classique pour calculer le nombre d'éléments différents d'une liste l. En effet, un algorithme naïf consisterait à vérifier l'égalité entre toutes les paires, et donc nécessiterait de l'ordre de $l.longueur()^2$ opérations, alors que la technique à base de table de hachage (en choisissant cette fois $m' = l.longueur()$), va en moyenne nécessiter de l'ordre de $l.longueur()$ opérations (sous réserve que la répartition dans les différentes listes soit équilibrée "en moyenne"). Voilà donc au passage une autre application des fonctions de hachage !

4 Codage de la classe Dictionnaire

On va maintenant écrire la classe `Dictionnaire` permettant de stocker des mots (`String`) en se basant sur la classe `HTNaive` de la section précédente.

Question 16.
Ecrire une classe `Dictionnaire` ayant comme unique attribut une instance de la classe `HTNaive`, et un constructeur `public Dictionnaire(int m)` permettant de choisir la taille m de la table sous-jacente.

Pour établir un lien entre mots et entiers, on va définir une fonction (statique) `BigInteger stringToBigInteger(String s)` qui associe à une chaine son entier correspondant. Ainsi, étant donné une String s que l'on veut ajouter dans notre dictionnaire, on va calculer `u = Dictionnaire.stringToBigInteger(s)`, puis ajouter u à la table. On suivra la même idée pour la recherche.
Pour une chaîne $s = c_k \dots c_0$, on va interpréter la chaîne comme un nombre écrit en base 256, et donc définir $stringToBigInteger(s) = x_k 256^k + x_{k-1} 256^{k-1} + .. + x_0 256^0$, où x_i) est le code ASCII de c_i (compris entre 0 et 255).

Question 17.
Définir une méthode `private static BigInteger stringToBigInteger(String s)` qui calcule la fonction comme définie ci-dessus. Indications : `char c = s.charAt(i)` permet de récupérer le caractère de rang i de s (à partir du rang 0), et `int x = c` entraîne la conversion de c en int, avec $0 \leq x \leq 255$ (x est le code ASCII de c).

On rappelle que l'on souhaite que la méthode de Dictionnaire `boolean contient(String s)` (que l'on n'a pas encore écrite) soit très rapide. Etant donné que cette méthode fait appel à `stringToBigInteger`, cette dernière méthode doit aussi être très rapide.

Question 18.
Si l'on calcule naïvement chaque terme $(x_i 256^i)$ indépendamment, en considérant que le calcul de 256^i demande $i-1$ multiplications pour chaque $i \geq 1$, combien de multiplications va t-on faire pour convertir une chaîne $s = c_k \dots c_0$ en entier (en fonction de k) ?
Si ce n'est pas déjà le cas, ré-écrire votre méthode pour avoir un nombre de multiplications linéaire en k, c'est-à-dire inférieur ou égal à Ck pour une certaine constante C. Indication : appliquer la méthode de Horner qui n'effectue que k multiplications en remarquant que $stringToBigInteger(s) = (\dots((x_k 256 + x_{k-1})256 + x_{k-2})\dots)256 + x_0$.

Question 19.
Ecrire la méthode `public boolean ajout(String s)` qui, si s était déjà présent dans le dictionnaire alors ne fait rien et retourne faux, et sinon ajoute s et retourne vrai.

Question 20.
Ecrire la méthode `public boolean contient(String s)` qui retourne vrai ssi s est dans le dictionnaire.

Question 21.
Ecrire les méthodes `public int getCardinal()`, `public int getMaxSize()`, `public int getNbListes()`, `public String toString()`, `public String toStringV2()`, dont les spécifications sont similaires à celles des méthodes correspondantes sur la table sous-jacente.

Question 22.
Testez

On va maintenant s'occuper de charger tout un texte depuis un fichier. Pour ce faire, nous utiliserons les classes File et Scanner de la façon suivante.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public int lectureMotsTexte(){
  String filename = "fichier.txt";
       //on suppose que fichier.txt est un fichier dans le meme dossier
       //que les .java
  File f = new File(fileName);
  ListeBigI res = new ListeBigI();
  Scanner sc;
       //un scanner est un objet permettant de "scanner" (parcourir)
       //une entrée (clavier, ou une chaîne, ou un File, etc)
  try {
      sc = new Scanner(f);
       //ici on construit le scanner avec comme entrée f
       //cette construction peut échouer (si par exemple fichier.txt n'existe pas)
  }
  catch(FileNotFoundException e){
                          //si la construction échoue, on passe ici
      System.out.println(("problème d'accès au fichier " + e.getMessage()));
      return 0;
  }
  sc.useDelimiter(" |\\n|,|;|:|\\.|!|\\?|-");
       //on définit les délimiteurs comme le caractère '\n', le caractère ',' etc...
       //cela définit maintenant la notion de "morceau" comme une suite
       //de caractères entre deux délimiteurs
  int nbmots = 0;
  while (sc.hasNext()) { //sc.hasNext() renvoie vrai ssi
                         //il reste encore un morceau à découvrir dans f
          String mot = sc.next(); //sc.next() renvoie le prochain morceau
          nbmots ++;
  }
  return nbmots;
}
```

Question 23.
Définir une méthode `public static ListeBigI calculeListeInt(String fileName)` qui retourne la liste de tous les entiers correspondant aux mots du fichier $fileName$ (y compris les éventuels doublons).

Question 24.
Définir un constructeur `Dictionnaire(String fileName, int m)` qui construit un dictionnaire basé sur une table de m listes contenant les entiers correspondant aux mots différents du fichier $fileName$.
Définir aussi un constructeur `public Dictionnaire(String fileName, double f)` faisant de même, mais basé sur une table de $m = |fileName| \times f$ listes, où $|fileName|$ est le nombre de mots différents du fichier.

5 On teste le tout!

On s'intéresse au `main` suivant qui construit un dictionnaire d (à partir d'un fichier, et avec des paramètres que l'on va ajuster), effectue un grand nombre de recherches de mots tirés au hasard, et affiche le temps total passé à faire ces recherches.

```
Dictionnaire d = ...;
System.out.println("maxSize : " + d.getMaxSize());
System.out.println("cardinal : " + d.getCardinal());
System.out.println("nbListes : " + d.getNbListes());
int nbRecherches = 100000;

deb=System.currentTimeMillis();
for(int i=0;i<nbRecherches;i++) {
    int tailleMot = random.nextInt(15) + 2; //2 <= tailleMot <= 16
    char[] mot = new char[tailleMot];
    for (int j = 0; j < mot.length; j++) {
        mot[j] = (char) ('a' + random.nextInt(26));
    }
    String motS = new String(mot);
    d.contient(motS);//on ne récupère même pas le résultat de la recherche!
}
fin=System.currentTimeMillis();
System.out.println("temps total : " + (fin-debut));
```

Question 25.
Testez ce code à partir du fichier randomWordsPetit avec un dictionnaire construit avec $m \in \{1, 2, 4, 8\}$. Observez

- si le temps est environ divisé par 2 quand m est multiplié par 2,
- si le $d.getMaxSize()$ est environ égal à $d.cardinal()/d.getNbListes()$ pour chaque valeur de m.

Question 26.
Testez ce code à partir du fichier randomWords avec un dictionnaire construit avec $f \in \{0.01, 0.1, 0.5, 1\}$. Avec ces paramètres, on espère avoir $d.getMaxSize()$ très petit (de l'ordre de la centaine pour $f = 0.01$, ce qui est "très petit" à l'échelle de l'ordinateur !), et donc les propriétés p_1 et p_2. Vérifiez donc que plus f est grand, plus $d.getMaxSize()$ est petit, et plus le temps total est court.

Question 27.
Combien de mots différents sont utilisés dans "LeRougeEtLeNoir.txt" ?

Question 28.
Modifier `HTNaive` afin de pouvoir déterminer plus précisement combien de temps est passé au total à effectuer le calcul de la fonction `h`, et à effectuer des appels à la méthode `contient` de la classe `ListeBigI`. Indication, rajouter deux attribut `private long totalTimeh` et `private long totalTimeContient`.