École Polytechnique

# INF411

# Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

Édition 2023

# Avant-propos

Ce polycopié est utilisé pour le cours INF411 intitulé *Les bases de la programmation et de l'algorithmique*. Ce cours fait suite au cours INF361 intitulé *Introduction à l'informatique* et précède le cours INF421 intitulé *Design and Analysis of Algorithms*.

Ce polycopié reprend, dans le chapitre 2, quelques éléments d'un précédent polycopié écrit, en plusieurs itérations, par Jean Berstel, Jean-Éric Pin, Philippe Baptiste, Luc Maranget et Olivier Bournez. Je les remercie sincèrement pour m'avoir autorisé à réutiliser une partie de ce polycopié. D'autres éléments sont repris, et adaptés, d'un ouvrage écrit en collaboration avec mon collègue Sylvain Conchon [4]. Je remercie également Didier Rémy pour son excellent paquet LaTeX `exercise`.

Enfin, je remercie très chaleureusement les différentes personnes qui ont pris le temps de relire tout ou partie de ce polycopié : Marie Albenque, Martin Clochard, Alain Couvreur, Stefania Dumbrava, Léon Gondelman, Mário Pereira, François Pottier, David Savourey.

On peut consulter la version PDF de ce polycopié, ainsi que l'intégralité du code Java, sur le site du cours :

`http://www.enseignement.polytechnique.fr/informatique/INF411/`

L'auteur peut être contacté par courrier électronique à l'adresse suivante :

`jean-christophe.filliatre@polytechnique.edu`

Version 11, juillet 2023. Compilée à partir du commit `9d6a5466`.

# Table des matières

# Première partie

# Préliminaires

# 1 Le langage Java

On rappelle ici certains points importants du langage de programmation Java. Ce chapitre n'est nullement exhaustif et suppose que le lecteur est déjà familier du langage Java, notamment grâce au cours INF361. On pourra aussi lire l'excellent ouvrage *Introduction to Programming in Java* de Sedgewick et Wayne [15].

## 1.1 Programmation orientée objets

Le concept central est celui de *classe*. La déclaration d'une classe introduit un nouveau type. En toute première approximation, une classe peut être vue comme un enregistrement.

```
class Polar {
  double rho;
  double theta;
}
```

Ici `rho` et `theta` sont les deux *champs* de la classe `Polar`, de type `double`. On crée une *instance* particulière d'une classe, appelée un *objet*, avec la construction `new`. Ainsi

```
Polar p = new Polar();
```

déclare une nouvelle variable locale `p`, de type `Polar`, dont la valeur est une nouvelle instance de la classe `Polar`. L'objet est alloué en mémoire. Ses champs reçoivent des valeurs par défaut (en l'occurrence ici le nombre flottant `0.0`). On peut accéder aux champs de `p`, et les modifier, avec la notation usuelle `p`.$x$. Ainsi on peut écrire

```
p.rho = 2;
p.theta = 3.14159265;
double x = p.rho * Math.cos(p.theta);
p.theta = p.theta / 2;
```

Pour allouer de nouveaux objets en initialisant leurs champs avec des valeurs particulières, autres que les valeurs par défaut, on peut introduire un ou plusieurs *constructeurs*. Un constructeur naturel pour la classe `Polar` prend les valeurs des champs `rho` et `theta` en arguments. On l'écrit ainsi (dans la classe `Polar`) :

```
Polar(double r, double t) {
  if (r < 0) throw new Error("Polar: negative length");
  rho = r;
  theta = t;
}
```

Ici, on ne se contente pas d'initialiser les champs. On vérifie également que `r` n'est pas négatif. Dans le cas contraire, on lève une exception. Ce constructeur nous permet d'écrire maintenant

```
Polar p = new Polar(2, 3.14159265);
```

**Attention**

Nous avions pu écrire plus haut `new Polar()` sans avoir défini de constructeur. En effet, toute classe possède un constructeur par défaut, sans argument. Mais une fois qu'un constructeur est ajouté à la classe `Polar`, le constructeur implicite sans argument disparaît. Dans l'exemple ci-dessus, si on tente d'écrire maintenant `new Polar()`, on obtient un message d'erreur du compilateur : `The constructor Polar() is undefined`. Rien ne nous empêche cependant de réintroduire un constructeur sans argument. Une classe peut en effet avoir plusieurs constructeurs, avec des arguments en nombre ou en nature différents. On parle de *surcharge*. La surcharge est expliquée plus loin.

### 1.1.1 Encapsulation

Supposons maintenant que l'on veuille maintenir l'*invariant* suivant pour tous les objets de la classe `Polar` :

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi$$

Pour cela on déclare les champs `rho` et `theta` *privés*, de sorte qu'ils ne sont plus visibles à l'extérieur de la classe `Polar`.

```
class Polar {
    private double rho, theta;
    Polar(double r, double t) { /* garantit l'invariant */ }
}
```

Si on cherche à accéder au champ `rho` depuis une autre classe, en écrivant par exemple `p.rho` pour un certain objet `p` de la classe `Polar`, on obtient un message d'erreur du compilateur :

```
File.java:19: rho has private access in Polar
```

Les objets remplissent donc un premier rôle d'*encapsulation*. La valeur du champ `rho` peut néanmoins être fournie par l'intermédiaire d'une *méthode*, c'est-à-dire d'une fonction fournie par la classe `Polar` et applicable à tout objet de cette classe.

```
class Polar {
    private double rho, theta;
    ...
    double norm() { return rho; }
}
```

Pour un objet `p` de type `Polar`, on appelle la méthode `norm` ainsi :

```
p.norm()
```

Naïvement, on peut voir cet appel de méthode comme un appel `norm(p)` à une *fonction* `norm` qui recevrait l'objet comme premier argument. Dans une méthode, cet argument implicite, qui est l'objet sur lequel on appelle une méthode, est désigné par le mot-clé `this`. Ainsi on peut réécrire la méthode `norm` ci-dessus de la manière suivante :

```
double norm() { return this.rho; }
```

On explicite le fait que `rho` désigne ici un champ de l'objet. En particulier, on évite une confusion possible avec une variable locale ou un paramètre de la méthode. De la même manière, nous aurions pu écrire le constructeur sous la forme

```
Polar(double r, double t) {
  this.rho = r;
  this.theta = t;
}
```

car, dans le constructeur, `this` désigne l'objet qui vient d'être alloué. Du coup, on peut même donner aux paramètres du constructeur les mêmes noms que les champs :

```
Polar(double rho, double theta) {
  this.rho = rho;
  this.theta = theta;
}
```

Il n'y a pas d'ambiguïté, puisque `rho` désigne le paramètre et `this.rho` le champ. On évite ainsi d'avoir à trouver un nom différent pour l'argument — quand on programme, le plus difficile est souvent de trouver des noms judicieux.

**Attention**

En revanche, il serait incorrect d'écrire le constructeur sous la forme

```
Polar(double rho, double theta) {
  rho = rho;
  theta = theta;
}
```

Bien qu'accepté par le compilateur, ces affectations sont sans effet : elles ne font qu'affecter les valeurs des paramètres `rho` et `theta` aux paramètres `rho` et `theta` eux-mêmes.

### 1.1.2 Champs et méthodes statiques

Il est possible de déclarer un champ comme *statique* et il est alors lié à la classe et non aux instances de cette classe ; dit autrement, il s'apparente à une variable globale.

```
class Polar {
  double rho, theta;
  static double two_pi = 6.283185307179586;
```

De même, une *méthode* peut être *statique* et elle s'apparente alors à une fonction traditionnelle.

```
  static double normalize(double x) {
    while (x < 0) x += two_pi;
    while (x >= two_pi) x -= two_pi;
    return x;
  }
```

Ce qui n'est pas statique est appelé *dynamique*. Dans une méthode statique, l'emploi de `this` n'est pas autorisé. En effet, il n'existe pas nécessairement d'objet particulier ayant été utilisé pour appeler cette méthode. Pour la même raison, une méthode statique ne peut pas appeler une méthode dynamique. À l'inverse, en revanche, une méthode dynamique peut parfaitement appeler une méthode statique. Enfin, on note que le point d'entrée d'un programme Java, à savoir sa méthode `main`, est une méthode statique :

```
  public static void main(String[] args) { ... }
```

### 1.1.3 Surcharge

Plusieurs méthodes d'une même classe peuvent porter le même nom, pourvu qu'elles aient des arguments en nombre et/ou en nature différents ; c'est ce que l'on appelle la *surcharge* (en anglais *overloading*). Ainsi on peut écrire dans la classe `Polar` deux méthodes `mult` pour multiplier respectivement par un autre nombre complexe en coordonnées polaires ou par un simple flottant.

```
class Polar {
  ...
  void mult(Polar p) {
    this.rho *= p.rho; this.theta = normalize(this.theta + p.theta);
  }
  void mult(double f) {
    this.rho *= f;
  }
}
```

On peut alors écrire des expressions comme `p.mult(p)` ou encore `p.mult(2.5)`. La surcharge est résolue par le compilateur, au moment du typage. Tout se passe comme si on avait écrit en fait deux méthodes avec des noms différents

```
class Polar {
  ...
  void mult_Polar(Polar p) {
    this.rho *= p.rho; this.theta = normalize(this.theta + p.theta);
  }
  void mult_double(double f) {
    this.rho *= f;
  }
}
```

puis les expressions `p.mult_Polar(p)` et `p.mult_double(2.5)`. Ce n'est donc rien d'autre qu'une facilité fournie par le langage pour ne pas avoir à introduire des noms différents. On peut surcharger autant les méthodes statiques que dynamiques, ainsi que les constructeurs (voir notamment l'encadré page 4).

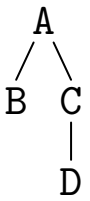
### 1.1.4 Héritage

Le concept central de la programmation orientée objet est celui d'*héritage* : une classe `B` peut être définie comme héritant d'une classe `A`, ce qui se note

```
class B extends A { ... }
```

Les objets de la classe `B` héritent alors de tous les champs et méthodes de la classe `A`, auxquels ils peuvent ajouter de nouveaux champs et de nouvelles méthodes. La notion d'héritage s'accompagne d'une notion de *sous-typage* : toute valeur de type `B` peut être vue comme une valeur de type `A`. En Java, chaque classe hérite d'au plus une classe ; on appelle cela l'*héritage simple* (par opposition à l'héritage multiple, qui existe dans d'autres langage orientés objets, comme C++). La relation d'héritage forme donc un arbre

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```




Prenons comme exemple un ensemble de classes pour représenter des objets graphiques (cercles, rectangles, etc.). On introduit en premier lieu une classe `Graphical` représentant n'importe quel objet graphique :

```
class Graphical {
  int x, y;          /* centre */
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  void draw() { /* ne fait rien */ }
}
```

On a quatre champs, pour le centre et les dimensions maximales de l'objet, et deux méthodes, `move` pour déplacer l'objet et `draw` pour le dessiner. Pour représenter un rectangle, on hérite de la classe `Graphical`.

```
class Rectangle extends Graphical {
```

On hérite donc des champs `x`, `y`, `width` et `height` et des méthodes `move` et `draw`. On peut écrire un constructeur qui prend en arguments deux coins du rectangle :

```
Rectangle(int x1, int y1, int x2, int y2) {
  this.x = (x1 + x2) / 2;
  this.y = (y1 + y2) / 2;
  this.width = Math.abs(x1 - x2);
  this.height = Math.abs(y1 - y2);
}
```

On peut utiliser directement toute méthode héritée de `Graphical`. On peut écrire par exemple

```
Rectangle p = new Rectangle(0, 0, 100, 50);
p.move(10, 5);
```

Pour le dessin, en revanche, on va *redéfinir* la méthode `draw` dans la classe `Rectangle` (en anglais on parle d'*overriding*)

```
class Rectangle extends Graphical {
  ...
  void draw() { /* dessine le rectangle */ }
}
```

et le rectangle sera alors effectivement dessiné quand on appelle

```
p.draw();
```

### Type statique et type dynamique

La construction `new C(...)` construit un objet de classe `C`, et la classe de cet objet ne peut être modifiée par la suite ; on l'appelle le *type dynamique* de l'objet. En revanche, le *type statique* d'une expression, tel qu'il est calculé par le compilateur, peut être différent du type dynamique, du fait de la relation de sous-typage introduite par l'héritage. Ainsi, on peut écrire

```
Graphical g = new Rectangle(0, 0, 100, 50);
g.draw(); // dessine le rectangle
```

Pour le compilateur, `g` a le type `Graphical`, mais le rectangle est effectivement dessiné : c'est bien la méthode `draw` de la classe `Rectangle` qui est exécutée.

On procède de même pour définir des cercles. Ici on ajoute un champ `radius` pour le rayon, afin de le conserver.

```
class Circle extends Graphical {
  int radius;
  Circle(int x, int y, int r) {
    this.x = x;
    this.y = y;
    this.radius = r;
    this.width = this.height = 2 * r;
  }
  void draw() { /* dessine le cercle */ }
}
```

Introduisons enfin un troisième type d'objet graphique, `Group`, qui est simplement la réunion de plusieurs objets graphiques. Un groupe hérite de `Graphical` et contient une liste d'objets de la classe `Graphical`. On utilise pour cela la classe `LinkedList` de la bibliothèque Java.

```
class Group extends Graphical {
  LinkedList<Graphical> group;

  Group() {
    this.group = new LinkedList<Graphical>();
  }
```

La classe `LinkedList` est une classe générique, paramétrée par le type des éléments contenus dans la liste, ici `Graphical`. On indique ce type avec la notation `<Graphical>` juste après le nom de la classe générique. (La notion de classe générique est expliquée en détail un peu plus loin.) Initialement la liste est vide. On définit une méthode `add` pour ajouter un objet graphique à cette liste.

```
  void add(Graphical g) {
    this.group.add(g);
    // + mise à jour de x,y,width,height
  }
```

Il reste à redéfinir les méthodes `draw` et `move`. Pour dessiner un groupe, il faut dessiner tous les éléments qui le composent, c'est-à-dire tous les éléments de la liste `this.group`, chaque élément `g` étant dessiné en appelant sa propre méthode `draw`. Pour parcourir la liste `this.group` on utilise la construction `for` de Java :

```
  void draw() {
    for (Graphical g : this.group)
      g.draw();
  }
```

Cette construction affecte successivement à la variable `g` les différents éléments de la liste `this.group` et, pour chacun, exécute le corps de la boucle. Ici le corps de la boucle est réduit à une seule instruction, à savoir l'appel à `g.draw`. De même, on redéfinit la méthode `move` dans la classe `Group` en appelant la méthode `move` de chaque élément, sans oublier de déplacer également le centre de tout le groupe.

```
void move(int dx, int dy) {
  this.x += dx;
  this.y += dy;
  for (Graphical g : this.group)
    g.move(dx, dy);
}
```

L'essence de la programmation orientée objet est résumée dans ces deux méthodes. On appelle la méthode `draw` (ou `move`) sur un objet `g` de type statique `Graphical`, sans savoir s'il s'agit d'un rectangle, d'un cercle, ou même d'un autre groupe. En fonction de la nature de cet objet, le code correspondant sera appelé. À cet endroit du programme, le compilateur ne peut pas le connaître. C'est pourquoi on parle d'*appel dynamique* de méthode. (D'une manière générale, « statique » désigne ce qui est connu/fait au moment de la compilation, et « dynamique » désigne ce qui est connu/fait au moment de l'exécution.)

#### La classe `Object`

Une classe qui n'est pas déclarée comme héritant d'une autre classe hérite de la classe `Object`. Il s'agit d'une classe prédéfinie dans la bibliothèque Java, de son vrai nom `java.lang.Object`. Par conséquent, toute classe hérite, directement ou indirectement, de la classe `Object`. Parmi les méthodes de la classe `Object`, dont toute classe hérite donc, on peut citer notamment les trois méthodes

```
public boolean equals(Object o);
public int hashCode();
public String toString();
```

dont nous reparlerons par la suite. En particulier, certaines classes peuvent (doivent) redéfinir ces méthodes de façon appropriée. Un exemple est donné dans la section 5.3.

### 1.1.5 Classes abstraites

Dans l'exemple donné plus haut d'une hiérarchie de classes pour représenter des objets graphiques, il n'y a jamais lieu de créer d'instance de la classe `Graphical`. Elle nous sert de type commun à tous les objets graphiques, mais elle ne représente pas un objet particulier. C'est ce que l'on appelle une *classe abstraite* et on peut l'indiquer avec le mot-clé `abstract`.

```
abstract class Graphical {
  ...
}
```

Ainsi il ne sera pas possible d'écrire `new Graphical()`. Du coup, plutôt que d'écrire dans la classe `graphical` une méthode `draw` qui ne fait rien, on peut se contenter de la *déclarer* comme une méthode abstraite.

```
abstract void draw();
```

Le compilateur nous imposera alors de la redéfinir dans les sous-classes de `Graphical`, à moins que ces sous-classes soient elles-mêmes abstraites.

### 1.1.6 Classes génériques

Une classe peut être paramétrée par une ou plusieurs classes. On parle alors de classe *générique*. L'exemple le plus simple est sûrement celui d'une classe `Pair` pour représenter une paire formée d'un objet d'une classe `A` et d'un autre d'une classe `B`[1]. Il s'agit donc d'une classe paramétrée par les classes `A` et `B`. On la déclare ainsi :

```
class Pair<A, B> {
```

Les paramètres, ici `A` et `B`, sont indiqués entre les symboles `<` et `>`. À l'intérieur de la classe `Pair`, on utilise `A` ou `B` comme tout autre nom de classe. Ainsi, on déclare deux champs `fst` et `snd` avec

```
  A fst;
  B snd;
```

et le constructeur naturel avec

```
  Pair(A a, B b) {
    this.fst = a;
    this.snd = b;
  }
```

(On note qu'on ne répète pas les paramètres dans le nom du constructeur, car ils sont identiques à ceux de la classe.) De la même manière, les paramètres peuvent être utilisés dans les déclarations et définitions de méthodes. Ainsi, on peut écrire une méthode renvoyant la première composante d'une paire, c'est-à-dire une valeur de type `A` :

```
  A getFirst() { return this.fst; }
```

Pour utiliser une telle classe générique, il convient d'*instancier*[2] les paramètres formels `A` et `B` par des paramètres effectifs, c'est-à-dire par deux expressions de type. Ainsi on peut écrire

```
  Pair<Integer, String> p0 = new Pair<Integer, String>(89, "Fibonacci");
```

pour déclarer une variable `p0` contenant une paire formée d'un entier et d'une chaîne de caractères. Une telle déclaration peut être faite aussi bien dans la classe `Pair` qu'à l'extérieur, dans une autre classe. Comme on le voit, la syntaxe pour réaliser l'instanciation est, sans surprise, la même que pour la déclaration. Comme on le voit également, l'instanciation doit être répétée après `new Pair`. On note que le premier paramètre a été instancié par `Integer` et non pas `int`. En effet, seule une expression de type dénotant une classe peut être utilisée pour instancier une classe générique, et `int` ne désigne pas une classe. La classe `Integer` de la bibliothèque Java a justement pour rôle d'encapsuler un entier de type `int` dans un objet. La création de cet objet est ajoutée automatiquement par le compilateur, ce qui nous permet d'écrire `89` au lieu de `new Integer(89)`. La bibliothèque Java contient des classes similaires `Boolean`, `Long`, `Double`, etc.

1. D'une façon assez surprenante, une telle classe n'existe pas dans la bibliothèque standard Java.
2. On nous pardonnera cet anglicisme.

#### Code statique générique

Pour écrire un code statique générique, on doit préciser les paramètres après le mot clé `static`, car ils ne coïncident pas nécessairement avec ceux de la classe. Ainsi, une méthode qui échange les deux composantes d'une paire s'écrit

```
static<A, B> Pair<B, A> swap(Pair<A, B> p) {
  return new Pair<B, A>(p.snd, p.fst);
}
```

Là encore, on peut écrire une telle déclaration aussi bien dans la classe `Pair` qu'à l'extérieur, dans une autre classe. Les paramètres d'un code statique ne sont pas nécessairement les mêmes que ceux de la classe générique. Par exemple on peut écrire

```
static<C> Pair<C, C> twice(C a) { return new Pair<C, C>(a, a); }
```

pour renvoyer une paire formée de deux fois le même objet. Lorsqu'on utilise une méthode statique générique, l'instanciation est inférée par le compilateur. Ainsi on écrit seulement

```
Pair<String, Integer> p1 = Pair.swap(p0);
```

Si cela est nécessaire, on peut donner l'instanciation explicitement, avec la syntaxe un peu surprenante suivante :

```
Pair<String, Integer> p1 = Pair.<Integer, String>swap(p0);
```

### 1.1.7 Interfaces

Le langage Java fournit un mécanisme pour réaliser un *contrat* entre une fournisseur de code et son client. Ce mécanisme s'appelle une *interface*. Une interface est un ensemble de méthodes. Voici par exemple une interface minimale pour une structure de pile contenant des entiers.

```
interface Stack {
  boolean isEmpty()
     void push(int x)
      int pop()
}
```

Elle déclare trois méthodes `isEmpty`, `push` et `pop`. Du côté du code client, cette interface peut être utilisée comme un type. On peut ainsi écrire une méthode `sum` qui vide une pile et renvoie la somme de ses éléments de la manière suivante :

```
static int sum(Stack s) {
  int r = 0;
  while (!s.isEmpty()) r += s.pop();
  return r;
}
```

Pour le compilateur, tout se passe comme si `Stack` était une classe avec trois méthodes `isEmpty`, `push` et `pop`. On peut appeler cette méthode avec toute classe qui déclare implémenter l'interface `Stack`.

Côté fournisseur, justement, cette déclaration se fait à l'aide du mot-clé `implements`, de la manière suivante :

```
class MyIntStack implements Stack {
  ..
}
```

Le compilateur va alors exiger la présence des trois méthodes `isEmpty`, `push` et `pop`, avec les types attendus. Bien entendu, la classe `MyIntStack` peut déclarer d'autres méthodes que celles de l'interface `Stack` et elles seront visibles. Une interface n'inclut pas le ou les constructeurs. Une classe peut implémenter plusieurs interfaces.

Bien qu'il ne s'agit pas d'héritage, une relation de sous-typage existe qui permet à tout objet d'une classe implémentant l'interface `Stack` d'être considéré comme étant de type `Stack`. Ainsi, on peut écrire

```
Stack s = new MyIntStack();
```

L'intérêt d'une telle déclaration est d'expliciter l'abstraction dont on a besoin. Ici, on affirme « j'ai besoin d'une pile `s` » et la suite du code se moque de savoir qu'il s'agit d'une valeur de type `MyIntStack`. En particulier, on peut choisir de remplacer plus tard `MyIntStack` par une autre classe qui implémente l'interface `Stack` et le reste du code n'aura pas besoin d'être modifié.

Une interface peut être générique. Voici un exemple d'interface générique fournie par la bibliothèque Java.

```
interface Comparable<K> {
  int compareTo(K k);
}
```

On l'utilise pour exiger que les objets d'une classe donnée soient comparables entre eux. Des exemples d'utilisation se trouvent dans les sections 6.3.4, 8.4 13.5 ou encore 14.2.

### 1.1.8 Règles de visibilité

Nous avons expliqué plus haut que le qualificatif `private` peut être utilisé pour limiter la visibilité d'un champ. Plus généralement, il existe quatre niveaux de visibilité différents en Java :

- `private` : visibilité limitée à la classe ;
- `protected` : visibilité limitées à la classe et à ses sous-classes ;
- aucun qualificatif : visibilité limitée au paquetage, c'est-à-dire au dossier dans lequel la classe est définie ;
- `public` : visibilité illimitée.

Ces qualificatifs de visibilité s'appliquent aussi bien aux champs qu'aux méthodes et aux constructeurs. Pour des raisons de simplicité, ce polycopié omet le qualificatif `public` la plupart du temps ; en pratique, il faudrait l'ajouter à toute classe ou méthode d'intérêt général.

## 1.2 Modèle d'exécution

Pour utiliser un langage de programmation correctement, et efficacement, il convient d'en comprendre le modèle d'exécution, c'est-à-dire la façon dont sont représentées les

valeurs qu'il manipule et le coût de ses différentes opérations, du moins à un certain niveau de détails. Cette section décrit quelques éléments du modèle d'exécution de Java. Pour le lecteur qui connaît le langage Python, cette section pourra avantageusement être complétée par la lecture de l'annexe B qui fait un bref comparatif des langages Java et Python.

### 1.2.1 Arithmétique des ordinateurs

Le langage Java fournit plusieurs types numériques primitifs, à savoir cinq types d'entiers (`byte`, `short`, `char`, `int` et `long`) et deux types de nombres à virgule flottante (`float` et `double`).

**Entiers.** Un entier est représenté en base 2, sur $n$ chiffres appelés *bits*. Ces chiffres sont conventionnellement numérotés de droite à gauche :

| $b_{n-1}$ | $b_{n-2}$ | ... | $b_1$ | $b_0$ |
|---|---|---|---|---|

Le bit $b_0$ est appelé le *bit de poids faible* et le bit $b_{n-1}$ le *bit de poids fort*. Selon le type Java, $n$ vaut 8, 16, 32 ou 64. Par la suite, on utilisera la notation $101010_2$ pour dénoter une suite de bits.

L'interprétation la plus simple de ces $n$ bits est celle d'un entier *non signé* en base 2, dont la valeur est donc

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

Les valeurs possibles vont de 0, c'est-à-dire $00...00_2$, à $2^n - 1$, c'est-à-dire $11...11_2$. C'est le cas du type `char` de Java, qui est un entier non signé de 16 bits. Ses valeurs possibles vont donc de 0 à $2^{16} - 1 = 65535$.

Pour représenter un entier *signé*, on interprète le bit de poids fort $b_{n-1}$ comme un bit de signe, la valeur 0 désignant un entier positif ou nul et la valeur 1 un entier strictement négatif. Plutôt que de simplement mettre un signe devant un entier représenté par les $n-1$ bits restants, les ordinateurs utilisent une représentation plus subtile appelée *complément à deux*. Elle consiste à interpréter les $n$ bits comme la valeur suivante :

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

Les valeurs possibles s'étendent donc de $-2^{n-1}$, c'est-à-dire $100...000_2$, à $2^{n-1} - 1$, c'est-à-dire $011...111_2$. On notera la dissymétrie de cette représentation, avec une valeur de plus à gauche de 0 qu'à droite. En revanche, il n'y a qu'une seule représentation de 0, à savoir $00...00_2$. En Java, les types `byte`, `short`, `int` et `long` désignent des entiers signés, sur respectivement 8, 16, 32 et 64 bits. Les plages de valeurs de ces types sont donc respectivement $-2^7..2^7 - 1$, $-2^{15}..2^{15} - 1$, $-2^{31}..2^{31} - 1$ et $-2^{63}..2^{63} - 1$.

Les constantes littérales peuvent être écrites dans le système décimal, de la façon usuelle, mais aussi hexadécimal (base 16) avec le préfixe `0x`, octal (base 8) avec le préfixe `0` ou encore binaire (base 2) avec le préfixe `0b`. Ainsi, `0x2A`, `052` et `0b101010` sont trois autres façons d'écrire la constante `42`.

Il est important de signaler qu'un calcul arithmétique peut provoquer un *débordement de capacité* et que ce dernier n'est pas signalé, ni par le compilateur (qui ne pourrait pas le faire de manière générale) ni à l'exécution. Ainsi, le résultat de `100000 * 100000` est `1410065408`. Le résultat peut même être du mauvais signe. Ainsi le résultat de `200000 * 100000` est `-1474836480`.

Outre les opérations arithmétiques élémentaires, le langage Java fournit également des opérations permettant de manipuler directement la représentation binaire d'un entier, c'est-à-dire d'un entier vu simplement comme $n$ bits. L'opération `~` est la négation logique, qui échange les 0 et les 1, et les opérations `&`, `|` et `^` sont respectivement le ET, le OU et le OU exclusif appliqués bit à bit aux bits de deux entiers. Il existe également des opérations de *décalage* des bits. L'opération `<<` $k$ est un décalage logique à gauche, qui insère $k$ zéros de poids faible. De même, l'opération `>>>` $k$ est un décalage logique à droite, qui insère $k$ zéros de poids fort. Enfin, l'opération `>>` $k$ est un décalage arithmétique à droite, qui réplique le bit de signe $k$ fois. Dans certaines circonstances, on se sert de ces opérations pour manipuler une valeur de type `int` qui ne représente pas directement un entier mais plutôt un petit tableau de booléens, de taille inférieure ou égale à 32. En particulier, on peut représenter facilement un sous-ensemble de $\{0, 1, \ldots, 31\}$ avec une valeur de type `int`, le $i$-ième bit indiquant la présence de l'élément $i$ dans l'ensemble. Des exemples sont donnés dans les chapitres 11 et 12.

**Nombres à virgule flottante.** Les ordinateurs fournissent également des nombres décimaux, appelés *nombres à virgule flottante* ou plus simplement *flottants*. Un tel nombre est également codé sur $n$ bits, dont certains sont interprétés comme un entier signé $m$ appelé mantisse et d'autres comme un autre entier signé $e$ appelé exposant. Le nombre flottant représenté est alors $m \times 2^e$.

En Java, le type `float` désigne un flottant représenté sur 32 bits. Ses valeurs s'étendent de $-3,4 \times 10^{38}$ à $3,4 \times 10^{38}$, le plus petit flottant positif représentable étant $1,4 \times 10^{-45}$. Le type `double` désigne un flottant représenté sur 64 bits. Ses valeurs s'étendent de $-1,8 \times 10^{308}$ à $1,8 \times 10^{308}$, le plus petit flottant positif représentable étant $4,9 \times 10^{-324}$. Une constante de type flottant peut être écrite en notation décimale, comme `3.14`, en notation scientifique, comme `1.2e-9` pour $1,2 \times 10^{-9}$, ou encore en notation hexadécimale, comme `0x1.5p3` pour $1.5_{16} \times 2^3 = 10,5$. Par défaut, une constante a pour type `double`, à moins que le suffixe `f` ne soit ajouté pour spécifier le type `float`.

Ce cours n'entre pas dans les détails de cette représentation, qui est complexe, mais plusieurs choses importantes se doivent d'être signalées. En premier lieu, il faut avoir conscience que la plupart des nombres réels, et même la plupart des nombres décimaux, ne sont pas représentables par un flottant. En conséquence, les résultats des calculs sont arrondis et il faut en tenir compte dans les programmes que l'on écrit. Ainsi, le nombre $0,1$ n'est pas représentable et un simple calcul comme $1732 \times 0,1$ donne en réalité un flottant qui n'est pas égal à $173,2$. En particulier, une condition dans un programme ne doit pas tester en général qu'un nombre flottant est nul, mais plutôt qu'il est inférieur à une borne donnée, par exemple $10^{-9}$. En revanche, si les calculs sont arrondis, ils le sont d'une manière qui est spécifiée par un standard, à savoir le standard IEEE 754 [9]. En particulier, ce standard spécifie que le résultat d'une opération, par exemple la multiplication $1732 \times 0,1$ ci-dessus, doit être le flottant le plus proche du résultat exact.

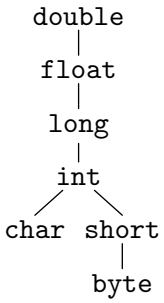



FIGURE 1.1 – Conversions automatiques entre les types numériques de Java.

**Conversions automatiques.** Les valeurs des différents types numériques de Java subissent des conversions automatiques d'un type vers un autre dans certaines circonstances. Ainsi, si une méthode doit renvoyer un entier de type `int`, on peut écrire `return c` où `c` est de type `char`. De même on peut affecter à une variable de type `long` un entier de type `int`. La figure 1.1 illustre les différentes conversions automatiques de Java, chaque trait de bas en haut étant vu comme une conversion et la conversion étant transitive. Les conversions entre les types entiers se font sans perte. En revanche, les conversions vers les types flottants peuvent impliquer un arrondi (on peut s'en convaincre par un argument combinatoire, par exemple en constatant que le type `long` contient plus de valeurs que le type `float`).

Lorsque la conversion n'est pas possible, le compilateur Java indique une erreur. Ainsi, on ne peut pas affecter une valeur de type `int` à une variable de type `char` ou encore renvoyer une valeur de type `float` dans une méthode qui doit renvoyer un entier.

Lors d'un calcul arithmétique, Java utilise le plus petit type à même de recevoir le résultat du calcul, en effectuant une promotion de certaines des opérandes si nécessaire. Ainsi, si on ajoute une valeur de type `char` et une valeur de type `int`, le résultat sera de type `int`. Plus subtilement, l'addition d'un `char` et d'un `short` sera de type `int`.

### 1.2.2 Mémoire

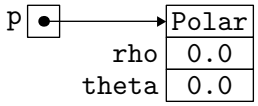
Cette section explique comment la mémoire est structurée et notamment comment les objets y sont représentés. Reprenons l'exemple de la classe `Polar` de la section précédente

```
class Polar {
  double rho, theta;
}
```

et construisons un nouvel objet de la classe `Polar` dans une variable locale `p` :

```
Polar p = new Polar();
```

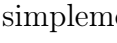
On a une situation que l'on peut schématiser ainsi :



Les petites boîtes correspondent à des zones de la mémoire. Les noms à côté de ces boîtes (`p`, `rho`, `theta`) n'ont pas de réelle incarnation en mémoire. En tant que noms, ils n'existent que dans le programme source. Une fois celui-ci compilé et exécuté, ces noms sont devenus des *adresses* désignant des zones de la mémoire, qui ne sont rien d'autre que des entiers. Ainsi la boîte `p` contient en réalité une adresse (par exemple 1381270477) à laquelle on trouve les petites boîtes dessinées à droite. Dans une est mentionnée la classe de l'objet, ici `Polar`. Là encore, il ne s'agit pas en mémoire d'un nom, mais d'une représentation plus bas niveau (en réalité une autre adresse mémoire vers une description de la classe `Polar`). Dans d'autres boîtes on trouve les valeurs des deux champs `rho` et `theta`. Là encore on a explicité le nom à côté de chaque champ mais ce nom n'est pas représenté en mémoire. Le compilateur sait qu'il a rangé le champ `rho` à une certaine distance de l'adresse de l'objet et c'est tout ce dont il a besoin pour retrouver la valeur du champ `rho`.

Notre schéma est donc une simplification de l'état de la mémoire, les zones mémoires apparaissent comme des cases (les variables portent un nom) et les adresses apparaissent comme des flèches qui pointent vers les cases, alors qu'en réalité il n'y a rien d'autre que des entiers rangés dans la mémoire à des adresses qui sont elles-mêmes des entiers. Par ailleurs, notre schéma est aussi une simplification car la représentation en mémoire d'un objet Java est plus complexe : elle contient aussi des informations sur l'état de l'objet. Mais ceci ne nous intéresse pas ici. Dans ce polycopié, on s'autorisera parfois même à ne pas écrire la classe dans la représentation d'un objet quand celle-ci est claire d'après le contexte.

### Tableaux

Un tableau est un objet un peu particulier, puisque ses différentes composantes ne sont pas désignées par des noms de champs mais par des indices entiers. Néanmoins l'idée reste la même : un tableau occupe une zone contiguë de mémoire, dont une petite partie décrit le type du tableau et sa longueur et le reste contient les différents éléments. Dans la suite, on représentera un tableau de manière simplifiée, avec uniquement ses éléments présentés horizontalement. Ainsi un tableau contient les trois entiers 1, 2 et 3 sera tout simplement représenté par 

.

### Allocation et libération

Comme expliqué ci-dessus, l'utilisation de la construction `new` de Java conduit à une *allocation* de mémoire. Celle-ci se fait dans la partie de la mémoire appelée le *tas* — l'autre étant la pile, décrite dans le paragraphe suivant. Si la mémoire vient à s'épuiser, l'exception `OutOfMemoryError` est levée. Au fur et à mesure de l'exécution du programme, de la mémoire peut être récupérée, lorsque les objets correspondants ne sont plus utilisés. Cette libération de mémoire n'est pas à la charge du programmeur (contrairement à d'autres langages comme C++) : elle est réalisée automatiquement par le *Garbage Collector* (ou GC). Celui-ci libère la mémoire allouée pour un objet lorsque cet objet ne peut plus être référencé à partir des variables du programme ou d'autres objets pouvant encore être référencés. La libération de mémoire est effectuée incrémentalement, c'est-à-dire par petites étapes, au fur et à mesure de l'exécution du programme. En première approximation, on peut considérer que le coût de la libération de mémoire est uniformément réparti sur l'ensemble de l'exécution. En particulier, on peut s'autoriser à penser que le coût

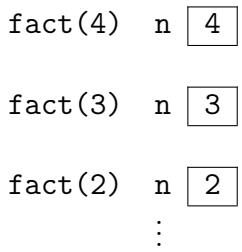
d'une expression `new` se limite à celui du code du constructeur, c'est-à-dire que le coût de l'allocation proprement dite est constant.

### Pile d'appels

Dans la plupart des langages de programmation, et en Java en particulier, les appels de fonctions/méthodes obéissent à une logique « dernier appelé, premier sorti », c'est-à-dire que, si une méthode `f` appelle une méthode `g`, l'appel à `g` terminera avant l'appel à `f`. Cette propriété permet au compilateur d'organiser les données locales à un appel de fonction (paramètres et variables locales) sur une pile. Illustrons ce principe avec l'exemple d'une méthode récursive[3] calculant la factorielle de `n`.

```
static int fact(int n) {
  if (n == 0) return 1;
  return n * fact(n-1);
}
```

Si on évalue l'expression `fact(4)`, alors le paramètre formel `n` de la méthode `fact` sera matérialisé quelque part en mémoire et recevra la valeur `4`. Puis l'évaluation de `fact(4)` va conduire à l'évaluation de `fact(3)`. De nouveau, le paramètre formel `n` de la méthode `fact` doit être matérialisé pour recevoir la valeur `3`. On comprend qu'on ne peut pas réutiliser le même emplacement que pour l'appel `fact(4)`, sinon la valeur `4` sera perdue, alors même qu'il nous reste à effectuer une multiplication par cette valeur à l'issue de l'appel à `fact(3)`. On a donc une seconde matérialisation en mémoire du paramètre `n`, et ainsi de suite. Lorsqu'on en est au calcul de `fact(2)`, on se retrouve donc dans la situation suivante :




On visualise bien une structure de pile (qui croît ici vers le bas). Lorsqu'on parvient finalement à l'appel à `fact(0)`, on atteint enfin une instruction `return`, qui conclut l'appel à `fact(0)`. La variable `n` contenant `0` est alors *dépilée* et on revient à l'appel à `fact(1)`. On peut alors effectuer la multiplication `1 * 1` puis c'est l'appel à `fact(1)` qui est terminé et la variable `n` contenant `1` qui est *dépilée*. Et ainsi de suite jusqu'au résultat final.

La pile a une capacité limitée. Si elle vient à s'épuiser, l'exception `StackOverflowError` est levée. Par défaut, la taille de pile est relativement petite, de l'ordre de 1 Mo, ce qui correspond à environ 10 000 appels imbriqués (bien entendu, cela dépend de l'occupation sur la pile de chaque appel de méthode). On peut modifier la taille de la pile avec l'option `-Xss` de la machine virtuelle Java. Ainsi

```
java -Xss100m Test
```

exécute le programme `Test` avec 100 Mo de pile.

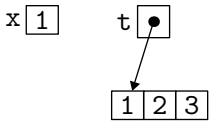
3. La pile d'appels n'est pas liée à la récursivité mais à la notion d'appels imbriqués, mais une fonction récursive conduit naturellement à des appels imbriqués.

### 1.2.3 Valeurs

Il y a en Java deux grandes catégories de valeurs : les *valeurs primitives* et les *objets*. La distinction est en fait technique, elle tient à la façon dont ces valeurs sont traitées par la machine, ou plus exactement sont rangées dans la mémoire. Une valeur primitive se suffit à elle-même ; il s'agit d'un entier, d'un caractère, ou encore d'un booléen. La valeur d'un objet est une *adresse*, désignant une zone de la mémoire. On parle aussi de *pointeur*. Écrivons par exemple

```
int x = 1 ;
int[] t = {1, 2, 3} ;
```

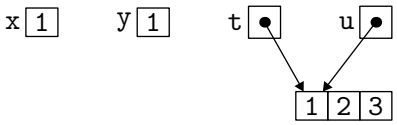
Les variables `x` et `t` sont deux cases, qui contiennent chacune une valeur, la première valeur étant primitive et la seconde un pointeur. Un schéma résume la situation :



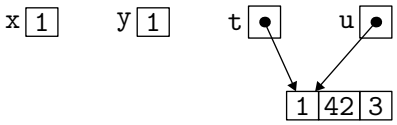

En particulier, la valeur de la variable `t` est un pointeur vers l'objet en mémoire représentant le tableau, c'est-à-dire contenant ses éléments. Si $x$ et $y$ sont deux variables, la construction $y = x$ se traduit par une copie de la *valeur* contenue dans la variable $x$ dans la variable $y$, que cette valeur soit un pointeur ou non. Ainsi, si on poursuit le code ci-dessus avec les deux instructions suivantes

```
int y = x ;
int[] u = t ;
```

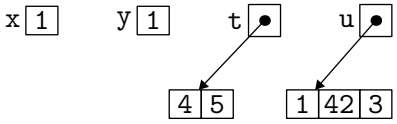
alors on obtient maintenant l'état mémoire suivant :




En particulier, les deux variables `t` et `u` pointent vers le même tableau. On dit que ce sont des *alias*. On peut s'en convaincre en modifiant un élément de `u` et en vérifiant que la modification s'observe également dans `t`. Si par exemple on exécute `u[1] = 42;` alors on obtient




ce que l'on peut observer facilement, par exemple en affichant la valeur de `t[1]`. Cependant, `t` et `u` ne sont pas liés à jamais. Si on affecte à `t` un nouveau tableau, par exemple avec `t = new int[] {4, 5};`, alors on a la situation suivante



où `u` désigne toujours le même tableau qu'auparavant.

Il existe un pointeur qui ne pointe nulle part : `null`. Nous pouvons l'employer partout où un pointeur est attendu. Dans les schémas nous représentons `null` par le symbole $\perp$. Puisque `null` ne pointe nulle part il n'est pas possible de le « *déréférencer* » c'est-à-dire d'aller voir où il pointe. Toute tentative se traduit par une erreur à l'exécution, à savoir le déclenchement de l'exception `NullPointerException`.

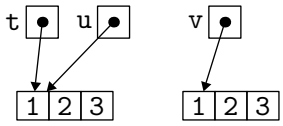
Toute valeur qui n'est pas initialisée (variable, champ, élément de tableau) reçoit une valeur par défaut. Le langage spécifie qu'il s'agit de 0 dans le cas d'un entier, d'un caractère ou d'un flottant, `false` dans le cas d'un booléen, et `null` dans les autres cas (objet ou tableau).

### Égalité des valeurs

L'opérateur d'égalité `==` de Java teste l'égalité de deux valeurs. De même, l'opérateur `!=` teste leur différence. Pour des valeurs primitives, cela coïncide avec l'égalité mathématique. En revanche, pour deux objets, c'est-à-dire deux valeurs qui sont des pointeurs, il s'agit tout simplement de l'égalité de ces deux pointeurs. Autrement dit, le programme

```
int[] t = {1, 2, 3} ;
int[] u = t ;
int[] v = {1, 2, 3} ;
System.out.println("t==u : " + (t == u) + ", t==v : " + (t == v)) ;
```

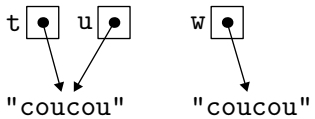
affiche `t==u : true, t==v : false`. Les pointeurs `t` et `u` sont égaux parce qu'ils pointent vers le même objet. Les pointeurs `t` et `v`, qui pointent vers des objets distincts, sont distincts. Cela peut se comprendre si on revient aux états mémoire simplifiés.




On dit parfois que `==` est l'*égalité physique*. L'égalité physique donne parfois des résultats surprenants. Ainsi le programme suivant

```
String t = "coucou" ;
String u = "coucou" ;
String v = "cou" ;
String w = v + v ;
System.out.println("t==u : " + (t == u) + ", t==w : " + (t == w)) ;
```

affiche `t==u : true, t==w : false`, ce qui révèle que les chaînes (objets) référencés par `t` et `u` sont exactement les mêmes (le compilateur les a partagées), tandis que `w` est une autre chaîne.



La plupart du temps, un programme a besoin de savoir si deux chaînes ont exactement les mêmes caractères et non si elles occupent la même zone de mémoire. Il en résulte principalement qu'il ne faut pas tester l'égalité des chaînes, et des objets en général le plus souvent, par `==`. Dans le cas des chaînes, il existe une méthode `equals` spécialisée qui compare les chaînes caractère par caractère. On dit que la méthode `equals` est l'*égalité structurelle* des chaînes. De manière générale, c'est à la charge du programmeur Java que d'écrire une méthode pour réaliser l'égalité structurelle, si besoin est. Un exemple d'égalité structurelle est donné dans la section 5.3.

### Passage par valeur

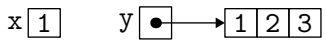
En Java, le mode de passage est *par valeur*. Cela signifie que, lorsqu'une méthode est appelée, les valeurs des arguments effectifs de l'appel sont *copiées* vers de nouveaux emplacements mémoire. Mais il faut bien garder à l'esprit qu'une telle valeur est soit une valeur primitive, soit un pointeur vers une zone de la mémoire, comme expliqué ci-dessus. En particulier, dans ce second cas, seul le *pointeur* est copié. Considérons par exemple la méthode `f` suivante

```
static void f(int a, int[] b) {
  a = 2;
  b[2] = 7;
}
```

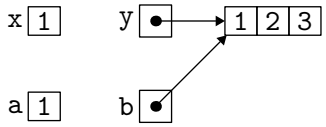
appelée dans le contexte suivant :

```
int x = 1;
int[] y = {1, 2, 3};
f(x, y);
```

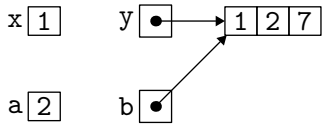
Juste avant l'appel à `f` on a la situation suivante :



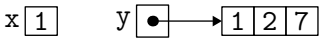

Juste après l'appel à `f` on a deux nouvelles variables `a` et `b` (les arguments formels de `f`), qui ont reçu respectivement les valeurs de `x` et `y` :




En particulier, les deux variables `y` et `b` sont des alias pour le même tableau. Les deux affectations `a = 2` et `b[2] = 7` conduisent donc à la situation suivante :

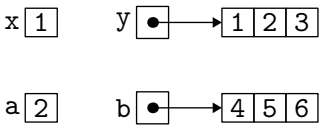



Après l'appel à `f`, les variables `a` et `b` sont détruites, et on se retrouve donc dans la situation finale suivante, où le *contenu* du tableau `y` a été modifié, mais pas celui de la variable `x` :

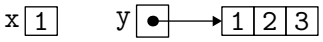



Cet exemple utilise un tableau, mais la situation serait la même si `y` était un objet et si la méthode `f` modifiait un champ de cet objet : la modification persisterait après l'appel à `f`.

Plus subtilement encore, si on remplace l'affectation `b[2] = 7` dans `f` par `b = new int[] {4, 5, 6}`, on se retrouve dans la situation suivante à la fin du code de `f` :




Après l'appel, on se retrouve donc exactement dans la situation de départ, c'est-à-dire

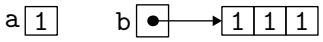



En particulier, le tableau `{4, 5, 6}` n'est plus nulle part référencé et sera donc récupéré par le GC.

Un objet alloué dans une méthode n'est pas systématiquement perdu pour autant. En effet, il peut être renvoyé comme résultat (ou stocké dans une autre structure de données). Si on considère par exemple la méthode suivante

```
static int[] g(int a) {
  int[] b = {a, a, a};
  return b;
}
```

qui alloue un tableau dans une variable locale `b`, alors l'appel à `g(1)` va conduire à la situation suivante




et c'est la *valeur* de `b` qui est renvoyée, c'est-à-dire le pointeur vers le tableau. Bien que les variables `a` et `b` sont détruites, le tableau survivra (si la valeur renvoyée par `g` est utilisée par la suite, bien entendu).

**Exercice 1.** Expliquer pourquoi le programme suivant ne peut afficher autre chose que `1`, quelle que soit la définition de la méthode `f` :

```
int x = 1;
f(x);
System.out.println(x);
```

Solution □

# 2 Notions de complexité

L'objet de la théorie de la complexité est de mesurer les ressources — principalement le temps et la mémoire — utilisées par un programme ou nécessaires à la résolution d'un problème. Ce chapitre vise à en donner les premières bases, en discutant quelques algorithmes et leur efficacité. D'autres exemples sont donnés dans les chapitres suivants, notamment dans le chapitre 13 consacré aux tris.

## 2.1 Complexité d'algorithmes et complexité de problèmes

### 2.1.1 La notion d'algorithme

La théorie de la *calculabilité*, née des travaux fondateurs d'Emil Post et Alan Turing, offre des outils pour formaliser la notion d'algorithme de façon très générale.

Elle s'intéresse en fait essentiellement à discuter les problèmes qui sont résolubles informatiquement, c'est-à-dire à distinguer les problèmes *décidables* (qui peuvent être résolus informatiquement) des problèmes *indécidables* (qui ne peuvent avoir de solution informatique). La calculabilité est très proche de la logique mathématique et de la théorie de la preuve : l'existence de problèmes qui n'admettent pas de solution informatique est très proche de l'existence de théorèmes vrais mais qui ne sont pas démontrables.

La théorie de la *complexité* se focalise sur les problèmes décidables, et s'intéresse aux ressources (temps, mémoire, etc.) nécessaires à la résolution de ces problèmes. C'est typiquement ce dont on a besoin pour mesurer l'efficacité des algorithmes considérés dans ce cours : on considère des problèmes qui admettent une solution, mais pour lesquels on cherche une solution "efficace".

Ces théories permettent de formaliser proprement la notion d'algorithme, en complète généralité, en faisant abstraction du langage utilisé, mais cela dépasserait complètement l'ambition de ce polycopié. Dans ce polycopié, on va prendre la notion suivante d'algorithme : un algorithme $\mathcal{A}$ est un programme Java qui prend en entrée une donnée $d$ et produit en sortie un résultat $\mathcal{A}(d)$. Cela peut par exemple être un algorithme de tri, qui prend en entrée une liste $d$ d'entiers, et produit en sortie cette même liste triée $\mathcal{A}(d)$. Cela peut aussi être par exemple un algorithme qui prend en entrée deux listes, *i.e.* un couple $d$ constitué de deux listes, et renvoie en sortie leur concaténation $\mathcal{A}(d)$.

### 2.1.2 La notion de ressource élémentaire

On mesure toujours l'efficacité, c'est-à-dire la complexité, d'un algorithme en terme d'une mesure élémentaire $\mu$ à valeur entière : cela peut être le nombre d'instructions effectuées, la taille de la mémoire utilisée, le nombre de comparaisons effectuées, ou toute autre mesure.

Il faut simplement qu'étant donnée une entrée $d$, on sache clairement associer à l'algorithme $\mathcal{A}$ sur l'entrée $d$, la valeur de cette mesure, notée $\mu(\mathcal{A}, d)$ : par exemple, pour un algorithme de tri, si la mesure élémentaire $\mu$ est le nombre de comparaisons effectuées, $\mu(\mathcal{A}, d)$ est le nombre de comparaisons effectuées sur l'entrée $d$ (une liste d'entiers) par l'algorithme de tri $\mathcal{A}$ pour produire le résultat $\mathcal{A}(d)$ (cette liste d'entiers triée).

Il est clair que $\mu(\mathcal{A}, d)$ est une fonction de l'entrée $d$. La qualité d'un algorithme $\mathcal{A}$ n'est donc pas un critère absolu, mais une fonction quantitative $\mu(\mathcal{A}, .)$ des données d'entrée vers les entiers.

### 2.1.3 Complexité d'un algorithme au pire cas

En pratique, pour pouvoir appréhender cette fonction, on cherche souvent à évaluer cette complexité pour les entrées d'une certaine *taille* : il y a souvent une fonction *taille* qui associe à chaque donnée d'entrée $d$, un entier $taille(d)$, qui correspond à un paramètre naturel. Par exemple, cette fonction peut être celle qui compte le nombre d'éléments dans la liste pour un algorithme de tri, la taille d'une matrice pour le calcul du déterminant, la somme des longueurs des listes pour un algorithme de concaténation.

Pour passer d'une fonction des données vers les entiers, à une fonction des entiers (les tailles) vers les entiers, on considère alors la complexité *au pire cas* : la complexité $\mu(\mathcal{A}, n)$ de l'algorithme $\mathcal{A}$ sur les entrées de taille $n$ est définie par

$$\mu(\mathcal{A}, n) = \max_{d \,|\, taille(d)=n} \mu(\mathcal{A}, d).$$

Autrement dit, la complexité $\mu(\mathcal{A}, n)$ est la complexité la pire sur les données de taille $n$. Par défaut, lorsqu'on parle de *complexité d'algorithme* en informatique, il s'agit de complexité au pire cas, comme ci-dessus.

Si l'on ne sait pas plus sur les données, on ne peut guère faire plus que d'avoir cette vision pessimiste des choses : cela revient à évaluer la complexité dans le pire des cas (le meilleur des cas n'a pas souvent un sens profond, et dans ce contexte le pessimisme est de loin plus significatif).

**Exemple.** Considérons le problème du calcul du maximum : on se donne en entrée une liste d'entiers naturels $e_1, e_2, \cdots, e_n$, avec $n \geq 1$, et on cherche à déterminer en sortie $M = \max_{1 \leq i \leq n} e_i$, c'est-à-dire le plus grand de ces entiers. Si l'entrée est rangée dans un tableau, la méthode Java suivante résout le problème.

```
static int max(int[] T) {
  int M = T[0];
  for (int i = 1; i < T.length; i++)
    if (M < T[i]) M = T[i];
  return M;
}
```

Si notre mesure élémentaire $\mu$ correspond au nombre de comparaisons entre éléments du tableau $T$, nous en faisons autant que d'itérations de la boucle, c'est-à-dire exactement $n-1$. Nous avons donc $\mu(\mathcal{A}, d) = n-1$ pour cet algorithme $\mathcal{A}$. Ce nombre est indépendant de la donnée $d$ de taille $n$, et donc $\mu(\mathcal{A}, n) = n - 1$.

### 2.1.4 Complexité moyenne d'un algorithme

Pour pouvoir en dire plus, il faut en savoir plus sur les données. Par exemple, qu'elles sont distribuées selon une certaine loi de probabilité. Dans ce cas, on peut alors parler de complexité *en moyenne* : la complexité moyenne $\mu(\mathcal{A}, n)$ de l'algorithme $\mathcal{A}$ sur les entrées de taille $n$ est définie par

$$\mu(\mathcal{A}, n) = \mathbb{E}[\mu(\mathcal{A}, d) \mid d \text{ entrée avec } \mathit{taille}(d) = n],$$

où $\mathbb{E}$ désigne l'espérance (la moyenne). Si l'on préfère,

$$\mu(\mathcal{A}, n) = \sum_{d \,|\, \mathit{taille}(d)=n} \pi(d)\mu(\mathcal{A}, d),$$

où $\pi(d)$ désigne la probabilité d'avoir la donnée $d$ parmi toutes les données de taille $n$.

En pratique, le pire cas est rarement atteint et l'analyse en moyenne peut sembler plus séduisante. Mais il est important de comprendre que l'on ne peut pas parler de moyenne sans loi de probabilité (sans distribution) sur les entrées, ce qui est souvent très délicat à estimer en pratique. Comment anticiper par exemple les matrices qui seront données à un algorithme de calcul de déterminant ? On fait parfois l'hypothèse que les données sont équiprobables (lorsque cela a un sens, comme lorsqu'on trie $n$ nombres entre 1 et $n$ et où l'on peut supposer que les permutations en entrée sont équiprobables), mais cela est bien souvent totalement arbitraire, et pas réellement justifiable. Enfin, les calculs de complexité en moyenne sont plus délicats à mettre en œuvre.

**Exemple.** Considérons le problème de la recherche d'un entier $v$ parmi $n$ entiers donnés $e_1, e_2, \cdots, e_n$, avec $n \geq 1$. Plus précisément, on cherche à déterminer s'il existe un indice $1 \leq i \leq n$ avec $e_i = v$. En supposant les entiers donnés dans un tableau, l'algorithme suivant résout le problème.

```
static boolean contains(int[] T, int v) {
  for (int i = 0; i < T.length; i++)
    if (T[i] == v) return true;
  return false;
}
```

Sa complexité au pire cas en nombre d'instructions élémentaires est linéaire en $n$, puisque la boucle est effectuée $n$ fois dans le pire cas. Supposons que les éléments de `T` sont des nombres entiers distribués de façon équiprobable entre 1 et $k$ (une constante). Il y a donc $k^n$ tableaux. Parmi ceux-ci, $(k-1)^n$ ne contiennent pas `v` et, dans ce cas, l'algorithme procède à exactement $n$ itérations. Dans le cas contraire, l'entier est dans le tableau et sa première occurrence est alors $i$ avec une probabilité de

$$\frac{(k-1)^{i-1}}{k^i}.$$

Il faut alors procéder à $i$ itérations. Au total, nous avons donc une complexité moyenne de

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i.$$

Or

$$\forall x, \sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(il suffit pour établir ce résultat de dériver l'identité $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) et donc

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1 + \frac{n}{k})\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

Lorsque $k$ est très grand devant $n$ (on effectue par exemple une recherche dans un tableau de $n = 1000$ éléments dont les valeurs sont parmi les $k = 2^{32}$ valeurs possibles de type `int`), alors $C \sim n$. La complexité moyenne est donc linéaire en la taille du tableau, ce qui ne nous surprend pas. Lorsqu'en revanche $k$ est petit devant $n$ (on effectue par exemple une recherche parmi peu de valeurs possibles), alors $C \sim k$. La complexité moyenne est donc linéaire en le nombre de valeurs, ce qui ne nous surprend pas non plus.

### 2.1.5 Complexité d'un problème

On peut aussi parler de la *complexité d'un problème* : cela permet de discuter de l'optimalité ou non d'un algorithme pour résoudre un problème donné. On fixe un problème $\mathcal{P}$ : par exemple celui de trier une liste d'entiers. Un algorithme $\mathcal{A}$ qui résout ce problème est un algorithme qui répond à la spécification du problème $\mathcal{P}$ : pour chaque donnée $d$, il produit la réponse correcte $\mathcal{A}(d)$. La complexité du problème $\mathcal{P}$ sur les entrées de taille $n$ est alors définie par

$$\mu(\mathcal{P}, n) = \inf_{\mathcal{A} \text{ algorithme qui résout } \mathcal{P}} \max_{d \text{ entrée avec } taille(d)=n} \mu(\mathcal{A}, d).$$

Autrement dit, on ne fait plus seulement varier les entrées de taille $n$, mais aussi l'algorithme. On considère le meilleur algorithme qui résout le problème, le meilleur étant celui avec la meilleure complexité au sens de la définition précédente, c'est-à-dire au pire cas. C'est donc la complexité du meilleur algorithme au pire cas.

L'intérêt de cette définition est le suivant : si un algorithme $\mathcal{A}$ possède la complexité $\mu(\mathcal{P}, n)$, *i.e.* est tel que $\mu(\mathcal{A}, n) = \mu(\mathcal{P}, n)$ pour tout $n$, alors cet algorithme est clairement optimal. Tout autre algorithme est moins performant, par définition. Cela permet donc de prouver qu'un algorithme est optimal.

Il y a une subtilité cachée dans la définition ci-dessus. Elle considère la complexité du problème $\mathcal{P}$ *sur les entrées de taille* $n$. En pratique, cependant, on écrit rarement un algorithme pour une taille d'entrées fixée, mais plutôt un algorithme qui fonctionne quelle que soit la taille des entrées. Plus subtilement encore, cet algorithme peut ou non procéder différemment selon la valeur de $n$. Une définition rigoureuse de la complexité d'un problème se doit de faire cette distinction ; on parle alors de complexité uniforme et non-uniforme. Mais ceci dépasse largement le cadre de ce cours.

**Exemple.** Reprenons l'exemple de la méthode `max` donné plus haut. On peut se poser la question de savoir s'il est possible de faire moins de $n-1$ comparaisons. La réponse est non ; dit autrement, cet algorithme est optimal en nombre de comparaisons. En effet, considérons la classe $\mathcal{C}$ des algorithmes qui résolvent le problème de la recherche du maximum de $n$ éléments en utilisant comme critère de décision les comparaisons entre éléments. Commençons par énoncer la propriété suivante : tout algorithme $\mathcal{A}$ de $\mathcal{C}$ est tel que tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand. En effet, soit $i_0$ le rang du maximum $M$ renvoyé par l'algorithme sur un tableau $T = e_1, e_2, \ldots, e_n$, c'est-à-dire $e_{i_0} = M = \max_{1 \leq i \leq n} e_i$. Raisonnons par l'absurde : soit $j_0 \neq i_0$ tel que $e_{j_0}$ ne soit pas comparé avec un élément plus grand que lui. L'élément $e_{j_0}$ n'a donc pas été comparé avec $e_{i_0}$ le maximum. Considérons alors le tableau $T' = e_1, e_2, \ldots, e_{j_0-1}, M+1, e_{j_0+1}, \ldots, e_n$ obtenu à partir de $T$ en remplaçant l'élément d'indice $j_0$ par $M+1$. L'algorithme $\mathcal{A}$ effectuera exactement les mêmes comparaisons sur $T$ et $T'$, sans comparer $T'[j_0]$ avec $T'[i_0]$ et renverra donc $T'[i_0]$, ce qui est incorrect. D'où une contradiction, qui prouve la propriété.

Il découle de la propriété qu'il n'est pas possible de déterminer le maximum de $n$ éléments en moins de $n-1$ comparaisons entre éléments. Autrement dit, la complexité du problème $\mathcal{P}$ du calcul du maximum sur les entrées de taille $n$ est $\mu(\mathcal{P}, n) = n-1$. L'algorithme précédent fonctionnant en $n-1$ telles comparaisons, il est optimal pour cette mesure de complexité.

## 2.2 Complexités asymptotiques

En informatique, on s'intéresse le plus souvent à l'ordre de grandeur (l'asymptotique) des complexités quand la taille $n$ des entrées devient très grande.

### 2.2.1 Ordres de grandeur

Dans le cas où la mesure élémentaire $\mu$ est le nombre d'instructions élémentaires, intéressons-nous au temps correspondant à des algorithmes de complexité $n$, $n \log_2 n$, $n^2$, $n^3$, $(\frac{3}{2})^n$, $2^n$ et $n!$ pour des entrées de taille $n$ croissantes, sur un processeur capable d'exécuter un million d'instructions élémentaires par seconde. Nous notons $\infty$ dans le tableau suivant dès que le temps dépasse $10^{25}$ années (ce tableau est emprunté à [10]).

| Complexité | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $(\frac{3}{2})^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | 4 s |
| $n = 30$ | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | 18 min | $10^{25}$ ans |
| $n = 50$ | $< 1$ s | $< 1$ s | $< 1$ s | $< 1$ s | 11 min | 36 ans | $\infty$ |
| $n = 100$ | $< 1$ s | $< 1$ s | $< 1$ s | $1s$ | 12,9 ans | $10^{17}$ ans | $\infty$ |
| $n = 1000$ | $< 1$ s | $< 1$ s | 1s | 18 min | $\infty$ | $\infty$ | $\infty$ |
| $n = 10000$ | $< 1$ s | $< 1$ s | 2 min | 12 jours | $\infty$ | $\infty$ | $\infty$ |
| $n = 100000$ | $< 1$ s | 2 s | 3 heures | 32 ans | $\infty$ | $\infty$ | $\infty$ |
| $n = 1000000$ | 1s | 20s | 12 jours | 31 710 ans | $\infty$ | $\infty$ | $\infty$ |

### 2.2.2 Conventions

Ce type d'expérience invite à considérer qu'une complexité en $(\frac{3}{2})^n$, $2^n$ ou $n!$ ne peut pas être considérée comme raisonnable. On peut discuter de savoir si une complexité en $n^{158}$ est en pratique raisonnable, mais depuis les années 1960 environ, la convention en informatique est que oui : toute complexité bornée par un polynôme en $n$ est considérée comme raisonnable. Si on préfère, cela revient à dire qu'une complexité est raisonnable dès qu'il existe des constantes $c$, $d$, et $n_0$ telles que la complexité est bornée par $cn^d$, pour $n > n_0$. Des complexités non raisonnables sont par exemple $n^{\log n}$, $(\frac{3}{2})^n$, $2^n$ et $n!$.

Cela offre beaucoup d'avantages : on peut raisonner à un temps (ou à un espace mémoire, ou à un nombre de comparaisons) polynomial près. Cela évite par exemple de préciser de façon trop fine le codage, par exemple comment sont codées les matrices pour un algorithme de calcul de déterminant : passer du codage d'une matrice par des listes à un codage par tableau se fait en temps polynomial et réciproquement.

D'autre part, on raisonne souvent à une constante multiplicative près. On considère que deux complexités qui ne diffèrent que par une constante multiplicative sont équivalentes : par exemple $9n^3$ et $67n^3$ sont considérés comme équivalents. Ces conventions expliquent que l'on parle souvent de complexité en temps de l'algorithme sans préciser finement la mesure de ressource élémentaire $\mu$. Dans ce polycopié, par exemple, on ne cherchera pas à préciser le temps de chaque instruction élémentaire Java. Dit autrement, on suppose dans ce polycopié qu'une opération arithmétique ou une affectation entre deux variables Java se fait en temps constant (unitaire) : cela s'appelle le modèle *RAM*.

### 2.2.3 Notation de Landau

En pratique, on discute d'asymptotique de complexités via la notation $O$ de Landau. Soient $f$ et $g$ deux fonctions définies des entiers naturels vers les entiers naturels (comme les fonctions de complexité d'algorithme ou de problèmes). On dit que

$$f(n) = O(g(n))$$

si et seulement si il existe deux constantes positives $n_0$ et $B$ telles que

$$\forall n \geq n_0, f(n) \leq Bg(n).$$

Ceci signifie que $f$ ne croît pas plus vite que $g$. En particulier $O(1)$ signifie "constant(e)". Par exemple, un algorithme qui fonctionne en temps $O(1)$ est un algorithme dont le temps d'exécution est constant et ne dépend pas de la taille des données. C'est donc un ensemble constant d'opérations élémentaires (exemple : l'addition de deux entiers avec les conventions données plus haut).

On dit d'un algorithme qu'il est linéaire s'il utilise $O(n)$ opérations élémentaires. Il est polynomial s'il existe une constante $a$ telle que le nombre total d'opérations élémentaires est $O(n^a)$ : c'est la notion de "raisonnable" introduite plus haut.

## 2.3 Quelques exemples

Nous donnons ici des exemples simples d'analyse d'algorithmes. De nombreux autres exemples sont dans le polycopié.

### 2.3.1 Factorielle

Prenons l'exemple archi-classique de la fonction factorielle, et intéressons-nous au nombre d'opérations arithmétiques (comparaisons, additions, soustractions, multiplications) nécessaires à son calcul. Considérons un premier programme calculant $n!$ à l'aide d'une boucle.

```
static int fact(int n) {
  int f = 1;
  for (int i = 2; i <= n; i++)
    f = f * i;
  return f;
}
```

Nous avons $n-1$ itérations au sein desquelles le nombre d'opérations élémentaires est 3 (une comparaison, une multiplication et une addition), plus une dernière comparaison pour sortir de la boucle. La complexité est donc $C(n) = 1 + 3(n-1) = O(n)$. La complexité de `fact` est donc linéaire. Considérons un second programme calculant $n!$, cette fois récursivement.

```
static int fact(int n) {
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

Soit $C(n)$ le nombre d'opérations nécessaires pour calculer `fact(n)`. Dans le cas récursif, nous avons alors $C(n) = 3 + C(n-1)$ car on effectue une comparaison, une soustraction (avant l'appel) et une multiplication (après l'appel). De plus $C(0) = 1$, car on effectue seulement une comparaison. On en déduit $C(n) = 1 + 3n = O(n)$. La complexité de `fact` est donc linéaire.

Sur cet exemple, il est intéressant de considérer également la complexité en mémoire. Pour la version utilisant une boucle, elle est constante, car limitée aux deux variables locales `f` et `i`. Pour la version récursive, en revanche, elle est en $O(n)$. En effet, la pile d'appels va contenir jusqu'à $n$ appels à `fact`, comme nous l'avons expliqué plus haut page 18.

### 2.3.2 Tours de Hanoï

Le très classique problème des "tours de Hanoï" consiste à déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en se servant d'une tour intermédiaire. Les règles suivantes doivent être respectées : on ne peut déplacer qu'un disque à la fois, et on ne peut placer un disque que sur un disque plus grand ou sur un emplacement vide.

Identifions les tours par un entier. Pour résoudre ce problème, il suffit de remarquer que, si l'on sait déplacer une tour de taille $n$ de la tour `ini` vers `dest`, alors pour déplacer une tour de taille $n+1$ de `ini` vers `dest`, il suffit de déplacer une tour de taille $n$ de `ini` vers `temp`, un disque de `ini` vers `dest` et finalement la tour de hauteur $n$ de `temp` vers `dest`.

```
static void hanoi(int n, int ini, int temp, int dest){
  if (n == 0) return; // rien à faire
  hanoi(n - 1, ini, dest, temp);
  System.out.println("deplace " + ini + " vers " + dest);
  hanoi(n - 1, temp, ini, dest);
}
```

Notons $C(n)$ le nombre d'instructions élémentaires pour calculer `hanoi(n, ini, temp, dest)`. Nous avons $C(n+1) \leq 2C(n) + \lambda$, où $\lambda$ est une constante, et $C(0) = 0$. On en déduit facilement par récurrence l'inégalité

$$C(n) \leq (2^n - 1)\lambda.$$

La méthode `hanoi` a donc une complexité exponentielle $O(2^n)$.

**Exercice 2.** Quelle est la complexité de la méthode suivante qui calcule le $n$-ième élément de la suite de Fibonacci ?

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

Note : C'est bien entendu une façon naïve de calculer la suite de Fibonacci ; on expliquera plus loin dans ce cours comment faire beaucoup mieux. Solution □

# Deuxième partie

# Structures de données élémentaires

# 3 Tableaux

De manière simple, un tableau n'est rien d'autre qu'une suite de valeurs stockées dans des cases mémoire contiguës. Ainsi on représentera graphiquement le tableau contenant la suite de valeurs entières $3, 7, 42, 1, 4, 8, 12$ de la manière suivante :

| 3 | 7 | 42 | 1 | 4 | 8 | 12 |
|---|---|---|---|---|---|---|

La particularité d'une structure de tableau est que le contenu de la $i$-ième case peut être lu ou modifié *en temps constant*.

En Java, on peut construire un tel tableau en énumérant ses éléments entre accolades, séparés par des virgules :

```
int[] a = {3, 7, 42, 1, 4, 8, 12};
```

Les cases sont numérotées à partir de 0. On accède à la première case avec la notation `a[0]`, à la seconde avec `a[1]`, etc. Si on tente d'accéder en dehors des cases valides du tableau, par exemple en écrivant `a[-1]`, alors on obtient une erreur :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
```

On modifie le contenu d'une case avec la construction d'affectation habituelle. Ainsi pour mettre à 0 la deuxième case du tableau `a`, on écrira

```
a[1] = 0;
```

Si l'indice ne désigne pas une position valide dans le tableau, l'affectation provoque la même exception que pour l'accès.

On peut obtenir la longueur du tableau `a` avec l'expression `a.length`. Ainsi `a.length` vaut 7 sur l'exemple précédent. L'accès à la longueur se fait en temps constant. Un tableau peut avoir la longueur 0.

Il existe d'autres procédés pour construire un tableau que d'énumérer explicitement ses éléments. On peut notamment construire un tableau de taille donnée avec la construction `new` :

```
int[] c = new int[100];
```

Ses éléments sont alors initialisés avec une valeur par défaut (ici 0 car il s'agit d'entiers).

## 3.1 Parcours d'un tableau

Supposons que l'on veuille faire la somme de tous les éléments d'un tableau d'entiers `a`. Un algorithme simple pour cela consiste à initialiser une variable `s` à 0 et à parcourir tous les éléments du tableau pour les ajouter un par un à cette variable. La méthode naturelle pour effectuer ce parcours consiste à utiliser une boucle `for`, pour affecter successivement à une variable `i` les valeurs 0, 1, ..., `a.length` − 1. Ainsi on peut écrire la méthode `sum` de la manière suivante :

```
static int sum(int[] a) {
  int s = 0;
  for (int i = 0; i < a.length; i++)
    s += a[i];
  return s;
}
```

Cependant, on peut faire encore plus simple car la boucle `for` de Java permet de parcourir directement tous les *éléments* du tableau `a` avec la construction `for(int x : a)`. Ainsi le programme se simplifie en

```
  int s = 0;
  for (int x : a)
    s += x;
  return s;
```

Ce programme effectue exactement `a.length` additions, soit une complexité linéaire.

Comme exemple plus complexe, considérons l'évaluation d'un polynôme

$$A(X) = \sum_{0 \le i < n} a_i X^i$$

On suppose que les coefficients du polynôme $A$ sont stockés dans un tableau `a`, le coefficient $a_i$ étant stocké dans `a[i]`. Ainsi le tableau `[1, 2, 3]` représente le polynôme $3X^2+2X+1$. Une méthode simple, mais naïve, consiste à écrire une boucle qui réalise exactement la somme ci-dessus.

```
static double evalPoly(double[] a, double x) {
  double s = 0.0;
  for (int i = 0; i < a.length; i++)
    s += a[i] * Math.pow(x, i);
  return s;
}
```

Une méthode plus efficace pour évaluer un polynôme est d'utiliser la *méthode de Horner*. Elle consiste à réécrire la somme ci-dessus de la manière suivante :

$$A(X) = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-2} + Xa_{n-1}) \dots))$$

Ainsi on évite le calcul des différentes puissances $X^i$, en factorisant intelligemment, et en ne faisant plus que des multiplications par $X$. Pour réaliser ce calcul, il faut parcourir le

tableau de la droite vers la gauche, pour que le traitement de la `i`-ième case de `a` consiste à multiplier par $X$ la somme courante puis à lui ajouter `a[i]`. Si la variable `s` contient la somme courante, la situation est donc la suivante :

$$A(X) = a_0 + X(\cdots(a_i + X(\underbrace{a_{i+1} + \cdots}_{\texttt{s}})))$$

Ainsi la méthode de Horner s'écrit

```
static double horner(double[] a, double x) {
  double s = 0.0;
  for (int i = a.length - 1; i >= 0; i--)
    s = a[i] + x * s;
  return s;
}
```

On constate facilement que ce programme effectue exactement `a.length` additions et autant de multiplications, soit une complexité linéaire.

**Exercice 3.** Écrire une méthode qui prend un tableau d'entiers `a` en argument et renvoie le tableau des sommes cumulées croissantes de `a`, autrement dit un tableau de même taille dont la $k$-ième composante vaut $\sum_{i=0}^{k}$ `a[`$i$`]`. La complexité doit être linéaire. Le tableau fourni en argument ne doit pas être modifié. Solution □

**Exercice 4.** Écrire une méthode qui renvoie un tableau contenant les $n$ premières valeurs de la suite de Fibonacci définie par

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

La complexité doit être linéaire. Solution □

**Exercice 5.** Écrire une méthode `void shuffle(int[] a)` qui mélange aléatoirement les éléments d'un tableau en utilisant l'algorithme suivant appelé « mélange de Knuth » (*Knuth shuffle*), où $n$ est la taille du tableau :

pour $i$ de 1 à $n-1$
  soit $j$ un entier aléatoire entre 0 et $i$ (inclus)
  échanger les éléments d'indices $i$ et $j$

On obtient un entier aléatoire entre 0 et $k-1$ avec `(int)(Math.random() * `$k$`)`. Cet algorithme sera notamment réutilisé plus loin dans l'exercice 62 et dans la section 13.2 concernant le tri rapide. Solution □

## 3.2 Recherche dans un tableau

### 3.2.1 Recherche par balayage

Supposons que l'on veuille déterminer si un tableau `a` contient une certaine valeur `v`. On ne va pas nécessairement examiner *tous* les éléments du tableau, car on souhaite interrompre le parcours dès que l'élément est trouvé. Une solution consiste à utiliser `return` dès que la valeur est trouvée :

```
static boolean contains(int[] a, int v) {
  for (int x : a)
    if (x == v)
      return true;
  return false;
}
```

Dans la section 2.1.4, nous avons montré que la complexité en moyenne de cet algorithme est linéaire.

**Exercice 6.** Écrire une variante de la méthode `contains` qui renvoie l'*indice* où la valeur `v` apparaît dans `a`, le cas échéant. Que faire lorsque `a` ne contient pas `v` ? Solution □

**Exercice 7.** Écrire une méthode qui renvoie l'élément maximal d'un tableau d'entiers. On discutera des diverses solutions pour traiter le cas d'un tableau de longueur 0.
Solution □

### 3.2.2 Recherche dichotomique dans un tableau trié

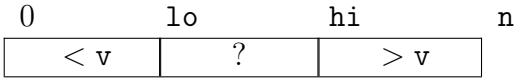
Dans le pire des cas, la recherche par balayage ci-dessus parcourt tout le tableau, et effectue donc $n$ comparaisons, où $n$ est la longueur du tableau. Dans certains cas, cependant, la recherche d'un élément dans un tableau peut être réalisée de manière plus efficace. C'est le cas par exemple lorsque le tableau est trié. On peut alors exploiter l'idée suivante : on coupe le tableau en deux par le milieu et on détermine si la valeur `v` doit être recherchée dans la moitié gauche ou droite. En effet, il suffit pour cela de la comparer avec la valeur centrale. Puis on répète le processus sur la portion sélectionnée.

Supposons par exemple que l'on cherche la valeur `9` dans le tableau `{1,3,5,6,9,12,14}`. La recherche s'effectue ainsi :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| on cherche dans `a[0:7]` | 1 | 3 | 5 | 6 | 9 | 12 | 14 |
| on compare `x`=9 avec `a[3]`=6 | 1 | 3 | 5 | **6** | 9 | 12 | 14 |
| on cherche dans `a[4:7]` | | | | | 9 | 12 | 14 |
| on compare `x`=9 avec `a[5]`=12 | | | | | 9 | **12** | 14 |
| on cherche dans `a[4:4]` | | | | | 9 | | |
| on compare `x`=9 avec `a[4]`=9 | | | | | **9** | | |

On note que seulement trois comparaisons ont été nécessaires pour trouver la valeur. C'est une application du principe *diviser pour régner*.

Pour écrire l'algorithme, on délimite la portion du tableau `a` dans laquelle la valeur `v` doit être recherchée à l'aide de deux indices `lo` et `hi`. On maintient l'*invariant* suivant : les valeurs strictement à gauche de `lo` sont inférieures à `v` et les valeurs à partir de `hi` sont supérieures à `v`, ce qui s'illustre ainsi




On commence par initialiser les variables `lo` et `hi` avec 0 et `a.length`, respectivement.

```
static boolean binarySearch(int[] a, int v) {
  int lo = 0, hi = a.length;
```

Tant que la portion à considérer contient au moins un élément

```
  while (lo < hi) {
```

on calcule l'indice de l'élément central, en faisant la moyenne de `lo` et `hi`. (L'exercice 9 justifie le calcul de la moyenne de cette façon.)

```
    int m = lo + (hi - lo) / 2;
```

Il est important de noter qu'on effectue ici une division *entière*. Comme elle est arrondie vers le bas, on obtiendra toujours une valeur comprise entre `lo` inclus et `hi` exclus. Si l'élément `a[m]` est l'élément recherché, on a terminé la recherche.

```
    if (a[m] == v)
      return true;
```

Sinon, on détermine si la recherche doit être poursuivie à gauche ou à droite. Si `a[m] < v`, on poursuit à droite :

```
    if (a[m] < v)
      lo = m + 1;
```

Sinon, on poursuit à gauche :

```
    else
      hi = m;
  }
```

Si on sort de la boucle `while`, c'est que l'élément ne se trouve pas dans le tableau, car il ne reste que des éléments strictement plus petits (à gauche de `lo`) ou strictement plus grands (à partir de `hi`). On renvoie alors `false` pour signaler l'échec.

```
  return false;
```

Le code complet est donné programme 1 page 38. Note : La bibliothèque standard de Java fournit une telle méthode `binarySearch` dans la classe `java.util.Arrays`.

Montrons maintenant que la complexité de cet algorithme est au pire $O(\log n)$ où $n$ est la longueur du tableau. En particulier, on effectue au pire un nombre logarithmique de comparaisons. La démonstration consiste à établir qu'après $k$ itérations de la boucle, on a l'inégalité

$$\texttt{hi} - \texttt{lo} \leq \frac{n}{2^k}.$$

La démonstration se fait par récurrence sur $k$. Initialement, on a $\texttt{lo} = 0$ et $\texttt{hi} = n$ et $k = 0$, donc l'inégalité est établie. Supposons maintenant l'inégalité vraie au rang $k$ et $\texttt{lo} < \texttt{hi}$. À la fin de la $k+1$-ième itération, on a soit `lo = m+1`, soit `hi = m`. Dans le premier cas, on a donc

$$\texttt{hi} - (\texttt{m}+1) \leq \texttt{hi} - \frac{\texttt{lo}+\texttt{hi}}{2} = \frac{\texttt{hi}-\texttt{lo}}{2} \leq \frac{n}{2^k \times 2} = \frac{n}{2^{k+1}}.$$

**Programme 1 — Recherche dichotomique dans un tableau trié**

```
static boolean binarySearch(int[] a, int v) {
  int lo = 0, hi = a.length; // on cherche dans [lo..hi[
  while (lo < hi) {
    int m = lo + (hi - lo) / 2;
    if (a[m] == v)
      return true;
    if (a[m] < v)
      lo = m + 1;
    else
      hi = m;
  }
  return false;
}
```

Le second cas est laissé au lecteur. On conclut ainsi : pour $k > \log_2(n)$, on a $\texttt{hi} - \texttt{lo} < 1$, c'est-à-dire $\texttt{hi} \leq \texttt{lo}$, et on sort donc de la boucle.

La complexité de la recherche dichotomique est donc $O(\log n)$, alors que celle de la recherche par balayage est $O(n)$. Il faut cependant ne pas oublier qu'elles ne s'appliquent pas dans les mêmes conditions : une recherche dichotomique exige que les données soient triées.

**Exercice 8.** Montrer que la méthode `binarySearch` termine toujours. Solution □

**Exercice 9.** Expliquer pourquoi ce n'est pas une bonne idée de calculer la valeur de `m` tout simplement comme `(lo + hi) / 2`. Solution □

## 3.3 Mode de passage des tableaux

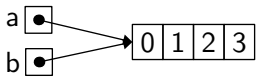
Considérons le programme suivant, où une méthode `f` reçoit un tableau `b` en argument et le modifie

```
static void f(int[] b) {
  b[2] = 42;
}
```

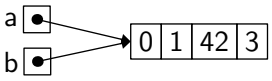
et où le programme principal construit un tableau `a` et le passe à la méthode `f` :

```
int[] a = {0, 1, 2, 3};
f(a);
```

Il est important de comprendre que, pendant l'exécution de la méthode `f`, la variable locale `b` est un alias pour le tableau `a`. Ainsi, à l'entrée de la méthode `f`, on a



et, après l'exécution de l'instruction `b[2] = 42;`, on a




En particulier, après l'appel à la méthode `f`, on a `a[2] == 42`. (Nous avions déjà expliqué cela section 1.2.3 mais il n'est pas inutile de le redire.)

Il est parfois utile d'écrire des méthodes qui modifient le contenu d'un tableau reçu en argument. Un exemple typique est celui d'une méthode qui échange le contenu de deux cases d'un tableau :

```
static void swap(int[] a, int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

Un autre exemple est celui d'une méthode qui trie un tableau ; plusieurs exemples seront donnés dans le chapitre 13.

## 3.4 Tableaux redimensionnables

La taille d'un tableau Java est déterminée à sa création et elle ne peut être modifiée ultérieurement. Il existe cependant de nombreuses situations où le nombre de données manipulées n'est pas connu à l'avance mais où, pour autant, on souhaite les stocker dans un tableau pour un accès en lecture et en écriture en temps constant. Dans cette section, nous présentons une solution simple à ce problème, connue sous le nom de *tableau redimensionnable* (*resizable array* en anglais).

On suppose, pour simplifier, qu'on ne s'intéresse ici qu'à des tableaux contenant des éléments de type `int`. On cherche donc à définir une classe `ResizableArray` fournissant un constructeur

```
ResizableArray(int len)
```

pour construire un nouveau tableau redimensionnable de taille `len`, et (au minimum) les méthodes suivantes :

```
int size()
void setSize(int len)
int get(int i)
void set(int i, int v)
```

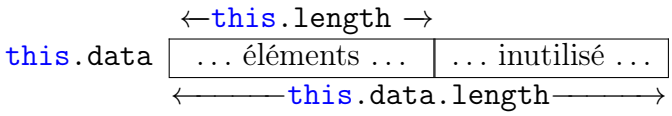
La méthode `size` renvoie la taille du tableau. À la différence d'un tableau usuel, cette taille peut être modifiée *a posteriori* avec la méthode `setSize`. Les méthodes `get` et `set` sont les opérations de lecture et d'écriture dans le tableau. Comme pour un tableau usuel, elles lèveront une exception si on cherche à accéder en dehors des bornes du tableau.

### 3.4.1 Principe

L'idée de la réalisation est très simple : on utilise un tableau usuel pour stocker les éléments et lorsqu'il devient trop petit, on en alloue un plus grand dans lequel on recopie les éléments du premier. Pour éviter de passer notre temps en allocations et en copies, on s'autorise à ce que le tableau de stockage soit trop grand, les éléments au-delà d'un certain indice n'étant plus significatifs. La classe `ResizableArray` est donc ainsi déclarée

```
class ResizableArray {
  private int length;
  private int[] data;
```

où le champ `data` est le tableau de stockage des éléments et `length` le nombre significatif d'éléments dans ce tableau, ce que l'on peut schématiser ainsi :




On maintiendra donc toujours l'invariant suivant :

$$0 \leq \texttt{length} \leq \texttt{data.length}$$

On note le caractère privé des champs `data` et `length`, ce qui nous permettra de maintenir l'invariant ci-dessus.

Pour créer un nouveau tableau redimensionnable de taille `len`, il suffit d'allouer un tableau usuel de cette taille-là dans `data`, sans oublier d'initialiser le champ `length` également :

```
  ResizableArray(int len) {
    this.length = len;
    this.data = new int[len];
  }
```

La taille du tableau redimensionnable est directement donnée par le champ `length`.

```
  int size() {
    return this.length;
  }
```

Pour accéder au `i`-ième élément du tableau redimensionnable, il convient de vérifier la validité de l'accès, car le tableau `this.data` peut contenir plus de `this.length` éléments.

```
  int get(int i) {
    if (i < 0 || i >= this.length)
      throw new ArrayIndexOutOfBoundsException(i);
    return this.data[i];
  }
```

L'affectation est analogue (voir page 42). Toute la subtilité est dans la méthode `setSize` qui redimensionne un tableau pour lui donner une nouvelle taille `len`. Plusieurs cas de figure se présentent. Si la nouvelle taille `len` est inférieure ou égale à la taille de `this.data`, il n'y a rien à faire, si ce n'est mettre le champ `length` à jour. Si en revanche `len` est plus grand la taille de `this.data`, il va falloir remplacer `data` par un tableau plus grand. On commence donc par effectuer ce test :

```
void setSize(int len) {
  int n = this.data.length;
  if (len > n) {
```

On alloue alors un tableau suffisamment grand. On pourrait choisir tout simplement `len` pour sa taille, mais on adopte une stratégie plus subtile consistant à au moins *doubler* la taille de `this.data` (nous la justifierons par la suite) :

```
    int[] a = new int[Math.max(len, 2 * n)];
```

On recopie alors les éléments significatifs de `this.data` vers ce nouveau tableau `a` :

```
    for (int i = 0; i < this.length; i++)
      a[i] = this.data[i];
```

puis on remplace `this.data` par ce nouveau tableau :

```
    this.data = a;
  }
```

L'ancien tableau sera ramassé par le GC. Enfin, on met à jour la valeur de `this.length`, quel que soit le résultat du test `len > n`

```
  this.length = len;
}
```

ce qui conclut le code de `setSize`. L'intégralité du code est donnée programme 2 page 42.

**Exercice 10.** Il peut être souhaitable de rediminuer parfois la taille du tableau, par exemple si elle devient grande par rapport au nombre d'éléments effectifs et que le tableau occupe beaucoup de mémoire. Modifier la méthode `setSize` pour qu'elle divise par deux la taille du tableau lorsque le nombre d'éléments devient inférieur au quart de la taille du tableau. Solution □

**Exercice 11.** Ajouter une méthode `int[] toArray()` qui renvoie un tableau usuel contenant les éléments du tableau redimensionnable.

Solution □

### 3.4.2 Application 1 : Lecture d'un fichier

On souhaite lire une liste d'entiers contenus dans un fichier, sous la forme d'un entier par ligne, et les stocker dans un tableau. On ne connaît pas le nombre de lignes du fichier. Bien entendu, on pourrait commencer par compter le nombre de lignes, pour allouer ensuite un tableau de cette taille-là avant de lire les lignes du fichier. Mais on peut plus simplement encore utiliser un tableau redimensionnable. En supposant que le fichier s'appelle `numbers.txt`, on commence par l'ouvrir de la manière suivante :

```
BufferedReader f = new BufferedReader(new FileReader("numbers.txt"));
```

Puis on alloue un nouveau tableau redimensionnable destiné à recevoir les entiers que l'on va lire. Initialement, il ne contient aucun élément :

**Programme 2 — Tableaux redimensionnables**

```
class ResizableArray {

  private int length; // nb d'éléments significatifs
  private int[] data; // invariant: 0 <= length <= data.length

  ResizableArray(int len) {
    this.length = len;
    this.data = new int[len];
  }

  int size() {
    return this.length;
  }

  int get(int i) {
    if (i < 0 || i >= this.length)
      throw new ArrayIndexOutOfBoundsException(i);
    return this.data[i];
  }

  void set(int i, int v) {
    if (i < 0 || i >= this.length)
      throw new ArrayIndexOutOfBoundsException(i);
    this.data[i] = v;
  }

  void setSize(int len) {
    int n = this.data.length;
    if (len > n) {
      int[] a = new int[Math.max(len, 2 * n)];
      for (int i = 0; i < this.length; i++)
        a[i] = this.data[i];
      this.data = a;
    }
    this.length = len;
  }
}
```

```
ResizableArray r = new ResizableArray(0);
```

On écrit ensuite une boucle dans laquelle on lit chaque ligne du fichier avec la méthode `readLine` du `BufferedReader` :

```
while (true) {
  String s = f.readLine();
```

La valeur `null` signale la fin du fichier, auquel cas on sort de la boucle avec `break` :

```
  if (s == null) break;
```

Dans le cas contraire, on étend la taille du tableau redimensionnable d'une unité, et on stocke l'entier lu dans la dernière case du tableau :

```
  int len = r.size();
  r.setSize(len + 1);
  r.set(len, Integer.parseInt(s));
}
```

Ceci conclut la lecture du fichier. Pour être complet, il faudrait aussi gérer les exceptions possiblement levées par `new FileReader` d'une part et par `f.readLine()` d'autre part, soit en les rattrapant (avec `try catch`), soit en les déclarant avec `throws`.

**Exercice 12.** Ajouter une méthode `void append(int v)` à la classe `ResizableArray` qui augmente la taille du tableau d'une unité et stocke la valeur `v` dans sa dernière case. Simplifier le programme ci-dessus en utilisant cette nouvelle méthode. Solution □

**Complexité.** Dans le programme ci-dessus, nous avons démarré avec un tableau redimensionnable de taille 0 et nous avons augmenté sa taille d'une unité à chaque lecture d'une nouvelle ligne. Si la méthode `setSize` avait effectué systématiquement une allocation d'un nouveau tableau et une recopie des éléments dans ce tableau, la complexité aurait été quadratique, puisque la lecture de la $i$-ième ligne du fichier aurait alors eu un coût $i$, d'où un coût total

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

pour un fichier de $n$ lignes. Cependant, la stratégie de `setSize` est plus subtile, car elle consiste à doubler (au minimum) la taille du tableau lorsqu'il doit être agrandi. Montrons que le coût total est alors linéaire. Supposons, sans perte de généralité, que $n \geq 2$ et posons $k = \lfloor \log_2(n) \rfloor$ c'est-à-dire $2^k \leq n < 2^{k+1}$. Au total, on aura effectué $k+2$ redimensionnements pour arriver à un tableau `data` de taille finale $2^{k+1}$. Après le $i$-ième redimensionnement, pour $i = 0, \ldots, k+1$, le tableau a une taille $2^i$ et le $i$-ième redimensionnement a donc coûté $2^i$. Le coût total est alors

$$\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1 = O(n).$$

Autrement dit, certaines opérations `setSize` ont un coût constant (lorsque le redimensionnement n'est pas nécessaire) et d'autres au contraire un coût non constant, mais

la complexité totale reste linéaire. Ramené à l'ensemble des $n$ opérations, tout se passe comme si chaque opération d'ajout d'un élément avait eu un coût constant. On parle de *complexité amortie* pour désigner la complexité d'une opération en moyenne sur un ensemble de $n$ opérations. Dans le cas présent, on peut donc dire que l'extension d'un tableau redimensionnable d'une unité a une complexité amortie $O(1)$.

### 3.4.3 Application 2 : Concaténation de chaînes

Donnons un autre exemple de l'utilité du tableau redimensionnable. Supposons que l'on souhaite construire une chaîne de caractères énumérant tous les entiers de 0 à $n$ sous la forme `"0, 1, 2, 3, ..., n"`. Si on écrit naïvement

```
String s = "0";
for (int i = 1; i <= n; i++)
  s += ", " + i;
```

alors la complexité est quadratique, car chaque concaténation de chaînes, pour construire le résultat de `s + ", " + i`, a un coût proportionnel à la longueur de `s`, c'est-à-dire proportionnel à `i`. Là encore, on peut avantageusement exploiter le principe du tableau redimensionnable pour construire la chaîne `s`. Il suffit d'adapter le code de `ResizableArray` pour des caractères plutôt que des entiers (ou encore en faire une classe générique — voir section 3.4.5 plus loin).

Plus simplement encore, la bibliothèque Java fournit une classe `StringBuffer` pour construire des chaînes de caractères incrémentalement sur le principe du tableau redimensionnable. Une méthode `append` permet de concaténer une chaîne à la fin d'un `StringBuffer`, d'où le code

```
StringBuffer buf = new StringBuffer("0");
for (int i = 1; i <= n; i++)
  buf.append(", " + i);
```

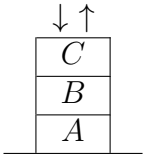
La chaîne finale peut être récupérée avec `buf.toString()`. Cette fois, la complexité est bien linéaire, par un raisonnement identique au précédent.

**Exercice 13.** Ajouter une méthode `String toString()` à la classe `ResizableArray`, qui renvoie le contenu d'un tableau redimensionnable sous la forme d'une chaîne de caractères telle que `"[3, 7, 2]"`. On se servira d'un `StringBuffer` pour construire cette chaîne.

Solution □

### 3.4.4 Application 3 : Structure de pile

On va mettre à profit la notion de tableau redimensionnable pour construire une *structure de pile*. Elle correspond exactement à l'image traditionnelle d'une pile de cartes ou d'assiettes posée sur une table. En particulier, on ne peut accéder qu'au dernier élément ajouté, qu'on appelle le *sommet* de la pile. Ainsi, si on a ajouté successivement $A$, puis $B$, puis $C$ dans une pile, on se retrouve dans la situation suivante



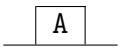
où $C$ est empilé sur $B$, lui-même empilé sur $A$. On peut soit retirer $C$ de la pile (on dit qu'on « dépile » $C$), soit ajouter un quatrième élément $D$ (on dit qu'on « empile » $D$). Si on veut accéder à l'élément $A$, il faut commencer par dépiler $C$ puis $B$. L'image associée à une pile est donc « dernier arrivé, premier sorti » (en anglais, on parle de LIFO pour *last in, first out*).

Nous allons écrire la structure de pile dans une classe `Stack`, en fournissant les opérations suivantes :

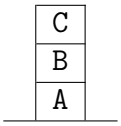
— le constructeur `Stack()` renvoie une nouvelle pile, initialement vide ;
— la méthode `pop()` dépile et renvoie le sommet de la pile ;
— la méthode `push(v)` empile la valeur `v`.
— la méthode `size()` renvoie le nombre d'éléments contenus dans la pile ;
— la méthode `isEmpty()` indique si la pile est vide ;
— la méthode `top()` renvoie le sommet de la pile, sans la modifier.

Seules les opérations `push` et `pop` modifient le contenu de la pile. Voici une illustration de l'utilisation de cette structure :

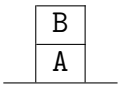
```
Stack s = new Stack();
s.push(A);
```




```
s.push(B);
s.push(C);
```




```
int x = s.pop();
// x vaut C
```




Comme on l'aura deviné, le tableau redimensionnable nous fournit exactement ce dont nous avons besoin pour réaliser une pile. En effet, il suffit de stocker le premier élément empilé à l'indice 0, puis le suivant à l'indice 1, etc. Le sommet de pile se situe donc à l'indice $n-1$ où $n$ est la taille du tableau redimensionnable. Pour empiler un élément, il suffit d'augmenter la taille du tableau d'une unité. Pour dépiler, il suffit de récupérer le dernier élément du tableau puis de diminuer sa taille d'une unité.

Pour bien faire les choses, cependant, il convient d'*encapsuler* le tableau redimensionnable dans la classe `Stack`, de manière à garantir la bonne abstraction de pile. Pour cela, il suffit d'en faire un champ privé

```
class Stack {
  private ResizableArray elts;
  ...
}
```

Ainsi, seules les méthodes fournies pourront en modifier le contenu. Le code complet est donné programme 3 page 46.

**Exercice 14.** Écrire une méthode `swap` qui échange les deux éléments au sommet d'une pile, d'abord à l'extérieur de la classe `Stack`, puis comme une nouvelle méthode de la classe `Stack`. Solution □

**Programme 3 — Structure de pile**

(réalisée à l'aide d'un tableau redimensionnable)

```
class Stack {

  private ResizableArray elts;

  Stack() {
    this.elts = new ResizableArray(0);
  }

  boolean isEmpty() {
    return this.elts.size() == 0;
  }

  int size() {
    return this.elts.size();
  }

  void push(int x) {
    int n = this.elts.size();
    this.elts.setSize(n + 1);
    this.elts.set(n, x);
  }

  int pop() {
    int n = this.elts.size();
    if (n == 0)
      throw new NoSuchElementException();
    int e = this.elts.get(n - 1);
    this.elts.setSize(n - 1);
    return e;
  }

  int top() {
    int n = this.elts.size();
    if (n == 0)
      throw new NoSuchElementException();
    return this.elts.get(n - 1);
  }

}
```

### 3.4.5 Code générique

Pour réaliser une version générique de la classe `ResizableArray`, on la paramètre par le type `T` des éléments.

```
class ResizableArray<T> {
  private int length;
  private T[] data;
```

Le code reste essentiellement le même, au remplacement du type `int` par le type `T` aux endroits opportuns. Il y a cependant deux subtilités. La première concerne la création d'un tableau de type `T[]`. Naturellement, on aimerait écrire dans le constructeur

```
  this.data = new T[len];
```

mais l'expression `new T[len]` est refusée par le compilateur, avec le message d'erreur

```
  Cannot create a generic array of T
```

C'est là une limitation du système de types de Java. Il existe plusieurs façons de contourner ce problème. Ici, on peut se contenter de créer un tableau de `Object` puis de forcer le type avec un transtypage (*cast* en anglais) :

```
  this.data = (T[])new Object[len];
```

Le compilateur émet un avertissement (`Unchecked cast from Object[] to T[]`) mais il n'y a pas de risque ici car il s'agit d'un champ privé que nous ne pourrons jamais confondre avec un tableau d'un autre type.

La seconde subtilité concerne la méthode `setSize`, dans le cas où la taille du tableau est diminuée. Jusqu'à présent, on ne faisait rien, si ce n'est mettre à jour le champ `length`. Mais dans la version générique, il faut prendre soin de ne pas conserver à tort des pointeurs vers des objets qui pourraient être récupérés par le GC car inutilisés par ailleurs. Il faut donc « effacer » les éléments à partir de l'indice `length`. On le fait en leur affectant la valeur `null`.

```
  void setSize(int len) {
    int n = this.data.length;
    if (len > n) {
      ... même code qu'auparavant ...
    } else {
      for (int i = len; i < n; i++)
        this.data[i] = null;
    }
    this.length = len;
  }
```

On maintient donc l'invariant suivant :

$$\forall i.\ \texttt{this.length} \leq i < \texttt{this.data.length} \Rightarrow \texttt{this.data}[i] = \texttt{null}$$

Dans la version non générique, nous n'avions pas ce problème, car les éléments étaient des entiers, et non des pointeurs.

La bibliothèque standard Java fournit des tableaux redimensionnables génériques, dans la classe `java.util.Vector<E>`, où `E` est le type des éléments. Ils sont tout à fait semblables à ceux que nous avons construits, avec notamment des méthodes `get`, `set` et `setSize`. Attention : `new Vector(n)` renvoie un tableau redimensionnable dont la capacité, c'est-à-dire la taille du tableau interne, est `n` mais dont le nombre d'éléments est 0. Si on veut un tableau redimensionnable de taille `n`, il faut d'abord le construire puis utiliser sa méthode `setSize(n)`.

La bibliothèque Java fournit également une structure générique de pile, dans la classe `java.util.Stack<E>`, dont la réalisation s'appuie sur la classe `Vector`. C'est tout à fait analogue à ce que nous avons fait dans la section 3.4.4.

La bibliothèque standard Java fournit également des tableaux redimensionnables dans une autre classe, à savoir `java.util.ArrayList<E>`. En première approximation, cette classe est tout à fait semblable à la classe `Vector`. La différence se situe d'une part dans la stratégie interne de redimensionnement (dans les deux cas, une extension d'un élément se fait en temps amorti $O(1)$) et d'autre part dans l'utilisation dans du code concurrent, ce qui dépasse le cadre de ce cours. En conséquence, on peut utiliser `Vector` et `ArrayList` de façon interchangeable.

# 4 Listes chaînées

Ce chapitre introduit une structure de donnée fondamentale en informatique, la *liste chaînée*. Il s'agit d'une structure *dynamique* au sens où, à la différence du tableau, elle est généralement allouée petit à petit, au fur et à mesure des besoins.

## 4.1 Listes simplement chaînées

Le principe d'une liste chaînée est le suivant : chaque élément de la liste est représenté par un objet et contient d'une part la valeur de cet élément et d'autre part un pointeur vers l'élément suivant de la liste. Si on suppose que les éléments sont ici des entiers, la déclaration d'une classe `Singly` pour représenter une telle liste chaînée est donc aussi simple que

```
class Singly {
  int element;
  Singly next;
}
```

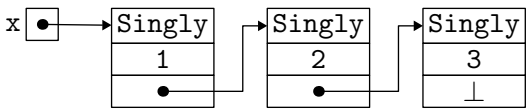
La valeur `null` nous sert à représenter la liste vide *i.e.* la liste ne contenant aucun élément. Le constructeur naturel de cette classe prend en arguments les valeurs des deux champs :

```
  Singly(int element, Singly next) {
    this.element = element;
    this.next = next;
  }
```

Ainsi, on peut construire une liste contenant les trois entiers `1`, `2` et `3`, stockée dans une variable `x`, de la façon suivante :

```
  Singly x = new Singly(1, new Singly(2, new Singly(3, null)));
```

On a donc construit en mémoire une structure qui a la forme suivante, où chaque élément de la liste est un bloc sur le tas :



Le bloc contenant la valeur 3 a été construit en premier, puis celui contenant la valeur 2, puis enfin celui contenant la valeur 1. Ce dernier est appelé la *tête* de la liste. Dans le cas très particulier où `x` est la liste vide, on écrit tout simplement

```
Singly x = null;
```

ce qui correspond à une situation où *aucun* objet n'a été alloué en mémoire :


Bien que de type `Singly`, une telle valeur n'en est pas pour autant un *objet* de la classe `Singly`, avec un champ `element` et un champ `next`. Par conséquent, il faudra pendre soin dans la suite de toujours bien traiter le cas de la liste vide, de manière à éviter l'exception `NullPointerException`. Dès la section suivante, nous verrons comment apporter une solution élégante à ce problème.

## Parcours d'une liste

La nature même d'une liste chaînée nous permet d'en parcourir les éléments très facilement. En effet, si la variable `x` désigne un certain élément de la liste, « passer » à l'élément suivant consiste en la simple affectation `x = x.next`. Le parcours s'arrête lorsqu'on atteint la valeur `null`. Le schéma d'une boucle parcourant tous les éléments d'une liste est donc le suivant :

```
while (x != null) {
  ...
  x = x.next;
}
```

Comme premier exemple, considérons une méthode statique `contains` qui détermine si un entier `x` apparaît dans une liste `s` donnée. On parcourt la liste `s` pour comparer successivement la valeur `x` avec tous les éléments de la liste.

```
static boolean contains(Singly s, int x) {
  while (s != null) {
```

Si l'élément courant est égal à `x`, on renvoie immédiatement `true`, ce qui achève la méthode `contains`.

```
    if (s.element == x) return true;
```

Sinon, on passe à l'élément suivant :

```
    s = s.next;
  }
```

Il est important de bien comprendre que cette affectation ne modifie pas la liste mais seulement la variable `s`, qui est locale à la méthode `contains`. Si on finit par sortir de la boucle, c'est que `x` n'apparaît pas dans la liste et on renvoie alors `false` :

```
  return false;
}
```

Il est important de noter que ce code fonctionne correctement sur une liste vide, c'est-à-dire lorsque `s` vaut `null`. En effet, on sort immédiatement de la boucle et on renvoie `false`, ce qui est le résultat attendu. Le code de la méthode `contains` est donné programme 4 page 51.

**Programme 4 — Listes simplement chaînées**

```
class Singly {
  int element;
  Singly next;

  Singly(int element, Singly next) {
    this.element = element;
    this.next = next;
  }

  static boolean contains(Singly s, int x) {
    while (s != null) {
      if (s.element == x) return true;
      s = s.next;
    }
    return false;
  }
}
```

**Complexité.** Quelle est la complexité de la méthode `contains` ? Supposons que la liste contient $n$ éléments. Une recherche infructueuse implique un parcours total de la liste, de coût $n$. Si en revanche la valeur `x` apparaît dans la liste, avec une première occurrence à la position $i$, alors on aura effectué exactement $i$ tours de boucle. Si on suppose que `x` apparaît avec équiprobabilité à toutes les positions possibles dans la liste, le coût moyen de sa recherche est donc

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

ce qui est également linéaire. La structure de liste chaînée n'est donc pas particulièrement adaptée à la recherche d'un élément ; elle a d'autres applications, qui sont décrites dans les prochaines sections.

**Exercice 15.** Écrire une méthode statique `int length(Singly s)` qui renvoie la longueur de la liste `s`.

Solution □

**Exercice 16.** Écrire une méthode statique `int get(Singly s, int i)` qui renvoie l'élément d'indice `i` de la liste `s`, l'élément de tête étant considéré d'indice 0. Lever une exception si `i` ne désigne pas un indice valide (par exemple `IllegalArgumentException`).

Solution □

## Affichage

À titre de deuxième exemple, écrivons une méthode statique `listToString` qui convertit une liste chaînée en une chaîne de caractères de la forme `"[1 -> 2 -> 3]"`. Comme nous l'avons expliqué plus haut (section 3.4.3), la façon efficace de construire une telle

chaîne est d'utiliser un `StringBuffer`. On commence donc par allouer un tel objet, avec une chaîne réduite à `"["` :

```
static String listToString(Singly s) {
  StringBuffer sb = new StringBuffer("[");
```

Puis on réalise le parcours de la liste, ainsi qu'expliqué ci-dessus. Pour chaque élément, on ajoute sa valeur, c'est-à-dire l'entier `s.element`, au `StringBuffer`, puis la chaîne `" -> "` s'il ne s'agit pas du dernier élément de la liste, c'est-à-dire si `s.next` n'est pas `null`.

```
  while (s != null) {
    sb.append(s.element);
    if (s.next != null) sb.append(" -> ");
    s = s.next;
  }
```

Une fois sorti de la boucle, on ajoute le crochet fermant et on renvoie la chaîne contenue dans le `StringBuffer`.

```
  return sb.append("]").toString();
}
```

Là encore, ce code fonctionne correctement sur une liste vide, renvoyant la chaîne `"[]"`.

**Exercice 17.** Quelle est la complexité de la méthode `listToString` ? (Il faut éventuellement relire ce qui avait été expliqué dans la section 3.4.3.)

Solution □

## Tirage aléatoire d'un élément

Comme troisième exemple de parcours d'une liste, considérons le problème suivant : étant donnée une liste non vide, renvoyer l'un de ses éléments aléatoirement, avec équiprobabilité. Bien entendu, on pourrait commencer par calculer la longueur $n$ de la liste (exercice 15), puis tirer un entier $i$ aléatoirement entre 0 inclus et $n$ exclus, et enfin accéder à l'élément d'indice $i$ de la liste (exercice 16). Cependant, on va s'imposer ici de n'effectuer qu'*un seul* parcours de la liste. Ce n'est pas forcément absurde ; on peut imaginer une situation où les éléments ne peuvent être tous stockés en mémoire car trop nombreux, par exemple.

L'idée est la suivante. On parcourt la liste en maintenant un élément « candidat » à la victoire dans le tirage. À l'examen du $i$-ième élément de la liste, on choisit alors de remplacer ce candidat par l'élément courant avec probabilité $\frac{1}{i}$. Une fois arrivé à la fin de la liste, on renvoie la valeur finale du candidat. Écrivons une méthode statique `randomElement` qui réalise ce tirage. On commence par évacuer le cas d'une liste vide, pour lequel le tirage ne peut être effectué :

```
static int randomElement(Singly s) {
  if (s == null) throw new IllegalArgumentException();
```

Sinon, on initialise notre candidat avec une valeur arbitraire (ici 0) et on conserve l'indice de l'élément courant de la liste dans une autre variable `index`.

**Programme 5 — Tirage aléatoire dans une liste**

```
static int randomElement(Singly s) {
  if (s == null) throw new IllegalArgumentException();
  int candidate = 0, index = 1;
  while (s != null) {
    if ((int)(index * Math.random()) == 0) candidate = s.element;
    index++;
    s = s.next;
  }
  return candidate;
}
```

```
  int candidate = 0, index = 1;
```

Puis on réalise le parcours de la liste. On remplace `candidate` par l'élément courant avec probabilité 1/`index`.

```
  while (s != null) {
    if ((int)(index * Math.random()) == 0) candidate = s.element;
```

Pour cela, on tire un entier aléatoirement entre 0 inclus et `index` exclus et on le compare à 0. Pour cela on utilise `Math.random()`, qui renvoie un flottant entre 0 inclus et 1 exclus, et on multiplie le résultat par `index`, ce qui donne un *flottant* entre 0 inclus et `index` exclus. Sa conversion en entier, avec `(int)(...)`, en fait bien un *entier* entre 0 inclus et `index` exclus. On passe ensuite à l'élément suivant, sans oublier d'incrémenter `index`.

```
    index++;
    s = s.next;
  }
```

Une fois sorti de la boucle, on renvoie la valeur de `candidate`.

```
  return candidate;
}
```

On note qu'au tout premier tour de boucle — qui existe car la liste est non vide — l'élément de la liste est nécessairement sélectionné car `Math.random()` $< 1$ et donc `(int)(1 * Math.random())` $= 0$. La valeur arbitraire que nous avions utilisée pour initialiser la variable `candidate` ne sera donc jamais renvoyée. On en déduit également que le programme fonctionne correctement sur une liste réduite à un élément. Le code complet est donné programme 5 page 53.

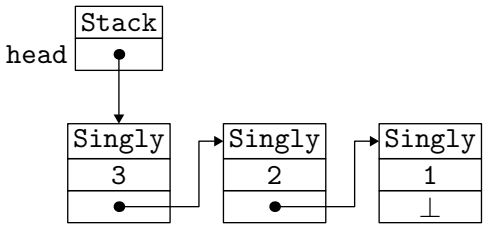
**Exercice 18.** Montrer que, si la liste contient $n$ éléments avec $n \geq 1$, chaque élément est choisi avec probabilité $\frac{1}{n}$. Solution □

## 4.2 Application 1 : Structure de pile

Une application immédiate de la liste simplement chaînée est la structure de pile, que nous avons déjà introduite dans la section 3.4.4. En effet, il suffit de voir la tête de la liste comme le sommet de la pile, et les opérations `push` et `pop` se font alors en temps constant. Exactement comme nous l'avions fait avec le tableau redimensionnable dans la section 3.4.4, on va *encapsuler* la liste chaînée dans une classe `Stack`, de manière à garantir la bonne abstraction de pile. On en fait donc un champ privé

```
class Stack {
  private Singly head;
  ...
}
```

Le code complet de la structure de pile est donné programme 6 page 55. Si on construit une pile avec `s = new Stack()`, dans laquelle on ajoute successivement les entiers 1, 2 et 3, dans cet ordre, avec `s.push(1); s.push(2); s.push(3)`, alors on se retrouve dans la situation suivante :




**Exercice 19.** Ajouter un champ privé `int size` à la classe `Stack`, contenant le nombre d'éléments de la pile, et une méthode `int size()` pour renvoyer sa valeur. Expliquer pourquoi le champ `size` doit être privé. Solution □

**Exercice 20.** Écrire une méthode dynamique publique `String toString()` pour la classe `Stack`, qui renvoie le contenu d'une pile sous la forme d'une chaîne de caractères telle que `"[1, 2, 3]"` où `1` est le sommet de la pile. On pourra s'inspirer de la méthode `listToString` donnée plus haut. Solution □

On pourra également reprendre les exercices de la section 3.4.4 consistant à utiliser la structure de pile.

## 4.3 Application 2 : Structure de file

Une autre application de la liste simplement chaînée, légèrement plus subtile, est la structure de *file*. Il s'agit d'une structure offrant la même interface qu'une pile, à savoir deux opérations principales `push` et `pop`, mais où les éléments sont renvoyés par `pop` dans l'ordre où ils ont été insérés avec `push`, c'est-à-dire suivant une logique « premier arrivé, premier sorti » (en anglais, on parle de FIFO pour *first in, first out*). Dit autrement, il s'agit ni plus ni moins de la file d'attente à la boulangerie.

On peut réaliser une file avec une liste simplement chaînée de la manière suivante : les éléments sont insérés par `push` au niveau du dernier élément de la liste et retirés par

**Programme 6 — Structure de pile**

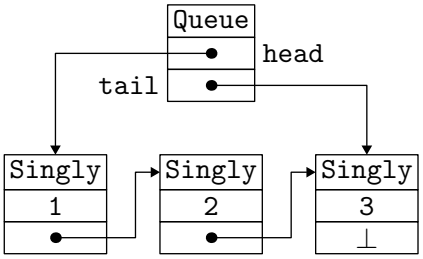
(réalisée à l'aide d'une liste simplement chaînée)

```
class Stack {

  private Singly head;

  Stack() {
    this.head = null;
  }

  boolean isEmpty() {
    return this.head == null;
  }

  void push(int x) {
    this.head = new Singly(x, this.head);
  }

  int top() {
    if (this.head == null)
      throw new NoSuchElementException();
    return this.head.element;
  }

  int pop() {
    if (this.head == null)
      throw new NoSuchElementException();
    int e = this.head.element;
    this.head = this.head.next;
    return e;
  }

}
```

`pop` au niveau du premier élément de la liste. Il faut donc conserver un pointeur sur le dernier élément de la liste. Tout comme dans la section précédente, on va encapsuler la liste chaînée dans une classe `Queue`. Cette fois, il y a deux champs privés, `head` et `tail`, pointant respectivement sur le premier et le dernier élément de la liste.

```
class Queue {
  private Singly head, tail;
  ...
}
```

Ainsi, si on construit une file avec `q = new Queue()`, dans laquelle on ajoute successivement les entiers 1, 2 et 3, dans cet ordre, avec `q.push(1); q.push(2); q.push(3)`, alors on se retrouve dans la situation suivante




où les insertions se font à droite et les retraits à gauche. Les éléments apparaissent donc chaînés « dans le mauvais sens ». Le code est plus subtil que pour une pile et mérite qu'on s'y attarde. Le constructeur se contente d'initialiser les deux champs à `null` :

```
  Queue() {
    this.head = this.tail = null;
  }
```

De manière générale, nous allons maintenir l'invariant que `this.head` vaut `null` si et seulement `this.tail` vaut `null`. En particulier, la file est vide si et seulement `this.head` vaut `null` :

```
boolean isEmpty() {
  return this.head == null;
}
```

Considérons maintenant l'insertion d'un nouvel élément, c'est-à-dire la méthode `push`. On commence par allouer un nouvel élément de liste `e`, qui va devenir le dernier élément de la liste chaînée.

```
  void push(int x) {
    Singly e = new Singly(x, null);
```

Il faut alors distinguer deux cas, selon que la file est vide ou non. Si elle est vide, alors `this.head` et `this.tail` pointent désormais tous les deux sur cet unique élément de liste :

```
    if (this.head == null)
      this.head = this.tail = e;
```

Dans le cas contraire, on ajoute `e` à la fin de la liste existante, dont le dernier élément est pointé par `this.tail`, sans oublier de mettre ensuite à jour le pointeur `this.tail` :

```
else {
  this.tail.next = e;
  this.tail = e;
}
```

Ceci conclut le code de `push`. Pour le retrait d'un élément, on procède à l'autre extrémité de la liste, c'est-à-dire du côté de `this.head`. On commence par évacuer le cas d'une liste vide :

```
int pop() {
  if (this.head == null)
    throw new NoSuchElementException();
```

Si en revanche la liste n'est pas vide, on peut accéder à son premier élément, qui nous donne la valeur à renvoyer :

```
int e = this.head.element;
```

Avant de la renvoyer, il faut supprimer le premier élément de la liste, ce qui est aussi simple que

```
this.head = this.head.next;
```

Cependant, pour maintenir notre invariant sur `this.head` et `this.tail`, on va mettre `this.tail` à `null` si `this.head` est devenu `null` :

```
if (this.head == null) this.tail = null;
```

Cette ligne de code n'est pas nécessaire pour la correction de notre structure de file. En effet, notre méthode `push` teste la valeur de `this.head` et non celle de `this.tail`. Cependant, mettre `this.tail` à `null` permet au GC de Java de récupérer la cellule de liste devenue maintenant inutile. Enfin, il n'y a plus qu'à renvoyer la valeur `e` :

```
return e;
```

Le code complet de la structure de file est donné programme 7 page 58.

**Exercice 21.** Ajouter un champ privé `int size` à la classe `Queue`, contenant le nombre d'éléments de la pile, et une méthode `int size()` pour renvoyer sa valeur. Expliquer pourquoi le champ `size` doit être privé. Solution □

**Exercice 22.** Une autre de réaliser une file, simple et efficace, consiste à utiliser *deux piles*. La première pile est utilisée pour ajouter des éléments à la file et la seconde pile est utilisée pour retirer des éléments de la file. Lorsque l'on souhaite retirer un élément et que la seconde file est vide, on y déverse tous les éléments de la première file. Implémenter cette idée une classe `Queue`. Discuter de l'efficacité de cette solution, selon que les deux piles sont réalisées avec des listes (section 4.2) ou avec des tableaux redimensionnables (section 3.4.4). Solution □

**Programme 7 — Structure de file**

(réalisée à l'aide d'une liste simplement chaînée)

```
class Queue {

  private Singly head, tail;

  Queue() {
    this.head = this.tail = null;
  }

  boolean isEmpty() {
    return this.head == null;
  }

  void push(int x) {
    Singly e = new Singly(x, null);
    if (this.head == null)
      this.head = this.tail = e;
    else {
      this.tail.next = e;
      this.tail = e;
    }
  }

  int peek() {
    if (this.head == null)
      throw new NoSuchElementException();
    return this.head.element;
  }

  int pop() {
    if (this.head == null)
      throw new NoSuchElementException();
    int e = this.head.element;
    this.head = this.head.next;
    if (this.head == null) this.tail = null;
    return e;
  }
}
```

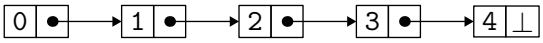
## 4.4 Listes cycliques

Bien que nous n'en ayons pas fait usage pour l'instant, on note que les listes chaînées peuvent être modifiées *a posteriori*. En effet, rien ne nous interdit de modifier la valeur du champ `element` ou du champ `next` d'un objet de la classe `Singly`. Si on modifie le champ `element`, on modifie le *contenu*. On peut ainsi modifier en place la liste $1 \to 2 \to 3$ pour en faire la liste $1 \to 4 \to 3$. Si on modifie le champ `next`, on modifie la *structure* de la liste. On peut ainsi ajouter un quatrième élément *à la fin* de la liste $1 \to 2 \to 3$ pour en faire la liste $1 \to 2 \to 3 \to 4$ en allant modifier le champ `next` de l'élément 3 pour le faire pointer faire un nouvel élément de liste.

En particulier, on peut modifier la structure d'une liste pour en faire une *liste cyclique*. Considérons par exemple le code suivant

```
Singly s4 = new Singly(4, null);
Singly s2 = new Singly(2, new Singly (3, s4));
Singly s0 = new Singly(0, new Singly (1, s2));
```

qui construit la liste à 5 éléments suivante :

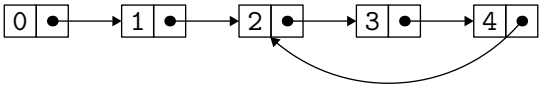



(4.1)

Si on modifie le champ `next` de son dernier élément `s4` pour qu'il pointe désormais sur l'élément `s2`, c'est-à-dire

```
s4.next = s2;
```

alors on se retrouve dans la situation suivante :




(4.2)

Cette liste ne contient plus aucun pointeur `null`, mais seulement des pointeurs vers d'autres éléments de la liste. D'une manière générale, on peut montrer que toute liste simplement chaînée est soit de la forme (4.1), c'est-à-dire une liste linéaire se terminant par `null`, soit de la forme (4.2), c'est-à-dire une « poêle à frire » avec un manche de longueur finie $\mu \geq 0$ et une boucle de longueur finie $\lambda \geq 1$ (dans l'exemple ci-dessus on a $\mu = 2$ et $\lambda = 3$).

Il est important de comprendre que les programmes que nous avons écrits plus haut, qui sont construits autour d'un parcours de liste, ne fonctionnent plus sur une liste cyclique, car ils ne terminent plus dans certains cas. En effet, le critère d'arrêt `s == null` ne sera jamais vérifié. Si on voulait les adapter pour qu'ils fonctionnent également sur des listes cycliques, il faudrait être à même de détecter la présence d'un cycle. Si on y réfléchit un instant, on comprend que le problème n'est pas trivial.

On présente ici un algorithme de détection de cycle, dû à Floyd, et connu sous le nom d'algorithme du lièvre et de la tortue. Comme son nom le suggère, il consiste à parcourir la liste à deux vitesses différentes : la tortue parcourt la liste à la vitesse 1 et le lièvre parcourt la même liste à la vitesse 2. Si à un quelconque moment, le lièvre atteint la fin de la liste, elle est déclarée sans cycle. Et si à un quelconque moment, le lièvre et la tortue se retrouvent à la même position, c'est que la liste contient un cycle. Le code est donné programme 8 page 60. La seule difficulté dans ce code consiste à correctement traiter les

**Programme 8 — Algorithme de détection de cycle de Floyd**

dit « algorithme du lièvre et de la tortue »

```
static boolean hasCycle(Singly s) {
  if (s == null) return false;
  Singly tortoise = s, hare = s.next;
  while (tortoise != hare) {
    if (hare == null) return false;
    hare = hare.next;
    if (hare == null) return false;
    hare = hare.next;
    tortoise = tortoise.next;
  }
  return true;
}
```

différents cas où le lièvre (la variable `hare`) peut atteindre la fin de la liste, afin d'éviter un `NullPointerException` dans le calcul de `hare.next`.

Toute la subtilité de cet algorithme réside dans la preuve de sa terminaison. Si la liste est non cyclique, alors il est clair que le lièvre finira par atteindre la valeur `null` et que la méthode renverra alors `false`. Dans le cas où la liste est cyclique, la preuve est plus délicate. Tant que la tortue est à l'extérieur du cycle, elle s'en approche à chaque étape de l'algorithme, ce qui assure la terminaison de cette première phase (en au plus $\mu$ étapes avec la notation ci-dessus). Et une fois la tortue présente dans le cycle, on note qu'elle ne peut être dépassée par le lièvre. Ainsi la distance qui les sépare diminue à chaque étape de l'algorithme, ce qui assure la terminaison de cette seconde phase (en au plus $\lambda$ étapes). Incidemment, on a montré que la complexité de cet algorithme est toujours au plus $n$ où $n$ est le nombre d'éléments de la liste. Cet algorithme est donc étonnamment efficace. Et il n'utilise que deux variables, soit un espace (supplémentaire) constant.

En pratique, cet algorithme est rarement utilisé pour adapter un parcours de liste au cas d'une liste cyclique. Son application se situe plutôt dans le contexte d'une « liste virtuelle » définie par un élément de départ $x_0$ et une fonction $f$ telle que $f(x)$ est l'élément qui suit $x$ dans la liste. Un générateur de nombres aléatoires est un exemple de telle fonction. L'algorithme de Floyd permet alors de calculer à partir de quel rang ce générateur entre dans un cycle et la longueur de ce cycle.

## 4.5 Listes doublement chaînées

Jusqu'à ici, nous avons défini et utilisé des listes *simplement* chaînées, c'est-à-dire où chaque élément contient un pointeur vers l'élément *suivant* dans la liste. Rien n'exclut, cependant, d'ajouter un pointeur vers l'élément *précédent* dans la liste. On parle alors de *liste doublement chaînée*. Une classe `Doubly` pour de telles listes, contenant toujours des entiers, possède donc les trois champs suivants

```
class Doubly {
  int element;
```

```
Doubly next, prev;
...
```

où `next` est le pointeur vers l'élément suivant, comme pour les listes simplement chaînées, et `prev` le pointeur vers l'élément précédent. Une liste réduite à un unique élément peut être allouée avec le constructeur suivant :

```
Doubly(int element) {
  this.element = element;
  this.next = this.prev = null;
}
```

Bien entendu, on pourrait aussi écrire un constructeur naturel qui prend en arguments les valeurs des trois champs. On ne le fait pas ici, et on choisit plutôt un style de construction de listes où de nouveaux éléments seront insérés avant ou après des éléments existants. Écrivons ainsi une méthode dynamique `insertAfter(int v)` qui ajoute un nouvel élément de valeur `v` *juste après* l'élément désigné par `this`. On commence par construire le nouvel élément `e`, avec le constructeur ci-dessus.

```
void insertAfter(int v) {
  Doubly e = new Doubly(v);
```

Il faut maintenant mettre à jour les différents pointeurs `next` et `prev` pour lier ensemble `this` et `e`. De manière évidente, on indique que l'élément qui précède `e` est `this`.

```
  e.prev = this;
```

Inversement, on souhaite indiquer que l'élément qui suit `this` est `e`. Cependant, il y avait peut-être un élément après `this`, c'est-à-dire désigné par `this.next`, et il convient alors de commencer par mettre à jour les pointeurs entre `e` et `this.next`.

```
  if (this.next != null) {
    e.next = this.next;
    e.next.prev = e;
  }
```

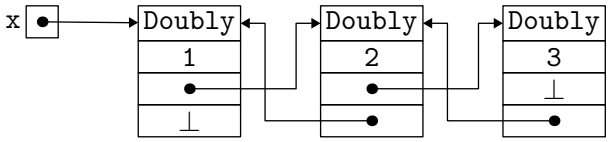
Enfin, on peut mettre à jour `this.next`, ce qui achève le code le la méthode `insertAfter`.

```
  this.next = e;
}
```

En utilisant le constructeur de la méthode `insertAfter`, on peut construire la liste contenant les entiers 1, 2, 3, dans cet ordre, en commençant par construire l'élément de valeur 1, puis en insérant successivement 3 et 2 après 1.

```
Doubly x = new Doubly(1);
x.insertAfter(3);
x.insertAfter(2);
```

On a donc alloué trois objets en mémoire, qui se référencent mutuellement de la manière suivante :




Il est important de noter que la situation étant devenue complètement symétrique, il n'y a pas forcément lieu de désigner la liste par son « premier » élément (l'élément de valeur 1 dans l'exemple ci-dessus). On pourrait tout aussi bien se donner un pointeur sur le dernier élément. Cependant, une liste doublement chaînée étant, entre autres choses, une liste simplement chaînée, on peut continuer à lui appliquer le même vocabulaire. Le premier élément est donc celui dont le champ `prev` vaut `null` et le dernier celui dont le champ `next` vaut `null`.

**Exercice 23.** Écrire de même une méthode `insertBefore(v)` qui insère un nouvel élément de valeur `v` juste avant `this`.

Solution □

**Suppression d'un élément.** Une propriété remarquable des listes doublement chaînées est qu'il est possible de supprimer un élément `e` de la liste sans connaître rien d'autre que sa propre valeur (son pointeur). En effet, ses deux champs `prev` et `next` nous donnent l'élément précédent et l'élément suivant et il suffit de les lier entre eux pour que `e` soit effectivement retiré de la liste. Écrivons une méthode dynamique `remove()` qui supprime l'élément `this` de la liste dont il fait partie. Elle est aussi simple que

```
void remove() {
  if (this.prev != null)
    this.prev.next = this.next;
  if (this.next != null)
    this.next.prev = this.prev;
}
```

La seule difficulté consiste à correctement traiter les cas où `this.prev` ou `this.next` valent `null`, c'est-à-dire lorsque `this` se trouve être le premier ou le dernier élément de la liste (voire les deux à la fois). Pour effectuer une telle suppression dans une liste simplement chaînée, il nous faudrait également un pointeur sur l'élément qui précède `e` dans la liste.

Il est important de noter que la méthode `remove` n'a pas l'effet escompté si elle est appliquée au premier élément de la liste. Dans l'exemple ci-dessus, un appel à `x.remove()` a effectivement pour effet de supprimer l'élément 1 de la liste, la réduisant à une liste ne contenant plus que 2 et 3, mais la variable `x` continue de pointer sur l'élément 1. Par ailleurs, les champs `prev` et `next` de cet élément n'ont pas été modifiés. Ainsi, un parcours de la liste `x` donnera toujours les valeurs 1, 2, 3.

Pour y remédier, plusieurs solutions peuvent être utilisées. On peut par exemple placer aux deux extrémités de la liste deux éléments fictifs, qui ne seront pas considérés comme faisant partie de la liste et qui ne seront jamais supprimés. On parle de *sentinelles*. Mieux encore, on peut encapsuler la liste doublement chaînée dans un objet qui maintient des pointeurs vers son premier et son dernier élément, exactement comme nous l'avons fait pour réaliser des piles et des files avec des listes simplement chaînées page 54.

Le code complet des listes doublement chaînées est donné programme 9 page 63.

**Programme 9 — Listes doublement chaînées**

```
class Doubly {

  int element;
  Doubly next, prev;

  Doubly(int element) {
    this.element = element;
    this.next = this.prev = null;
  }

  void insertAfter(int v) {
    Doubly e = new Doubly(v);
    e.prev = this;
    if (this.next != null) {
      e.next = this.next;
      e.next.prev = e;
    }
    this.next = e;
  }

  void remove() {
    if (this.prev != null)
      this.prev.next = this.next;
    if (this.next != null)
      this.next.prev = this.prev;
  }

}
```

**Application : le problème de Josephus.** Utilisons la structure de liste doublement chaînée pour résoudre le problème suivant, dit problème de Josephus. Des joueurs sont placés en cercle. Ils choisissent un entier $p$ et procèdent alors à une élection de la manière suivante. Partant du joueur 1, ils comptent jusqu'à $p$ et éliminent le $p$-ième joueur, qui sort du cercle. Puis, partant du joueur suivant, ils éliminent de nouveau le $p$-ième joueur, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'un joueur. Si $n$ désigne le nombre de joueurs au départ, on note $J(n,p)$ le numéro du joueur ainsi élu. Avec $n=7$ et $p=5$ on élimine successivement les joueurs 5, 3, 2, 4, 7, 1 et le gagnant est donc le joueur 6 *i.e.* $J(7,5)=6$.

Écrivons une méthode statique `josephus(int n, int p)` qui calcule la valeur de $J(n,p)$ en utilisant une liste doublement chaînée cyclique, représentant le cercle des joueurs. La méthode `remove` ci-dessus pourra alors être utilisée directement pour éliminer un joueur. On commence par écrire une méthode statique `circle(int n)` qui construit une liste doublement chaînée cyclique de longueur `n`. Elle commence par créer le premier élément, de valeur 1 (on suppose ici $n \geq 1$).

```
static Doubly circle(int n) {
  Doubly l1 = new Doubly(1);
```

Puis elle ajoute successivement tous les éléments $n, \ldots, 2$ juste après l'élément 1. En procédant dans cet ordre, on aura bien au final l'élément 2 juste après l'élément 1.

```
  for (int i = n; i >= 2; i--) {
    l1.insertAfter(i);
```

La difficulté consiste à refermer correctement la liste, pour créer le cycle (le code écrit jusqu'à présent ne permet que de construire des listes terminées par `null` de chaque côté). Pour cela, il suffit de lier ensemble les éléments 1 et $n$. À l'issue de la boucle, il ne sera pas aisé de retrouver l'élément $n$. On le fait donc *dans* la boucle, lorsque `i` vaut $n$.

```
    if (i == n) { l1.prev = l1.next; l1.next.next = l1; }
  }
```

En réalité, ce test est positif dès le premier tour de boucle (et seulement au premier). Mais le traiter à l'extérieur de la boucle nous obligerait à faire un cas particulier pour $n=1$. À l'issue de la boucle, on renvoie le premier élément.

```
  return l1;
}
```

On passe maintenant à la méthode `josephus`. Elle commence par appeler la méthode `circle` pour construire le cercle `c` des joueurs.

```
static int josephus(int n, int p) {
  Doubly c = circle(n);
```

Puis elle effectue une boucle tant qu'il reste au moins deux joueurs dans le cercle. On teste cette condition en comparant `c` et `c.next`.

```
  while (c != c.next) {
```

**Programme 10 — Le problème de Josephus**

```
// construit la liste circulaire 1,2,...,n et renvoie l'élément 1
static Doubly circle(int n) {
  Doubly l1 = new Doubly(1);
  for (int i = n; i >= 2; i--) {
    l1.insertAfter(i);
    if (i == n) { l1.prev = l1.next; l1.next.next = l1; }
  }
  return l1;
}

static int josephus(int n, int p) {
  Doubly c = circle(n);
  while (c != c.next) { // tant qu'il reste au moins deux joueurs
    for (int i = 1; i < p; i++)
      c = c.next;
    c.remove(); // on élimine le p-ième
    c = c.next;
  }
  return c.element;
}
```

Il s'agit ici d'une comparaison *physique* (avec `!=`), qui compare les valeurs en tant que pointeurs. À chaque tour de boucle, on procède à l'élimination d'un joueur. Pour cela, on avance $p-1$ fois dans le cercle, avec `c = c.next`, puis on élimine le joueur `c` ainsi obtenu avec `c.remove`.

```
    for (int i = 1; i < p; i++)
      c = c.next;
    c.remove();
```

Puis on passe à l'élément suivant. Bien que `c` vient d'être supprimé de la liste, son pointeur `next` désigne toujours l'élément suivant dans la liste. On peut donc écrire

```
    c = c.next;
  }
```

Ceci achève la boucle `while`. Le gagnant est le dernier élément dans la liste.

```
  return c.element;
}
```

Le code complet est donné programme 10 page 65. Pour plus de détails concernant ce problème, et notamment une solution analytique, on pourra consulter *Concrete Mathematics* [7, Sec. 1.3].

**Exercice 24.** Réécrire la méthode `josephus` en utilisant une liste cyclique *simplement* chaînée. Indication : dans la boucle interne, conserver un pointeur sur l'élément précédent, de manière à pouvoir supprimer facilement le $p$-ième élément en sortie de boucle.
Solution □

**Exercice 25.** Réécrire la méthode `josephus` en utilisant un tableau d'entiers plutôt qu'une liste chaînée. Solution □

## 4.6 Code générique

Écrire une version générique des listes chaînées est immédiat. Il suffit de paramétrer les classes `Singly` et `Doubly` par le type `E` des éléments. Pour les listes simplement chaînées, on écrit donc

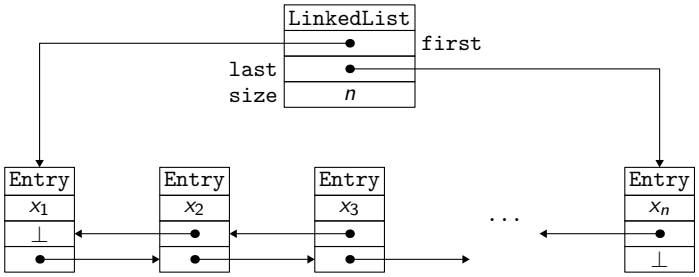
```
class Singly<E> {
  E element;
  Singly<E> next;
```

et pour les listes doublement chaînées

```
class Doubly<E> {
  E element;
  Doubly<E> next, prev;
```

Le reste du code est le même, au remplacement près de `int` par `E` aux endroits opportuns.

La bibliothèque Java fournit une classe générique de listes doublement chaînées dans `java.util.LinkedList<E>`. Sa structure conserve un pointeur vers le premier et le dernier élément de la liste, ainsi que le nombre d'éléments.




On peut ainsi opérer des deux côtés de la liste, avec des méthodes comme `addFirst`, `removeFirst`, `addLast` et `removeLast`. En particulier, la bibliothèque Java fournit une interface générique de la structure de file dans `java.util.Queue<E>`, dont `LinkedList` est une implémentation. On peut donc écrire notamment

```
Queue<T> q = new LinkedList<T>();
```

pour un certain type `T`. On peut également utiliser `LinkedList` pour réaliser une pile, mais la bibliothèque Java lui préfère une implémentation plus efficace utilisant `Vector`, proposée dans `java.util.Stack<E>`, comme nous l'avons expliqué page 48.

# 5 Tables de hachage

Supposons qu'un programme ait besoin de manipuler un *ensemble* de chaînes de caractères (des noms de personnes, des mot-clés, des URL, etc.) de telle manière que l'on puisse, efficacement, d'une part ajouter une nouvelle chaîne dans l'ensemble, et d'autre part chercher si une chaîne appartient à l'ensemble. Avec les structures de données vues jusqu'à présent, c'est-à-dire les tableaux et les listes, ce n'est pas facile. L'ajout peut certes se faire en temps constant — par exemple au début ou à la fin d'une liste ou à la fin d'un tableau redimensionnable — mais la recherche prendra un temps $O(n)$ si l'ensemble contient $n$ éléments. Bien entendu, on peut accélérer la recherche en maintenant les chaînes dans un tableau trié, mais c'est alors l'ajout qui prendra un temps $O(n)$ dans le pire des cas. Dans ce chapitre, nous présentons une solution simple et efficace à ce problème : les tables de hachage.

L'idée est très simple. Si les éléments étaient des entiers entre 0 et $m-1$, on utiliserait directement un tableau de taille $m$. Comme ce n'est pas le cas, on va se ramener à cette situation en utilisant une fonction $f$ associant aux différentes chaînes un entier dans $0..m-1$. Bien entendu, il est impossible de trouver une telle fonction *injective* en général. Il va donc falloir gérer les *collisions*, c'est-à-dire les cas où deux ou plusieurs chaînes ont la même valeur par $f$. Dans ce cas, on va les stocker dans un même « paquet ». Si la répartition entre les différents paquets est équilibrée, alors chaque paquet ne contient qu'un petit nombre de chaînes. On peut alors retrouver rapidement un élément car il ne reste plus qu'à le chercher dans son paquet. Si on réalise chaque paquet par une simple liste chaînée, ce qui convient parfaitement, une table de hachage n'est au final rien d'autre qu'un tableau de listes.

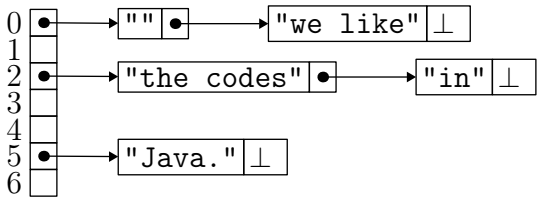
Considérons par exemple une table de hachage constituée de $m = 7$ paquets et contenant les 5 chaînes de caractères suivantes :

```
"", "We like", "the codes", "in", "Java."
```

Pour définir la fonction $f$, on commence par définir une fonction $h$, appelée *fonction de hachage*, associant un entier quelconque à chaque élément, puis on définit la fonction $f$ comme

$$f(s) = h(s) \bmod m.$$

Ainsi, l'opération *modulo* garantit que la valeur de $f$ est bien dans $0..m-1$. Supposons que l'on prenne simplement pour $h(s)$ la longueur de la chaîne $s$. Alors on obtient la structure suivante :




Ainsi le paquet 2 contient les deux chaînes `"the codes"` et `"in"`, respectivement de longueurs 9 et 2, car ces deux chaînes ont pour image 2 par la fonction $f$. (Mais l'ordre dans lequel ces deux chaînes apparaissent dans la liste peut varier suivant l'ordre d'insertion des éléments dans la table.)

## 5.1 Réalisation

Réalisons une telle table de hachage dans une classe `HashTable`. On commence par introduire une classe de liste simplement chaînée, `Bucket`, pour représenter les paquets (en anglais on parle de « seau » plutôt que de « paquet »).

```
class Bucket {
  String element;
  Bucket next;
  Bucket(String element, Bucket next) {
    this.element = element;
    this.next = next;
  }
}
```

La classe `HashTable` ne contient qu'un seul champ, à savoir le tableau des différents paquets :

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

Pour écrire le constructeur, il faut se donner une valeur pour $m$, c'est-à-dire un nombre de paquets. Idéalement, cette taille devrait être du même ordre de grandeur que le nombre d'éléments qui seront stockés dans la table. L'utilisateur pourrait éventuellement fournir cette information, par exemple sous la forme d'un argument du constructeur, mais ce n'est pas toujours possible. Considérons donc pour l'instant une situation simplifiée où cette taille est une constante complètement arbitraire, à savoir $m = 17$.

```
  final private static int M = 17;

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

(Nous verrons plus loin comment supprimer le caractère arbitraire de cette constante.) On procède alors à l'écriture de la fonction de hachage proprement dite. Il y a de nombreuses façons de la choisir, plus ou moins heureuses. De façon un peu moins naïve que la simple longueur de la chaîne, on peut chercher à combiner les valeurs des différents caractères de la chaîne, comme par exemple

```
private int hash(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++)
    h = s.charAt(i) + 19 * h;
  return (h & 0x7fffffff) % M;
}
```

Il s'agit ni plus ni moins de l'évaluation (par la méthode de Horner) d'un polynôme dont les coefficients seraient les caractères de `s` en un point complètement arbitraire, à savoir ici 19. Quoique l'on fasse, le plus important réside dans la dernière ligne, qui assure que la valeur finale est bien un indice légal dans le tableau `buckets`. En effet, l'opération de modulo `%` donne un résultat du même signe que son premier argument. On pourrait donc obtenir ici un résultat négatif si on écrivait directement `h % M`, car un débordement de la capacité du type `int` pendant le calcul de `h` peut conduire à une valeur de `h` négative. La solution consiste ici à assurer que la valeur finale de `h` est bien positive ou nulle en masquant son bit de signe avec `& 0x7fffffff`. On utilise ici la notation hexadécimale pour écrire l'entier $2^{31} - 1$, ce qui est beaucoup plus simple que la notation décimale [1].

On en arrive à l'opération d'ajout d'une chaîne `s` dans la table de hachage. C'est d'une simplicité enfantine : on calcule l'indice du paquet, avec la méthode `hash`, et on ajoute simplement `s` en tête de la liste correspondante.

```
void add(String s) {
  int i = hash(s);
  this.buckets[i] = new Bucket(s, this.buckets[i]);
}
```

Une variante consisterait à vérifier que `s` ne fait pas déjà partie de cette liste (voir l'exercice 26). Mais la version ci-dessus a l'avantage de garantir une complexité $O(1)$ dans tous les cas. Et on peut parfaitement être dans une situation où on sait que `s` ne fait pas partie de la table — par exemple parce qu'on a effectué le test d'appartenance au préalable.

Pour réaliser le test d'appartenance, justement, on procède de la même façon, en utilisant la méthode `hash` pour déterminer dans quel paquet la chaîne à rechercher doit se trouver, si elle est présente. Pour chercher dans la liste correspondante, on ajoute par exemple une méthode statique `contains` à la classe `Bucket` :

```
static boolean contains(Bucket b, String s) {
  for (; b != null; b = b.next)
    if (b.element.equals(s)) return true;
  return false;
}
```

La méthode `contains` de la classe `HashTable` est alors réduite à une seule ligne :

```
boolean contains(String s) {
  return Bucket.contains(this.buckets[hash(s)], s);
}
```

Le code complet est donné programme 11 page 70.

1. En revanche, il ne serait pas correct d'écrire `Math.abs(h) % M` car si `h` est égal au plus petit entier, c'est-à-dire $-2^{31}$, alors `Math.abs(h)` vaudra $-2^{31}$ et le résultat de `hash` sera négatif.

**Programme 11 — Tables de hachage**

```
class Bucket {
  String element;
  Bucket next;
  Bucket(String element, Bucket next) {
    this.element = element;
    this.next = next;
  }
  static boolean contains(Bucket b, String s) {
    for (; b != null; b = b.next)
      if (b.element.equals(s)) return true;
    return false;
  }
}

class HashTable {

  private Bucket[] buckets;

  final private static int M = 17;

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

  private int hash(String s) {
    int h = 0;
    for (int i = 0; i < s.length(); i++)
      h = s.charAt(i) + 19 * h;
    return (h & 0x7fffffff) % M;
  }

  void add(String s) {
    int i = hash(s);
    this.buckets[i] = new Bucket(s, this.buckets[i]);
  }

  boolean contains(String s) {
    return Bucket.contains(this.buckets[hash(s)], s);
  }
}
```

**Exercice 26.** Modifier la méthode `add` de `HashTable` pour qu'elle ne fasse rien lorsque l'élément est déjà contenu dans la table. Solution □

**Exercice 27.** Ajouter un champ privé `size` à la classe `HashTable` contenant le nombre total d'éléments de la table, ainsi qu'une méthode `int size()` renvoyant sa valeur. On pourra supposer avoir déjà effectué la modification proposée dans l'exercice précédent. Pourquoi le champ `size` doit-il être privé ? Solution □

**Exercice 28.** Ajouter une méthode `void remove(String s)` pour supprimer un élément `s` de la table de hachage. Quel est l'impact sur la méthode `add` ?
Solution □

## 5.2 Redimensionnement

Le code que nous venons de présenter est en pratique trop naïf. Le nombre d'éléments contenus dans la table peut devenir grand par rapport à la taille du tableau. Cette *charge* implique de gros paquets, qui dégradent les performances des opérations (ici seulement de l'opération `contains`). Pour y remédier, il faut modifier la taille du tableau *dynamiquement*, en fonction de la charge de la table. On commence par modifier légèrement la méthode `hash` pour obtenir une valeur modulo `this.buckets.length` et non plus modulo `M` :

```
return (h & 0x7fffffff) % this.buckets.length;
```

Puis on choisit une stratégie de redimensionnement. Par exemple, on peut choisir de doubler la taille $m$ du tableau dès que le nombre total d'éléments dépasse $m/2$. On suppose pour cela que l'on a ajouté un champ `size` à la classe `HashTable` qui contient le nombre total d'éléments (voir l'exercice 27 ci-dessus). Il suffit alors d'ajouter une ligne au début (ou à la fin) de la méthode `add` pour appeler une méthode de redimensionnement `resize` si nécessaire.

```
void add(String s) {
  if (this.size > this.buckets.length/2) resize();
  ...
```

Tout le travail se fait dans cette nouvelle méthode `resize`. On commence par calculer la nouvelle taille du tableau, comme le double de la taille actuelle :

```
private void resize() {
  int n = 2 * this.buckets.length;
```

Puis on alloue un nouveau tableau de cette taille-là dans `this.buckets`, sans oublier de conserver un pointeur sur son ancienne valeur :

```
Bucket[] old = this.buckets;
this.buckets = new Bucket[n];
```

Enfin, on « re-hache » toutes les valeurs de l'ancien tableau `old` vers le nouveau tableau `this.bucket`. On le fait en parcourant toutes les listes de `old` avec une boucle `for` et, pour chaque liste, tous ses éléments avec une seconde boucle `for` :

```
for (Bucket b : old)
  for (; b != null; b = b.next) {
    int i = hash(b.element);
    this.buckets[i] = new Bucket(b.element, this.buckets[i]);
  }
```

Tout se passe correctement, car la méthode `hash` a été modifiée pour rendre une valeur modulo la taille de `this.buckets`. Le reste du code de la classe `HashTable` est inchangé, en particulier la méthode `contains`. Pour ce qui est de l'initialisation, on peut continuer à utiliser la constante arbitraire `M` pour allouer le tableau, car il sera agrandi si nécessaire.

**Complexité.** Nous n'avons pas choisi la stratégie consistant à doubler la taille du tableau par hasard. Exactement comme nous l'avons fait pour les tableaux redimensionnables (voir page 43), on peut montrer que l'insertion successive de $n$ éléments dans la table de hachage aura un coût total $O(n)$. Certains appels à `add` sont plus coûteux que d'autres, et même d'une complexité proportionnelle au nombre d'éléments déjà dans la table, mais la *complexité amortie* de `add` reste $O(1)$.

La complexité de la recherche est plus difficile à évaluer, car elle dépend de la « qualité » de la fonction de hachage. Si la fonction de hachage envoie tous les éléments dans le même paquet — c'est le cas par exemple si elle est constante — alors la complexité de `contains` sera clairement $O(n)$. Si au contraire la fonction de hachage répartit bien les éléments dans les différents paquets, alors la taille de chaque paquet peut être bornée par une constante et la complexité de `contains` sera alors $O(1)$. La mise au point d'une fonction de hachage se fait empiriquement, par exemple en mesurant la taille maximale et moyenne des paquets. Sur des types tels que des chaînes de caractères, ou encore des tableaux d'entiers, une fonction telle que celle que nous avons donnée plus haut donne des résultats très satisfaisants.

**Exercice 29.** Vu que l'on double la taille du tableau à chaque fois que la table de hachage est agrandie, on peut maintenir l'invariant que cette taille est toujours une puissance de deux. L'intérêt est que l'on peut alors simplifier le calcul

```
return (h & 0x7fffffff) % this.buckets.length;
```

pour éviter notamment l'opération `%`. Expliquer comment. Solution □

## 5.3 Code générique

Bien entendu, la structure de table de hachage que nous venons de présenter s'adapte facilement à des éléments autres que des chaînes de caractères. Il suffit pour cela de modifier d'une part la fonction de hachage (ici notre méthode `hash`) et d'autre part l'égalité utilisée pour comparer les éléments (ici la méthode `equals` de la classe `String`). La bibliothèque Java propose justement une version générique des tables de hachage, sous la forme d'une classe `java.util.HashSet<E>` pour des ensembles dont les éléments sont d'un type `E` et d'une classe `java.util.HashMap<K, V>` pour des dictionnaires associant à des valeurs de type `K` des valeurs de type `V`.

Dans les deux cas, il faut équiper les types `E` et `K` d'une fonction de hachage et d'une égalité adaptées. On le fait en *redéfinissant* les méthodes `int hashCode()` et

`boolean equals(Object)` héritées de la classe `Object`. Si par exemple on définit une classe `Pair` pour des paires de chaînes de caractères, de la forme

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

alors il conviendra de l'équiper d'une fonction de hachage d'une part, par exemple en faisant la somme des valeurs de hachage des deux chaînes `fst` et `snd`

```
  public int hashCode() {
    return this.fst.hashCode() + this.snd.hashCode();
  }
```

et d'une égalité structurelle d'autre part en comparant les deux paires membre à membre. Il y a là une subtilité : la méthode `equals` est définie dans la classe `Object` avec un argument de type `Object` et il faut donc respecter ce profil de méthode pour la redéfinir. On doit donc écrire

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

où `(Pair)o` est une conversion explicite — car potentiellement non sûre — de la classe `Object` vers la classe `Pair`. Si cette méthode `equals` n'est utilisée que depuis le code de `HashSet<Pair>` ou de `HashMap<Pair, V>`, on a la garantie que cette conversion n'échouera jamais. En effet, le typage de Java nous garantit qu'un ensemble de type `HashSet<Pair>` (resp. un dictionnaire de type `HashMap<Pair, V>`) ne pourra contenir que des éléments (resp. des clés) de type `Pair`.

Il convient d'expliquer soigneusement un piège dans lequel on aurait pu facilement tomber. Naturellement, on aurait plutôt écrit la méthode suivante :

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

Mais, bien qu'accepté par le compilateur, ce code ne donne pas les résultats attendus. En effet, la méthode `equals` est maintenant *surchargée* et non plus *redéfinie* : il y a deux méthodes `equals`, l'une prenant un argument de type `Object` et l'autre prenant un argument de type `Pair`. Comme le code de `HashSet` et `HashMap` est écrit en utilisant la méthode `equals` ayant un argument de type `Object` (même si cela peut surprendre), alors c'est la première qui est utilisée, c'est-à-dire celle directement héritée de la classe `Object`. Il se trouve qu'elle coïncide avec l'égalité physique, c'est-à-dire avec l'opération `==`, ce qui n'est pas en accord avec l'égalité structurelle que nous souhaitons ici sur le type `Pair`.

Une façon d'éviter ce piège consiste à indiquer au compilateur Java qu'il s'agit d'une redéfinition de méthode, à l'aide de la directive `@Override` placée juste avant la définition de la méthode :

```
@Override
public boolean equals(Object o) {
  ...
```

Le compilateur vérifie alors qu'il s'agit bien là d'une redéfinition, c'est-à-dire qu'il existe effectivement une telle méthode, avec ce type-là, dans la super-classe. Dans le cas contraire, la compilation échoue.

Quelle que soit la façon de redéfinir les méthodes `hashCode` et `equals`, il convient de toujours maintenir la propriété suivante :

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

Autrement dit, des éléments égaux doivent être rangés dans le même seau.

## 5.4 Brève comparaison des tableaux, listes et tables de hachage

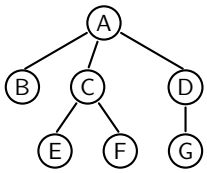
On peut chercher à comparer les structures de données que nous avons déjà vues, à savoir les tableaux, les listes et les tables de hachage, du point de vue des opérations ensemblistes que sont l'ajout d'un élément (`add`), l'accès au $i$-ième élément (`get`) et la recherche d'un élément (`contains`). Les différentes complexités sont les suivantes :

| | `add` | `get` | `contains` |
|---|---|---|---|
| tableau | $O(1)$ amorti | $O(1)$ | $O(n)$ |
| tableau trié | $O(n)$ | $O(1)$ | $O(\log n)$ |
| liste | $O(1)$ | $O(i)$ | $O(n)$ |
| table de hachage | $O(1)$ amorti | — | $O(1)$ |

Pour une table de hachage, l'accès au $i$-ième élément n'est pas défini. Il est cependant possible de combiner les structures de table de hachage et de liste chaînée pour conserver, à côté d'une table de hachage, l'ordre d'insertion des éléments. Les bibliothèques Java `java.util.LinkedHashSet<E>` et `java.util.LinkedHashMap<E>` font cela.

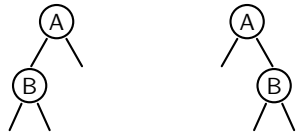
# 6 Arbres

La notion d'*arbre* est définie récursivement. Un arbre est un ensemble fini de *nœuds*, étiquetés par des valeurs, où un nœud particulier $r$ est appelé la *racine* de l'arbre et les autres nœuds forment des arbres disjoints appelés les *fils* (ou *sous-arbres*) de $r$. En informatique, les arbres poussent vers le bas. Ainsi




représente un arbre de racine A ayant trois fils. Un nœud qui ne possède aucun fils est appelé une *feuille*. Les feuilles de l'arbre ci-dessus sont B, E, F et G. La *hauteur* d'un arbre est définie comme le nombre de nœuds le long du plus long chemin de la racine à une feuille (ou, de manière équivalente, comme la longueur de ce chemin, plus un). La hauteur de l'arbre ci-dessus est donc trois.

La notion d'*arbre binaire* est également définie récursivement. Un arbre binaire est soit vide, soit un nœud possédant exactement deux fils appelés fils gauche et fils droit. Un arbre binaire n'est pas un cas particulier d'arbre, car on distingue les sous-arbres gauche et droit (on parle d'arbre positionnel). Ainsi, les deux arbres suivants sont distincts :




On montre facilement qu'un arbre binaire de hauteur $h$ possède au plus $2^h - 1$ nœuds, par récurrence forte sur $h$. Pour $h = 0$, c'est clair. Pour $h > 0$, on a deux sous-arbres de hauteurs au plus $h - 1$, d'où un total d'au plus $1 + 2^{h-1} - 1 + 2^{h-1} - 1 = 2^h - 1$ nœuds.

**Exercice 30.** Soit un arbre binaire tel que, pour tout nœud, les sous-arbres gauche et droit contiennent le même nombre de nœuds, à un près. Montrer qu'alors sa hauteur est logarithmique en son nombre de nœuds. Solution □

## 6.1 Représentation des arbres

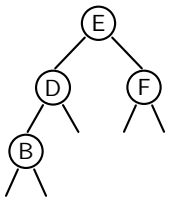
Considérons pour l'instant des arbres binaires uniquement, dont les nœuds contiennent des entiers. Une classe `Tree` pour représenter les nœuds de tels arbres est donc

```
class Tree {
  int value;
  Tree left, right;
}
```

où les champs `left` et `right` contiennent respectivement le fils gauche et le fils droit. L'arbre vide est représenté par `null`. Cette représentation n'est en rien différente de la représentation d'une liste doublement chaînée (voir page 63), aux noms des champs près. Ce qui change, c'est l'invariant de structure. Pour une liste doublement chaînée, la structure imposée par construction était linéaire : tout élément suivait son précédent et précédait son suivant. Ici la structure imposée par construction sera celle d'un arbre. En se donnant le constructeur naturel de la classe `Tree`, à savoir

```
Tree(Tree left, int value, Tree right) {
  this.left = left;
  this.value = value;
  this.right = right;
}
```

et des entiers B, D, E et F, on peut construire un arbre avec l'expression `new Tree(new Tree(new Tree(B, null, null), D, null), E, new Tree(null, F, null))`. On le dessine de façon simplifiée, sans expliciter les objets comme des petites boîtes avec des champs ; cela prendrait trop de place.




Plus loin dans ce chapitre, nous montrerons d'autres façons de construire des arbres, dans les sections 6.4 et 7.2.

## 6.2 Opérations élémentaires sur les arbres

La méthode la plus simple à écrire sur un arbre est sûrement celle qui compte le nombre de ses éléments. On procède naturellement récursivement :

```
static int size(Tree t) {
  if (t == null) return 0;
  return 1 + size(t.left) + size(t.right);
}
```

Il est facile de montrer que sa complexité est proportionnelle au nombre de nœuds de l'arbre. Ce code est particulièrement simple mais il possède néanmoins le défaut d'un éventuel débordement de pile, *i.e.* le déclenchement de l'exception `StackOverflowError`, si la hauteur de l'arbre est très grande. (Les exercices 31 et 32 proposent de le vérifier.) Pour y remédier, il faudrait réécrire la méthode `size` avec une boucle. Mais, à la différence d'une liste chaînée pour laquelle nous aurions pu calculer la longueur à l'aide d'une boucle, on ne voit pas ici comment faire cela facilement. C'est en fait possible [1] et l'exercice 33 propose une solution. Néanmoins, il existe de nombreuses situations où la hauteur des arbres est limitée (comme par exemple les arbres équilibrés de la section 6.3.2 ci-dessous) et où il n'y a donc pas lieu de se soucier d'un éventuel `StackOverflowError`.

**Exercice 31.** Écrire une méthode statique `Tree leftDeepTree(int n)` qui construit un arbre *linéaire gauche* contenant `n` nœuds. Un arbre linéaire gauche est un arbre où chaque nœud ne possède pas de fils droit. Solution □

**Exercice 32.** Déterminer une valeur de `n` pour laquelle le résultat de `leftDeepTree(n)` provoque une exception `StackOverflowError` lorsqu'il est passé en argument à la méthode `size`. Solution □

**Exercice 33.** Réécrire la méthode `size` avec une boucle `while`. Indication : utiliser une structure de données contenant des sous-arbres dont il faut calculer la taille. Solution □

**Exercice 34.** Écrire une méthode statique récursive `int height(Tree t)` qui renvoie la hauteur d'un arbre. Solution □

**Exercice 35.** Réécrire la méthode `height` de l'exercice précédent sans utiliser de récursivité. Indication : parcourir les nœuds de l'arbre « niveau par niveau » en utilisant une file. Solution □

**Parcours.** De même que nous avions écrit des méthodes parcourant les éléments d'une liste chaînée, on peut chercher à parcourir les éléments d'un arbre, par exemple pour les afficher tous. Supposons par exemple que l'on veuille afficher les éléments « de la gauche vers la droite », c'est-à-dire d'abord les éléments du fils gauche, puis la racine, puis les éléments du fils droit. Là encore, il est naturel de procéder récursivement et le parcours est aussi simple que

```
static void inorderTraversal(Tree t) {
  if (t == null) return;
  inorderTraversal(t.left);
  System.out.println(t.value);
  inorderTraversal(t.right);
}
```

Un tel parcours est appelé un *parcours infixe* de l'arbre (*inorder traversal* en anglais). Si on affiche la valeur de la racine avant le parcours du fils gauche (resp. après le parcours du fils droit) on parle de *parcours préfixe* (resp. *postfixe*) de l'arbre.

1. Il est même toujours possible de remplacer une fonction récursive par une boucle.

## 6.3 Arbres binaires de recherche

Si les éléments qui étiquettent les nœuds d'un arbre sont totalement ordonnés — c'est le cas des entiers, par exemple — alors il est possible de donner plus de structure à un arbre binaire en maintenant l'invariant suivant :

> Pour tout nœud de l'arbre, de valeur $x$, les éléments situés dans le fils gauche sont plus petits que $x$ et ceux situés dans le fils droit sont plus grands que $x$.

On appelle cela un *arbre binaire de recherche*. En particulier, on en déduit que les éléments apparaissent dans l'ordre croissant lorsque l'arbre est parcouru dans l'ordre infixe. Nous allons exploiter cette structure pour écrire des opérations de recherche et de modification efficaces. Par exemple, chercher un élément dans un arbre binaire de recherche ne requiert pas de parcourir tout l'arbre : il suffit de descendre à gauche ou à droite selon la comparaison entre l'élément recherché et la racine de l'arbre. Dans ce qui suit, on considère des arbres binaires de recherche dont les valeurs sont des entiers. On se donne donc la classe suivante pour les représenter.

```
class BST {
  int value;
  BST left, right;
}
```

On suppose écrit un constructeur analogue à celui de la classe `Tree`.

### 6.3.1 Opérations élémentaires

**Plus petit élément.** La structure d'arbre binaire de recherche permet notamment d'obtenir facilement son plus petit élément. Il suffit en effet de descendre le long de la branche gauche, tant que cela est possible. La méthode `getMin` réalise ce parcours, avec une boucle `while` :

```
  static int getMin(BST b) {
    while (b.left != null) b = b.left;
    return b.value;
  }
```

Elle suppose que `b` n'est pas `null` et contient donc au moins un élément. Cette méthode sera réutilisée plus loin pour écrire la méthode de suppression dans un arbre binaire de recherche. Le cas de `null` y sera alors traité de façon particulière.

**Recherche d'un élément.** La recherche d'un élément `x` consiste à descendre dans l'arbre jusqu'à ce qu'on atteigne soit un nœud contenant la valeur `x`, soit `null`. Lorsque le nœud ne contient pas `x`, la propriété d'arbre binaire de recherche nous indique de quel côté poursuivre la descente. Là encore, on peut écrire cette descente sous la forme d'une boucle `while`.

```
  static boolean contains(BST b, int x) {
    while (b != null) {
      if (x == b.value) return true;
```

```
      b = (x < b.value) ? b.left : b.right;
    }
    return false;
  }
```

**Exercice 36.** Réécrire la méthode `contains` récursivement. Solution □

**Exercice 37.** Écrire une méthode `static int floor(BST b, int x)` qui renvoie le plus grand élément de `b` inférieur ou égal à `x`, s'il existe, et lève une exception sinon. Solution □

**Insertion d'un élément.** L'insertion d'un élément `x` dans un arbre binaire de recherche `b` consiste à trouver l'emplacement de `x` dans `b`, en suivant le même principe que pour la recherche. On écrit pour cela une méthode `add` :

```
  static BST add(BST b, int x) {
```

Cette méthode renvoie la racine de l'arbre, une fois l'insertion réalisée. On procède récursivement. Si `b` est vide, on se contente de construire un arbre contenant uniquement `x`.

```
    if (b == null)
      return new BST(null, x, null);
```

Dans l'autre cas, on compare l'élément `x` à la racine de `b`, et on poursuit récursivement l'insertion à gauche ou à droite lorsque la comparaison est stricte :

```
    if (x < b.value)
      b.left = add(b.left, x);
    else if (x > b.value)
      b.right = add(b.right, x);
```

On prend soin de mettre à jour `b.left` ou `b.right`, selon le cas, avec le résultat de l'appel récursif. Cette affectation est en fait inutile pour tous les nœuds internes le long de la descente, mais nécessaire pour le dernier nœud rencontré, auquel on ajoute un nouveau sous-arbre. Dans le cas où `x` est égal à `b.value`, on ne fait rien. Dans tous les cas, on termine la méthode `add` en renvoyant `b`.

```
    return b;
  }
```

On fait ici le choix de ne pas construire d'arbre contenant de doublon, mais on aurait très bien pu choisir de renvoyer au contraire un arbre contenant une occurrence supplémentaire de `x`. Le choix que nous faisons ici est cohérent avec l'utilisation des arbres binaires de recherche que nous allons faire plus loin pour réaliser une structure d'ensemble.

**Exercice 38.** Écrire la variante de la méthode `add` qui ajoute `x` dans l'arbre dans tous les cas, *i.e.* même si `x` y apparaît déjà. (On réalise donc un multi-ensemble plutôt qu'un ensemble.) Solution □

**Exercice 39.** Pourquoi est-il difficile d'écrire la méthode `add` avec une boucle `while` plutôt que récursivement ? Solution □

**Suppression d'un élément.** La suppression d'un élément `x` dans un arbre binaire de recherche `b` procède de la même manière que pour l'insertion, c'est-à-dire par une descente récursive vers la position potentielle de `x`. Si `b` est vide, on se contente de renvoyer l'arbre vide.

```
static BST remove(BST b, int x) {
  if (b == null)
    return null;
```

Sinon, on compare l'élément `x` à la racine de `b` et on poursuit récursivement la suppression à gauche ou à droite lorsque la comparaison est stricte :

```
  if (x < b.value)
    b.left = remove(b.left, x);
  else if (x > b.value)
    b.right = remove(b.right, x);
```

Lorsqu'il y a égalité, en revanche, on se retrouve confronté à une difficulté : il faut supprimer la racine de l'arbre, c'est-à-dire renvoyer un arbre contenant exactement les éléments de `b.left` et `b.right`, mais il n'y a pas de moyen simple de réaliser cette union. On souhaite autant que possible conserver `b.left` ou `b.right` inchangé, pour limiter la quantité de nœuds à modifier. La propriété d'arbre binaire de recherche nous suggère alors de placer à la racine du nouvel arbre, soit le plus grand élément de `b.left`, soit le plus petit élément de `b.right`. Vu que nous avons déjà une méthode `getMin`, nous allons opter pour la seconde solution. Il convient de traiter correctement le cas où `b.right` ne possède aucun élément. Dans ce cas, il suffit de renvoyer `b.left`.

```
  else { // x == b.value
    if (b.right == null)
      return b.left;
```

Sinon, la racine devient `getMin(b.right)` et cet élément est supprimé du sous-arbre droit avec une méthode `removeMin` que nous allons écrire dans un instant.

```
    b.value = getMin(b.right);
    b.right = removeMin(b.right);
  }
  return b;
}
```

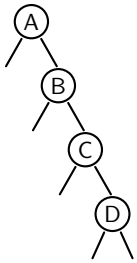
Dans tous les cas, on achève le code de `remove` en renvoyant `b`, le principe étant le même que pour `add`. Écrivons maintenant la méthode `removeMin`. C'est un cas particulier de `remove`, beaucoup plus simple, où l'on descend uniquement à gauche, jusqu'à trouver un nœud n'ayant pas de sous-arbre gauche.

```
static BST removeMin(BST b) {
  if (b.left == null)
    return b.right;
  b.left = removeMin(b.left);
  return b;
}
```

Il est important de noter que cette méthode suppose `b` différent de `null` (dans le cas contraire, `b.left` provoquerait un `NullPointerException`). Cette propriété est bien assurée par la méthode `remove`. Le code complet de la classe `BST` est donné programme 12 page 82.

### 6.3.2 Équilibrage

Telles que nous venons de les écrire dans la section précédente, les différentes opérations sur les arbres binaires de recherche ont une complexité linéaire, c'est-à-dire $O(n)$ où $n$ est le nombre d'éléments contenus dans l'arbre. En effet, notre insertion peut tout à fait conduire à un « peigne » c'est-à-dire un arbre de la forme




Il suffit en effet d'insérer les éléments dans l'ordre A, B, C, D. Une insertion dans l'ordre inverse donnerait de même un peigne, dans l'autre sens. Au-delà de la dégradation des performances, un tel arbre linéaire peut provoquer un débordement de pile dans les méthodes récursives telles que `add` ou `remove`, se traduisant par une exception `StackOverflowError`.

Dans cette section, nous allons *équilibrer* les arbres binaires de recherche, de manière à garantir une *hauteur logarithmique* en le nombre d'éléments. Ainsi les différentes opérations auront une complexité $O(\log n)$ et le débordement de pile sera évité. Il existe de nombreuses manières d'équilibrer un arbre binaire de recherche. Nous optons ici pour une solution connue sous le nom d'AVL (de leurs auteurs Adelson-Velsky et Landis [1]). Elle consiste à maintenir l'invariant suivant :

> Pour tout nœud, les hauteurs de ses sous-arbres gauche et droit diffèrent d'au plus une unité.

Écrivons une nouvelle classe `AVL` pour les arbres binaires de recherche équilibrés. On reprend la structure de la classe `BST`, à laquelle on ajoute un champ `height` contenant la hauteur de l'arbre :

```
class AVL {
  int value;
  AVL left, right;
  int height;
  ...
```

Pour traiter correctement le cas d'un arbre vide, on se donne la méthode suivante pour renvoyer la hauteur d'un arbre :

```
  static int height(AVL a) {
    return (a == null) ? 0 : a.height;
  }
```

**Programme 12 — Arbres binaires de recherche**

```
class BST {
  int value;
  BST left, right;
  BST(BST left, int value, BST right) {
    this.left = left; this.value = value; this.right = right;
  }
  static boolean contains(BST b, int x) {
    while (b != null) {
      if (b.value == x) return true;
      b = (x < b.value) ? b.left : b.right;
    }
    return false;
  }
  static BST add(BST b, int x) {
    if (b == null) return new BST(null, x, null);
    if (x < b.value)
      b.left = add(b.left, x);
    else if (x > b.value)
      b.right = add(b.right, x);
    return b;
  }
  static int getMin(BST b) { // suppose b != null
    while (b.left != null) b = b.left;
    return b.value;
  }
  static BST removeMin(BST b) { // suppose b != null
    if (b.left == null) return b.right;
    b.left = removeMin(b.left);
    return b;
  }
  static BST remove(BST b, int x) {
    if (b == null) return null;
    if (x < b.value)
      b.left = remove(b.left, x);
    else if (x > b.value)
      b.right = remove(b.right, x);
    else { // x == b.value
      if (b.right == null)
        return b.left;
      b.value = getMin(b.right);
      b.right = removeMin(b.right);
    }
    return b;
  }
}
```

Dès lors, on peut écrire un constructeur qui calcule la hauteur de l'arbre en fonction des hauteurs de ses sous-arbres `left` et `right`.

```
AVL(AVL left, int value, AVL right) {
  this.left = left;
  this.value = value;
  this.right = right;
  this.height = 1 + Math.max(height(left), height(right));
}
```

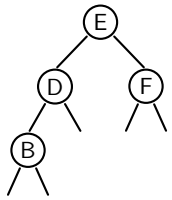
Il n'y a pas là de circularité malsaine : la méthode `height` permet de renvoyer la hauteur d'un arbre *déjà construit* et le constructeur s'en sert pour calculer la hauteur *au moment de la construction*, avec des arbres `left` et `right` déjà construits.

Les méthodes qui ne construisent pas d'arbres, mais ne font que les consulter, sont exactement les mêmes que dans la classe `BST` (en remplaçant partout `BST` par `AVL`, bien évidemment). C'est le cas des méthodes `getMin` et `contains`. En revanche, pour les méthodes qui construisent des arbres, c'est-à-dire les méthodes `add`, `removeMin` et `remove`, une modification est nécessaire, car il faut parfois rétablir l'équilibrage. On va écrire une méthode

```
static AVL balance(AVL t) { ... }
```

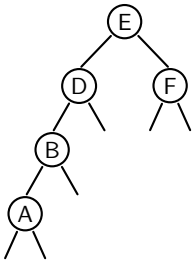
qui se comportera comme l'identité, au sens où elle renverra un arbre binaire de recherche contenant exactement les mêmes éléments que `t`, mais qui pourra effectuer des opérations de rééquilibrage si nécessaire. Avec une telle méthode, il suffit de remplacer, à la dernière ligne des trois méthodes `add`, `removeMin` et `remove`, l'instruction `return b` par `return balance(b)`. Le code est donné programme 13.

Il reste à écrire le code de la méthode `balance`. Illustrons l'idée de rééquilibrage sur un exemple. Si on considère l'arbre suivant (qui est bien un AVL)




(6.1)

et que l'on insère la valeur A avec la méthode d'insertion dans les arbres binaires de recherche, alors on obtient l'arbre



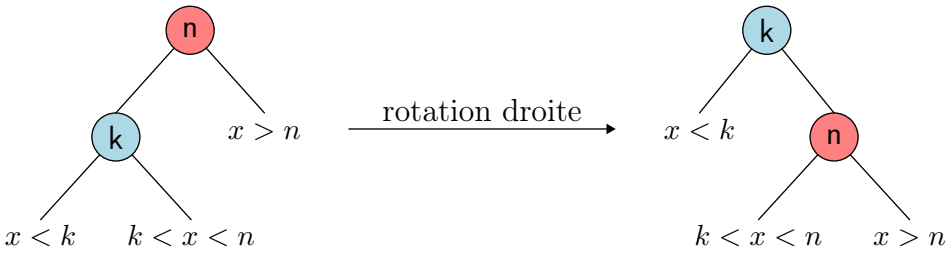

(6.2)

qui n'est pas équilibré, puisque la différence de hauteurs entre les sous-arbres gauche et droit du nœud E est maintenant de deux. Il est néanmoins facile de rétablir l'équilibre.

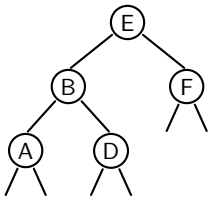
**Programme 13 — Arbres binaires de recherche équilibrés AVL (1/2)**

```
class AVL {
  int value;
  AVL left, right;
  int height;

  AVL(AVL left, int value, AVL right) {
    this.left = left;
    this.value = value;
    this.right = right;
    this.height = 1 + Math.max(height(left), height(right));
  }

  static int height(AVL a) {
    return (a == null) ? 0 : a.height;
  }

  static boolean contains(AVL a, int x) {
    // même code que pour BST
  }

  static int getMin(AVL b) {
    // même code que pour BST
  }

  static AVL add(AVL b, int x) {
    // même code que pour BST, sauf
    return balance(b);
  }
  static AVL removeMin(AVL b) {
    // même code que pour BST, sauf
    return balance(b);
  }
  static AVL remove(AVL b, int x) {
    // même code que pour BST, sauf
    return balance(b);
  }
}
```

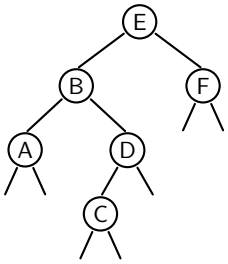
En effet, il est possible d'effectuer des transformations locales sur les nœuds d'un arbre qui conservent la propriété d'arbre binaire de recherche. Un exemple de telle opération est la *rotation droite*, qui s'illustre ainsi :



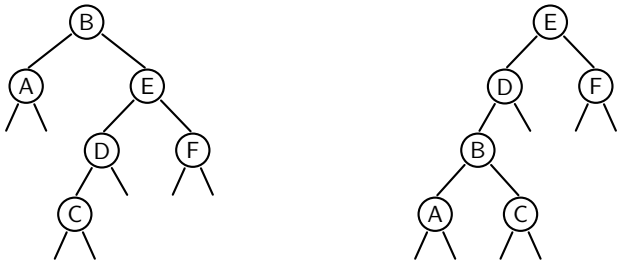

Cette opération remplace la racine $n$ par la racine $k$ du sous-arbre gauche et déplace le sous-arbre contenant les éléments compris entre $k$ et $n$. On note que cette opération ne modifie que deux nœuds dans la structure de l'arbre. De manière symétrique, on peut effectuer une rotation gauche. Ainsi l'arbre (6.2) peut être rééquilibré en effectuant une rotation droite sur le sous-arbre de racine D. On obtient alors l'arbre



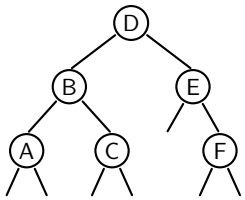

qui est bien un AVL. Une simple rotation, gauche ou droite, ne suffit pas nécessairement à rétablir l'équilibre. Si par exemple on insère maintenant C, on obtient l'arbre




qui n'est pas un AVL. On peut alors tenter d'effectuer une rotation droite à la racine E ou une rotation gauche au nœud B, mais on obtient les deux arbres suivants




qui ne sont toujours pas des AVL. Cependant, celui de droite peut être facilement rééquilibré en effectuant une rotation droite sur la racine E. On obtient alors l'AVL




Cette double opération s'appelle une *rotation gauche-droite*. On a évidemment l'opération symétrique de rotation droite-gauche. Ces quatre opérations, à savoir les deux rotations simples et les deux rotations doubles, suffisent à rééquilibrer les AVL en toute circonstance.

Écrivons maintenant le code de la méthode `balance` pour mettre en œuvre l'équilibrage. On commence par écrire une méthode `rotateRight` pour effectuer une rotation simple vers la droite. Elle suppose que l'arbre `t` passé en argument n'est pas `null`, de même que son sous-arbre gauche.

```
static AVL rotateRight(AVL t) {
  assert t != null && t.left != null;
```

La rotation proprement dite s'effectue avec deux affectations, en prenant soin de copier la valeur de `t.left` dans une variable `l`.

```
    AVL l = t.left;
    t.left = l.right;
    l.right = t;
```

On met ensuite à jour les hauteurs des sous-arbres `t` et `l` dont on vient de modifier la structure.

```
    t.height = 1 + Math.max(height(t.left), height(t.right));
    l.height = 1 + Math.max(height(l.left), height(l.right));
```

Pour terminer, on renvoie ce qui est maintenant la nouvelle racine, c'est-à-dire `l`.

```
    return l;
}
```

On écrit de même une méthode `rotateLeft` pour effectuer une rotation simple vers la gauche.

On peut maintenant écrire le code de la méthode `balance`. On commence par calculer les hauteurs `hl` et `hr` des deux sous-arbres gauche et droit

```
static AVL balance(AVL t) {
  assert t != null;
  AVL l = t.left, r = t.right;
  int hl = height(l), hr = height(r);
```

On considère en premier lieu le cas où le déséquilibre est causé par le sous-arbre gauche `l` :

```
    if (hl > hr + 1) {
```

Une simple rotation droite suffit lorsque le sous-arbre gauche `ll` de `l` est au moins aussi haut que son sous-arbre droit `lr` :

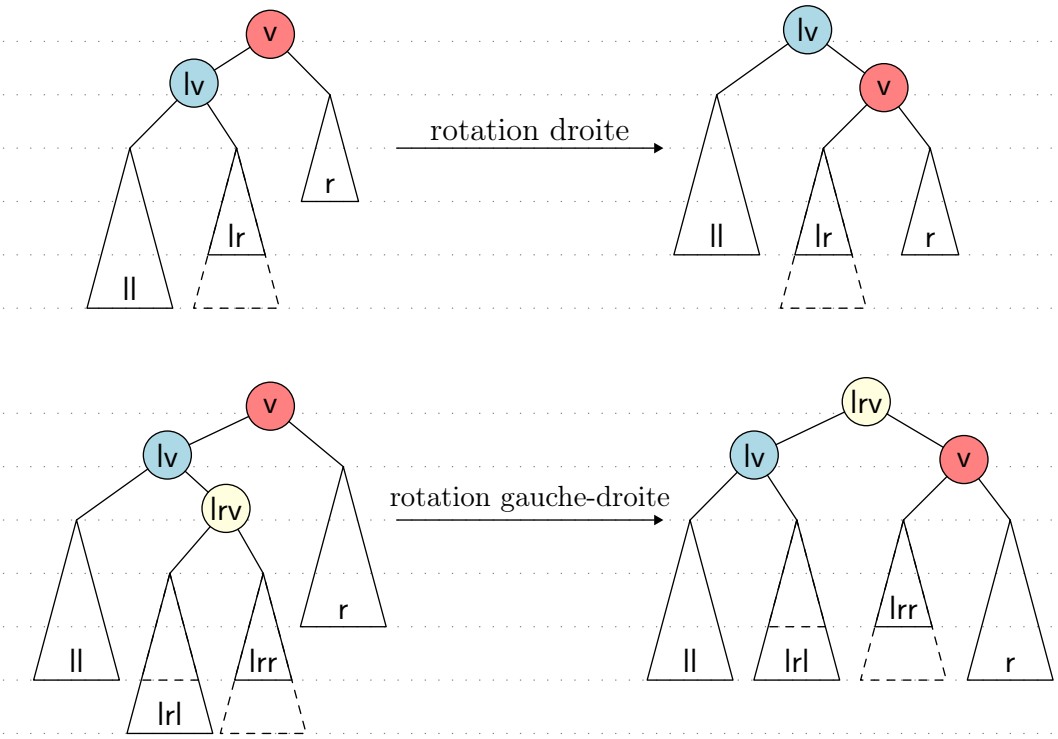



FIGURE 6.1 – Rotations vers la droite dans un AVL.

```
AVL ll = l.left, lr = l.right;
if (height(ll) >= height(lr))
  return rotateRight(t);
```

On note que `l` ne peut être `null` ici, car `hl > hr+1`. Cette rotation est illustrée figure 6.1 (en haut). En revanche, dans le cas où `ll` est moins haut que `lr`, il faut effectuer une double rotation gauche-droite.

```
else {
  t.left = rotateLeft(t.left);
  return rotateRight(t);
}
```

La propriété d'AVL est bien garantie, comme le montre la figure 6.1 (en bas). On notera que le déséquilibre peut être causé par lrl ou lrr, indifféremment, et que dans les deux cas la double rotation gauche-droite rétablit bien l'équilibre. On traite de manière symétrique le cas où `r` est la cause du déséquilibre

```
} else if (hr > hl + 1) {
  ...
```

(le code complet est donné page 89). Enfin, si les hauteurs de `l` et `r` diffèrent d'au plus un, il n'y a aucune rotation à faire mais on doit tout de même mettre la hauteur de `t` à jour avant de le renvoyer

```
} else {
  t.height = 1 + Math.max(hl, hr);
```

```
    return t;
  }
```

ce qui achève le code de la méthode `balance`. Le code complet est donné programme 14.

**Hauteur d'un AVL.** Montrons qu'un AVL a effectivement une hauteur logarithmique en son nombre d'éléments. Considérons un AVL de hauteur $h$ et cherchons à encadrer son nombre $n$ d'éléments. Clairement $n \leq 2^h - 1$, comme dans tout arbre binaire. Inversement, quelle est la plus petite valeur possible pour $n$ ? Elle sera atteinte pour un arbre ayant un sous-arbre de hauteur $h-1$ et un autre de hauteur $h-2$ (car dans le cas contraire on pourrait encore enlever des éléments à l'un des deux sous-arbres tout en conservant la propriété d'AVL). En notant $N_h$ le plus petit nombre d'éléments dans un AVL de hauteur $h$, on a donc $N_h = 1 + N_{h-1} + N_{h-2}$, ce qui se réécrit $N_h + 1 = (N_{h-1} + 1) + (N_{h-2} + 1)$. On reconnaît là la relation de récurrence définissant la suite de Fibonacci. Comme on a par ailleurs $N_0 = 0$ et $N_1 = 1$, c'est-à-dire $N_0 + 1 = 1$ et $N_1 + 1 = 2$, on en déduit $N_h + 1 = F_{h+2}$ où $(F_i)$ est la suite de Fibonacci. On a l'inégalité $F_i > \phi^i/\sqrt{5} - 1$ où $\phi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or, d'où

$$n \geq F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2$$

En prenant le logarithme (à base 2) de cette inégalité, on en déduit la majoration recherchée sur la hauteur $h$ en fonction du nombre d'éléments $n$ :

$$\begin{aligned} h &< \frac{1}{\log_2 \phi} \log_2(n+2) + \frac{\log_2 \sqrt{5}}{\log_2 \phi} - 2 \\ &\approx 1,44 \log_2(n+2) - 0,328 \end{aligned}$$

Un AVL a donc bien une hauteur logarithmique en son nombre d'éléments. Comme nous l'avons dit plus haut, cela garantit une complexité $O(\log n)$ pour les toutes les opérations, mais aussi l'absence de `StackOverflowError`.

### 6.3.3 Structure d'ensemble

En utilisant la classe `AVL` décrite dans la section précédente, écrivons une classe `AVLSet` pour représenter des ensembles d'entiers, avec l'interface suivante :

```
boolean isEmpty();
boolean contains(int x);
void add(int x);
void remove(int x);
```

Exactement comme nous l'avons fait précédemment pour construire des structures de pile et de file au dessus de la structure de liste chaînée, nous encapsulons un objet de type `AVL` dans cette nouvelle classe `AVLSet` :

```
class AVLSet {
  private AVL root;
  ...
}
```

**Programme 14 — Arbres binaires de recherche équilibrés AVL (2/2)**

```
private static AVL rotateRight(AVL t) {
  assert t != null && t.left != null;
  AVL l = t.left;
  t.left = l.right;
  l.right = t;
  t.height = 1 + Math.max(height(t.left), height(t.right));
  l.height = 1 + Math.max(height(l.left), height(l.right));
  return l;
}

private static AVL rotateLeft(AVL t) {
  assert t != null && t.right != null;
  AVL r = t.right;
  t.right = r.left;
  r.left = t;
  t.height = 1 + Math.max(height(t.left), height(t.right));
  r.height = 1 + Math.max(height(r.left), height(r.right));
  return r;
}

private static AVL balance(AVL t) {
  assert t != null;
  AVL l = t.left, r = t.right;
  int hl = height(l), hr = height(r);
  if (hl > hr + 1) {
    AVL ll = l.left, lr = l.right;
    if (height(ll) >= height(lr))
      return rotateRight(t);
    else {
      t.left = rotateLeft(t.left);
      return rotateRight(t);
    }
  } else if (hr > hl + 1) {
    AVL rl = r.left, rr = r.right;
    if (height(rr) >= height(rl))
      return rotateLeft(t);
    else {
      t.right = rotateRight(t.right);
      return rotateLeft(t);
    }
  } else {
    t.height = 1 + Math.max(hl, hr);
    return t;
  }
}
```

**Programme 15 — Structure d'ensemble réalisée avec un AVL**

```
class AVLSet {
  private AVL root;

  AVLSet() {
    this.root = null;
  }

  boolean isEmpty() {
    return this.root == null;
  }

  boolean contains(int x) {
    return AVL.contains(this.root, x);
  }

  void add(int x) {
    this.root = AVL.add(x, this.root);
  }

  void remove(int x) {
    this.root = AVL.remove(x, this.root);
  }
}
```

Le reste du code est immédiat ; il est donné programme 15 page 90.

**Exercice 40.** Ajouter à la classe `AVLSet` un champ privé `size` contenant le nombre d'éléments de l'ensemble et une méthode `int size()` qui en renvoie la valeur. Modifier les méthodes `add` et `remove` pour mettre à jour la valeur de ce champ. Il faudra traiter correctement le cas où l'élément ajouté par `add` est déjà dans l'ensemble et celui où l'élément supprimé par `remove` n'est pas dans l'ensemble. Solution □

**Exercice 41.** Ajouter à la classe `AVL` une méthode `LinkedList<Integer> toList(AVL t)` qui renvoie la liste ordonnée des éléments de l'arbre `t`. Solution □

**Exercice 42.** Ajouter à la classe `AVL` une méthode `AVL ofList(Queue<Integer> l)` qui prend en argument une file de $N$ entiers, supposée triée par ordre croissant, et renvoie un AVL contenant ces $N$ entiers, en temps $O(N)$. Indication : généraliser avec une méthode qui construit un arbre avec les $n$ premiers éléments de la file seulement. Solution □

**Exercice 43.** Déduire des deux exercices précédents des méthodes réalisant l'union, l'intersection et la différence ensembliste de deux AVL en temps $O(n+m)$, où $n$ et $m$ sont les nombres d'éléments de chaque AVL. Solution □

### 6.3.4 Code générique

Pour écrire une version générique des arbres binaires de recherche, par exemple des AVL donnés page 84, on paramètre le code par le type `E` des éléments :

```
class AVL<E> {
  E value;
  AVL<E> left, right;
  int height;
```

Cependant, cela ne suffit pas. Le code a besoin de pouvoir comparer les éléments entre eux, par exemple dans les méthodes `contains`, `add` et `remove`. Pour l'instant, nous avons utilisé une comparaison directe entre entiers, avec les opérateurs `==`, `<` et `>`. Pour comparer des éléments de type `E`, ce n'est plus possible. On va donc exiger que la classe `E` fournisse une méthode pour comparer deux éléments. Pour cela on utilise l'interface suivante :

```
  interface Comparable<K> {
    int compareTo(K k);
  }
```

Le signe de l'entier renvoyé par `compareTo` indique comment `this` se compare à `k`. Une telle interface fait déjà partie de la bibliothèque Java, dans `java.lang.Comparable<T>`. On va exiger que le paramètre `E` de la classe `AVL` implémente l'interface `Comparable<E>`, ce que l'on écrit ainsi :

```
class AVL<E extends Comparable<E>> {
  ...
```

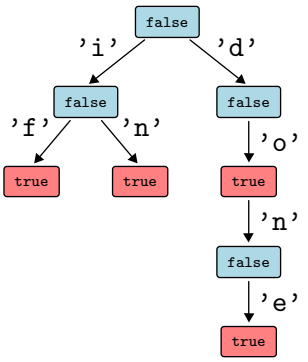
À l'intérieur de la classe `AVL`, on peut comparer deux éléments `x` et `y` en testant le signe de `x.compareTo(y)`. Pour écrire une méthode statique, on doit préciser de nouveau le paramètre de type et sa contrainte (voir page 12). Ainsi la méthode `contains` de la classe `AVL` s'écrit

```
static<E extends Comparable<E>> boolean contains(AVL<E> a, E x) {
  while (a != null) {
    int c = x.compareTo(a.value);
    if (c == 0) return true;
    a = (c < 0) ? a.left : a.right;
  }
  return false;
}
```

La bibliothèque Java propose des arbres binaires de recherche équilibrés (des arbres rouges et noirs en l'occurrence [5]), à savoir la classe `java.util.TreeSet<E>` pour des ensembles dont les éléments sont de type `E` et la classe `java.util.TreeMap<K, V>` pour des dictionnaires dont les clés sont de type `K` et les valeurs de type `V`. Là aussi, la hauteur des arbres est logarithmique.

## 6.4 Arbres de préfixes

On s'intéresse dans cette section à une autre structure d'arbre, pour représenter des ensembles de *mots* (ici du type `String` de Java). Dans ces arbres, chaque branche est étiquetée par une lettre et chaque nœud contient un booléen indiquant si la séquence de lettres menant de la racine de l'arbre à ce nœud est un mot appartenant à l'ensemble. Par exemple, l'arbre représentant l'ensemble de mots {`"if"`, `"in"`, `"do"`, `"done"`} est le suivant :




Un tel arbre est appelé un *arbre de préfixes*, plus connu sous le nom de *trie* en anglais. L'intérêt d'une telle structure de donnée est de borner le temps de recherche d'un élément dans un ensemble à la longueur du mot le plus long de cet ensemble, quelque soit le nombre de mots qu'il contient. Plus précisément, cette propriété est garantie seulement si toutes les feuilles d'un arbre de préfixes représentent bien un mot de l'ensemble, c'est-à-dire si elles contiennent toutes une valeur booléenne à vrai. Cette *bonne formation* des arbres de préfixes sera maintenue par toutes les opérations définies ci-dessous.

Écrivons une classe `Trie` pour représenter de tels arbres. On utilise la bibliothèque `HashMap` pour représenter le branchement à chaque nœud par une table de hachage :

```
class Trie {
  private boolean word;
  private HashMap<Character, Trie> branches;
  ...
```

Ainsi, dans l'exemple ci-dessus, le champ `branches` de la racine de l'arbre est une table de hachage contenant deux entrées, une associant le caractère `'i'` au sous-arbre de gauche, et une autre associant le caractère `'d'` au sous-arbre de droite.

L'arbre de préfixes vide est représenté par un arbre réduit à un unique nœud où le champ `word` vaut `false` et `branches` est un dictionnaire vide :

```
Trie() {
  this.word = false;
  this.branches = new HashMap<Character, Trie>();
}
```

**Recherche d'un élément.** Écrivons une méthode `contains` qui détermine si une chaîne `s` appartient à un arbre de préfixes.

```
boolean contains(String s) {
```

La recherche consiste à descendre dans l'arbre en suivant les lettres de `s`. On le fait ici à l'aide d'une boucle `for`, en se servant d'une variable `t` contenant le nœud de l'arbre où l'on se trouve à chaque instant.

```
  Trie t = this;
  for (int i = 0; i < s.length(); i++) { // invariant t != null
    t = t.branches.get(s.charAt(i));
```

Si `t.branches` ne contient pas d'entrée pour le `i`-ième caractère de `s`, alors la méthode `get` ci-dessus renverra `null`. Dans ce cas, on conclut immédiatement que `s` n'appartient pas à `t`.

```
    if (t == null) return false;
  }
```

Dans le cas contraire, on passe au caractère suivant. Si on sort de la boucle, c'est qu'on est parvenu jusqu'au dernier caractère de `s`. Il suffit alors de renvoyer le booléen présent dans le nœud qui a été atteint.

```
  return t.word;
}
```

**Insertion d'un élément.** L'insertion d'un mot `s` dans un arbre de préfixes consiste à descendre le long de la branche étiquetée par les lettres de `s`, de manière similaire au parcours effectué pour la recherche. C'est cependant légèrement plus subtil, car il faut éventuellement créer de nouvelles branches dans l'arbre pendant la descente. Comme pour la recherche, on procède à la descente avec une boucle `for` parcourant les caractères du mot et une variable `t` contenant le sous-arbre courant.

```
void add(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) { // invariant t != null
    char c = s.charAt(i);
```

Avant de suivre le branchement donné par le caractère `c`, on sauvegarde la table de hachage dans une variable locale `b`.

```
HashMap<Character, Trie> b = t.branches;
t = b.get(c);
```

Si la branche correspondant au caractère `c` n'existe pas, la méthode `get` aura renvoyé `null`. Il faut alors ajouter une nouvelle branche. On le fait en créant un nouvel arbre d'une part, et en l'ajoutant à la table `b` d'autre part.

```
if (t == null) {
  t = new Trie();
  b.put(c, t);
}
```

On peut alors passer au caractère suivant, car on a assuré que `t` n'est pas `null`. Une fois sorti de la boucle, il ne reste plus qu'à positionner le booléen à `true` pour indiquer la présence du mot `s`.

```
}
t.word = true;
```

Si le mot `s` était déjà présent dans l'arbre, cette affectation est sans effet.

Le code complet est donné programme 16 page 95. La structure d'arbre de préfixes peut être généralisée à toute valeur pouvant être vue comme une suite de lettres, quelle que soit la nature de ces lettres. C'est le cas par exemple pour une liste. C'est aussi le cas d'un entier, si on voit ses bits comme formant un mot avec les lettres `0` et `1`. Dans ce dernier cas, on parle d'arbre de Patricia [14].

**Exercice 44.** Ajouter à la classe `Trie` une méthode `void remove(String s)` qui supprime l'occurrence de la chaîne `s`, si elle existe.

Solution □

**Exercice 45.** La méthode `remove` de l'exercice précédent peut conduire à des branches vides, *i.e.* ne contenant plus aucun mot, ce qui dégrade les performances de la recherche. Modifier la méthode `remove` pour qu'elle supprime les branches devenues vides. Il s'agit donc de maintenir l'invariant qu'un champ `branches` ne contient jamais une entrée vers un arbre ne contenant aucun mot. Indication : on pourra procéder récursivement et se servir de la méthode suivante

```
boolean isEmpty() {
  return !this.word && this.branches.isEmpty();
}
```

qui teste si un arbre ne contient aucun mot — à supposer que l'invariant ci-dessus est effectivement maintenu, bien entendu. Solution □

**Exercice 46.** Optimiser la structure de `Trie` pour que le champ `branches` des feuilles de l'arbre ne contiennent pas une table de hachage vide, mais plutôt la valeur `null`.

Solution □

**Programme 16 — Arbres de préfixes**

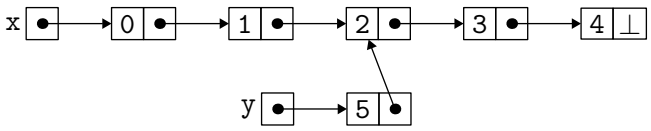
```
class Trie {

  private boolean word;
  private HashMap<Character, Trie> branches;

  Trie() {
    this.word = false;
    this.branches = new HashMap<Character, Trie>();
  }

  boolean contains(String s) {
    Trie t = this;
    for (int i = 0; i < s.length(); i++) { // invariant t != null
      t = t.branches.get(s.charAt(i));
      if (t == null) return false;
    }
    return t.word;
  }

  void add(String s) {
    Trie t = this;
    for (int i = 0; i < s.length(); i++) { // invariant t != null
      char c = s.charAt(i);
      Map<Character, Trie> b = t.branches;
      t = b.get(c);
      if (t == null) {
        t = new Trie();
        b.put(c, t);
      }
    }
    t.word = true;
  }
}
```

# 7 Structures de données immuables

Ce chapitre montre l'intérêt de structures de données qui ne peuvent plus être modifiées une fois construites. Bien que la bibliothèque standard de Java en contienne peu, un exemple notable est la classe `String` des chaînes de caractères.

## 7.1 Principe et intérêt

Le caractère modifiable d'une structure de données n'est pas sans danger. Prenons l'exemple des listes simplement chaînées (section 4.1). Supposons par exemple que nous soyons parvenu à la situation suivante après plusieurs étapes de constructions de listes :




La variable `x` pointe sur une liste $0 \to 1 \to 2 \to 3 \to 4$ et la variable `y` pointe sur une liste $5 \to 2 \to 3 \to 4$ mais, détail important, elles *partagent* une partie de leurs éléments, à savoir la queue de liste $2 \to 3 \to 4$. Si maintenant le détenteur de la variable `x` décide de modifier le contenu de la liste désignée par `x`, par exemple pour remplacer la valeur 3 par 17, ou encore d'en modifier la structure, pour faire reboucler l'élément 4 vers l'élément 2, alors ce changement affectera la liste `y` également. À cet égard, c'est la même situation d'*alias* que nous avons déjà évoquée avec les tableaux (voir page 19).

Il existe de nombreuses situations dans lesquelles on sait pertinemment qu'une liste ne sera pas modifiée après sa création. On peut donc chercher à le garantir, comme un invariant du programme. Une solution consisterait à faire des champs `element` et `next` des champs privés et à n'exporter que des méthodes qui ne modifient pas les listes. Une solution encore meilleure consiste à déclarer les champs `element` et `next` comme `final`. Ceci implique qu'ils ne peuvent plus être modifiés au-delà du constructeur (le compilateur le vérifie), ce qui est exactement ce que nous recherchons.

```
class Singly {
  final int element;
  final Singly next;
  ...
}
```

Dès lors, une situation de partage telle que celle illustrée ci-dessus n'est plus problématique. En effet, la portion de liste partagée ne peut être modifiée ni par le détenteur de `x` ni par celui de `y`, et donc son partage ne présente plus aucun danger. Au contraire, il permet même une économie d'espace.

Lorsqu'une structure de données ne fournit aucune opération permettant d'en modifier le contenu, on parle de structure de données *immuable* [1]. Un style de programmation qui ne fait usage que de structures de données immuables est dit *purement applicatif*. L'un des intérêts de ce style de programmation est une diminution significative du risque d'erreurs dans les programmes. En particulier, il devient beaucoup plus facile de raisonner sur le code, en utilisant le raisonnement mathématique usuel, sans avoir à se soucier constamment de l'*état* des structures de données. Un autre avantage est la possibilité d'un *partage* significatif entre différentes structures de données, et une possible économie substantiel de mémoire.

Ceci n'est évidemment pas limité aux listes. On peut ainsi garantir le caractère immuable d'un arbre binaire de recherche en ajoutant simplement le qualificatif `final` à ses trois champs :

```
class BST {
  final int value;
  final BST left, right;
}
```

Certaines méthodes, comme `getMin` ou `contains`, restent inchangées car elles ne font que consulter la structure de l'arbre, sans chercher à la modifier. D'autres méthodes, en revanche, doivent être écrites différemment sur les arbres immuables. Ainsi, la méthode `add` qui insère un élément dans un arbre binaire de recherche renvoie maintenant un *nouvel arbre*. On peut l'écrire ainsi :

```
static BST add(BST b, int x) {
  if (b == null)
    return new BST(null, x, null);
  if (x < b.value)
    return new BST(add(b.left, x), b.value, b.right);
  if (x > b.value)
    return new BST(b.left, b.value, add(b.right, x));
  return b; // x déjà dans b
}
```

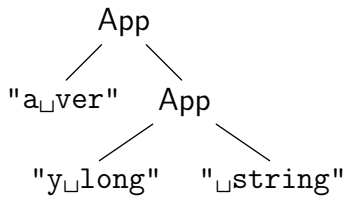
De nouveaux nœuds sont donc construits tout le long du chemin parcouru par l'insertion (là où auparavant on se contentait d'affectations). Cela peut paraître coûteux, mais pour un arbre équilibré, tel qu'un AVL, il n'y a qu'un nombre logarithmique de nœuds le long de ce chemin. L'insertion a donc un coût logarithmique en espace dans ce cas.

**Exercice 47.** La structure d'arbres de préfixes (section 6.4) est une structure de données modifiable. Expliquer pourquoi, à la différence des arbres binaires, on ne peut pas en faire facilement une structure immuable en ajoutant simplement le qualificatif `final` sur les deux champs `word` et `branches`. Solution □

1. Un concept plus général est celui de structure de données *persistante*, où les modifications peuvent exister à l'intérieur de la structure mais ne sont pas observables depuis l'extérieur.

## 7.2 Exemple : structure de corde

On présente ici un exemple de structure immuable, les cordes. Il s'agit d'une structure pour représenter de grandes chaînes de caractères efficacement, et permettre notamment des opérations de concaténation et d'extraction de sous-chaînes sans impliquer de copies. La structure de corde s'appuie sur une idée très simple : une corde n'est rien d'autre qu'un arbre binaire dont les feuilles sont des chaînes (usuelles) de caractères et dont les nœuds internes sont vus comme des concaténations. Ainsi l'arbre




est une des multiples façons de représenter la chaîne `"a very long string"`. Deux considérations nous poussent à raffiner légèrement l'idée ci-dessus. D'une part, de nombreux algorithmes auront besoin d'un accès efficace à la longueur d'une corde, notamment pour décider de descendre dans le sous-arbre gauche ou dans le sous-arbre droit d'un nœud `App`. Il est donc souhaitable d'ajouter la taille de la corde comme une décoration de chaque nœud interne. D'autre part, il est important de pouvoir partager des sous-chaînes entre les cordes elles-mêmes et avec les chaînes usuelles qui ont été utilisées pour les construire. Dès lors, plutôt que d'utiliser une chaîne complète dans chaque feuille, on va stocker plus d'information pour désigner un fragment d'une chaîne Java, par exemple sous la forme de deux entiers indiquant un indice et une longueur. Pour représenter de tels arbres, on pourrait imaginer la classe suivante

```
class Rope {
  final int length;
  final String word;       // feuille
  final Rope left, right;  // noeud interne
  ...
}
```

où le champ `length` est utilisé systématiquement, le suivant dans le cas d'une feuille uniquement et les deux derniers dans le cas d'un nœud interne uniquement. Cette représentation a tout de même le défaut d'être inutilement gourmande : des champs sont systématiquement gâchés dans chaque objet. Nous allons adopter une représentation plus subtile, en tirant parti de l'héritage de classes fourni par Java.

On commence par écrire une classe `Rope` représentant une corde quelconque, c'est-à-dire aussi bien une feuille qu'un nœud interne. On y stocke la longueur de la corde, puisque c'est là l'information commune aux deux types de nœuds.

```
abstract class Rope {
  final int length;
  Rope(int length) { this.length = length; }
}
```

Cette classe est déclarée abstraite, ce qui signifie qu'on ne peut pas construire d'objet de cette classe. Elle va nous servir de *type* pour les cordes, mais les *objets* représentant

effectivement les cordes vont appartenir à deux sous-classes de `Rope`, représentant respectivement les feuilles et les nœuds internes. On les définit ainsi :

```
class Str extends Rope {
  final String str;
}
class App extends Rope {
  final Rope left, right;
}
```

Par le principe de l'héritage, un objet de la classe `Str` a donc deux champs, à savoir `length` et `str`, et un objet de la classe `App` a trois champs, à savoir `length`, `left` et `right`. Les classes `Str` et `App` ne sont pas abstraites et on les utilisera justement pour construire des cordes. Commençons par le code des constructeurs. Pour la classe `Str`, l'argument est une chaîne de caractères.

```
  Str(String str) {
    super(str.length());
    this.str = str;
  }
```

La première ligne est un appel explicite au constructeur de la super-classe, c'est-à-dire au constructeur de la classe `Rope`[2]. On est obligé de procéder ainsi, car le champ `length` de `Rope` est déclaré `final` et doit donc être initialisé dans un constructeur de la classe `Rope`. Pour la classe `App`, c'est légèrement plus subtil, car on *calcule* la longueur de la corde comme la somme des longueurs de ses deux morceaux :

```
  App(Rope left, Rope right) {
    super(left.length + right.length);
    this.left = left;
    this.right = right;
  }
```

Avec ces constructeurs, on peut déjà construire des cordes. La corde donnée en exemple plus haut peut être construite avec

```
  Rope r = new App(new Str("a ver"),
                   new App(new Str("y long"),
                           new Str(" string")));
```

**Accès à un caractère.** Écrivons maintenant une méthode `char get(int i)` qui renvoie le `i`-ième caractère d'une corde. On la *déclare* dans la classe `Rope`, car on veut pouvoir accéder au `i`-ième caractère d'une corde sans connaître sa nature. Ainsi, on veut pouvoir écrire `r.get(3)` avec `r` de type `Rope` comme dans l'exemple ci-dessus. Mais on ne peut pas *définir* `get` dans la classe `Rope`. Aussi on la déclare comme une méthode abstraite.

2. De manière générale, le code d'un constructeur commence toujours par l'appel au code d'un autre constructeur : soit il s'agit d'un appel implicite au constructeur de la super-classe (comme si on avait écrit `super()`) ; soit il s'agit d'un appel explicite à un constructeur de la même classe, avec `this(...)`, ou de la super-classe, avec `super(...)`.

```
abstract char get(int i);
```

Pour que le code soit maintenant accepté par le compilateur, il faut définir la méthode `get` dans les deux sous-classes `Str` et `App`. Dans la classe `Str`, c'est immédiat.

```
char get(int i) {
  return this.str.charAt(i);
}
```

Dans la classe `App`, il faut déterminer si le caractère `i` se trouve dans la sous-chaîne de gauche ou de droite et appeler `get` récursivement.

```
char get(int i) {
  return (i < this.left.length) ?
         this.left.get(i) : this.right.get(i - this.left.length);
}
```

Toute la subtilité de la programmation orientée objet se trouve ici : on ne connaît pas la nature de `this.left` et `this.right` et pour autant on peut appeler leur méthode `get`. Le bon morceau de code sera appelé, par la vertu de l'appel dynamique de méthode.

**Exercice 48.** Modifier le code des méthodes `get` pour qu'il vérifie que `i` désigne bien une position valide dans la corde. Dans le cas contraire, lever une exception. Solution □

**Extraction de sous-chaîne.** On ajoute maintenant une méthode pour extraire une sous-corde. Comme pour `get`, on commence par la déclarer abstraite dans la classe `Rope` :

```
abstract Rope sub(int begin, int end);
```

Puis on la définit dans chacune des sous-classes. Dans la classe `Str`, c'est immédiat :

```
Rope sub(int begin, int end) {
  return new Str(this.str.substring(begin, end));
}
```

Dans la classe `App`, c'est plus subtil. En effet, la sous-corde peut se retrouver soit entièrement dans la corde de gauche, soit entièrement dans la corde de droite, soit à cheval sur les deux. On commence par calculer combien de caractères se trouvent dans la partie droite :

```
Rope sub(int begin, int end) {
  int endr = end - this.left.length;
```

Si cette quantité est négative ou nulle, c'est que le résultat se trouve tout entier dans la corde `this.left`.

```
  if (endr <= 0)
    return this.left.sub(begin, end);
```

Sinon, on détermine si au contraire le résultat se trouve tout entier dans la corde `this.right`.

```
int beginr = begin - this.left.length;
if (beginr >= 0)
  return this.right.sub(beginr, endr);
```

Dans le dernier cas, le résultat est la concaténation d'une portion de `this.left` et d'une portion de `this.right`, récursivement calculées avec `sub`.

```
return new App(this.left.sub(begin, this.left.length),
               this.right.sub(0, endr));
```

**Exercice 49.** Modifier le code des méthodes `sub` pour qu'il vérifie que `begin` et `end` désignent bien une portion valide de la corde. Dans le cas contraire, lever une exception. Solution □

**Exercice 50.** Ajouter des qualificatifs appropriés (`private`, `protected`) sur les différents champs des classes `Rope`, `Str` et `App`. Solution □

**Exercice 51.** Ajouter une méthode `String toString()` qui renvoie la chaîne Java définie par une corde. On prendra soin de le faire efficacement à l'aide d'un `StringBuffer`. Solution □

**Exercice 52.** Modifier la méthode `sub` pour qu'elle renvoie directement `this` lorsque `begin` et `end` désigne la corde toute entière. Quel est l'intérêt ? Solution □

**Exercice 53.** Pour améliorer l'efficacité des cordes, on peut utiliser l'idée suivante : dès que l'on cherche à concaténer deux cordes dont la somme des longueurs ne dépasse pas une constante donnée (par exemple 256 caractères) alors on construit directement un nœud de type `Str` plutôt qu'un nœud `App`. Écrire une méthode `Rope append(Rope r)` qui concatène deux cordes (`this` et `r`) en utilisant cette idée. On pourra réutiliser la méthode `toString` de l'exercice précédent. Solution □

### Programme 17 — Cordes

```
abstract class Rope {
  int length;
  Rope(int length) { this.length = length; }
  abstract char get(int i);
  abstract Rope sub(int begin, int end);
}

class Str extends Rope {
  String str;
  Str(String str) {
    super(str.length());
    this.str = str;
  }
  char get(int i) {
    return this.str.charAt(i);
  }
  Rope sub(int begin, int end) {
    return new Str(this.str.substring(begin, end));
  }
}

class App extends Rope {
  Rope left, right;
  App(Rope left, Rope right) {
    super(left.length + right.length);
    this.left = left;
    this.right = right;
  }
  char get(int i) {
    return (i < this.left.length) ?
           this.left.get(i) : this.right.get(i - this.left.length);
  }
  Rope sub(int begin, int end) {
    int endr = end - this.left.length;
    if (endr <= 0)
      return this.left.sub(begin, end);
    int beginr = begin - this.left.length;
    if (beginr >= 0)
      return this.right.sub(beginr, endr);
    return new App(this.left.sub(begin, this.left.length),
        this.right.sub(0, endr));
  }
}
```

# 8 Files de priorité

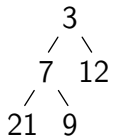
Dans le chapitre 4 nous avons vu comment la structure de liste chaînée permettait de réaliser facilement une structure de file. Dans ce chapitre, nous considérons maintenant des files dans lesquelles les éléments se voient associer des priorités. Dans une telle file, dite *files de priorité* (en anglais *priority queue*), les éléments sortent dans l'ordre fixé par leur priorité et non plus dans l'ordre d'arrivée. L'interface que l'on cherche à définir va donc ressembler à quelque chose comme

```
boolean isEmpty();
int size();
void add(int x);
int getMin();
void removeMin();
```

Dans cette interface, la notion de minimalité coïncide avec la notion de plus grande priorité. Contrairement aux files, on préfère distinguer l'accès au premier élément et sa suppression, par deux opérations distinctes, pour des raisons d'efficacité qui seront expliquées plus loin. Ainsi, la méthode `getMin` renvoie l'élément le plus prioritaire de la file et la méthode `removeMin` le supprime. On trouvera des applications des files de priorités plus loin dans les chapitres 13 et 14.

## 8.1 Structure de tas

Pour réaliser une file de priorité, il faut recourir à une structure de données plus complexe que pour une simple file. Une solution consiste à organiser les éléments sous la forme d'un *tas* (*heap* en anglais). Un tas est un arbre binaire où, à chaque nœud, l'élément stocké est plus prioritaire que les deux éléments situés immédiatement au-dessous. Ainsi, un tas contenant les éléments $\{3, 7, 9, 12, 21\}$, ordonnés par petitesse, peut prendre la forme suivante :



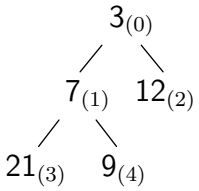

(8.1)

On note qu'il existe d'autres tas contenant ces mêmes éléments. Par définition, l'élément le plus prioritaire est situé à la racine et on peut donc y accéder en temps constant. Les

deux sections suivantes proposent deux façons différentes de représenter un tel tas.

## 8.2 Représentation dans un tableau

Dans cette section, on choisit de représenter une file de priorité par un tas dont la propriété supplémentaire est d'être un arbre binaire complet, c'est-à-dire un arbre binaire où tous les niveaux sont remplis, sauf peut-être le dernier, qui est alors partiellement rempli à gauche. Nous pourrions réutiliser ce qui a été introduit au chapitre précédent pour représenter un tel arbre. Mais il se trouve qu'un arbre binaire complet peut être facilement représenté dans un tableau. L'idée consiste à numéroter les nœuds de l'arbre de haut en bas et de gauche à droite, à partir de 0. Par exemple, le résultat de cette numérotation sur le tas (8.1) donne l'étiquetage




Cette numérotation permet de représenter le tas dans un tableau. Ainsi, le tas ci-dessus correspond au tableau à 5 éléments suivant :

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 3 | 7 | 12 | 21 | 9 |

De manière générale, la racine de l'arbre occupe la case d'indice 0 et les racines des deux sous-arbres du nœud stocké à la case $i$ sont stockées respectivement aux cases $2i + 1$ et $2i + 2$. Inversement, le père du nœud $i$ est stocké en $\lfloor (i - 1)/2 \rfloor$.

De cette structure de tas, on déduit les différentes opérations de la file de priorité de la manière suivante. Le plus petit élément est situé à la racine de l'arbre, c'est-à-dire à l'indice 0 du tableau. On y accède donc en temps constant. Pour ajouter un nouvel élément dans un tas, on le place tout en bas à droite du tas et on le fait remonter à sa place. Pour supprimer le plus petit élément, on le remplace par l'élément situé tout en bas à droite du tas, que l'on fait alors descendre à sa place. Ces deux opérations sont décrites en détail dans les deux sections suivantes. Ce que l'on peut déjà comprendre, c'est que leur coût est proportionnel à la hauteur de l'arbre. Un arbre binaire complet ayant une hauteur logarithmique, l'ajout et le retrait dans un tas ont donc un coût $O(\log n)$ où $n$ est le nombre d'éléments dans le tas.

Pour mettre en œuvre cette structure de tas, il reste un petit problème. On ne connaît pas *a priori* la taille de la file de priorité. On pourrait fixer à l'avance une taille maximale pour la file de priorité mais une solution plus élégante consiste à utiliser un tableau redimensionnable. De tels tableaux sont présentés dans le chapitre 3 et on va donc réutiliser ici la classe `ResizableArray` présentée plus haut. Un tas n'est donc rien d'autre qu'un objet encapsulant un tableau redimensionnable (ici dans un champ appelé `elts`) :

```
class Heap {
  private ResizableArray elts;
```

Le constructeur se contente d'allouer un nouveau tableau redimensionnable, qui ne contient initialement aucun élément :

```
Heap() {
  this.elts = new ResizableArray(0);
}
```

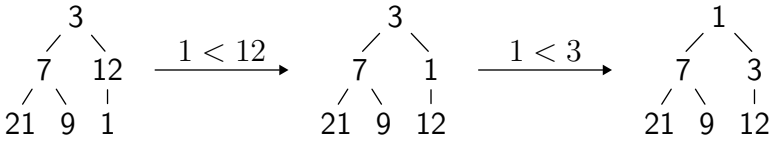
Le nombre d'éléments contenus dans le tas est exactement celui du tableau redimensionnable, d'où un code immédiat pour les deux méthodes `size` et `isEmpty` :

```
int size() {
  return this.elts.size();
}
boolean isEmpty() {
  return this.elts.size() == 0;
}
```

La méthode `getMin` renvoie la racine du tas, si elle existe, et lève une exception sinon. Comme expliqué ci-dessus, la racine du tas est stockée à l'indice 0.

```
int getMin() {
  if (this.elts.size() == 0)
    throw new NoSuchElementException();
  return this.elts.get(0);
}
```

**Insertion d'un élément.** L'insertion d'un élément `x` dans un tas consiste à étendre le tableau d'une case, à y mettre la valeur `x`, puis à faire « remonter » `x` jusqu'à la bonne position. Pour cela, on utilise l'algorithme suivant : tant que `x` est plus petit que son père, c'est-à-dire la valeur située immédiatement au dessus dans l'arbre, on échange leurs deux valeurs et on recommence. Par exemple, l'ajout de 1 dans le tas (8.1) est réalisé en trois étapes :




On commence donc par écrire une méthode récursive `moveUp(int x, int i)` qui insère un élément `x` dans le tas, en partant de la position `i`. Cette méthode suppose que l'arbre de racine `i` obtenu en plaçant `x` en `i` est un tas. La méthode `moveUp` considère tout d'abord le cas où `i` vaut 0, c'est-à-dire où on est arrivé à la racine. Il suffit alors d'insérer `x` à la position `i`.

```
private void moveUp(int x, int i) {
  if (i == 0) {
    this.elts.set(i, x);
```

S'il s'agit en revanche d'un nœud interne, on calcule l'indice `fi` du père de `i` et la valeur `y` stockée dans ce nœud.

```
    } else {
      int fi = (i - 1) / 2;
      int y = this.elts.get(fi);
```

Si `y` est supérieur à `x`, il s'agit de faire remonter `x` en descendant la valeur `y` à la place `i` puis en appelant récursivement `moveUp` à partir de `fi`.

```
      if (y > x) {
        this.elts.set(i, y);
        moveUp(x, fi);
```

Si en revanche `y` est inférieur ou égal à `x`, alors `x` a atteint sa place définitive et il suffit de l'y affecter.

```
      } else
        this.elts.set(i, x);
```

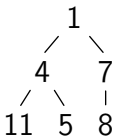
Ceci achève le code de `moveUp`. La méthode `add` procède alors en deux temps. Elle augmente la taille du tableau d'une unité, en ajoutant une case à la fin du tableau, puis appelle la méthode `moveUp` à partir de cette case.

```
  void add(int x) {
    int n = this.elts.size();
    this.elts.setSize(n + 1);
    moveUp(x, n);
  }
```

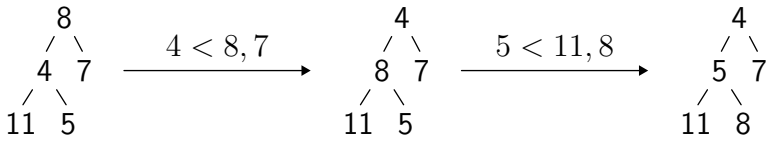
On note que `add` préserve bien la structure de tas. D'une part, on a étendu le tableau d'une unité vers la droite, ce qui revient à ajouter un élément en bas à droite de l'arbre, et on conserve donc la structure d'arbre binaire complet. D'autre part, l'invariant de `moveUp` est bien respecté, car on démarre avec un arbre de racine `n` réduit à un élément, qui est donc trivialement un tas. Comme expliqué plus haut, la méthode `add` a une complexité $O(\log n)$ où $n$ est le nombre d'éléments de la file de priorité.

**Exercice 54.** Réécrire la méthode `moveUp` à l'aide d'une boucle `while`. Solution □

**Suppression du plus petit élément.** Supprimer le plus petit élément d'un tas est légèrement plus délicat que d'insérer un nouvel élément. La raison en est qu'il s'agit de supprimer la racine de l'arbre et qu'il faut donc trouver par quel élément la remplacer. L'idée consiste à choisir l'élément tout en bas à droite du tas, c'est-à-dire l'élément occupant la dernière case du tableau, comme candidat, puis à le faire descendre dans le tas jusqu'à sa place, un peu comme on a fait monter le nouvel élément lors de l'insertion. Supposons par exemple que l'on veuille supprimer le plus petit élément du tas suivant :



On remplace la racine, c'est-à-dire 1, par l'élément tout en bas à droite, c'est-à-dire 8. Puis on fait descendre 8 jusqu'à ce qu'il atteigne sa place. Pour cela, on compare 8 avec les racines $a$ et $b$ des deux sous-arbres. Si $a$ et $b$ sont tous les deux plus grands que 8, la descente est terminée. Sinon, on échange 8 avec le plus petit des deux nœuds $a$ et $b$, et on continue la descente. Sur l'exemple, 8 est successivement échangé avec 4 et 5 :




Écrivons une méthode récursive `moveDown(int x, int i)` qui réalise la descente d'un élément `x` à sa place, en partant de l'indice `i`.

```
private void moveDown(int x, int i) {
  int n = this.elts.size();
```

On commence par déterminer l'indice `j` du plus petit des deux fils du nœud `i`. Il faut soigneusement tenir compte du fait que ces deux nœuds n'existent peut-être pas.

```
  int j = 2 * i + 1;
  if (j + 1 < n && this.elts.get(j + 1) < this.elts.get(j))
    j++;
```

Si le nœud `j` existe, et qu'il contient une valeur plus petite que `x`, alors `x` doit descendre. On fait donc remonter la valeur située à l'indice `j`, à la position `i`, puis on procède récursivement à partir de l'indice `j`, pour poursuivre la descente.

```
  if (j < n && this.elts.get(j) < x) {
    this.elts.set(i, this.elts.get(j));
    moveDown(x, j);
```

Sinon, c'est que la valeur `x` a terminé sa descente. Il suffit donc de l'affecter à la position `i`.

```
  } else
    this.elts.set(i, x);
```

Ceci achève le code de la méthode `moveDown`. La méthode `removeMin` de suppression du plus petit élément d'un tas s'en déduit alors facilement. On commence par traiter le cas pathologique d'un tas vide.

```
void removeMin() {
  int n = this.elts.size() - 1;
  if (n < 0) throw new NoSuchElementException();
```

Puis on extrait la valeur `x` située tout en bas à droite du tas, c'est-à-dire à la dernière position du tableau, avant de diminuer la taille du tableau d'une unité, puis d'appeler la méthode `moveDown` pour placer `x` à sa place, en partant de la racine du tas, c'est-à-dire de la position 0.

```
  int x = this.elts.get(n);
  this.elts.setSize(n);
  if (n > 0) moveDown(x, 0);
}
```

La structure est bien préservée. D'une part, on a supprimé l'élément situé tout en bas à droite du tas et on conserve donc une structure d'arbre binaire complet. D'autre part, `moveDown` assure que la structure de tas est bien rétablie. Comme expliqué plus haut, la méthode `removeMin` a une complexité $O(\log n)$ où $n$ est le nombre d'éléments de la file de priorité. Le code complet de la classe `Heap` est donné programme 18 page 111.

**Exercice 55.** On peut utiliser la structure de tas pour réaliser un tri efficace très facilement, appelé *tri par tas* (en anglais *heapsort*). L'idée est la suivante : on insère tous les éléments à trier dans un tas, puis on les ressort successivement avec les méthodes `getMin` et `removeMin`. Écrire une méthode `void sort(int[] a)` pour trier un tableau en utilisant cet algorithme. Quel est la complexité de ce tri ? (Le tri par tas est décrit en détail section 13.4.) Solution □

## 8.3 Représentation comme un arbre

Cette section présente une autre représentation de la structure de tas, avec la particularité de proposer une fonction efficace de fusion de deux tas. Les tas sont ici directement représentés par des arbres binaires. Contrairement aux AVL, aucune information de nature à assurer l'équilibrage n'est stockée dans les nœuds. Nous verrons plus loin que ces tas offrent néanmoins une complexité amortie logarithmique. On parle de tas *auto-équilibrés*. En anglais, ces tas s'appellent des *skew heaps*.

On introduit une classe `SkewHeap`, qui encapsule un arbre binaire (le tas) et son nombre d'éléments. On réutilise la classe `Tree` de la section 6.1.

```
class SkewHeap {
  private Tree root;
  private int size;
```

On maintiendra l'invariant que le champ `size` contient toujours le nombre d'éléments de l'arbre stocké dans le champ `root`. Le constructeur est immédiat. On rappelle que l'arbre vide est représenté par `null`. Ce n'est pas gênant car le champ `root` est un champ privé de la classe `SkewHeap`.

```
  SkewHeap() {
    this.root = null;
    this.size = 0;
  }
```

Les méthodes `isEmpty` et `size` sont également immédiates. On note qu'elles s'exécutent en temps constant.

```
  boolean isEmpty() {
    return this.size == 0;
  }
  int size() {
    return this.size;
  }
```

**Programme 18 — Structure de tas (dans un tableau)**

```
class Heap {
  private ResizableArray elts;
  Heap() { this.elts = new ResizableArray(0); }
  int size() { return this.elts.size(); }
  boolean isEmpty() { return this.elts.size() == 0; }
  private void moveUp(int x, int i) {
    if (i == 0) {
      this.elts.set(i, x);
    } else {
      int fi = (i - 1) / 2;
      int y = this.elts.get(fi);
      if (y > x) {
        this.elts.set(i, y);
        moveUp(x, fi);
      } else
        this.elts.set(i, x);
    }
  }
  void add(int x) {
    int n = this.elts.size();
    this.elts.setSize(n + 1);
    moveUp(x, n);
  }
  int getMin() {
    if (this.elts.size() == 0) throw new NoSuchElementException();
    return this.elts.get(0);
  }
  private void moveDown(int x, int i) {
    int n = this.elts.size();
    int j = 2 * i + 1;
    if (j + 1 < n && this.elts.get(j + 1) < this.elts.get(j))
      j++;
    if (j < n && this.elts.get(j) < x) {
      this.elts.set(i, this.elts.get(j));
      moveDown(x, j);
    } else
      this.elts.set(i, x);
  }
  void removeMin() {
    int n = this.elts.size() - 1;
    if (n < 0) throw new NoSuchElementException();
    int x = this.elts.get(n);
    this.elts.setSize(n);
    if (n > 0) moveDown(x, 0);
  }
}
```

La méthode `isEmpty` pourrait tout aussi bien tester si `this.root` est `null`. Enfin la méthode `getMin` renvoie le plus petit élément, c'est-à-dire la racine du tas. On prend cependant soin de tester que l'arbre est non vide.

```
int getMin() {
  if (this.isEmpty()) throw new NoSuchElementException();
  return this.root.value;
}
```

**Opération de fusion.** Toute la subtilité de ces tas auto-équilibrés tient dans une méthode `merge` qui fusionne deux tas. On l'écrit comme une méthode statique et privée qui prend en arguments deux arbres `t1` et `t2`, supposés être des tas. Le résultat renvoyé est la racine de l'arbre obtenu.

```
private static Tree merge(Tree t1, Tree t2) {
```

Si l'un des deux tas est vide, c'est immédiat.

```
  if (t1 == null) return t2;
  if (t2 == null) return t1;
```

Si en revanche aucun des tas n'est vide, on construit le tas résultant de la fusion de la manière suivante. Sa racine est clairement la plus petite des deux racines de `t1` et `t2`. Supposons que la racine de `t1` soit la plus petite.

```
  if (t1.value <= t2.value)
```

La racine de `t1` sera la racine du résultat. On doit maintenant déterminer ses deux sous-arbres. Il y a plusieurs possibilités, obtenues en appelant récursivement `merge` sur deux des trois arbres `t1.left`, `t1.right` et `t2` et en choisissant de mettre le résultat comme sous-arbre gauche ou droit. Parmi toutes ces possibilités, on choisit celle qui échange les deux sous-arbres de `t1`, de manière à assurer l'auto-équilibrage. Ainsi, `t1.right` prend la place de `t1.left` et est fusionné avec `t2`.

```
    Tree l1 = t1.left;
    t1.left = merge(t1.right, t2);
    t1.right = l1;
    return t1;
```

L'autre situation, où la racine de `t2` est la plus petite, est symétrique.

```
  } else {
    Tree l2 = t2.left;
    t2.left = merge(t2.right, t1);
    t2.right = l2;
    return t2;
  }
}
```

Ceci achève la méthode `merge`.

**Autres opérations.** De cette opération `merge` on déduit facilement les méthodes `add` et `removeMin`. En effet, pour ajouter un nouvel élément `x` au tas, il suffit de fusionner ce dernier avec un arbre réduit à l'élément `x`, sans oublier de mettre à jour le champ `size`.

```
void add(int x) {
  this.root = merge(this.root, new Tree(null, x, null));
  this.size++;
}
```

Pour supprimer le plus petit élément, c'est-à-dire la racine du tas, il suffit de fusionner les deux sous-arbres gauche et droit. On commence par tester si le tas est effectivement non vide.

```
int removeMin() {
  if (this.isEmpty()) throw new NoSuchElementException();
```

Le cas échéant, on conserve sa racine dans une variable `res` (pour la renvoyer comme résultat) et on fusionne les deux sous-arbres avec la méthode `merge`.

```
  int res = this.root.value;
  this.root = merge(this.root.left, this.root.right);
```

Enfin, on met à jour le champ `size` et on renvoie le plus petit élément `res`.

```
  this.size--;
  return res;
}
```

Le code complet de la classe `SkewHeap` est donné programme 19 page 114.

**Exercice 56.** Ajouter à la classe `SkewHeap` une méthode `void merge(SkewHeap that)` qui ajoute au tas `this` le contenu du tas `that`. Solution □

**Complexité.** On peut montrer que l'insertion successive de $n$ éléments en partant d'un tas vide a un coût total $O(n \log n)$. On peut donc considérer que chaque insertion a un coût amorti $\log n$. Il s'agit uniquement d'un coût amorti car il se peut néanmoins qu'une insertion particulière ait un coût plus grand, de l'ordre de $O(n)$ dans le pire des cas. On peut montrer également que la suppression successive des $n$ éléments d'un tas, avec une application répétée de `removeMin`, a un coût total $O(n \log n)$. On peut donc considérer de même que chaque suppression a un coût amorti $\log n$. Là encore, il s'agit d'un coût amorti, le pire des cas d'une suppression particulière pouvant être linéaire.

## 8.4 Code générique

Pour réaliser une version générique des files de priorité, on procède comme pour le code générique des AVL (section 6.3.4), avec un paramètre de type `E` sur lequel on exige une méthode de comparaison. On écrit donc quelque chose comme

```
class Heap<E extends Comparable<E>> {
  ...
```

**Programme 19 — Structure de tas (arbre auto-équilibré)**

```
class SkewHeap {
  private Tree root;
  private int size; // nombre de noeuds de root

  SkewHeap() {
    this.root = null;
    this.size = 0;
  }

  boolean isEmpty() { return this.size == 0; }
  int size() { return this.size; }

  int getMin() {
    if (this.isEmpty()) throw new NoSuchElementException();
    return this.root.value;
  }

  private static Tree merge(Tree t1, Tree t2) {
    if (t1 == null) return t2;
    if (t2 == null) return t1;
    if (t1.value <= t2.value) {
      Tree l1 = t1.left;
      t1.left = merge(t1.right, t2);
      t1.right = l1;
      return t1;
    } else {
      Tree l2 = t2.left;
      t2.left = merge(t2.right, t1);
      t2.right = l2;
      return t2;
    }
  }

  void add(int x) {
    this.root = merge(this.root, new Tree(null, x, null));
    this.size++;
  }

  int removeMin() {
    if (this.isEmpty()) throw new NoSuchElementException();
    int res = this.root.value;
    this.root = merge(this.root.left, this.root.right);
    this.size--;
    return res;
  }
}
```

S'il s'agit d'une représentation dans un tableau, on utilise par exemple les tableaux redimensionnables génériques de la section 3.4.5 (ou plus simplement la classe `Vector<E>` de la bibliothèque Java). S'il s'agit d'arbres binaires, on utilise des arbres génériques, comme au chapitre 6. Le reste du code est alors facilement adapté. Lorsqu'il s'agit de comparer deux éléments `x` et `y`, on n'écrit plus `x < y` mais `x.compareTo(y) < 0`.

La bibliothèque Java propose une telle structure de données générique dans la classe `java.util.PriorityQueue<E>`. Si la classe `E` implémente l'interface `Comparable<E>`, leur méthode `compareTo` est utilisée pour comparer les éléments. Dans le cas contraire, l'utilisateur peut fournir un comparateur au moment de la création de la file de priorité, sous la forme d'un objet qui implémente l'interface `java.util.Comparator<T>` :

```
interface Comparator<T> {
  int compare(T x, T y);
}
```

C'est alors la méthode `compare` de ce comparateur qui est utilisée pour ordonner les éléments.

On obtient le plus petit élément (`peek`) et le nombre d'éléments (`size`) en temps constant. Les opérations d'ajout d'un élément (`add`) et de retrait du plus petit élément (`poll`) sont en $O(\log N)$ si la file de priorité contient $N$ éléments (en considérant que les comparaisons d'éléments se font en temps constant).

**Exercice 57.** Expliquer comment utiliser la bibliothèque `PriorityQueue` pour calculer les $K$ plus grandes valeurs parmi $N$. Donner la complexité de cette méthode. Solution □

# 9 Classes disjointes

Ce chapitre présente une structure de données pour le problème des classes disjointes, connue sous le nom de *union-find*. Ce problème consiste à maintenir dans une structure de données une partition d'un ensemble fini, c'est-à-dire un découpage en sous-ensembles disjoints que l'on appelle des « classes ». On souhaite pouvoir déterminer si deux éléments appartiennent à la même classe et réunir deux classes en une seule. Ce sont ces deux opérations qui ont donné le nom de structure *union-find*.

## 9.1 Principe

Sans perte de généralité, on suppose que l'ensemble à partitionner est celui des $n$ entiers $\{0, 1, \ldots, n-1\}$. On cherche à construire une classe `UnionFind` avec l'interface suivante :

```
class UnionFind {
        UnionFind(int n)
    int find(int i)
   void union(int i, int j)
}
```

Le constructeur `UnionFind`($n$) construit une nouvelle partition de $\{0, 1, \ldots, n-1\}$ où chaque élément forme une classe à lui tout seul. L'opération `find`($i$) détermine la classe de l'élément $i$, sous la forme d'un entier considéré comme l'unique représentant de cette classe. En particulier, on détermine si deux éléments sont dans la même classe en comparant les résultats donnés par `find` pour chacun. Enfin, l'opération `union`($i, j$) réunit les deux classes des éléments $i$ et $j$, la structure de données étant modifiée en place.

L'idée principale est de lier entre eux les éléments d'une même classe. Dans chaque classe, ces liaisons forment des chemins qui mènent tous à un unique représentant, qui est le seul élément lié à lui-même. La figure 9.1 montre un exemple où l'ensemble $\{0, 1, \ldots, 7\}$ est partitionné en deux classes dont les représentants sont respectivement 3 et 4. Il est possible de représenter une telle structure en utilisant des nœuds alloués en mémoire individuellement (voir exercice 61). Cependant, il est plus simple et souvent plus efficace d'utiliser un tableau qui lie chaque entier à un autre entier de la même classe. Ces liaisons mènent toujours au représentant de la classe, qui est associé à sa propre valeur dans le tableau. Ainsi, la partition de la figure 9.1 est représentée par le tableau suivant :

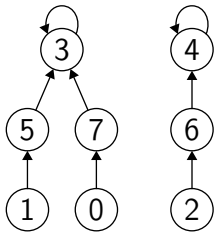



FIGURE 9.1 – Une partition en deux classes de $\{0, 1, \ldots, 7\}$

| 0 | 1 | | | $\cdots$ | | | 7 |
|---|---|---|---|---|---|---|---|
| 7 | 5 | 6 | 3 | 4 | 3 | 4 | 3 |

L'opération `find` se contente de suivre les liaisons jusqu'à trouver le représentant. L'opération `union` commence par trouver les représentants des deux éléments, puis lie l'un des deux représentants à l'autre. Afin d'atteindre de bonnes performances, on apporte deux améliorations. La première consiste à *compresser les chemins* pendant la recherche effectuée par `find` : cela consiste à lier directement au représentant tous les éléments trouvés sur le chemin parcouru pour l'atteindre. La seconde consiste à maintenir, pour chaque représentant, une valeur appelée *rang* qui représente la longueur maximale que pourrait avoir un chemin dans cette classe. Cette information est stockée dans un second tableau et est utilisée par la fonction `union` pour choisir entre les deux représentants possibles d'une union. Cette structure de données est attribuée à McIlroy et Morris [2] et sa complexité a été analysée par Tarjan [16].

## 9.2 Réalisation

Décrivons maintenant le code de la structure *union-find*, dans une classe `UnionFind` dont une instance représente une partition. Cette classe contient deux tableaux privés : `link` qui contient les liaisons et `rank` qui contient le rang de chaque classe.

```
class UnionFind {
  private int[] link;
  private int[] rank;
```

L'information contenue dans `rank` n'est significative que pour des éléments $i$ qui sont des représentants, c'est-à-dire pour lesquels `link[`$i$`]` $= i$. Initialement, chaque élément forme une classe à lui tout seul, c'est-à-dire est son propre représentant, et le rang de chaque classe vaut 0.

```
  UnionFind(int n) {
    if (n < 0) throw new IllegalArgumentException();
    this.link = new int[n];
    for (int i = 0; i < n; i++) this.link[i] = i;
    this.rank = new int[n];
  }
```

La fonction `find` calcule le représentant d'un élément `i`. Elle s'écrit naturellement comme une fonction récursive. On commence par calculer l'élément `p` lié à `i` dans le tableau `link`. Si c'est `i` lui-même, on a terminé et `i` est le représentant de la classe.

```
int find(int i) {
  if (i < 0 || i >= this.link.length)
    throw new ArrayIndexOutOfBoundsException(i);
  int p = this.link[i];
  if (p == i) return i;
```

Sinon, on calcule récursivement le représentant `r` de la classe avec l'appel `find(p)`. Avant de renvoyer `r`, on réalise la compression de chemins, c'est-à-dire qu'on lie directement `i` à `r`.

```
  int r = this.find(p);
  this.link[i] = r;
  return r;
}
```

Ainsi, la prochaine fois que l'on appellera `find` sur `i`, on trouvera `r` directement. Bien entendu, il se trouvait peut-être que `i` était déjà lié à `r` et dans ce cas l'affectation est sans effet.

L'opération `union` regroupe en une seule les classes de deux éléments `i` et `j`. On commence par calculer leurs représentants respectifs `ri` et `rj`. S'ils sont égaux, il n'y a rien à faire.

```
void union(int i, int j) {
  int ri = this.find(i);
  int rj = this.find(j);
  if (ri == rj) return; // déjà dans la même classe
```

Sinon, on compare les rangs des deux classes. Si celui de `ri` est strictement plus petit que celui de `rj`, on fait de `rj` le représentant de l'union, c'est-à-dire qu'on lie `ri` à `rj`.

```
  if (this.rank[ri] < this.rank[rj])
    this.link[ri] = rj;
```

Le rang n'a pas besoin d'être mis à jour pour cette nouvelle classe. En effet, seuls les chemins de l'ancienne classe de `ri` ont vu leur longueur augmentée d'une unité et cette nouvelle longueur n'excède pas le rang de `rj`. Si en revanche c'est le rang de `rj` qui est le plus petit, on procède symétriquement.

```
  else {
    this.link[rj] = ri;
```

Dans le cas où les deux classes ont le même rang, l'information de rang doit alors être mise à jour, car la longueur du plus long chemin est susceptible d'augmenter d'une unité.

```
    if (this.rank[ri] == this.rank[rj])
      this.rank[ri]++;
  }
}
```

Il est important de noter que la fonction `union` utilise la fonction `find` et réalise donc des compressions de chemin, même dans le cas où il s'avère que `i` et `j` sont déjà dans la même classe. Le code complet de la classe `UnionFind` est donné programme 20 page 121. On a ajouté une méthode `sameClass` qui détermine si `i` et `j` sont dans la même classe d'équivalence en comparant les représentants renvoyés par `find`.

**Complexité.** Il est facile de montrer que la complexité de `find` et `union` est en $O(\log n)$. Pour cela, on commence par montrer l'invariant suivant : une classe de rang $k$

— a des chemins de longueur au plus $k$ ;

— possède au moins $2^k$ élément.

On le prouve par récurrence sur le nombre d'opérations `union`. C'est vrai initialement car toutes les classes ont le rang 0. Supposons maintenant l'invariant et effectuons une nouvelle opération `union`, donnant une classe de rang $k$. Il y a deux cas de figure. La nouvelle classe peut être la réunion d'une classe de rang $k$ et d'une classe de rang $k' < k$. Dans ce cas, les nouveaux chemins ont une longueur maximale $k' + 1 \leq k$. Et par ailleurs la première des deux classes contenait déjà au moins $2^k$ éléments par hypothèse. L'autre cas de figure correspond à la réunion de deux classes de rang $k - 1$. Dans ce cas, les nouveaux chemins ont une longueur au plus $k - 1 + 1 = k$ et la nouvelle classe contient au moins $2^{k-1} + 2^{k-1} = 2^k$ éléments. Ceci achève la preuve de notre invariant.

Il est clair que la complexité de `find` est proportionnelle à la longueur du chemin parcouru. On déduit donc de l'invariant que `find` a une complexité au plus égale au rang de la classe. Or l'invariant implique également que toute classe a un rang au plus $\log n$. Dès lors, la complexité de `find` est $O(\log n)$. En particulier, on est assuré que `find` ne fera pas déborder la pile d'appels. La complexité de `union` est également $O(\log n)$, car `union` ne fait que deux appels à `find` puis un nombre constant d'opérations.

La complexité est en réalité bien meilleure. On peut montrer qu'une suite de $m$ opérations `find` et `union` réalisées sur une structure contenant $n$ éléments s'exécute en un temps total $O(m\,\alpha(n, m))$, où $\alpha$ est une fonction qui croît extrêmement lentement. Elle croît si lentement qu'on peut la considérer comme constante pour toute application pratique — vues les valeurs de $n$ et $m$ que les limites de mémoire et de temps nous autorisent à admettre — ce qui nous permet de supposer un temps amorti constant pour chaque opération. Cette analyse de complexité est subtile et dépasse largement le cadre de ce polycopié. On en trouvera une version détaillée dans *Introduction to Algorithms* [5, chap. 22].

**Exercice 58.** Expliquer comment on peut utiliser la structure *union-find* pour décider si une égalité est conséquence d'autres égalités :

$$x_1 = x_7 \wedge x_3 = x_8 \wedge \cdots \Rightarrow x_4 = x_{17} \ ?$$

Solution □

**Exercice 59.** Ajouter à la structure *union-find* une méthode `int numClasses()` donnant le nombre de classes distinctes. On s'efforcera de fournir cette valeur en temps constant, en maintenant la valeur comme un champ supplémentaire. Solution □

**Exercice 60.** Si les éléments ne sont pas des entiers consécutifs, on peut remplacer les deux tableaux `rank` et `link` par deux tables de hachage. Réécrire la classe `UnionFind` en utilisant cette idée. Solution □

**Programme 20 — Structure de classes disjointes (*union-find*)**

```
class UnionFind {

  private int[] link;
  private int[] rank;

  UnionFind(int n) {
    if (n < 0) throw new IllegalArgumentException();
    this.link = new int[n];
    for (int i = 0; i < n; i++) this.link[i] = i;
    this.rank = new int[n];
  }

  int find(int i) {
    if (i < 0 || i >= this.link.length)
      throw new ArrayIndexOutOfBoundsException(i);
    int p = this.link[i];
    if (p == i) return i;
    int r = this.find(p);
    this.link[i] = r;
    return r;
  }

  void union(int i, int j) {
    int ri = this.find(i);
    int rj = this.find(j);
    if (ri == rj) return;
    if (this.rank[ri] < this.rank[rj])
      this.link[ri] = rj;
    else {
      this.link[rj] = ri;
      if (this.rank[ri] == this.rank[rj])
        this.rank[ri]++;
    }
  }

  boolean sameClass(int i, int j) {
    return this.find(i) == this.find(j);
  }
}
```

**Exercice 61.** Une autre solution pour réaliser la structure *union-find* consiste à ne pas utiliser de tableaux, mais à représenter directement chaque élément comme un objet contenant deux champs `rank` et `link`. Si `E` désigne le type des éléments, on peut définir la classe générique suivante :

```
class Elt<E> {
  private E value;
  private Elt<E> link;
  private int rank;
  ...
}
```

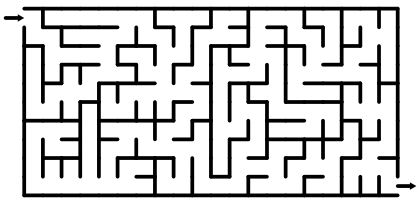
Il n'est plus nécessaire de maintenir d'information globale sur la structure *union-find*, car chaque élément contient toute l'information nécessaire. (Attention cependant à ne pas partager une valeur de type `Elt<E>` entre plusieurs partitions.) L'interface de la classe `Elt` est la suivante :

```
        Elt(E x)
Elt<E> find()
  void union(Elt<E> e)
```

Le constructeur `Elt(E x)` construit une classe contenant un unique élément, de valeur `x`. On pourra choisir la convention qu'un pointeur `link` est `null` lorsqu'il s'agit d'un représentant. Écrire ce constructeur ainsi que les méthodes `find` et `union`.

Solution □

**Exercice 62.** On peut utiliser la structure *union-find* pour construire efficacement un labyrinthe parfait, c'est-à-dire un labyrinthe où il existe un chemin et un seul entre deux cases. Voici un exemple de tel labyrinthe :



On procède de la manière suivante. On crée une structure *union-find* dont les éléments sont les différentes cases. L'idée est que deux cases sont dans la même classe si et seulement si elles sont reliées par un chemin. Initialement, toutes les cases du labyrinthe sont séparées les unes des autres par des murs. Puis on considère toutes les paires de cases adjacentes (verticalement et horizontalement) dans un ordre aléatoire. Pour chaque paire $(c_1, c_2)$ on compare les classes des cases $c_1$ et $c_2$. Si elles sont identiques, on ne fait rien. Sinon, on supprime le mur qui sépare $c_1$ et $c_2$ et on réunit les deux classes avec `union`. Écrire un code qui construit un labyrinthe selon cette méthode.

Indication : pour parcourir toutes les paires de cases adjacentes dans un ordre aléatoire, le plus simple est de construire un tableau contenant toutes ces paires, puis de le mélanger aléatoirement en utilisant le *mélange de Knuth* (exercice 5 page 35).

Justifier que, à l'issue de la construction, chaque case est reliée à toute autre case par un unique chemin.

Solution □

# Troisième partie

# Algorithmes élémentaires

# 10 Arithmétique

Ce chapitre regroupe un certain nombre d'algorithmes fondamentaux ayant pour thème commun l'arithmétique.

## 10.1 Algorithme d'Euclide

Le plus célèbre des algorithmes est très certainement l'algorithme d'Euclide. Il permet de calculer le plus grand diviseur commun de deux entiers (dit « pgcd », en anglais *gcd* pour *greatest common divisor*). Étant donnés deux entiers positifs ou nuls $u$ et $v$, l'algorithme d'Euclide répète le calcul

$$(u, v) \leftarrow (v, u \bmod v) \tag{10.1}$$

jusqu'à ce que $v$ soit nul, et renvoie alors la valeur de $u$, qui est le pgcd des valeurs initiales de $u$ et $v$. Le code est immédiat ; il est donné programme 21 page 125. La terminaison de cet algorithme est assurée par la décroissance stricte de `v` et le fait que `v` reste par ailleurs positif ou nul. On a en effet l'invariant de boucle évident $u, v \geq 0$. La correction de l'algorithme repose sur le fait que l'instruction (10.2) préserve le plus grand diviseur commun. Quand on parvient à $v = 0$ on renvoie alors $u$ c'est-à-dire $\gcd(u, 0)$, qui est donc le pgcd des valeurs initiales de $u$ et $v$.

La complexité de l'algorithme d'Euclide est donnée par le théorème de Lamé, qui stipule que si l'algorithme effectue $s$ itérations pour $u > v > 0$ alors $u \geq F_{s+1}$ et $v \geq F_s$

**Programme 21 — Algorithme d'Euclide**

```
static int gcd(int u, int v) {
  while (v != 0) {
    int tmp = v;
    v = u % v;
    u = tmp;
  }
  return u;
}
```

**Programme 22 — Algorithme d'Euclide étendu**

```
static int[] extendedGcd(int u0, int v0) {
  int[] u = { 1, 0, u0 }, v = { 0, 1, v0 };
  while (v[2] != 0) {
    int q = u[2] / v[2];
    int[] t = { u[0] - q * v[0], u[1] - q * v[1], u[2] - q * v[2] };
    u = v;
    v = t;
  }
  return u;
}
```

où $(F_n)$ est la suite de Fibonacci. On en déduit facilement que $s = O(\log u)$. Une analyse détaillée est donnée dans *The Art of Computer Programming* [11, sec. 4.5.3].

**Exercice 63.** L'algorithme d'Euclide que l'on vient de présenter suppose $u, v \geq 0$. Si $u$ ou $v$ est négatif, il peut renvoyer un résultat négatif (l'opération `%` de Java renvoie une valeur du même signe que son premier argument). Modifier le code de la méthode `gcd` pour qu'elle renvoie toujours un résultat positif ou nul, quel que soit le signe de ses arguments. Solution □

**Exercice 64.** Dans quel cas la méthode `gcd` peut-elle renvoyer zéro ? Solution □

**Exercice 65.** Le résultat de complexité donné ci-dessus suppose $u > v$. Montrer que, dans le cas général, la complexité est $O(\log(\max(u, v)))$. Solution □

**Algorithme d'Euclide étendu.** On peut facilement modifier algorithme d'Euclide pour qu'il calcule également les coefficients de Bézout, c'est-à-dire un triplet d'entiers $(r_0, r_1, r_2)$ tels que

$$r_0 u + r_1 v = r_2 = \gcd(u, v).$$

Pour cela, on ne travaille plus avec seulement deux entiers $u$ et $v$, mais avec deux triplets d'entiers $\vec{u} = (u_0, u_1, u_2)$ et $\vec{v} = (v_0, v_1, v_2)$. Initialement, on prend $\vec{u} = (1, 0, u)$ et $\vec{v} = (0, 1, v)$. Puis, exactement comme avec l'algorithme d'Euclide, on répète l'instruction

$$(\vec{u}, \vec{v}) \leftarrow (\vec{v}, \vec{u} - q\vec{v}) \quad \text{avec } q = \lfloor u_2/v_2 \rfloor \tag{10.2}$$

jusqu'à ce que $v_2 = 0$, et on renvoie $\vec{u}$. On appelle cela l'algorithme d'Euclide étendu. Une traduction littérale de cet algorithme en Java, utilisant des tableaux pour représenter les vecteurs $\vec{u}$ et $\vec{v}$, est donnée programme 22 page 126. (L'exercice 66 en propose une écriture un peu plus efficace.) On se convainc facilement que la troisième composante du vecteur renvoyé est bien le pgcd de $u$ et $v$. En effet, si on se focalise sur le calcul de $u_2$ et $v_2$ uniquement, on retrouve les mêmes calculs que ceux effectués dans la méthode `gcd` avec les variables $u$ et $v$, à la seule différence que $u_2 \mod v_2$ est maintenant calculé par $u_2 - \lfloor u_2/v_2 \rfloor v_2$. Pour justifier la correction de l'algorithme d'Euclide étendu, on note que l'invariant suivant est maintenu :

$$\begin{aligned} u_0 u + u_1 v &= u_2 \\ v_0 u + v_1 v &= v_2 \end{aligned}$$

**Programme 23 — Exponentiation rapide**

```
static int exp(int x, int n) {
  if (n == 0) return 1;
  int r = exp(x * x, n / 2);
  return (n % 2 == 0) ? r : x * r;
}
```

En particulier, la première identité est exactement celle que l'on voulait pour le résultat. La complexité reste la même que pour la méthode `gcd`. Le nombre d'opérations effectuées à chaque tour de boucle est certes supérieur, mais il reste borné et le nombre d'itérations est exactement le même. La complexité est donc toujours $O(\log(\max(u, v)))$.

**Exercice 66.** Modifier la méthode `extendedGcd` pour qu'elle n'alloue pas de tableau intermédiaire. Solution □

**Exercice 67.** Soient $u, v, m$ trois entiers strictement positifs tels que $\gcd(v, m) = 1$. On appelle quotient de $u$ par $v$ modulo $m$ tout entier $w$ tel que $0 \leq w < m$ et $u \equiv vw \pmod{m}$. Écrire une méthode calculant le quotient de $u$ par $v$ modulo $m$. Solution □

## 10.2 Exponentiation rapide

L'algorithme d'exponentiation rapide (en anglais *exponentiation by squaring*) consiste à calculer $x^n$ en exploitant les identités suivantes :

$$\begin{cases} x^{2k} &= (x^2)^k \\ x^{2k+1} &= x(x^2)^k \end{cases}$$

Sa traduction en Java, pour $x$ de type `int`, est immédiate. On peut écrire par exemple le code donné programme 23 page 127. Il existe de multiples variantes. On peut par exemple faire un cas particulier pour `n` $= 1$ mais ce n'est pas vraiment utile. Voir aussi l'exercice 68. Quoiqu'il en soit, l'idée centrale reste la suivante : pour calculer $x^n$, cet algorithme effectue un nombre de multiplications proportionnel à $\log(n)$, ce qui est une amélioration significative par rapport à l'algorithme naïf qui effectue exactement $n-1$ multiplications. (On verra plus loin pourquoi on s'intéresse uniquement aux multiplications, et pas aux calculs faits par ailleurs sur $n$ lui-même.) On peut s'en convaincre aisément en montrant par récurrence sur $k$ que, si $2^{k-1} \leq n < 2^k$, alors la méthode `exp` effectue exactement $k$ appels récursifs. Chaque appel récursif à `exp` effectuant une ou deux multiplications, on en déduit le résultat ci-dessus.

Les applications de cet algorithme sont innombrables, car rien n'impose à $x$ d'être de type `int`. Dès lors qu'on dispose d'une unité et d'une opération associative, c'est-à-dire d'un monoïde $M$, alors on peut appliquer cet algorithme pour calculer $x^n$ avec $x \in M$ et $n \in \mathbb{N}$. Donnons un exemple. Les nombres de la suite de Fibonacci $(F_n)$ vérifient l'identité suivante :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}. \tag{10.3}$$

Autrement dit, on peut calculer $F_n$ en élevant une matrice $2\times 2$ à la puissance $n$. Avec l'algorithme d'exponentiation rapide, on peut le faire en $O(\log n)$ opérations arithmétiques, ce qui est une amélioration significative par rapport à un calcul direct en temps linéaire [1].

**Exercice 68.** Écrire une variante de la méthode `exp` qui repose sur les identités suivantes :

$$\begin{cases} x^{2k} &= (x^k)^2 \\ x^{2k+1} &= x(x^k)^2 \end{cases}$$

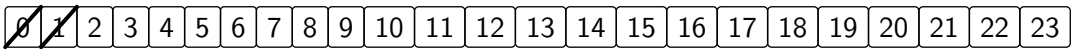
Y a-t-il une différence d'efficacité ? Solution □

**Exercice 69.** Écrire l'exponentiation rapide avec une boucle `while`. Solution □

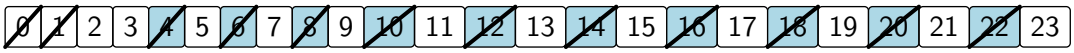
**Exercice 70.** Écrire un programme qui calcule $F_n$ en utilisant l'équation (10.3) et l'algorithme d'exponentiation rapide. On écrira une classe minimale pour des matrices $2 \times 2$ à coefficients dans `int`, avec la matrice identité, la multiplication et l'exponentiation rapide. Solution □

## 10.3 Crible d'Ératosthène

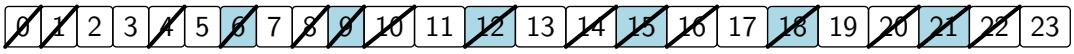
De nombreuses applications requièrent un test de primalité (l'entier $n$ est-il premier ?) ou le calcul exhaustif des nombres premiers jusqu'à un certain rang. Le crible d'Ératosthène est un algorithme qui détermine, pour un certain entier $N$, la primalité de tous les entiers $n \leq N$. Illustrons son fonctionnement avec $N = 23$. On écrit tous les entiers de 0 à $N$. On va éliminer progressivement tous les entiers qui ne sont pas premiers — d'où le nom de *crible*. Initialement, on se contente de dire que 0 et 1 ne sont pas premiers.




Puis on détermine le premier entier non encore éliminé. Il s'agit de 2. On élimine alors tous ses multiples, à savoir ici tous les entiers pairs supérieurs à 2.




Puis on recommence. Le prochain entier non éliminé est 3. On élimine donc à leur tour tous les multiples de 3.




On note que certains étaient déjà éliminés (les multiples de 6, en l'occurrence) mais ce n'est pas grave. Le prochain entier non éliminé est 5. Comme $5 \times 5 > 23$ le crible est terminé. En effet, tout multiple de 5, c'est-à-dire $k \times 5$, est soit déjà éliminé si $k < 5$, soit au-delà de 23 si $k \geq 5$. Les nombres premiers inférieurs ou égaux à $N$ sont alors tous les nombres qui n'ont pas été éliminés, c'est-à-dire ici 2, 3, 5, 7, 11, 13, 17, 19 et 23.

Écrivons une méthode `sieve` qui réalise le crible d'Ératosthène à l'aide d'un tableau de booléens, qui sera renvoyé au final.

1. Attention cependant à ne pas conclure hâtivement qu'on sait calculer $F_n$ pour de grandes valeurs de $n$. Les éléments de la suite de Fibonacci croissent en effet de manière exponentielle. Si on a recours à des entiers en précision arbitraire, le coût des opérations arithmétiques elles-mêmes doit être pris en compte, et la complexité ne sera pas $O(\log n)$. Et dans le cas contraire, on aura rapidement un débordement arithmétique.

**Programme 24 — Crible d'Ératosthène**

```
static boolean[] sieve(int max) {
  boolean[] prime = new boolean[max + 1];
  for (int i = 2; i <= max; i++)
    prime[i] = true;
  for (int n = 2; n * n <= max; n++)
    if (prime[n])
      for (int m = n * n; m <= max; m += n)
        prime[m] = false;
  return prime;
}
```

```
static boolean[] sieve(int max) {
  boolean[] prime = new boolean[max + 1];
```

Initialement, le tableau `prime` contient la valeur `false` dans toutes ses cases (voir page 20). On commence donc par écrire `true` dans toutes les cases d'indice $i \geq 2$.

```
  for (int i = 2; i <= max; i++)
    prime[i] = true;
```

La boucle principale du crible parcourt alors les entiers de 2 à $\lfloor\sqrt{\texttt{max}}\rfloor$ et teste à chaque fois leur primalité.

```
  for (int n = 2; n * n <= max; n++)
    if (prime[n])
```

Le cas échéant, elle élimine les multiples de `n`, à l'aide d'une seconde boucle. La même raison qui nous permet d'arrêter le crible dès que $\texttt{n} \times \texttt{n} > N$ nous permet de démarrer cette élimination à $\texttt{n} \times \texttt{n}$ (plutôt que $2\texttt{n}$), les multiples plus petits ayant déjà été éliminés.

```
      for (int m = n * n; m <= max; m += n)
        prime[m] = false;
```

Il ne reste plus qu'à renvoyer le tableau de booléens (des exercices plus bas proposent de renvoyer plutôt un tableau contenant les nombres premiers trouvés).

```
  return prime;
}
```

Le code complet est donné programme 24 page 129.

Évaluons la complexité du crible d'Ératosthène. La complexité en espace est clairement $O(N)$. La complexité en temps nous amène à considérer le coût de chaque itération de la boucle principale. S'il ne s'agit pas d'un nombre premier, le coût est constant (on ne fait rien). Mais lorsqu'il s'agit d'un nombre premier $p$, alors la boucle interne à un coût $\frac{N}{p}$ car on considère tous les multiples de $p$ (en fait, un peu moins car on commence l'itération à $p^2$, mais cela ne change pas l'asymptotique). Le coût total est donc

$$N + \sum_{p \leq N} \frac{N}{p}$$

où la somme est faite sur les nombres premiers. Un théorème d'Euler nous dit que $\sum_{p \leq N} \frac{1}{p} \sim \ln(\ln(N))$ d'où une complexité $N \ln(\ln(N))$ pour le crible d'Ératosthène.

**Exercice 71.** L'entier 2 étant le seul nombre premier pair, on peut optimiser le code de la méthode `sieve` en traitant à part le cas des nombres pairs et en progressant de 2 en 2 à partir de 3 dans la boucle principale. Mettre en œuvre cette idée. Solution □

**Exercice 72.** Le type `boolean[]` a une représentation mémoire un peu gourmande : chaque booléen y est en effet représenté par un octet, soit 8 bits là où un seul suffirait. La bibliothèque Java y remédie en proposant une classe `BitSet` où les tableaux de booléens sont représentés d'une manière plus compacte. Écrire une variante de la méthode `sieve` renvoyant un résultat de type `BitSet` plutôt que `boolean[]`. Solution □

**Exercice 73.** Écrire une méthode `int[] firstPrimesUpto(int max)` qui renvoie un tableau contenant tous les nombres premiers inférieurs ou égaux à `max`, dans l'ordre croissant.

Solution □

**Exercice 74.** Écrire une méthode `int[] firstNPrimes(int n)` qui renvoie un tableau contenant les `n` premiers nombres premiers. Indication : si $p_n$ désigne le $n$-ième nombre premier, on a l'inégalité $p_n < n \log n + n \log \log n$ dès que $n \geq 6$. Solution □

# 11 Programmation dynamique et mémoïsation

La programmation dynamique et la mémoïsation sont deux techniques très proches qui s'appuient sur l'idée naturelle suivante : ne pas recalculer deux fois la même chose. Illustrons-les avec l'exemple très simple du calcul de la suite de Fibonacci. On rappelle que cette suite d'entiers $(F_n)$ est définie par :

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

Écrire une méthode récursive qui réalise ce calcul en suivant cette définition est immédiat. (On calcule ici avec le type `long` car les nombres de Fibonacci deviennent rapidement très grands.)

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

Mais c'est aussi très naïf. Ici le problème n'est pas lié à un éventuel débordement de pile mais au fait que, lors du calcul de $F_n$, on recalcule de nombreuses fois les mêmes valeurs de $F_i$. Ainsi pour calculer ne serait-ce que $F_5$, on va calculer deux fois $F_3$ et trois fois $F_2$. De manière plus générale, on peut montrer que la méthode ci-dessus est de complexité exponentielle, en $O(\phi^n)$ où $\phi$ est le nombre d'or $\frac{1+\sqrt{5}}{2}$ (c'était l'objet de l'exercice 2). On peut l'observer empiriquement. Sur une machine de 2011, on observe qu'il faut 2 secondes pour calculer $F_{42}$, 3 secondes pour $F_{43}$, 5 secondes pour $F_{44}$, etc. On reconnaît là justement les nombres de la suite de Fibonacci. On extrapole qu'il faudrait 89 secondes pour calculer $F_{50}$ et ceci se vérifie à la demi seconde près !

## 11.1 Mémoïsation

Puisqu'on a compris qu'on calculait plusieurs fois la même chose, une idée naturelle consiste à stocker les résultats déjà calculés dans une table. Il s'agit donc d'une table

associant à certains entiers $i$ la valeur de $F_i$. Dès lors, on procède ainsi : pour calculer `fib(n)` on regarde si la table possède une entrée pour `n`. Le cas échéant, on renvoie la valeur correspondante. Sinon, on calcule `fib(n)`, toujours comme `fib(n-2)+fib(n-1)`, c'est-à-dire récursivement, puis on ajoute le résultat dans la table, avant de le renvoyer. Cette technique consistant à utiliser une table pour stocker les résultats déjà calculés s'appelle la *mémoïsation* (en anglais *memoization*, une terminologie forgée par le chercheur Donald Michie en 1968).

Mettons en œuvre cette idée dans une méthode `fibMemo`. On commence par introduire une table de hachage pour stocker les résultats déjà calculés

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

La méthode `fibMemo` a une structure identique à la méthode `fib`. En particulier, elle commence par traiter les cas de base $F_0$ et $F_1$.

```
static long fibMemo(int n) {
  if (n <= 1) return n;
```

On aurait pu ne pas faire de cas particulier en stockant les valeurs de $F_0$ et $F_1$ dans la table ; c'est affaire de style uniquement. Lorsque `n` $\geq 2$ on commence par regarder dans la table `memo` si la valeur de `fib(n)` ne s'y trouve pas déjà. Le cas échéant, on la renvoie.

```
  Long l = memo.get(n);
  if (l != null) return l;
```

On utilise ici le fait que la méthode `get` de la table de hachage renvoie la valeur `null` lorsqu'il n'y a pas d'entrée pour la clé donnée. Dans ce cas, justement, on calcule le résultat exactement comme pour la méthode `fib` c'est-à-dire avec deux appels récursifs.

```
  l = fibMemo(n - 2) + fibMemo(n - 1);
```

Puis on le stocke dans la table `memo`, avant de le renvoyer.

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

Ceci conclut la méthode `fibMemo`. Son efficacité est bien meilleure que celle de `fib`. Le calcul de $F_{50}$, par exemple, est devenu instantané (au lieu de 89 secondes avec `fib`). On peut montrer que la complexité de `fibMemo` est linéaire. Le calcul n'est pas complètement trivial mais, intuitivement, on comprend que le calcul de $F_n$ n'implique plus maintenant que le calcul des valeurs de $F_i$ pour $i \leq n$ une seule fois chacune. Le code complet de la méthode `fibMemo` est donné programme 25 page 133. On notera que la table `memo` est définie à *l'extérieur* de la méthode `fibMemo`, car elle doit être la même pour tous les appels récursifs.

**Programme 25 — Calcul de $F_n$ par mémoïsation et par programmation dynamique**

```
// mémoïsation
static HashMap<Integer, Long> memo = new HashMap<Integer, Long>();
static long fibMemo(int n) {
  if (n <= 1) return n;
  Long l = memo.get(n);
  if (l != null) return l;
  l = fibMemo(n - 2) + fibMemo(n - 1);
  memo.put(n, l);
  return l;
}

// programmation dynamique
static long fibDP(int n) {
  long[] f = new long[n + 1];
  f[1] = 1;
  for (int i = 2; i <= n; i++)
    f[i] = f[i - 2] + f[i - 1];
  return f[n];
}
```

## 11.2 Programmation dynamique

Il peut sembler inutilement coûteux d'utiliser une table de hachage pour mémoriser les calculs. En effet, on ne va stocker au final que les valeurs de $F_i$ pour $i \leq n$ et donc un simple tableau de taille $n+1$ suffit. Si on voulait réécrire la méthode `fibMemo` ci-dessus avec cette idée, on pourrait par exemple remplir le tableau initialement avec la valeur $-1$ dans chaque case pour signifier que la valeur n'a pas encore été calculée. Mais on peut aussi procéder différemment, en remplissant le tableau dans un certain ordre. En l'occurrence ici, on voit bien qu'il suffit de le remplir dans l'ordre croissant, car le calcul de $F_i$ nécessite le calcul des $F_j$ pour $j < i$. Cette technique consistant à utiliser une table et à la remplir progressivement avec les résultats des calculs intermédiaires s'appelle la *programmation dynamique* (en anglais *dynamic programming*, souvent abrégé DP).

Mettons en œuvre cette idée dans une méthode `fibDP`. On commence allouer un tableau `f` de taille `n + 1` destiné à contenir les valeurs des $F_i$.

```
static long fibDP(int n) {
  long[] f = new long[n + 1];
```

Ce tableau peut être alloué à *l'intérieur* de la méthode `fibDP` car celle-ci, à la différence de `fibMemo`, ne va pas être récursive. Puis on remplit les cases du tableau `f` de bas en haut, c'est-à-dire dans le sens des indices croissants, en faisant un cas particulier pour `f[1]`.

```
  f[1] = 1;
```

```
    for (int i = 2; i <= n; i++)
      f[i] = f[i - 2] + f[i - 1];
```

Une fois le tableau rempli, il ne reste plus qu'à renvoyer la valeur contenue dans sa dernière case.

```
    return f[n];
  }
```

Ceci conclut la méthode `fibDP`. Comme pour `fibMemo`, son efficacité est linéaire (ici cela se voit facilement) et le calcul de $F_{50}$, par exemple, est également instantané. Le code complet de la méthode `fibDP` est donné programme 25 page 133.

## 11.3 Comparaison

Le code des méthodes `fibMemo` et `fibDP` peut nous laisser penser que la programmation dynamique est plus simple à mettre en œuvre que la mémoïsation. Sur cet exemple, c'est vrai. Mais il faut comprendre que, pour écrire `fibDP`, nous avons exploité deux informations capitales : le fait de savoir qu'il fallait calculer les $F_i$ pour tous les $i \leq n$, et le fait de savoir qu'on pouvait les calculer dans l'ordre croissant. De manière générale, les entrées de la fonction à calculer ne sont pas nécessairement des indices consécutifs, ou ne sont même pas des entiers, et les dépendances entre les diverses valeurs à calculer ne sont pas nécessairement aussi simples. En pratique, la mémoïsation est plus simple à mettre en œuvre : il suffit en effet de rajouter quelques lignes pour consulter et remplir la table de hachage, sans modifier la structure de la méthode.

Il existe cependant quelques rares situations où la programmation dynamique peut être préférée à la mémoïsation. Prenons l'exemple du calcul des coefficients binomiaux $C(n,k)$ dont la définition récursive est la suivante :

$$\begin{cases} C(n,0) &= 1 \\ C(n,n) &= 1 \\ C(n,k) &= C(n-1,k-1) + C(n-1,k) \text{ pour } 0 < k < n. \end{cases}$$

On peut appliquer à cette définition aussi bien la mémoïsation que la programmation dynamique. Dans le premier cas, on aura une table de hachage indexée par le couple $(n,k)$, et dans le second cas, on aura une matrice indexée par $n$ et $k$. Cependant, si on cherche à calculer $C(n,k)$ pour $n = 2 \times 10^5$ et $k = 10^5$, alors il est probable qu'on va dépasser les capacités mémoire de la machine (sur une machine de bureau raisonnable), dans le premier cas en remplissant la table de hachage et dans le second cas en tentant d'allouer la matrice. La raison est que la complexité est ici en temps *et en espace* en $O(nk)$. Dans l'exemple ci-dessus, il faudrait stocker au minimum 15 milliards de résultats.

Pourtant, à y regarder de plus près, le calcul des $C(n,k)$ pour une certaine valeur de $n$ ne nécessite que les valeurs des $C(n-1,k)$. Dès lors on peut les calculer pour des valeurs de $n$ croissantes, sans qu'il soit utile de conserver toutes les valeurs calculées jusque là. On le visualise mieux en dessinant le triangle de Pascal

$$
\begin{array}{cccccc}
1 & & & & & \\
1 & 1 & & & & \\
1 & 2 & 1 & & & \\
1 & 3 & 3 & 1 & & \\
1 & 4 & 6 & 4 & 1 & \\
1 & 5 & 10 & 10 & 5 & 1 \\
\vdots & & & & &
\end{array}
$$

et en expliquant que l'on va le calculer ligne à ligne, en ne conservant à chaque fois que la ligne précédente pour le calcul de la ligne suivante. Mettons cette idée en œuvre dans une méthode `cnkSmartDP(int n, int k)`. On commence par allouer un tableau `row` de taille `n+1` qui va contenir une ligne du triangle de Pascal.

```
static long cnkSmartDP(int n, int k) {
  int[] row = new int[n+1];
```

(On pourrait se contenter d'un tableau de taille `k+1` ; voir l'exercice 75.) Pour l'instant, ce tableau ne contient que des valeurs nulles. On initialise sa toute première case avec 1.

```
  row[0] = 1;
```

Cette valeur ne bougera plus car la première colonne du triangle de Pascal ne contient que des 1. On écrit ensuite une boucle pour calculer la ligne `i` du triangle de Pascal :

```
  for (int i = 1; i <= n; i++)
```

Ce calcul va se faire en place dans le tableau `row`, sachant qu'il contient la ligne $\texttt{i} - 1$. Pour ne pas se marcher sur les pieds, on va calculer les nouvelles valeurs de la droite vers la gauche, car elles ne dépendent pas de valeurs situées plus à droite. On procède avec une seconde boucle :

```
    for (int j = i; j >= 1; j--)
      row[j] += row[j-1];
```

On exploite ici le fait que `row[i]` contient 0, ce qui nous dispense d'un cas particulier. On s'arrête à $\texttt{j} = 1$ car la valeur `row[0]` n'a pas besoin d'être modifiée, comme expliqué plus haut. Une fois qu'on est sorti de cette double boucle, le tableau `row` contient la ligne `n` du triangle de Pascal, et on n'a plus qu'à renvoyer `row[k]` :

```
  return row[k];
}
```

La complexité en temps reste $O(nk)$ mais la complexité en mémoire n'est plus que $O(n)$. Dans l'exemple donné plus haut, avec $n = 2 \times 10^5$ et $k = 10^5$, le résultat est instantané (on notera cependant qu'il provoque un débordement arithmétique ; voir l'exercice 76).

La conclusion de cette petite expérience est que la programmation dynamique peut parfois être plus avantageuse que la mémoïsation, car elle permet un contrôle plus fin des ressources mémoire. Mais dans les très nombreuses situations où ce contrôle n'est pas nécessaire, la mémoïsation est plus simple à mettre en œuvre.

**Exercice 75.** Modifier la méthode `cnkSmartDP` pour ne pas calculer les valeurs du triangle de Pascal au-delà de la colonne `k`. Solution □

**Exercice 76.** Modifier la méthode `cnkSmartDP` pour qu'elle renvoie un résultat de type `BigInteger`. Il s'agit là d'une classe de la bibliothèque Java représentant des entiers en précision arbitraire. Attention : la complexité n'est plus $O(nk)$ car les additions ne sont plus des opérations atomiques ; leur coût dépend de la taille des opérandes, qui grandit vite dans le triangle de Pascal. Solution □

**Exercice 77.** Modifier la méthode `fibDP` pour qu'elle n'utilise plus de tableau, mais seulement deux entiers. Solution □

## 11.4 Exemple : le compte est bon

*Le compte est bon* est une épreuve du jeu télévisé *Des chiffres et des lettres* où les candidats doivent construire un nombre cible, tiré au hasard entre 100 et 999, à partir de six nombres donnés et des quatre opérations élémentaires. Chacun des six nombres donnés au départ doit être utilisé au plus une fois. La soustraction n'est autorisée que pour donner un résultat positif et la division que lorsqu'elle est exacte. Si le nombre cible ne peut être atteint, on cherche l'entier atteignable le plus proche possible de la cible.

On peut avantageusement utiliser la programmation dynamique (ou la mémoïsation) pour résoudre ce problème. L'idée consiste à calculer les nombres atteignables à partir de tout sous-ensemble des nombres donnés au départ. Si on se donne l'ensemble de départ sous la forme d'un tableau d'entiers `int[] s`, on peut représenter un sous-ensemble de `s` par un simple entier compris entre 0 inclus et $2^{\texttt{s.length}}$ exclus, dont les bits à 1 indiquent la présence de l'élément correspondant. Si par exemple `s` contient six éléments, un sous-ensemble de `s` est représenté par un entier entre 0 et 63. Ainsi, l'entier 41 s'écrit en binaire $101001_2$ et représente donc les éléments `s[5]`, `s[3]` et `s[0]`. L'ensemble `s` tout entier est représenté par 63, c'est-à-dire $111111_2$, ou encore `(1 << s.length) - 1` en Java de manière générale.

On se donne un dictionnaire pour stocker les résultats de nos calculs. À chaque entier `u` représentant un sous-ensemble de `s`, on lui associe l'ensemble des nombres atteignables à partir de `u`. Cet ensemble est lui-même représenté par un dictionnaire, donnant pour chaque entier atteignable la chaîne indiquant sa construction. On déclare donc

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

On commence par remplir ce dictionnaire avec les sous-ensembles de `s` qui sont des singletons, représentés par les entiers de la forme $2^i$ avec $0 \leq i <$ `s.length`.

```
for (int i = 0; i < s.length; i++) {
  Map<Integer, String> m = new HashMap<>();
  m.put(s[i], "" + s[i]);
  res.put(1 << i, m);
}
```

On considère ensuite tout sous-ensemble `u` de `s`, en considérant tous les entiers de 3 à $2^{\texttt{s.length}} - 1$. On part de 3 car 2 est un singleton, que l'on vient de traiter. Si `u` est déjà dans `res`, c'est qu'il s'agit d'un singleton et on passe directement au sous-ensemble suivant. Sinon, on crée un nouveau dictionnaire `m` que l'on associe à `u`.

```
for (int u = 3; u < 1 << s.length; u++) {
  if (res.containsKey(u))
    continue;
  Map<Integer, String> m = new HashMap<>();
  res.put(u, m);
```

Il faut maintenant remplir ce dictionnaire `m` avec tous les entiers atteignables à partir de `u`. Pour cela, on considère tout sous-ensemble strict de `u`, noté `left`, et son complémentaire `u\left`, noté `right`.

```
  for (int left = 1; left < u; left++)
    if ((left & u) == left) { // left est un sous-ensemble de u
      int right = u & ~left;
```

Il est important de remarquer ici que `left` et `right` sont tous deux des entiers strictement plus petits que `u` et qu'on a donc déjà calculé les entiers atteignables à partir de `left` et de `right` puisqu'on procède par valeurs croissantes de `u`. Il ne reste plus qu'à ajouter tous les entiers que l'on peut construire à partir des éléments de `left` et de `right`. En particulier, on ajoute tous les éléments atteignables à partir de `left`, pour traduire le fait que les éléments de `u` ne sont pas nécessairement tous utilisés. Puis on combine les éléments atteignables à partir de `left` et `right` avec les quatre opérations élémentaires.

```
      m.putAll(res.get(left));
      for (int x: res.get(left).keySet()) {
        String ex = res.get(left).get(x);
        for (int y: res.get(right).keySet()) {
          String ey = res.get(right).get(y);
          m.put(x + y, "("+ex+"+"+ey+")");
          m.put(x * y, "("+ex+"*"+ey+")");
          if (x >= y) m.put(x - y, "("+ex+"-"+ey+")");
          if (y != 0 && x % y == 0) m.put(x / y, "("+ex+"/"+ey+")");
        }
      }
```

L'intégralité du programme est donnée programme 26 page 138. Avec ce code, il faut seulement 116 millisecondes pour déterminer les 8684 entiers que l'on peut former à partir de l'ensemble $\{2, 5, 7, 13, 17, 23\}$. On peut notamment construire l'entier 338 et notre dictionnaire donne la chaîne `"((((23+17)+7)+5)*13)/2)"` comme solution. Le plus petit entier que l'on ne peut pas construire est 1182.

**Exercice 78.** Le programme 26 reste-t-il correct si le tableau `s` contient des doublons, *i.e.*, s'il s'agit d'un multi-ensemble et non pas d'un ensemble ? Solution □

**Exercice 79.** Étendre le programme 26 pour résoudre le problème initial, c'est-à-dire, étant donné un nombre cible, déterminer comment le construire, le cas échéant, ou l'entier le plus proche que l'on puisse construire sinon. Solution □

**Exercice 80.** Expliquer comment modifier le programme 26 pour calculer, pour chaque sous-ensemble $U$ de $S$, les entiers que l'on peut obtenir en utilisant les nombres de $U$ *exactement une fois chacun* et non pas au plus une fois. Solution □

**Programme 26 — Le compte est bon**

```
// calcule tous les entiers que l'on peut construire avec
// tout sous-ensemble de s
// un sous-ensemble de s est un masque de {0,...,|s|-1}
// représenté par un entier entre 0 et 2^|s|-1
static Map<Integer, Map<Integer, String>> computeAll(int[] s) {
  Map<Integer, Map<Integer, String>> res = new HashMap<>();
  // on commence par les singletons
  for (int i = 0; i < s.length; i++) {
    Map<Integer, String> m = new HashMap<>();
    m.put(s[i], "" + s[i]);
    res.put(1 << i, m);
  }
  int n = 1 << s.length;
  // ensuite pour tout sous-ensemble u
  for (int u = 3; u < n; u++) {
    if (res.containsKey(u)) // déjà fait (u est un singleton)
      continue;
    Map<Integer, String> m = new HashMap<>();
    res.put(u, m);
    for (int left = 1; left < u; left++)
      if ((left & u) == left) { // left est un sous-ensemble de u
        int right = u & ~left;
        m.putAll(res.get(left));
        for (int x: res.get(left).keySet()) {
          String ex = res.get(left).get(x);
          for (int y: res.get(right).keySet()) {
            String ey = res.get(right).get(y);
            m.put(x + y, "("+ex+"+"+ey+")");
            m.put(x * y, "("+ex+"*"+ey+")");
            if (x >= y) m.put(x - y, "("+ex+"-"+ey+")");
            if (y != 0 && x % y == 0) m.put(x / y, "("+ex+"/"+ey+")");
          }
        }
      }
  }
  return res;
}
```

# 12 Rebroussement (*backtracking*)

La technique du *rebroussement* (en anglais *backtracking*) consiste à construire la solution d'un problème incrémentalement, en s'interrompant dès que l'on peut déterminer à coup sûr qu'une solution partielle ne pourra être complétée. Dès lors, on rebrousse chemin pour changer l'une des décisions prises précédemment. On peut s'arrêter dès qu'une solution est trouvée ou énumérer toutes les solutions. Le rebroussement est une technique que chacun a déjà utilisé empiriquement, pour résoudre des jeux logiques tels que les mots croisés par exemple. La technique du rebroussement est parfois appelée également *retour sur trace* dans la littérature.

En termes de programmation, le rebroussement est plus une méthode générale qu'un algorithme. Sa mise en œuvre va être différente pour chaque problème, les constantes étant seulement les idées générales d'exploration systématique et d'interruption prématurée. Dans ce chapitre, on illustre le principe du rebroussement sur deux problèmes très classiques, à savoir le Sudoku et le problème des $N$ reines.

## 12.1 Le problème du Sudoku

Le problème du Sudoku consiste à remplir une grille $9 \times 9$ en utilisant les chiffres de 1 à 9 en obéissant aux contraintes suivantes : chaque ligne, chaque colonne et chaque sous-groupe $3 \times 3$ doit contenir exactement une seule occurrence de chaque chiffre. Le problème est en général posé sous la forme d'une grille où certaines cases sont déjà remplies. Voici un exemple :

| | | | 3 | 1 | 6 | | 5 | 9 |
|---|---|---|---|---|---|---|---|---|
| | | 6 | | | | 8 | | 7 |
| | | | | | | 2 | | |
| | 5 | | | 3 | | | 9 | |
| 7 | 9 | | 6 | | 2 | | 1 | 8 |
| | 1 | | | 8 | | | 4 | |
| | | 8 | | | | | | |
| 3 | | 9 | | | | 6 | | |
| 5 | 6 | | 8 | 4 | 7 | | | |

Les sous-groupes $3 \times 3$ y sont délimités par des traits gras. On se propose d'utiliser la technique du rebroussement pour résoudre ce problème.

On choisit de représenter la grille par un simple tableau de taille 81 contenant des valeurs entre 0 et 9, la valeur 0 représentant une case encore vide.

```
class Sudoku {
  private int[] grid = new int[81];
```

Si $i$ (resp. $j$) est un numéro de ligne (resp. de colonne) compris entre 0 et 8, la case $(i, j)$ de la grille est identifiée à l'indice de tableau $9i+j$. On se donne trois méthodes pour calculer respectivement le numéro de ligne, de colonne et de sous-groupe de la case représentée par l'indice `c`.

```
  int row(int c) { return c / 9; }
  int col(int c) { return c % 9; }
  int group(int c) { return 3 * (row(c) / 3) + col(c) / 3; }
```

Il est en particulier très facile d'en déduire si deux cases `c1` et `c2` appartiennent à la même colonne, la même ligne ou le même sous-groupe.

```
  boolean sameZone(int c1, int c2) {
    return row(c1) == row(c2)
        || col(c1) == col(c2)
        || group(c1) == group(c2);
  }
```

Pour interrompre prématurément notre recherche d'une solution, on écrit une méthode booléenne `check` qui vérifie si la case `p` contient une valeur en conflit avec une autre case. Pour cela, on parcourt toutes les cases

```
  boolean check(int p) {
    for (int c = 0; c < 81; c++)
```

et, pour chacune, on teste l'existence d'un conflit avec la case `p`. Le cas échéant, on le signale immédiatement.

```
      if (c != p && sameZone(p, c) && this.grid[p] == this.grid[c])
        return false;
```

Si en revanche on parvient à la fin de boucle, on signale l'absence de conflit.

```
    return true;
  }
```

L'algorithme de rebroussement proprement dit est écrit sous la forme d'une méthode récursive `solve()`. Cette méthode cherche une solution pour le problème se trouvant actuellement dans le tableau `grid`. Elle renvoie le booléen `true` dès qu'elle a trouvé une solution et cette solution est contenue dans le tableau `grid`. Si en revanche aucune solution n'existe, elle renvoie `false` et le contenu de `grid` est laissé inchangé.

```
  boolean solve() {
```

La méthode `solve` cherche la première case vide de la grille, par un simple parcours de toutes les cases.

```
    for (int c = 0; c < 81; c++)
      if (this.grid[c] == 0) {
```

Pour cette case, on va essayer successivement toutes les valeurs `v` possibles.

```
        for (int v = 1; v < 10; v++) {
          this.grid[c] = v;
```

La valeur `v` étant affectée à la case `c`, on teste l'absence de conflit avec la méthode `check`. Le cas échéant, on rappelle `solve` récursivement pour continuer la résolution du problème. Si `solve` trouve une solution, on termine immédiatement en signalant le succès de la recherche.

```
          if (check(c) && solve())
            return true;
        }
```

On note que la méthode `solve` n'est pas appelée si `check` renvoie `false`, car l'opérateur `&&` est paresseux.

Si en revanche on sort de la boucle `for`, c'est que les 9 valeurs possibles ont toutes été essayées sans succès. Dans ce cas, on restaure la valeur 0 dans la case `c`, puis on signale l'échec de la recherche.

```
        this.grid[c] = 0;
        return false;
      }
```

Il est important de comprendre que seule la première case vide a été considérée. En effet, si une solution existe, alors la valeur correspondante de la case `c` aurait dû mener à cette solution. Il est donc inutile de considérer les autres cases vides. Ce serait une perte de temps considérable.

Si en revanche on sort de la boucle `for`, c'est que la grille ne contient aucune case vide. Dans ce cas, on signale le succès.

```
    return true;
  }
```

Le code complet est donné programme 27 page 142. Un tel code résout un problème de Sudoku en quelques centièmes de seconde.

**Exercice 81.** Ajouter un constructeur à la classe `Sudoku` permettant de donner une configuration initiale, par exemple sous la forme d'une chaîne de 81 caractères.

Solution □

**Exercice 82.** Ajouter une méthode `print` à la classe `Sudoku` pour imprimer le contenu de la grille.

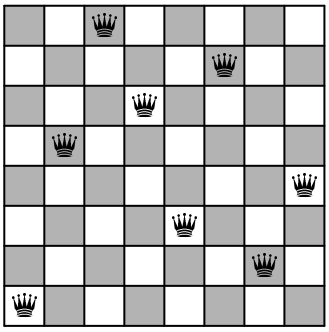
Solution □

**Programme 27 — Résoudre le problème du Sudoku**

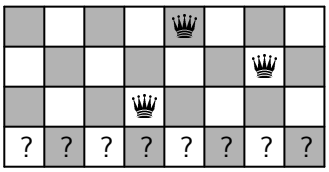
```
class Sudoku {

  private int[] grid = new int[81];

  int row(int c) { return c / 9; }
  int col(int c) { return c % 9; }
  int group(int c) { return 3 * (row(c) / 3) + col(c) / 3; }

  boolean sameZone(int c1, int c2) {
    return row(c1) == row(c2)
        || col(c1) == col(c2)
        || group(c1) == group(c2);
  }

  // vérifie que la valeur dans la case p est compatible avec les autres cases
  boolean check(int p) {
    for (int c = 0; c < 81; c++)
      if (c != p && sameZone(p, c) && this.grid[p] == this.grid[c])
        return false;
    return true;
  }

  // essaye de résoudre la grille courante et signale le succès
  boolean solve() {
    for (int c = 0; c < 81; c++)
      if (this.grid[c] == 0) {
        for (int v = 1; v < 10; v++) {
          this.grid[c] = v;
          if (check(c) && solve())
            return true;
        }
        this.grid[c] = 0;
        return false;
      }
    return true;
  }
}
```

## 12.2 Le problème des $N$ reines

Illustrons le technique du rebroussement sur un second exemple : le problème des $N$ reines. Il s'agit de placer $N$ reines sur un échiquier $N \times N$ de telle sorte qu'aucune reine ne soit en prise avec une autre. Voici par exemple l'une des 92 solutions pour $N = 8$ :



On va procéder de façon relativement brutale, par exploration de toutes les possibilités. On fait cependant preuve d'un peu d'intelligence en remarquant qu'une solution comporte nécessairement une et une seule reine sur chaque ligne de l'échiquier. Du coup, on va chercher à remplir l'échiquier ligne par ligne, en positionnant à chaque fois une reine sans qu'elle soit en prise avec les reines déjà posées. Ainsi, si on a déjà posé trois reines sur les trois premières lignes de l'échiquier, alors on en vient à chercher une position valide sur la quatrième ligne :




Si on en trouve une, alors on place une reine à cet endroit et on poursuit l'exploration avec la ligne suivante. Sinon, *on fait machine arrière* sur l'un des choix précédents, et on recommence. Si on parvient à remplir la dernière ligne, on a trouvé une solution. En procédant ainsi de manière systématique, on ne ratera pas de solution.

Écrivons une méthode `int[] findSolution(int n)` qui met en œuvre cette technique et renvoie la solution trouvée, le cas échéant, ou lève une exception pour signaler l'absence de solution. On commence par allouer un tableau `cols` qui contiendra, pour chaque ligne de l'échiquier, la colonne où se situe la reine placée sur cette ligne.

```
static int[] findSolution(int n) {
  int[] cols = new int[n];
```

C'est donc un tableau de taille `n`, contenant des entiers entre 0 et `n` − 1. Le cœur de l'algorithme de rebroussement va être écrit dans une seconde méthode, `findSolutionRec`, à laquelle on passe le tableau d'une part et un indice indiquant la prochaine ligne de l'échiquier à remplir d'autre part. Elle renvoie un booléen signalant le succès de la recherche. Il suffit donc de l'appeler et de traiter correctement sa valeur de retour.

```
  if (findSolutionRec(cols, 0)) return cols;
  throw new Error("no solution for n=" + n);
}
```

On en vient à la méthode `findSolutionRec` proprement dite. Elle est écrite récursivement. Le cas de base correspond à un échiquier où toutes les lignes ont été remplies. On renvoie alors `true` pour signaler le succès de la recherche.

```
static boolean findSolutionRec(int[] cols, int r) {
  if (r == cols.length)
    return true;
```

Dans le cas contraire, il faut essayer successivement toutes les colonnes possibles pour la reine de la ligne `r`. Pour chacune, on enregistre le choix qui est fait dans le tableau `cols`.

```
  for (int c = 0; c < cols.length; c++) {
    cols[r] = c;
```

Puis on teste que ce choix est cohérent avec les choix précédents. Pour cela on va écrire (plus loin) une méthode `check` qui fait cette vérification. Si le test est positif, on appelle récursivement `findSolutionRec` pour continuer le remplissage à partir de la ligne suivante. En cas de succès, on renvoie `true` immédiatement.

```
    if (check(cols, r) && findSolutionRec(cols, r + 1))
      return true;
```

Dans le cas contraire, on poursuit la boucle avec la valeur suivante de `c`. Si on finit par sortir de la boucle, c'est que toutes les colonnes ont été essayées sans succès. On signale alors une recherche infructueuse.

```
  }
  return false;
}
```

Il nous reste à écrire la méthode `check` qui vérifie que le choix qui vient juste d'être fait pour la ligne `r` est cohérent avec les choix précédents. C'est une simple boucle sur les `r` premières lignes.

```
static boolean check(int[] cols, int r) {
  for (int q = 0; q < r; q++)
```

Pour chaque ligne `q`, on vérifie que les deux reines ne sont sur la même colonne et ni sur une même diagonale. Si c'est le cas, on échoue immédiatement.

```
    if (cols[q] == cols[r] || Math.abs(cols[q] - cols[r]) == r - q)
      return false;
```

On notera l'utilisation de la valeur absolue pour éviter de distinguer les deux types de diagonales (montante et descendante). Si en revanche on sort de la boucle, alors on signale une vérification positive.

```
  return true;
}
```

Le code complet est donné programme 28 page 145. Il est important de bien comprendre que ce programme s'interrompt à la première solution trouvée. C'est le rôle du second `return true` placé au milieu de la boucle de la méthode `findSolutionRec`.

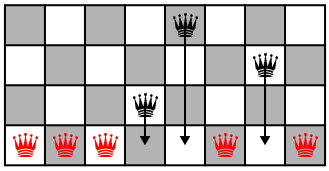
**Exercice 83.** Modifier le programme précédent pour qu'il dénombre toutes les solutions. Les solutions ne seront pas renvoyées, mais seulement leur nombre total. Pour $1 \leq N \leq 9$, on doit obtenir les valeurs 1, 0, 0, 2, 10, 4, 40, 92, 352. Solution □

**Programme 28 — Résoudre le problème des $N$ reines**

```
static boolean check(int[] cols, int r) {
  for (int q = 0; q < r; q++)
    if (cols[q] == cols[r] || Math.abs(cols[q] - cols[r]) == r - q)
      return false;
  return true;
}

static boolean findSolutionRec(int[] cols, int r) {
  if (r == cols.length)
    return true;
  for (int c = 0; c < cols.length; c++) {
    cols[r] = c;
    if (check(cols, r) && findSolutionRec(cols, r + 1))
      return true;
  }
  return false;
}

static int[] findSolution(int n) {
  int[] cols = new int[n];
  if (findSolutionRec(cols, 0))
    return cols;
  throw new Error("no solution for n=" + n);
}
```

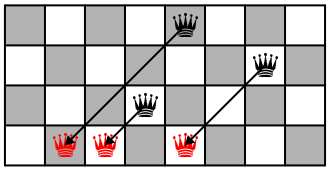
**Optimisation.** Si on s'intéresse au problème du dénombrement de toutes les solutions, on ne connaît pas à ce jour de méthode qui soit fondamentalement meilleure que la recherche exhaustive que nous venons de présenter [1]. Néanmoins, cela ne nous empêche pas de chercher à optimiser le programme précédent. Une idée naturelle consiste à maintenir, pour chaque ligne de l'échiquier, les colonnes sur lesquelles on peut encore placer une reine. Ainsi, plutôt que d'essayer systématiquement les $N$ colonnes de la ligne courante, on peut espérer avoir à en examiner « beaucoup moins » que $N$ et, en particulier, faire machine arrière plus rapidement.

Illustrons cette idée avec $N = 8$. Supposons qu'on ait déjà placé des reines sur les trois premières lignes. Alors seules cinq colonnes (ici dessinées en rouge) doivent être considérées pour la quatrième ligne.



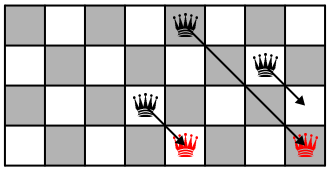
(1)

Par ailleurs, trois positions de la quatrième ligne sont en prise avec les reines déjà placées le long d'une diagonale ascendante. Ces trois positions (ici en rouge) ne doivent pas être considérées.



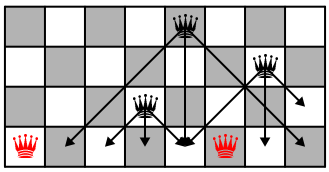
(2)

De même, deux positions de la quatrième ligne sont en prise avec des reines déjà placées le long d'une diagonale descendante. Ces deux positions (ici en rouge) ne doivent pas être considérées.



(3)

Au final, ce sont donc six positions de la quatrième ligne qui ne peuvent plus être considérées. Il ne reste finalement que deux positions (ici en rouge) à considérer, au lieu de 8 dans le programme précédent.



(4)

Mettons en œuvre cette idée dans un programme qui dénombre les solutions du problème des $N$ reines. Il nous faut représenter les ensembles de colonnes à considérer, ou à exclure, qui sont matérialisées en rouge dans les figures ci-dessus. Comme il s'agit de petits ensembles dont les éléments sont dans $\{0, \ldots, N-1\}$, on peut avantageusement les représenter par des valeurs de type `int`, le bit $i$ indiquant la présence de l'entier $i$ dans l'ensemble [2]. Ainsi, dans la figure 1 ci-dessus, les cinq colonnes à considérer sur la quatrième ligne peuvent être représentées par l'entier dont l'écriture binaire est $11100101_2$, c'est-à-dire 229 (l'usage est d'écrire les bits de poids faible à droite). De la même manière, les trois positions en prise par une diagonale ascendante (figure 2) correspondent à l'entier $104 = 01101000_2$, et les deux positions en prise par une diagonale descendante (figure 3) à l'entier $9 = 00001001_2$. L'intérêt de cette représentation est que les opérations sur les entiers fournies par la machine vont nous permettre de calculer très efficacement certaines opérations « ensemblistes ». Ainsi, si `a`, `b` et `c` sont trois variables de type `int` contenant respectivement les entiers 229, 104 et 9, c'est-à-dire les trois ensembles ci-dessus, alors on peut calculer l'ensemble correspondant à la figure 4 avec l'expression `a & ~b & ~c`.

1. Si en revanche le problème est de trouver *une* solution, alors il existe un algorithme polynômial.
2. comme nous l'avons brièvement expliqué dans la section 1.2.1 et déjà fait dans la section 11.4.

L'opérateur `&` de Java est le ET bit à bit et l'opérateur `~` le NON bit à bit. En termes ensemblistes, on vient de calculer (`a`\`b`)\`c`.

Sur la base de cette idée, écrivons une méthode récursive `countSolutionsRec` qui prend justement en arguments les trois entiers `a`, `b` et `c`.

```
static int countSolutionsRec(int a, int b, int c) {
```

L'entier `a` désigne les colonnes restant à pourvoir, l'entier `b` (resp. `c`) les colonnes interdites car en prise sur une diagonale ascendante (resp. descendante). La méthode renvoie le nombre de solutions qui sont compatibles avec ces arguments. La recherche parvient à son terme lorsque `a` devient vide, c'est-à-dire 0. On signale alors la découverte d'une solution.

```
    if (a == 0) return 1;
```

Dans le cas contraire, on calcule les colonnes à considérer avec l'expression `a & ~b & ~c`, comme expliqué ci-dessus, dans une variable `e`. On initialise également une variable `f` pour tenir le compte des solutions.

```
    int e = a & ~b & ~c, f = 0;
```

Notre objectif est maintenant de parcourir tous les éléments de l'ensemble représenté par `e`, le plus efficacement possible. Nous allons parcourir les bits de `e` qui sont à 1, et les supprimer au fur et à mesure, jusqu'à ce que `e` devienne nul.

```
    while (e != 0) {
```

Il existe une astuce arithmétique qui nous permet d'extraire exactement un bit d'un entier non nul. Elle exploite la représentation en complément à deux des entiers, en combinant le ET bit à bit et la négation (entière) :

```
        int d = e & -e;
```

Pour s'en convaincre, il faut se souvenir qu'en complément à deux, `-e` est égal à `~e` + 1. (Un excellent ouvrage plein de telles astuces est *Hacker's Delight* [17].) Ce bit `d` de `e` que l'on vient d'extraire représente une colonne à considérer. Il nous suffit donc de procéder à un appel récursif, en mettant à jour les valeurs de `a`, `b` et `c` en conséquence.

```
        f += countSolutionsRec(a - d, (b + d) << 1, (c + d) >> 1);
```

Le bit `d` a été retiré de `a` avec une soustraction (il n'y a pas de retenue car `d` était nécessairement un bit de `a`), et a été ajouté à `b` et `c` avec une addition (de même il n'y a pas de retenue car `d` n'était pas un bit de `b` ou `c`). Les valeurs de `b` et `c` sont décalées d'un bit, respectivement vers la gauche et vers la droite avec les opérateurs `<<` et `>>`, pour exprimer le passage à la ligne suivante. Une fois le résultat accumulé dans `f`, on supprime le bit `d` de `e`, pour passer maintenant au bit suivant de `e`.

```
        e -= d;
    }
```

Une fois sorti de la boucle, il n'y a plus qu'à renvoyer la valeur de `f`.

```
    return f;
}
```

**Programme 29 — Le problème des $N$ reines (dénombrement)**

```
static int countSolutionsRec(int a, int b, int c) {
  if (a == 0) return 1;
  int f = 0, e = a & ~b & ~c;
  while (e != 0) {
    int d = e & -e;
    f += countSolutionsRec(a - d, (b + d) << 1, (c + d) >> 1);
    e -= d;
  }
  return f;
}

static int countSolutions(int n) {
  return countSolutionsRec(~(~0 << n), 0, 0);
}
```

Le programme principal se contente d'un appel à `countSolutionsRec`, avec une valeur de `a` représentant l'ensemble $\{0, \ldots, N-1\}$

```
static int countSolutions(int n) {
  return countSolutionsRec(~(~0 << n), 0, 0);
}
```

Le code complet est donné programme 29 page 148. Si on compare les performances de ce programme avec le précédent (en supposant l'avoir adapté comme suggéré dans l'exercice 83), on observe un gain notable de performance. Ainsi pour $N = 14$, on dénombre les 365 596 solutions en un quart de seconde, au lieu de près de 8 secondes. Plus intéressant que le temps lui-même est le nombre de décisions prises. On le donne ici pour les deux versions du dénombrement, pour quelques valeurs de $N$.

| $N$ | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|
| version naïve | $3,5 \times 10^5$ | $1,8 \times 10^6$ | $1,0 \times 10^7$ | $6,0 \times 10^7$ | $3,8 \times 10^8$ |
| version optimisée | $3,6 \times 10^4$ | $1,7 \times 10^5$ | $8,6 \times 10^5$ | $4,7 \times 10^6$ | $2,7 \times 10^7$ |
| rapport | 9,80 | 10,8 | 11,8 | 12,8 | 13,8 |

Comme on le constate empiriquement, le rapport augmente avec $N$, approximativement comme $N$.

# 13 Tri

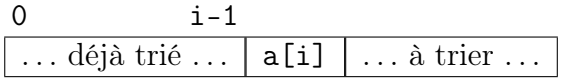
Ce chapitre présente plusieurs algorithmes de tri. On suppose pour simplifier qu'on trie des tableaux d'entiers (type `int[]`), dans l'ordre croissant. À la fin du chapitre, nous expliquons comment généraliser à des éléments d'un type quelconque. On note $N$ le nombre d'éléments à trier. Pour chaque tri présenté, on indique sa complexité en nombre de comparaisons effectuées et en nombre d'affectations.

On rappelle que la complexité optimale d'un tri effectuant uniquement des comparaisons d'éléments est en $O(N \log N)$. En effet, on peut visualiser un tel algorithme comme un arbre binaire. Chaque nœud interne représente une comparaison effectuée, le sous-arbre gauche (resp. droit) représentant la suite de l'algorithme lorsque le test est positif (resp. négatif). Chaque feuille représente un résultat possible, c'est-à-dire une permutation effectuée sur la séquence initiale. Si on suppose les $N$ éléments distincts, il y a $N!$ permutations possibles, donc au moins $N!$ feuilles à cet arbre. Sa hauteur est donc au moins égale à $\log N!$. Or le plus long chemin de la racine à une feuille représente le plus grand nombre de comparaisons effectuées par l'algorithme sur une entrée. Il existe donc une entrée pour laquelle le nombre de comparaisons est au moins $\log(N!)$. Par la formule de Stirling, on sait que $\log(N!) \sim N \log N$. Pour une preuve plus détaillée, on pourra consulter *The Art of Computer Programming* [12, Sec. 5.3].

## 13.1 Tri par insertion

Le tri par insertion est sans doute le tri le plus naturel. Il consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. C'est ce que l'on fait naturellement quand on trie un jeu de cartes ou un paquet de copies.

Le tri par insertion d'un tableau `a` s'effectue en place. Il consiste à insérer successivement chaque élément `a[i]` dans la portion du tableau `a[0..i-1]` déjà triée, ce qui correspond à la situation suivante :




On commence par une boucle `for` pour parcourir le tableau :

```
static void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
```

**Programme 30 — Tri par insertion**

```
static void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    int v = a[i], j = i;
    for (; 0 < j && v < a[j-1]; j--)
      a[j] = a[j-1];
    a[j] = v;
  }
}
```

Pour insérer l'élément `a[i]` à la bonne place, on utilise une seconde boucle qui décale vers la droite les éléments tant qu'ils sont supérieurs à `a[i]`.

```
    int v = a[i], j = i;
    for (; 0 < j && v < a[j-1]; j--)
      a[j] = a[j-1];
```

Une fois sorti de la boucle, il reste à positionner `a[i]` à sa place, c'est-à-dire à la position donnée par `j` :

```
    a[j] = v;
  }
```

Le code complet est donné programme 30 page 150.

**Complexité.** On note que la méthode `insertionSort` effectue exactement le même nombre de comparaisons et d'affectations. Lorsque la boucle `for` insère l'élément `a[i]` à la position $\texttt{i} - k$, elle effectue $k + 1$ comparaisons. Au mieux $k$ vaut 0 et au pire $k$ vaut `i`, ce qui donne au final le tableau suivant :

| | meilleur cas | moyenne | pire cas |
|---|---|---|---|
| comparaisons | $N$ | $N^2/4$ | $N^2/2$ |
| affectations | $N$ | $N^2/4$ | $N^2/2$ |

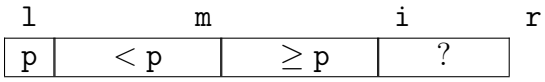
## 13.2 Tri rapide

Le tri rapide consiste à appliquer la méthode *diviser pour régner* : on partage les éléments à trier en deux sous-ensembles, les éléments du premier étant plus petits que les éléments du second, puis on trie récursivement chaque sous-ensemble. En pratique, on réalise le partage à l'aide d'un élément $p$ arbitraire de l'ensemble à trier, appelé *pivot*. Les deux sous-ensembles sont alors respectivement les éléments plus petits et plus grands que $p$.

Le tri rapide d'un tableau s'effectue en place. On le paramètre par deux indices `l` et `r` indiquant la portion du tableau à trier, `l` étant inclus et `r` exclus. On commence par écrire une méthode `partition` qui va déplacer les éléments autour du pivot. On choisit l'élément `a[l]` comme pivot, arbitrairement :

```
static int partition(int[] a, int l, int r) {
  int p = a[l];
```

On suppose ici `l` $<$ `r`, ce qui nous autorise à accéder à `a[l]`. Le principe consiste alors à parcourir le tableau de la gauche vers la droite, entre les indices `l` (inclus) et `r` (exclus), avec une boucle `for`. À chaque tour de boucle, la situation est la suivante :




L'indice `i` de la boucle dénote le prochain élément à considérer et l'indice `m` partitionne la portion déjà parcourue.

```
  int m = l;
  for (int i = l + 1; i < r; i++)
```

Si `a[i]` est supérieur ou égal à `p`, il n'y a rien à faire. Dans le cas contraire, pour conserver l'invariant de boucle, il suffit d'incrémenter `m` et d'échanger `a[i]` et `a[m]`.

```
    if (a[i] < p)
      swap(a, i, ++m);
```

(Le code de la méthode `swap` est donné page 152.) Une fois sorti de la boucle, on met le pivot à sa place, c'est-à-dire à la position `m`, et on renvoie cet indice.

```
  swap(a, l, m);
  return m;
```

On peut bien entendu se dispenser de l'appel à `swap` lorsque `l` $=$ `m`, mais cela ne change rien fondamentalement. On écrit alors la partie récursive du tri rapide sous la forme d'une méthode `quickrec` qui prend les mêmes arguments que la méthode `partition`. Si `l` $\geq$ `r-1`, il y a au plus un élément à trier et il n'y a donc rien à faire.

```
static void quickrec(int[] a, int l, int r) {
  if (l >= r-1) return;
```

Sinon, on partitionne les éléments entre `l` et `r`.

```
  int m = partition(a, l, r);
```

Après cet appel, le pivot `a[m]` se retrouve à sa place définitive. On effectue alors deux appels récursifs pour trier `a`[`l`..`m`[ et `a`[`m+1`..`r`[,

```
  quickrec(a, l, m);
  quickrec(a, m + 1, r);
```

ce qui achève la méthode `quickrec`. Pour trier un tableau, il suffit d'appeler `quickrec` sur la totalité de ses éléments.

```
static void quicksort(int[] a) {
  quickrec(a, 0, a.length);
}
```

Le code complet est donné programme 31 page 152.

**Programme 31 — Tri rapide**

```
static void swap(int[] a, int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}

// suppose l < r i.e. au moins un élément
static int partition(int[] a, int l, int r) {
  int p = a[l];
  int m = l;
  for (int i = l + 1; i < r; i++)
    if (a[i] < p)
      swap(a, i, ++m);
  swap(a, l, m);
  return m;
}

static void quickrec(int[] a, int l, int r) {
  if (l >= r-1) return;
  int m = partition(a, l, r);
  quickrec(a, l, m);
  quickrec(a, m + 1, r);
}

static void quicksort(int[] a) {
  quickrec(a, 0, a.length);
}
```

**Complexité.** La méthode `partition` fait toujours exactement `r - l - 1` comparaisons. Pour la méthode `quicksort`, notons $C(N)$ le nombre de comparaisons qu'elle effectue sur une portion de tableau longueur $N$. Si la méthode `partition` a déterminé une portion de longueur $K$ et une autre de longueur $N-1-K$, on a donc au total

$$C(N) = N - 1 + C(K) + C(N-1-K).$$

Le pire des cas correspond à $K = 0$, ce qui donne $C(N) = N - 1 + C(N-1)$, d'où $C(N) \sim \frac{N^2}{2}$. Le meilleur des cas correspond à une portion coupée en deux moitiés égales, c'est-à-dire $K = N/2$. On en déduit facilement $C(N) \sim N \log N$. Pour le nombre de comparaisons en moyenne, on considère que les $N$ places finales possibles pour le pivot sont équiprobables, ce qui donne

$$\begin{aligned} C(N) &= N - 1 + \frac{1}{N} \sum_{0 \leq K \leq N-1} C(K) + C(N-1-K) \\ &= N - 1 + \frac{2}{N} \sum_{0 \leq K \leq N-1} C(K). \end{aligned}$$

Après un peu d'algèbre (laissée au lecteur), on parvient à

$$\frac{C(N)}{N+1} = \frac{C(N-1)}{N} + \frac{2}{N+1} - \frac{2}{N(N+1)}.$$

Il s'agit d'une somme téléscopique, qui permet de conclure que

$$\frac{C(N)}{N+1} \sim 2 \log N$$

et donc que $C(N) \sim 2N \log N$.

En ce qui concerne le nombre d'affectations, on note que la méthode `partition` effectue autant d'appels à `swap` que d'incrémentations de `m`. Le meilleur des cas est atteint lorsque le pivot est toujours à sa place. Il n'y a alors aucune affectation. Il est important de noter que ce cas ne correspond pas à la meilleure complexité en termes de comparaisons (qui est alors quadratique). En moyenne, toutes les positions finales pour le pivot étant équiprobables, on a donc moins de `r-l+1` affectations (chaque appel à `swap` réalise deux affectations), d'où un calcul analogue au nombre moyen de comparaisons. Dans le pire des cas, le pivot se retrouve toujours à la position `r-1`. La méthode `partition` effectue alors $2$(`r - l`) affectations, d'où un total de $N^2$ affectations. Au final, on obtient donc les résultats suivants :

| | meilleur cas | | moyenne | pire cas |
|---|---|---|---|---|
| comparaisons | $N \log N$ | | $2N \log N$ | $N^2/2$ |
| affectations | | 0 | $2N \log N$ | $N^2$ |

**Améliorations.** Choisir systématiquement le premier élément de l'intervalle `a[l..r[` comme pivot n'est pas forcément une bonne idée. Par exemple, si le tableau est déjà trié, on se retrouvera avec une complexité quadratique. Il est préférable de choisir le pivot aléatoirement parmi les valeurs de `a[l..r[`. Une solution très simple consiste à mélanger le

tableau *avant* de commencer le tri rapide. L'exercice 5 propose justement une méthode très simple pour effectuer un tel mélange.

Toutefois, si les valeurs du tableau sont toutes identiques, cela ne suffira pas. En effet, le pivot se retrouvera à une extrémité de l'intervalle et on aura toujours une complexité quadratique. La solution à ce problème consiste en une méthode `partition` un peu plus subtile, proposée dans l'exercice 84 ci-dessous. Avec ces deux améliorations, on peut considérer en pratique que le tri rapide est toujours en $O(N \log N)$.

Comme on le voit, réaliser un tri rapide en prenant soin d'éviter un pire cas quadratique n'est pas si facile que cela. Dans les sections suivantes, nous présentons deux autres tris, le tri par tas et le tri fusion, qui ont tous les deux une complexité $O(N \log N)$ dans le pire des cas tout en étant plus simples à réaliser. Néanmoins, le tri rapide est souvent préféré en pratique, car meilleur en temps que le tri par tas et meilleur en espace que le tri fusion.

**Exercice 84.** Modifier la méthode `partition` pour qu'elle sépare les éléments strictement plus petits que le pivot (à gauche), les éléments égaux au pivot (au milieu) et les éléments strictement plus grands que le pivot (à droite). Au lieu de deux indices `m` et `i` découpant le segment de tableau en trois parties, comme illustré sur la figure page 151, on utilisera trois indices découpant le segment de tableau en quatre parties. (Un tel découpage en trois est aussi l'objet de l'exercice 91 plus loin.) La nouvelle méthode `partition` doit maintenant renvoyer deux indices. Modifier la méthode `quick_rec` en conséquence. Solution □

**Exercice 85.** Pour éviter le débordement de pile potentiel de la méthode `quickrec`, une idée consiste à effectuer d'abord l'appel récursif sur la plus petite des deux portions, puis à remplacer le second appel récursif par une boucle `while` en modifiant la valeur de `l` ou `r` en conséquence. Montrer que la taille de pile est alors logarithmique dans le pire des cas. Solution □

**Exercice 86.** Une idée classique pour accélérer un algorithme de tri consiste à effectuer un tri par insertion quand le nombre d'éléments à trier est petit, *i.e.* devient inférieur à une constante fixée à l'avance (par exemple 5). Modifier le tri rapide pour prendre en compte cette idée. On pourra reprendre la méthode `insertionSort` de la section précédente (figure 30) et la généraliser en lui passant deux indices `l` et `r` pour délimiter la portion du tableau à trier. Solution □

## 13.3 Tri fusion

Comme le tri rapide, le tri fusion applique le principe *diviser pour régner*. Il partage les éléments à trier en deux parties de même taille, sans chercher à comparer leurs éléments. Une fois les deux parties triées récursivement, il les fusionne, d'où le nom de tri fusion. Ainsi on évite le pire cas du tri rapide où les deux parties sont de tailles disproportionnées.

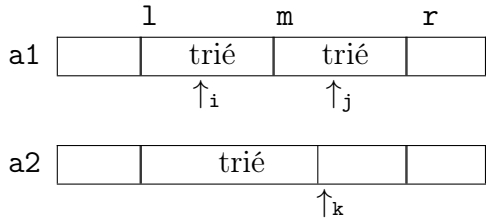
On va chercher à réaliser le tri en place, en délimitant la portion à trier par deux indices `l` (inclus) et `r` (exclus). Pour le partage, il suffit de calculer l'indice médian $\texttt{m} = \frac{\texttt{l}+\texttt{r}}{2}$. On trie alors récursivement les deux parties délimitées par `l` et `m` d'une part, et `m` et `r` d'autre part. Il reste à effectuer la fusion. Il s'avère extrêmement difficile de la réaliser en place. Le plus simple est d'utiliser un second tableau, alloué une et une seule fois au début du tri.

On commence par écrire une méthode `merge` qui réalise la fusion. Elle prend en arguments deux tableaux, `a1` et `a2`, et les trois indices `l`, `m` et `r`. Les portions `a1`[`l`..`m`[ et

`a1[m..r[` sont supposées triées. L'objectif est de les fusionner dans `a2[l..r[`. Pour cela, on va parcourir les deux portions de `a1` avec deux variables `i` et `j` et la portion de `a2` à remplir avec une boucle `for`.

```
static void merge(int[] a1, int[] a2, int l, int m, int r) {
  int i = l, j = m;
  for (int k = l; k < r; k++)
```

À chaque tour de boucle, la situation est donc la suivante :




Il faut alors déterminer la prochaine valeur à placer en `a2[k]`. Il s'agit de la plus petite des deux valeurs `a1[i]` et `a1[j]`. Il convient cependant de traiter correctement le cas où il n'y a plus d'élément dans l'une des deux moitiés. On détermine si l'élément doit être pris dans la moitié gauche avec le test suivant :

```
    if (i < m && (j == r || a1[i] <= a1[j]))
```

Dans les deux cas, on copie l'élément dans `a2[k]` et on incrémente l'indice correspondant.

```
      a2[k] = a1[i++];
    else
      a2[k] = a1[j++];
```

Ceci achève la méthode `merge`. La partie récursive du tri fusion est matérialisée par une méthode récursive `mergesortrec` qui prend en arguments deux tableaux `a` et `tmp` et deux indices `l` et `r` délimitant la portion à trier. Elle trie `a[l..r[` en se servant du tableau `tmp` comme temporaire. Si le segment contient au plus un élément, c'est-à-dire si `l` $\geq$ `r-1`, il n'y a rien à faire.

```
static void mergesortrec(int[] a, int[] tmp, int l, int r) {
  if (l >= r-1) return;
```

Sinon, on partage l'intervalle deux moitiés égales en calculant l'élément médian `m`

```
  int m = l + (r - l) / 2;
```

(Le calcul de `m` comme `(l + r) / 2` pourrait provoquer un débordement de capacité arithmétique ; voir exercice 9.) On trie alors récursivement les deux portions `a[l..m[` et `a[m..r[`, toujours en utilisant `tmp` comme temporaire.

```
  mergesortrec(a, tmp, l, m);
  mergesortrec(a, tmp, m, r);
```

On recopie ensuite tous les éléments de la portion `a[l..r[` dans `tmp` avant de faire la fusion proprement dite avec la méthode `merge` :

**Programme 32 — Tri fusion**

```
static void merge(int[] a1, int[] a2, int l, int m, int r) {
  int i = l, j = m;
  for (int k = l; k < r; k++)
    if (i < m && (j == r || a1[i] <= a1[j]))
      a2[k] = a1[i++];
    else
      a2[k] = a1[j++];
}

static void mergesortrec(int[] a, int[] tmp, int l, int r) {
  if (l >= r-1) return;
  int m = l + (r - l) / 2;
  mergesortrec(a, tmp, l, m);
  mergesortrec(a, tmp, m, r);
  for (int i = l; i < r; i++)
    tmp[i] = a[i];
  merge(tmp, a, l, m, r);
}

static void mergesort(int[] a) {
  mergesortrec(a, new int[a.length], 0, a.length);
}
```

```
  for (int i = l; i < r; i++)
    tmp[i] = a[i];
  merge(tmp, a, l, m, r);
```

Ceci achève la méthode `mergesortrec`. Le tri d'un tableau complet `a` consiste alors en un simple appel à `mergesortrec`, en allouant un temporaire de la même taille que `a` :

```
static void mergesort(int[] a) {
  mergesortrec(a, new int[a.length], 0, a.length);
}
```

Le code complet est donné programme 32 page 156. On peut encore améliorer l'efficacité de ce code. Comme pour le tri rapide, on peut utiliser un tri par insertion quand la portion à trier devient suffisamment petite (voir exercice 86). Une autre idée, indépendante, consiste à éviter la copie de `a` vers `tmp`. L'exercice 87 propose une solution.

**Complexité.** Si on note $C(N)$ (resp. $f(N)$) le nombre total de comparaisons effectuées par `mergesortrec` (resp. `merge`), on a l'équation

$$C(N) = 2C(N/2) + f(N)$$

car les deux appels récursifs se font sur deux portions de même longueur $N/2$. Dans le meilleur des cas, la méthode `merge` n'examine que les éléments de l'une des deux portions

car ils sont tous plus petits que ceux de l'autre portion. Dans ce cas $f(N) = N/2$ et donc $C(N) \sim \frac{1}{2}N \log N$. Dans le pire des cas, tous les éléments sont examinés par `merge` et donc $f(N) = N - 1$, d'où $C(N) \sim N \log N$. L'analyse en moyenne est plus subtile (voir [12, ex 2 p. 646]) et donne $f(N) = N - 2 + o(1)$, d'où $C(N) \sim N \log N$ également.

Le nombre d'affectations est le même dans tous les cas : $N$ affectations dans la méthode `merge` (chaque élément est copié de `a1` vers `a2`) et $N$ affectations effectuées par `mergesortrec`. Si on note $A(N)$ le nombre total d'affectations pour `mergesort`, on a donc

$$A(N) = 2A(N/2) + 2N,$$

d'où un total de $2N \log N$ affectations.

| | meilleur cas | moyenne | pire cas |
|---|---|---|---|
| comparaisons | $\frac{1}{2}N \log N$ | $N \log N$ | $N \log N$ |
| affectations | $2N \log N$ | $2N \log N$ | $2N \log N$ |

**Exercice 87.** Pour éviter la copie de `a` vers `tmp` dans la méthode `mergesortrec`, une idée consiste à trier les deux moitiés du tableau `a` tout en les déplaçant vers le tableau `tmp`, puis à fusionner de `tmp` vers `a` comme on le fait déjà. Cependant, pour trier les éléments de `a` vers `tmp`, il faut, inversement, trier les deux moitiés en place puis fusionner vers `tmp`. On a donc besoin de deux méthodes de tri mutuellement récursives. On peut cependant n'en n'écrire qu'une seule, en passant un paramètre supplémentaire indiquant si le tri doit être fait en place ou vers `tmp`. Modifier les méthodes `mergesortrec` et `mergesort` en suivant cette idée. Solution □

**Exercice 88.** Le tri fusion est une bonne méthode pour trier des listes. Supposons par exemple que l'on souhaite trier des listes de type `LinkedList<Integer>`. Écrire tout d'abord une méthode `split` qui prend en arguments trois listes `l1`, `l2` et `l3` et met la moitié des éléments de `l1` dans `l2` et l'autre moitié dans `l3` (par exemple un élément sur deux). La méthode `split` ne doit pas modifier la liste `l1`. Écrire ensuite une méthode `merge` qui prend en arguments deux listes `l1` et `l2`, supposées triées, et renvoie la fusion de ces deux listes. Elle peut vider ses deux arguments de leurs éléments. Écrire une méthode récursive `mergesort` qui prend une liste en argument et renvoie une nouvelle liste triée contenant les mêmes éléments. Elle ne doit pas modifier son argument. Solution □

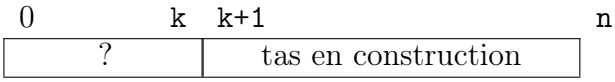
**Exercice 89.** Le tri fusion permet également de trier des listes *en place*, c'est-à-dire sans aucune allocation supplémentaire, dès lors que le contenu et la structure des listes peuvent être modifiés. Considérons par exemple les listes d'entiers du type `Singly` de la section 4.1. Écrire tout d'abord une méthode `Singly split(Singly l)` qui coupe la liste `l` en son milieu, c'est-à-dire remplace le champ `next` qui relie la première moitié à la seconde moitié par `null` et renvoie le premier élément de la seconde moitié. On pourra supposer que la liste `l` contient au moins deux éléments. Écrire ensuite une méthode `Singly merge(Singly l1, Singly l2)` qui fusionne deux listes `l1` et `l2`, supposées triées, et renvoie le premier élément du résultat. La méthode `merge` doit procéder en place, sans allouer de nouvel objet de la classe `Singly`. On pourra commencer par écrire la méthode `merge` récursivement puis on en fera une version itérative avec une boucle `while`, afin d'éviter tout débordement de pile. Écrire enfin une méthode récursive `mergesort` qui prend une liste en argument et la trie en place. Expliquer pourquoi la méthode `mergesort` ne peut pas provoquer de débordement de pile. Solution □

## 13.4 Tri par tas

Le tri par tas consiste à utiliser une file de priorité, comme celles présentées au chapitre 8. Une telle structure permet l'ajout d'un élément et le retrait du plus petit élément. L'idée du tri par tas est alors la suivante : on construit une file de priorité contenant tous les éléments à trier puis on retire les éléments un par un. L'opération de retrait donnant le plus petit élément à chaque fois, les éléments sont sortis dans l'ordre croissant.

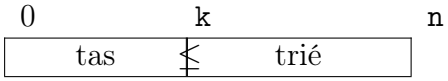
*A priori*, n'importe quelle structure de file de priorité convient. En particulier, une structure où les opérations d'ajout et de retrait ont un coût $O(\log N)$ donne immédiatement un tri optimal en $O(N \log N)$. Cependant, on peut réaliser le tri par tas *en place*, en construisant la structure de tas directement à l'intérieur du tableau que l'on cherche à trier. L'organisation du tas dans le tableau est exactement la même que dans la section 8.1 : les fils gauche et droit d'un nœud stocké à l'indice $i$ sont respectivement stockés aux indices $2i + 1$ et $2i + 2$. La seule différence est que l'on va construire un tas pour la relation d'ordre inverse, c'est-à-dire un tas où le plus grand élément se trouve à la racine.

Pour construire le tas, on considère les éléments du tableau de la droite vers la gauche. À chaque tour, on a une situation de la forme




où la partie `a[k+1..n[` contient la partie basse du tas en construction, c'est-à-dire une forêt de tas enracinés aux indices $i$ tels que $\texttt{k} < i < 2\texttt{k} + 3$. On fait alors descendre la valeur `a[k]` à sa place dans le tas de racine `k`. Une fois tous les éléments parcourus, on a un unique tas enraciné en 0.

La seconde étape consiste alors à déconstruire le tas. Pour cela, on échange sa racine $r$ en `a[0]` avec l'élément $v$ en `a[n-1]`. La valeur $r$ se trouve alors à sa place. On rétablit ensuite la structure de tas sur `a[0..n-1[`, en faisant descendre $v$ à sa place dans un tas de taille `n-1` enraciné en 0. Puis on répète l'opération pour les positions `n-1`, `n-2`, etc. À chaque tour `k`, on a la situation suivante




c'est-à-dire un tas dans la portion `a[0..k[`, dont tous les éléments sont plus petits que ceux de la partie `a[k..n[`, qui est triée.

Les deux étapes de l'algorithme ci-dessus utilisent la même opération consistant à faire descendre une valeur jusqu'à sa place dans un tas. On la réalise à l'aide d'une méthode récursive `moveDown` qui prend en arguments le tableau `a`, un indice `k`, une valeur `v` et une limite `n` sur les indices.

```
static void moveDown(int[] a, int k, int v, int n) {
```

On fait l'hypothèse qu'on a déjà un tas $h_1$ enraciné en `2k+1` dès lors que $\texttt{2k+1} < \texttt{n}$, et de même un tas $h_2$ enraciné en `2k+2` dès lors que $\texttt{2k+2} < \texttt{n}$. L'objectif est de construire un tas enraciné en `k`, contenant `v` et tous les éléments de $h_1$ et $h_2$. On commence par déterminer si le tas enraciné en `k` est réduit à une feuille, c'est-à-dire si le tas $h_1$ n'existe pas. Si c'est le cas, on affecte la valeur `v` à `a[k]` et on a terminé.

```
int r = 2 * k + 1;
if (r >= n)
  a[k] = v;
```

Sinon, on détermine dans `r` l'indice de la plus grande des deux racines de $h_1$ et $h_2$, en traitant avec soin le cas où $h_2$ n'existe pas.

```
else {
  if (r + 1 < n && a[r] < a[r + 1]) r++;
```

Si la valeur `v` est supérieure ou égale à `a[r]`, la descente est terminée et il suffit d'affecter `v` à `a[k]`.

```
  if (a[r] <= v)
    a[k] = v;
```

Sinon, on fait remonter la valeur `a[r]` et on poursuit la descente de `v` avec un appel récursif sur la position `r`.

```
  else {
    a[k] = a[r];
    moveDown(a, r, v, n);
  }
```

Ceci achève la méthode `moveDown`. La méthode de tri proprement dite prend un tableau `a` en argument

```
static void heapsort(int[] a) {
  int n = a.length;
```

Elle commence par construire le tas de bas en haut par des appels à `moveDown`. On évite les appels inutiles sur des tas réduits à des feuilles en commençant la boucle à $\lfloor \frac{n}{2} \rfloor - 1$ (en effet, pour tout indice `k` strictement supérieur, on a `2k+1` $\geq$ `n`).

```
  for (int k = n / 2 - 1; k >= 0; k--)
    moveDown(a, k, a[k], n);
```

Une fois le tas entièrement construit, on en extrait les éléments un par un dans l'ordre décroissant. Comme expliqué ci-dessus, pour chaque indice `k`, on échange `a[0]` avec la valeur `v` en `a[k]` puis on fait descendre `v` à sa place.

```
  for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
  }
```

On note que la spécification de `moveDown` nous permet d'éviter d'affecter `v` en `a[0]` avant d'entamer le descente. Le code complet est donné programme 33 page 160.

**Programme 33 — Tri par tas**

```
static void moveDown(int[] a, int k, int v, int n) {
  int r = 2 * k + 1;
  if (r >= n)
    a[k] = v;
  else {
    if (r + 1 < n && a[r] < a[r + 1])
      r++;
    if (a[r] <= v)
      a[k] = v;
    else {
      a[k] = a[r];
      moveDown(a, r, v, n);
    }
  }
}

static void heapsort(int[] a) {
  int n = a.length;
  for (int k = n / 2 - 1; k >= 0; k--)
    moveDown(a, k, a[k], n);
  for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
  }
}
```

**Complexité.** Considérons tout d'abord le coût de la méthode `moveDown`. Le nombre d'appels récursifs est majoré par $\log \mathtt{n}$, puisque la valeur de `k` est doublée à chaque appel. D'autre part, `moveDown` effectue au plus deux comparaisons et une affectation à chaque appel. Dans le pire des cas, on obtient un total de $2 \log \mathtt{n}$ comparaisons et $\log \mathtt{n}$ affectations.

Pour le tri lui-même, on peut grossièrement majorer le nombre de comparaisons effectuées dans chaque appel à `moveDown` par $2 \log N$, soit un total $C(N)$ au pire égal à $3N \log N$ comparaisons (qui se décompose en $N \log N$ pour la première étape et $2N \log N$ pour la seconde). De même, le nombre total d'affectations est au pire $\frac{3}{2} N \log N$.

En réalité, on peut être plus précis et montrer notamment que la première étape de l'algorithme, à savoir la construction du tas, n'a qu'un coût linéaire (voir par exemple [5, Sec. 7.3]). Dès lors, seule la seconde partie de l'algorithme contribue à la complexité asymptotique et donc $C(N) \sim 2N \log N$. Pour une analyse en moyenne du tri par tas, on renvoie à *The Art of Computer Programming* [12, p. 152]. Incidemment, on comprend en passant qu'on peut facilement construire un tas à partir d'un tableau, en temps linéaire. C'est là une opération qui pourrait avantageusement être offerte par les files de priorité de la section 8.2.

Ce tri s'effectue facilement en mémoire constante. En effet, la méthode `moveDown` peut être réécrite sous forme d'une boucle — même si on ne risque pas ici un débordement de pile, la hauteur étant logarithmique.

## 13.5 Code générique

Pour écrire un algorithme de tri générique, il suffit de se donner un paramètre de type `K` pour les éléments, et d'exiger que ce type soit muni d'une relation de comparaison, c'est-à-dire implémente l'interface `java.lang.Comparable<T>`, comme nous l'avons déjà fait pour les AVL (section 6.3.4) et les files de priorité (section 8.4). Ainsi le tri par insertion (programme 30 page 150) s'écrit dans sa version générique de la manière suivante :

```
static <K extends Comparable<K>> void insertionSort(K[] a) {
  for (int i = 1; i < a.length; i++) {
    K v = a[i];
    int j = i;
    for (; 0 < j && v.compareTo(a[j - 1]) < 0; j--)
      a[j] = a[j - 1];
    a[j] = v;
  }
}
```

## 13.6 Exercices supplémentaires

**Exercice 90.** Écrire une méthode `void twoWaySort(boolean[] a)` qui trie en place un tableau de booléens, avec la convention `false` < `true`. La complexité doit être proportionnelle au nombre d'éléments.

Solution □

**Exercice 91.** (Le drapeau hollandais de Dijkstra) Écrire une méthode qui trie en place un tableau contenant des valeurs représentant les trois couleurs du drapeau hollandais, à savoir

```
enum Color { Blue, White, Red }
```

On pourra procéder de deux façons : soit en comptant le nombre d'occurrences de chaque couleur, soit en n'effectuant que des échanges dans le tableau. Dans les deux cas, la complexité doit être proportionnelle au nombre d'éléments. Solution □

**Exercice 92.** Plus généralement, on considère le cas d'un tableau contenant $k$ valeurs distinctes. Pour simplifier, on suppose qu'il s'agit des entiers $0, \ldots, k-1$. Écrire une méthode qui trie un tel tableau en place en temps $O(\max(k, N))$ où $N$ est la taille du tableau. Solution □

# 14 Compression de données

La compression de données consiste à tenter de réduire l'espace occupé par une information. On l'utilise quotidiennement, par exemple en téléchargeant des fichiers ou encore sans le savoir en utilisant des logiciels qui compressent des données pour économiser les ressources. L'exemple typique est celui des formats d'image et de vidéo qui sont le plus souvent compressés. Ce chapitre illustre la compression de données avec un algorithme simple, à savoir l'algorithme de Huffman [8]. Il va notamment nous permettre de mettre en pratique les arbres de préfixes (section 6.4) et les files de priorité (chapitre 8).

**Exercice 93.** Expliquer pourquoi on ne peut pas écrire un programme de compression de fichiers qui parvienne systématiquement à diminuer strictement la taille du fichier qu'il compresse. Solution □

## 14.1 L'algorithme de Huffman

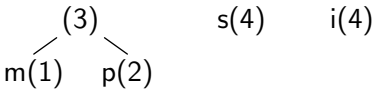
On suppose que le texte à compresser est une suite de caractères et que le résultat de la compression est une suite de bits. L'algorithme de Huffman repose sur l'idée suivante : si certains caractères du texte à compresser apparaissent souvent, il est préférable de les représenter par un code court. Par exemple, dans le texte `"mississippi"`, les caractères `'i'` et `'s'` apparaissent souvent, à savoir quatre fois chacun. On peut ainsi choisir de représenter le caractère `'i'` par la séquence `0`, le caractère `'s'` par la séquence `11` et les caractères `'m'` et `'p'` par des séquences plus longues encore, par exemple respectivement `100` et `101`. Le texte compressé sera alors `100011110111101011010`.

Les séquences pour les caractères `'i'`, `'s'`, `'m'` et `'p'` n'ont pas été choisies au hasard. Elles ont en effet la propriété qu'aucune n'est un préfixe d'une autre, permettant ainsi un décodage sans ambiguïté. On appelle cela un *code préfixe*. Il se trouve qu'il est très facile de construire un tel code si les caractères considérés forment les feuilles d'un arbre binaire. Prenons par exemple l'arbre suivant :

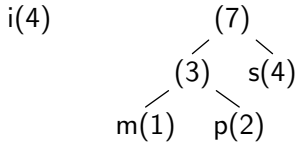
Il suffit alors d'associer à chaque caractère le chemin qui l'atteint depuis la racine, un `0` dénotant une descente vers la gauche et un `1` une descente vers la droite. Par construction, un tel code est un code préfixe. On a déjà croisé une telle représentation avec les arbres préfixes dans la section 6.4, même si le problème n'était pas posé en ces termes.

L'algorithme de Huffman permet de construire, étant donné un nombre d'occurrences pour chacun des caractères, un arbre ayant la propriété d'être le meilleur possible pour cette distribution (dans un sens qui sera expliqué plus loin). La fréquence des caractères peut être calculée avec une première passe ou donnée à l'avance s'il s'agit par exemple d'un texte écrit dans un langage pour lequel on connaît la distribution statistique des caractères. Si on reprend l'exemple de la chaîne `"mississippi"`, les nombres d'occurrences des caractères sont les suivantes :

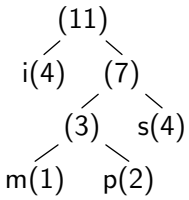

L'algorithme de Huffman procède alors ainsi. Il sélectionne les deux caractères avec les nombres d'occurrences les plus faibles, à savoir ici les caractères `'m'` et `'p'`, et les réunit en un arbre auquel il donne un nombre d'occurrences égal à la somme des nombres d'occurrences des deux caractères. On a donc la situation suivante :




Puis on recommence avec ces trois « arbres », c'est-à-dire qu'on en sélectionne deux ayant les occurrences les plus faibles, ici 3 et 4, et on les réunit en un nouvel arbre, ce qui donne par exemple ceci :




Une dernière étape de ce procédé nous donne au final l'arbre suivant :




C'est l'arbre que nous avions proposé initialement. Il se trouve qu'il est optimal, pour un sens que nous donnons maintenant.

**Optimalité.** Supposons que chaque caractère $c_i$ apparaisse avec la fréquence $f_i$. La propriété de l'arbre construit par l'algorithme de Huffman est qu'il minimise la quantité

$$S = \sum_i f_i \times d_i$$

où $d_i$ est la profondeur du caractère $c_i$ dans l'arbre, c'est-à-dire la longueur du code du caractère $c_i$. Montrons-le par l'absurde, en supposant qu'il existe un arbre pour lequel la

somme $S$ est strictement plus petite que celle obtenue avec l'algorithme de Huffman. On choisit un tel arbre $T$ qui minimise le nombre $n$ de caractères. Sans perte de généralité, supposons que $c_0$ et $c_1$ sont les deux caractères choisis initialement par l'algorithme de Huffman, c'est-à-dire deux caractères avec les fréquences les plus basses. On peut supposer que ces deux caractères sont des feuilles de $T$, car on n'augmente pas la somme $S$ en les échangeant avec des feuilles. De même, on peut supposer que ce sont deux feuilles d'un même nœud, car on peut toujours les échanger avec d'autres feuilles. Si on remplace alors ce nœud par une feuille de fréquence $f_0+f_1$, la somme $S$ diminue de $f_0+f_1$. En particulier, cette diminution ne dépend pas de la profondeur du nœud. Du coup, on vient de trouver un arbre meilleur que celui donné par l'algorithme de Huffman pour $n-1$ caractères, ce qui est une contradiction.

## 14.2 Réalisation

On commence par introduire des classes pour représenter les arbres de préfixes utilisés dans l'algorithme de Huffman. Qu'il s'agisse d'une feuille désignant un caractère ou d'un nœud interne, tout arbre contient un nombre d'occurrences qui lui permettra d'être comparé à un autre arbre. On introduit donc une classe abstraite `HuffmanTree` pour représenter un arbre, quelle que soit sa nature.

```
abstract class HuffmanTree implements Comparable<HuffmanTree> {
  int freq;
  public int compareTo(HuffmanTree that) {
    return this.freq - that.freq;
  }
}
```

Le nombre d'occurrences est stocké dans le champ `freq`. Cette classe implémente l'interface `Comparable` et sa méthode `compareTo` compare les valeurs stockées dans le champ `freq`. Une feuille est représentée par une sous-classe `Leaf` dont le champ `c` contient le caractère qu'elle désigne.

```
class Leaf extends HuffmanTree {
  final char c;
  Leaf(char c) {
    this.c = c;
    this.freq = 0;
  }
}
```

Le nombre d'occurrences, hérité de la classe `HuffmanTree`, est fixé initialement à zéro. Enfin, un nœud interne est représentée par une seconde sous-classe `Node` dont les deux champs `left` et `right` contiennent les deux sous-arbres.

```
class Node extends HuffmanTree {
  HuffmanTree left, right;
  Node(HuffmanTree left, HuffmanTree right) {
    this.left = left;
```

```
    this.right = right;
    this.freq = left.freq + right.freq;
  }
}
```

Le constructeur calcule le nombre d'occurrences, là encore hérité de la classe `HuffmanTree`, comme la somme des nombres d'occurrences des deux sous-arbres.

Écrivons maintenant le code de l'algorithme de Huffman dans une classe `Huffman`. Cette classe contient l'arbre de préfixes dans un champ `tree` et le code associé à chaque caractère dans un second champ `code`, sous la forme d'une table.

```
class Huffman {
  private HuffmanTree tree;
  private Map<Character, String> codes;
```

On va se contenter ici de construire des messages encodés sous la forme de chaînes de caractères `'0'` et `'1'` ; en pratique il s'agirait de bits. C'est pourquoi la table `codes` associe de simples chaînes aux caractères de l'alphabet.

On suppose que les fréquences d'apparition des différents caractères sont données initialement, sous la forme d'une collection de feuilles, c'est-à-dire d'une valeur `alphabet` de type `Collection<Leaf>`. L'exercice 94 propose le calcul de ces fréquences. Le constructeur prend alors la forme suivante :

```
  Huffman(Collection<Leaf> alphabet) {
    if (alphabet.size() <= 1) throw new IllegalArgumentException();
    this.tree = buildTree(alphabet);
    this.codes = new HashMap<Character, String>();
    this.tree.traverse("", this.codes);
  }
```

La méthode `buildTree` construit l'arbre de préfixes à partir de l'alphabet donné et la méthode `traverse` le parcourt pour remplir la table `codes`.

Commençons par le code de la méthode `buildTree`. Pour suivre l'algorithme présenté dans la section précédente, qui sélectionne à chaque fois les deux arbres les plus petits, on utilise une file de priorité. Ce peut être la classe `Heap` présentée au chapitre 8 ou encore la classe `java.util.PriorityQueue<E>` de la bibliothèque Java.

```
  HuffmanTree buildTree(Collection<Leaf> alphabet) {
    Heap<HuffmanTree> pq = new Heap<HuffmanTree>();
```

Cette file de priorité contient des arbres, c'est-à-dire des valeurs de type `HuffmanTree`. On commence par la remplir avec toutes les feuilles contenues dans l'alphabet passé en argument.

```
    for (Leaf l: alphabet)
      pq.add(l);
```

Puis on applique l'algorithme de construction de l'arbre proprement dit. Tant que la file contient au moins deux éléments, on en retire les deux plus petits, que l'on fusionne en un seul arbre qui est remis dans la file de priorité.

```
    while (pq.size() > 1) {
      HuffmanTree left = pq.removeMin();
      HuffmanTree right = pq.removeMin();
      pq.add(new Node(left, right));
    }
```

Lorsqu'on sort de la boucle, la file ne contient plus qu'un seul arbre, qui est le résultat renvoyé.

```
    return pq.getMin();
  }
```

Une fois l'arbre construit, on peut remplir la table `codes`. Il suffit pour cela de parcourir l'arbre, en maintenant le chemin depuis la racine, et de remplir la table chaque fois qu'on atteint une feuille. Écrivons pour cela une méthode `traverse` dans la classe `HuffmanTree`. Elle prend en arguments le chemin `prefix`, sous la forme d'une chaîne de caractères, et une table `m` à remplir.

```
abstract class HuffmanTree ... {
  ...
  abstract void traverse(String prefix, Map<Character, String> m);
}
```

On définit ensuite cette méthode dans les deux sous-classes. Dans la classe `Leaf`, il suffit de remplir la table `m` en associant la chaîne `prefix` au caractère représenté par la feuille.

```
class Leaf extends HuffmanTree {
  ...
  void traverse(String prefix, Map<Character, String> m) {
    m.put(this.c, prefix);
  }
}
```

Dans la classe `Node`, il s'agit de descendre récursivement dans les deux sous-arbres gauche et droit, en mettant à jour le chemin à chaque fois.

```
class Node extends HuffmanTree {
  ...
  void traverse(String prefix, Map<Character, String> m) {
    this.left.traverse(prefix + '0', m);
    this.right.traverse(prefix + '1', m);
  }
}
```

Ceci achève le code de la construction de l'arbre et du remplissage de la table.

**Encodage et décodage.** Il reste à expliquer comment écrire les deux méthodes qui encodent et décodent chacune respectivement un texte donné. Commençons par la méthode d'encodage. On utilise un `StringBuffer` pour construire le résultat (voir page 44).

```
String encode(String msg) {
  StringBuffer sb = new StringBuffer();
```

Pour chaque caractère de la chaîne à encoder, on concatène son code au résultat.

```
  for (int i = 0; i < msg.length(); i++)
    sb.append(this.codes.get(msg.charAt(i)));
```

Il n'y a plus qu'à renvoyer la chaîne contenue dans `sb`.

```
  return sb.toString();
}
```

Le décodage est un peu plus subtil. La méthode `decode` reçoit en arguments une chaîne de caractères formée de `0` et de `1`, supposée avoir été encodée avec cet objet. Là encore, on utilise un `StringBuffer` pour construire le résultat.

```
String decode(String msg) {
  StringBuffer sb = new StringBuffer();
```

On va avancer progressivement dans le message encodé, en décodant les caractères un par un. La variable `i` indique le caractère courant du message codé et on procède tant qu'on n'a pas atteint la fin du message.

```
  int i = 0;
  while (i < msg.length()) {
```

Pour décoder un caractère, il faut descendre dans l'arbre `this.tree` en suivant le chemin désigné par les `0` et les `1` du message, jusqu'à atteindre une feuille. Une solution simple consiste à écrire une méthode `find` dans la classe `HuffmanTree` pour cela, qui prend en arguments le message `msg` et la position `i`, et renvoie le caractère obtenu. On l'ajoute alors à la chaîne décodée.

```
    char c = this.tree.find(msg, i);
    sb.append(c);
```

Pour mettre à jour la variable `i`, il suffit de l'augmenter de la longueur du code du caractère qui vient juste d'être décodé. Celle-ci est facilement obtenue grâce à la table `this.codes`.

```
    i += this.codes.get(c).length();
  }
```

Une autre solution aurait été de faire renvoyer cette longueur par la méthode `find` mais il n'est pas aisé de renvoyer deux résultats. Une fois sorti de la boucle, on renvoie la chaîne construite.

```
  return sb.toString();
}
```

Il reste à écrire le code de la méthode `find` qui descend dans l'arbre. Comme pour la méthode `traverse` plus haut, on commence par la déclarer dans la classe abstraite `HuffmanTree`

```
abstract class HuffmanTree ... {
  ...
  abstract char find(String s, int i);
}
```

puis on la définit dans chacune des deux sous-classes. Dans la classe `Leaf`, il suffit de renvoyer le caractère contenu dans la feuille.

```
class Leaf extends HuffmanTree {
  ...
  char find(String s, int i) {
    return this.c;
  }
}
```

Dans la classe `Node`, il s'agit de descendre vers la gauche ou vers la droite, selon que le `i`-ième caractère de la chaîne `s` vaut `'0'` ou `'1'`.

```
class Node extends HuffmanTree {
  ...
  char find(String s, int i) {
    if (i == s.length())
      throw new Error("corrupted code; bad alphabet?");
    return (s.charAt(i) == '0' ? this.left : this.right).find(s, i+1);
  }
}
```

On prend soin de tester un éventuel débordement au delà de la fin de la chaîne. Cela peut se produire si on tente de décoder un message qui n'a pas été encodé avec cet arbre-là. Le code complet est donné dans les programmes 34 et 35. Il est important de noter qu'une implémentation réaliste devrait, outre le fait d'utiliser des bits plutôt que des caractères, encoder également l'arbre comme une partie du message ou, à défaut, la distribution des caractères. Sans cette information, il n'est pas possible de décoder.

**Exercice 94.** Écrire une méthode statique

```
Collection<Leaf> buildAlphabet(String s)
```

qui calcule les nombres d'occurrences des différents caractères d'une chaîne `s` et les renvoie sous la forme d'une collection de feuilles. Indication : on pourra utiliser une table de hachage associant des feuilles à des caractères. Une fois cette table remplie, sa méthode `values()` permet de renvoyer la collection de feuilles directement. Solution □

**Programme 34 — Algorithme de Huffman (structure d'arbre)**

```
abstract class HuffmanTree implements Comparable<HuffmanTree> {
  int freq;
  public int compareTo(HuffmanTree that) {
    return this.freq - that.freq;
  }
  abstract void traverse(String prefix, Map<Character, String> m);
  abstract char find(String s, int i);
}

class Leaf extends HuffmanTree {
  final char c;
  Leaf(char c) {
    this.c = c;
    this.freq = 0;
  }
  void traverse(String prefix, Map<Character, String> m) {
    m.put(this.c, prefix);
  }
  char find(String s, int i) {
    return this.c;
  }
}

class Node extends HuffmanTree {
  HuffmanTree left, right;
  Node(HuffmanTree left, HuffmanTree right) {
    this.left = left;
    this.right = right;
    this.freq = left.freq + right.freq;
  }
  void traverse(String prefix, Map<Character, String> m) {
    this.left.traverse(prefix + '0', m);
    this.right.traverse(prefix + '1', m);
  }
  char find(String s, int i) {
    if (i == s.length())
      throw new Error("corrupted code; bad alphabet?");
    return (s.charAt(i) == '0' ? this.left : this.right).find(s, i+1);
  }
}
```

**Programme 35 — Algorithme de Huffman (codage et décodage)**

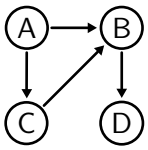
```
class Huffman {
  private HuffmanTree tree;
  private Map<Character, String> codes;

  Huffman(Collection<Leaf> alphabet) {
    if (alphabet.size() <= 1) throw new IllegalArgumentException();
    this.tree = buildTree(alphabet);
    this.codes = new HashMap<Character, String>();
    this.tree.traverse("", this.codes);
  }

  HuffmanTree buildTree(Collection<Leaf> alphabet) {
    Heap<HuffmanTree> pq = new Heap<HuffmanTree>();
    for (Leaf l: alphabet)
      pq.add(l);
    while (pq.size() > 1) {
      HuffmanTree left = pq.removeMin();
      HuffmanTree right = pq.removeMin();
      pq.add(new Node(left, right));
    }
    return pq.getMin();
  }

  String encode(String msg) {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < msg.length(); i++)
      sb.append(this.codes.get(msg.charAt(i)));
    return sb.toString();
  }

  String decode(String msg) {
    StringBuffer sb = new StringBuffer();
    int i = 0;
    while (i < msg.length()) {
      char c = this.tree.find(msg, i);
      sb.append(c);
      i += this.codes.get(c).length();
    }
    return sb.toString();
  }
}
```

# Quatrième partie

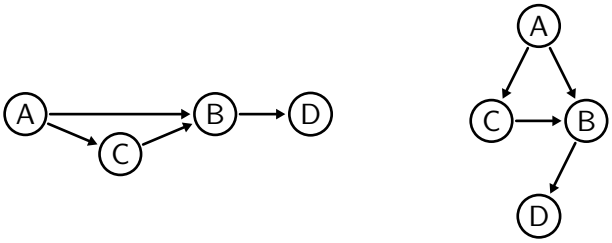
# Graphes

# 15 Définition et représentation

La structure de graphes est une structure de données fondamentale en informatique. Un *graphe orienté* est la donnée d'un ensemble de *sommets* $V$ et d'un ensemble d'*arcs* $E \subseteq V \times V$. Voici un exemple de graphe




où $V = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ et $E = \{(\mathsf{A}, \mathsf{B}), (\mathsf{A}, \mathsf{C}), (\mathsf{C}, \mathsf{B}), (\mathsf{B}, \mathsf{D})\}$. Il est important de comprendre que le dessin importe peu. Ainsi, les deux graphes suivants sont identiques au graphe ci-dessus :




Seule la donnée des ensembles $V$ et $E$ définit le graphe. La relation $E$ n'étant pas nécessairement symétrique, comme dans l'exemple ci-dessus, on parle de *graphe orienté*. Si $(x, y) \in E$ on dit que $y$ est un successeur de $x$ et on note $x \to y$ la présence de cet arc. Un arc d'un sommet vers lui-même, comme sur cet exemple, est appelé une *boucle*. Le degré entrant (resp. sortant) d'un sommet est le nombre d'arcs qui pointent vers ce sommet (resp. qui sortent de ce sommet).

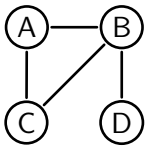
Un *chemin* du sommet $u$ au sommet $v$ est une séquence $x_0, \ldots, x_n$ de sommets tels que $x_0 = u$, $x_n = v$ et $x_i \to x_{i+1}$ pour $0 \leq i < n$ :

$$u = x_0 \to x_1 \to \cdots \to x_{n-1} \to x_n = v$$

La longueur d'un tel chemin est $n$, c'est-à-dire le nombre d'arcs qui le constitue. On note $x_0 \to^\star x_n$ la présence d'un chemin entre les sommets $x_0$ et $x_n$. Il y a toujours un chemin de longueur 0 entre un sommet $u$ et lui-même. Un *chemin simple* est un chemin sans

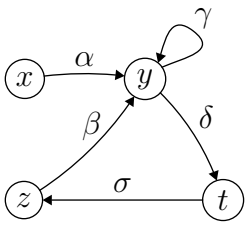
répétition d'arc. Un *cycle* est un chemin simple de $u$ à $u$ de longueur supérieure ou égale à 2. Un graphe orienté qui ne contient pas de cycle est appelé un DAG pour *Directed Acyclic Graph*.

Lorsque la relation $E$ est symétrique, on parle de *graphe non orienté* et on le dessine de cette façon :




On note alors $x-y$ la présence d'un arc entre deux sommets et on dit que ces deux sommets sont *adjacents*[1]. On dit qu'un graphe non orienté est *connexe* lorsqu'il existe toujours un chemin entre deux sommets. Un cycle dans un graphe non orienté est constitué d'au moins trois arcs. Un graphe non orienté connexe et acyclique est appelé un *arbre* (voir la section 16.3).

Les sommets comme les arcs peuvent porter une information ; on parle alors de *graphe étiqueté*. Voici un exemple de graphe orienté étiqueté :




Il est important de noter que l'étiquette d'un sommet n'est pas la même chose que le sommet lui-même. En particulier, deux sommets peuvent porter la même étiquette. Formellement, un graphe étiqueté est donc la donnée supplémentaire de deux fonctions donnant respectivement l'étiquette d'un sommet de $V$ et l'étiquette d'un arc de $E$.

## 15.1 Matrice d'adjacence

On considère dans cette section le cas où les sommets sont représentés par des entiers, et plus précisément par les entiers consécutifs $0, \ldots, N-1$. Dit autrement, on a $V = \{0, \ldots, N-1\}$. Le plus naturel pour représenter un tel graphe est sans doute une matrice $M$, de taille $N \times N$ où chaque élément $M_{i,j}$ indique la présence d'un arc entre les sommets $i$ et $j$. En supposant des graphes non étiquetés, il suffit d'utiliser une matrice de booléens :

```
class AdjMatrix {
  int n; // les sommets sont 0,...,n-1
  boolean[][] m;
}
```

1. Le vocabulaire de la théorie des graphes diffère selon que les graphes sont ou non orientés. Ainsi, on parle parfois de nœuds plutôt que de sommets, d'arêtes plutôt que d'arcs, etc. Dans la suite, nous nous en tiendrons à un unique vocabulaire. En particulier, nous verrons que (1) la représentation en machine d'un graphe orienté peut être également utilisée pour un graphe non orienté et (2) beaucoup d'algorithmes sur les graphes orientés s'appliquent sans changement sur les graphes non orientés.

**Programme 36 — Graphes orientés représentés par une matrice d'adjacence**

```
class AdjMatrix {

  final int n; // les sommets sont 0,...,n-1
  final boolean[][] m;

  AdjMatrix(int n) {
    this.n = n;
    this.m = new boolean[n][n];
  }

  boolean hasEdge(int x, int y) {
    return this.m[x][y];
  }

  void addEdge(int x, int y) {
    this.m[x][y] = true;
  }

  void removeEdge(int x, int y) {
    this.m[x][y] = false;
  }
}
```

(On conserve ici le nombre de sommets dans un champ `n`, même si celui-ci est égal à la dimension de la matrice.) Le constructeur prend la valeur de `n` et alloue une matrice de dimension `n` × `n`.

```
  AdjMatrix(int n) {
    this.n = n;
    this.m = new boolean[n][n];
  }
```

Cette matrice est initialisée avec `false`, ce qui représente donc un graphe ne contenant aucun arc. Les opérations d'ajout, de suppression ou de test de présence d'un arc sont immédiates. Pour des graphes orientés, elles sont données programme 36 page 177. Elles ont toutes une complexité $O(1)$ c'est-à-dire un coût constant.

Une matrice d'adjacence occupe clairement un espace quadratique, c'est-à-dire en $O(N^2)$. C'est une représentation adaptée aux graphes *denses*, c'est-à-dire aux graphes où le nombre d'arcs $E$ est justement de l'ordre de $N^2$.

**Exercice 95.** Modifier les matrices d'adjacence pour des graphes où les arcs sont étiquetés (par exemple par des entiers). Solution □

## 15.2 Listes d'adjacence

Dans le cas de graphes peu denses, une alternative aux matrices d'adjacence consiste à utiliser un tableau donnant, pour chaque sommet, la liste de ses successeurs. On parle de *listes d'adjacence*. Ainsi pour le graphe dessiné page 176, ces listes sont $[y]$ pour le sommet $x$, $[y, t]$ pour le sommet $y$, etc. Il nous suffit donc d'utiliser un tableau de listes. Cependant, tester la présence d'un élément dans une liste coûte un peu cher, et donc tester la présence d'un arc dans le graphe le sera également. Il semble plus opportun d'utiliser par exemple une table de hachage pour représenter les successeurs d'un sommet donné, par exemple avec la bibliothèque `HashSet` de Java. Puisqu'on en est arrivé à l'idée d'utiliser une table de hachage, où les sommets n'ont plus besoin d'être les entiers $0, \ldots, N-1$, autant pousser cette idée jusqu'au bout et représenter un graphe par une table de hachage associant à chaque sommet l'ensemble de ses successeurs, lui-même représenté par une table de hachage. On s'affranchit ainsi complètement du fait que les sommets sont des entiers consécutifs, et même du fait que ce sont des entiers. On conserve cependant l'appellation de listes d'adjacence, même s'il n'y a pas de liste dans cette représentation.

On construit donc une structure de graphe générique, paramétrée par un type de sommets `V`, avec l'adjacence stockée dans un champ `adj`.

```
class Graph<V> {
  private Map<V, Set<V>> adj;
```

Le code est donné programme 37 page 179. Il utilise les classes `HashMap` et `HashSet` de la bibliothèque standard et on suppose donc que la classe `V` redéfinit correctement les méthodes `hashCode` et `equals`. On prendra le temps de bien comprendre les petites différences avec les matrices d'adjacence. En particulier, il convient de traiter correctement les cas de figure où la table d'adjacence d'un sommet donné n'existe pas. En particulier, `addEdge(x, y)` s'assure que le sommet `x` est bien un sommet du graphe, en appelant `addVertex`. Si le sommet existe déjà, cette méthode ne fera rien. Si en revanche il n'existe pas encore, alors sa table d'adjacence sera créée et ajoutée dans `adj`. De même, la méthode `removeEdge` prend soin de vérifier que l'adjacence du sommet `x` existe bien, et ne fait rien sinon.

Les opérations de test, d'ajout et de suppression d'un arc sont toutes en $O(1)$ (amorti) car il ne s'agit que d'opérations d'ajout et de suppression dans des tables de hachage. Concernant le coût en espace, les listes d'adjacence ont une complexité optimale de $O(N+E)$.

Pour les algorithmes sur les graphes que nous allons écrire dans le chapitre suivant, il est nécessaire de pouvoir accéder à l'ensemble des sommets du graphe d'une part et à l'ensemble des successeurs d'un sommet donné d'autre part. Plutôt que d'exposer la table de hachage qui contient la relation d'adjacence (on l'a d'ailleurs déclarée comme privée), il suffit d'exporter deux méthodes `vertices` et `successors` (voir à la fin de la classe `Graph`). Dès lors, pour parcourir tous les sommets d'un graphe `g`, il suffit d'écrire

```
  for (V v: g.vertices()) ...
```

et pour parcourir tous les successeurs d'un sommet `v` de `g`, il suffit d'écrire

```
  for (V w: g.successors(v)) ...
```

**Programme 37 — Graphes orientés représentés par des « listes » d'adjacence**

```
class Graph<V> {

  private Map<V, Set<V>> adj;

  Graph() {
    this.adj = new HashMap<V, Set<V>>();
  }

  void addVertex(V x) {
    Set<V> l = this.adj.get(x);
    if (l == null) this.adj.put(x, new HashSet<V>());
  }

  boolean hasEdge(V x, V y) {
    Set<V> l = this.adj.get(x);
    return l != null && l.contains(y);
  }

  void addEdge(V x, V y) {
    addVertex(x);
    this.adj.get(x).add(y);
  }

  void removeEdge(V x, V y) {
    Set<V> l = this.adj.get(x);
    if (l != null) l.remove(y);
  }

  Set<V> vertices() {
    return this.adj.keySet();
  }

  Set<V> successors(V v) {
    return this.adj.get(v);
  }
}
```

Le coût de ces parcours est respectivement $O(V)$ et $O(d(\mathtt{v}))$ où $d(\mathtt{v})$ est le degré sortant du sommet `v`, ce qui est optimal.

**Exercice 96.** Ajouter une méthode `nbVertices` donnant le nombre de sommets en temps constant. Solution □

**Exercice 97.** Ajouter une méthode `int nbEdges()` donnant le nombre d'arcs en temps constant. Indication : maintenir le nombre d'arcs dans la structure de graphe, en mettant à jour sa valeur dans `addEdge` et `removeEdge`. Solution □

**Exercice 98.** Modifier les listes d'adjacence pour des graphes où les arcs sont étiquetés (par exemple par des entiers). On pourra remplacer l'ensemble des successeurs du sommet $x$ par un dictionnaire (`HashMap`) donnant pour chaque successeur $y$ l'étiquette de l'arc $x \to y$. Solution □

## 15.3 Graphe non orienté

Le plus simple pour représenter des graphes non orientés est de conserver la même structure que pour des graphes orientés, mais en maintenant l'invariant que pour chaque arc $a \to b$ on a également l'arc $b \to a$. Cette redondance d'information peut paraître inutilement coûteuse mais c'est là la meilleure solution. En particulier, de nombreux algorithmes sur les graphes orientés s'appliquent également aux graphes non orientés. Avec ce choix de représentation, on pourra les réutiliser directement.

Pour mettre cette idée en pratique, il suffit de modifier les deux méthodes `addEdge` et `removeEdge` pour qu'elles opèrent de façon symétrique sur les arcs. Avec une matrice d'adjacence, par exemple, la méthode `addEdge` devient

```
void addEdge(int x, int y) {
  this.m[x][y] = true;
  this.m[y][x] = true;
}
```

On modifierait de même la méthode `removeEdge`.

Une façon élégante de le faire consiste à hériter de la classe réalisant les graphes orientés et à y redéfinir uniquement les méthodes `addEdge` et `removeEdge`. Ainsi, le programme 38 page 181 réalise des graphes non orientés au-dessus de la classe `Graph<V>` du programme 37 page 179. Ainsi, on hérite de toutes les méthodes inchangées, telles que `addVertex`, `hasEdge`, `vertices`, `successors`, etc. Pour pouvoir faire cela, il faut déclarer le champ `adj` de la classe `Graph` comme `protected` (au lieu de `private`).

**Exercice 99.** Expliquer comment exploiter la structure *union-find* (chapitre 9) pour tester si un graphe non orienté est connexe. Solution □

**Exercice 100.** Expliquer comment exploiter la structure *union-find* (chapitre 9) pour tester si un graphe non orienté est acyclique. Solution □

**Programme 38 — Graphes non orientés**

```
class Undirected<V> extends Graph<V> {

  @Override
  void addEdge(V x, V y) {
    super.addEdge(x, y);
    super.addEdge(y, x);
  }

  @Override
  void removeEdge(V x, V y) {
    super.removeEdge(x, y);
    super.removeEdge(y, x);
  }

}
```

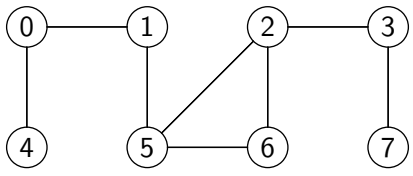
# 16 Algorithmes élémentaires sur les graphes

Dans ce chapitre, on utilise la structure de graphes générique `Graph<V>` du programme 37 page 179, où la classe `V` désigne le type des sommets. On note $V$ le nombre de sommets et $E$ le nombre d'arcs. La représentation du graphe étant par listes d'adjacence, l'occupation mémoire du graphe est $O(V+E)$. Lorsque les exemples impliquent des graphes non orientés, on suppose que la représentation reste la même, avec une adjacence symétrique.

## 16.1 Parcours de graphes

Dans cette section, on présente deux algorithmes qui permettent de parcourir tous les sommets d'un graphe (qu'il soit orienté ou non). Ces parcours, ou des variantes de ces parcours, se retrouvent dans de très nombreux algorithmes sur les graphes. Il convient donc de les comprendre et de les maîtriser.

Avant même de rentrer dans les détails d'un parcours particulier, on peut déjà comprendre que la présence possible de *cycle* dans un graphe rend le parcours plus complexe que celui d'une liste ou d'un arbre. Prenons l'exemple du graphe non orienté suivant [1] :




(16.1)

Quel que soit son fonctionnement, un parcours qui démarrerait du sommet 4 parviendra à un moment où à un autre au sommet 5 et il ne doit pas entrer alors dans un boucle infinie du fait de la présence du cycle $5-2-6$. Dans le cas d'un arbre, un tel cycle n'est pas possible et nous avons pu écrire le parcours infixe d'un arbre assez facilement (voir page 77). L'arbre vide `null` était la condition d'arrêt du parcours récursif. Dans le cas d'une liste chaînée, nous avons évoqué la possibilité de listes cycliques (section 4.4). Nous avons certes donné un algorithme pour détecter un tel cycle (l'algorithme du lièvre et de

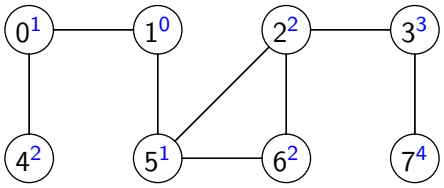
1. Les exemples de ce chapitre sont inspirés de *Introduction to Algorithms* [5].

la tortue, page 59), mais il exploite de façon cruciale le fait que chaque élément de la liste ne possède qu'au plus un successeur. Dans le cas d'un graphe, ce n'est plus vrai.

Nous allons donc devoir *marquer* les sommets atteints par le parcours, d'une façon ou d'une autre. Nous utiliserons ici une table de hachage. Une autre solution consiste à modifier directement les sommets, si le type `V` le permet.

### 16.1.1 Parcours en largeur

Le premier parcours que nous présentons, dit parcours en largeur (en anglais *breadth-first search* ou BFS), consiste à explorer le graphe « en cercles concentriques » en partant d'un sommet particulier $s$ appelé la source. On parcourt d'abord les sommets situés à une distance d'un arc de $s$, puis les sommets situés à une distance de deux arcs, etc. Sur le graphe donné en exemple plus haut (16.1), et en partant du sommet 1, on explore en premier lieu les sommets 0 et 5, directement reliés au sommet 1, puis les sommets 2, 4 et 6, puis le sommet 3, puis enfin le sommet 7. On le redessine ici avec les distances à la source (le sommet 1) indiquées en exposant.




Pour mettre en œuvre ce parcours, on va utiliser une table de hachage. Elle contiendra les sommets déjà atteints par le parcours, en leur associant de plus la distance à la source. On renverra cette table comme résultat du parcours. On écrit donc une méthode statique avec le type suivant :

```
static <V> HashMap<V, Integer> bfs(Graph<V> g, V source) {
```

La table, appelée ici `visited`, est créée au tout début de la méthode et on y met initialement la source, avec la distance 0.

```
    HashMap<V, Integer> visited = new HashMap<V, Integer>();
    visited.put(source, 0);
```

Le parcours proprement dit repose sur l'utilisation d'une *file*, dans laquelle les sommets vont être insérés au fur et à mesure de leur découverte. L'idée est que la file contient, à chaque instant, des sommets situés à distance $d$ de la source, suivis de sommets à distance $d+1$ :

$$\leftarrow \boxed{\text{sommets à distance } d \mid \text{sommets à distance } d+1} \leftarrow$$

C'est là la matérialisation de notre idée de « cercles concentriques », plus précisément des deux cercles concentriques consécutifs en cours de considération. Cette propriété est cruciale pour la correction du parcours en largeur [2]. Ici on utilise la bibliothèque `LinkedList` pour réaliser la file. Initialement, elle contient uniquement la source.

```
    Queue<V> q = new LinkedList<V>();
    q.add(source);
```

2. Pour une preuve détaillée de la correction du parcours en largeur, on pourra consulter [5, chap. 23].

Un autre invariant important est que tout sommet présent dans la file est également présent dans la table `visited`. On procède alors à une boucle, tant que la file n'est pas vide. Le cas échéant, on extrait le premier élément `v` de la file et on récupère sa distance `d` à la source dans `visited`.

```
    while (!q.isEmpty()) {
      V v = q.poll();
      int d = visited.get(v);
```

On examine alors chaque successeur `w` de `v`.

```
      for (V w : g.successors(v))
```

S'il n'avait pas encore été découvert, c'est-à-dire s'il n'était pas dans la table `visited`, alors on l'ajoute dans la file d'une part, et dans la table `visited` avec la distance `d+1` d'autre part.

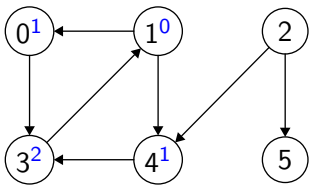
```
        if (!visited.containsKey(w)) {
          q.add(w);
          visited.put(w, d+1);
        }
```

Ceci achève la boucle sur les successeurs de `v`. On passe alors à l'élément suivant de la file, et ainsi de suite. Une fois sorti de la boucle principale, le parcours est achevé et on renvoie la table `visited`.

```
    }
    return visited;
  }
```

Le code complet est donné programme 39 page 186. Il s'applique aussi bien à un graphe non orienté qu'à un graphe orienté. La complexité est facile à déterminer. Chaque sommet est mis dans la file au plus une fois et donc examiné au plus une fois. Chaque arc est donc considéré au plus une fois, lorsque son origine est examinée. La complexité est donc $O(V + E)$, ce qui est optimal. La complexité en espace est $O(V)$ car la file, comme la table de hachage, peut contenir (presque) tous les sommets dans le pire des cas.

On note qu'il peut rester des sommets non atteints par le parcours en largeur. Ce sont les sommets $v$ pour lesquels il n'existe pas de chemin entre la source et $v$. Sur le graphe suivant, en partant de la source 1, seuls les sommets 1, 0, 3 et 4 seront atteints.




Dit autrement, le parcours en largeur détermine l'ensemble des sommets accessibles depuis la source, et donne même pour chacun la distance minimale en nombre d'arcs depuis la source.

Comme on l'a fait remarquer plus haut, la file a une structure bien particulière, avec des sommets à distance $d$, suivis de sommets à distance $d + 1$. On comprend donc que la

**Programme 39 — Parcours en largeur (BFS)**

```
class BFS {

  static <V> HashMap<V, Integer> bfs(Graph<V> g, V source) {
    HashMap<V, Integer> visited = new HashMap<V, Integer>();
    visited.put(source, 0);
    Queue<V> q = new LinkedList<V>();
    q.add(source);
    while (!q.isEmpty()) {
      V v = q.poll();
      int d = visited.get(v);
      for (V w : g.successors(v))
        if (!visited.containsKey(w)) {
          q.add(w);
          visited.put(w, d+1);
        }
    }
    return visited;
  }

}
```

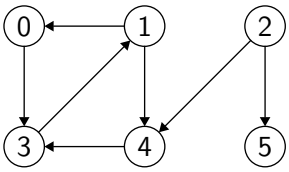
structure de file n'est pas vraiment nécessaire. Deux « sacs » suffisent, l'un contenant les sommets à distance $d$ et l'autre les sommets à distance $d+1$. On peut les matérialiser par exemple par des listes. Lorsque le sac $d$ vient à s'épuiser, on le remplit avec le contenu du sac $d+1$, qui est lui-même vidé. (On les échange, c'est plus simple.) Cela ne change en rien la complexité.

**Exercice 101.** Modifier la méthode `bfs` pour conserver le chemin entre la source et chaque sommet atteint par le parcours. Une façon simple de procéder consiste à stocker, pour chaque sommet atteint, le sommet qui a permis de l'atteindre, par exemple dans une table de hachage. Le chemin est donc décrit « à l'envers », du sommet atteint vers la source. Solution □

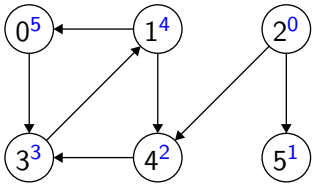
**Exercice 102.** En s'inspirant du parcours en largeur d'un graphe, écrire une méthode qui parcourt les nœuds d'un *arbre* en largeur. Solution □

### 16.1.2 Parcours en profondeur

Le second parcours de graphe que nous présentons, dit parcours en profondeur (en anglais *depth-first search* ou DFS) applique l'algorithme de rebroussement (*backtracking*) vu au chapitre 12 : tant qu'on peut progresser en suivant un arc, on le fait, et sinon on fait machine arrière. Comme pour le parcours en largeur, on marque les sommets atteints au fur et à mesure de leur découverte, pour éviter de tomber dans un cycle. Prenons l'exemple du graphe suivant




et d'un parcours en profondeur qui démarre du sommet 2. Deux arcs sortent de ce sommet, $2 \to 4$ et $2 \to 5$ et on choisit (arbitrairement) de considérer en premier l'arc $2 \to 5$. On passe donc au sommet 5. Aucun arc ne sort de 5 ; c'est une impasse. On revient alors au sommet 2, dont on considère maintenant le second arc sortant, $2 \to 4$. De 4, on ne peut que suivre l'arc $4 \to 3$ puis, de même, de 3 on ne peut que suivre l'arc $3 \to 1$. Du sommet 1 sortent deux arcs, $1 \to 0$ et $1 \to 4$. On choisit de suivre en premier lieu l'arc $1 \to 4$. Il mène à un sommet déjà visité, et on fait donc machine arrière. De retour sur 1, on considère l'autre arc, $1 \to 0$, qui nous mène à 0. De là le seul arc sortant mène à 3, là encore déjà visité. On revient donc à 0, puis à 1, puis à 3, puis à 4, puis enfin à 2. Le parcours est terminé. Si on redessine le graphe avec l'ordre de découverte des sommets en exposant, on obtient ceci :




Comme pour le parcours en largeur, on va utiliser une table de hachage contenant les sommets déjà atteints par le parcours. Elle donnera l'ordre de découverte de chaque sommet. Sans surprise, on choisit d'écrire le parcours en profondeur comme une méthode récursive. Pour éviter de lui passer la table et le graphe en arguments systématiquement, on va écrire le parcours en profondeur dans une méthode dynamique, dans une classe dont le constructeur reçoit le graphe en argument :

```
class DFS<V> {
  private final Graph<V> g;
  private final HashMap<V, Integer> visited;
  private int count;

  DFS(Graph<V> g) {
    this.g = g;
    this.visited = new HashMap<V, Integer>();
    this.count = 0;
  }
```

Le champ `count` est le compteur qui nous servira à associer, dans la table `visited`, chaque sommet avec l'instant de sa découverte. Le parcours en profondeur proprement dit est alors écrit dans une méthode récursive `dfs` prenant un sommet `v` en argument. Son code est d'une simplicité enfantine :

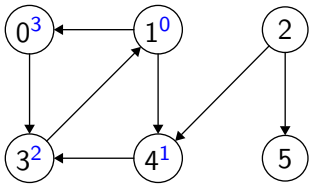
```
  void dfs(V v) {
    if (this.visited.containsKey(v)) return;
    this.visited.put(v, this.count++);
```

```
    for (V w : this.g.successors(v))
      dfs(w);
  }
```

Si le sommet `v` a déjà été atteint, on ne fait rien. Sinon, on le marque comme déjà atteint, en lui donnant le numéro `count`. Puis on considère chaque successeur `w`, sur lequel on lance récursivement un parcours en profondeur. On ne peut imaginer plus simple. Un détail, cependant, est crucial : on a ajouté `v` dans la table `visited` *avant* de considérer ses successeurs. C'est là ce qui nous empêche de tourner indéfiniment dans un cycle.

La complexité est $O(V + E)$, par le même argument que pour le parcours en largeur. Le parcours en profondeur est donc également optimal. La complexité en espace est légèrement plus subtile, car il faut comprendre que c'est ici la pile des appels récursifs qui contient les sommets en cours de visite (et joue le rôle de la file dans le parcours en largeur). Dans le pire des cas, tous les sommets peuvent être présents sur la pile, d'où une complexité en espace $O(V)$.

Comme le parcours en largeur, le parcours en profondeur a déterminé l'ensemble des sommets accessibles depuis la source `v`. Voici un autre exemple où le parcours en profondeur est lancé à partir du sommet 1.




Les sommets 2 et 5 ne sont pas atteints. Dans de nombreuses applications du parcours en profondeur, on souhaite parcourir *tous* les sommets du graphe, et non pas seulement ceux qui sont accessibles depuis un certain sommet `v`. Pour cela, il suffit de lancer la méthode `dfs` sur tous les sommets du graphe :

```
void dfs() {
  for (V v : this.g.vertices())
    dfs(v);
}
```

(La surcharge nous permet d'appeler également cette méthode `dfs`.) Pour un sommet déjà visité par un précédent parcours, l'appel `dfs(v)` va nous redonner la main immédiatement, et sera donc sans effet. Le code complet est donné programme 40 page 189. On l'a complété par une méthode `getNum` qui permet de consulter le contenu de `visited` (une fois le parcours effectué).

Comme on vient de l'expliquer, le parcours en profondeur est, comme le parcours en largeur, un moyen de déterminer l'existence d'un chemin entre un sommet particulier, la source, et les autres sommets du graphe. Si c'est là le seul objectif (par exemple, la distance minimale ne nous intéresse pas), alors le parcours en profondeur est généralement plus efficace. En effet, son occupation mémoire (la pile d'appels) sera le plus souvent bien inférieure à celle du parcours en largeur. L'exemple typique est celui d'un arbre, où l'occupation mémoire sera limitée par la hauteur de l'arbre pour un parcours en profondeur, mais pourra être aussi importante que l'arbre tout entier dans le cas d'un parcours en largeur. Le parcours en profondeur a beaucoup d'autres application, qui dépassent largement le cadre de ce cours ; voir par exemple *Introduction to Algorithms* [5].

**Programme 40 — Parcours en profondeur (DFS)**

```
class DFS<V> {

  private final Graph<V> g;
  private final HashMap<V, Integer> visited;
  private int count;

  DFS(Graph<V> g) {
    this.g = g;
    this.visited = new HashMap<V, Integer>();
    this.count = 0;
  }

  void dfs(V v) {
    if (this.visited.containsKey(v)) return;
    this.visited.put(v, this.count++);
    for (V w : this.g.successors(v))
      dfs(w);
  }

  void dfs() {
    for (V v : this.g.vertices())
      dfs(v);
  }

  int getNum(V v) {
    return this.visited.get(v);
  }
}
```

**Exercice 103.** Modifier la classe `DFS` pour conserver le chemin entre la source et chaque sommet atteint par le parcours. Une façon simple de procéder consiste à stocker, pour chaque sommet atteint, le sommet qui a permis de l'atteindre, par exemple dans une table de hachage. Le chemin est donc décrit « à l'envers », du sommet atteint vers la source.
Solution □

**Exercice 104.** En quoi le parcours en profondeur est-il différent/semblable du parcours infixe d'un arbre décrit page 77 ?
Solution □

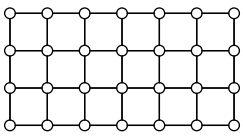
**Exercice 105.** Réécrire la méthode `dfs` en utilisant une boucle `while` plutôt qu'une méthode récursive. Indication : on utilisera une *pile* contenant des sommets à partir desquels il faut effectuer le parcours en profondeur. Il n'est pas nécessaire que les sommets soient numérotés exactement comme dans la version récursive. Attention : ce n'est pas aussi simple que remplacer la file par une pile dans le code du parcours en largeur.
Solution □

**Exercice 106.** Soit un graphe orienté $G$ ne contenant pas de cycle (un DAG). Un *tri topologique* de $G$ est une liste de ses sommets compatible avec les arcs, c'est-à-dire où un sommet $x$ apparaît avant un sommet $y$ dès lors qu'on a un arc $x \to y$. Modifier le programme 40 pour qu'il renvoie un tri topologique, sous la forme d'une méthode `List<V> topologicalSort()`. On pourra introduire une liste de type `LinkedList<V>` comme un nouveau champ, dans laquelle le sommet `v` est ajouté par la méthode `dfs`.
Solution □

**Exercice 107.** Le parcours en profondeur peut être modifié pour détecter la présence d'un cycle dans le graphe. Lorsque la méthode `dfs` tombe sur un sommet déjà visité, on ne sait pas *a priori* si on vient de trouver un cycle ; il peut s'agir en effet d'un sommet déjà atteint par un autre chemin, parallèle. Il faut donc modifier le marquage des sommets pour utiliser non pas deux états (atteint / non atteint) mais trois : non atteint / en cours de visite / visité. Modifier la classe `DFS` en conséquence, par exemple en ajoutant une méthode `boolean hasCycle()` qui détermine la présence d'un cycle.

Question subsidiaire : Dans le cas très particulier d'une liste simplement chaînée, en quoi cela est-il plus/moins efficace que l'algorithme du lièvre et de la tortue (page 59) ?
Solution □

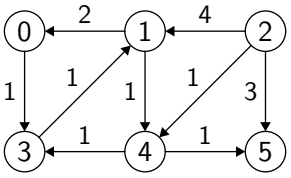
**Exercice 108.** On peut utiliser un parcours en profondeur pour construire un labyrinthe parfait — c'est-à-dire un labyrinthe où il existe un chemin et un seul entre deux cases — dans une grille $n \times m$. Pour cela, on considère au départ le graphe où toutes les cases de la grilles sont reliées à leurs voisines :



Puis on effectue un parcours en profondeur à partir d'un sommet quelconque (par exemple celui en haut à gauche, mais ce n'est pas important). Quand on parcourt les successeurs d'un sommet, on le fait dans un ordre aléatoire. Une fois le parcours effectué, le labyrinthe est obtenu en considérant qu'on peut passer d'un sommet à un autre si l'arc correspondant a été emprunté pendant le parcours en profondeur.
Solution □

## 16.2 Plus court chemin : algorithme de Dijkstra

On considère ici des graphes dont les arcs sont étiquetés par des poids (entiers ou réels) *positifs ou nuls* et on s'intéresse au problème de trouver le plus court chemin d'un sommet à un autre sommet, la longueur n'étant plus le nombre d'arcs mais la somme des poids le long du chemin. Ainsi, si on considère le graphe




alors le plus court chemin du sommet 2 au sommet 0 est de longueur 5. Il s'agit du chemin $2 \to 4 \to 3 \to 1 \to 0$. En particulier, il est plus court que le chemin $2 \to 1 \to 0$, de longueur 6, même si celui-ci contient moins d'arcs. De même, le plus court chemin du sommet 2 au sommet 5 est de longueur 2, en passant par le sommet 4.

L'algorithme que nous présentons ici pour résoudre ce problème s'appelle l'algorithme de Dijkstra. C'est une variation du parcours en largeur. Comme pour ce dernier, on se donne un sommet de départ, la source, et on procède par « cercles concentriques ». La différence est ici que les rayons de ces cercles représentent une distance en terme de poids total et non en terme du nombre d'arcs. Ainsi dans l'exemple ci-dessus, en partant de la source 2, on atteint d'abord les sommets à distance 1 (à savoir 4), puis à distance 2 (à savoir 3 et 5), puis à distance 3 (à savoir 1), puis enfin à distance 5 (à savoir 0). La difficulté de mise en œuvre vient du fait qu'on peut atteindre un sommet avec une certaine distance, par exemple le sommet 5 avec l'arc $2 \to 5$, puis trouver plus tard un chemin plus court en empruntant d'autres arcs, par exemple $2 \to 4 \to 5$. On ne peut plus se contenter d'une file comme dans le parcours en largeur ; on va utiliser une *file de priorité* (voir chapitre 8). Elle contiendra les sommets déjà atteints, ordonnés par distance à la source. Lorsqu'un meilleur chemin est trouvé, le sommet est remis dans la file avec une plus grande priorité, c'est-à-dire une distance plus petite [3].

Décrivons le code Java de l'algorithme de Dijkstra. Il faut se donner le poids de chaque arc. On pourrait envisager modifier la structure de graphe pour qu'elle contienne le poids de chaque arc. De manière équivalente, on choisit ici de se donner plutôt une fonction de poids, comme un objet qui implémente l'interface suivante

```
interface Weight<V> {
  int weight(V x, V y);
}
```

c'est-à-dire qui fournit une méthode `weight` donnant le poids de l'arc `x` $\to$ `y`. Cette méthode ne sera appelée sur des arguments `x` et `y` que lorsqu'il existe effectivement un arc entre `x` et `y` dans le graphe.

Pour réaliser la file de priorité, on utilise la bibliothèque Java `PriorityQueue`. Elle va contenir des paires $(v, d)$ où $v$ est un sommet et $d$ sa distance à la source. On représente ces paires avec la classe

3. Une autre solution consisterait à utiliser une structure de file de priorité où il est possible de *modifier* la priorité d'un élément se trouvant déjà dans la file. Bien que de telles structures existent, elles sont complexes à mettre en œuvre et, bien qu'asymptotiquement meilleures, leur utilisation n'apporte pas nécessairement un gain en pratique. La solution que nous présentons ici est un très bon compromis.

```
class Node<V> {
  final V node;
  final int dist;
}
```

Pour que ces paires soient effectivement ordonnées par la distance, et utilisées en conséquence par la classe `PriorityQueue`, il faut que la classe `Node` implémente l'interface `Comparable<Node<V>>`, et fournisse donc une méthode `compareTo`. Le code est immédiat ; il est donné page 194.

On en vient au code de l'algorithme proprement dit. On l'écrit comme une méthode `shortestPaths`, qui prend le graphe, la source et la fonction de poids en arguments, et qui renvoie une table donnant les sommets atteints et leur distance à la source.

```
  static <V> HashMap<V, Integer>
         shortestPaths(Graph<V> g, V source, Weight<V> w) {
```

On commence par créer un ensemble `visited` contenant les sommets pour lesquels on a déjà trouvé le plus court chemin.

```
    HashSet<V> visited = new HashSet<V>();
```

Puis on crée une table `distance` contenant les distances déjà connues. On y met initialement la source avec la distance 0. Les distances dans cette table ne sont pas forcément optimales ; elles pourront être améliorées au fur et à mesure du parcours.

```
    HashMap<V, Integer> distance = new HashMap<V, Integer>();
    distance.put(source, 0);
```

Enfin, on crée la file de priorité, `pqueue`, et on y insère la source avec la distance 0.

```
    PriorityQueue<Node<V>> pqueue = new PriorityQueue<Node<V>>();
    pqueue.add(new Node<V>(source, 0));
```

Comme pour le parcours en largeur, on procède alors à une boucle, tant que la file n'est pas vide.

```
    while (!pqueue.isEmpty()) {
```

Le cas échéant, on extrait le premier élément de la file. S'il appartient à `visited`, c'est que l'on a déjà trouvé le plus court chemin jusqu'à ce sommet. On l'ignore donc, en passant directement à l'itération suivante de la boucle.

```
      Node<V> n = pqueue.poll();
      if (visited.contains(n.node)) continue;
```

Cette situation peut effectivement se produire lorsqu'un premier chemin est trouvé puis un autre, plus court, trouvé plus tard. Ce dernier passe alors dans la file de priorité devant le premier. Lorsque le chemin plus long finira par sortir de la file, il faudra l'ignorer. Si le sommet n'appartient pas à `visited`, c'est qu'on vient de déterminer le plus court chemin. On ajoute donc le sommet à `visited`.

```
      visited.add(n.node);
```

Puis on examine chaque successeur `v`. La distance à `v` en empruntant l'arc correspondant est la somme de la distance à `n.node`, c'est-à-dire `n.dist`, et du poids de l'arc, donné par la méthode `w.weight`.

```
      for (V v: g.successors(n.node)) {
        int d = n.dist + w.weight(n.node, v);
```

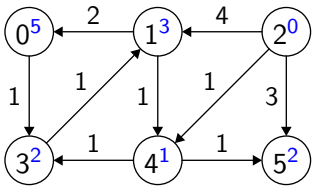
Plusieurs cas de figure sont possibles pour le sommet `v`. Soit c'est la première fois qu'on l'atteint, soit il était déjà dans `distance`. Dans ce dernier cas, on peut ou non améliorer la distance à `v` en passant par `n.node`. On regroupe les cas où `distance` doit être mise à jour dans un seul test.

```
        if (!distance.containsKey(v) || d < distance.get(v)) {
          distance.put(v, d);
          pqueue.add(new Node<V>(v, d));
        }
      }
    }
```

On a d'une part mis à jour `distance` et d'autre part inséré `v` dans la file avec la nouvelle distance `d`. Une fois tous les successeurs traités, on réitère la boucle principale. Une fois qu'on est sorti de celle-ci, tous les sommets atteignables sont dans `distance`, avec leur distance minimale à la source. C'est ce que l'on renvoie.

```
    return distance;
  }
```

Le code complet est donné programme 41 page 194. Le résultat de l'algorithme de Dijkstra sur le graphe donné en exemple plus haut, à partir de la source 2, est ici dessiné avec les distances obtenues au final pour chaque sommet en exposant :




Sur cet exemple, tous les sommets ont été atteints par le parcours. Comme pour les parcours en largeur et en profondeur, ce n'est pas toujours le cas : seuls les sommets pour lesquels il existe un chemin depuis la source seront atteints.

**Complexité.** Évaluons la complexité de l'algorithme de Dijkstra, dans le pire des cas. La file de priorité peut contenir jusqu'à $E$ éléments, car l'algorithme visite chaque arc au plus une fois, et chaque considération d'un arc peut conduire à l'insertion d'un élément dans la file. En supposant que les opérations `add` et `poll` de la file de priorité ont un coût logarithmique (c'est le cas pour la bibliothèque `PriorityQueue` et pour les files de priorité décrites au chapitre 8), chaque opération sur la file a donc un coût $O(\log E)$, c'est-à-dire $O(\log V)$ car $E \leq V^2$. D'où un coût total $O(E \log V)$.

**Programme 41 — Algorithme de Dijkstra (plus court chemin)**

```
interface Weight<V> {
  int weight(V x, V y);
}

class Node<V> implements Comparable<Node<V>> {
  final V node;
  final int dist;
  Node(V node, int dist) {
    this.node = node;
    this.dist = dist;
  }
  public int compareTo(Node<V> n) {
    return this.dist - n.dist;
  }
}

class Dijkstra {

  static <V> HashMap<V, Integer>
          shortestPaths(Graph<V> g, V source, Weight<V> w) {
    HashSet<V> visited = new HashSet<V>();
    HashMap<V, Integer> distance = new HashMap<V, Integer>();
    distance.put(source, 0);
    PriorityQueue<Node<V>> pqueue = new PriorityQueue<Node<V>>();
    pqueue.add(new Node<V>(source, 0));
    while (!pqueue.isEmpty()) {
      Node<V> n = pqueue.poll();
      if (visited.contains(n.node)) continue;
      visited.add(n.node);
      for (V v: g.successors(n.node)) {
        int d = n.dist + w.weight(n.node, v);
        if (!distance.containsKey(v) || d < distance.get(v)) {
          distance.put(v, d);
          pqueue.add(new Node<V>(v, d));
        }
      }
    }
    return distance;
  }

}
```

**Correction.** Il n'est pas complètement évident de se persuader que l'algorithme de Dijkstra est correct. Montrons qu'à la fin de la méthode `shortestPaths`, la table `visited` contient exactement les sommets atteignables depuis la source et `distance` donne pour ces sommets la longueur du plus court chemin. On le fait en établissant des *invariants de boucle*, c'est-à-dire des propriétés qui sont vraies à chaque tour de la boucle `while` constituant le cœur de l'algorithme de Dijkstra. Les deux premiers invariants stipulent que la source fait toujours partie des sommets déjà considérés et que sa distance est toujours égale à 0.

$$\texttt{source} \in \texttt{visited} \cup \texttt{pqueue} \tag{16.2}$$

$$\texttt{distance}[\texttt{source}] = 0 \tag{16.3}$$

Le troisième invariant stipule que `distance` contient effectivement la longueur d'un chemin pour tout sommet déjà considéré.

$$\forall v \in \texttt{visited} \cup \texttt{pqueue}, \quad \texttt{source} \xrightarrow{\texttt{distance}[v]}{}^{\star} v \tag{16.4}$$

Pour les sommets dans `visited`, le quatrième invariant stipule plus précisément qu'il s'agit de la longueur d'un plus court chemin.

$$\forall v \in \texttt{visited}, \forall d, \ \text{ si } \texttt{source} \xrightarrow{d}{}^{\star} v \text{ alors } \texttt{distance}[v] \leq d \tag{16.5}$$

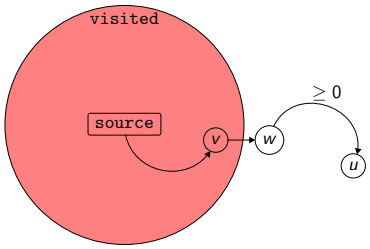
Le cinquième invariant stipule que, pour tout arc $v \to w$ déjà considéré, la distance à $w$ n'excède pas celle du chemin passant par $v$.

$$\begin{aligned} &\forall v \in \texttt{visited}, \forall w \text{ t.q. } v \xrightarrow{d} w, \\ &\quad w \in \texttt{visited} \cup \texttt{pqueue} \text{ et } \texttt{distance}[w] \leq \texttt{distance}[v] + d \end{aligned} \tag{16.6}$$

Enfin, le sixième invariant indique que tout sommet $v$ à une distance inférieure au plus petit élément de `pqueue` est nécessairement déjà dans `visited`.

$$\forall v, \ \text{ si } \texttt{source} \xrightarrow{d}{}^{\star} v \text{ et } d < \min(\texttt{pqueue}) \text{ alors } v \in \texttt{visited} \tag{16.7}$$

Montrer que ces six propriétés sont effectivement des invariants de boucle nécessite de montrer que d'une part elles sont établies initialement (*i.e.*, avant la boucle) et que d'autre part elles sont préservées par toute exécution du corps de la boucle. La première partie de cette preuve est simple, car, initialement, `visited` est vide et `pqueue` ne contient que le sommet `source`. La préservation des invariants est plus subtile. Les deux premiers invariants sont clairement préservés, car la source passe de `pqueue` à `visited` à la première itération, puis y reste. Par ailleurs, sa distance est nulle et donc ne peut être améliorée par la suite. L'invariant (16.4) est préservé car chaque mise à jour de `distance` correspond à la somme de la longueur d'un chemin jusqu'à `n.node`, pour lequel l'invariant est supposé, et du poids d'un arc sortant de ce sommet. Pour montrer la préservation de l'invariant (16.5), considérons un sommet $u$ et l'instant où `distance`$[u]$ est fixée, c'est-à-dire l'instant où $u$ sort de la file pour être ajouté à `visited`. Un chemin `source` $\to^{\star} u$ strictement plus court que `distance`$[u]$ sortirait de `visited` par un certain arc $v \to w$.




Mais alors on aurait `distance`$[w] <$ `distance`$[u]$ ce qui contredit le choix de $u$. La préservation de l'invariant (16.6) découle directement du fait que, lorsqu'un sommet est ajouté à `visited`, tous les arcs sortant de ce sommet sont examinés. Enfin, l'invariant (16.7) est préservé par un argument analogue à celui de la préservation de l'invariant (16.5) : un chemin plus court que min(`pqueue`) vers un sommet qui n'est pas dans `visited` devrait nécessairement sortir de `visited` par un arc dont l'extrémité est dans `pqueue`, en vertu de l'invariant (16.6), et contredirait donc la minimalité de min(`pqueue`). On note que le caractère positif ou nul du poids de chaque arc a été utilisé dans la preuve de préservation des invariants (16.5) et (16.7).

Il reste à déduire de ces invariants de boucle la correction de l'algorithme de Dijkstra. On sort de la boucle lorsque la file de priorité `pqueue` est vide. L'invariant (16.4) nous assure alors que `visited` ne contient que des sommets atteignables depuis la source. Inversement, tout sommet atteignable depuis la source est nécessairement dans `visited`. En effet, la source y appartient, en vertu de l'invariant (16.2), et un chemin de la source à un sommet $v$ en dehors de `visited` devrait donc sortir de `visited` par un certain arc. Mais cela contredirait alors l'invariant (16.6). L'ensemble `visited` contient donc exactement les sommets atteignables depuis la source et l'invariant (16.5) stipule que `distance` contient bien la longueur du plus court chemin pour chacun de ces sommets.

**Exercice 109.** Modifier la méthode `shortestPaths` pour conserver le chemin entre la source et chaque sommet atteint par le parcours. Une façon simple de procéder consiste à stocker, pour chaque sommet atteint, le sommet qui a permis de l'atteindre, par exemple dans une table de hachage. Le chemin est donc décrit « à l'envers », du sommet atteint vers la source. Attention : lorsqu'un chemin est amélioré, il faut mettre à jour cette table.
Solution □

## 16.3 Arbre couvrant minimal : algorithme de Kruskal

Dans cette section, on s'intéresse uniquement à des graphes non orientés, connexes et *sans boucles*, c'est-à-dire sans arc de la forme $a - a$.

**Arbre.** Un graphe non orienté connexe et acyclique est appelé un *arbre*. Tout arbre qui possède $N$ sommets est composé d'exactement $N - 1$ arcs.
*Démonstration :* Par récurrence forte sur $N$. C'est clair pour $N = 1$. Soit un graphe connexe acyclique de $N \geq 2$ sommets. Soit $v$ l'un de ses sommets. Il y a au moins un arc issu de $v$, car $G$ est connexe. Les $k \geq 1$ arcs issus de $v$ relient $v$ à autant de graphes $G_1, \ldots, G_k$ qui sont eux-mêmes connexes et acycliques. Par hypothèse de récurrence, chaque graphe $G_i$ possède $N_i$ sommets et $N_i - 1$ arcs, avec $N = 1 + N_1 + \cdots + N_k$. Par ailleurs, les graphes $G_i$ ne sont pas reliés entre eux, sans quoi il y aurait un cycle dans $G$. Le nombre d'arcs de $G$ est donc $k + (N_1 - 1) + \cdots + (N_k - 1) = N - 1$. □

**Arbre couvrant.** Étant donné un graphe $G$, un *arbre couvrant* de $G$ est un sous-ensemble $T$ d'arcs de $G$ tel que

1. $T$ est un arbre ;
2. chaque sommet de $G$ est l'extrémité d'au moins un arc de $T$ (on dit que $T$ *couvre* tous les sommets de $G$).

Voici par exemple un graphe de six sommets à gauche et l'un de ses arbres couvrants à droite.


Il y a bien sûr d'autres arbres couvrants de ce même graphe.

**Arbre couvrant minimal.** Soit un graphe $G$ dont les arcs sont étiquetés par des poids, tels que des entiers ou des flottants par exemple. Un *arbre couvrant minimal* de $G$ est un arbre couvrant de $G$ dont la somme des poids des arcs est minimale. Voici un exemple de graphe étiqueté par des poids, à gauche, et un arbre couvrant minimal à droite.


Le poids total est ici 12 et il n'y a pas d'arbre couvrant de poids inférieur.

**L'algorithme de Kruskal.** Étant donné un graphe $G$, l'algorithme de Kruskal construit un arbre couvrant minimal pour $G$. En supposant que les sommets de $G$ sont les entiers $0, 1, \ldots, V-1$, le fonctionnement de l'algorithme de Kruskal est le suivant :

1. soit $U$ une structure *union-find* pour les sommets $0, 1, \ldots, V-1$ de $G$ ;
2. soit $Q$ une file de priorité contenant tous les arcs de $G$, ordonnés par leur poids ;
3. soit $T$ une liste d'arcs, initialement vide ;
4. tant que $T$ contient moins que $V-1$ arcs :
   (a) retirer un arc $x-y$ de poids minimal de la file de priorité $Q$,
   (b) si $x$ et $y$ ne sont pas dans la même classe pour $U$, alors
      i. ajouter l'arc $x-y$ à $T$,
      ii. fusionner dans $U$ les classes de $x$ et $y$.

À la fin de l'algorithme, la liste $T$ contient un arbre couvrant minimal pour $G$. Illustrons le fonctionnement de cet algorithme sur l'exemple donné plus haut.

| arc $x-y$ | poids | action | $U$ |
|---|---|---|---|
| | | | $\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$ |
| $0-1$ | 1 | ajouté | $\{0,1\},\{2\},\{3\},\{4\},\{5\}$ |
| $0-2$ | 2 | ajouté | $\{0,1,2\},\{3\},\{4\},\{5\}$ |
| $4-5$ | 2 | ajouté | $\{0,1,2\},\{3\},\{4,5\}$ |
| $1-2$ | 3 | ignoré | — |
| $1-3$ | 3 | ajouté | $\{0,1,2,3\},\{4,5\}$ |
| $2-3$ | 4 | ignoré | — |
| $3-4$ | 4 | ajouté | $\{0,1,2,3,4,5\}$ |

On s'arrête là car $T$ contient alors 5 arcs. En particulier, les arcs $3-5$ (de poids 5) et $2-4$ (de poids 6) ne sont pas retirés de la file. Ce n'est pas la seule exécution possible, car il y a plusieurs arcs de même poids. Ainsi, si on considère l'arc $3-4$ avant l'arc $2-3$, on fait une étape de moins. Mais dans tous les cas, on obtiendra bien un arbre couvrant de poids total 12.

Le programme 42 page 199 contient un code Java de l'algorithme de Kruskal. Ce programme utilise les classes `PriorityQueue` et `LinkedList` de la bibliothèque standard de Java pour réaliser $Q$ et $T$, et la classe `UnionFind` du programme 20 page 121 pour réaliser $U$. Les éléments de la file de priorité sont des arcs, comparés par leur poids. On se donne pour cela une classe `Kedge`. Elle est également utilisée pour le résultat renvoyé par la méthode `kruskal`. Le graphe est matérialisé par une classe `Kgraph` qui suppose que les sommets sont les entiers $0, 1, \ldots, V-1$ et qui fournit la liste de tous les arcs du graphe avec une méthode `allEdges`. Le code ne pose pas de difficulté particulière ; il suit fidèlement la description de l'algorithme. Si on trouve contraignant que les sommets soient les $0, 1, \ldots, V-1$, on peut utiliser une structure *union-find* plus générale, telle que celle proposée dans l'exercice 61 page 122.

**Complexité.** Soit $V$ le nombre de sommets et $E$ le nombre d'arcs de $G$. Le coût en espace est clairement $O(E)$, car la file contient initialement tous les arcs de $G$, c'est-à-dire $O(V^2)$. Pour ce qui est du temps, dans le pire des cas, tous les arcs de $G$ sont insérés et retirés de la file de priorité, soit $E$ tours de boucle. Chaque retrait de la file coûte $\log(E)$ et chaque opération *union-find* peut être considérée de temps constant amorti (voir chapitre 9). Le coût total est donc $O(E \log(E))$, c'est-à-dire $O(V^2 \log(V))$ car $\log(E) = O(\log(V^2)) = O(\log(V))$.

**Correction.** Commençons par justifier que l'algorithme de Kruskal termine toujours et que l'étape 4a de l'algorithme n'échoue jamais sur une file vide. À chaque tour de boucle, la file $Q$ contient un arc de moins. On finira donc forcément par sortir de la boucle ou échouer sur l'étape 4a parce que $Q$ est vide. Montrons que ce second cas ne peut arriver. Par construction, l'ensemble d'arcs $T$ ne contient jamais de cycle. C'est donc à chaque instant un ensemble de sous-graphes connexes acycliques disjoints les uns des autres. Si on parvenait en 4a avec une file vide, alors cela veut dire que tous les arcs ont été considérés. Mais le graphe de départ étant supposé connexe, l'ensemble $T$ est forcément connexe, donc réduit à un seul graphe acyclique. C'est donc un arbre couvrant de $G$. Mais nous avons montré plus haut que dans ce cas $|T| = V-1$, ce qui contredit le fait d'être encore dans la boucle.

Montrons maintenant que l'algorithme de Kruskal renvoie bien un arbre couvrant minimal. Commençons par montrer que le résultat est bien un arbre couvrant. On vient de voir qu'on termine nécessairement avec $|T| = V-1$. Si $T$ contenait plusieurs composantes, alors chacune de ces composantes contiendrait $C-1$ arcs dès lors qu'elle contient $C$ sommets (car il s'agit d'un arbre) et donc $T$ ne pourrait contenir $V-1$ arcs au total. L'ensemble $T$ est donc bien un arbre. C'est en particulier un arbre couvrant de ses propres sommets, ce qui signifie qu'ils sont au nombre de $V$. L'ensemble $T$ est donc bien un arbre couvrant de $G$.

Reste à montrer qu'il est minimal. Pour cela, on montre l'invariant suivant pour la boucle « tant que » : l'ensemble $T$ est un sous-ensemble d'un arbre couvrant minimal. C'est vrai initialement, car $T = \emptyset$. Supposons l'hypothèse vérifiée à une étape arbitraire

**Programme 42 — Algorithme de Kruskal (arbre couvrant minimal)**

```
class Kedge implements Comparable<Kedge> {
  final int src, dst;
  final double weight;
  Kedge(int src, int dst, double weight) {
    this.src = src;
    this.dst = dst;
    this.weight = weight;
  }
  public int compareTo(Kedge that) {
    return this.weight < that.weight ? - 1 :
           this.weight > that.weight ? + 1 : 0;
  }
}

class Kgraph extends Graph<Integer> {
  final int V; // les sommets sont 0,1,...,V-1
  List<Kedge> allEdges() { // renvoie tous les arcs du graphe
    ...
  }
}

class Kruskal {
  static List<Kedge> kruskal(Kgraph g) {
    List<Kedge> mst = new LinkedList<>();
    UnionFind uf = new UnionFind(g.V);
    PriorityQueue<Kedge> q = new PriorityQueue<>();
    q.addAll(g.allEdges());
    while (mst.size() < g.V - 1) {
      Kedge e = q.remove();
      if (uf.sameClass(e.src, e.dst)) continue;
      uf.union(e.src, e.dst);
      mst.add(e);
    }
    return mst;
}
```

de l'algorithme, c'est-à-dire que $T$ est contenu dans un arbre couvrant minimal $M$, et considérons l'arc $x - y$ suivant à sortir de la file. On distingue plusieurs cas :

— Si l'arc n'est pas ajouté à $T$, l'hypothèse reste trivialement vraie.
— Si l'arc est ajouté à $T$ et faisait déjà partie de $M$, alors l'hypothèse est préservée (avec le même $M$).
— Si enfin l'arc est ajouté à $T$ mais ne faisait par partie de $M$, alors $M \cup \{x - y\}$ contient un cycle passant par l'arc $x - y$. Il existe donc un arc $a$ qui relie la composante (actuelle) de $x$ à celle de $y$. Cet arc n'a pas encore été considéré (sinon, les composantes de $x$ et $y$ seraient déjà réunies) et donc son poids est supérieur ou égal à celui de l'arc $x - y$. Dès lors, l'ensemble $M' = M \backslash \{a\} \cup \{x - y\}$ est un arbre couvrant de poids inférieur ou égal à $M$ et contenant le nouvel arc.

À l'issue de l'algorithme, $T$ est donc contenu dans un arbre couvrant minimal. Comme il a déjà été prouvé que c'est un arbre couvrant, c'est donc un arbre couvrant minimal.

**Exercice 110.** Écrire une méthode

```
static boolean isSpanningTree(Kgraph g, List<Kedge> t)
```

qui détermine si la liste d'arcs `t` constitue un arbre couvrant de `g`. On suppose que le graphe `g` est connexe. Indication : pour tester l'absence de cycle, on pourra utiliser une structure *union-find*. Donner la complexité en temps de cette méthode, en fonction du nombre de sommets de `g`. Solution □

**Notes bibliographiques.** L'algorithme de Dijkstra est dû à l'informaticien néerlandais Edsger W. Dijkstra et date de 1959 [6]. L'algorithme de Kruskal est dû à l'informaticien américain Joseph Kruskal et date de 1956 [13].

# Annexes

# A Solutions des exercices

## Exercice 1, page 22

Lors de l'appel à la méthode `f`, la valeur de la variable locale `x` est *copiée* dans une nouvelle variable, à savoir l'argument formel de `f`. Quoi que fasse la méthode `f` avec cette nouvelle variable, cela ne peut affecter la variable `x`.

## Exercice 2, page 30

Notons $(F_n)$ la suite de Fibonacci calculée par la méthode `fib`. Il est facile de voir que le calcul de `fib(`$n$`)` nécessite exactement $F_{n+1} - 1$ additions. Or $F_n \sim \phi^n/\sqrt{5}$, où $\phi = (1 + \sqrt{5})/2$ est le nombre d'or, et on en déduit donc que la méthode `fib` a une complexité exponentielle.

## Exercice 3, page 35

Il faut évidemment éviter de recalculer la somme à chaque fois, ce qui serait quadratique. Une solution consiste à maintenir la somme dans une variable `s` :

```
static int[] cumulSums(int[] a) {
  int[] c = new int[a.length];
  int s = 0;
  for (int i = 0; i < a.length; i++) {
    s += a[i];
    c[i] = s;
  }
  return c;
}
```

Bien entendu, on peut se passer de cette variable en utilisant la valeur précédente, puisque `c[i]` = `c[i-1]` + `a[i]`. Mais il faut alors faire attention au cas `i` = 0 et éventuellement aussi au cas d'un tableau `a` de taille nulle, deux problèmes évités par la solution ci-dessus.

## Exercice 4, page 35

Il n'a pas de difficulté ici, si ce n'est qu'il faut faire attention au cas `n` $\leq 1$.

```
static int[] fib(int n) {
  int[] f = new int[n];
  if (n > 1) f[1] = 1;
  for (int i = 2; i < n; i++)
    f[i] = f[i - 2] + f[i - 1];
  return f;
}
```

## Exercice 5, page 35

On suit l'algorithme proposé, l'échange se faisant en utilisant une variable auxiliaire `tmp`. Attention à bien écrire `i+1` pour tirer un entier compris entre `0` et `i` *inclus*.

```
static void shuffle(int[] a) {
  for (int i = 1; i < a.length; i++) {
    int j = (int)(Math.random() * (i + 1));
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}
```

## Exercice 6, page 36

Dès que la valeur est trouvée, on renvoie l'indice correspondant. On choisit donc ici de renvoyer le plus petit indice où la valeur apparaît (car c'est le plus simple). Mais on pouvait tout aussi bien parcourir le tableau dans l'autre sens et donc renvoyer l'indice le plus grand.

```
static int indexOf(int[] a, int v) {
  for (int i = 0; i < a.length; i++)
    if (a[i] == v)
      return i;
  throw new NoSuchElementException();
}
```

Dans le cas où `v` n'apparaît pas dans le tableau, on a choisi ici de lever une exception, à savoir l'exception `NoSuchElementException` de la bibliothèque Java. C'est là une solution propre et idiomatique. On aurait pu aussi renvoyer une valeur non significative, telle que -1 par exemple, mais c'est plus dangereux car cette valeur pourrait être utilisée à tort comme un indice de tableau dans le code qui appelle `indexOf`.

## Exercice 7, page 36

Un tableau de taille nulle constitue l'unique difficulté de cette exercice. On pourrait lever une exception dans ce cas, ou encore supposer que le tableau contient au moins une valeur. On peut alors initialiser la variable `m` avec `a[0]`. Une autre solution consiste à l'initialiser avec le plus petit entier, `Integer.MIN_VALUE` (à savoir $-2^{31}$). Il n'y a alors pas de cas particulier à traiter.

```
static int max(int[] a) {
  int m = Integer.MIN_VALUE;
  for (int v: a)
    if (v > m)
      m = v;
  return m;
}
```

## Exercice 8, page 38

La quantité `hi` − `lo` décroît strictement à chaque tour de boucle. Si la valeur `v` n'apparaît pas dans le tableau, cette quantité finira donc par être négative ou nulle, mettant ainsi fin à la boucle `while`. Remarque : Dans le calcul de complexité qu'on a fait plus haut, on a de fait majoré le nombre de tours de boucle par $\log n$. Mais l'argument concernant le signe de `hi` − `lo` est une façon plus simple de justifier la terminaison.

## Exercice 9, page 38

Pour un tableau de plus de $2^{30}$ éléments, le calcul de l'index `m` sous la forme `(lo+hi)/2` peut provoquer un débordement de la capacité du type `int`. Si par exemple `lo` $= 2^{30}$ et `hi` $= 2^{30} + 1$, alors `lo+hi` dépasse la plus grande valeur du type `int`, à savoir $2^{31} - 1$. On obtient alors une valeur négative pour `lo+hi`, donc pour `m`, ce qui provoque ensuite un accès en dehors des bornes du tableau [1]. En revanche, le calcul

```
int m = lo + (hi - lo) / 2;
```

est correct. En effet, la soustraction `hi-lo` se fait sans débordement arithmétique car on est ici sous l'hypothèse `lo` < `hi`. Puis l'addition se fait également sans débordement arithmétique car le résultat est plus petit que `hi`. Une autre solution consiste à écrire

```
int m = (lo + hi) >>> 1;
```

Il s'agit ici d'un décalage logique à droite (*i.e.*, sans réplication du bit de signe). S'il y a débordement arithmétique, ce ne peut être que d'un seul bit, le bit de signe en l'occurrence, et ce décalage nous redonne bien la bonne valeur.

1. C'est un bug célèbre de la bibliothèque de Java [3].

## Exercice 10, page 41

Il suffit d'ajouter un `else` au test `if (len > n)` pour tester si la condition est réalisée et, le cas échéant, prendre le préfixe de `this.data` qui contient `n / 2` éléments.

```
else if (4 * len < n)
  this.data = Arrays.copyOfRange(this.data, 0, n / 2);
```

## Exercice 11, page 41

On se sert ici de la fonction de bibliothèque `copyOfRange`, comme dans l'exercice précédent.

```
int[] toArray() {
  return Arrays.copyOfRange(this.data, 0, this.length);
}
```

Bien entendu, on peut faire la même chose de façon élémentaire, en allouant un nouveau tableau et en y copiant les éléments un par un, mais `Arrays.copyOfRange` le fait plus efficacement.

## Exercice 12, page 43

Le code ne pose pas de difficulté et correspond exactement à ce qui vient d'être fait dans la boucle qui lit les lignes du fichier :

```
void append(int v) {
  int n = this.length;
  this.setSize(n + 1);
  this.data[n] = v;
}
```

On simplifie alors le code de lecture du fichier en remplaçant les trois dernières lignes par

```
r.append(Integer.parseInt(s));
```

Ce qu'il important de comprendre ici, c'est qu'une suite répétée de $n$ opérations `append` aura une complexité totale $O(n)$, comme l'explique le paragraphe qui suit cet exercice. En particulier, la lecture du fichier se fait donc en un temps proportionnel à la taille du fichier, même si certains des appels à `append` conduisent à une recopie complète du tableau.

## Exercice 13, page 44

Il suffit d'ajouter les éléments un par un dans le `StringBuffer`. La seule difficulté concerne les virgules à insérer entre les éléments. On choisit ici de traiter le cas du premier élément de façon particulière, pour ne plus avoir à tester ensuite dans la boucle si une virgule doit ou non être ajoutée.

```
public String toString() {
  StringBuffer b = new StringBuffer("[");
  if (0 < this.length)
    b.append(this.data[0]);
  for (int i = 1; i < this.length; i++)
    b.append(", ").append(this.data[i]);
  return b.append("]").toString();
}
```

On notera que la méthode `append` le `StringBuffer`, ce qu'on utilise ici pour chaîner plusieurs opérations. Comme on vient juste de l'expliquer, la complexité de cette méthode `toString` est linéaire. Une solution utilisant des concaténations de chaînes de caractères (avec +) aurait été quadratique.

## Exercice 14, page 45

Il n'y a pas de difficulté ici : on retire les deux éléments au sommet de la pile avec `pop` puis on les réinsère dans l'ordre inverse. On pourrait laisser `pop` échouer si la pile contient moins de deux éléments, mais on choisit ici de lever une autre exception, plus explicite.

```
static void swap(Stack s) {
  if (s.size() <= 1) throw new IllegalArgumentException();
  int x = s.pop();
  int y = s.pop();
  s.push(x);
  s.push(y);
}
```

Le code ci-dessus est correct, mais il modifie quatre fois la taille du tableau redimensionnable. C'est inutilement coûteux, car la taille ne change pas au final. C'est pourquoi il est préférable d'écrire cette méthode `swap` dans la classe `Stack`. On peut le faire ainsi :

```
void swap() {
  int n = this.elts.size();
  if (n <= 1) throw new IllegalArgumentException();
  int tmp = this.elts.get(n - 1);
  this.elts.set(n - 1, this.elts.get(n - 2));
  this.elts.set(n - 2, tmp);
}
```

## Exercice 15, page 51

C'est un parcours analogue à celui de la méthode `contains`. Pour changer, on choisit ici de l'écrire ici avec une boucle `for`.

```
static int length(Singly s) {
  int len = 0;
  for ( ; s != null; s = s.next)
```

```
      len++;
    return len;
  }
```

Comme pour `contains`, on se sert de la variable locale `s` pour le parcours.

## Exercice 16, page 51

On parcourt la liste tout en décrémentant le compteur `i`. Si ce dernier vaut `0`, on renvoie l'élément de la liste qu'on a atteint. Si en revanche on atteint la fin de la liste, alors on lève l'exception pour signaler que l'indice ne correspondait pas à une position valide dans la liste.

```
static int get(Singly s, int i) {
  for (; s != null; s = s.next)
    if (i-- == 0) return s.element;
  throw new IllegalArgumentException();
}
```

On notera que `i` est décrémenté *après* avoir été comparé à 0. Ce code est correct même lorsque la valeur initiale de `i` est strictement négative, car elle reste alors strictement négative pendant tout le parcours. Bien entendu, il s'agit alors d'un parcours inutile, qu'on peut éviter en testant dès le départ si `i` est strictement négatif. Le code ci-dessus privilégie la concision à l'efficacité.

## Exercice 17, page 52

La classe `StringBuffer` est réalisée par un tableau redimensionnable. On a donc une complexité totale linéaire pour la méthode `listToString`, même si certaines des opérations `sb.append` réalisées coûtent plus cher que d'autres.

## Exercice 18, page 53

Notons $x_1, \ldots, x_n$ les $n$ éléments de la liste. Montrons par récurrence

$(H_i)$ après l'examen de $x_1, \ldots, x_i$, on a `candidate` $= x_j$ avec probabilité $\frac{1}{i}$

L'initialisation $(H_1)$ est claire. Supposons $(H_i)$.

— $x_{i+1}$ est sélectionné avec probabilité $\frac{1}{i+1}$ ;

— pour $1 \leq j \leq i$, $x_j$ est sélectionné avec probabilité $\frac{1}{i} \times \frac{i}{i+1} = \frac{1}{i+1}$.

## Exercice 19, page 54

Le champ `size` est initialisé à 0 dans le constructeur. Il est mis à jour dans la méthode `push` (où il est incrémenté) et dans la méthode `pop` (où il est décrémenté). Il est important qu'il s'agisse d'un champ privé, afin de maintenir l'invariant que sa valeur est égale au nombre d'éléments de la liste. Sans le caractère privé, sa valeur pourrait être modifiée par le code client de la classe `Stack` (de même que sans le caractère privé du champ `head`, la liste pourrait être modifiée directement et l'invariant ne serait plus forcément maintenu).

## Exercice 20, page 54

C'est effectivement très semblable à la méthode `listToString` de la page 51, à ceci près qu'on doit ici introduire une variable locale `s` pour réaliser le parcours. Il serait en effet incorrect de se servir directement de `head` pour réaliser le parcours, car ceci modifierait le contenu de la pile.

```
public String toString() {
  StringBuffer sb = new StringBuffer("[");
  for (Singly s = this.head; s != null; s = s.next) {
    sb.append(s.element);
    if (s.next != null) sb.append(", ");
  }
  return sb.append("]").toString();
}
```

## Exercice 21, page 57

Voir l'exercice 19 page 54 et sa solution.

## Exercice 22, page 57

On encapsule deux listes `entry` et `exit` dans la classe `Queue`.

```
class Queue {
  private Stack entry, exit;
  Queue() {
    this.entry = new Stack();
    this.exit = new Stack();
  }
```

Les opérations `isEmpty` et `push` sont immédiates.

```
  int size() { return this.entry.size() + this.exit.size(); }
  boolean isEmpty() { return this.size() == 0; }
  void push(int x) { this.entry.push(x); }
```

Les opérations `peek` et `pop` sont plus délicates, car il faut éventuellement déverser les éléments de la pile `entry` dans la pile `exit`, lorsque celle-ci est vide. On se donne une méthode pour cela.

```
  private void ensuresOutput() {
    if (this.isEmpty())
      throw new NoSuchElementException();
    if (this.exit.isEmpty())
      while (!this.entry.isEmpty())
        this.exit.push(this.entry.pop());
  }
```

Les méthodes `peek` et `pop` s'en déduisent alors aisément.

```
int peek() { ensuresOutput(); return this.exit.top(); }
int pop()  { ensuresOutput(); return this.exit.pop(); }
```

L'efficacité de cette solution dépend de l'efficacité de la structure de pile sous-jacente. Il est raisonnable de supposer que les opérations `push`, `top` et `pop` sur les piles sont toutes en temps constant (structure de liste) ou en temps constant amorti (structure de tableau redimensionnable).

Dès lors, la méthode `push` de la file se fait également en temps constant (éventuellement amorti), puisqu'elle se contente d'une opération `push` sur la pile `entry`. Pour les méthodes `peek` et `pop` de la file, c'est plus complexe, car la méthode `ensuresOutput` peut avoir un coût arbitrairement grand. Si par exemple on ajoute $N$ éléments dans une file initialement vide, puis que l'on retire un unique élément, alors le coût de ce retrait sera $O(N)$, car `ensuresOutput` va déplacer les $N$ éléments de la pile `entry` vers la pile `exit`. Cependant, le coût de `ensuresOutput` s'*amortie* sur l'ensemble des $N$ opérations. En effet, chaque élément ne peut être déplacé qu'une seule fois de la pile `entry` vers la pile `exit`. Dès lors, les opérations `peek` et `pop` ont un coût amorti $O(1)$. C'est donc là une méthode efficace pour représenter une file.

Par rapport à la solution utilisant une liste chaînée (section 4.3), cette solution est plus économe en espace (elle utilise environ deux fois moins d'espace).

## Exercice 23, page 62

On procède comme pour `insertAfter`, en prenant soin de tester s'il existe un élément à gauche de `this`.

```
void insertBefore(int v) {
  Doubly e = new Doubly(v);
  e.next = this;
  if (this.prev != null) {
    e.prev = this.prev;
    e.prev.next = e;
  }
  this.prev = e;
}
```

## Exercice 24, page 65

On commence par écrire une méthode pour construire une liste simplement chaînée cyclique contenant les entiers 1, 2, ..., $n$. On choisit de renvoyer l'élément contenant $n$, ici contenu dans la variable `last`.

```
static Singly circle(int n) {
  Singly last = new Singly(n, null);
  Singly first = last;
  for (int i = n - 1; i >= 1; i--)
    first = new Singly(i, first);
  last.next = first; // on ferme la boucle
  return last;
}
```

Puis on adapte le code de la méthode `josephus`. La variable `c` contient toujours l'élément courant de la liste. Comme suggéré dans l'énoncé, une seconde variable `pred` désigne l'élément qui précède `c`.

```
static int josephus(int n, int p) {
  Singly pred = circle(n);
  Singly c = pred.next;
  while (c != pred) { // tant qu'il reste au moins deux joueurs
    for (int i = 1; i < p; i++) {
      pred = c; c = c.next; // on avance p-1 fois
    }
    pred.next = c.next; c = c.next; // on élimine le p-ième
  }
  return c.element;
}
```

## Exercice 25, page 66

L'idée est très simple : on reprend la solution de l'exercice précédent, c'est-à-dire la solution utilisant une liste circulaire simplement chaînée, et on la réalise en utilisant un tableau `next` qui lie chaque élément à l'élément suivant dans la liste. On commence donc par allouer ce tableau et par le remplir. Attention : dans le problème tel qu'il est posé, les joueurs sont numérotés de 1 à $n$ mais ici on utilise les indices de tableau 0 à $n-1$. Il faudra donc penser à ajouter 1 à la valeur renvoyée au final.

```
static int josephus(int n, int p) {
  int[] next = new int[n];
  for (int i = 0; i < n; i++)
    next[i] = (i + 1) % n;
```

On reprend alors le code de l'exercice précédent, en remplaçant systématiquement $e$`.next` par `next[`$e$`]`.

```
  int pred = n - 1, c = 0;
  while (c != pred) { // tant qu'il reste au moins deux joueurs
    for (int i = 1; i < p; i++) {
      pred = c; c = next[c]; // on avance p-1 fois
    }
    next[pred] = next[c]; c = next[c];
  }
  return c+1; // on renvoie le joueur, pas l'indice
}
```

## Exercice 26, page 69

On se sert de `Bucket.contains` pour tester si l'élément est déjà contenu dans la table. Le cas échéant, on termine tout de suite, sans rien faire.

```
void add(String s) {
  int i = hash(s);
  if (Bucket.contains(this.buckets[i], s)) return;
  this.buckets[i] = new Bucket(s, this.buckets[i]);
}
```

On note que la valeur de hachage n'est calculée qu'une seule fois. Si on s'était servi plutôt de la méthode `contains` de `HashTable`, elle aurait été calculée deux fois.

## Exercice 27, page 71

Si on suppose avoir réalisé l'exercice précédent, il suffit d'incrémenter le champ `size` lorsque l'élément est effectivement ajouté à la table.

```
void add(String s) {
  int i = hash(s);
  if (Bucket.contains(this.buckets[i], s)) return;
  this.buckets[i] = new Bucket(s, this.buckets[i]);
  this.size++;
}
```

Il est important que `size` soit un champ privé, afin de maintenir l'invariant que sa valeur est égale au nombre d'éléments de la table. Sans le caractère privé, sa valeur pourrait être modifiée par le code client de la classe `HashTable`. C'est pourquoi on doit fournir une méthode `size` pour rendre accessible la valeur de `size`.

```
int size() {
  return this.size;
}
```

## Exercice 28, page 71

Il y a une petite difficulté ici, car les paquets peuvent contenir des doublons. Dès lors, supprimer un élément nécessite de supprimer *toutes* ses occurrences dans le paquet concerné. Une autre solution consiste à modifier la méthode `add` suivant l'exercice 26 pour qu'elle ne crée pas de doublon, *i.e.*, qu'elle ne fasse rien lorsque l'élément est déjà contenu dans la table. On peut alors écrire une méthode statique `remove` dans la classe `Bucket` qui supprime la *première* occurrence de `s` trouvée dans la liste `b`.

```
static Bucket remove(Bucket b, String s) {
  if (b == null) return null;
  if (b.element.equals(s)) return b.next;
  b.next = remove(b.next, s);
  return b;
}
```

Cette méthode renvoie la tête de la liste modifiée, car elle est différente lorsque c'est le premier élément de la liste qui est supprimé. Il suffit alors d'utiliser cette méthode pour écrire la méthode `remove` de la classe `HashTable`. Si on suppose qu'on a ajouté un champ `size` (voir l'exercice précédent), alors il faut prendre soin de ne pas le décrémenter lorsque l'élément n'est pas contenu dans la table.

```
void remove(String s) {
  int i = hash(s);
  if (!Bucket.contains(this.buckets[i], s)) return;
  this.buckets[i] = Bucket.remove(this.buckets[i], s);
  this.size--;
}
```

## Exercice 29, page 72

Si `this.buckets.length` vaut $2^k$, alors `this.buckets.length-1` s'écrit en binaire $00\cdots011\cdots1$ avec exactement $k$ chiffres 1 de poids faible. Du coup, le calcul de `h` modulo $2^k$ peut s'écrire directement avec un ET logique :

```
return h & (this.buckets.length-1);
```

On a gagné sur deux tableaux : d'une part, on masque l'éventuel bit de signe de `h` (ce que l'on faisait avant avec `& 0x7fffffff`) ; d'autre part, on remplace une opération coûteuse (`%`) par une opération très efficace (`&`). La bibliothèque standard de Java procède ainsi.

## Exercice 30, page 75

Soit $N$ le nombre de nœuds et $h$ la hauteur. Montrons par récurrence sur $N$ qu'on a l'inégalité

$$2^{h-1} - 1 < N \leq 2^h - 1.$$

C'est vrai pour un arbre vide, où $N = h = 0$. Si $N = 2M + 1$, alors l'arbre a deux sous-arbres de taille $M$ et de hauteur $h-1$, avec $2^{h-2} - 1 < M \leq 2^{h-1} - 1$ par hypothèse de récurrence. (Les deux sous-arbres ont forcément la même hauteur, car $M$ ne peut être encadré par deux puissances de deux consécutives que d'une seule façon.) Du coup, $2^{h-1} - 1 < 2M + 1 \leq 2^h - 1$. Si en revanche $N = 2M$, alors l'arbre a un sous-arbre de taille $M$ et de hauteur $h-1$ et un sous-arbre de taille $M-1$ et de hauteur $h' \leq h-1$. Par hypothèse de récurrence sur le sous-arbre de taille $M$, on a $2^{h-2} - 1 < M \leq 2^{h-1} - 1$. Or $N \geq 2$ implique $h \geq 2$. Dès lors, $2^{h-2}$ est entier et donc $2^{h-2} \leq M$. On en déduit $2^{h-1} - 1 < 2M < 2^h - 1$.

## Exercice 31, page 77

Il est très facile d'écrire une telle méthode récursivement. On a choisi ici de numéroter les nœuds avec les entiers de 1 à `n`, de bas en haut.

```
static Tree leftDeepTree(int n) {
  if (n == 0) return null;
  return new Tree(leftDeepTree(n - 1), n, null);
}
```

## Exercice 32, page 77

Si on tente cette expérience avec le code de `leftDeepTree` proposé ci-dessus, c'est en fait cette méthode-là qui va provoquer le débordement de pile. On commence donc par réécrire la méthode `leftDeepTree` sans utiliser de récursivité.

```
static Tree leftDeepTree(int n) {
  Tree t = null;
  while (n-- > 0)
    t = new Tree(t, n, null);
  return t;
}
```

On peut alors se livrer à l'expérience proposée, on procédant par dichotomie. La valeur obtenue dépend de la taille de la pile, qui varie selon la version de la machine virtuelle, l'architecture et le système. Pour une taille de pile de 1 Mo, on va trouver une valeur de l'ordre de 15 000. Comme expliqué page 18, on peut augmenter la taille de pile avec l'option `-Xss` de la machine virtuelle Java.

## Exercice 33, page 77

C'est une question assez difficile. Pour une liste, ce serait facile, car il s'agirait d'un simple parcours linéaire. Mais ici, le parcours n'est pas linéaire : il faut calculer la taille du sous-arbre gauche et celle du sous-arbre droit. Une solution consiste à stocker les sous-arbres pour lesquels on doit encore calculer la taille dans une structure de données. Le plus simple est d'utiliser une pile. Initialement, on y met l'arbre `t` dont on souhaite calculer la taille.

```
static int sizeNonRec(Tree t) {
  Stack<Tree> s = new Stack<Tree>();
  s.push(t);
  int size = 0;
```

Puis le programme consiste à retirer les éléments de la pile, un par un, pour en calculer la taille. Si l'élément retiré est `null`, il n'y a rien à faire. On utiliser l'instruction `continue` pour passer directement à l'itération suivante de la boucle.

```
  while (!s.isEmpty()) {
    Tree n = s.pop();
    if (n == null) continue;
```

Sinon, on vient de découvrir un nouveau nœud et on incrémente donc la variable `size`. Puis on ajoute les sous-arbres gauche et droit dans la pile.

```
    size++;
    s.push(n.left);
    s.push(n.right);
  }
```

Une fois la pile vide, on renvoie la valeur de `size`.

```
    return size;
  }
```

Si l'arbre contient $N$ nœuds, on aura mis au total $N+1$ arbres vides dans la pile. On peut éviter ce surcoût en garantissant que la pile ne contient jamais d'arbre vide. On peut alors supprimer le test `n == null` mais il faut en revanche tester si `n.left` et `n.right` ne sont pas vides avant de les ajouter à la pile. Et, bien entendu, il faut penser à tester initialement si `t` est vide ou non.

## Exercice 34, page 77

On choisit une écriture récursive. On utilise la fonction de bibliothèque `Math.max`. On rappelle que l'arbre vide (`null`) a pour hauteur 0.

```
  static int height(Tree t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
  }
```

## Exercice 35, page 77

On se sert d'une fileq qui va contenir tous les sous-arbres non vides d'un même niveau. Initialement, on y met l'arbre `t` tout entier, s'il est non vide. Une variable `height` va servir à compter les niveaux.

```
  static int heightNonRec(Tree t) {
    Queue<Tree> q = new LinkedList<Tree>();
    if (t != null) q.add(t);
    int height = 0;
```

Tant que la file est non vide, c'est que l'on vient de découvrir un nouveau niveau et on incrémente `height`.

```
    while (!q.isEmpty()) {
      height++;
```

Pour passer au niveau suivant, il faut retirer tous les éléments de la file et y ajouter leurs sous-arbres. On pourrait utiliser une file temporaire pour cela, mais il est encore plus simple d'utiliser une boucle `for` car `q.size` nous donne le nombre d'éléments de la file. On prend soin de ne pas ajouter d'arbre vide dans la file.

```
      for (int n = q.size(); n > 0; n--) {
        Tree x = q.remove();
        if (x.left  != null) q.add(x.left);
        if (x.right != null) q.add(x.right);
      }
    }
```

Une fois sorti de la boucle, on renvoie la valeur de `height`.

```
    return height;
  }
```

## Exercice 36, page 79

Il convient de bien traiter les deux cas d'arrêt correspondant respectivement à un échec et à un succès.

```
static boolean contains(BST b, int x) {
  if (b == null) return false;
  if (x == b.value) return true;
  return contains(x < b.value ? b.left : b.right, x);
}
```

## Exercice 37, page 79

On va procéder par une descente dans l'arbre, analogue à celle effectuée par `contains`, tout en maintenant dans une variable `c` le plus grand élément de `b` inférieur à `x` qu'on a vu jusqu'à présent. Pour distinguer le cas où un tel élément n'a pas encore été rencontré, on donne le type `Integer` à `c`, ce qui nous permet d'utiliser la valeur `null` comme initialisation.

```
static int floor(BST b, int x) {
  Integer c = null;
```

La descente proprement dite est semblable à celle de `contains`, si ce n'est qu'il faut mettre `c` à jour lorsqu'on se déplace vers la droite.

```
  while (b != null) { // invariant : c <= x
    if (x == b.value) return x;
    if (x < b.value) b = b.left;
    else { c = b.value; b = b.right; }
  }
```

Une fois sorti de la boucle, on teste si `c` est toujours `null`. Le cas échéant, cela signifie que `x` est plus petit que tous les éléments de `b` et on lève alors une exception. Sinon, on renvoie `c`.

```
  if (c == null) throw new NoSuchElementException();
  return c;
}
```

On notera que le compilateur Java a inséré automatiquement deux conversions entre les types `int` et `Integer`, dans un sens pour affecter à `c` l'entier `b.value` et dans l'autre sens pour permettre `return c` à la fin du programme.

## Exercice 38, page 79

Le code actuel de `add` ne fait rien lorsque l'élément se trouve déjà dans l'arbre. Pour l'y ajouter systématiquement, il suffit de remplacer le test

```
  if (x < b.value)
```

par le test

```
if (x <= b.value)
```

Ainsi, en cas d'égalité, on poursuit l'insertion vers la gauche. On finira donc par tomber sur un arbre vide et donc par insérer l'élément.

## Exercice 39, page 79

Il n'est pas difficile de descendre dans l'arbre avec une boucle `while`. C'est ce qu'on a déjà fait pour la méthode `contains`. Ce qui est difficile ici, c'est de réaliser la modification de l'arbre une fois l'emplacement d'insertion trouvé. En effet, si on attend d'arriver sur la valeur `null`, alors il sera trop tard pour effectuer la modification, à moins de conserver en permanence le père du nœud considéré dans une autre variable. Une telle solution serait tout à fait possible, mais il est en fait plus simple encore de tester si un sous-arbre est `null` avant d'y descendre. On commence par traiter le cas particulier d'un arbre vide, puis on s'engage dans une boucle où la variable `t` désigne le sous-arbre en cours d'inspection.

```
static BST add(BST b, int x) {
  if (b == null) return new BST(null, x, null);
  BST t = b;
  while (true) { // invariant t != null
```

Notre invariant est que `t` reste toujours non `null`. Si on trouve la valeur `x`, il n'y a rien à faire. Sinon, on compare `x` à `t.value` pour savoir s'il faut descendre à gauche ou à droite. C'est là qu'on teste si le sous-arbre correspondant est vide. Le cas échéant, on effectue la modification et on termine avec `return b`.

```
    if (x == t.value) return b;
    if (x < t.value)
      if (t.left != null) t = t.left;
      else { t.left = new BST(null, x, null); return b; }
    else
      if (t.right != null) t = t.right;
      else { t.right = new BST(null, x, null); return b; }
  }
}
```

On a écrit ici une « boucle infinie » avec `while true` mais on finira toujours par en sortir avec `return b`.

## Exercice 40, page 88

Le plus simple consiste à utiliser `contains` pour déterminer si l'élément va être effectivement ajouté par `add` ou retiré par `remove`. Pour `add`, par exemple, on peut écrire

```
void add(Integer x) {
  if (!AVL.contains(this.root, x)) this.size++;
  this.root = AVL.add(this.root, x);
}
```

Cela peut paraître coûteux, mais les deux parcours sont tous les deux de même complexité $O(\log n)$ et on conserve donc des opérations logarithmiques au final. Une autre solution consisterait à maintenir un champ `size` dans la classe `AVL` plutôt que dans la classe `AVLSet`. Mais il faudrait alors penser à le mettre à jour pendant les opérations de rotation notamment.

## Exercice 41, page 91

Comme il s'agit d'un arbre binaire de recherche, la liste ordonnée est obtenue par un parcours infixe. On commence par écrire une méthode récursive qui prend en argument supplémentaire la liste `l` dans laquelle il faut ajouter les éléments.

```
static void toList(LinkedList<Integer> l, AVL t) {
  if (t == null) return;
  toList(l, t.left);
  l.add(t.value);
  toList(l, t.right);
}
```

Puis il suffit de l'appeler avec une liste initialement vide, que l'on renvoie une fois l'appel terminé.

```
static LinkedList<Integer> toList(AVL t) {
  LinkedList<Integer> l = new LinkedList<Integer>();
  toList(l, t);
  return l;
}
```

La complexité est clairement linéaire en le nombre d'éléments.

## Exercice 42, page 91

On suit l'indication, en écrivant une méthode auxiliaire qui construit un AVL à partir des `n` premiers éléments de la file. Si `n` vaut 0, c'est immédiat.

```
static AVL ofList(Queue<Integer> l, int n) {
  if (n == 0) return null;
```

Sinon, on construit un arbre avec $\lfloor \frac{n-1}{2} \rfloor$ éléments à gauche et $\lceil \frac{n-1}{2} \rceil$ éléments à droite. On commence par un appel récursif pour construire le sous-arbre gauche, puis on retire l'élément qui sera à la racine de la file, puis enfin on construit le sous-arbre droit par un second appel récursif. Ainsi, on garantit que l'ordre des éléments est préservé.

```
  int n1 = (n - 1) / 2;
  AVL left = ofList(l, n1);
  int v = l.poll();
  AVL right = ofList(l, n - n1 - 1);
  return new AVL(left, v, right);
}
```

La méthode demandée s'obtient alors simplement en prenant pour le nombre d'éléments la longueur de la file.

```
static AVL ofList(Queue<Integer> l) {
  return ofList(l, l.size());
}
```

## Exercice 43, page 91

L'idée est simple : on transforme les arbres en listes avec `toList`, puis on effectue l'opération ensembliste sur les listes, puis enfin on transforme la liste obtenue en arbre avec `ofList`. Les listes étant ordonnées, les opérations ensemblistes sur les listes peuvent être effectuées en temps linéaire. Prenons l'exemple de l'union.

```
static AVL union(AVL t1, AVL t2) {
  return ofList(listUnion(toList(t1), toList(t2)));
}
```

L'union de deux listes triées est un exercice indépendant. On peut par exemple l'écrire ainsi :

```
static LinkedList<Integer>
       listUnion(LinkedList<Integer> l1, LinkedList<Integer> l2) {
  LinkedList<Integer> l = new LinkedList<Integer>();
  while (!l1.isEmpty() || !l2.isEmpty()) {
    if (l2.isEmpty() || !l1.isEmpty() && l1.peek() <= l2.peek())
     l.add(l1.remove());
    else
     l.add(l2.remove());
  }
  return l;
}
```

L'intersection et la différence s'écrivent de façon similaire.

## Exercice 44, page 94

Comme pour la méthode `add`, on écrit une boucle `for` qui descend dans l'arbre en suivant les caractères de la chaîne `s`. Si à un certain moment l'arbre devient `null`, on termine sort directement avec `return` car la chaîne `s` ne se trouve pas dans l'arbre.

```
void remove(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) { // invariant t != null
    t = t.branches.get(s.charAt(i));
    if (t == null) return;
  }
  t.word = false;
}
```

Le défaut d'une telle méthode `remove` est qu'elle conduit à des sous-arbres vides, qui occupent inutilement de l'espace et augmentent potentiellement le coût de futures opérations. L'exercice suivant propose d'y remédier.

## Exercice 45, page 94

Comme suggéré par l'énoncé, on va procéder récursivement. Comme il s'agit de parcourir les caractères de la chaîne, on ajoute en argument le prochain indice de la chaîne à considérer. Si on parvient au bout de la chaîne, on positionne le champ `word` à `false` pour signifier la suppression.

```
void removeRec(String s, int i) {
  if (i == s.length()) { this.word = false; return; }
```

Sinon, on descend dans l'arbre en suivant le caractère `i`. Si la branche correspondante n'existe pas, on termine immédiatement avec `return`.

```
  char c = s.charAt(i);
  Trie b = this.branches.get(c);
  if (b == null) return;
```

Sinon, on poursuit la descente récursivement. La modification apportée par rapport à l'exercice précédent consiste en la toute dernière instruction, après l'appel récursif. Si le sous-arbre `b` est devenu vide, alors on le supprime de la table `this.branches`. On se sert de la méthode `isEmpty` suggérée dans l'énoncé.

```
  b.removeRec(s, i+1);
  if (b.isEmpty()) this.branches.remove(c);
}
```

Note : c'est justement parce qu'on a souhaité effectuer une instruction après la suppression qu'on a privilégié une écriture récursive. Pour écrire `remove`, il suffit maintenant d'appeler `removeRec` avec l'indice 0.

```
void remove(String s) {
  removeRec(s, 0);
}
```

## Exercice 46, page 94

Ce n'est pas une question difficile en soi, mais il y a beaucoup de petites modifications à faire. Dans le constructeur, on n'initialise plus le champ `branches`.

```
  this.branches = null;
```

Dans la méthode `contains`, il faut tester si `branches` est `null` avant de chercher à descendre dans le sous-arbre correspondant au `i`-ième caractère.

```
  for (int i = 0; i < s.length(); i++) { // invariant t != null
    if (t.branches == null) return false;
    ...
```

De même, dans la boucle de la méthode `add`, il faut créer la table `branches` si elle n'existe pas encore.

```
Map<Character, Trie2> b = t.branches;
if (b == null) b = t.branches = new HashMap<Character, Trie2>();
...
```

Dans la méthode `remove`, il faut tester si `branches` est `null` avant de chercher à accéder au sous-arbre, comme pour `contains`.

```
if (this.branches == null) return;
```

Enfin, si on a réalisé l'optimisation proposée dans l'exercice précédent, il faut alors ajouter une dernière ligne à la méthode `remove` pour remettre `branches` à `null` lorsque la table de hachage redevient vide.

```
...
if (b.isEmpty()) this.branches.remove(c);
if (this.branches.isEmpty()) this.branches = null;
```

## Exercice 47, page 98

Si on ajoute le qualificatif `final` sur le champ `branches`, cela signifie que la valeur de ce champ, en tant que pointeur vers un objet de la classe `HashMap` ne pourra plus être modifiée. En revanche, le *contenu* de cette table de hachage pourra continuer à être modifié (avec la méthode `put` de la table de hachage, par exemple).

Si on veut réaliser des arbres de préfixes immuables, il faut que le branchement soit réalisé par une structure elle-même immuable. Une solution consisterait à *copier* la table de hachage contenue dans `branches` chaque fois que l'on souhaite effectuer une modification. Cette solution est relativement coûteuse. Une meilleure solution consiste à utiliser une structure de dictionnaire immuable plutôt que `HashMap`. La bibliothèque standard de Java ne fournit pas de telle structure. On peut en construire une par exemple à l'aide d'arbres binaires de recherche immuables (équilibrés ou pas), comme expliqué ci-dessus. Mais c'est pas mal de travail.

## Exercice 48, page 101

Le plus simple consiste à écrire une méthode `checkIndex` dans la classe `Rope` pour effectuer cette vérification. On la déclare `protected` pour qu'elle ne soit visible que dans les sous-classes.

```
protected void checkIndex(int i) {
  if (i < 0 || i >= this.length) throw new IllegalArgumentException();
}
```

Il suffit alors de l'appeler au début de chacune des deux méthodes `get`.

```
char get(int i) {
  checkIndex(i);
  ...
```

## Exercice 49, page 102

On procède exactement comme dans l'exercice 48, en écrivant une seule méthode de vérification dans la classe `Rope`

```
protected void checkRange(int begin, int end) {
  if (begin < 0 || end < begin || end > this.length)
    throw new IllegalArgumentException();
}
```

que l'on appelle ensuite au début des deux méthodes `sub`.

## Exercice 50, page 102

Le champs `length` de la classe `Rope` est déclaré comme `protected`, car on veut pouvoir y accéder depuis les sous-classes `Str` et `App`. Les champs propres aux classes `Str` et `App` peuvent (et doivent) être déclarés comme `private`.

## Exercice 51, page 102

L'idée est ici d'éviter de faire des concaténations de chaînes dans la classe `App`, qui seraient coûteuses. On peut facilement écrire une méthode `toString` de complexité linéaire (amortie) en utilisant un `StringBuffer` dans lequel on va concaténer tous les morceaux de la corde, dans l'ordre. On commence par écrire une méthode `toString` dans la classe `Rope` qui construit ce `StringBuffer` et le passe à une méthode auxiliaire `toStringHelper` chargée de le remplir.

```
class Rope {
  public String toString() {
    StringBuffer b = new StringBuffer();
    toStringHelper(b);
    return b.toString();
  }
```

Cette méthode auxiliaire est déclarée dans cette même classe `Rope` comme une méthode abstraite.

```
  abstract void toStringHelper(StringBuffer b);
}
```

Il n'y a plus qu'à implémenter cette méthode dans les deux sous-classes. Dans la classe `Str`, c'est immédiat.

```
void toStringHelper(StringBuffer b) {
  b.append(this.str);
}
```

Dans la classe `App`, on procède à deux appels récursifs, en commençant bien sûr par le sous-arbre de gauche.

```
void toStringHelper(StringBuffer b) {
  this.left.toStringHelper(b);
  this.right.toStringHelper(b);
}
```

## Exercice 52, page 102

Il suffit d'ajouter la ligne suivante au début du code de `sub`, que ce soit dans la classe `Str` ou dans la classe `App` :

```
if (begin == 0 && end == this.length) return this;
```

On évite ainsi de descendre jusqu'aux feuilles de la corde et de la reconstruire inutilement. Outre le gain de temps, on économise de l'espace en partageant autant que possible les morceaux de cordes déjà construits.

## Exercice 53, page 102

Pour bien faire les choses, on commence par introduire une constante pour la borne choisie :

```
private static final int SMALL_LENGTH = 256;
```

La méthode `append` commence par construire la concaténation comme précédemment, avec `new App`. Si le résultat a une longueur inférieure à la borne choisie, alors on construit une feuille avec `new Str`. La chaîne de cette feuille est simplement obtenue avec la méthode `toString`.

```
Rope append(Rope r) {
  if (this.length == 0) return r;
  if (r.length == 0) return this;
  Rope r1 = new App(this, r);
  if (r1.length < SMALL_LENGTH) return new Str(r1.toString());
  return r1;
}
```

C'est légèrement inefficace de construire le nœud `App` dans le cas où on va finalement construire un nœud `Str`, mais cela facilite grandement le reste du code.

## Exercice 54, page 108

Tant qu'on n'est pas arrivé à la racine de l'arbre, on calcule l'indice du père de `i`.

```
void moveUp(E x, int i) {
  while (i > 0) {
    int fi = (i - 1) / 2;
```

On compare alors l'élément `y` en `fi` avec `x`. Si `y` est inférieur ou égal, on sort de la boucle avec `break`.

```
      E y = this.elts.get(fi);
      if (y.compareTo(x) <= 0) break;
```

Sinon, on fait descendre `y` et `i` prend la valeur de `fi`.

```
      this.elts.set(i, y);
      i = fi;
    }
```

Une fois sorti de la boucle (soit par le `break`, soit parce que `i` vaut 0), on affecte la valeur de `x` à la case `i`.

```
    this.elts.set(i, x);
  }
```

## Exercice 55, page 110

Il n'y a pas de surprise. Une première boucle met tous les éléments du tableau dans un tas initialement vide et une second boucle les ressort dans l'ordre pour les remettre dans le tableau.

```
static void heapsort(int[] a) {
  Heap h = new Heap();
  for (int x : a) h.add(x);
  for (int i = 0; i < a.length; i++) {
    a[i] = h.getMin();
    h.removeMin();
  }
}
```

La complexité se déduit immédiatement du fait que `add` et `getMin` ont chacune une complexité $O(\log n)$ où $n$ est le nombre d'éléments du tas. Du coup, chacune des deux boucles a une complexité totale

$$\log 1 + \log 2 + \cdots + \log N \sim N \log N$$

où $N$ est le nombre total d'éléments. La complexité de ce tri par tas est donc $O(N \log N)$. Le chapitre 13 établit que c'est là une complexité optimale pour un tri n'effectuant que des comparaisons. En revanche, ce tri utilise un espace supplémentaire en $O(N)$.

## Exercice 56, page 113

Comme on dispose déjà de la méthode `merge` sur les arbres, il est immédiat de réaliser la fusion de deux tas. Il faut juste penser à mettre à jour le champ `size`.

```
void merge(SkewHeap<E> that) {
  this.root = merge(this.root, that.root);
  this.size += that.size;
}
```

## Exercice 57, page 115

On crée une file de priorité, dans laquelle on met les $K$ premières valeurs. (S'il y en a moins que $K$, le problème est immédiatement résolu.) Ensuite, pour chaque nouvelle valeur $x$, on la compare à la plus petite valeur $m$ dans la file de priorité (obtenue avec `peek`). Si $x > m$, on supprime $m$ de la file de priorité et on ajoute $x$. La file de priorité contient en permanence $K$ éléments et les opérations d'ajout et de retrait se font donc en

temps $\log K$. On a une complexité totale en $O(N \log K)$ en temps. En espace, la complexité est $O(K)$ car on ne conserve jamais plus de $K$ valeurs en mémoire. C'est particulièrement intéressant si $N$ est grand devant $K$. Ainsi, si les $N$ valeurs sont lues à la volée, on peut parvenir à déterminer les $K$ plus grandes, quand bien même les $N$ valeurs ne tiennent pas toutes simultanément en mémoire.

Note : Nous verrons dans la section 13.4 qu'il est possible de construire un tas à partir de $N$ valeurs en temps $O(N)$ (par exemple à partir d'un tableau), ce qui est plus rapide que de les ajouter une par une avec une opération logarithmique. Dès lors, dans le cas où les $N$ valeurs tiennent toutes simultanément en mémoire, une autre solution consiste à construire un tas avec toutes les valeurs, avec cette fois la relation inverse, c'est-à-dire avec retrait du plus grand élément. Il n'y a plus ensuite qu'à extraire les $K$ plus grandes valeurs avec $K$ appels à `removeMax`. Cette fois, la complexité est $O(N+K \log N)$. Selon les valeurs respectives de $N$ et $K$, cela peut être plus intéressant que la solution précédente, même si cela utilise plus de mémoire.

## Exercice 58, page 120

Si les variables concernées sont $x_0, \ldots, x_{n-1}$, on crée une structure *union-find* de taille $n$ puis on fait `union(`$i$`,`$j$`)` pour chaque égalité $x_i = x_j$ supposée. Enfin, on teste si $x_i = x_j$ est une conséquence avec `find(`$i$`)==find(`$j$`)`.

## Exercice 59, page 120

On rajoute un champ pour le nombre de classes. Ce champ est privé car on veut garantir qu'il correspond bien au nombre de classes à chaque instant. Une méthode en donne la valeur.

```
private int numClasses;
int numClasses() { return this.numClasses; }
```

Dans le constructeur, `numClasses` est initialisé à `n`.

```
UnionFind(int n) {
  ...
  this.numClasses = n;
```

Enfin, il faut mettre à jour `numClasses` dans la méthode `union`.

```
void union(int i, int j) {
  ...
  if (ri == rj) return; // déjà dans la même classe
  this.numClasses--;
  ...
```

## Exercice 60, page 120

Écrivons directement un code générique, c'est-à-dire paramétré par le type `E` des éléments. Les deux champs `link` et `rank` sont maintenant des tables de hachage indexées par les éléments.

```
class HashUnionFind<E> {
  private HashMap<E, E> link;
  private HashMap<E, Integer> rank;
```

Comme les éléments ne sont plus nécessairement des entiers consécutifs, on ne passe plus d'argument au constructeur. Celui-ci se contente donc de créer deux tables de hachage vides.

```
HashUnionFind() {
  this.link = new HashMap<>();
  this.rank = new HashMap<E, Integer>();
}
```

L'ajout des éléments peut alors se faire par une méthode `add`, qui crée une nouvelle classe réduite à un singleton.

```
void add(E x) {
  this.link.put(x, x);
  this.rank.put(x, 0);
}
```

On a conservé ici l'idée qu'un représentant est associé à lui-même dans `link`. Il ne reste plus qu'à adapter les méthodes `find` et `union`. Cela consiste uniquement à remplacer `link[`$i$`]` par `link.get(`$i$`)`, `link[`$i$`]=`$j$ par `link.put(`$i$`,`$j$`)`, etc.

## Exercice 61, page 122

Le constructeur construit une classe réduite à un élément. Son code est immédiat. On adopte ici la convention suggérée par l'énoncé, à savoir que le champ `link` d'un représentant vaut `null`.

```
Elt(E x) {
  this.value = x;
  this.link = null;
  this.rank = 0;
}
```

La méthode `find` devient une méthode dynamique, qui renvoie le représentant de la classe de `this`. On procède comme précédemment, en commençant par vérifier s'il s'agit d'un représentant. Le cas échéant, on renvoie l'élément lui-même, c'est-à-dire `this`. Sinon, on calcule le représentant récursivement, sans oublier de réaliser la compression de chemin avant de renvoyer le résultat.

```
Elt<E> find() {
  if (this.link == null) return this;
  Elt<E> repr = this.link.find();
  this.link = repr; // compression de chemin
  return repr;
}
```

La méthode `union` devient également une méthode dynamique. Elle réalise l'union des classes de `this` et de son argument `that`. Le code est là encore identique à ce que nous avons écrit précédemment.

```
void union(Elt<E> that) {
  Elt<E> rthis = this.find();
  Elt<E> rthat = that.find();
  if (rthis == rthat) return; // déjà dans la même classe
  if (rthis.rank < rthat.rank)
    rthis.link = rthat;
  else {
    rthat.link = rthis;
    if (rthis.rank == rthat.rank) rthis.rank++;
  }
}
```

Il est important de noter qu'on compare ici les représentants `rthis` et `rthat` directement avec l'égalité physique. En effet, le représentant d'une classe est unique par construction.

## Exercice 62, page 122

On commence par introduire une classe `Wall` pour représenter un mur (potentiel) entre deux cases adjacentes. Le booléen `closed` indique si le mur est présent. Le champ `direction` indique s'il s'agit de la séparation avec la case située à droite (`Vertical`) ou en dessous (`Horizontal`).

```
enum Direction { Vertical, Horizontal };
class Wall {
  final int x, y;
  final Direction direction;
  boolean closed;
}
```

On omet ici le code du constructeur, qui initialise notamment `closed` à `true`. On écrit alors une méthode `buildMaze` qui prend en arguments les dimensions du labyrinthe. On commence par construire tous les murs possibles. Pour s'épargner la peine de les dénombrer, on les stocke dans un tableau redimensionnable.

```
void buildMaze(int width, int height) {
  Vector<Wall> walls = new Vector<Wall>();
  for (int x = 0; x < width; x++)
    for (int y = 0; y < height; y++) {
```

```
      if (x < width - 1)
        walls.add(new Wall(x, y, Direction.Vertical));
      if (y < height - 1)
        walls.add(new Wall(x, y, Direction.Horizontal));
    }
```

Puis on mélange ce tableau. On suppose ici avoir écrit une méthode `shuffle` effectuant un tel mélange, par exemple en suivant l'exercice 5 page 35.

```
shuffle(walls);
```

Comme suggéré dans l'énoncé, on crée alors une structure *union-find* contenant toutes les cases du labyrinthe. La case $(x, y)$ peut être identifiée à l'entier $x * \texttt{height} + y$.

```
int n = width * height;
UnionFind uf = new UnionFind(n); // codage : cell = x * height + y
```

Enfin, on parcourt tous les murs dans l'ordre donné par le mélange. Pour chaque mur, on calcule les deux entiers `cell` et `next` correspondant aux deux cases situées de part et d'autre du mur. Si les deux cases sont déjà dans la même classe, on passe directement à l'itération suivante de la boucle avec `continue`. Sinon, on fait l'union des deux classes et on « ouvre » le mur entre les deux cases en positionnant son champ `closed` à `false`.

```
for (Wall w: walls) {
  int cell = w.x * height + w.y;
  int next = cell +
             (w.direction == Direction.Horizontal ? 1 : height);
  if (uf.find(cell) == uf.find(next)) continue;
  uf.union(cell, next);
  w.closed = false;
}
```

Ceci achève la construction du labyrinthe. On peut maintenant le dessiner facilement, par exemple en parcourant la liste des murs donnée par `walls`.

Justifions qu'il s'agit là d'un labyrinthe parfait, par récurrence sur la dernière boucle du programme ci-dessus. L'hypothèse de récurrence est qu'à tout moment deux cases sont reliées par un chemin si et seulement si elles appartiennent à la même classe et que ce chemin est alors unique. Initialement, c'est trivialement vrai car chaque classe ne contient qu'une seule case. Supposons la propriété vraie et effectuons un tour de boucle. Si les deux cases `cell` et `next` choisies sont déjà dans la même classe, on ne fait rien et la propriété reste donc trivialement vérifiée. Si en revanche on réunit les deux classes, considérons deux cases $a$ et $b$ dans cette nouvelle classe. Si $a$ et $b$ sont toutes deux dans l'ancienne classe de `cell`, appelons-la $C$, alors elles sont reliées par un unique chemin dans $C$. Si un autre chemin existait, il devrait emprunter deux fois `w` et ce ne serait pas un chemin. De même si $a$ et $b$ sont toutes deux dans la classe de `next`. Si enfin $a$ est dans la classe de `cell` et $b$ dans la classe de `next` (ou le contraire), alors par hypothèse il existe un unique chemin de $a$ à `cell` et un unique chemin de `next` à $b$, donc un unique chemin de $a$ à $b$.

## Exercice 63, page 126

Une solution très simple consiste à s'assurer que `u` et `v` sont bien positifs ou nuls avant de commencer le calcul, en en prenant l'opposé si nécessaire.

```
static int gcd(int u, int v) {
  if (v < 0) v = -v;
  if (u < 0) u = -u;
  ...
```

## Exercice 64, page 126

La valeur renvoyée est celle de `u`. Si au moins un tour de boucle est effectué, alors `u` prend la valeur de `v` précédente, qui est alors nécessairement non nulle (c'est le test de la boucle). Donc si `gcd` renvoie zéro, c'est qu'aucun tour de boucle n'a été effectué, ce qui veut dire que $\texttt{u} = \texttt{v} = 0$.

## Exercice 65, page 126

Si $u = v$, alors un seul tour de boucle est effectué car $v$ va prendre immédiatement la valeur 0. La complexité est donc $O(1)$. Si $v > u$, alors le tout premier tour de boucle va échanger les valeurs de $u$ et $v$ car $u \mod v = u$. On se retrouve alors dans les conditions du théorème de Lamé, c'est-à-dire avec une complexité $O(\log v)$. La complexité est donc dans tous les cas en $O(\log(\max(u, v))$.

## Exercice 66, page 127

Le plus simple est de ne pas utiliser de tableau du tout, sauf à la toute fin de la méthode pour renvoyer le résultat. On utilise donc ici trois entiers `u0`, `u1`, `u2` pour ce qui était auparavant le tableau `u` et de même pour les tableaux `v` et `t`.

```
static int[] extendedGcd(int u, int v) {
  int u0 = 1, u1 = 0, u2 = u, v0 = 0, v1 = 1, v2 = v;
  while (v2 != 0) {
    int q = u2 / v2;
    int t0 = u0 - q * v0, t1 = u1 - q * v1, t2 = u2 - q * v2;
    u0 = v0; u1 = v1; u2 = v2;
    v0 = t0; v1 = t1; v2 = t2;
  }
  return new int[] { u0, u1, u2 };
}
```

Bien qu'on ne change pas la complexité, cette version-là est bien meilleure que celle utilisant des tableaux car tous les calculs se font maintenant avec des variables allouées sur la pile et non plus des tableaux alloués sur le tas. En particulier, on ne sollicite plus du tout le GC, sauf à la toute fin de la fonction.

## Exercice 67, page 127

Vue l'hypothèse $\gcd(v, m) = 1$, l'algorithme d'Euclide étendu nous donne deux entiers $a$ et $b$ tels que

$$av + bm = \gcd(v, m) = 1$$

En multipliant cette égalité par $u$, on obtient

$$(au)v + (bu)m = u$$

et donc $au$ est une solution, ou plus exactement $(au) \mod m$. En écrivant le code, il faut faire attention au fait que $a$ peut être négatif.

```
static int divMod(int u, int v, int m) {
  int[] g = extendedGcd(v, m);
  int r = (g[0] * u) % m;
  return r < 0 ? r + m : r; // assure un résultat positif ou nul
}
```

## Exercice 68, page 128

Cette variante consiste à prendre le carré après l'appel récursif, plutôt qu'avant.

```
static int exp(int x, int n) {
  if (n == 0) return 1;
  int r = exp(x, n / 2);
  return n % 2 == 0 ? r * r : x * r * r;
}
```

Il n'y a pas de différence d'efficacité, car les nombre d'appels récursifs, de divisions et de multiplications restent exactement les mêmes.

## Exercice 69, page 128

Une variable `r` contient la valeur qui sera renvoyée au final. On l'initialise à 1. Tant que `n` n'est pas nul, la boucle va mettre `x` au carré et diviser `n` par deux. Auparavant, elle multiplie `r` par `x` si `n` est impair.

```
static int exp(int x, int n) {
  int r = 1;
  while (n != 0) {
    if (n % 2 == 1) r *= x;
    x *= x;
    n /= 2;
  }
  return r;
}
```

La correction de ce calcul est assurée par l'invariant $\mathtt{r} \times \mathtt{x}^{\mathtt{n}} = \mathtt{x}_0^{\mathtt{n}_0}$ où $\mathtt{x}_0$ et $\mathtt{n}_0$ désignent les valeurs initiales de `x` et `n`.

## Exercice 70, page 128

Comme suggéré dans l'énoncé, on commence par écrire une classe `Mat22` pour les matrices $2 \times 2$. La multiplication de matrice est laissée en exercice. (Ici, elle peut être écrite « brutalement » avec 8 multiplications et 4 additions.)

```
class Mat22 {
  int m[][];
  Mat22(int m00, int m01, int m10, int m11) {
    this.m = new int[][] { { m00, m01 }, { m10, m11 } };
  }
  static Mat22 identity = new Mat22(1, 0, 0, 1);
  Mat22 mul(Mat22 x) { ... }
```

Puis on écrit l'algorithme d'exponentiation rapide sur ce type. Si on choisit d'écrire une méthode statique, le code est identique à celui du programme 23. Seuls les types et les opérations atomiques changent.

```
  static Mat22 exp(Mat22 x, int n) {
    if (n == 0) return identity;
    Mat22 r = exp(x.mul(x), n / 2);
    return n % 2 == 0 ? r : x.mul(r);
  }
}
```

On peut tout aussi bien écrire une méthode dynamique, avec pour type `Mat22 exp(int n)`, qui met la matrice `this` à la puissance `n`. On le laisse en exercice. Une fois l'exponentiation rapide écrire dans la classe `Mat22`, on en déduit facilement le calcul de $F_n$.

```
static Mat22 Mfib = new Mat22(1, 1, 1, 0);
static int fib(int n) {
  Mat22 r = Mat22.exp(Mfib, n);
  return r.m[0][1];
}
```

## Exercice 71, page 130

Il suffit de modifier les deux boucles pour ne parcourir que les entiers impaires à partir de de 3, sans oublier d'indiquer au préalable que 2 est premier.

```
  prime[2] = true;
  for (int i = 3; i <= max; i += 2)
    prime[i] = true;
  for (int n = 3; n * n <= max; n += 2)
    ..
```

Le reste est inchangé.

## Exercice 72, page 130

La classe `BitSet` s'utilise comme un tableau, avec une méthode `get` pour accéder à un élément, une méthode `clear` pour mettre un élément à `false` et une méthode `set` pour mettre un élément à `true`.

## Exercice 73, page 130

Le plus simple est de réutiliser la fonction `sieve`. On l'appelle puis on compte le nombre de nombres premiers.

```
static int[] firstPrimesUpto(int max) {
  boolean[] prime = sieve(max);
  int count = 0;
  for (boolean b: prime) if (b) count++;
```

On alloue alors un tableau de cette taille dans lequel on range les nombres premiers par un second parcours.

```
  int[] res = new int[count];
  for (int i = 2, next = 0; i <= max; i++)
    if (prime[i])
     res[next++] = i;
  return res;
}
```

## Exercice 74, page 130

Là encore, le plus simple est de se resservir de la méthode `sieve`. On commence par calculer une limite pour le crible, avec la propriété donnée en indication, en prenant soin d'assurer $n \geq 6$ dans la formule.

```
static int[] firstNPrimes(int n) {
  int m = Math.max(6, n);
  int max = (int)(m * (Math.log(m) + Math.log(Math.log(m))));
  boolean[] prime = sieve(max);
```

Puis il suffit de ranger les nombres premiers trouvés dans un nouveau tableau, de taille `n`. Attention : on s'arrête dès qu'on en a trouvé `n` et non pas lorsque tout le tableau `prime` est parcouru, car il peut y en avoir plus que `n`.

```
  int[] res = new int[n];
  for (int i = 2, next = 0; next < n; i++)
    if (prime[i])
      res[next++] = i;
  return res;
}
```

## Exercice 75, page 135

Il suffit de modifier la boucle interne pour ne pas démarrer à `i` mais à $\min(\texttt{i}, \texttt{k})$.

```
for (int j = Math.min(i, k); j >= 1; j--)
  row[j] += row[j-1];
```

## Exercice 76, page 135

La classe `java.math.BigInteger` de la bibliothèque standard de Java fournit un type immuable pour des entiers en précision arbitraire. On trouve en particulier tout ce dont on a besoin ici, à savoir deux constantes `BigInteger.ZERO` et `BigInteger.ONE` et une méthode `add`. La seule subtilité concerne l'initialisation du tableau `row`. Avec des éléments de type `int`, les valeurs sont initialisées par 0, ce qui nous convient. Avec des éléments de type `BigInteger`, en revanche, elles sont initialisées par `null`, car un entier de type `BigInteger` est un objet. Il faut donc ajouter une boucle explicite pour l'initialisation de `row`.

```
BigInteger[] row = new BigInteger[k+1];
row[0] = BigInteger.ONE;
for (int j = 1; j <= k; j++)
  row[j] = BigInteger.ZERO;
```

Le reste est facilement adapté. On peut alors calculer $C(2000, 1000)$ en une fraction de seconde. On obtient un entier de 601 chiffres `20481516...49120`.

## Exercice 77, page 136

Comme pour le triangle de Pascal, on réalise qu'il est inutile de conserver toutes les valeurs de la suite de Fibonacci déjà calculées. Seules les deux plus récentes sont nécessaires. Dès lors, le programme se simplifie en utilisant deux variables entières `a` et `b` contenant deux nombres consécutifs de la suite. Initialement, on leur donne les valeurs $F_0 = 0$ et $F_1 = 1$. Puis on répète `n` fois l'opération $a, b \leftarrow b, a + b$.

```
static long fibDP(int n) {
  int a = 0, b = 1; // a = F(k), b = F(k+1)
  while (n-- > 0) {
    b = b + a;
    a = b - a;
  }
  return a;
}
```

## Exercice 78, page 137

Le programme reste correct. En effet, on représente un sous-ensemble du tableau `s` par un masque des indices sélectionnés et non pas comme un sous-ensemble d'éléments de ce tableau.

## Exercice 79, page 137

On commence par appeler la méthode `computeAll` pour connaître tous les entiers atteignables. Ensuite, il suffit de considérer tous les entiers par distance croissante à la cible, jusqu'à ce que l'un d'eux soit atteignable.

```
static String solve(int[] s, int target) {
  Map<Integer, Map<Integer, String>> res = computeAll(s);
  Map<Integer, String> tous = res.get((1 << s.length) - 1);
  for (int d = 0; true; d++) {
    if (tous.containsKey(target-d))
      return (target-d) + " = " + tous.get(target-d);
    if (tous.containsKey(target+d))
      return (target+d) + " = " + tous.get(target+d);
  }
}
```

Note : le compilateur Java ne réclame pas d'instruction `return` à la fin de cette méthode, car on ne peut pas sortir de la boucle `for` autrement que par `return`.

## Exercice 80, page 137

Il suffit de supprimer la ligne

```
m.putAll(res.get(left));
```

qui ajoute à `m` tous les entiers que l'on former à partir de `left`.

## Exercice 81, page 141

Si le problème initial est donné sous la forme d'une chaîne de 81 caractères, il suffit de convertir chacun d'entre eux en un entier.

```
Sudoku(String s) {
  this.grid = new int[81];
  for (int i = 0; i < 81; i++)
    this.grid[i] = s.charAt(i) - '0';
}
```

Les caractères étant des entiers en Java, on calcule l'entier correspondant par soustraction avec le caractère `'0'`, car les caractères `'0'`, `'1'`, ..., `'9'` sont consécutifs dans le jeu de caractères.

## Exercice 82, page 141

On peut faire un petit effort pour que le résultat soit joli et notamment pour que les sous-groupes $3 \times 3$ soient bien identifiés. Pour parcourir toutes les cases, on a le choix entre une boucle de 0 à 80 ou deux boucles imbriquées de 0 à 8. C'est ce dernier choix qui est fait ici.

```
  void print() {
    for (int i = 0; i < 9; i++) {
      if (i % 3 == 0) System.out.println("+--+--+--+");
      for (int j = 0; j < 9; j++) {
        if (j % 3 == 0) System.out.print("|");
        System.out.print(this.grid[9*i+j]);
      }
      System.out.println("|");
    }
    System.out.println("+--+--+--+");
  }
```

```
+---+---+---+
|795|324|168|
|821|956|743|
|643|187|259|
+---+---+---+
|538|261|497|
|964|573|812|
|217|498|635|
+---+---+---+
|479|815|326|
|156|732|984|
|382|649|571|
+---+---+---+
```

## Exercice 83, page 144

On commence par écrire une méthode récursive `countRec` sur le modèle de la méthode `findSolutionRec`. En particulier, on réutilise la méthode `check`.

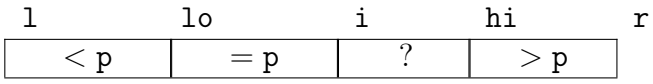
```
static int countRec(int[] cols, int r) {
  if (r == cols.length) return 1;
  int f = 0;
  for (int c = 0; c < cols.length; c++) {
    cols[r] = c;
    if (check(cols, r)) f += countRec(cols, r + 1);
  }
  return f;
}
```

La variable `f` sert à sommer les nombres de solutions trouvés récursivement. On note que si aucune reine ne peut être placée sur la ligne, les tests de `check` sont tous faux et la somme renvoyée vaut donc 0. Enfin, on écrit une méthode `count` qui appelle `countRec` pour obtenir le nombre total.

```
static int count(int n) {
  int[] cols = new int[n];
  return countRec(cols, 0);
}
```

## Exercice 84, page 154

On prend toujours `p = a[l]` comme pivot. Pour découper `a[l..r[` en trois, on va parcourir l'intervalle de gauche à droite avec un indice `i`, en maintenant le découpage suivant :




On initialise `lo` à `l` et on démarre le découpage avec `i` = `l`+1, ce qui décompte la présence du pivot dans la portion centrale.

```
static void quickrec(int[] a, int l, int r) {
  if (l >= r - 1) return;
  int p = a[l], lo = l, hi = r;
```

Le partage proprement dit est réalisé en comparant chaque valeur `a[i]` au pivot. Quand il y a égalité, il suffit de passer à la valeur suivante. Quand `a[i]` $<$ `p`, on échange `a[i]` et `a[lo]` et on incrémente `i` et `lo`. Enfin, quand `a[i]` $>$ `p`, on décrémente `hi` puis on échange `a[i]` et `a[hi]`.

```
  for (int i = l+1; i < hi; ) {
    if (a[i] == p)
      i++;
    else if (a[i] < p)
      swap(a, i++, lo++);
    else
      swap(a, i, --hi);
  }
```

Enfin, on conclut par les deux appels récursifs, sur les portions `a[l..lo[` et `a[hi..r[`.

```
  quickrec(a, l, lo);
  quickrec(a, hi, r);
}
```

## Exercice 85, page 154

Si `m` désigne la valeur renvoyée par `partition`, alors le premier intervalle à trier a pour longueur `m-l` et le second pour longueur `r-m-1`. On compare donc ces deux valeurs pour effectuer l'appel récursif sur le plus petit des deux intervalles. Pour ce qui est de l'autre intervalle, on met à jour `l` ou `r`, selon le cas, et on place tout le code dans une boucle `while`.

```
static void quickrec(int[] a, int l, int r) {
  while (r - l > 1) {
    int m = partition(a, l, r);
    if (m - l < r - m - 1) {
      quickrec(a, l, m);
      l = m + 1;
    } else {
      quickrec(a, m + 1, r);
      r = m;
    }
  }
}
```

Vu que l'appel récursif est effectué sur le plus petit des deux intervalles, sa longueur est au moins deux fois plus petite que `r-l`. La longueur de l'intervalle étant divisée par au moins 2 à chaque fois, il ne peut y avoir qu'un nombre logarithmique d'appels imbriqués avant qu'on atteigne une largeur d'intervalle au plus égale à 1.

## Exercice 86, page 154

Il suffit de remplacer la première ligne de `quickrec` correspondant au cas où l'intervalle contient au plus un élément

```
if (l >= r - 1) return;
```

par

```
if (r - l < cutoff) { insertionSort(a, l, r); return; }
```

où `cutoff` est une constante que l'on a introduite par ailleurs et dont la valeur peut être déterminée empiriquement.

```
private static final int cutoff = 5;
```

## Exercice 87, page 157

Comme suggéré dans l'énoncé, on ajoute à `mergesortrec` un argument booléen `inplace` qui indique si le tri doit être réalisé en place dans `a` (valeur `true`) ou tout en déplaçant les valeurs vers `tmp`.

```
static void mergesortrec(int[] a, int[] tmp, int l, int r,
                         boolean inplace) {
  if (l >= r-1) return;
  int m = l + (r - l) / 2;
  mergesortrec(a, tmp, l, m, !inplace);
  mergesortrec(a, tmp, m, r, !inplace);
  if (inplace) merge(tmp, a, l, m, r); else merge(a, tmp, l, m, r);
}
```

Pour la méthode principale `mergesort`, il y a une petite subtilité. Il ne faut pas prendre pour `tmp` un tableau de contenu quelconque mais une *copie* du tableau `a`. En effet, la méthode `mergesortrec` ne fait rien lorsque `l >= r-1` mais elle doit tout de même assurer le déplacement vers `tmp` lorsque `inplace` vaut `false`. Partir d'une copie de `a` suffit à le garantir.

```
static void mergesort(int[] a) {
  mergesortrec(a, Arrays.copyOf(a, a.length), 0, a.length, true);
}
```

## Exercice 88, page 157

La méthode `split` peut être écrite en utilisant un booléen `b` dont on change la valeur à chaque élément. La liste `l` reçue en argument n'est pas modifiée, mais seulement parcourue par la boucle `for`.

```
static void split(LinkedList<Integer> l,
                  LinkedList<Integer> l1, LinkedList<Integer> l2) {
  boolean b = false;
  for (int x : l)
    ((b = !b) ? l1 : l2).add(x);
}
```

La fusion de deux listes `l1` et `l2` dans une nouvelle liste n'est pas difficile, d'autant qu'on peut ici vider les deux listes au fur et à mesure. Il faut cependant traiter soigneusement le moment où l'une des listes vient à s'épuiser.

```
static LinkedList<Integer> merge(LinkedList<Integer> l1,
                                 LinkedList<Integer> l2) {
  LinkedList<Integer> res = new LinkedList<Integer>();
  while (!l1.isEmpty() && !l2.isEmpty())
    if (l1.getFirst() <= l2.getFirst())
      res.addLast(l1.removeFirst());
    else
      res.addLast(l2.removeFirst());
  while (!l1.isEmpty()) res.addLast(l1.removeFirst());
  while (!l2.isEmpty()) res.addLast(l2.removeFirst());
  return res;
}
```

On a utilisé ici `getFirst` qui renvoie le premier élément d'une liste sans le retirer et `removeFirst` qui retire et renvoie le premier élément. Les éléments sont ajoutés au résultat avec `addLast`, pour préserver l'ordre. (Les méthodes `removeFirst` et `addLast` existent aussi sous les noms plus courts `poll` et `add` mais on a choisi ici d'être explicite.)

On écrit enfin la méthode `mergesort`. Elle partage la liste en deux listes `l1` et `l2` avec `split`, les trie récursivement, puis les fusionne. Il faut penser au cas de base d'une liste ayant au plus un élément pour assurer la terminaison.

```
static LinkedList<Integer> mergesort(LinkedList<Integer> l) {
  if (l.size() <= 1) return l;
  LinkedList<Integer> l1 = new LinkedList<Integer>(),
                      l2 = new LinkedList<Integer>();
  split(l, l1, l2);
  return merge(mergesort(l1), mergesort(l2));
}
```

On peut vérifier que `mergesort` ne modifie pas la liste `l`. On a choisi ici de renvoyer la liste `l` elle-même lorsqu'elle possède au plus un élément. Bien entendu, on pourrait choisir d'en faire plutôt une copie si on a besoin d'assurer que le résultat est toujours une liste différente de l'argument.

## Exercice 89, page 157

La méthode `split` ne pose pas de difficulté particulière. On commence par calculer la longueur `n` de la liste avec un premier parcours. Puis on fait un second parcours pour

avancer `(n-1)/2` fois dans la liste. L'élément `last` auquel on parvient est alors le dernier élément de la première moitié. On coupe la liste en affectant `null` à `last.next`, non sans avoir auparavant conservé la seconde moitié de la liste dans une variable `l2` dont la valeur est renvoyée.

```
static Singly split(Singly l) {
  int n = 0;
  for (Singly l1 = l; l1 != null; l1 = l1.next) n++;
  assert (n >= 2); // on a supposé au moins deux éléments
  Singly last = l;
  for (int i = 0; i < (n-1)/2; i++) last = last.next;
  Singly l2 = last.next;
  last.next = null;
  return l2;
}
```

La fusion en place est plus subtile. On pourrait l'écrire récursivement sans trop de mal mais, comme suggéré dans l'énoncé, cela conduit à des débordements de pile trop rapidement. Pour l'écrire avec une boucle, on va maintenir en permanence la situation suivante

$$\begin{array}{lllll} \texttt{res} \to & \cdots \to & \texttt{l1} \to & \cdots \to & \texttt{null} \\ & & \texttt{l2} \to & \cdots \to & \texttt{null} \end{array}$$

où `res` est la tête de liste qui sera renvoyée au final et où on assure de plus que `l1.element` $\leq$ `l2.element`. La portion de liste entre `res` et `l1` est déjà triée. Les listes `l1` et `l2` contiennent des éléments plus grands, qui restent à fusionner. Initialement, on compare les têtes de `l1` et `l2` et on échange leur rôle si nécessaire.

```
static Singly merge(Singly l1, Singly l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  Singly res;
  if (l1.element <= l2.element) { res = l1; }
  else { res = l2; l2 = l1; l1 = res; }
```

Puis on insère le premier élément de `l2`. Pour cela, on avance dans la liste `l1` tant que l'élément de `l1` est inférieur ou égal au premier élément de `l2`.

```
  while (l2 != null) {
    int v = l2.element;
    assert (l1.element <= v); // invariant
    while (l1.next != null && l1.next.element <= v) l1 = l1.next;
    Singly n = l1.next;
    l1.next = l2;
    l2 = n;
  }
  return res;
}
```

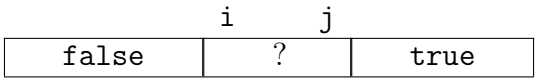
On peut enfin écrire la méthode `mergesort`. Si la liste contient au plus un élément, on ne fait rien. Sinon, on peut appeler `split`, trier les deux moitiés puis enfin les fusionner.

```
static Singly mergesort(Singly l) {
  if (l == null || l.next == null) return l;
  Singly l2 = split(l);
  return merge(mergesort(l), mergesort(l2));
}
```

Les méthodes `split` et `merge` ont été écrites avec des boucles. Seule la méthode `mergesort` utilise de la place sur la pile. Vu que la longueur de la liste est divisée par deux à chaque étape, le nombre d'appels imbriqués à `mergesort` est logarithmique. On ne risque donc pas de débordement de pile, car il faudrait pour cela une liste de longueur $L$ avec $\log L$ de l'ordre de plusieurs milliers et il est impossible de construire une telle liste (dans des limites de temps et surtout ici d'espace raisonnables).

## Exercice 90, page 161

Une façon simple de procéder consiste à délimiter une portion centrale non encore triée, avec des valeurs `false` à sa gauche et des valeurs `true` à sa droite.



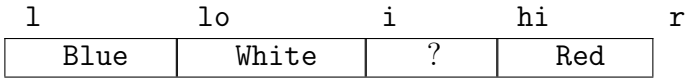

Le code s'en déduit immédiatement.

```
static void twoWaySort(boolean[] a) {
  int i = 0, j = a.length - 1;
  while (i < j) {
    if (!a[i]) i++;
    else if (a[j]) j--;
    else swap(a, i++, j--);
  }
}
```

La complexité est clairement linéaire, car chaque tour de boucle diminue d'au moins une unité la largeur de l'intervalle `i..j`.

## Exercice 91, page 162

Choisissons de procéder par échanges uniquement. C'est alors le même principe que pour l'exercice 84. Il n'est pas inutile de faire un petit dessin.




Pour écrire le code, on exploite le fait qu'on peut utiliser la construction `switch` de Java sur un type énuméré. On rappelle que sans l'instruction `break`, l'exécution se poursuivrait avec le code du cas suivant.

```
static void dutchFlag(Color[] a) {
  int b = 0, i = 0, r = a.length;
  while (i < r)
```

```
      switch (a[i]) {
      case Blue:
        swap(a, i++, b++);
        break;
      case White:
        i++;
        break;
      case Red:
        swap(a, i, --r);
        break;
      }
  }
```

## Exercice 92, page 162

On commence par compter le nombre d'occurrences de chacune des $k$ valeurs possibles. On le fait dans un tableau `count` de taille $k$.

```
  static void countingSort(int[] a, int k) {
    int[] count = new int[k];
    for (int v: a) count[v]++;
```

Puis on remplit le tableau `a` avec `count`[0] valeurs 0, `count`[1] valeurs 1, etc.

```
    int next = 0;
    for (int v = 0; v < k; v++)
      while (count[v]-- > 0)
        a[next++] = v;
  }
```

## Exercice 93, page 163

C'est évident : il suffirait d'itérer ce processus de compression pour parvenir à un fichier de taille nulle, ce qui est absurde.

## Exercice 94, page 169

Comme suggéré dans l'énoncé, on introduit une table de hachage `index` qui associe des feuilles à des caractères. Lorsqu'on rencontre un caractère pour la première fois, on crée une nouvelle feuille et on ajoute l'association dans la table. Puis, que la feuille ait été trouvée dans la table ou qu'on vienne de la créer, on incrémente sa fréquence d'une unité. (Une feuille nouvellement créée a une fréquence 0.)

```
  static Collection<Leaf> buildAlphabet(String text) {
    HashMap<Character, Leaf> index = new HashMap<Character, Leaf>();
    for (int i = 0; i < text.length(); i++) {
      Character c = text.charAt(i);
```

```
      Leaf l = index.get(c);
      if (l == null) { l = new Leaf(c); index.put(c, l); }
      l.freq++;
    }
    return index.values();
  }
```

Pour renvoyer une collection de feuilles au final, on a utilisé la méthode `values` de la table de hachage. Elle renvoie l'ensemble des valeurs, en oubliant les clés, mais ce n'est pas un problème ici car les caractères sont déjà stockés à l'intérieur des feuilles.

## Exercice 95, page 177

Si `L` est le type qui étiquette les arcs, alors la matrice d'adjacence est déclarée de type `L[][]` plutôt que `boolean[][]`. Si `L` est une classe, on prend alors la convention que la valeur `null` désigne l'absence d'arc. (Sinon, il faut une valeur particulière ou utiliser un second tableau.) La méthode `addEdge` doit prendre l'étiquette comme paramètre supplémentaire.

```
  void addEdge(int x, L label, int y) {
    this.m[x][y] = label;
  }
```

Les méthodes `hasEdge` et `removeEdge` sont adaptées pour tester et affecter `null` respectivement.

## Exercice 96, page 178

C'est exactement le nombre d'entrées dans la table de hachage `adj` et on l'obtient avec `adj.size()`.

```
  int nbVertices() { return this.adj.size(); }
```

## Exercice 97, page 180

Contrairement au nombre de sommets, il faut ici maintenir le nombre d'arcs au fur et à mesure de la construction du graphe. Supposons un nouveau champ `nbEdges` à cet effet. Pour `addEdge`, il faut incrémenter ce compteur si l'arc n'était pas déjà présent. Il se trouve que la méthode `add` de la classe `HashSet` renvoie précisément un booléen indiquant si l'insertion a eu lieu (c'est-à-dire si l'élément était absent auparavant). On peut donc écrire

```
  void addEdge(V x, V y) {
    if (this.adj.get(x).add(y)) this.nbEdges++;
  }
```

De même, la méthode `remove` de `HashSet` renvoie un booléen indiquant si la suppression a effectivement eu lieu. On peut donc écrire de façon analogue

```
void removeEdge(V x, V y) {
  if (this.adj.get(x).remove(y))
    this.nbEdges--;
}
```

## Exercice 98, page 180

Si `L` désigne le type des étiquettes des arcs, la table d'adjacence a maintenant le type suivant :

```
Map<V, Map<V, L>> adj;
```

Pour ajouter un arc `x` $\to$ `y` avec une étiquette `label`, il suffit de chercher la table de hachage associée à `x`, puis d'y ajouter l'entrée `y` $\mapsto$ `label`.

```
void addEdge(V x, L label, V y) {
  this.adj.get(x).put(y, label);
}
```

La subtilité vient du fait qu'il peut déjà y avoir un arc entre `x` et `y`, avec potentiellement une autre étiquette. Il est alors écrasé. On pourrait aussi choisir d'échouer avec une exception. En revanche, cette représentation ne permet pas d'avoir plusieurs arcs entre `x` et `y` (ce qu'on appelle un multi-graphe). Le code de `removeEdge`, en revanche, ne change pas. En effet, supprimer `y` dans un ensemble ou dans une table s'écrit de la même façon.

## Exercice 99, page 180

On construit une structure *union-find* avec tous les sommets. Initialement, chaque classe contient un unique sommet. On parcourt alors tous les arcs du graphe et, pour chaque arc $x - y$, on unifie les classes des sommets $x$ et $y$ (avec l'opération *union*). Il ne reste plus qu'à vérifier que tous les sommets sont au final dans la même classe d'équivalence (avec l'opération *find*).

## Exercice 100, page 180

On construit une structure *union-find* avec tous les sommets. Initialement, chaque classe contient un unique sommet. On parcourt alors tous les arcs du graphe et, pour chaque arc $x - y$,

— si $x$ et $y$ sont dans la même classe, on a trouvé un cycle ;
— sinon, on unifie les classes des sommets $x$ et $y$ (avec l'opération *union*).

## Exercice 101, page 186

Comme suggéré dans l'énoncé, on commence par déclarer une seconde table de hachage, `pred`, dans laquelle tout sommet atteint par le parcours sera associé au sommet qui a permis de l'atteindre.

```
HashMap<V, V> pred;
```

Avant d'entrer dans la boucle `while`, on ajoute la source dans cette table en l'associant à `null`. Ceci facilitera le code qu'on écrira plus tard pour reconstruire le chemin.

```
pred.put(source, null);
```

Lorsqu'on atteint un sommet `w` pour la première fois, il faut l'ajouter dans la table `pred`. Le sommet qui a permis de l'atteindre est `v`, le sommet dont on est en train de parcourir les successeurs.

```
if (!this.visited.containsKey(w)) {
  q.add(w);
  visited.put(w, d+1);
  pred.put(w, v);
}
```

À ce point-là, on comprend que les tables `visited` et `pred` sont en partie redondantes, car elles permettent toutes les deux de caractériser les sommets atteints. Si par exemple on ne cherchait pas à conserver la distance dans la table `visited`, alors la seule table `pred` suffirait.

Une fois que le parcours en largeur est terminé, on peut chercher à reconstruire le chemin qui mène de la source à un sommet `v` qui a été atteint. Si par exemple on veut le stocker dans une `LinkedList`, alors on peut procéder ainsi :

```
LinkedList<V> path = new LinkedList<V>();
while (v != null) {
  path.addFirst(v);
  v = pred.get(v);
}
```

On utilise `addFirst` pour que le chemin soit au final dans le bon ordre. En effet, cette boucle remonte de `v` jusqu'à la source et on aura bien ainsi le début du chemin, côté source, au début de la liste. On voit ici l'intérêt d'avoir associé la source à `null` dans `pred`, pour sortir de la boucle.

## Exercice 102, page 186

Comme pour un graphe, on va utiliser une file pour écrire le parcours en largeur d'un arbre. Cette file va contenir des arbres, plus précisément des sous-arbres de l'arbre initial. Dans le cas d'un arbre, on sait qu'il ne peut y avoir de cycle. Il n'est donc pas nécessaire d'utiliser de table `visited`. Enfin, tout nœud non nul a exactement deux successeurs. Toutes ces remarques amènent à un code très simple :

```
static void BFS(Tree t) {
  Queue<Tree> q = new LinkedList<>();
  q.add(t);
  while (!q.isEmpty()) {
    Tree s = q.poll();
    if (s == null) continue;
    // traiter s.value, par exemple l'afficher
    q.add(s.left);
```

```
      q.add(s.right);
    }
  }
```

Une (petite) amélioration consiste à éviter de mettre des valeurs `null` dans la file. Cela ne change pas la complexité, bien entendu.

## Exercice 103, page 188

C'est la même chose que la solution de l'exercice 101 pour le parcours en largeur. On commence par ajouter la table `pred` comme un nouveau champ.

```
private final HashMap<V, V> pred;
```

Pour effectuer une modification minimale à la méthode `dfs`, le plus simple est de lui ajouter le sommet dont on vient comme un second argument `from`. On peut alors remplir la table `pred` lorsque le sommet est atteint pour la première fois.

```
void dfs(V from, V v) {
  if (this.visited.containsKey(v)) return;
  this.visited.put(v, this.count++);
  this.pred.put(v, from);
  for (V w : this.g.successors(v))
    dfs(v, w);
}
```

Pour retrouver la méthode `dfs` d'origine, il suffit de prendre `null` comme valeur de `from`. Ainsi, comme pour le parcours en largeur, les sommets atteints par le parcours en profondeur seront exactement les clés de `pred`, y compris la source.

```
void dfs(V v) {
  dfs(null, v);
}
```

Pour ce qui est de la reconstruction du chemin, voir la solution de l'exercice 101.

## Exercice 104, page 190

Le parcours d'un arbre est un parcours de graphe sans cycle et où chaque sommet a toujours zéro ou deux successeurs. Du coup, le code de la méthode `dfs` se simplifie en

```
void dfs(Tree t) {
  if (t == null) return;
  dfs(t.left);
  dfs(t.right);
```

C'est donc bien la même chose qu'un parcours infixe/préfixe/postfixe (selon le moment où on choisit de traiter la racine de `t`).

## Exercice 105, page 190

Le code ressemble effectivement à celui du parcours en largeur. Mais si on veut que le parcours se poursuive tant que des arcs peuvent être empruntés, il ne faut pas ajouter les sommets dans `visited` au moment où ils sont ajoutés à la pile, mais seulement au moment où ils en sont retirés. On obtient le code suivant :

```
void dfs(V v) {
  Stack<V> stack = new Stack<V>();
  stack.add(v);
  while (!stack.isEmpty()) {
    v = stack.pop();
    if (this.visited.containsKey(v)) continue;
    this.visited.put(v, this.count++);
    for (V w : this.g.successors(v))
      stack.add(w);
  }
}
```

On n'obtient pas nécessairement le même ordre de parcours que dans la version récursive. En effet, les successeurs de `v` étant mis sur une pile, ils vont être considérés exactement dans l'ordre inverse. Mais cela reste un parcours en profondeur. Dit autrement, tout se passe comme si la méthode `g.successors` nous donnait les successeurs dans un ordre différent.

## Exercice 106, page 190

Ajoutons un nouveau champ comme suggéré dans l'énoncé.

```
private final LinkedList<V> list;
```

Modifions la méthode `dfs` pour ajouter le sommet `v` à cette liste lorsqu'il est atteint pour la première fois. L'idée est de l'ajouter avec la méthode `addFirst`, c'est-à-dire en tête de liste, une fois que l'appel à `dfs(v)` est terminé.

```
void dfs(V v) {
  if (this.visited.contains(v)) return;
  this.visited.add(v);
  for (V w : this.g.successors(v))
    dfs(w);
  this.list.addFirst(v);
}
```

Ainsi, tout sommet descendant de `v` est bien ajouté à la liste avant `v`, soit antérieurement à cet appel à `dfs(v)`, soit par l'un des appels récursifs `dfs(w)`. La méthode `topologicalSort` assure que `dfs` a bien été appelé sur tous les sommets du graphe avant de renvoyer la liste.

```
List<V> topologicalSort() {
  for (V v : this.g.vertices())
```

```
      dfs(v);
    return list;
  }
```

Ce qui est assez subtil ici, c'est que l'ordre dans lequel tous ces appels à `dfs` sont réalisés n'importe pas.

## Exercice 107, page 190

On commence par introduire une type pour les trois états possibles d'un sommet. La couleur `White` désigne un sommet non encore atteint, la couleur `Gray` un sommet en cours de visite et la couleur `Black` un sommet dont la visite est terminée.

```
enum Color { White, Gray, Black};
```

La table de hachage `visited` est remplacée par une table donnant la couleur de chaque sommet.

```
private final HashMap<V, Color> color;
```

Prenons alors la convention que la méthode `dfs` renvoie `true` dès qu'elle découvre un cycle. Le code de `dfs` commencer par tester la couleur du sommet `v`. La couleur `Gray` signifie qu'un cycle vient d'être découvert. La couleur `Black` signifie qu'il n'y a rien à faire ; c'est l'analogue du `return` dans le code initial de `dfs`. Sinon, on positionne la couleur à `Gray` et on appelle `dfs` récursivement sur les successeurs. Si l'un de ces appels détecte un cycle, on renvoie immédiatement `true`. On termine en positionnant la couleur à `Black`.

```
boolean dfs(V v) {
  if (this.color.get(v) == Color.Gray) return true;
  if (this.color.get(v) == Color.Black) return false;
  this.color.put(v, Color.Gray);
  for (V w : this.g.successors(v))
    if (dfs(w)) return true;
  this.color.put(v, Color.Black);
  return false;
}
```

La méthode demandée commence par mettre la couleur de chaque sommet à `White` puis appelle `dfs` sur tous les sommets. Là encore, on s'interrompt dès qu'un cycle est découvert.

```
boolean hasCycle() {
  for (V v : this.g.vertices())
    color.put(v, Color.White);
  for (V v : this.g.vertices())
    if (dfs(v)) return true;
  return false;
}
```

On aurait pu aussi prendre la convention que ne pas être dans la table de hachage est identique à avoir la couleur `White`. Du coup, deux valeurs suffisent.

Dans le cas particulier d'une liste chaînée, c'est plus coûteux que l'algorithme du lièvre et de la tortue, car on stocke ici une quantité d'information proportionnelle aux nombre de sommets rencontrés.

## Exercice 108, page 190

On commence par construire le graphe correspondant à la grille. Il y a plusieurs façons de le faire. On peut par exemple adopter des sommets qui sont des entiers, en adoptant le codage $im + j$ pour le sommet $(i, j)$ de la grille $n \times m$ (comme dans l'exercice 62 page 122) et ainsi pouvoir utiliser une matrice d'adjacence, ou encore se donner une classe de la forme

```
class Cell { final int i, j; ... }
```

que l'on équipe d'un constructeur adéquat et de méthodes `hashCode` et `equals` afin de pouvoir construire un graphe par listes d'adjacence. En supposant qu'on opte pour la seconde solution, on se donne une classe qui va représenter le labyrinthe comme un graphe :

```
class Maze extends Graph<Cell> {
  private Graph<Cell> grid;
  Maze(int n, int m) {
    // ajouter tous les sommets à this et grid
    // et tous les arcs de la grille à grid
    ...
  }
```

(Ici, la classe `Graph` est celle du programme 37 page 179.) Il y a donc deux graphes en jeu : le graphe représentant la grille complète, dans le champ `grid`, et le graphe représentant le labyrinthe que l'on construit, dans l'objet `this`.

On se donne ensuite un ensemble pour marquer les sommets visités par le parcours en profondeur.

```
  private HashSet<Cell> visited = new HashSet<>();
```

Le parcours en profondeur est réalisé par une méthode récursive `dfs`, avec un argument supplémentaire `from` indiquant le sommet qui a permis d'atteindre le sommet `v` (comme dans l'exercice 103 page 190).

```
  void dfs(Cell from, Cell v) {
    if (this.visited.contains(v)) return;
    this.visited.add(v);
    if (from != null) { this.addEdge(from, v); this.addEdge(v, from); }
    for (Cell w: shuffleSuccessors(v))
      dfs(v, w);
  }
```

Lorsqu'un arc est emprunté par le parcours, alors il est ajouté au labyrinthe. (Comme `from` vaudra `null` initialement, on prend soin de tester sa valeur.) Les voisins de `v` sont parcourus dans un ordre aléatoire grâce à la méthode `shuffleSuccessors` qui mélange le résultat renvoyé par la méthode `successors` (par exemple en utilisant l'exercice 5 page 35).

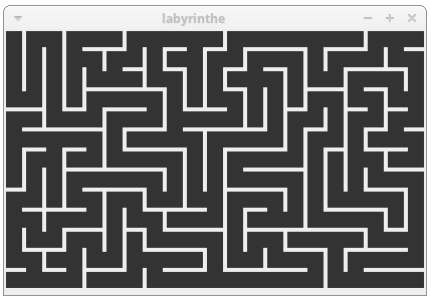
```
  private Vector<Cell> shuffleSuccessors(Cell v) {
    Vector<Cell> s = new Vector<>();
    for (Cell w: this.grid.successors(v))
      s.add(w);
```

```
    KnuthShuffle.shuffle(s);
    return s;
  }
```

Il ne reste plus qu'à lancer le parcours, par exemple à partir du sommet $(0, 0)$.

```
dfs(null, new Cell(0, 0));
```

On peut le faire à la toute fin du constructeur de la classe `Maze`. Voici un résultat possible sur une grille de dimension $21 \times 13$.




Note : Cet algorithme donne toutefois des labyrinthes « moins bons » que ceux donnés par l'exercice 62 page 122, parce qu'ils sont trop linéaires.

## Exercice 109, page 196

Voir la solution des exercices 101 et 103.

## Exercice 110, page 200

On commence par vérifier qu'il y a exactement $V - 1$ arcs dans la liste `t`. Ensuite, comme suggéré, on se sert d'une structure *union-find* pour tester la présence d'un cycle.

```
static boolean isSpanningTree(Kgraph g, List<Kedge> t) {
  if (t.size() != g.V - 1) return false;
  UnionFind uf = new UnionFind(g.V);
  for (Kedge e: t) {
    if (uf.find(e.src) == uf.find(e.dst)) return false;
    uf.union(e.src, e.dst);
  }
  return true;
}
```

On peut supposer que la complexité de la méthode `size` de `t` est $O(1)$ (c'est le cas notamment si `t` est réalisée par la classe `LinkedList`). La construction de la structure *union-find* est en $O(V)$. Ensuite, on fait une boucle sur tous les arcs de `t`, au nombre de $V - 1$, et pour chacun on exécute un code que l'on peut considérer être en $O(1)$ amorti (deux `find` et un `union` ; voir chapitre 9). D'où une complexité totale en $O(V)$.

# B

# Bref comparatif Java/Python

Il n'est pas inintéressant de comparer les langages Java et Python. Dans ce qui suit, nous faisons une énumération, non exhaustive, des différences mais aussi des ressemblances entre Java et Python.

Si les langages de surface de Java et Python sont très différents, leurs modèles d'exécution sont en revanche très proches. On a décrit celui de Java dans la section 1.2. On y a expliqué notamment que le mode de passage de Java est *par valeur* et que les valeurs sont soit des valeurs primitives (entiers, caractères, booléens, etc.) soit des adresses vers des objets alloués sur le tas. Dans le cas de Python, on a également du passage par valeur uniquement et les valeurs sont *toutes* des adresses vers des objets alloués sur le tas. La différence est minime et il est facile de transposer sa représentation du modèle d'exécution de l'un à l'autre.

**Typage statique et dynamique.** Une différence fondamentale entre les langages Java et Python est que le premier est typé *statiquement* là où le second ne l'est que *dynamiquement*. Ainsi, le langage Python nous permet d'écrire une fonction comme

```
def f(x):
  return "a" * x
```

sans préciser le type du paramètre `x`. Plus tard, on peut l'appeler avec un entier, ce qui va fonctionner, ou encore avec une chaîne de caractères, ce qui va échouer. Dans ce second cas, cet échec aura lieu *pendant l'exécution* du programme. Même si le source du programme Python contient `f("b")`, le programme n'est pas rejeté pour autant. Et si son exécution ne conduit jamais à évaluer l'expression `f("b")`, alors le fait que multiplier une chaîne par une autre ne fait aucun sens ne sera jamais signalé.

Dans le cas de Java, en revanche, on doit donner un type au paramètre `x` et le compilateur Java va vérifier, *pendant la compilation*, que l'utilisation de la variable `x` est légitime. Dit autrement, un grand nombre d'erreurs sont découvertes en Java pendant la compilation, plutôt que pendant l'exécution. Bien sûr, il reste encore des erreurs à l'exécution en Java, comme par exemple `NullPointerException`.

Une conséquence du typage dynamique de Python est que les structures de données sont naturellement *hétérogènes*, c'est-à-dire qu'elles peuvent contenir des valeurs de types différents. Ainsi, un tableau Python peut contenir simultanément des éléments de différents types, un dictionnaire Python peut contenir des clés (et des valeurs) de différents

types, etc. En Java, en revanche, le type des éléments est fixé et le compilateur vérifie que les éléments ajoutés ont bien le type attendu.

**Égalité.** En Java, l'opération `==` représente l'*égalité physique* c'est-à-dire la comparaison immédiate des valeurs primitives comme des entiers et sinon la comparaison des *adresses*. (Voir page 20.) En Python, cela correspond à l'opération `is`. L'opération `==` de Python, en revanche, est une comparaison *structurelle*, c'est-à-dire en profondeur. Ainsi, si deux tableaux Python `a` et `b` différents ont la même taille et contiennent les mêmes valeurs, alors l'expression `a==b` va renvoyer vrai. En Java, il faut programmer soi-même une telle comparaison structurelle, typiquement dans une méthode `equals`.

**Entiers.** En Java, le type `int` représente des entiers signés 32 bits, avec une capacité qui est donc limitée et de possibles débordements arithmétiques. En Python, en revanche, les entiers sont de *précision arbitraire* et les opérations arithmétiques ne provoquent donc jamais de débordement. Ceci est rendu possible en allouant les entiers sur le tas, comme des objets. Il s'agit là d'objets *immuables*, ce qui fait qu'on peut conserver la même abstraction qu'en Java en imaginant les entiers de Python comme des valeurs primitives. Si on veut des entiers de précision arbitraire en Java, il faut utiliser la bibliothèque `java.math.BigInteger`.

Attention : Java et Python n'implémentent pas la même division en présence d'opérandes négatives. Le modulo de Java a le signe de la première opérande, celui de Python le signe de la seconde opérande.

**Chaînes de caractères.** Les chaînes de caractères ne sont pas nécessairement représentées en interne de la même façon dans les deux langages. (Ainsi, pendant longtemps, Java utilisait exclusivement UTF-16 comme encodage de l'Unicode.) Mais dans les deux cas, les chaînes sont des objets *immuables*.

**Tableaux.** Le langage Python fournit nativement une structure appelée *liste*, qui cache en réalité un *tableau redimensionnable*, exactement au sens de la section 3.4 page 39, et donc comparables aux classes `ArrayList` et `Vector` de la bibliothèque Java. Le vocabulaire de "listes" est assez malheureux.

On peut en particulier utiliser les "listes" de Python comme des tableaux, sans jamais les redimensionner. La syntaxe est comparable à celle de Java :

```java
int[] a = {1, 2, 3};
a[1] = 4;
System.out.println(a[1]);
int n = a.length;
for (int x: a)
  ...
```

```python
a = [1, 2, 3]
a[1] = 4
print(a[1])
n = len(a)
for x in a:
  ...
```

À la différence de Java, cependant, les "listes" de Python sont hétérogènes, c'est-à-dire qu'elles peuvent contenir des valeurs de types différents.

**Pile, file, file de priorité.** Les "listes" de Python peuvent être étendues ou raccourcies du côté droit, en particulier avec les méthodes `append` et `pop`. Avec ces deux opérations, on a notamment une structure de *pile*, exactement comme nous l'avons fait dans la section 3.4.4 page 44. Pour les structures de file et de file de priorité, il existe des classes `Queue` et `PriorityQueue` dans la bibliothèque `queue` de Python.

**Listes chaînées.** L'équivalent Python de la bibliothèque `LinkedList` est la structure `deque` de la bibliothèque `collections`. Comme avec `LinkedList`, on peut opérer efficacement aux deux extrémités de la liste pour ajouter ou supprimer des éléments. On peut en particulier s'en servir comme une file, en alternative à la classe `Queue` décrite ci-dessus.

**Tables de hachage.** Python fournit nativement des ensembles et des dictionnaires implémentés par des tables de hachage, analogues de `HashSet` et `HashMap` en Java, toutefois avec une syntaxe plus concise.

```
HashMap<K,V> h = new HashMap<>();
h.put(k, v);
if (h.containsKey(k))
  v = h.get(k);
```

```
h = {}
h[k] = v
if k in h:
  v = h[k]
```

Il y a cependant une différence importante : les tables de hachage de Python utilisent *par défaut une égalité structurelle* — qui coïncide avec l'opération `==` de Python — là où les tables de hachage de Java vont utiliser par défaut une égalité physique. Par conséquent, il est courant de devoir redéfinir en Java les méthodes `hashCode` et `equals` des classes utilisées comme éléments ou comme clés (comme expliqué section 5.3 page 72), là où Python offre le plus souvent le comportement attendu par défaut. Il reste toutefois possible de redéfinir les opérations d'égalité et de hachage sur ses propres classes en Python.

# C Lexique Français-Anglais

| français | anglais | voir pages |
|---|---|---|
| affectation | assignment | |
| arbre | tree | 75 |
| arbre binaire | binary tree | 75 |
| arbre de préfixes | trie | 92 |
| arc | (directed) edge | 175 |
| arête | edge | 175 |
| champ | field | 3 |
| chemin | path | 175 |
| corde | rope | 99 |
| degré entrant | indegree | 175 |
| degré sortant | outdegree | 175 |
| feuille | leaf | 75 |
| file | queue (FIFO) | 54 |
| file de priorité | priority queue | 105 |
| flottant | floating-point number | 14 |
| graphe | graph | 175 |
| graphe (non) orienté | (un)directed graph | 175 |
| graphe orienté acyclique | directed acyclic graph (DAG) | 190 |
| graphe étiqueté | labeled graph | 176 |
| immuable (structure de données) | immutable (data structure) | 97 |
| infixe (parcours) | inorder traversal | 77 |
| liste | liste | 49 |
| liste d'adjacence | adjacency list | 178 |
| matrice d'adjacence | adjacency matrix | 176 |
| mémoïsation | memoization | 131 |
| méthode | method | 4 |
| parcours en largeur | breadth-first search (BFS) | 184 |
| parcours en profondeur | depth-first search (DFS) | 186 |
| pgcd | gcd | 125 |
| pile | stack (LIFO) | 44, 54 |
| plus court chemin | shortest path | 191 |

| français | anglais | voir pages |
|---|---|---|
| programmation dynamique | dynamic programming (DP) | 133 |
| racine | root | 75 |
| rebroussement | backtracking | 139 |
| redéfinition | overriding | 8 |
| seau | bucket | 68 |
| sommet | vertex | 175 |
| surcharge | overloading | 6 |
| table de hachage | hash table | 67 |
| tableau | array | 33 |
| tableau redimensionnable | resizable array | 39 |
| tas | heap | 105 |
| transtypage | cast | 47 |
| tri | sorting | 149 |
| tri en place | in-place sort | |
| tri fusion | mergesort | 154 |
| tri par insertion | insertion sort | 149 |
| tri par tas | heapsort | 158 |
| tri rapide | quicksort | 150 |
| tri topologique | topological sort | 190 |

# Bibliographie


[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5) :1259–1263, September 1962.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[3] Joshua Bloch. Nearly all binary searches and mergesorts are broken, 2006. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.

[4] Sylvain Conchon and Jean-Christophe Filliâtre. *Apprendre à programmer avec OCaml. Algorithmes et structures de données*. Eyrolles, September 2014.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1) :269–271, December 1959.

[7] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics, a Foundation for Computer Science*. Addison-Wesley, 1989.

[8] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9) :1098–1101, September 1952.

[9] IEEE standard for floating-point arithmetic, 2008. https://dx.doi.org/10.1109/IEEESTD.2008.4610935.

[10] J. Kleinberg and E. Tardos. *Algorithm design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2005.

[11] Donald E. Knuth. *The Art of Computer Programming, volume 2 (3rd ed.) : Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[12] Donald E. Knuth. *The Art of Computer Programming, volume 3 : (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[13] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, 1956.

[14] Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4) :514–534, 1968.

[15] Robert Sedgewick and Kevin Wayne. *Introduction to Programming in Java.* Addison Wesley, 2008.

[16] Robert Endre Tarjan. *Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM*, 22(2) :215–225, 1975.

[17] Henry S. Warren. *Hacker's Delight.* Addison-Wesley, 2003.

# Index