École Polytechnique

CSC_41011

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

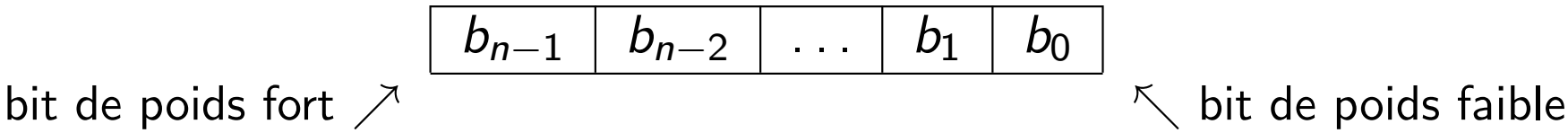
tables de hachage, mémoïsation

aujourd'hui

1. rappel : arithmétique des ordinateurs
2. tables de hachage
3. les tables de hachage de Java
4. application : mémoïsation

un entier est représenté en **base 2**, sur n chiffres appelés **bits**

conventionnellement numérotés de droite à gauche



typiquement, n vaut 8, 16, 32, ou 64

$$\begin{aligned} \text{bits} &= b_{n-1} b_{n-2} \dots b_1 b_0 \\ \text{valeur} &= \sum_{i=0}^{n-1} b_i 2^i \end{aligned}$$

bits	valeur
000...000	0
000...001	1
000...010	2
⋮	⋮
111...110	$2^n - 2$
111...111	$2^n - 1$

type	n	valeurs
`char`	16	$0 .. 2^{16} - 1$

$$\texttt{'P'} = 80 = 0000000001010000$$

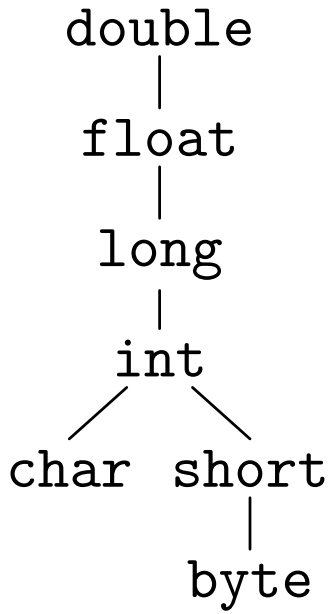
le bit de poids fort b_{n-1} est le **bit de signe**

$$\begin{aligned} \text{bits} &= b_{n-1}\, b_{n-2} \ldots b_1\, b_0 \\ \text{valeur} &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \end{aligned}$$

bits	valeur
100...000	-2^{n-1}
100...001	$-2^{n-1}+1$
⋮	⋮
111...110	-2
111...111	-1
000...000	0
000...001	1
000...010	2
⋮	⋮
011...110	$2^{n-1}-2$
011...111	$2^{n-1}-1$

type	n	valeurs
`byte`	8	$-2^7 \,..\, 2^7 - 1$
`short`	16	$-2^{15} \,..\, 2^{15} - 1$
`int`	32	$-2^{31} \,..\, 2^{31} - 1$
`long`	64	$-2^{63} \,..\, 2^{63} - 1$

conversions automatiques



une valeur de type `char` tient dans un `int`

```
int n = s.charAt(i);
```

mais une valeur de type `int` ne tient pas dans un `char`

```
char c = a.length;
         ^^^^^^^^

Type mismatch: cannot convert from int to char
```

on peut néanmoins forcer avec un *cast* (transtypage)

```
char c = (char)a.length;
```

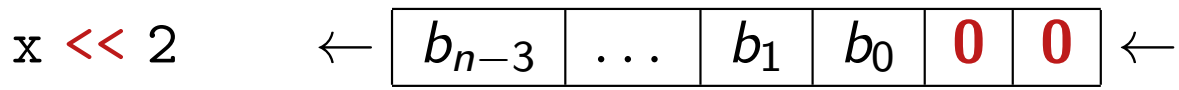
Java fournit des opérations pour manipuler la représentation binaire

- opérations logiques : `~` `&` `|` `^`
- opérations de décalage : `<<` `>>` `>>>`

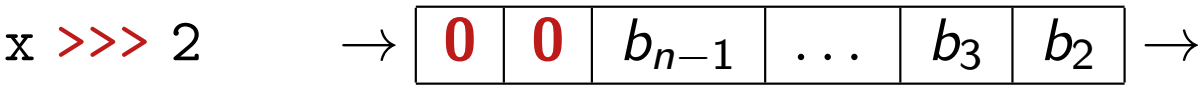
opération		exemple
~ négation	x	00101001
	~x	11010110
& ET	x	00101001
	y	01101100
	x & y	00101000
| OU	x	00101001
	y	01101100
	x | y	01101101
^ OU exclusif	x	00101001
	y	01101100
	x ^ y	01000101

attention ! pas la puissance →

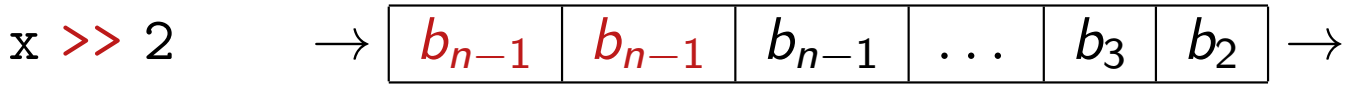
- décalage logique à gauche (insère des 0 de poids faible)



- décalage logique à droite (insère des 0 de poids fort)



- décalage arithmétique à droite (réplique le bit de signe)



débordement de capacité

un calcul arithmétique peut provoquer un **débordement de capacité** (qui n'est pas signalé)

```
System.out.println(100000 * 100000);
```

```
1410065408
```

le résultat peut même être du mauvais signe

```
System.out.println(200000 * 100000);
```

```
-1474836480
```

tables de hachage

produire du **texte aléatoire**, pas trop mauvais, avec l'idée suivante

analyse

- choisir un texte (assez grand)
- considérer tous les triplets de mots consécutifs (A, B, C)

synthèse

- choisir deux mots A et B
- choisir C au hasard parmi les triplets (A, B, C)
- recommencer avec B et C

ici on prend le texte intégral de deux œuvres de Jules Verne

- *Le tour du monde en quatre-vingts jours*
- *Voyage au centre de la terre*

(librement accessibles sur le projet Gutenberg)

Phileas Fogg avait accompli ce tour du monde
en quatre-vingts jours !

(Phileas, Fogg)	$\rightarrow$ avait
(Fogg, avait)	$\rightarrow$ accompli
(avait, accompli)	$\rightarrow$ ce
(accompli, ce)	$\rightarrow$ tour
(ce, tour)	$\rightarrow$ du
(tour, du)	$\rightarrow$ monde
(du, monde)	$\rightarrow$ en
etc.	

199 occurrences des deux mots consécutifs `Phileas Fogg`

(`Phileas`, `Fogg`)	$\rightarrow$ `avait`	(17 fois)
	$\rightarrow$ `était`	(19 fois)
	$\rightarrow$ `distribua`	(1 fois)
	$\rightarrow$ `au`	(2 fois)
	etc.	

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

tirage aléatoire

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

```
Phileas Fogg et ses compagnons s'aventurent
à la surface de la bête
```

il nous faut un **dictionnaire**
associant à des paires de mots (A, B) des multiensembles de mots C

$$\begin{aligned}
(\texttt{Phileas}, \texttt{Fogg}) &\mapsto \{\!\{\texttt{avait}, \texttt{et}, \texttt{avait}, \texttt{salua}, \ldots\}\!\} \\
(\texttt{Fogg}, \texttt{et}) &\mapsto \{\!\{\texttt{le}, \texttt{ses}, \texttt{Sir}, \ldots\}\!\} \\
(\texttt{et}, \texttt{ses}) &\mapsto \{\!\{\texttt{chaussettes}, \texttt{deux}, \texttt{compagnons}, \ldots\}\!\} \\
\ldots &\mapsto \ldots
\end{aligned}$$

un dictionnaire associe des **valeurs** de type `V` à des **clés** de type `K`

```
class Dict<K, V> {
       Dict()
  void put(K key, V value)  // ajoute une nouvelle entrée
     V get(K key)           // renvoie la valeur associée
       ...                  // remove, size, clear, etc.
}
```

si les clés étaient des entiers dans $0..M-1$ un tableau suffirait

on va se ramener à cette situation avec une **fonction**

$$f : \mathtt{K} \to 0..M-1$$

c'est l'idée à la base des tables de hachage

on commence par définir une **fonction de hachage** $h : \mathtt{K} \to \mathbb{Z}$

puis on pose

$$f(k) = h(k) \bmod M$$

on range alors la clé k à l'indice $f(k)$ dans un tableau de taille M

$$
\begin{aligned}
\texttt{K} &= \texttt{String} \\
h(k) &= k\texttt{.length()} \\
M &= 7
\end{aligned}
$$

k	$h(k)$	$h(k) \mod 7$
$k_1 =$ "We like"	7	0
$k_2 =$ "in"	2	2
$k_3 =$ "Java."	5	5
$k_4 =$ "the codes"	9	2

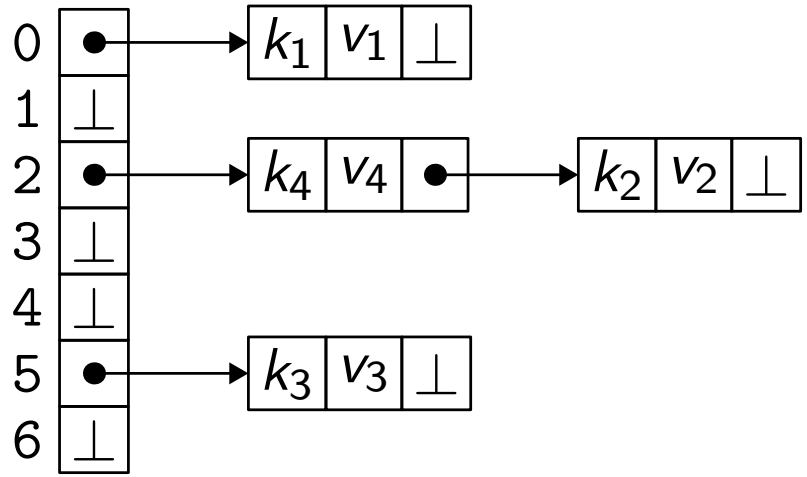
certaines clés peuvent donner le même indice

k	$h(k)$	$h(k) \mod 7$
$k_1 =$ "We like"	7	0
$k_2 =$ "in"	2	2
$k_3 =$ "Java."	5	5
$k_4 =$ "the codes"	9	2

on parle de **collision**

chaque case du tableau contient plusieurs valeurs, dans une liste

une telle liste s'appelle un **seau** (en anglais *bucket*)



- les tee-shirts dans le tiroir 0
- les pulls dans le tiroir 1
- les pantalons dans le tiroir 2

mise en œuvre

nos clés sont ici des paires de chaînes de caractères

```
class Pair {
  final String fst, snd;

  Pair(String fst, String snd) {
    this.fst = fst;
    this.snd = snd;
  }
```

on choisit de faire la somme

```
class Pair {
  ...
  int hash() {
    return hashString(this.fst) + hashString(this.snd);
  }
```

reste à écrire `hashString`

```
  int hashString(String s) {
    return ???
  }
```

pour une chaîne on choisit

$$s_0 31^{n-1} + s_1 31^{n-2} + \cdots + s_{n-1}$$

```
int hashString(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
  return h;
}
```

exemple :

$$\begin{aligned} & \texttt{hashString("Phileas")} \\ = \; & \texttt{'P'}31^6 + \texttt{'h'}31^5 + \texttt{'i'}31^4 + \texttt{'l'}31^3 + \texttt{'e'}31^2 + \texttt{'a'}31 + \texttt{'s'} \\ = \; & 80 \times 31^6 + 104 \times 31^5 + 105 \times 31^4 + 108 \times 31^3 + \ldots \\ = \; & 1063569468 \text{ (en fait 74078013500 mais débordement !)} \end{aligned}$$

rien d'autre qu'une liste chaînée de couples (clé, valeur)

```
class Bucket {
            Pair key;   //    clé = paire de mots (w1, w2)
  Vector<String> value; // valeur = liste de mots w3
          Bucket next;

  // et son constructeur
}
```

une table de hachage est un tableau de seaux

```
class HashTable {
  private Bucket[] buckets;
```

le nombre de seaux est choisi arbitrairement

```
  private final static int M = 5003;

  HashTable() {
    this.buckets = new Bucket[M];
  }
```

(on peut ajouter un second constructeur prenant `M` en argument)

ajout dans la table

```
void put(Pair key, Vector<String> value) {
  int i = key.hash() % M;
  this.buckets[i] = new Bucket(key, value, this.buckets[i]);
}
```

a la complexité de `key.hash()`

en pratique $O(1)$

note : on ne cherche pas ici à savoir si `key` est déjà dans la table
(ce sera fait ailleurs, plus loin)

`key.hash()` peut être négatif, en cas de débordement arithmétique

`key.hash() % M` est alors également négatif

et

```
int i = Math.abs(key.hash()) % M;
```

reste incorrect car `Math.abs(` -2^{31}`)` $= -2^{31}$

- rectifier le modulo

```
int i = key.hash() % M;
if (i < 0) i += M;
```

- masquer le bit de signe

```
int i = (key.hash() & 0x7fffffff) % M;
```

(car `0x7fffffff` $= 2^{31} - 1 = 01111\ldots111_2$)

- prendre `M` $= 2^k$ puis

```
int i = key.hash() & (M - 1);
```

car alors `M` $- 1 = 2^k - 1 = 00\ldots0011\ldots11_2$
(évite l'opération % qui est coûteuse)

pour chercher dans la table, il faut se donner une **égalité** sur les clés (une méthode `equals`)

cette égalité doit être **cohérente** avec la fonction de hachage

$$\forall x\, y,\ x\texttt{.equals}(y) \Rightarrow x\texttt{.hash()} = y\texttt{.hash()}$$

égalité sur les clés

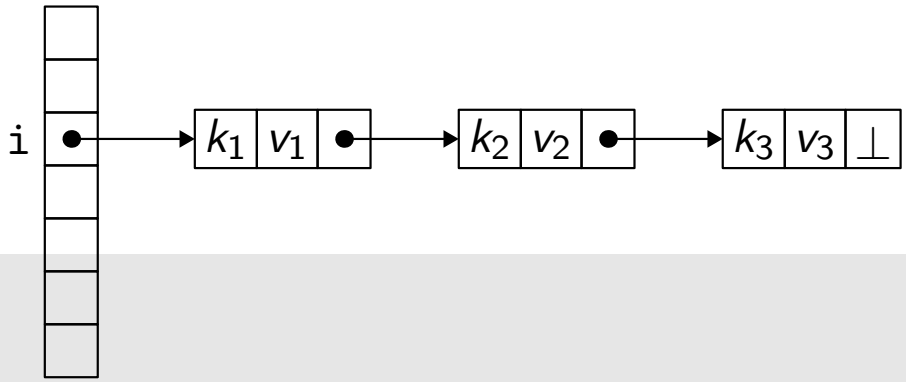
```
class Pair {
  ...
  boolean equals(Pair p) {
    return this.fst.equals(p.fst) && this.snd.equals(p.snd);
  }
}
```

(rappel : on compare des chaînes avec `equals`, pas avec `==`)

rechercher dans un seau

```
class Bucket {
  Pair key;
  Vector<String> value;
  Bucket next;
  ...
  static Vector<String> get(Bucket b, Pair k) {
    for (; b != null; b = b.next)
      if (b.key.equals(k))
        return b.value;
    return null; // pas dans le seau
  }
}
```

rechercher dans la table




```
class HashTable {
  ...
  Vector<String> get(Pair key) {
    int i = (key.hash() & 0x7fffffff) % M; // comme dans put
    return Bucket.get(this.buckets[i], key);
  }
}
```

complexité : (au pire) la longueur du seau

on aurait pu réaliser également chaque seau avec

- un tableau redimensionnable pour les clés
- un autre pour les valeurs

retour sur le problème initial

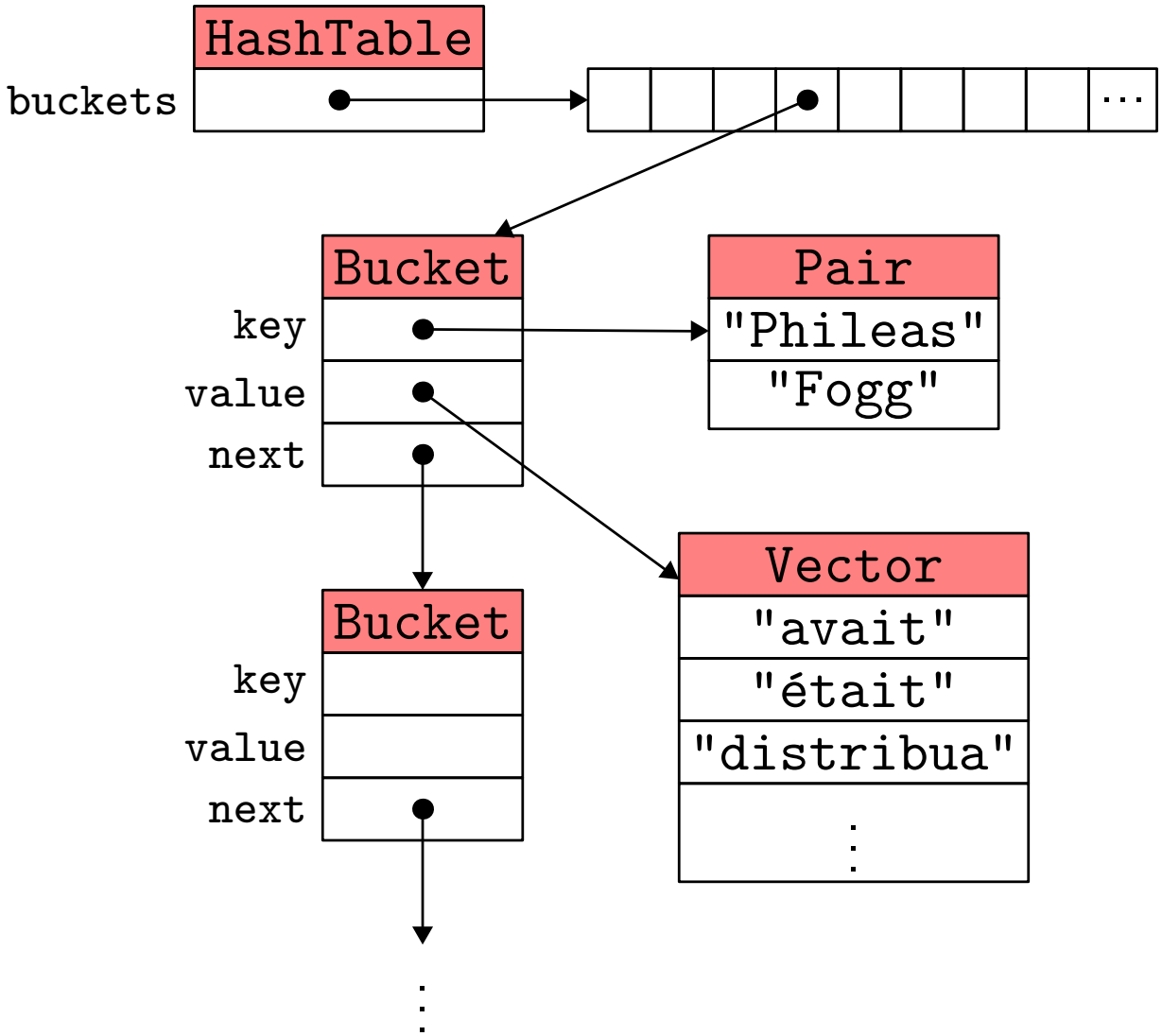
nos chaînes de Markov

```
static HashTable chains = new HashTable();
```

pour chaque triplet de mots consécutifs `w1 w2 w3` du texte

```
Pair p = new Pair(w1, w2);
Vector<String> l = chains.get(p);
if (l == null) {
  l = new Vector<String>();
  chains.put(p, l);
}
l.add(w3);
```

(le code complet est donné sur la page du cours)


HashTable
buckets
Bucket
key
value
next
Pair
"Phileas"
"Fogg"
Vector
"avait"
"était"
"distribua"
Bucket
key
value
next

pour générer le texte aléatoirement,
il faut choisir un élément au hasard parmi les mots qui suivent (w_1, w_2)

c'est-à-dire choisir au hasard dans `chains.get((`w_1, w_2`))`

```
// ici on suppose que v contient au moins un élément
static String randomElement(Vector<String> v) {
  int i = (int) (Math.random() * v.size());
  return v.get(i);
}
```

on a maintenant tous les éléments pour terminer notre programme

1. pour démarrer, on choisit une paire (w_1, w_2) au hasard dans la table
   - ou mieux encore deux mots au début d'une phrase

2. puis on répète
   2.1 choisir w_3 au hasard dans `chains.get((`w_1, w_2`))`
   2.2 décaler $w_1, w_2 \leftarrow w_2, w_3$

(le code complet est donné sur la page du cours)

La veille, le soleil s'était couché dans une tête très grosse et assez naïve...

Parbleu ! – Le tour du monde entier, si son cratère aboutit au centre du globe !

Puis je m'endormis sur un espace de plusieurs centaines d'atmosphères

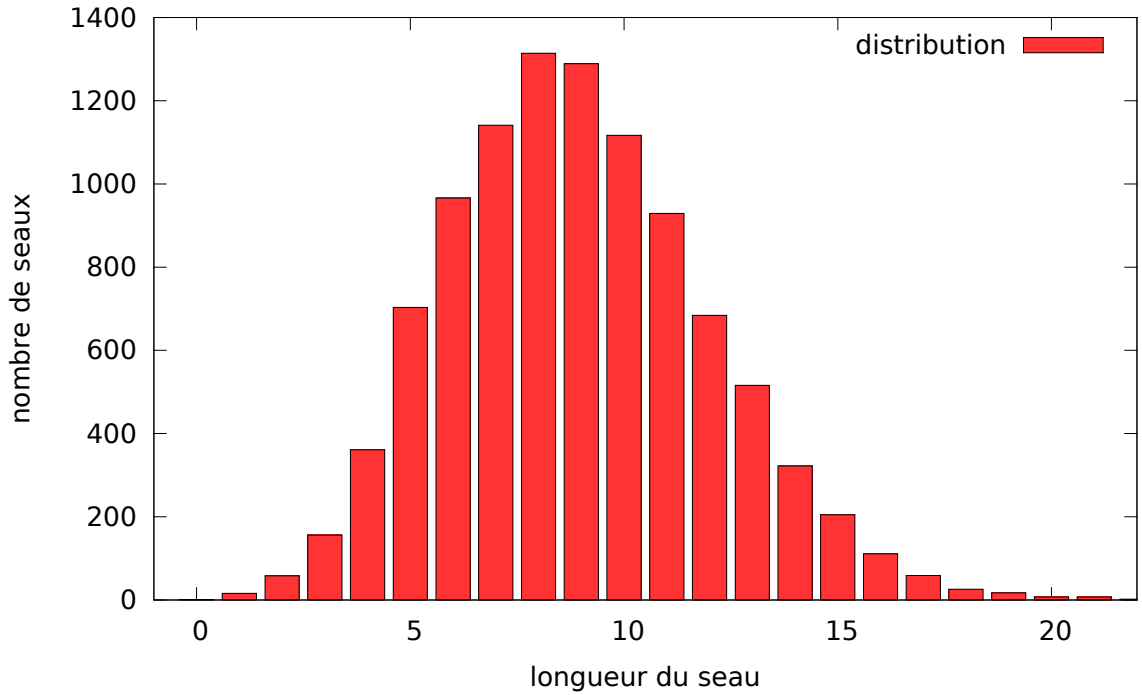
(mais aussi beaucoup de phrases incorrectes)

efficacité

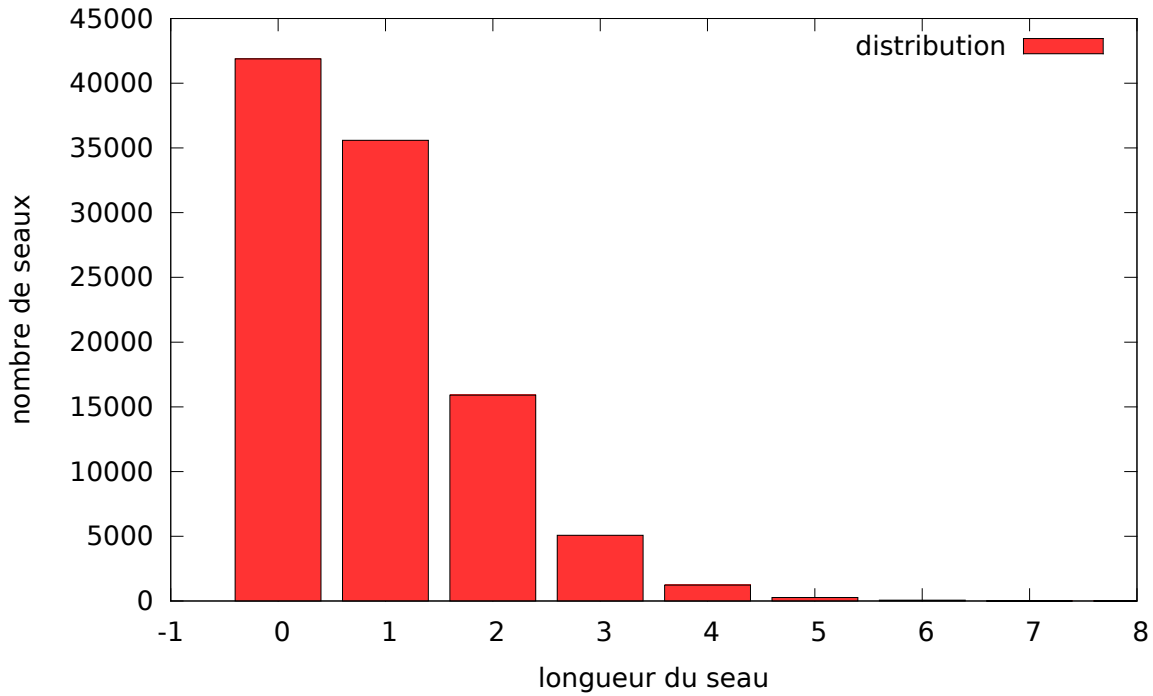
$N = 89\,244$ entrées dans la table (*i.e.* paires différentes)

observons les longueurs des seaux,
pour différentes valeurs de M (et donc de la **charge** N/M)

pour $M = 10\,007$ (charge $N/M = 8,9$)



pour $M = 100\,003$ (charge $N/M = 0,89$)



on choisit M de l'ordre du nombre d'entrées

cela suppose qu'on le connaît ; **et sinon ?**

utiliser un **tableau redimensionnable**

une stratégie possible :

quand la charge atteint 1, doubler la valeur de M

de manière générale, quand la charge atteint une certaine valeur
(la bibliothèque Java utilise 0,75 par défaut)

cela veut dire allouer un tableau deux fois plus grand
et y ré-insérer toutes les entrées

cela coûte $O(N)$

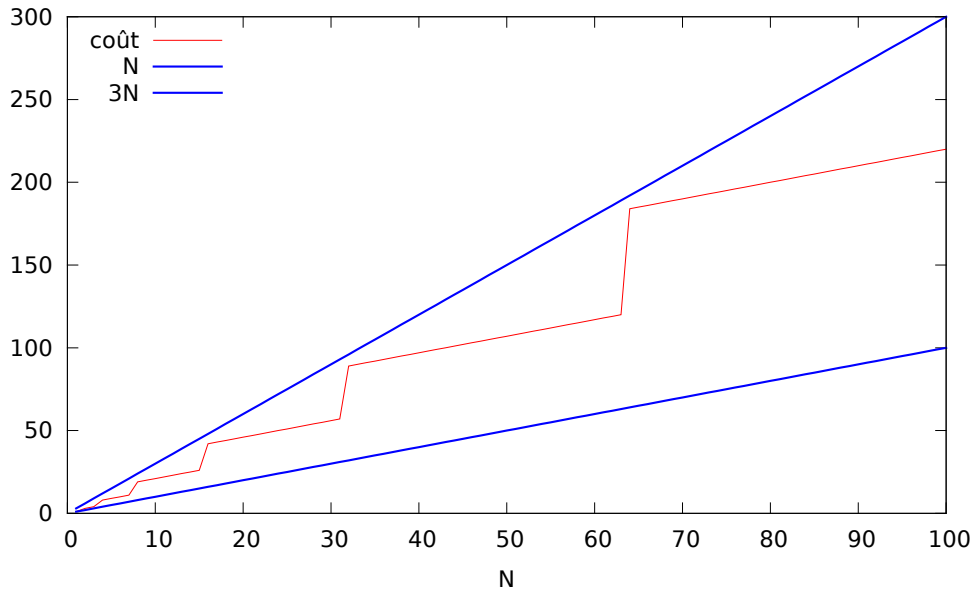
`put` n'a plus une complexité $O(1)$

certains appels à `put` sont $O(1)$ (pas de redimensionnement)
d'autres $O(N)$ où N est le nombre d'entrées (redimensionnement)

mais la complexité **amortie** de `put` reste $O(1)$

c'est le même calcul
que la semaine dernière :

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$



les tables de hachage de Java

deux classes génériques

- `java.util.HashMap<K, V>`
  dictionnaire associant des valeurs de type `V` à des clés de type `K`

- `java.util.HashSet<E>`
  ensemble d'éléments de type `E`

structure de dictionnaire

chaque clé est associée à au plus une valeur

```
h.put(k, v);  // ajoute une entrée
h.get(k);     // recherche (renvoie null si pas d'entrée)
h.remove(k);  // supprime une entrée

h.isEmpty()   // le dictionnaire est-il vide ?
h.size()      // le nombre d'entrées
...
```

on peut parcourir toutes les entrées avec

```
for (Entry<K, V> e : h.entrySet()) ...
```

structure d'ensemble

```
s.add(e);        // ajoute un élément
s.contains(e);   // teste l'appartenance
s.remove(e);     // supprime un élément

s.isEmpty()      // l'ensemble est-il vide ?
s.size()         // le cardinal
...
```

on peut parcourir tous les éléments avec

```
for (E x : s) ...
```

que ce soit pour `HashMap` ou `HashSet`,
il faut munir les types `K` et `E` d'une fonction de hachage et d'une égalité

tout objet Java possède deux méthodes **héritées** de la classe `Object`

```
      int hashCode()
boolean equals(Object o)
```

méthodes `hashCode` et `equals`

dans la classe `Object`

- `hashCode` renvoie un entier arbitraire
- `equals` coïncide avec `==`

dans une autre classe, il faudra **redéfinir** ces deux méthodes (si besoin)

il faut redéfinir `hashCode` et `equals` de manière **cohérente**

$$\forall x\, y,\ x\texttt{.equals}(y) \Rightarrow x\texttt{.hashCode()} = y\texttt{.hashCode()}$$

dans la classe `String`

- `hashCode` renvoie un entier calculé **en fonction des caractères**

$$\texttt{s.hashCode()} = \texttt{s}_0 31^{n-1} + \texttt{s}_1 31^{n-2} + \cdots + \texttt{s}_{n-1}$$

- `equals` est **l'égalité structurelle** des chaînes

$$\texttt{s.equals(t)} \textit{ ssi } \texttt{s.length()} = \texttt{t.length()} \textit{ et } \forall i,\ \texttt{s}_i = \texttt{t}_i$$

(également déjà fait dans les classes `Integer`, `Char`, etc.)

```
class Pair {
  final String fst, snd;
```

pour la fonction de hachage, c'est facile :

```
  @Override // signifie au compilateur une redéfinition
  public int hashCode() {
    return this.fst.hashCode() + this.snd.hashCode();
  }
```

pour l'égalité, c'est a priori facile : on compare les deux champs `fst` et les deux champs `snd` avec l'égalité des chaînes

il y a une difficulté technique, cependant

```
public boolean equals(Object o) {
  ...
}
```

l'argument est **de type** `Object`, et non pas `Pair`

il faut donc écrire

```
public boolean equals(Object o) {
  Pair p = (Pair)o;
  return this.fst.equals(p.fst) && this.snd.equals(p.snd);
}
```

le cast `(Pair)o` donne lieu à un test dynamique

il n'échouera pas si cette méthode
est appelée depuis `HashMap` ou `HashSet`

si on écrit plutôt

```
public boolean equals(Pair p) {
  return this.fst.equals(p.fst) && this.snd.equals(p.snd);
}
```

alors la méthode `equals` est **surchargée** au lieu d'être **redéfinie**

et c'est toujours la méthode `equals` de la classe `Object` qui sera appelée par `HashMap`

(surcharge/redéfinition : relire éventuellement le chapitre 1 du poly)

mais si on ajoute `@Override`

```
@Override
public boolean equals(Pair p) {
  return this.fst.equals(p.fst) && this.snd.equals(p.snd);
}
```

le compilateur nous signale notre erreur

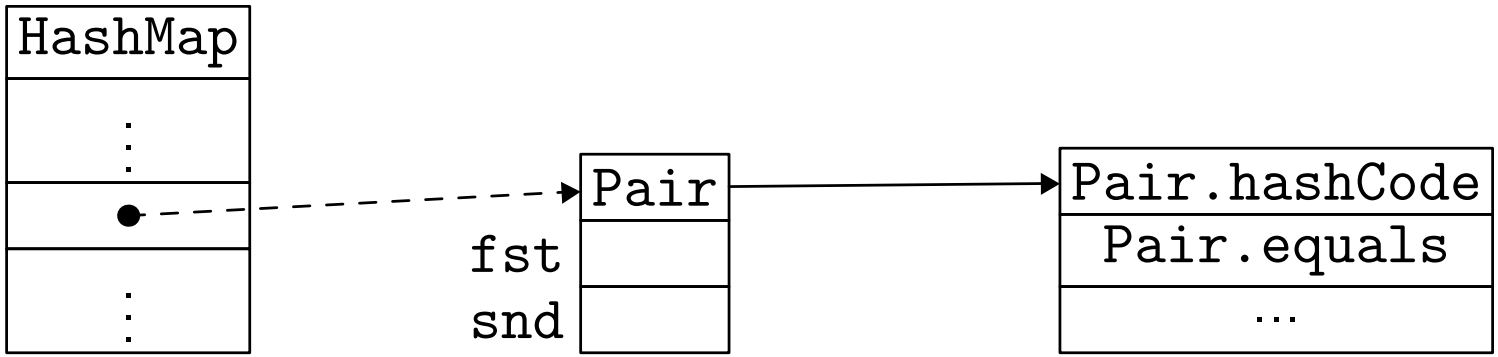
```
The method equals(Pair) of type Pair must override
or implement a supertype method
```

car il n'y a pas de telle méthode à redéfinir

le code de `HashMap` ou de `HashSet` ne connaît pas la classe `Pair`

```
class HashMap<K, V> {
  ...
  void put(K k, V v) { ... k.hashCode() ... }
}
```

pourtant il utilise bien les méthodes `hashCode` et `equals` de la classe `Pair` grâce à l'**appel dynamique** de méthode



une table de hachage offre une structure très efficace pour

- ajouter
- chercher
- supprimer

en supposant une fonction de hachage répartissant bien les valeurs

en revanche, **pas d'ordre** sur les éléments

structure d'ensemble, contenant N éléments

	`add`	`contains`	`get(`i`)`	
tableau	$O(1)$ amorti	$O(N)$	$O(1)$	
tableau trié	$O(N)$	$O(\log N)$	$O(1)$	(voir poly)
liste	$O(1)$	$O(N)$	$O(i)$	
table de hachage	$O(1)$ **amorti**	$O(1)$	—	

si vous avez besoin d'un ensemble ou d'un dictionnaire,
utilisez une table de hachage

si vous avez programmé en Python, vous avez sûrement déjà utilisé des tables de hachage, peut-être sans le savoir

```
>>> d = {}
```

```
HashMap<String, Integer> d =
  new HashMap<>();
```

```
>>> d["foo"] = 42
>>> print(d["foo"])
42
```

```
d.put("foo", 42);
System.out.prinln(d.get("foo"));
```

```
>>> d["bar"]
KeyError: 'bar'
```

```
d.get("bar")
// renvoie null
```

il ne faut jamais modifier une clé après l'avoir ajoutée à une table de hachage (sans quoi elle n'est plus dans le bon seau)

bonne pratique : n'utiliser que des clés **immuables**, comme `String` ou encore notre classe `Pair`

```
class Pair {
  final String fst, snd;
  ...
```

application : mémoïsation

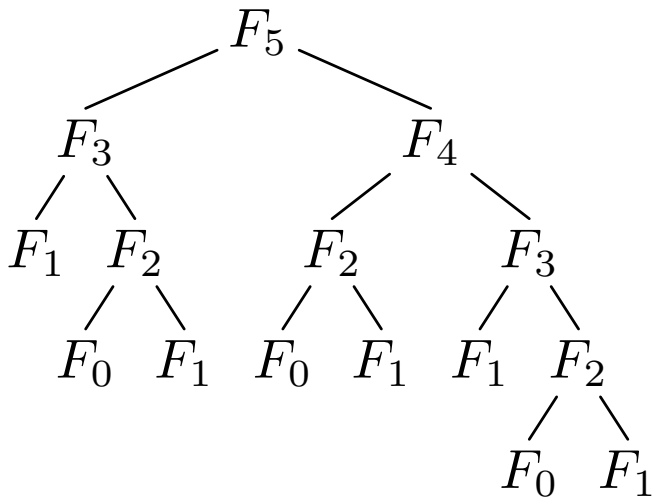
ne pas faire deux fois le même calcul

la suite de Fibonacci

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \text{ pour } n \geq 2 \end{cases}$$

```
static long fib(int n) {
  if (n <= 1) return n;
  return fib(n - 2) + fib(n - 1);
}
```

on calcule plusieurs fois la même chose



on peut montrer que le calcul de F_n est en $O(\phi^n)$
(cf exercice 2 du poly)

idée : stocker les résultats déjà calculés dans une table

```
static HashMap<Integer, Long> memo =
   new HashMap<Integer, Long>();
```

n F_n

on commence par regarder si le calcul a déjà été fait

```
  static long fibMemo(int n) {
    Long l = memo.get(n);
    if (l != null) return l; // si oui, on le renvoie
```

sinon, on fait le calcul

```
    if (n <= 1)
      l = (long)n;
    else
      l = fibMemo(n - 2) + fibMemo(n - 1);
```

puis on le stocke et on le renvoie

```
    memo.put(n, l);
    return l;
  }
```

le calcul de F_n est maintenant en $O(n)$
(`fibMemo(`k`)` est appelé au plus deux fois pour chaque k)

on calcule ainsi $F_{80} = 23416728348467685$ instantanément

bien entendu, sur cet exemple un simple tableau aurait suffit

et on aurait même pu le remplir dans l'ordre

F_0	F_1	F_2	$\ldots$	F_i	$\ldots$	F_{80}

dans ce cas, on parle de **programmation dynamique**
(voir le poly chapitre 11)

- les arguments ne sont pas forcément des entiers
- même dans ce cas, on ne calcule pas forcément $f(0)$, $f(1)$, ..., $f(n)$

la mémoïsation est alors une approche avantageuse

```
static HashMap<Argument, Resultat> memo = new HashMap<>();
```

```
static Resultat f(Argument x) {
  Resultat r = memo.get(x);
  if (r != null) return r;
  // calcul de r = f(x)
  ...
  memo.put(x, r);
  return r;
}
```

dans une matrice d'entiers $N \times N$

sélectionner N éléments, un sur chaque ligne et sur chaque colonne, de somme maximale

ici $N = 15$

un exemple plus complexe

7	53	183	439	863	497	383	563	79	973	287	63	343	169	583
627	343	773	959	943	767	473	103	699	303	957	703	583	639	913
447	283	463	29	23	487	463	993	119	883	327	493	423	159	743
217	623	3	399	853	407	103	983	89	463	290	516	212	462	350
960	376	682	962	300	780	486	502	912	800	250	346	172	812	350
870	456	192	162	593	473	915	45	989	873	823	965	425	329	803
973	965	905	919	133	673	665	235	509	613	673	815	165	992	326
322	148	972	962	286	255	941	541	265	323	925	281	601	95	973
445	721	11	525	473	65	511	164	138	672	18	428	154	448	848
414	456	310	312	798	104	566	520	302	248	694	976	430	392	198
184	829	373	181	631	101	969	613	840	740	778	458	284	760	390
821	461	843	513	17	901	711	993	293	157	274	94	192	156	574
34	124	4	878	450	476	712	914	838	669	875	299	823	329	699
815	559	813	459	522	788	168	586	966	232	308	833	251	631	107
813	883	451	509	615	77	281	613	459	205	380	274	302	35	805

un exemple plus complexe

7	53	183	439	863	497	383	563	79	973	287	63	343	169	583
627	343	773	959	943	767	473	103	699	303	957	703	583	639	913
447	283	463	29	23	487	463	993	119	883	327	493	423	159	743
217	623	3	399	853	407	103	983	89	463	290	516	212	462	350
960	376	682	962	300	780	486	502	912	800	250	346	172	812	350
870	456	192	162	593	473	915	45	989	873	823	965	425	329	803
973	965	905	919	133	673	665	235	509	613	673	815	165	992	326
322	148	972	962	286	255	941	541	265	323	925	281	601	95	973
445	721	11	525	473	65	511	164	138	672	18	428	154	448	848
414	456	310	312	798	104	566	520	302	248	694	976	430	392	198
184	829	373	181	631	101	969	613	840	740	778	458	284	760	390
821	461	843	513	17	901	711	993	293	157	274	94	192	156	574
34	124	4	878	450	476	712	914	838	669	875	299	823	329	699
815	559	813	459	522	788	168	586	966	232	308	833	251	631	107
813	883	451	509	615	77	281	613	459	205	380	274	302	35	805

$$563 + 699 + \cdots + 522 + 451 = 7805$$

on généralise le problème : quel maximum $f(i, C)$ pour

- pour des lignes $\geq i$
- pour des colonnes dans l'ensemble C de cardinal $N - i$

?

$$\begin{aligned} f(N, \emptyset) &= 0 \\ f(i, C) &= \max_{j \in C} \mathtt{m}[i][j] + f(i+1,\ C \backslash \{j\}) \end{aligned}$$

la solution est alors

$$f(0,\ \{0, \ldots, N-1\})$$

d'où le code

```
final static int m[][] = { { 7, 53, ... }, ... };
final static int n = m.length;
```

$$f(i, C) \quad = \quad \max_{j \in C} \; \mathtt{m}[i][j] + f(i+1, C \backslash \{j\})$$

```
static int f(int i, Set c) {
  if (i == n) return 0;
  int s = 0;
  for (int j = 0; j < n ; j++)
    if (c.contains(j))
      s = Math.max(s, m[i][j] + f(i + 1, c.remove(j)));
  return s;
}
```

(pour une représentation astucieuse et efficace de c,
voir le poly page 146)

le programme ne termine pas

il y a beaucoup de calculs : $15! \approx 1,3 \times 10^{12}$

et pourtant, il n'y a que

- 15 valeurs pour i
- 2^{15} valeurs pour C

c'est-à-dire seulement

$$15 \times 2^{15} = 15 \times 32\,768 = \mathbf{491\,520}$$

calculs $f(i, C)$ différents au plus

c'est **beaucoup moins** que 15!

on calcule **plusieurs fois la même chose**

en effet

si on choisit $m[0][0]$ puis $m[1][1]$

7	53	183	439	...
627	343	773	959	...
⋮				

ou bien $m[0][1]$ puis $m[1][0]$

7	53	183	439	...
627	343	773	959	...
⋮				

on poursuit dans les deux cas avec $f(2, \{2, \ldots, 14\})$

```
class IC { ... } // une paire (int, Set)
```

```
final static HashMap<IC, Integer> memo =
        new HashMap<IC, Integer>();
```

(i, C) $f(i, C)$

lorsque le résultat est déjà dans la table, c'est $O(1)$

la complexité est maintenant $O(N^2 \times 2^N)$

plus précisément, le corps de `f` n'est exécuté qu'au plus 491 520 fois

c'est maintenant instantané (74ms) ; on trouve 13 938

7	53	183	439	863	497	383	563	79	973	287	63	343	169	583
627	343	773	959	943	767	473	103	699	303	957	703	583	639	913
447	283	463	29	23	487	463	993	119	883	327	493	423	159	743
217	623	3	399	853	407	103	983	89	463	290	516	212	462	350
960	376	682	962	300	780	486	502	912	800	250	346	172	812	350
870	456	192	162	593	473	915	45	989	873	823	965	425	329	803
973	965	905	919	133	673	665	235	509	613	673	815	165	992	326
322	148	972	962	286	255	941	541	265	323	925	281	601	95	973
445	721	11	525	473	65	511	164	138	672	18	428	154	448	848
414	456	310	312	798	104	566	520	302	248	694	976	430	392	198
184	829	373	181	631	101	969	613	840	740	778	458	284	760	390
821	461	843	513	17	901	711	993	293	157	274	94	192	156	574
34	124	4	878	450	476	712	914	838	669	875	299	823	329	699
815	559	813	459	522	788	168	586	966	232	308	833	251	631	107
813	883	451	509	615	77	281	613	459	205	380	274	302	35	805

(si on veut les 15 nombres, il faut modifier un peu le programme)

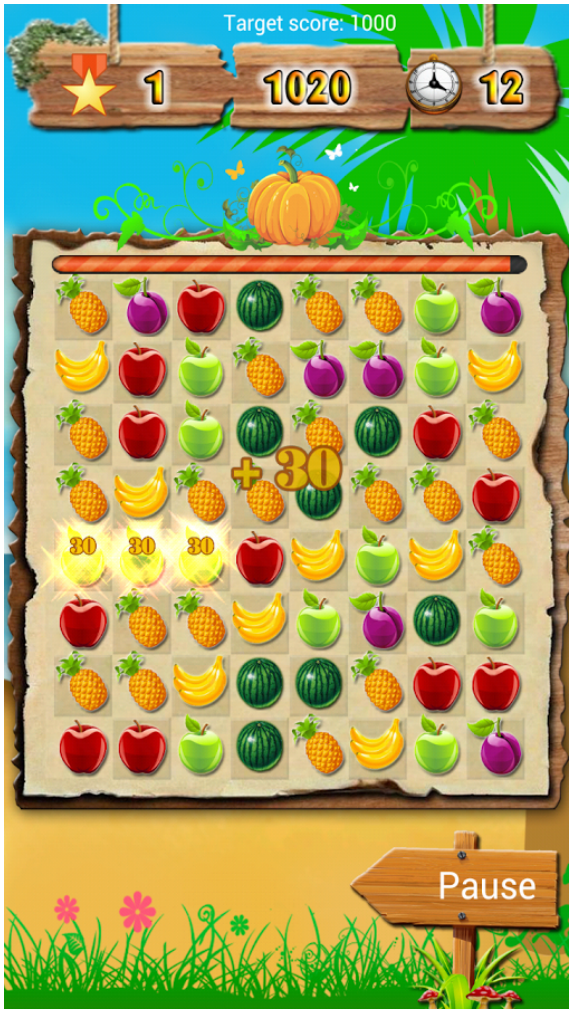
le TD de cette semaine

petit problème de combinatoire :

combien de grilles ne contenant jamais trois fruits identiques consécutifs ?

pour une grille 10×10 et deux fruits différents
il y a déjà des millions de milliards de combinaisons

saurez-vous en calculer le nombre exact ?



- **lire le poly**, chapitres
  - 1.2.1 Arithmétique des ordinateurs
  - 5 Tables de hachage
  - 11 Programmation dynamique et mémoïsation

  il y a des **exercices** dans le poly
  suggestions : ex 30 p 73, ex 80 p 139

- **bloc 3** : classes disjointes