INF 421 Luc Maranget

Associations

Luc.Maranget@inria.fr

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Les tables d'associations

- Qu'est-ce que c'est ? Des associations de n'importe quoi à n'importe quoi.
- Comment ça marche ? Trois implémentations.
  - Avec des listes.
  - Avec notre table de hachage.
  - Avec les *tables de hachage* de la bibliothèque.

La table d'association

Soit des « *informations* », munies d'une « *clé* ».

- La clé identifie d'information.
  - Nom (et prénom) → numéro de téléphone dans l'annuaire.
  - Matricule → soldat.
  - Numéro d'immatriculation → véhicule, dans le fichier des cartes grises.
- On se place dans le cas où la clé identifie une information unique (par ex. fichier des cartes grises).

La table d'association

Ensemble *dynamique* d'informations.

Type abstrait de données, défini par les opérations.

- Trouver l'information associée à une clé donnée.
- Ajouter une nouvelle association entre une clé et une information.
- Retirer une clé de la table (avec l'information associée).

Et plus précisément (unicité des clés)

- S'il existe déjà une information associée à la clé dans la table, alors la nouvelle information remplace l'ancienne.
- Sinon, une nouvelle association est ajoutée à la table.

Application

On veut produire une statistique des mots d'un texte.

- Voici par exemple un texte bien connu.

```
% cat marseillaise.txt
Allons ! Enfants de la Patrie !
Le jour de gloire est arrivé !
etc.
```

- Statistique des fréquences des mots de la Marseillaise (mots de taille ≤ 4, majuscules enlevées).

```
% java Freq marseillaise.txt
nous: 10
vous: 8
français: 7
  etc.
```

Conception de `Freq`

En trois étapes.

1. Lire le texte mot-à-mot.
2. Compter les occurences des mots.
3. Produire un bel affichage (par fréquences décroisantes).

Plus précisément.

1. Procéder selon le principe du `Reader`.
2. Par une table d'association des mots aux comptes. (clé = un mot → information = un compte).
3. Par un tri à la fin.

Lecture des mots, tri

On suppose donnés :

- Une classe des objets « lecteur de mots » (`WordReader`).
  - Constructeur `WordReader (String name)` pour créer le lecteur des mots d'un fichier de nom `name`.
  - Méthode `String read()`, renvoie le mot suivant (ou **null** quand c'est fini).
- Un tri des paires, mot × compte (notées $w \rightarrow c$). Selon l'ordre total :

$$(w_1 \rightarrow c_1) < (w_2 \rightarrow c_2)$$

$$\Updownarrow$$

$$(c_1 > c_2) \vee (c_1 = c_2 \wedge w_1 < w_2)$$

Définition précise de notre table

Une table est un objet `Assoc`. C'est une table d'association des mots (`String`) aux **int**.

- Méthode **int** `get(String w)`, renvoie le compte associé à `w`.
  - Ou bien 0 si `w` n'est pas dans la table.
- Méthode **void** `put(String w, int c)`, associe le compte `c` au mot `w`.

Spécification en Java, par une *interface* (commentée).

```
interface Assoc {
  /* Créer/remplacer l'association key → val */
  void put(String key, int val) ;
  /* Trouver l'entier associé à key, ou zéro. */
  int get(String key) ;
}
```

Code de main

```
class Freq {
  ...
  static void countFile(String name, Assoc t) {
    WordReader wr = new WordReader(name) ;
    count(wr, t) ;
    wr.close() ;      // Fermer l'entrée (plus propre)
    Sort.println(t) ; // Affichage trié
  }

  public static void main(String [] arg) {
    countFile(arg[0], ...)
  }

}
```

Code de `count`

```
static void count(WordReader in, Assoc t) {
  for (String word = in.read() ;
       word != null ;
       word = in.read()) {
    if (word.length() >= 4) {
      word = word.toLowerCase() ;
      t.put(word, t.get(word)+1) ;
    }
  }
}
```

Remarquer : `Assoc` n'est pas une classe, mais on peut l'utiliser comme un type, celui du deuxième argument de `count`.

Les listes d'associations

La façon la plus simple d'associer un compte à un mot ?

Une liste de paires `String` → **int**.

```
class AList {
  private String word ;
  private int count ;
  private AList next ;

  AList (String word, int count, AList next) {
    this.word = word ; this.count = count ; this.next = next ;
  }
}
```

Attention

- On suppose qu'à un mot est associé au plus un compte.
- La table vide est représentée par… **null**.

La table avec une liste

Étant donnés un mot (`String`) w et une liste d'association (`AList`) p. On suppose écrite une méthode `lookCell(`p`, `w`)` qui renvoie

- La cellule de liste q de la liste p, telle que q`.word` égal à w.
- Ou **null**, sinon.

On peut maintenant écrire recherche et mise à jour (`get`/`put`),

Recherche

Écrivons une méthode `get` *statique* (Pourquoi ? **null** est une liste).

- `get` prend une `AList` et un mot w et renvoie le compte.
- Trouver la cellule q, avec q`.word` égal à w.
- Si q est **null**, renvoyer `0`.
- Sinon, renvoyer q`.count`

```
class AList {
    ...
  static int get(AList p, String w) {
    AList q = lookCell(p, w) ;
    if (q == null) return 0
    else return q.count ;
  }
}
```

Creation/Mise à jour d'une association de la liste

Écrivons une autre méthode *statique* dans la classe `AList`.

- `put` prend une `AList` p, un mot w, un compte c, et renvoie la table augmentée de la paire $(w \rightarrow c)$.
- Trouver la cellule q.
- Si q est **null**, ajouter une association.
- Sinon, modifier l'association et renvoyer la table modifiée.

```
static AList put(AList p, String w, int c) {
  AList q = lookCell(p, w) ;
  if (q == null) {
    return new AList (w, c, p) ;
  } else {
  q.count = c ;
  return p ;
}}
```

Code de lookCell

```
static AList lookCell(AList p, String w) {
  for ( ; p != null ; p = p.next) {
      if (w.equals(p.word)) {
        return p ;
      }
  }
  return null ; // Pas trouvé
}
```

Remarquer le **return** dans la boucle, et le **return** après la boucle.

Résumé de la classe AList

```
class AList {
  ⋮ // Un constructeur, la méthode lookCell
  static int get(AList p, String w) { ... }
  static AList put(AList p, String w, int count) { ... }
}
```

Un objet de la classe AList peut-il prendre Assoc comme type ?

```
interface Assoc {
  int get(String key) ;
  void put(String key, int val) ;
}
```

Non, essentiellement parce que get et put sont des méthodes statiques de la classe AList.

Il faut encapsuler

```
class L implements Assoc {
  private AList p ;

  L() { p = null ; }

  public int get(String w) { return AList.get(p, w) ; }

  public void put(String w, int c) { p = AList.put(p, w, c) ; }
}
```

- On peut demander au compilateur `javac` de vérifier qu'un objet `L` peut prendre le type `Assoc`. On dit que la classe `L` *implémente* l'interface `Assoc`.
- Les méthodes déclarées dans l'interface sont obligatoirement **public** (c'est comme ça).

Bénéfice

On peut maintenant écrire :

```
class Freq {
  ...
  public static void main(String [] arg) {
    countFile(arg[0], new L()) ;
  }
}
```

Limitation

La classe `L` n'est pas très efficace.

- Pour N mots distincts $\rightarrow O(N^2)$.

Vrai bénéfice (de l'interface)

Nous pouvons fabriquer une autre implémentation (`H`) de l'interface `Assoc` (plus efficace que `L`), et l'utiliser à la place.

Par exemple, un programme de test des diverses implémentations des tables d'association :

```
// java Freq -L file -> liste d'association
// java Freq -H file -> autre implémentation
class Freq {
  ...
  public static void main(String arg) {
    if (arg[0].equals("-L")) countFile(arg[1], new L()) ;
    else if (arg[0].equals("-H")) countFile(arg[1], new H()) ;
    ...
  }
}
```

Détour : une vue des tableaux

Nous connaissons les tableaux :

- Accès : `t[`k`]`
- Mise à jour : `t[`k`] =` ...

Nous pouvons donc voir un tableau comme une table d'association dont les clés sont des **int** dans `[0...t.length[`.

Limitations :

- Les *clefs* k sont nécessairement des entiers,
- pris dans un intervalle donné.

Idée du hachage

Soit une clé quelconque k (pour les mots k est un `String`).

- Se donner une fonction h de *hachage* des clés vers les entiers.
- Si...
  - Le co-domaine de h est de la forme $[0 \ldots m[$,
  - Et h injective $(h(k_1) = h(k_2) \Rightarrow k_1 = k_2)$
- Alors on peut employer un tableau de taille m comme table d'association.

Hachage en théorie

- Transformer les clés en entiers, facile. Par exemple :

  Une chaîne w est vue comme l'écriture en base 2^{16} d'un entier.

- Dans l'intervalle $[0 \ldots m[$, facile.

  Prendre le reste de la division euclidienne par m.

- Fonction de hachage injective, difficile, et surtout contradictoire avec le point précédent.

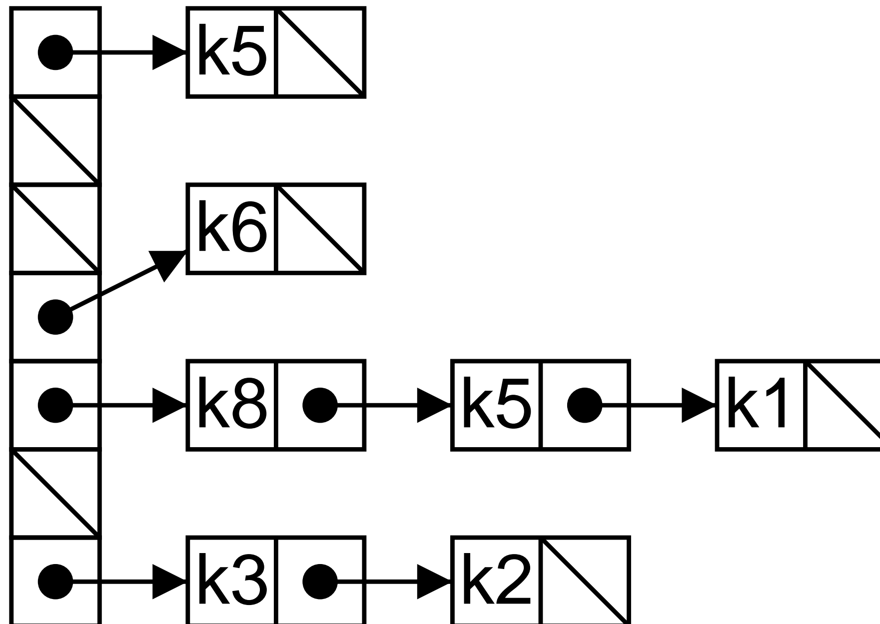
Lorsque $h(k_1) = h(k_2)$ pour $k_1 \neq k_2$, on dit qu'il y a une *collision*.

Idée du hachage II, résolution des collisions

On suppose donc des collisions :

$$\exists k\, k', k \neq k' \text{ et } h(k) = h(k')$$

Le chaînage : la case i du tableau t contient la liste des $(k \to v)$ avec $h(k) = i$.



Implémentation

- Bon à remarquer : les « listes d'informations » sont des listes `AList`.
- Bon à savoir : java fournit une méthode de hachage, ici la méthode **int** `hashCode()` des `String`.

$$h(w) = w.\texttt{hashCode()} \bmod m$$

```
class H implements Assoc {
  private final static int SZ = 1024 ;
  private AList [] t ;
  H() { t = new AList [SZ] ; }

  // Fonction de hachage h.
  private int hash(String w) {
    return Math.abs(w.hashCode()) % t.length ;
  }
```

Chercher dans bonne liste

```
public int get(String w) { // Chercher dans t[h(w)]
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (q == null)
    return 0 ;
  else
    return q.count ;
}
```

Ajouter/Remplacer dans la bonne liste

```
public void put(String w, int c) {
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (r == null)
    t[i] = new AList(w, c, t[i]) ;
  else
    q.count = c ;
}
```

Coût du hachage

On se donne quelques hypothèses.

- Le coût du calcul de h est en $O(1)$ (ou négligé).
- Le hachage est uniforme :

  Pour une table de taille m, $h(w)$ vaut i dans $[0 \ldots m[$ avec une probabilité $1/m$.

Alors, le coût *moyen* d'une recherche/ajout dans une table contenant n associations est en $O(1 + n/m)$ (admis).

Donc si m est de l'ordre de n : $O(1)$.

On note $\alpha = n/m$ le facteur de charge, c'est aussi la longueur moyenne des listes de collision (intuitivement clair ?).

Remarques sur le coût en $O(1)$

- m de l'ordre de n veut aussi dire : facteur de charge α borné par une constante.
- Le coût dans le cas le pire est $O(n)$ (toutes les clés sont en collision).
- Mais adopter le hachage c'est :
  - Faire confiance au hasard (hachage uniforme).
  - Procéder à beaucoup de de `put`/`get` (coût en moyenne significatif du coût en pratique).

    Produire les statistiques d'un texte de N mots :

$$N \times O(1) \sim O(N)$$

Redimensionnement

Ce qui était vrai des piles/files l'est aussi des tables de hachages.

Il est pratique et normal que ce soit le code de la table de hachage qui se charge de maintenir α borné.

```
private final static double alpha = 4.0 ; // borne sur la charge
private int nbKeys = 0 ; // n (m est t.length)

public void put(String w, int c) {
  int i = hash(w) ;
  AList q = AList.lookCell(t[i], w) ;
  if (r == null) {
    t[i] = new AList(w, c, t[i]) ;
    nbKeys++ ; // Car vient d'ajouter une association
    if (t.length * alpha <= nbKeys) // La charge est trop forte
      resize() ;
  } else
    q.count = c ;
}
```

La méthode `resize`

Pour assurer un coût amorti en $O(1)$

- Progression géométrique de la taille du tableau.
- Coût de `resize` proportionnel à la taille du tableau.

Et voilà.

```
private void resize() {
  AList [] old = t ;

  t = new AList [2*old.length] ; // Remplacer d'abord
  for (int i = 0 ; i < old.length ; i++) { // Copier old -> t
    for (AList p = old[i] ; p != null ; p = p.next) {
      int h = hash(p,word) ;
      t[h] = new AList (p.word, p.count, t[h]) ;
    }
  }
}
```

Bibliothèque

Les tables de hachage sont d'un emploi très courant.

La classe de la bibliothèque est générique à deux paramètres : `HashMap <K,V>` (package `java.util`).

- `K` classe des clés (pour nous `String`).
- `V` classe des valeurs (pour nous `Integer`).

Le codage est des plus facile.

```
import java.util.* ;

class Lib implements Assoc {
  private HashMap <String,Integer> t ;
  Lib() { t = new HashMap <String,Integer> () ; }
  ...
}
```

Chercher, remplacer/ajouter dans un HashMap

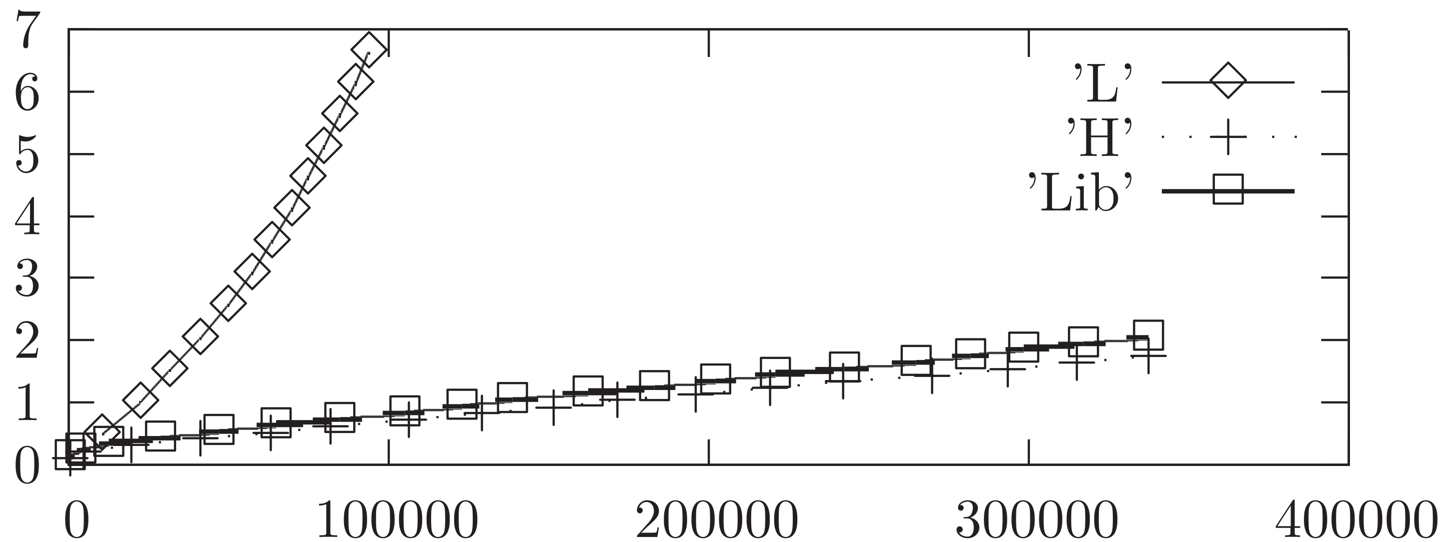
```
public int get(String w) {
 Integer c = t.get(w) ;
 if (c == null) // Si pas d'assoc (w → c) t.get(w) -> null
   return 0 ;
 else
   return c ; // Compris comme 'return c.intValue()'
}

public void put(String w, int c) {
 t.put(w, c) ; // ...c....  -> ... Integer.valueOf(c) ...
}
```

Et c'est tout !

Performance

Faire la statistique des mots de sources Java. Temps d'exécution de `count` (sec.), fonction du nombre de mots (significatifs) lus.



On note :

- Comportement supra-linéaire des listes.
- Comportement linéaire des tables de hachage.

Et au cas où vous vous poseriez la question

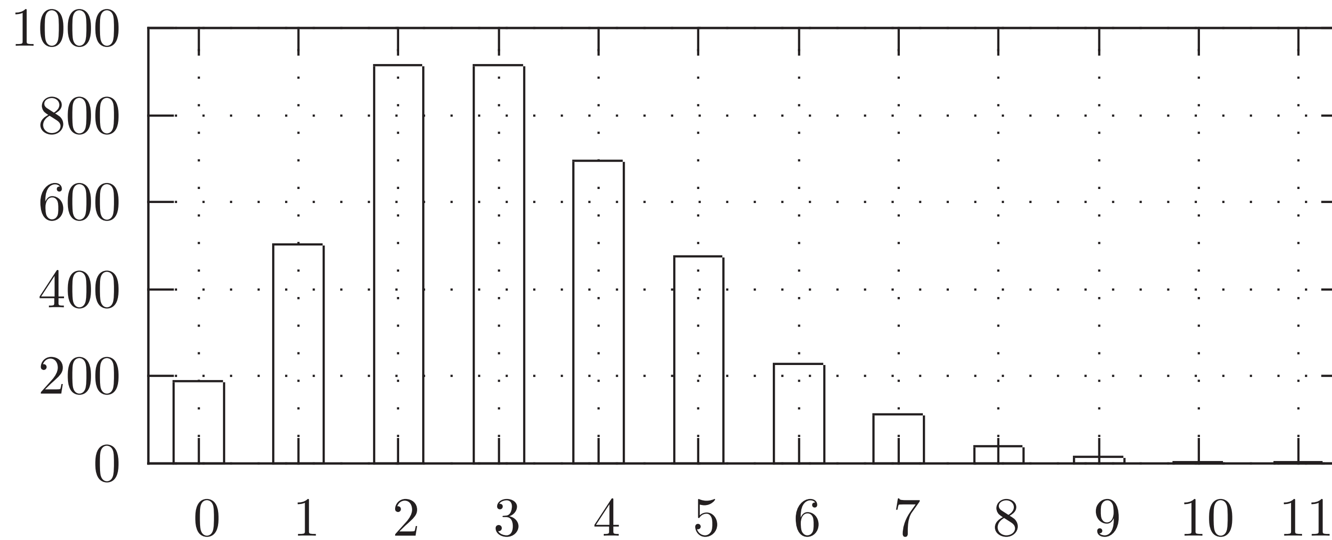
Statistique des sources Java

```
return: 13221       private: 4216       abnorme: 1
static: 9666                            aborted: 1
null: 8191            etc.
string: 6672                              etc.
system: 6526        abandoned: 1
void: 6007          abba: 1             zhoulai: 1
public: 5464        abbcdefgh: 1        ziegler: 1
else: 5376          ability: 1          zones: 1
println: 4650       abnormally: 1       zyvabis: 1
```

Bilan des collisions

L'efficacité provient principalement du « bon » hachage des clés

Histogramme des effectifs des listes de collisions, selon leur taille.
Pour $n = 12904$, $m = 4096$, $\alpha = 3.15$.



Remarque : Diagramme fabriqué par `gnuplot` à partir de données brutes : `2 916`, `4 695`, etc. Nous allons voir comment obtenir ces données.

Hachage des chaînes en Java

Il est précisé dans la documentation.

$$a_0 a_1 \ldots a_{n-1}$$

$$\Downarrow$$

$$a_0 \cdot 31^{n-1} + a_1 \cdot 31^{n-2} + \cdots + a_{n-2} \cdot 31 + a_{n-1}$$

```
public int hashCode() {
  int h = 0 ;
  for (int k = 0 ; k < this.length ; k++)
    h = 31 * h + this.charAt(k) ;
  return h ;
}
```

Idée : Appliquer le « *mélangeur* » $M(x, y) \rightarrow p * x + y$, à x valeur de hachage du préfixe de la chaîne, et y valeur de hachage d'un caractère (lui même ici).

Les tables de hachages sont partout

Statistique de la longueur des listes de collision de la classe `H`.

```
void collisions() {
  HashMap<Integer,Integer> c =
    new HashMap<Integer, Integer> () ;
  // c est une table taille -> effectif
  for (int k = 0 ; k < t.length ; k++) { // Pour toutes les listes
    int len = 0 ;
    for (AList p = t[k] ; p != null ; p = p.next) len++ ;
    // len est la taille de t[k]
    Integer old = c.get(len) ;
    if (old == null) {
      c.put(len,1) ;
    } else {
      c.put(len, old + 1) ;
    }
  }
  // Afficher la table 'c'
}
```

Les interfaces sont partout I

Une interface souvent utilisée : `Iterator<E>` (package `java.util`, doc[a]), un itérateur sur une « *collection* » de `E`.

Deux méthodes intéressantes :

- Reste-t-il un élément ? **boolean** `hasNext()`
- Extraire l'élément suivant : `E next()`

Usage :

```
Iterator<E> it = ...
while (it.hasNext()) {
  E e = it.next() ;
  // Traiter e
}
```

[a] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html

Les interfaces sont partout II

De nombreuses classes de la bibliothèque (par exemple `Set`) implémentent l'interface `Iterable<E>` (package **`java.lang`**, doc[a])

Une méthode :

- Récupérer un itérateur `Iterable<E> iterator()`

Usage :

```
Iterable<E> es = ...
Iterator<E> it = es.iterator() ;
while (it.hasNext()) { E e = it.next() ; ... }
```

Notation abrégée :

```
Iterable<E> es = ...
for (E e : es) {
  ...
}
```

[a] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Iterable.html

Afficher la table des tailles

La table `c` est un `HashMap<Integer,Integer>`.

Les `HashMap<K,V>` possèdent une méthode :

```
public Set<K> keySet()
```

Qui renvoie l'ensemble des clés présentes dans la table. Or, `Set<K>` implémente l'interface `Iterable<K>`. On peut donc écrire :

```
for (Integer k : c.keySet()) {
  System.err.println(k + " " + c.get(k)) ;
}
```

Qui est une notation particulièrement compacte pour :

```
Iterable<Integer> ks = c.keySet() ;
Iterator<Integer> it = ks.iterator() ;
while (it.hasNext()) {
  Integer k = it.next() ;
  System.err.println(k + " " + c.get(k)) ; }
```

Afficher un HashMap<K,V>, la totale

Itérer sur les clés de la table implique une recherche (get) de la valeur associée, c'est bien dommage.

Or, si t est un HashMap<K,V>.

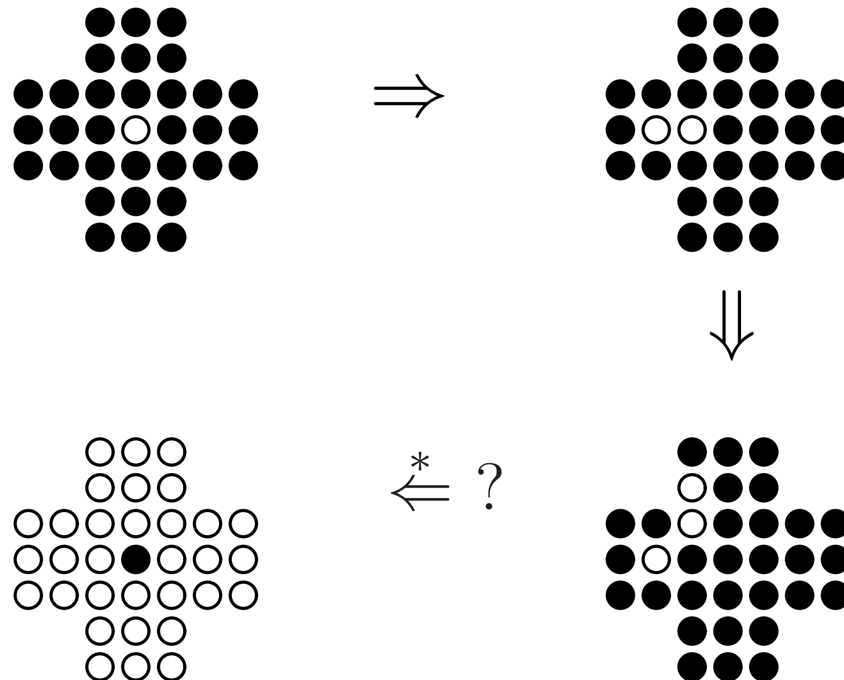
- t.entrySet() renvoie un Set<Map.Entry<K,V>>.
- Map.Entry<K,V> est une interface (paires de K → V).
- Si e est un Map.Entry<K,V> (noté $k \rightarrow v$).
  - e.getKey() pour récupérer la clé k.
  - e.getValue() pour récupérer la valeur v.
- En outre, Set<E> implémente l'interface Iterable<E>.

On peut donc afficher ainsi :

```
for (Map.Entry<Integer,Integer> e : c.entrySet())
  System.err.println(e.getKey() + " " + e.getValue()) ;
```

Autre example d'emploi des associations

- Une petite base de données, par exemple, associer des notes (HC × CC) à des élèves (Nom × Prénom).
- Une mémoire. Soit le jeu du solitaire.



Une configuration du jeu $\sim$ un entier sur 33 bits (ou 49 bits).
— hachage *parfait*, pas de collisions.

Résoudre le solitaire

Soit une configuration C (objet classe `Soli`).

On suppose,

- Un champ **long** `config`, le codage entier d'une configuration.
- Une méthode `Iterable<Soli> play()` qui renvoie les configurations atteignables en un coup à partir de **this**.

Résolution du jeu par force brute.

```
static boolean canWin(Soli C, int nBoules) {
  if (nBoules == 1) return true ;
  for (Soli D : C.play()) {
     if (canWin(D, nBoules-1)) return true ;
  }
  return false ;
}
```

Problème : le programme tourne sans répondre.

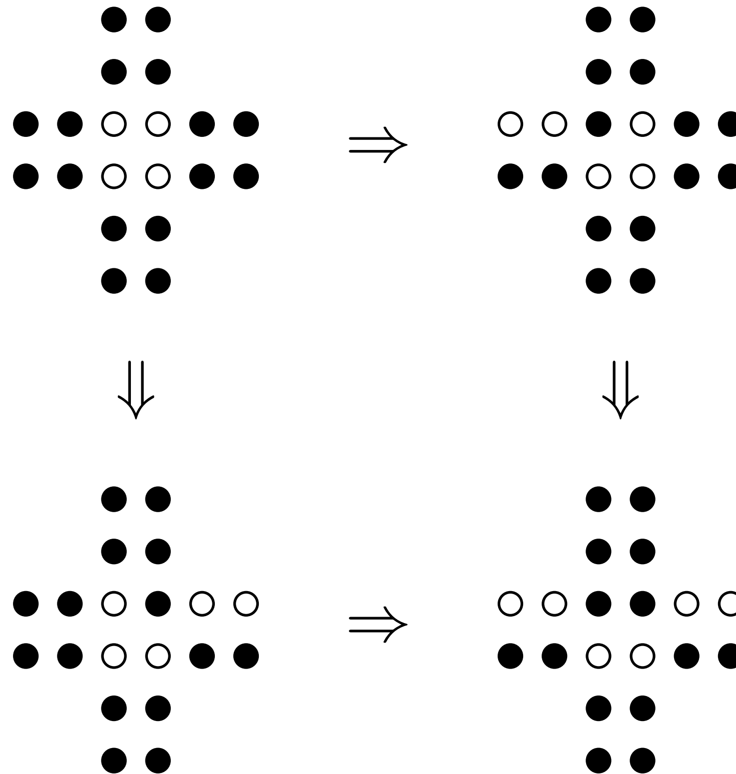
Problème dans la résolution

Lors de l'exploration,...

$$
\begin{array}{cccc}
C & \Rightarrow^* & C_1 & \\
\Downarrow^* & & \Downarrow^* & \\
C_2 & \Rightarrow^* & C' & \Rightarrow^* \ldots
\end{array}
$$

- On retrouve des configurations déjà explorées.
- Et donc des calculs sont refaits ($C' \Rightarrow^* \ldots$).

Par exemple



À gauche, puis à droite. Ou bien, à droite, puis à gauche.

Exploration avec une mémoire

```
static HashSet<Soli> seen = new HashSet<Soli> () ;
// Comme le HashMap, sans les valeurs

static boolean canWin(Soli C, int nBoules) {
  if (nBoules == 1) return true ;
  if (seen.contains(C)) return false ;
  for (Soli D : C.play()) {
    if (canWin(D, nBoules-1)) return true ;
  }
  seen.add(C) ;
  return false ;
}
```

Remarquer :

- Un tableau `seen` aurait 2^{33} (8 GiGa) cases.
- Une table de hachage `seen` a une taille de l'ordre du nombre de configurations effectivement stockées.

Obscur `Object`

Ce n'est pas fini. Quelle fonction de hachage employons nous ?

Autrement dit, quelle est la méthode **`hashCode()`** de nos clés `Soli` ?

- Tout objet (*i.e.* valeur crée par **new** `Class (...)`) peut être vu comme de type `Object`. C'est le *sous-typage*.
- À la naissance, tout objet possède déjà les méthodes définies par la classe `Object`. C'est *l'héritage*.

Exemple de méthode ainsi (pré)-définie pour tous les objets ?

- La méthode `toString`. Mais aussi...
- Les méthodes (publiques) **boolean** `equals(Object o)` et **int** `hashCode()`, bien utiles pour... les tables de hachage,
  - ▷ **`hashCode`** pour trouver la bonne liste de collisions.
  - ▷ **`equals`** pour chercher dedans.

Il faut redéfinir les méthodes

Les méthodes héritées ne nous conviennent pas (égalités et hachage de l'addresse en mémoire).

C'est la la même histoire que `toString` (amphi 01)

- Si o est un objet, le code `out.print(`o`)`, revient à afficher la chaîne renvoyée par l'appel de méthode o`.toString()`.
- Par défaut, o`.toString()` affiche la valeur de o (une flèche).

  `out.print(new Pair(1,2))` ⇒ `Pair@6f0472`

- On *redéfinit* la méthode `toString` (dans la classe `Pair`).

```
public String toString()
  { return "(" + x + ", " + y + ")" ; }
```

- Et alors

  `out.print(new Pair(1,2))` ⇒ `(1, 2)`

Redéfinition de equals

Premier essai :

```
boolean equals(Soli s) {
  return this.config == s.config ;
}
```

S'agit-il d'une *redéfinition* de `equals` des `Object` ?

Non ! Pourquoi ?

- La signature de `equals` des objets est

  ```
  public boolean equals (Object o)
  ```

  Notre `Soli` possède maintenant deux méthodes `equals` : `equals(Soli s)` (ci-dessus) et `equals(Object o)` (héritée).

- En outre, **boolean** `equals(Soli s)` n'est pas publique (et le code de la classe `java.util.HashSet` ne pourrait pas l'appeler).

Redéfinition de `equals (Object o)`

Premier essai (dans la classe `Soli`).

```
public boolean equals(Object o) {
  return this.config == o.config ;
}
```

Est-ce que ça fonctionne ? Non ! Que se passe-t-il ?

La compilateur refuse de compiler la classe `Soli`.

```
% javac Soli.java
Soli.java:15: cannot resolve symbol
symbol  : variable config
location: class java.lang.Object
  return this.config == o.config ;
                         ^
```

Redéfinition de `equals (Object o)`

Dans l'essai précédent, le compilateur n'a aucun moyen de savoir que l'`Object` que l'on lui donne est toujours un `Soli`.

Il faut le lui dire !

```
public boolean equals(Object o) {
  Soli s =
   (Soli)o ; // downcast (conversion de type vers le bas)
  return this.config == s.config ;
}
```

Que se passe-t-il avec *s*.`equals("coucou")` ?

Un échec (à l'exécution) ! Mais plus précisément ?

L'exception `java.lang.ClassCastException` est levée.

Redéfinition de `hashCode ()`

Il s'agit produire un **int** à partir de `config` (un **long**).

```
public int hashCode() {
// Un calcul à partir de this.config
}
```

Plus précisément, nous avons un ensemble de clés k :

Des **long** pas franchement arbitraires.

Et nous voulons :

Une répartition uniforme des $h(k)$ dans $[0 \ldots m[$.

Pas si facile…

- L'espace des configurations n'est pas bien connu.
- La taille m est variable au cours du temps.
- Et d'ailleurs `hashCode` ne connaît pas m.

Changement de point de vue

Une (bonne) fonction de hachage h des **long**, vers les **int** .

- Se calcule pour un coût modique.
- Est surjective.
- Ressemble à une fonction aléatoire, la plus quelconque possible, quand on l'applique à 1, 2, etc.
- Pour des clés « proches » produit des résultats « éloignés »
- Par ex, changer un bit dans la clé, doit changer la moitié des bits de $h(k)$.
- D'autres critères du même genre etc.

Avec une telle fonction, la valeur de hachage finale est

$$h(k) \bmod m$$

Compliqué, imitons bêtement `hashCode` des chaînes

Considérons un **long** comme une chaîne de 4 **short**.

$$\underbrace{b_0 b_1 \cdots b_{15}}_{B_0}\underbrace{b_{16} b_{17} \cdots b_{31}}_{B_1}\underbrace{b_{32} b_{33} \cdots b_{47}}_{B_2}\underbrace{b_{48} b_{49} \cdots b_{63}}_{B_3}$$

Et calculons $31^3 \times B_3 + 31^2 \times B_2 + 31 \times B_1 + B_0$.

```
public int hashCode() {
  int h0 = (int)config & 0xffff ;
  int h1 = 31*h0 + ((int)(config >>> 16) & 0xffff) ;
  int h2 = 31*h1 + ((int)(config >>> 32) & 0xffff) ;
  int h3 = 31*h2 + ((int)(config >>> 48) & 0xffff) ;
  return h3 ;
}
```

Avec “`config >>>` $16 \cdot k$ `& 0xffff`” calcule $(C \,\mathsf{div}\, 2^{16\cdot k}) \,\mathsf{mod}\, 2^{16}$.

Et finalement

```
%java Soli
.......   .......   .......   .......   .......   .......   .......   .......
.......   .......   .......   .......   .......   .......   .......   .......
.......   .......   .......   .......   .......   ......X   ......X   ......X
......X   ....XX.   .....X.   .....X.   .....X.   .....XX   .....XX   ....XXX
.......   .......   ....X..   .....XX   ...XX.X   ...XX..   ...X.XX   ...XXXX
.......   .......   ....X..   ....X..   ....X..   ....X..   ....X..   .......
.......   .......   .......   .......   .......   .......   .......   .......

etc.

....X..   ...XX..   ...XX..   ...XX..   ...XX..   ..XXX..   ..XXX..   ..XXX..
..X.X..   ..XXX..   ..X.X..   ..X.X..   ....X..   ..X.X..   ..X.X..   ..XXX..
X..XXXX   X...XXX   X..XXXX   X..XXXX   X.XXXXX   X..XXXX   XXX.XXX   XXXXXXX
..X.XXX   ..X.XXX   ..XXXXX   XX.XXXX   XXXXXXX   XXXXXXX   XXXXXXX   XXX.XXX
XXXXXXX   XXXXXXX   XXXXXXX   XXXXXXX   XXXXXXX   XXXXXXX   XXXXXXX   XXXXXXX
..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..
..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..   ..XXX..
```

Quelques détails

- La table est utile.
  - À la fin la table contient 619781 configurations, et elle a servi 1890164 fois (3 fois sur 4).
  - Un autre essai (choix différent de l'ordre des successeurs) donne 7621719 et 32669905 soit 8 utilisations de la mémoire pour 10 appels à `canWin`.
- En un sens nous n'avons pas de chance.

  Un choix miraculeux du mouvement à chaque étape mène à la solution en explorant 32 configurations seulement (et sans nécessité de les mémoriser).
- La technique de mémorisation a ses limites.

  Si on veut *toutes* les solutions, ça ne fonctionne plus : la table épuise la mémoire disponible.