Introduction à l'informatique

École polytechnique

François Morain

11 avril 2011

Table des matières

Introduction

Les audacieux font fortune à Java.

Ce polycopié s'adresse à des élèves de première année ayant peu ou pas de connaissances en informatique. Une partie de ce cours constitue une introduction générale à l'informatique, aux logiciels, matériels, environnements informatiques et à la science sous-jacente.

Une autre partie consiste à établir les bases de la programmation et de l'algorithmique, en étudiant un langage. On introduit des structures de données simples : scalaires, chaînes de caractères, tableaux, et des structures de contrôle élémentaires comme l'itération, la récursivité.

Nous avons choisi JAVA pour cette introduction à la programmation car c'est un langage typé assez répandu qui permet de s'initier aux diverses constructions présentes dans la plupart des langages de programmation modernes. Il est présent de plus en plus dans les téléphones portables (avec Android).

À ces cours sont couplés des séances de travaux dirigés et pratiques qui sont beaucoup plus qu'un complément au cours, puisque c'est en écrivant des programmes que l'on apprend l'informatique.

Comment lire ce polycopié ? La première partie décrit les principaux traits d'un langage de programmation (ici JAVA), ainsi que les principes généraux de la programmation simple. Une deuxième partie présente quelques grandes classes de problèmes que les ordinateurs traitent plutôt bien.

Un passage indiqué par une étoile (*) peut être sauté en première lecture.

Remerciements pour la version 3.0 :

P. Chassignet m'a aidé a rédiger le nouveau chapitre 14, qu'il en soit sincèrement remercié. Merci à É. Duris pour sa relecture attentive de la partie génie logicielle (nouvelle elle aussi). Enfin, rien n'est possible sans des relecteurs : L. Castelli Aleardi, Y. Ponty, S. Redon, O. Serre.

Remerciements pour les versions 2001–2010 :

G. Guillerm m'a aidé pour le chapitre INTERNET, J. Marchand pour le courrier électronique, T. Besançon pour NFS. Qu'ils en soient remercié ici, ainsi que E. Thomé pour ses coups de main, V. Ménissier-Morain pour son aide. Je remercie également les relecteurs de la présente version : T. Clausen, É. Duris, C. Gwiggner, J.-R. Reinhard ; E. Waller, ce dernier étant également le béta-testeur de mes transparents d'amphis, ce qui les a rendus d'autant plus justes.

Remerciements pour la version initiale (2000) : Je remercie chaleureusement Jean-Jacques Lévy et Robert Cori pour m'avoir permis de réutiliser des parties de leurs polycopiés anciens ou nouveaux.

Le polycopié a été écrit avec LaTeX, il est consultable à l'adresse :
http://www.enseignement.polytechnique.fr/informatique/INF311

Les erreurs seront corrigées dès qu'elles me seront signalées et les mises à jour éventuelles seront effectuées sur la version mise sur le web.

Polycopié, version 3.0, 11 avril 2011

Première partie

Introduction à la programmation

Chapitre 1

Les premiers pas en JAVA

Dans ce chapitre on donne quelques éléments simples de la programmation avec le langage JAVA : types, variables, affectations, fonctions. Ce sont des traits communs à tous les langages de programmation.

1.1 Le premier programme

1.1.1 Écriture et exécution

Commençons par un exemple simple de programme. C'est un classique, il s'agit simplement d'afficher `Bonjour !` à l'écran.

```
// Voici mon premier programme
public class Premier{
    public static void main(String[] args){
        System.out.println("Bonjour !");
        return;
    }
}
```

Pour exécuter ce programme il faut commencer par le copier dans un fichier. Pour cela on utilise un éditeur de texte (par exemple `nedit`) pour créer un fichier de nom `Premier.java` (le même nom que celui qui suit **`class`**). Ce fichier doit contenir le texte du programme. Après avoir tapé le texte, on doit le traduire (les informaticiens disent *compiler*) dans un langage que comprend l'ordinateur. Cette compilation se fait à l'aide de la commande[1]

```
unix% javac Premier.java
```

Ceci a pour effet de faire construire par le compilateur un fichier `Premier.class`, que la machine virtuelle de JAVA rendra compréhensible pour l'ordinateur :

```
unix% java Premier
```

[1] Une ligne commençant par `unix%` indique une commande tapée en Unix.

On voit s'afficher :

```
Bonjour !
```

1.1.2 Analyse de ce programme

Un langage de programmation est comme un langage humain. Il y a un ensemble de lettres avec lesquelles on forme des mots. Les mots forment des phrases, les phrases des paragraphes, ceux-ci forment des chapitres qui rassemblés donnent naissance à un livre. L'alphabet de JAVA est peu ou prou l'alphabet que nous connaissons, avec des lettres, des chiffres, quelques signes de ponctuation. Les mots seront les *mots-clefs* du langage (comme **`class`**, **`public`**, etc.), ou formeront les noms des *variables* que nous utiliserons plus loin. Les phrases seront pour nous des *instructions*, les paragraphes des *fonctions* (appelées *méthodes* dans la terminologie des langages à objets). Les chapitres seront les *classes*, les livres des programmes que nous pourrons exécuter et utiliser.

Le premier chapitre d'un livre est l'amorce du livre et ne peut généralement être sauté. En JAVA, un programme débute toujours à partir d'une méthode spéciale, appelée `main` et dont la syntaxe immuable est :

```
public static void main(String[] args)
```

Nous verrons plus loin ce que veulent dire les mots magiques **`public`**, **`static`** et **`void`**, `args` contient quant à lui des arguments qu'on peut passer au programme. Reprenons la méthode `main` :

```
public static void main(String[] args){
    System.out.println("Bonjour !");
    return;
}
```

Les accolades { et } servent à constituer un bloc d'instructions ; elles doivent englober les instructions d'une méthode, de même qu'une paire d'accolades doit englober l'ensemble des méthodes d'une classe.

Notons qu'en JAVA les instructions se terminent toutes par un `;` (point-virgule). Ainsi, dans la suite le symbole `I` signifiera soit une instruction (qui se termine donc par `;`) soit une suite d'instructions (chacune finissant par `;`) placées entre accolades.

La méthode effectuant le travail est la méthode `System.out.println` qui appartient à une classe prédéfinie, la classe `System`. En JAVA, les classes peuvent s'appeler les unes les autres, ce qui permet une approche modulaire de la programmation : on n'a pas à récrire tout le temps la même chose.

Notons que nous avons écrit les instructions de chaque ligne en respectant un décalage bien précis (on parle d'*indentation*). La méthode `System.out.println` étant exécutée à l'intérieur de la méthode `main`, elle est décalée de plusieurs blancs (ici 4) sur la droite. L'indentation permet de bien structurer ses programmes, elle est systématiquement utilisée partout.

La dernière instruction présente dans la méthode `main` est l'instruction **`return;`** que nous comprendrons pour le moment comme voulant dire : rendons la main à l'utilisateur qui nous a lancé. Nous en préciserons le sens à la section 1.7.

La dernière chose à dire sur ce petit programme est qu'il contient un commentaire, repéré par `//` et se terminant à la fin de la ligne. Les commentaires ne sont utiles

qu'à des lecteurs (humains) du texte du programme, ils n'auront aucun effet sur la compilation ou l'exécution. Ils sont très utiles pour comprendre le programme.

1.2 Faire des calculs simples

On peut se servir de JAVA pour réaliser les opérations d'une calculette élémentaire : on affecte la valeur d'une expression à une variable et on demande ensuite l'affichage de la valeur de la variable en question. Bien entendu, un langage de programmation n'est pas fait uniquement pour cela, toutefois cela nous donne quelques exemples de programmes simples ; nous passerons plus tard à des programmes plus complexes.

```
// Voici mon deuxième programme
public class PremierCalcul{
    public static void main(String[] args){
        int a;

        a = 5 * 3;
        System.out.println(a);
        a = 287 % 7;
        System.out.println(a);
        return;
    }
}
```

Dans ce programme on voit apparaître une variable de nom `a` qui est déclarée au début. Comme les valeurs qu'elle prend sont des entiers elle est dite de *type* entier. Le mot `int`[2] qui précède le nom de la variable est une déclaration de type. Il indique que la variable est de type entier et ne prendra donc que des valeurs entières lors de l'exécution du programme. Par la suite, on lui affecte deux fois une valeur qui est ensuite affichée. Les résultats affichés seront 15 et 0. Dans l'opération `a % b`, le symbole `%` désigne l'opération *modulo* qui est le reste de la division euclidienne de a par b (quand a et b sont positifs).

Insistons un peu sur la façon dont le programme est exécuté par l'ordinateur. Celui-ci lit les instructions du programme une à une en commençant par la méthode `main`, et les traite dans l'ordre où elles apparaissent. Il s'arrête dès qu'il rencontre l'instruction **`return;`**, qui est généralement la dernière présente dans une méthode. Nous reviendrons sur le mode de traitement des instructions quand nous introduirons de nouvelles constructions (itération, récursion).

1.3 Types primitifs

Un *type* en programmation précise l'ensemble des valeurs que peut prendre une variable ; les opérations que l'on peut effectuer sur une variable dépendent de son type.

Le type des variables que l'on utilise dans un programme JAVA doit être déclaré. Parmi les types possibles, les plus simples sont les types primitifs. Il y a peu de types primitifs : les entiers, les réels, les caractères et les booléens.

[2] Une abréviation de l'anglais *integer*, le *g* étant prononcé comme un *j* français.

Les principaux types entiers sont `int` et `long`, le premier utilise 32 bits (un bit vaut 0 ou 1, c'est un chiffre binaire) pour représenter un nombre ; sachant que le premier bit est réservé au signe, un `int` fait référence à un entier de l'intervalle $[-2^{31}, 2^{31}-1]$. Si lors d'un calcul, un nombre dépasse cette valeur le résulat obtenu n'est pas utilisable. Le type `long` permet d'avoir des mots de 64 bits (entiers de l'intervalle $[-2^{63}, 2^{63}-1]$) et on peut donc travailler sur des entiers plus grands. Il y a aussi les types `byte` et `short` qui permettent d'utiliser des mots de 8 et 16 bits. Les opérations sur les `int` sont toutes les opérations arithmétiques classiques : les opérations de comparaison : égal (`==`), différent de (`!=`), plus petit (`<`), plus grand (`>`) et les opérations de calcul comme addition (`+`), soustraction (`-`), multiplication (`*`), division (`/`), reste (`%`). Dans ce dernier cas, précisons que `a/b` calcule le quotient de la division euclidienne de `a` par `b` et que `a % b` en calcule le reste. Par suite

```
int q = 2/3;
```

contient le quotient de la division euclidienne de 2 par 3, c'est-à-dire 0.

Les types réels (en fait, des nombres dont le développement binaire est fini) sont `float` et `double`, le premier se contente d'une précision dite simple, le second donne la possibilité d'une plus grande précision, on dit que l'on a une double précision.

Les caractères sont déclarés par le type `char` au standard Unicode. Ils sont codés sur 16 bits et permettent de représenter toutes les langues de la planète (les caractères habituels des claviers des langues européennes se codent uniquement sur 8 bits). Le standard Unicode respecte l'ordre alphabétique. Ainsi le code de 'a' est inférieur à celui de 'd', et celui de 'A' à celui de 'D'.

Le type des booléens est `boolean` et ses deux valeurs possibles sont `true` et `false`. Les opérations sont **et**, **ou**, et **non** ; elles se notent respectivement `&&, ||, !`. Si `a` et `b` sont deux booléens, le résultat de `a && b` est `true` si et seulement si `a` et `b` sont tous deux égaux à `true`. Celui de `a || b` est `true` si et seulement si l'un de `a` ou `b` est égal à `true`. Enfin `!a` est `true` quand `a` est `false` et réciproquement. Les booléens sont utilisés dans les conditions décrites au chapitre suivant.

1.4 Déclaration des variables

La déclaration du type des variables est obligatoire en JAVA, mais elle peut se faire à l'intérieur d'une méthode et pas nécessairement au début de celle-ci. Une déclaration se présente sous la forme d'un nom de type suivi soit d'un nom de variable, soit d'une suite de noms de variables séparés par des virgules. En voici quelques exemples :

```
int a, b, c;
float x;
char ch;
boolean u, v;
```

1.5 Affectation

On a vu qu'une variable a un nom et un type. L'opération la plus courante sur les variables est l'affectation d'une valeur. Elle s'écrit :

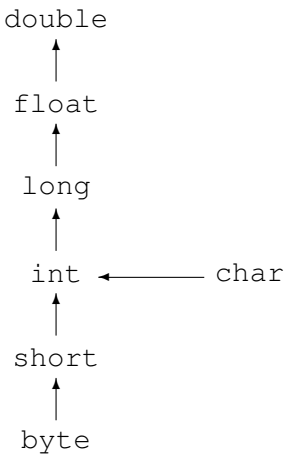


FIG. 1.1 – Coercions implicites.

```
x = E;
```

où `E` est une expression qui peut contenir des constantes et des variables. Lors d'une affectation, l'expression `E` est évaluée et le résultat de son évaluation est affecté à la variable `x`. Lorsque l'expression `E` contient des variables leur contenu est égal à la dernière valeur qui leur a été affectée.

Par exemple, l'affectation

```
x = x + a;
```

consiste à augmenter la valeur de `x` de la quantité `a`. Par exemple

```
x = 12;
x = x + 5;
```

donnera à `x` la valeur 17.

Pour une affectation

```
x = E;
```

le type de l'expression `E` et celui de la variable `x` doivent être compatibles. Dans un très petit nombre de cas cette exigence n'est pas appliquée, il s'agit alors des conversions implicites de types. Les conversions implicites suivent la figure 1.1. Pour toute opération, on convertit toujours au plus petit commun majorant des types des opérandes. Des conversions explicites sont aussi possibles, et recommandées dans le doute. On peut les faire par l'opération dite de coercion (*cast*) suivante

```
x = (nom-type) E;
```

L'expression E est alors convertie dans le type indiqué entre parenthèses devant l'expression. L'opérateur = d'affectation est un opérateur comme les autres dans les expressions. Il subit donc les mêmes lois de conversion. Toutefois, il se distingue des autres opérations par le type du résultat. Pour un opérateur ordinaire, le type du résultat est le type commun obtenu par conversion des deux opérandes. Pour une affectation, le type du résultat est le type de l'expression à gauche de l'affectation. Il faut donc faire une conversion explicite sur l'expression de droite pour que le résultat soit cohérent avec le type de l'expression de gauche. C'est le cas de l'exemple

```
int x = (int)12.7;
```

où le flottant `12.7` sera converti en entier, soit 12.

1.6 Opérations

La plupart des opérations arithmétiques courantes sont disponibles en JAVA, ainsi que les opérations sur les booléens (voir chapitre suivant). Ces opérations ont un ordre de priorité correspondant aux conventions usuelles.

1.6.1 Règles d'évaluation

Les principales opérations sont `+,-,*,/, %` pour l'addition, soustraction, multiplication, division et le reste de la division (modulo). Il y a des règles de priorité, ainsi l'opération de multiplication a une plus grande priorité que l'addition, cela signifie que les multiplications sont faites avant les additions. La présence de parenthèses permet de mieux contrôler le résultat. Par exemple `3 + 5 * 6` est évalué à 33 ; par contre `(3 + 5) * 6` est évalué à 48. Une expression a toujours un type et le résultat de son évaluation est une valeur ayant ce type.

On utilise souvent des raccourcis pour les instructions du type

```
x = x + a;
```

qu'on a tendance à écrire de façon équivalente, mais plus compacte :

```
x += a;
```

1.6.2 Incrémentation et décrémentation

Soit `i` une variable de type **`int`**. On peut l'*incrémenter*, c'est-à-dire lui additionner 1 à l'aide de l'instruction :

```
i = i + 1;
```

C'est une instruction tellement fréquente (particulièrement dans l'écriture des boucles, cf. chapitre suivant), qu'il existe deux raccourcis : `i++` et `++i`. Dans le premier cas, il s'agit d'une *post-incrémentation*, dans le second d'une *pré-incrémentation*. Expliquons la différence entre les deux. Le code

```
i = 2;
j = i++;
```

donne la valeur 3 à `i` et 2 à `j`, car le code est équivalent à :

```
i = 2;
j = i;
i = i + 1;
```

on incrémente en tout dernier lieu. Par contre :

```
i = 2;
j = ++i;
```

est équivalent quant à lui à :

```
i = 2;
i = i + 1;
j = i;
```

et donc on termine avec `i=3` et `j=3`.

Il existe aussi des raccourcis pour la décrémentation : `i = i-1` peut s'écrire aussi `i--` ou `--i` avec les mêmes règles que pour `++`.

1.7 Méthodes

Le programme suivant, qui calcule la circonférence d'un cercle en méthode de son rayon, contient deux méthodes, `main`, que nous avons déjà rencontrée, ainsi qu'une nouvelle méthode, `circonference`, qui prend en argument un réel `r` et retourne la valeur de la circonférence, qui est aussi un réel :

```
// Calcul de circonférence
public class Cercle{
    static float pi = (float)Math.PI;

    public static float circonference(float r){
        return 2. * pi * r;
    }

    public static void main(String[] args){
        float c = circonference(1.5);

        System.out.print("Circonférence:  ");
        System.out.println(c);
        return;
    }
}
```

De façon générale, une méthode peut avoir plusieurs arguments, qui peuvent être de types différents et retourne une valeur (dont le type doit être aussi précisé). Certaines méthodes ne retournent aucune valeur. Elles sont alors déclarées de type `void`. C'est le cas particulier de la méthode `main` de notre exemple. Pour bien indiquer dans ce cas le point où la méthode renvoie la main à l'appelant, nous utiliserons souvent un **`return;`** explicite, qui est en fait optionnel. Il y a aussi des cas où il n'y a pas de paramètres lorsque la méthode effectue toujours les mêmes opérations sur les mêmes valeurs.

L'en-tête d'une méthode décrit le type du résultat d'abord puis les types des paramètres qui figurent entre parenthèses.

Les programmes que l'on a vu ci-dessus contiennent une seule méthode appelée `main`. Lorsqu'on effectue la commande `java Nom-classe`, c'est la méthode `main` se trouvant dans cette classe qui est exécutée en premier.

Une méthode peut appeler une autre méthode ou être appelée par une autre méthode, il faut alors donner des arguments aux paramètres d'appel.

Ce programme contient deux méthodes dans une même classe, la première méthode a un paramètre `r` et utilise la constante `PI` qui se trouve dans la classe `Math`, cette constante est de type `double` il faut donc la convertir au type `float` pour affecter sa valeur à un nombre de ce type.

Le résultat est fourni, on dit plutôt *retourné* à l'appelant par `return`. L'appelant est ici la méthode `main` qui après avoir effectué l'appel, affiche le résultat.

Chapitre 2

Suite d'instructions

Dans ce chapitre on s'intéresse à deux types d'instructions : les instructions conditionnelles, qui permettent d'effectuer une opération dans le cas où une certaine condition est satisfaite et les itérations qui donnent la possibilité de répéter plusieurs fois la même instruction (pour des valeurs différentes des variables !).

2.1 Expressions booléennes

Le point commun aux diverses instructions décrites ci-dessous est qu'elles utilisent des expressions *booléennes*, c'est-à-dire dont l'évaluation donne l'une des deux valeurs `true` ou `false`.

2.1.1 Opérateurs de comparaisons

Les opérateurs booléens les plus simples sont

`==`	`!=`	`<`	`>`	`<=`	`>=`
égal	différent	plus petit	plus grand	plus petit ou égal	plus grand ou égal

Le résultat d'une comparaison sur des variables de type primitif :

```
a == b
```

est égal à `true` si l'évaluation de la variable `a` et de la variable `b` donnent le même résultat, il est égal à `false` sinon. Par exemple, `(5-2) == 3` a pour valeur `true`, mais `22/7 == 3.14159` a pour valeur `false`.

Remarque : Attention à ne pas écrire `a = b` qui est une affectation et pas une comparaison.

L'opérateur `!=` est l'opposé de ==, ainsi `a != b` prend la valeur `true` si l'évaluation de `a` et de `b` donne des valeurs différentes.

Les opérateurs de comparaison <, >, <=, >= ont des significations évidentes lorsqu'il s'agit de comparer des nombres. Noter qu'ils peuvent aussi servir à comparer des caractères ; pour les caractères latins courants c'est l'ordre alphabétique qui est exprimé.

2.1.2 Connecteurs

On peut construire des expressions booléennes comportant plusieurs comparateurs en utilisant les connecteurs `&&`, qui signifie *et*, `||` qui signifie *ou* et `!` qui signifie *non*.

Ainsi `C1 && C2` est évalué à `true` si et seulement si les deux expressions `C1` et `C2` le sont. De même `C1 || C2` est évalué à `true` si l'une deux expressions `C1` ou `C2` l'est.

Par exemple

```
!( ((a<c) && (c<b) && (b<d)) || ((c<a) && (a<d) && (d<b)) )
```

est une façon de tester si deux intervalles $[a, b]$ et $[c, d]$ sont disjoints ou contenus l'un dans l'autre.

Règle d'évaluation : en JAVA, l'évaluation de l'expression `C1 && C2` s'effectue dans l'ordre `C1` puis `C2` si nécessaire ; ainsi si `C1` est évaluée à `false` alors `C2` n'est pas évaluée. C'est aussi le cas pour `C1 || C2` qui est évaluée à `true` si c'est le cas pour `C1` et ceci sans que `C2` ne soit évaluée. Par exemple l'évaluation de l'expression

```
(3 > 4) && (2/0 > 0)
```

donne pour résultat `false` alors que

```
(2/0 > 0) && (3 > 4)
```

donne lieu à une erreur provoquée par la division par zéro et levant une exception (voir annexe).

2.2 Instructions conditionnelles

Il s'agit d'instructions permettant de n'effectuer une opération que si une certaine condition est satisfaite ou de programmer une alternative entre deux options.

2.2.1 If-else

La plus simple de ces instructions est celle de la forme :

```
if(C)
    I1
else
    I2
```

Dans cette écriture `C` est une expression booléenne (attention à ne pas oublier les parenthèses autour) ; `I1` et `I2` sont formées ou bien d'une seule instruction ou bien d'une suite d'instructions à l'intérieur d'une paire d'accolades { }. On rappelle que chaque instruction de JAVA se termine par un point virgule (`;` , symbole qui fait donc partie de l'instruction). Par exemple, les instructions

```
if(a >= 0)
    b = 1;
else
    b = -1;
```

permettent de calculer le signe de a et de le mettre dans b.

La partie else I2 est facultative, elle est omise si la suite I2 est vide c'est à dire s'il n'y a aucune instruction à exécuter dans le cas où C est évaluée à false.

On peut avoir plusieurs branches séparées par des else if comme par exemple dans :

```
if(a == 0 )       x = 1;
else if (a < 0)   x = 2;
else if (a > -5)  x = 3;
else              x = 4;
```

qui donne 4 valeurs possibles pour x suivant les valeurs de a.

2.2.2 Forme compacte

Il existe une forme compacte de l'instruction conditionnelle utilisée comme un opérateur ternaire (à trois opérandes) dont le premier est un booléen et les deux autres sont de type primitif. Cet opérateur s'écrit C ? E1 : E2. Elle est utilisée quand un if else paraît lourd, par exemple pour le calcul d'une valeur absolue :

```
x = (a > b)? a - b : b - a;
```

2.2.3 Aiguillage

Quand diverses instructions sont à réaliser suivant les valeurs que prend une variable, plusieurs if imbriqués deviennent lourds à mettre en œuvre, on peut les remplacer avantageusement par un aiguillage switch. Un tel aiguillage a la forme suivante dans laquelle x est une variable *d'un type primitif* (entier, caractère ou booléen, pas réel) et a,b,c sont des constantes représentant des valeurs que peut prendre cette variable. Lors de l'exécution les valeurs après chaque case sont testées l'une après l'autre jusqu'à obtenir celle prise par x ou arriver à default, ensuite toutes les instructions sont exécutées en séquence jusqu'à la fin. Par exemple dans l'instruction :

```
switch(x){
case a  : I1
case b  : I2
case c  : I3
default : I4
}
```

Si la variable x est évaluée à b alors toutes les suites d'instructions I2, I3, I4 seront exécutées, à moins que l'une d'entre elles ne contienne un break qui interrompt cette suite. Si la variable est évaluée à une valeur différente de a,b,c c'est la suite I4 qui est exécutée.

Pour sortir de l'instruction avant la fin, il faut passer par une instruction break . Le programme suivant est un exemple typique d'utilisation :

```
switch(c){
case 's':
    System.out.println("samedi est un jour de week-end");
```

```
        break;
    case 'd':
        System.out.println("dimanche est un jour de week-end");
        break;
    default:
        System.out.print(c);
        System.out.println(" n'est pas un jour de week-end");
        break;
    }
```

permet d'afficher les jours du week-end. Noter l'absence d'accolades dans les différents cas. Si l'on écrit plutôt de façon erronée en oubliant les **break** :

```
    switch(c){
    case 's':
        System.out.println("samedi est un jour de week-end");
    case 'd':
        System.out.println("dimanche est un jour de week-end");
    default:
        System.out.print(c);
        System.out.println(" n'est pas un jour de week-end");
        break;
  }
```

on obtiendra, dans le cas où `c` s'évalue à `'s'` :

```
samedi est un jour de week-end
dimanche est un jour de week-end
s n'est pas un jour de week-end
```

2.3 Itérations

Une itération permet de répéter plusieurs fois la même suite d'instructions. Elle est utilisée pour évaluer une somme, une suite récurrente, le calcul d'un plus grand commun diviseur par exemple. Elle sert aussi pour effectuer des traitements plus informatiques comme la lecture d'un fichier. On a l'habitude de distinguer les *boucles pour* (**for**) des *boucles tant-que* (**while**). Les premières sont utilisées lorsqu'on connaît, lors de l'écriture du programme, le nombre de fois où les opérations doivent être itérées, les secondes servent à exprimer des tests d'arrêt dont le résultat n'est pas prévisible à l'avance. Par exemple, le calcul d'une somme de valeurs pour i variant de 1 à n relève plutôt de la catégorie boucle-pour, celui du calcul d'un plus grand commun diviseur par l'algorithme d'Euclide relève plutôt d'une boucle tant-que.

2.3.1 Boucles pour (**for**)

L'itération de type boucle-pour en JAVA est un peu déroutante pour ceux qui la découvrent pour la première fois. L'exemple le plus courant est celui où on exécute une suite d'opérations pour `i` variant de `1` à `n`, comme dans :

```
int i;
for(i = 1; i <= n; i++)
    System.out.println(i);
```

Ici, on a affiché tous les entiers entre 1 et n. Prenons l'exemple de $n = 2$ et déroulons les calculs faits par l'ordinateur :

- étape 1 : i vaut 1, il est plus petit que n, on exécute l'instruction
  `System.out.println(i);`
  et on incrémente i ;
- étape 2 : i vaut 2, il est plus petit que n, on exécute l'instruction
  `System.out.println(i);`
  et on incrémente i ;
- étape 3 : i vaut 3, il est plus grand que n, on sort de la boucle.

Une forme encore plus courante est celle où on déclare `i` dans la boucle :

```
for(int i = 1; i <= n; i++)
    System.out.println(i);
```

Dans ce cas, on n'a pas accès à la variable `i` en dehors du corps de la boucle.
Un autre exemple est le calcul de la somme

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

qui se fait par

```
double s = 0.0;
for(int i = 1; i <= n; i++)
    s = s + 1/((double)i);
```

La conversion explicite en **`double`** est ici nécessaire, car sinon la ligne plus naturelle :

```
s = s + 1/i;
```

conduit à évaluer d'abord `1/i` comme une opération entière, autrement dit le quotient de 1 par i, i.e., 0. Et la valeur finale de `s` serait toujours 1.0.

La forme générale est la suivante :

```
for(Init; C; Inc)
    I
```

Dans cette écriture `Init` est une initialisation (pouvant comporter une déclaration), `Inc` est une incrémentation, et `C` un test d'arrêt, ce sont des expressions qui ne se terminent pas par un point virgule. Quant à `I`, c'est le corps de la boucle constitué d'une seule instruction ou d'une suite d'instructions entre accolades. `Init` est exécutée en premier, ensuite la condition `C` est évaluée si sa valeur est `true` alors la suite d'instructions `I` est exécutée suivie de l'instruction d'incrémentation `Inc` et un nouveau tour de boucle reprend avec l'évaluation de `C`. Noter que `Init` (tout comme `Inc`) peut être composée d'une seule expression ou bien de plusieurs, séparées par des `,` (virgules).

Noter que les instructions `Init` ou `Inc` de la forme générale (ou même les deux) peuvent être vides. Il n'y a alors pas d'initialisation ou pas d'incrémentation ; l'initialisation peut, dans ce cas, figurer avant le `for` et l'incrémentation à l'intérieur de `I`.

Insistons sur le fait que la boucle

```
for(int i = 1; i <= n; i++)
    System.out.println(i);
```

peut également s'écrire :

```
for(int i = 1; i <= n; i++)
{
    System.out.println(i);
}
```

pour faire ressortir le bloc d'instructions, ou encore :

```
for(int i = 1; i <= n; i++){
    System.out.println(i);
}
```

ce qui fait gagner une ligne...

2.3.2 Itérations tant que

Une telle instruction a la forme suivante :

```
while(C)
    I
```

où `C` est une condition et `I` une instruction ou un bloc d'instructions. L'itération évalue `C` et exécute `I` si le résultat est `true`, cette suite est répétée tant que l'évaluation de `C` donne la valeur `true`.

Un exemple classique de l'utilisation de `while` est le calcul du pgcd de deux nombres par l'algorithme d'Euclide. Cet algorithme consiste à remplacer le calcul de $\text{pgcd}(a, b)$ par celui de $\text{pgcd}(b, r)$ où r est le reste de la division de a par b et ceci tant que $r \neq 0$.

```
while(b != 0){
    r = a % b;
    a = b;
    b = r;
}
```

Examinons ce qu'il se passe avec $a = 28$, $b = 16$.
- étape 1 : $b = 16$ est non nul, on exécute le corps de la boucle, et on calcule $r = 12$;
- étape 2 : $a = 16$, $b = 12$ est non nul, on calcule $r = 4$;
- étape 3 : $a = 12$, $b = 4$, on calcule $r = 0$;
- étape 4 : $a = 4$, $b = 0$ et on sort de la boucle.

Notons enfin que boucles pour ou tant-que sont presque toujours interchangeables. Ainsi une forme équivalente de

```
    double s = 0.0;
    for(int i = 1; i <= n; i++)
        s += 1/((double)i);
```

est

```
    double s = 0.0;
    int i = 1;
    while(i <= n){
        s += 1/((double)i);
        i++;
    }
```

mais que la première forme est plus compacte que la seconde. On a tendance à utiliser une boucle `for` quand on peut prévoir le nombre d'itérations, et `while` dans les autres cas.

2.3.3 Itérations répéter tant que

Il s'agit ici d'effectuer l'instruction `I` et de ne la répéter que si la condition `C` est vérifiée. La syntaxe est :

```
    do
       I
    while(C)
```

À titre d'exemple, le problème de Syracuse est le suivant : soit m un entier plus grand que 1. On définit la suite u_n par $u_0 = m$ et

$$u_{n+1} = \begin{cases} u_n \div 2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

(la notation $n \div 2$ désigne le quotient de la division euclidienne de n par 2). Il est conjecturé, mais non encore prouvé que pour tout m, la suite prend la valeur 1 au bout d'un temps fini.

Pour vérifier numériquement cette conjecture, on écrit le programme JAVA suivant :

```
public class Syracuse{
    public static void main(String[] args){
        int n = Integer.parseInt(args[0]);

        do{
            if((n % 2) == 0)
                n /= 2;
            else
                n = 3*n+1;
        } while(n > 1);
        return;
    }
}
```

que l'on appelle par :

```
unix% java Syracuse 101
```

L'instruction magique `Integer.parseInt(args[0]);` permet de récupérer la valeur de l'entier 101 passé sur la ligne de commande.

2.4 Terminaison des programmes

Le programme que nous venons de voir peut être considéré comme étrange, voire dangereux. En effet, si la conjecture est fausse, alors le programme ne va jamais s'arrêter, on dit qu'il *ne termine pas*. Le problème de la terminaison des programmes est fondamental en programmation. Il faut toujours se demander si le programme qu'on écrit va terminer. D'un point de vue théorique, il est impossible de trouver un algorithme pour faire cela (cf. chapitre 6). D'un point de vue pratique, on doit examiner chaque boucle ou itération et prouver que chacune termine.

Voici quelques erreurs classiques, qui toutes simulent le mouvement perpétuel :

```
int i = 0;
while(true)
    i++;
```

ou bien

```
for(i = 0; i >= 0; i++)
    ;
```

On s'attachera à prouver que les algorithmes que nous étudions terminent bien.

2.5 Instructions de rupture de contrôle

Il y a trois telles instructions qui sont `return`, `break` et `continue`. L'instruction `return` doit être utilisée dans toutes les fonctions qui calculent un résultat (cf. chapitre suivant).

Les deux autres instructions de rupture sont beaucoup moins utilisées et peuvent être omises en première lecture. L'instruction `break` permet d'interrompre une suite d'instructions dans une boucle pour passer à l'instruction qui suit la boucle dans le texte du programme.

L'instruction `continue` a un effet similaire à celui de `break`, mais redonne le contrôle à l'itération suivante de la boucle au lieu d'en sortir.

2.6 Exemples

2.6.1 Méthode de Newton

On rappelle que si f est une fonction suffisamment raisonnable de la variable réelle x, alors la suite

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

converge vers une racine de f à partir d'un point de départ bien choisi.

Si $f(x) = x^2 - a$ avec $a > 0$, la suite converge vers $\sqrt{a}$. Dans ce cas particulier, la récurrence s'écrit :

$$x_{n+1} = x_n - (x_n^2 - a)/(2x_n) = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

La suite (x_n) converge, car elle est décroissante, minorée par $\sqrt{a}$. On en déduit que la suite $|x_{n+1} - x_n|$ tend vers 0. On itère la suite en partant de $x_0 = a$, et on s'arrête quand la différence entre deux valeurs consécutives est plus petite que $\varepsilon > 0$ donné. Cette façon de faire est plus stable numériquement (et moins coûteuse) que de tester $|x_n^2 - a| \leqslant \varepsilon$. Si on veut calculer $\sqrt{2}$ par cette méthode en JAVA, on écrit :

```
public class Newton{
    public static void main(String[] args){
        double a = 2.0, x, xold, eps;

        x = a;
        eps = 1e-10;
        do{
            // recopie de la valeur ancienne
            xold = x;
            // calcul de la nouvelle valeur
            x = (xold+a/xold)/2;
            System.out.println(x);
        } while(Math.abs(x-xold) > eps);
        System.out.print("Sqrt(a)=");
        System.out.println(x);
        return;
    }
}
```

ce qui donne :

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623730949
Sqrt(a)=1.4142135623730949
```

On peut également vérifier le calcul en comparant avec la fonction `Math.sqrt()` de JAVA.

Comment prouve-t-on que cet algorithme termine ? On aborde ici un point parfois épineux. En effet, nous avons montré que la suite (x_n) tend *mathématiquement* vers une limite, et que donc, *mathématiquement*, la différence $|x_n - x_{n-1}|$ tend vers 0. Les flottants utilisés par les langages de programmation sont une pauvre approximation des réels. Il convient donc d'être prudent dans leur utilisation, ce qui est un sujet de recherche et d'applications très riche et important.

Pour illustrer les problèmes de convergence, essayez de faire varier la valeur de la constante `eps` du programme.

Exercice 1. On considère la suite calculant $1/\sqrt{a}$ par la méthode de Newton, en utilisant $f(x) = a - 1/x^2$:

$$x_{n+1} = \frac{x_n}{2}\,(3 - ax_n^2).$$

Écrire une fonction JAVA qui calcule cette suite, et en déduire le calcul de $\sqrt{a}$. Cette suite converge-t-elle plus ou moins vite que la suite donnée ci-dessus ?

Chapitre 3

Méthodes : théorie et pratique

Nous donnons dans ce chapitre un aperçu général sur l'utilisation des méthodes (fonctions) dans un langage de programmation classique, sans nous occuper de la problématique objet, sur laquelle nous reviendrons dans le chapitre 5.

3.1 Pourquoi écrire des méthodes

Reprenons l'exemple du chapitre précédent :

```
public class Newton{
    public static void main(String[] args){
        double a = 2.0, x, xold, eps;

        x = a;
        eps = 1e-10;
        do{
            // recopie de la valeur ancienne
            xold = x;
            // calcul de la nouvelle valeur
            x = (xold+a/xold)/2;
            System.out.println(x);
        } while(Math.abs(x-xold) > eps);
        System.out.print("Sqrt(a)=");
        System.out.println(x);
        return;
    }
}
```

Nous avons écrit le programme implantant l'algorithme de Newton dans la méthode d'appel (la méthode `main`). Si nous avons besoin de faire tourner l'algorithme pour plusieurs valeurs de a dans le même temps, nous allons devoir recopier le programme à chaque fois. Le plus simple est donc d'écrire une méthode à part, qui ne fait que les calculs liés à Newton :

```
public class Newton2{
```

```
    static double sqrtNewton(double a, double eps){
        double xold, x = a;

        do{
            // recopie de la valeur ancienne
            xold = x;
            // calcul de la nouvelle valeur
            x = (xold+a/xold)/2;
            // System.out.println(x); // (1) pour deboguer
        } while(Math.abs(x-xold) > eps);
        return x;
    }

    public static void main(String[] args){
        double r;

        r = sqrtNewton(2, 1e-10);
        System.out.print("Sqrt(2)=");
        System.out.println(r);
        r = sqrtNewton(3, 1e-10);
        System.out.print("Sqrt(3)=");
        System.out.println(r);
    }
}
```

Remarquons également que nous avons séparé le calcul proprement dit de l'affichage du résultat. Si l'on a besoin de déboguer les calculs successifs, on peut décommenter la ligne (1) dans le code.

Écrire des méthodes remplit plusieurs rôles : au-delà de la possibilité de réutilisation des méthodes à différents endroits du programme, le plus important est de clarifier la structure du programme, pour le rendre lisible et compréhensible par d'autres personnes que le programmeur original.

3.2 Comment écrire des méthodes

3.2.1 Syntaxe

Une méthode prend des arguments en paramètres et donne en général un résultat. Elle se déclare par :

```
public static typeRes nomFonction(type1 nom1, ..., typek nomk)
```

Dans cette écriture `typeRes` est le type du résultat.

La *signature* d'une méthode est constituée de la suite ordonnée des types des paramètres.

Le résultat du calcul de la méthode doit être indiqué après un `return`. Il est obligatoire de prévoir une telle instruction dans toutes les branches d'une méthode. L'exécution d'un `return` a pour effet d'interrompre le calcul de la méthode en rendant le résultat à l'appelant.

On fait appel à une méthode par

```
nomFonction(var1, var2, ... , vark)
```

En général cet appel se situe dans une affectation.

En résumé, une syntaxe très courante est la suivante :

```
public static typeRes nomFonction(type1 nom1, ..., typek nomk){
    typeRes r;

    r = ...;
    return r;
}
...
public static void main(String[] args){
    type1 n1;
    type2 n2;
    ...
    typek nk;
    typeRes s;

    ...
    s = nomFonction(n1, n2, ..., nk);
    ...
    return;
}
```

3.2.2 Le type spécial `void`

Le type du résultat peut être `void`, dans ce cas la méthode ne rend pas de résultat. Elle opère par *effet de bord*, par exemple en affichant des valeurs à l'écran ou en modifiant des variables globales. Il est déconseillé d'écrire des méthodes qui procèdent par effet de bord, sauf bien entendu pour les affichages.

Un exemple typique est celui de la procédure principale :

```
// Voici mon premier programme
public class Premier{
    public static void main(String[] args){
        System.out.println("Bonjour !");
        return;
    }
}
```

Notons que le `return` n'est pas obligatoire dans une méthode de type `void`, à moins qu'elle ne permette de sortir de la méthode dans un branchement. Nous la mettrons souvent pour marquer l'endroit où on sort de la méthode, et par souci d'homogénéité de l'écriture.

3.2.3 La surcharge

En JAVA on peut définir plusieurs méthodes qui ont le même nom à condition que leurs signatures soient différentes. On appelle *surcharge* cette possibilité. Le compilateur doit être à même de déterminer la méthode dont il est question à partir du type des paramètres d'appel. En JAVA, l'opérateur + est surchargé : non seulement il permet de faire des additions, mais il permet de concaténer des chaînes de caractères (voir la section 5.5 pour plus d'information). Par exemple, reprenant le programme de calcul de racine carrée, on aurait pu écrire :

```
    public static void main(String[] args){
        double r;

        r = sqrtNewton(2, 1e-10);
        System.out.println("Sqrt(2)=" + r);
    }
```

3.3 Visibilité des variables

Les arguments d'une méthode sont passés par valeurs, c'est à dire que leur valeurs sont recopiées lors de l'appel. Après la fin du travail de la méthode les nouvelles valeurs, qui peuvent avoir été attribuées à ces variables, ne sont plus accessibles.

Ainsi il n'est pas possible d'écrire une méthode qui échange les valeurs de deux variables passées en paramètre, sauf à procéder par des moyens détournés peu recommandés.

Reprenons l'exemple donné au premier chapitre :

```
// Calcul de circonférence
public class Cercle{
    static float pi = (float)Math.PI;

    public static float circonference(float r) {
        return 2. * pi * r;
    }

    public static void main(String[] args){
        float c = circonference (1.5);

        System.out.print("Circonférence:  ");
        System.out.println(c);
        return;
    }
}
```

La variable `r` présente dans la définition de `circonference` est *instanciée* au moment de l'appel de la méthode par la méthode `main`. Tout se passe comme si le programme réalisait l'affectation `r = 1.5` au moment d'entrer dans `f`.

Dans l'exemple précédent, la variable `pi` est une *variable de classe*, ce qui veut dire qu'elle est connue et partagée par toutes les méthodes présentes dans la classe (ainsi

que par tous les objets de la classe, voir chapitre 5), ce qui explique qu'on peut l'utiliser dans la méthode `circonference`.

Pour des raisons de propreté des programmes, on ne souhaite pas qu'il existe beaucoup de ces variables de classe. L'idéal est que chaque méthode travaille sur ses propres variables, indépendamment des autres méthodes de la classe, autant que cela soit possible. Regardons ce qui se passe quand on écrit :

```
public class Essai{
    public static int f(int n){
        int m = n+1;

        return 2*m;
    }

    public static void main(String[] args){
        System.out.print("résultat=");
        System.out.println(f(4));
        return;
    }
}
```

La variable `m` n'est connue (on dit *vue*) que par la méthode `f`. En particulier, on ne peut l'utiliser dans la méthode `main` ou toute autre méthode qui serait dans la classe.

Compliquons encore :

```
public class Essai{
    public static int f(int n){
        int m = n+1;

        return 2*m;
    }

    public static void main(String[] args){
        int m = 3;

        System.out.print("résultat=");
        System.out.print(f(4));
        System.out.print(" m=");
        System.out.println(m);
        return;
    }
}
```

Qu'est-ce qui s'affiche à l'écran ? On a le choix entre :

```
résultat=10 m=5
```

ou

```
résultat=10 m=3
```

D'après ce qu'on vient de dire, la variable `m` de la méthode `f` n'est connue que de `f`, donc pas de `main` et c'est la seconde réponse qui est correcte. On peut imaginer que la variable `m` de `f` a comme nom réel `m-de-la-méthode-f`, alors que l'autre a pour nom `m-de-la-méthode-main`. Le compilateur et le programme ne peuvent donc pas faire de confusion.

3.4 Quelques conseils pour écrire un (petit) programme

Un beau programme est difficile à décrire, à peu près aussi difficile à caractériser qu'un beau tableau, ou une belle preuve. Il existe quand même quelques règles simples. Le premier lecteur d'un programme est soi-même. Si je n'arrive pas à me relire, il est difficile de croire que quelqu'un d'autre le pourra. On peut être amené à écrire un programme, le laisser dormir pendant quelques mois, puis avoir à le réutiliser. Si le programme est bien écrit, il sera facile à relire.

Grosso modo, la démarche d'écriture de petits ou gros programmes est à peu près la même, à un facteur d'échelle près. On découpe en tranches indépendantes le problème à résoudre, ce qui conduit à isoler des méthodes à écrire. Une fois cette architecture mise en place, il n'y a plus qu'à programmer chacune de celles-ci. Même après un découpage *a priori* du programme en méthodes, il arrive qu'on soit amené à écrire d'autres méthodes. Quand le décide-t-on ? De façon générale, pour ne pas dupliquer du code. Une autre règle simple est qu'un morceau de code ne doit jamais dépasser une page d'écran. Si cela arrive, on doit couper en deux ou plus. La clarté y gagnera.

La méthode `main` d'un programme JAVA doit ressembler à une sorte de table des matières de ce qui va suivre. Elle doit se contenter d'appeler les principales méthodes du programme. *A priori*, elle ne doit pas faire de calculs elle-même.

Les noms de méthode (comme ceux des variables) ne doivent pas se résumer à une lettre. Il est tentant pour un programmeur de succomber à la facilité et d'imaginer pouvoir programmer toutes les méthodes du monde en réutilisant sans cesse les mêmes noms de variables, de préférence avec un seul caractère par variable. Faire cela conduit rapidement à écrire du code non lisible, à commencer par soi. Ce style de programmation est donc proscrit. Les noms doivent être pertinents. Nous aurions pu écrire le programme concernant les cercles de la façon suivante :

```
public class D{
    static float z = (float)Math.PI;

    public static float e(float s){
        return 2. * z * s;
    }

    public static void main(String[] args){
        float y = e(1.5);

        System.out.println(y);
        return;
    }
}
```

ce qui aurait rendu la chose un peu plus difficile à lire.

Un programme doit être aéré : on écrit une instruction par ligne, on ne mégotte pas sur les lignes blanches. De la même façon, on doit commenter ses programmes. Il ne sert à rien de mettre des commentaires triviaux à toutes les lignes, mais tous les points difficiles du programme doivent avoir en regard quelques commentaires. Un bon début consiste à placer au-dessus de chaque méthode que l'on écrit quelques lignes décrivant le travail de la méthode, les paramètres d'appel, etc. Que dire de plus sur le sujet ? Le plus important pour un programmeur est d'adopter rapidement un style de programmation (nombre d'espaces, placement des accolades, etc.) et de s'y tenir.

Finissons avec un programme horrible, qui est le contre-exemple typique à ce qui précède :

```
public class mystere{public static void main(String[] args){
int
z=
Integer.parseInt(args[0]);doif((z%2)==0)
z
/=2;
else z=3*z+1;while(z>1);}}
```

Chapitre 4

Tableaux

La possibilité de manipuler des tableaux se retrouve dans tous les langages de programmation ; toutefois JAVA, qui est un langage avec des objets, manipule les tableaux d'une façon particulière que l'on va décrire ici.

4.1 Déclaration, construction, initialisation

L'utilisation d'un tableau permet d'avoir à sa disposition un très grand nombre de variables en utilisant un seul nom et donc en effectuant une seule déclaration. En effet, si on déclare un tableau de nom `tab` et de taille `n` contenant des valeurs de type `typ`, on a à sa disposition les variables `tab[0],tab[1], ..., tab[n-1]` qui se comportent comme des variables ordinaires de type `typ`.

En JAVA, on sépare la déclaration d'une variable de type tableau, la construction effective d'un tableau et l'initialisation du tableau.

La *déclaration* d'une variable de type tableau de nom `tab` dont les éléments sont de type `typ`, s'effectue par[1] :

```
typ[] tab;
```

Lorsque l'on a déclaré un tableau en JAVA on ne peut pas encore l'utiliser complètement. Il est en effet interdit par exemple d'affecter une valeur aux variables `tab[i]`, car il faut commencer par construire le tableau, ce qui signifie qu'il faut réserver de la place en mémoire (on parle d'*allocation mémoire*) avant de s'en servir.

L'opération de *construction* s'effectue en utilisant un `new`, ce qui donne :

```
tab = new typ[taille];
```

Dans cette instruction, `taille` est une constante entière ou une variable de type entier dont l'évaluation doit pouvoir être effectuée à l'exécution. Une fois qu'un tableau est créé avec une certaine taille, celle-ci ne peut plus être modifiée.

On peut aussi regrouper la déclaration et la construction en une seule ligne par :

```
typ[] tab = new typ[taille];
```

[1] ou de manière équivalente par `typ tab[];`. Nous préférerons la première façon de faire car elle respecte la convention suivant laquelle dans une déclaration, le type d'une variable figure complètement avant le nom de celle-ci. La seconde correspond à ce qui se fait en langage C.

L'exemple de programme le plus typique est le suivant :

```
int[] tab = new int[10];

for(int i = 0; i < 10; i++)
    tab[i] = i;
```

Pour des tableaux de petite taille on peut en même temps construire et initialiser un tableau et initialiser les valeurs contenues dans le tableau. L'exemple suivant regroupe les 3 opérations de déclaration, construction et initialisation de valeurs en utilisant une affectation suivie de {, } :

```
int[] tab = {1,2,4,8,16,32,64,128,256,512,1024};
```

La taille d'un tableau `tab` peut s'obtenir grâce à l'expression `tab.length`. Complétons l'exemple précédent :

```
int[] tab = {1,2,4,8,16,32,64,128,256,512,1024};

for(int i = 0; i < tab.length; i++)
    System.out.println(tab[i]);
```

Insistons encore une fois lourdement sur le fait qu'un tableau `tab` de n éléments en JAVA commence nécessairement à l'indice 0, le dernier élément accessible étant `tab[n-1]`.

Si `tab` est un tableau dont les éléments sont de type `typ`, on peut alors considérer `tab[i]` comme une variable et effectuer sur celle-ci toutes les opérations admissibles concernant le type `typ`, bien entendu l'indice `i` doit être inférieur à la taille du tableau donnée lors de sa construction. JAVA vérifie cette condition à l'exécution et une exception est levée si elle n'est pas satisfaite.

Donnons un exemple simple d'utilisation d'un tableau. Recherchons le plus petit élément dans un tableau donné :

```
public static int plusPetit(int[] x){
    int k = 0, n = x.length;

    for(int i = 1; i < n; i++)
        // invariant : k est l'indice du plus petit
        //             {\'e}l{\'e}ment de x[0..i-1]
        if(x[i] < x[k])
            k = i;
    return x[k];
}
```

4.2 Représentation en mémoire et conséquences

La mémoire accessible au programme peut être vue comme un ensemble de cases qui vont contenir des valeurs associées aux variables qu'on utilise. C'est le compilateur qui se charge d'associer aux noms symboliques les cases correspondantes, qui sont repérées par des numéros (des indices dans un grand tableau, appelés encore *adresses*).

Le programmeur moderne n'a pas à se soucier des adresses réelles, il laisse ce soin au compilateur (et au programme). Aux temps historiques, la programmation se faisait en manipulant directement les adresses mémoire des objets, ce qui était pour le moins peu confortable[2].

Quand on écrit :

```
int i = 3, j;
```

une case mémoire[3] est réservée pour chaque variable, et celle pour `i` remplie avec la valeur 3. Quand on exécute :

```
j = i;
```

le programme va chercher la valeur présente dans la case affectée à `i` et la recopie dans la case correspondant à `j`.

Que se passe-t-il maintenant quand on déclare un tableau ?

```
int[] tab;
```

Le compilateur réserve de la place pour la variable `tab` correspondante, mais pour le moment, aucune place n'est réservée pour les éléments qui constitueront `tab`. C'est ce qui explique que quand on écrit :

```
public class Bug1{
    public static void main(String[] args){
        int[] tab;

        tab[0] = 1;
    }
}
```

on obtient à l'exécution :

```
java.lang.NullPointerException at Bug1.main(Bug1.java:5)
```

C'est une erreur tellement fréquente que les compilateurs récents détectent ce genre de problème à la compilation (si possible).

Quand on manipule un tableau, on travaille en fait de façon indirecte avec lui, comme si on utilisait une armoire pour ranger ses affaires. Il faut toujours imaginer qu'écrire

```
tab[2] = 3;
```

veut dire au compilateur "retrouve l'endroit où tu as stocké `tab` en mémoire et met à jour la case d'indice 2 avec 3". En fait, le compilateur se rappelle d'abord où il a rangé son armoire, puis en déduit quel tiroir utiliser.

La valeur d'une variable tableau est une *référence*, c'est-à-dire l'adresse où elle est rangée en mémoire. Par exemple, la suite d'instructions suivante va avoir l'effet indiqué dans la mémoire :

[2] Tempérons un peu : dans des applications critiques (cartes à puce par exemple), on sait encore descendre à ce niveau là, quand on sait mieux que le compilateur comment gérer la mémoire. Ce sujet dépasse le cadre du cours, mais est enseigné en année 2.

[3] Une case mémoire pour un `int` de JAVA est formée de 4 octets consécutifs.

`int[] t;`

t
@0

null

`t=new int[3];`

t		
@10		
@10 : **t[0]**	@14 : **t[1]**	@18 : **t[2]**
0	0	0

`t[0]=2;`

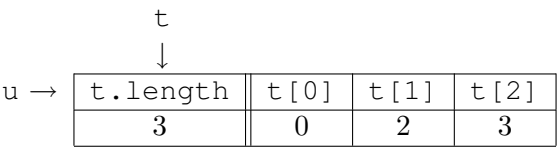
t		
@10		
@10 : **t[0]**	@14 : **t[1]**	@18 : **t[2]**
2	0	0

Après allocation, le tableau `t` est repéré par son adresse @10, et les trois cases par les adresses @10, @14 et @18. On remplit alors `t[0]` avec le nombre 2.

Expliquons maintenant ce qui se passe quand on écrit, avec l'idée de faire une copie de `t` :

```
int[] t = {1, 2, 3}; // t = @500

int[] u = t; // u = @500
u[0] = 0;
System.out.println(t[0]);
```

Les variables `u` et `t` désignent le même tableau, en programmation on dit qu'elles *font référence* au même tableau, c'est-à-dire à la même suite d'emplacements dans la mémoire



C'est ce qui explique que modifier `u[0]`, c'est modifier l'emplacement mémoire référencé par `u`, emplacement qui est également référencé par `t`. Et donc le programme affiche la nouvelle valeur de `t[0]`, à savoir 0.

Si on souhaite recopier le contenu d'un tableau dans un autre il faut écrire une fonction :

```
public static int[] copier(int[] x){
    int n = x.length;
    int[] y = new int[n];

    for(int i = 0; i < n; i++)
        y[i] = x[i];
    return y;
}
```

Noter aussi que l'opération de comparaison de deux tableaux `x == y` est évaluée à `true` dans le cas où `x` et `y` référencent le même tableau (par exemple si on a effectué

l'affectation `y = x`). Si on souhaite vérifier l'égalité des contenus, il faut écrire une fonction particulière :

```
public static boolean estEgal(int[] x, int[] y){
    if(x.length != y.length) return false;
    for(int i = 0; i < x.length; i++)
        if(x[i] != y[i])
            return false;
    return true;
}
```

Dans cette fonction, on compare les éléments terme à terme et on s'arrête dès que deux éléments sont distincts, en sortant de la boucle et de la fonction dans le même mouvement.

4.3 Tableaux à plusieurs dimensions, matrices

Un tableau à plusieurs dimensions est considéré en JAVA comme un tableau de tableaux. Par exemple, les matrices sont des tableaux à deux dimensions, plus précisément des tableaux de lignes. Leur déclaration peut se faire par :

```
typ[][] tab;
```

On doit aussi le construire à l'aide de `new`. L'instruction

```
tab = new typ[N][M];
```

construit un tableau à deux dimensions, qui est un tableau de N lignes à M colonnes. L'instruction `tab.length` retourne le nombre de lignes, alors que `tab[i].length` retourne la longueur du tableau `tab[i]`, c'est-à-dire le nombre de colonnes.

On peut aussi, comme pour les tableaux à une dimension, faire une affectation de valeurs en une seule fois :

```
int[2][3] tab = {{1,2,3},{4,5,6}};
```

qui déclare et initialise un tableau à 2 lignes et 3 colonnes. On peut écrire de façon équivalente :

```
int[][] tab = {{1,2,3},{4,5,6}};
```

Comme une matrice est un tableau de lignes, on peut fabriquer des matrices bizarres. Par exemple, pour déclarer une matrice dont la première ligne a 5 colonnes, la deuxième ligne 1 colonne et la troisième 2, on écrit

```
public static void main(String[] args){
    int[][] M = new int[3][];

    M[0] = new int[5];
    M[1] = new int[1];
    M[2] = new int[2];
}
```

Par contre, l'instruction :

```
int[][] N = new int[][3];
```

est incorrecte. On ne peut définir un tableau de colonnes.

On peut continuer à écrire un petit programme qui se sert de cela :

```
class Tab3{
    static void ecrire(int[] t){
        for(int j = 0; j < t.length; j++)
            System.out.println(t[j]);
    }
    public static void main(String[] args){
        int[][] M = new int[3][];

        M[0] = new int[5];
        M[1] = new int[1];
        M[2] = new int[2];
        for(int i = 0; i < M.length; i++)
            ecrire(M[i]);
    }
}
```

4.4 Les tableaux comme arguments de fonction

Les valeurs des variables tableaux (les références) peuvent être passées en argument, on peut aussi les retourner :

```
class Tab2{
    static int[] construire(int n){
        int[] t = new int[n];

        for(int i = 0; i < n; i++)
            t[i] = i;
        return t;
    }

    public static void main(String[] args){
        int[] t = construire(3);

        for(int i = 0; i < t.length; i++)
            System.out.println(t[i]);
    }
}
```

Considérons maintenant le programme suivant :

```
public class Test{

    public static void f(int[] t){
        t[0] = -10;
        return;
    }

    public static void main(String[] args){
        int[] t = {1, 2, 3};

        f(t);
        System.out.println("t[0]="+t[0]);
        return;
    }
}
```

Que s'affiche-t-il ? Pas 1 comme on pourrait le croire, mais -10. En effet, nous voyons là un exemple de *passage par référence* : le tableau `t` n'est pas recopié à l'entrée de la fonction `f`, mais on a donné à la fonction `f` la référence de `t`, c'est-à-dire le moyen de savoir où `t` est gardé en mémoire par le programme. On travaille donc sur le tableau `t` lui-même. Cela permet d'éviter des recopies fastidieuses de tableaux, qui sont souvent très gros. La lisibilité des programmes peut s'en ressentir, mais c'est la façon courante de programmer.

4.5 Exemples d'utilisation des tableaux

4.5.1 Algorithmique des tableaux

Nous allons écrire des fonctions de traitement de problèmes simples sur des tableaux contenant des entiers.

Commençons par remplir un tableau avec des entiers aléatoires de $[0, M[$, on écrit :

```
public class Tableaux{

    static int M = 128;

    // initialisation
    public static int[] aleatoire(int N){
        int[] t = new int[N];

        for(int i = 0; i < N; i++)
            t[i] = (int)(M * Math.random());
        return t;
    }
}
```

Ici, il faut convertir de force le résultat de `Math.random()*M` en entier de manière explicite, car `Math.random()` retourne un `double`.

Pour tester facilement les programmes, on écrit aussi une fonction qui affiche les éléments d'un tableau `t`, un entier par ligne :

```
// affichage à l'écran
public static void afficher(int[] t){
    for(int i = 0; i < t.length; i++)
        System.out.println(t[i]);
    return;
}
```

Le tableau `t` étant donné, un nombre m est-il élément de `t` ? On écrit pour cela une fonction qui retourne le plus petit indice i pour lequel `t[i]=m` s'il existe et -1 si aucun indice ne vérifie cette condition :

```
// retourne le plus petit i tel que t[i] = m s'il existe
// et -1 sinon.
public static int recherche(int[] t, int m){
    for(int i = 0; i < t.length; i++)
        if(t[i] == m)
            return i;
    return -1;
}
```

Passons maintenant à un exercice plus complexe. Le tableau `t` contient des entiers de l'intervalle $[0, M-1]$ qui ne sont éventuellement pas tous distincts. On veut savoir quels entiers sont présents dans le tableau et à combien d'exemplaire. Pour cela, on introduit un tableau auxiliaire `compteur`, de taille M, puis on parcourt `t` et pour chaque valeur `t[i]` on incrémente la valeur `compteur[t[i]]`.

À la fin du parcourt de `t`, il ne reste plus qu'à afficher les valeurs non nulles contenues dans `compteur` :

```
public static void afficher(int[] compteur){
    for(int i = 0; i < M; i++)
        if(compteur[i] > 0){
             System.out.print(i+" est utilisé ");
             System.out.println(compteur[i]+" fois");
        }
}

public static void compter(int[] t){
    int[] compteur = new int[M];

    for(int i = 0; i < M; i++)
        compteur[i] = 0;
    for(int i = 0; i < t.length; i++)
        compteur[t[i]] += 1;
    afficher(compteur);
}
```

4.5.2 Un peu d'algèbre linéaire

Un tableau est la structure de donnée la plus simple qui puisse représenter un vecteur. Un tableau de tableaux représente une matrice de manière similaire. Écrivons un programme qui calcule l'opération de multiplication d'un vecteur par une matrice à gauche. Si v est un vecteur colonne à m lignes et A une matrice $n \times m$, alors $w = Av$ est un vecteur colonne à n lignes. On a :

$$w_i = \sum_{k=0}^{m-1} A_{i,k} v_k$$

pour $0 \leqslant i < n$. On écrit d'abord la multiplication :

```
static double[] multMatriceVecteur(double[][] A,
                                   double[] v){
    int n = A.length;
    int m = A[0].length;
    double[] w = new double[n];

    // calcul de w = A * v
    for(int i = 0; i < n; i++){
        w[i] = 0;
        for(int k = 0; k < m; k++)
            w[i] += A[i][k] * v[k];
    }
    return w;
}
```

```
static double[] multMatriceVecteur(double[][] A,
                                   double[] v){
    int n = A.length;
    int m = A[0].length;
    double[] w = new double[n];

    // calcul de w = A * v
    for(int i = 0; i < n; i++){
        w[i] = 0;
        for(int k = 0; k < m; k++)
            w[i] += A[i][k] * v[k];
    }
    return w;
}
```

puis le programme principal :

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

```
        int n = 3, m = 4;
        double[][] A = new double[n][m]; // A est n x m
        double[] v = new double[m]; // v est m x 1
        double[] w;

        // initialisation de A
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                A[i][j] = Math.random();
        // initialisation de v
        for(int i = 0; i < m; i++)
            v[i] = Math.random();

        w = multMatriceVecteur(A, v); // (*)

        // affichage
        for(int i = 0; i < n; i++)
            System.out.println("w["+i+"]="+w[i]);
        return;
    }
```

```
    public static void main(String[] args){
        int n = 3, m = 4;
        double[][] A = new double[n][m]; // A est n x m
        double[] v = new double[m]; // v est m x 1
        double[] w;

        // initialisation de A
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                A[i][j] = Math.random();
        // initialisation de v
        for(int i = 0; i < m; i++)
            v[i] = Math.random();

        w = multMatriceVecteur(A, v); // (*)

        // affichage
        for(int i = 0; i < n; i++)
            System.out.println("w["+i+"]="+w[i]);
        return;
    }
```

On peut récrire la fonction de multiplication en passant les arguments par effets de bord sans avoir à créer le résultat dans la fonction, ce qui peut être coûteux quand on doit effectuer de nombreux calculs avec des vecteurs. On écrit alors plutôt :

```
static void multMatriceVecteur(double[] w,
                               double[][] A,
                               double[] v){
    int n = A.length;
    int m = A[0].length;

    // calcul de w = A * v
    for(int i = 0; i < n; i++){
        w[i] = 0;
        for(int k = 0; k < m; k++)
            w[i] += A[i][k] * v[k];
    }
}
```

```
static void multMatriceVecteur(double[] w,
                               double[][] A,
                               double[] v){
    int n = A.length;
    int m = A[0].length;

    // calcul de w = A * v
    for(int i = 0; i < n; i++){
        w[i] = 0;
        for(int k = 0; k < m; k++)
            w[i] += A[i][k] * v[k];
    }
}
```

Dans le programme principal, on remplacerait la ligne (*) par :

```
    w = new double[n];
    multMatriceVecteur(w, A, v);
```

4.5.3 Le crible d'Eratosthene

On cherche ici à trouver tous les nombres premiers de l'intervalle $[1, N]$. La solution déjà connue des Grecs consiste à écrire tous les nombres de l'intervalle les uns à la suite des autres. Le plus petit nombre premier est 2. On raye alors tous les multiples de 2 plus grands que 2 de l'intervalle, ils ne risquent pas d'être premiers. Le premier nombre qui n'a pas été rayé au-delà du nombre premier courant est lui-même premier, c'est le suivant à traiter. On raye ainsi les multiples de 3 sauf 3, etc. On s'arrête quand on

s'apprête à éliminer les multiples de $p > \sqrt{N}$ (rappelons que tout nombre non premier plus petit que N a un diviseur premier $\leqslant \sqrt{N}$).

Comment modéliser le crible ? On utilise un tableau de booléens `estpremier`, de taille $N+1$, qui représentera l'intervalle $[1, N]$. Il est initialisé à `true` au départ, car aucun nombre n'est rayé. À la fin du calcul, `p` $\geqslant 2$ est premier si et seulement si `estpremier[p] == true`. On trouve le programme complet dans la figure 4.1.

Remarquons que la ligne

```
kp = 2*p;
```

peut être avantageusement remplacée par

```
kp = p*p;
```

car tous les multiples de p de la forme up avec $u < p$ ont déjà été rayés du tableau à une étape précédente.

Il existe de nombreuses astuces permettant d'accélérer le crible. Notons également que l'on peut se servir du tableau des nombres premiers pour trouver les petits facteurs de petits entiers. Ce n'est pas la meilleure méthode connue pour trouver des nombres premiers ou factoriser les nombres qui ne le sont pas. Revenez donc me voir en cours de Cryptologie si ça vous intéresse.

4.5.4 Jouons à la bataille rangée

On peut également se servir de tableaux pour représenter des objets *a priori* plus compliqués. Nous allons décrire ici une variante simplifiée du célèbre jeu de bataille, que nous appelerons *bataille rangée*. La règle est simple : le donneur distribue 32 cartes (numérotées de 1 à 32) à deux joueurs, sous la forme de deux piles de cartes, face sur le dessous. À chaque tour, les deux joueurs, appelés Alice et Bob, retournent la carte du dessus de leur pile. Si la carte d'Alice est plus forte que celle de Bob, elle marque un point ; si sa carte est plus faible, c'est Bob qui marque un point. Gagne celui des deux joueurs qui a marqué le plus de points à la fin des piles.

Le programme de jeu doit contenir deux phases : dans la première, le programme bat et distribue les cartes entre les deux joueurs. Dans un second temps, le jeu se déroule.

Nous allons stocker les cartes dans un tableau `donne[0..32[` avec la convention que la carte du dessus se trouve en position 31.

Pour la première phase, battre le jeu revient à fabriquer une permutation au hasard des éléments du tableau `donne`. L'algorithme le plus efficace pour cela utilise un générateur aléatoire (la fonction `Math.random()` de JAVA, qui renvoie un réel aléatoire entre 0 et 1), et fonctionne selon le principe suivant. On commence par tirer un indice j au hasard entre 0 et 31 et on permute `donne[j]` et `donne[31]`. On continue alors avec le reste du tableau, en tirant un indice entre 0 et 30, etc. La fonction JAVA est alors (nous allons ici systématiquement utiliser le passage par référence des tableaux) :

```
static void battre(int[] donne){
    int n = donne.length, i, j, tmp;

    for(i = n-1; i > 0; i--){
        // on choisit un entier j de [0..i]
```

```
// Retourne le tableau des nombres premiers
// de l'intervalle [2..N]
public static int[] Eratosthene(int N){
    boolean[] estpremier = new boolean[N+1];
    int p, kp, nbp;

    // initialisation
    for(int n = 2; n < N+1; n++)
        estpremier[n] = true;
    // boucle d'{\'e}limination
    p = 2;
    while(p*p <= N){
        // élimination des multiples de p
        // on a déjà éliminé les multiples de q < p
        kp = 2*p; // (cf. remarque)
        while(kp <= N){
            estpremier[kp] = false;
            kp += p;
        }
        // recherche du nombre premier suivant
        do{
            p++;
        } while(!estpremier[p]);
    }
    // comptons tous les nombres premiers <= N
    nbp = 0;
    for(int n = 2; n <= N; n++)
        if(estpremier[n])
            nbp++;

    // mettons les nombres premiers dans un tableau
    int[] tp = new int[nbp];
    for(int n = 2, i = 0; n <= N; n++)
        if(estpremier[n])
              tp[i++] = n;
    return tp;
}
```

FIG. 4.1 – Crible d'Eratosthene.

```
            j = (int)(Math.random() * (i+1));
            // on permute donne[i] et donne[j]
            tmp = donne[i];
            donne[i] = donne[j];
            donne[j] = tmp;
        }
    }
```

```
    static void battre(int[] donne){
        int n = donne.length, i, j, tmp;

        for(i = n-1; i > 0; i--){
            // on choisit un entier j de [0..i]
            j = (int)(Math.random() * (i+1));
            // on permute donne[i] et donne[j]
            tmp = donne[i];
            donne[i] = donne[j];
            donne[j] = tmp;
        }
    }
```

La fonction qui crée une donne à partir d'un paquet de n cartes est alors :

```
    static int[] creerJeu(int n){
        int[] jeu = new int[n];

        for(int i = 0; i < n; i++)
            jeu[i] = i+1;
        battre(jeu);
        return jeu;
    }
```

```
    static int[] creerJeu(int n){
        int[] jeu = new int[n];

        for(int i = 0; i < n; i++)
            jeu[i] = i+1;
        battre(jeu);
        return jeu;
    }
```

et nous donnons maintenant le programme principal :

```
    public static void main(String[] args){
        int[] donne;
```

```
        donne = creerJeu(32);
        afficher(donne);
        jouer(donne);
    }
```

```
    public static void main(String[] args){
        int[] donne;

        donne = creerJeu(32);
        afficher(donne);
        jouer(donne);
    }
```

Nous allons maintenant jouer. Cela se passe en deux temps : dans le premier, le donneur distribue les cartes entre les deux joueurs, Alice et Bob. Dans le second, les deux joueurs jouent, et on affiche le nom du vainqueur, celui-ci étant déterminé à partir du signe du gain d'Alice (voir plus bas) :

```
    static void jouer(int[] donne){
        int[] jeuA = new int[donne.length/2];
        int[] jeuB = new int[donne.length/2];
        int gainA;

        distribuer(jeuA, jeuB, donne);
        gainA = jouerAB(jeuA, jeuB);
        if(gainA > 0) System.out.println("A gagne");
        else if(gainA < 0) System.out.println("B gagne");
        else System.out.println("A et B sont ex aequo");
    }
```

```
    static void jouer(int[] donne){
        int[] jeuA = new int[donne.length/2];
        int[] jeuB = new int[donne.length/2];
        int gainA;

        distribuer(jeuA, jeuB, donne);
        gainA = jouerAB(jeuA, jeuB);
        if(gainA > 0) System.out.println("A gagne");
        else if(gainA < 0) System.out.println("B gagne");
        else System.out.println("A et B sont ex aequo");
    }
```

Le tableau `donne[0..31]` est distribué en deux tas, en commençant par Alice, qui va recevoir les cartes de rang pair, et Bob celles de rang impair. Les cartes sont données à partir de l'indice 31 :

```
// donne[] contient les cartes qu'on distribue à partir
// de la fin. On remplit jeuA et jeuB à partir de 0
static void distribuer(int[] jeuA,int[] jeuB,int[] donne){
    int iA = 0, iB = 0;

    for(int i = donne.length-1; i >= 0; i--)
        if((i % 2) == 0)
            jeuA[iA++] = donne[i];
        else
            jeuB[iB++] = donne[i];
}
```


```
// donne[] contient les cartes qu'on distribue à partir
// de la fin. On remplit jeuA et jeuB à partir de 0
static void distribuer(int[] jeuA,int[] jeuB,int[] donne){
    int iA = 0, iB = 0;

    for(int i = donne.length-1; i >= 0; i--)
        if((i % 2) == 0)
            jeuA[iA++] = donne[i];
        else
            jeuB[iB++] = donne[i];
}
```


On s'intéresse au gain d'Alice, obtenu par la différence entre le nombre de cartes gagnées par Alice et celles gagnées par Bob. Il suffit de mettre à jour cette variable au cours du calcul :

```
static int jouerAB(int[] jeuA, int[] jeuB){
    int gainA = 0;

    for(int i = jeuA.length-1; i >= 0; i--)
        if(jeuA[i] > jeuB[i])
            gainA++;
        else if(jeuA[i] < jeuB[i])
            gainA--;
    return gainA;
}
```

```
static int jouerAB(int[] jeuA, int[] jeuB){
    int gainA = 0;

    for(int i = jeuA.length-1; i >= 0; i--)
        if(jeuA[i] > jeuB[i])
```

```
                gainA++;
            else if(jeuA[i] < jeuB[i])
                gainA--;
        return gainA;
    }
```

Exercice. (Programmation du jeu de bataille) Dans le jeu de bataille (toujours avec les cartes 1..32), le joueur qui remporte un pli le stocke dans une deuxième pile à côté de sa pile courante, les cartes étant stockées dans l'ordre d'arrivée (la première arrivée étant mise au bas de la pile), formant une nouvelle pile. Quand il a fini sa première pile, il la remplace par la seconde et continue à jouer. Le jeu s'arrête quand un des deux joueurs n'a plus de cartes. Programmer ce jeu.

4.5.5 Pile

On a utilisé ci-dessus un tableau pour stocker une pile de cartes, la dernière arrivée étant utilisée aussitôt. Ce concept de *pile* est fondamental en informatique.

Par exemple, considérons le programme JAVA :

```
public static int g(int n){
    return 2*n;
}
public static int f(int n){
    return g(n)+1;
}
public static void main(String[] args){
    int m = f(3); // (1)

    return;
}
```

Quand la fonction `main` s'exécute, l'ordinateur doit exécuter l'instruction (1). Pour ce faire, il garde sous le coude cette instruction, appelle la fonction `f`, qui appelle elle-même la fonction `g`, puis revient à ses moutons en remplissant la variable `m`. Garder sous le coude se traduit en fait par le stockage dans une pile des appels de cette information.

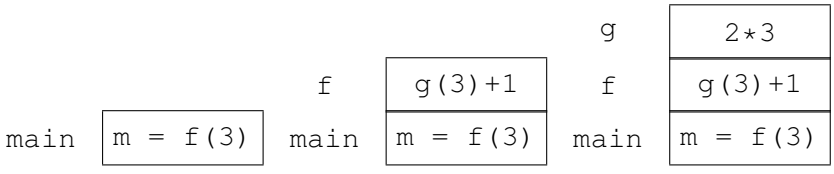


FIG. 4.2 – Pile des appels.

Le programme `main` appelle la fonction `f` avec l'argument 3, et `f` elle-même appelle `g` avec l'argument 3, et celle-ci retourne la valeur 6, qui est ensuite retournée à `f`, et ainsi de suite :

Cette notion sera complétée au chapitre 6.

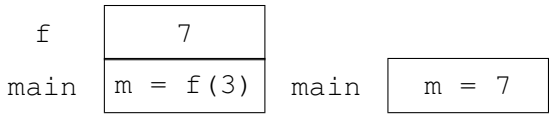


FIG. 4.3 – Pile des appels (suite).

Chapitre 5

Classes, objets

Ce chapitre est consacré à l'organisation générale d'une classe en JAVA, car jusqu'ici nous nous sommes plutôt intéressés aux différentes instructions de base du langage.

5.1 Introduction

Nous avons pour le moment utilisé des types primitifs, ou des tableaux constitués d'éléments du même type (primitif). En fonction des problèmes, on peut vouloir agréger des éléments de types différents, en créant ainsi de nouveaux types.

5.1.1 Déclaration et création

On peut créer de nouveaux types en JAVA. Cela se fait par la création d'une *classe*, qui est ainsi la description abstraite d'un ensemble. Par exemple, si l'on veut représenter les points du plan, on écrira :

```
public class Point{
    int abs, ord;
}
```

Un *objet* de la classe sera une instance de cette classe. Par certains côtés, c'est déjà ce qu'on a vu avec les tableaux :

```
int[] t;
```

déclare une variable `t` qui sera de type tableau d'entiers. Comme pour les tableaux, on devra allouer de la place pour un objet, par la même syntaxe :

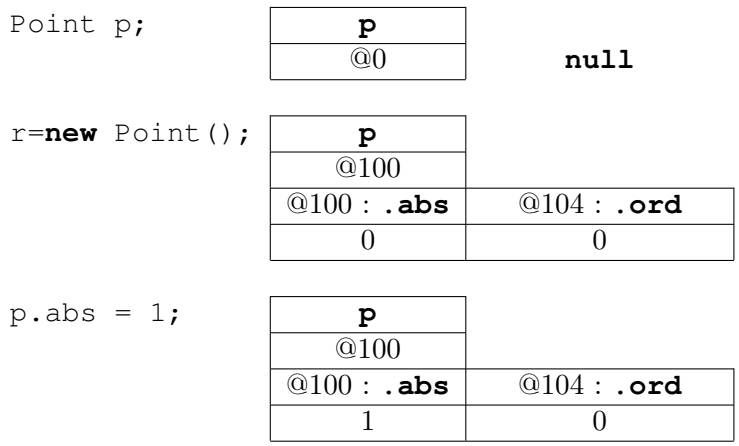
```
    public static void main(String[] args){
        Point p;                // (1)

        p = new Point();        // (2)
        p.abs = 1;
        p.ord = 2;
    }
```

On a déclaré à la ligne (1) une variable `p` de type `Point`. À la ligne (2), on crée une instance de la classe `Point`, un objet ; pour être plus précis, on a fait appel au constructeur implicite de la classe `Point` (nous verrons d'autres constructeurs explicites plus loin). La variable `p` est une référence à cet objet, tout comme pour les tableaux. `abs` et `ord` sont des *champs* d'un objet de la classe ; `p.abs` et `p.ord` se manipulent comme des variables de type entier.

5.1.2 Objet et référence

Insistons sur le fait qu'une variable de type `Point` contient une référence à l'objet créé en mémoire, comme dans le cas des tableaux.

D'un point de vue graphique, on a ainsi (comme pour les tableaux) :



Cela explique que la recopie d'objet doive être considérée avec soin. Le programme suivant :

```
public static void main(String[] args){
    Point p = new Point(), q;

    p.abs = 1;
    p.ord = 2;
    q = p;
    q.abs = 3;
    q.ord = 4;
    System.out.print(p.abs);
    System.out.print(q.abs);
}
```

va afficher

```
3
3
```

La variable `q` contient une référence à l'objet déjà référencé par `p`. Pour recopier le contenu de `p`, il faut écrire à la place :

```
    public static void main(String[] args){
        Point p = new Point(), q;

        p.abs = 1;
        p.ord = 1;
        q = new Point();
        q.abs = p.abs;
        q.ord = q.abs;
        System.out.print(p.abs);
        System.out.print(q.abs);
    }
```

Remarquons que tester `p == q` ne teste en fait que l'égalité des références, pas l'égalité des contenus. Dans l'exemple, les deux références sont différentes, même si les deux contenus sont égaux.

Il est utile de garder à l'esprit qu'un objet est créé une seule fois pendant l'exécution d'un programme. Dans la suite du programme, sa référence peut être passée aux autres variables du même type.

5.1.3 Constructeurs

Par défaut, chaque classe est équipée d'un *constructeur implicite*, comme dans l'exemple donné précédemment. On peut également créer un constructeur explicite, pour simplifier l'écriture de la création d'objets. Par exemple, on aurait pu écrire :

```
public class Point{
    int abs;
    int ord;

    public Point(int a, int o){
        this.abs = a;
        this.ord = o;
    }

    public static void main(String[] args){
        Point p = new Point(1, 2);
    }
}
```

Le mot clef **this** fait référence à l'objet qui vient d'être créé.

Remarque : quand on déclare un constructeur explicite, on perd automatiquement le constructeur implicite. Dans l'exemple précédent, un appel :

```
    Point p = new Point();
```

provoquera une erreur à la compilation.

5.2 Autres composants d'une classe

Une classe est bien plus qu'un simple type. Elle permet également de regrouper les fonctions de base opérant sur les objets de la classe, ainsi que divers variables ou constantes partagées par toutes les méthodes, mais aussi les objets.

5.2.1 Méthodes de classe et méthodes d'objet

On n'utilise pratiquement que des méthodes de classe dans ce cours. Une *méthode de classe* n'est rien d'autre que l'expression complète pour méthode, comme nous l'avons utilisé jusqu'à présent. Une telle méthode est associée à la classe dans laquelle elle est définie. D'un point de vue syntaxique, on déclare une telle fonction en la faisant précéder du mot réservé `static`.

On peut ajouter à la classe `Point` des méthodes sur les points et droites du plan, une autre pour manipuler des segments, une troisième pour les polygones etc. Cela donne une structure modulaire, plus agréable à lire, pour les programmes. Par exemple :

```
public class Point{
    ...
    public static void afficher(Point p){
        System.out.print("("+p.x+", "+p.y+")");
    }
}
```

Il existe également des *méthodes d'objet*, qui sont associées à un objet de la classe. L'appel se fait alors par `NomObjet.NomFonction`. Dans la description d'une méthode non statique on fait référence à l'objet qui a servi à l'appel par le nom `this`.

Affichons les coordonnées d'un point (voir 5.1.1 pour la définition de la classe `Point`) :

```
    public void afficher(){
        System.out.println(" Point de coordonn{\'e}es "
                           + this.abs +"  " +  this.ord);
    }
```

qui sera utilisé par exemple dans

```
    p.afficher();
```

On a utilisé la méthode d'objet pour afficher `p`. Il existe une alternative, avec la méthode `toString()`, définie par :

```
    public String toString(){
        return "(" + this.abs + ", " + this.ord + ")";
    }
```

Elle permet d'afficher facilement un objet de type `Point`. En effet, si l'on veut afficher le point `p`, il suffit alors d'utiliser l'instruction :

```
    System.out.print(p);
```

et cela affichera le point sous forme d'un couple (x, y). Terminons par une méthode qui retourne l'opposé d'un point par rapport à l'origine :

```
    public Point oppose(){
        return new Point(- this.abs, - this.ord);
    }
```

5.2.2 Passage par référence

Le même phénomène déjà décrit pour les tableaux à la section 4.4 se produit pour les objets, ce que l'on voit avec l'exemple qui suit :

```
public class Abscisse{
    int x;

    public static void f(Abscisse a){
        a.x = 2;
        return;
    }

    public static void main(String[] args){
        Abscisse a = new Abscisse();

        a.x = -1;
        f(a);
        System.out.println("a="+a.x);
        return;
    }
}
```

La réponse est `a=2` et non `a=-1`.

5.2.3 Variables de classe

Les variables de classes sont communes à une classe donnée, et se comportent comme les variables globales dans d'autres langages. Considérons le cas (artificiel) où on veut garder en mémoire le nombre de points créés. On va utiliser une variable `nbPoints` pour cela :

```
public class Point{
    int abs, ord;

    static int nbPoints = 0;

    public Point(){
        nbPoints++;
    }

    public static void main(String[] args){
        Point P = new Point();
```

```
        System.out.println("Nombre de points créées : "
                           + nbPoints);
    }
}
```

Une *constante* se déclare en rajoutant le mot clef **final** :

```
    final static int promotion = 2010;
```

elle ne pourra pas être modifiée par les méthodes de la classe, ni par aucune autre classe.

5.2.4 Utiliser plusieurs classes

Lorsque l'on utilise une classe dans une autre classe, on doit faire précéder les noms des méthodes du nom de la première classe suivie d'un point.

```
public class Exemple{
    public static void main(String[] args){
        Point p = new Point(0, 0);

        Point.afficher(p);
        return;
    }
}
```

Attention : il est souvent imposé par le compilateur qu'il n'y ait qu'une seule classe **public** par fichier. Il nous arrivera cependant dans la suite du poly de présenter plusieurs classes publiques l'une immédiatement à la suite de l'autre.

5.3 Autre exemple de classe

Les classes présentées pour le moment agrégeaient des types identiques. On peut définir la classe des produits présents dans un magasin par

```
public class Produit{
    String nom;
    int nb;
    double prix;
}
```

On peut alors gérer un stock de produits, et donc faire des tableaux d'objets. On doit d'abord allouer le tableau, puis chaque élément :

```
public class GestionStock{
    public static void main(String[] args){
        Produit[] s;

        s = new Produit[10];      // place pour le tableau
        s[0] = new Produit();     // place pour l'objet
```

```
        s[0].nom = "ordinateur";
        s[0].nb = 5;
        s[0].prix = 7522.50;
    }
}
```

5.4 Public et private

Nous avons déjà rencontré le mot réservé `public` qui permet par exemple à `java` de lancer un programme dans sa syntaxe immuable :

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

On doit garder en mémoire que `public` désigne les méthodes, champs, ou constantes qui doivent être visibles de l'extérieur de la classe. C'est le cas de la méthode `afficher` de la classe `Point` décrite ci-dessus. Elles pourront donc être appelées d'une autre classe, ici de la classe `Exemple`.

Quand on ne souhaite pas permettre un appel de ce type, on déclare alors une méthode avec le mot réservé `private`. Cela permet par exemple de protéger certaines variables ou constantes qui ne doivent pas être connues de l'extérieur, ou bien encore de forcer l'accès aux champs d'un objet en passant par des méthodes publiques, et non par les champs eux-mêmes. On en verra un exemple avec le cas des `String` au chapitre 5.6.

5.5 Un exemple de classe prédéfinie : la classe `String`

5.5.1 Propriétés

Une chaîne de caractères est une suite de symboles que l'on peut taper sur un clavier ou lire sur un écran. La déclaration d'une variable susceptible de contenir une chaîne de caractères se fait par

```
String u;
```

Un point important est que l'on ne peut pas modifier une chaîne de caractères, on dit qu'elle est non *mutable*. On peut par contre l'afficher, la recopier, accéder à la valeur d'un des caractères et effectuer un certain nombre d'opérations comme la concaténation, l'obtention d'une sous-chaîne, on peut aussi vérifier l'égalité de deux chaînes de caractères.

La façon la plus simple de créer une chaîne est d'utiliser des constantes comme :

```
String s = "123";
```

On peut également concaténer des chaînes, ce qui est très facile à l'aide de l'opérateur + qui est surchargé :

```
String s = "123" + "x" + "[]";
```

On peut également fabriquer une chaîne à partir de variables :

```
int i = 3;
String s = "La variable i vaut " + i;
```

qui permettra un affichage agréable en cours de programme. Comment comprendre cette syntaxe ? Face à une telle demande, le compilateur va convertir la valeur de la variable `i` sous forme de chaîne de caractères qui sera ensuite concaténée à la chaîne constante. Dans un cas plus général, une expression telle que :

```
MonObjet o;
String s = "Voici mon objet : " + o;
```

donnera le résultat attendu si une méthode d'objet `toString` est disponible pour la classe `MonObjet`. Sinon, l'adresse de `o` en mémoire est affichée (comme pour les tableaux). On trouvera un exemple commenté au chapitre 16.

Voici d'autres exemples :

```
String v = new String(u);
```

recopie la chaîne `u` dans la chaîne `v`.

```
int l = u.length();
```

donne la longueur de la chaîne `u`. Noter que `length` est une fonction sur les chaînes de caractères, tandis que sur les tableaux, c'est une valeur ; ceci explique la différence d'écriture : les parenthèses pour la fonction sur les chaînes de caractères sont absentes dans le cas des tableaux.

```
char x = u.charAt(i);
```

donne à `x` la valeur du i-ème caractère de la chaîne `u`, noter que le premier caractère s'obtient par `u.charAt(0)`.

On peut simuler (artificiellement) le comportement de la classe `String` de la façon suivante, ce qui donne un exemple d'utilisation de `private` :

```
public class Chaine{
    private char[] s;

    public Chaine(char[] t){
        this.s = new char[t.length];
        for(int i = 0; i < t.length; i++)
            this.s[i] = t[i];
    }

    // s.length()
    public int longueur(){
        return s.length;
    }

    // s.charAt(i)
    public char caractere(int i){
        return s[i];
    }
}
```

```
public class TestChaine{
    public static void main(String[] args){
        char[] t = {'a', 'b', 'c'};
        Chaine str = new Chaine(t);

        System.out.println(str.caractere(0)); // correct
        System.out.println(str.s[0]); // erreur
    }
}
```

Ainsi, on sait accéder au i-ième caractère en lecture, mais il n'y a aucun moyen d'y accéder en écriture. On a empêché les classes extérieures à `Chaine` d'accéder à la représentation interne de l'objet. De cette façon, on peut changer celle-ci en fonction des besoins (cela sera expliqué de façon plus complète dans les cours de deuxième année).

```
u.compareTo(v);
```

entre deux `String` a pour résultat un nombre entier négatif si `u` précède `v` dans l'ordre lexicographique (celui du dictionnaire), 0 si les chaînes `u` et `v` sont égales, et un nombre positif si `v` précède `u`.

```
w = u.concat(v); // équivalent de w = u + v;
```

construit une nouvelle chaîne obtenue par concaténation de `u` suivie de `v`. Noter que `v.concat(u)` est une chaîne différente de la précédente.

5.5.2 Arguments de `main`

La méthode `main` qui figure dans tout programme que l'on souhaite exécuter doit avoir un paramètre de type tableau de chaînes de caractères. On déclare alors la méthode par

```
public static void main(String[] args)
```

Pour comprendre l'intérêt de tels paramètres, supposons que la méthode `main` se trouve à l'intérieur d'un programme qui commence par `public class Classex`. On peut alors utiliser les valeurs et variables `args.length`, `args[0]`, `args[1]`, ...à l'intérieur de la procédure `main`. Celles-ci correspondent aux chaînes de caractères qui suivent `java Classex` lorsque l'utilisateur demande d'exécuter son programme.

Par exemple si on a écrit une procédure main :

```
public static void main(String[] args){
    for(int i = args.length-1; i >= 0; i--)
        System.out.print(args[i] + "  ");
    System.out.println();
}
```

et qu'une fois celle-ci compilée on demande l'exécution par

```
java Classex marquise d'amour me faites mourir
```

on obtient comme résultat

```
mourir faites me d'amour marquise
```

Noter que l'on peut transformer une chaîne de caractères `u` composée de chiffres décimaux en un entier par la fonction `Integer.parseInt()` comme dans le programme suivant :

```
public class Additionner{

  public static void main(String[] args){
      if(args.length != 2)
          System.out.println("mauvais nombre d'arguments");
      else{
          int s = Integer.parseInt(args[0]);

          s += Integer.parseInt(args[1]);
          System.out.println (s);
      }
  }
}
```

On peut alors demander

```
java Additionner 1052 958
```

l'interpréteur répond :

```
2010
```

Notons qu'il existe d'autres fonctions de conversion en double, long, etc..

5.6 Pour aller plus loin

La programmation objet est un paradigme de programmation dans lequel les programmes sont dirigés par les données. Au niveau de programmation du cours, cette façon de programmer apparaît essentiellement comme une différence syntaxique. Il faudra attendre les cours de deuxième année (INF431) pour voir l'intérêt de la programmation objet, avec la notion d'héritage.

Chapitre 6

Récursivité

Jusqu'à présent, nous avons programmé des fonctions simples, qui éventuellement en appelaient d'autres. Rien n'empêche d'imaginer qu'une fonction puisse s'appeler elle-même. C'est ce qu'on appelle une fonction *récursive*. L'intérêt d'une telle fonction peut ne pas apparaître clairement au premier abord, ou encore faire peur. D'un certain point de vue, elles sont en fait proches du formalisme de la relation de récurrence en mathématique. Bien utilisées, les fonctions récursives permettent dans certains cas d'écrire des programmes beaucoup plus lisibles, et permettent d'imaginer des algorithmes dont l'analyse sera facile et l'implantation récursive aisée. Nous introduirons ainsi plus tard un concept fondamental de l'algorithmique, le principe de *diviser-pour-résoudre*. Les fonctions récursives seront indispensables dans le traitement des types récursifs, qui seront introduits en INF-421.

Finalement, on verra que l'introduction de fonctions récursives ne se limite pas à une nouvelle syntaxe, mais qu'elle permet d'aborder des problèmes importants de l'informatique, comme la non-terminaison des problèmes ou l'indécidabilité de certains problèmes.

6.1 Premiers exemples

L'exemple le plus simple est celui du calcul de $n!$. Rappelons que $0! = 1! = 1$ et que $n! = n \times (n-1)!$. De manière itérative, on écrit :

```
public static int factorielle(int n){
    int f = 1;

    for(int k = n; k > 1; k--)
        f *= k;
    return f;
}
```

qui implante le calcul par accumulation du produit dans la variable `f`.

De manière récursive, on peut écrire :

```
public static int fact(int n){
    if(n == 0) return 1; // cas de base
    else return n * fact(n-1);
```

```
    }
```

On a collé d'aussi près que possible à la définition mathématique. On commence par le cas de base de la récursion, puis on écrit la relation de récurrence.

La syntaxe la plus générale d'une fonction récursive est :

```
public static <type_de_retour> <nomFct>(<args>){
    [déclaration de variables]
    [test d'arrêt]
    [suite d'instructions]
    [appel de <nomFct>(<args'>)]
    [suite d'instructions]
    return <résultat>;
}
```

Regardons d'un peu plus près comment fonctionne un programme récursif, sur l'exemple de la factorielle. L'ordinateur qui exécute le programme voit qu'on lui demande de calculer `fact(3)`. Il va en effet stocker dans un tableau le fait qu'on veut cette valeur, mais qu'on ne pourra la calculer qu'après avoir obtenu la valeur de `fact(2)`. On procède ainsi (on dit qu'on *empile les appels* dans ce tableau, qui est une pile) jusqu'à demander la valeur de `fact(0)` (voir figure 6.1).

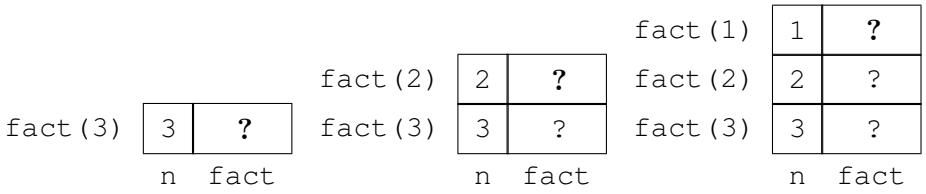


FIG. 6.1 – Empilement des appels récursifs.

Arrivé au bout, il ne reste plus qu'à *dépiler* les appels, pour de proche en proche pouvoir calculer la valeur de `fact(3)`, cf. figure 6.2.

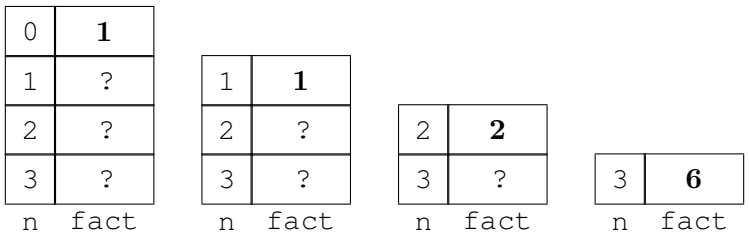


FIG. 6.2 – Dépilement des appels récursifs.

La récursivité ne marche que si on ne fait pas déborder cette pile d'appels. Imaginez que nous ayons écrit :

```
public static int fact(int n){
    if(n == 0) return 1; // cas de base
    else return n * fact(n+1);
}
```

Nous aurions rempli la pièce du sol au plafond sans atteindre la fin du calcul. On dit dans ce cas là que la fonction ne termine pas. C'est un problème fondamental de l'informatique de pouvoir *prouver* qu'une fonction (ou un algorithme) termine. On voit apparaître là une caractéristique primordiale de la programmation, qui nous rapproche de ce que l'on demande en mathématiques.

Exercice. On considère la fonction Java suivante :

```
public static int f(int n){
    if(n > 100)
        return n - 10;
    else
        return f(f(n+11));
}
```

Montrer que la fonction retourne 91 si $n \leqslant 100$ et $n - 10$ si $n > 100$.

6.2 Des exemples moins élémentaires

Encore un mot sur le programme de factorielle. Il s'agit d'un cas facile de *récursivité terminale*, c'est-à-dire que ce n'est jamais qu'une boucle `for` déguisée.

6.2.1 Écriture binaire des entiers

Prenons un cas où la récursivité apporte plus. Rappelons que tout entier strictement positif n peut s'écrire sous la forme

$$n = \sum_{i=0}^{p} b_i 2^i = b_0 + b_1 2 + b_2 2^2 + \cdots + b_p 2^p, \quad b_i \in \{0, 1\}$$

avec $p \geqslant 0$. L'algorithme naturel pour récupérer les chiffres binaires (les b_i) consiste à effectuer la division euclidienne de n par 2, ce qui nous donne $n = 2q_1 + b_0$, puis celle de q par 2, ce qui fournit $q_1 = 2q_2 + b_1$, etc. Supposons que l'on veuille afficher à l'écran les chiffres binaires de n, dans l'ordre naturel, c'est-à-dire les poids forts à gauche, comme on le fait en base 10. Pour $n = 13 = 1 + 0 \cdot 2 + 1 \cdot 2^2 + 1 \cdot 2^3$, on doit voir

```
1101
```

La fonction la plus simple à écrire est :

```
public static void binaire(int n){
    while(n != 0){
        System.out.println(n%2);
        n = n/2;
    }
```

```
        return;
    }
```

Malheureusement, elle affiche plutôt :

```
1011
```

c'est-à-dire l'ordre inverse. On aurait pu également écrire la fonction récursive :

```
    public static void binaireRec(int n){
        if(n > 0){
            System.out.print(n%2);
            binaireRec(n/2);
        }
        return;
    }
```

qui affiche elle aussi dans l'ordre inverse. Regardons une *trace* du programme, c'est-à-dire qu'on en déroule le fonctionnement, de façon analogue au mécanisme d'empilement/dépilement :

1. On affiche 13 modulo 2, c'est-à-dire b_0, puis on appelle `binaireRec`(6).
2. On affiche 6 modulo 2 ($= b_1$), et on appelle `binaireRec`(3).
3. On affiche 3 modulo 2 ($= b_2$), et on appelle `binaireRec`(1).
4. On affiche 1 modulo 2 ($= b_3$), et on appelle `binaireRec`(0). Le programme s'arrête après avoir dépilé les appels.

Il suffit de permuter deux lignes dans le programme précédent

```
    public static void binaireRec2(int n){
        if(n > 0){
            binaireRec2(n/2);
            System.out.print(n%2);
        }
        return;
    }
```

pour que le programme affiche dans le bon ordre ! Où est le miracle ? Avec la même trace :

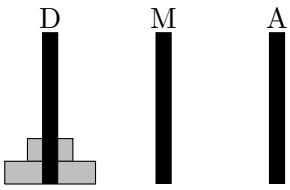
1. On appelle `binaireRec2`(6).
2. On appelle `binaireRec2`(3).
3. On appelle `binaireRec2`(1).
4. On appelle `binaireRec2`(0), qui ne fait rien.

3.' On revient au dernier appel, et maintenant on affiche $b_3 = 1 \bmod 2$;
2.' on affiche $b_2 = 3 \bmod 2$, etc.
C'est le programme qui nous a épargné la peine de nous rappeler nous-mêmes dans quel ordre nous devions faire les choses. On aurait pu par exemple les réaliser avec un tableau qui stockerait les b_i avant de les afficher. Nous avons laissé à la pile de récursivité cette gestion.

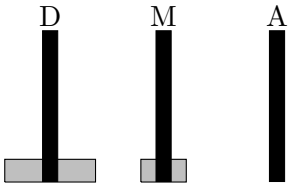
6.2.2 Les tours de Hanoi

Il s'agit là d'un jeu inspiré par une fausse légende créée par le mathématicien français Édouard Lucas. Il s'agit de trois poteaux sur lesquels peuvent coulisser des rondelles de taille croissante. Au début du jeu, toutes les rondelles sont sur le même poteau, classées par ordre décroissant de taille à partir du bas. Il s'agit de faire bouger toutes les rondelles, de façon à les amener sur un autre poteau donné. On déplace une rondelle à chaque fois, et on n'a pas le droit de mettre une grande rondelle sur une petite. Par contre, on a le droit d'utiliser un troisième poteau si on le souhaite. Nous appelerons les poteaux D (départ), M (milieu), A (arrivée).

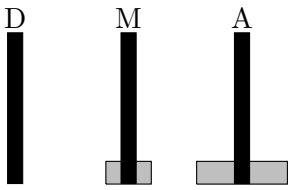
La résolution du problème avec deux rondelles se fait à la main, à l'aide des mouvements suivants. La position de départ est :



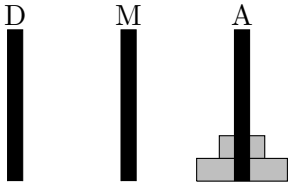
On commence par déplacer la petite rondelle sur le poteau M :



puis on met la grande en place sur le poteau A :



et enfin la petite rejoint la grande :



La solution générale s'en déduit (cf. figure 6.3). Le principe est de solidariser les $n-1$ premières rondelles. Pour résoudre le problème, on fait bouger ce tas de $n-1$ pièces du poteau D vers le poteau M (à l'aide du poteau A), puis on bouge la grande rondelle vers A, puis il ne reste plus qu'à bouger le tas de M vers A en utilisant D. Dans ce dernier mouvement, la grande rondelle sera toujours en dessous, ce qui ne créera pas de problème.

Imaginant que les poteaux D, M, A sont de type entier, on arrive à la fonction suivante :

```
public static void Hanoi(int n, int D, int M, int A){
    if(n > 0){
        Hanoi(n-1, D, A, M);
        System.out.println("On bouge "+D+" vers "+A);
        Hanoi(n-1, M, D, A);
    }
}
```

6.3 Un piège subtil : les nombres de Fibonacci

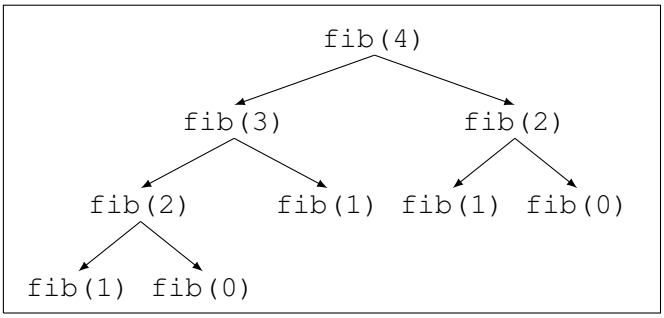
Supposons que nous voulions écrire une fonction qui calcule le n-ième terme de la suite de Fibonacci, définie par $F_0 = 0$, $F_1 = 1$ et

$$\forall n \geqslant 2, F_n = F_{n-1} + F_{n-2}.$$

Le programme naturellement récursif est simplement :

```
public static int fib(int n){
    if(n <= 1) return n; // cas de base
    else return fib(n-1)+fib(n-2);
}
```

On peut tracer l'arbre des appels pour cette fonction, qui généralise la pile des appels :



Le programme marche, il termine. Le problème se situe dans le nombre d'appels à la fonction. Si on note $A(n)$ le nombre d'appels nécessaires au calcul de F_n, il est facile de voir que ce nombre vérifie la récurrence :

$$A(n) = A(n-1) + A(n-2)$$

qui est la même que celle de F_n. Rappelons le résultat suivant :

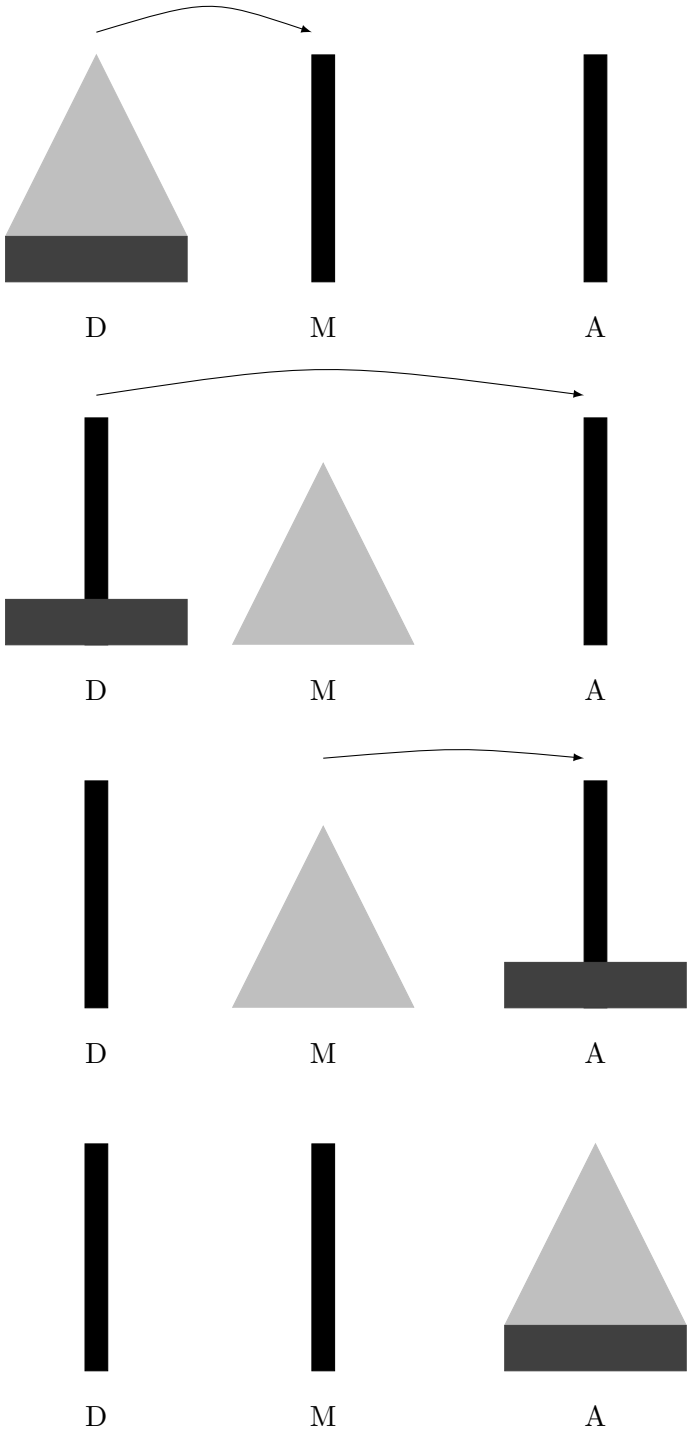


FIG. 6.3 – Les tours de Hanoi.

Proposition 1 *Avec* $\phi = (1+\sqrt{5})/2 \approx 1.618\ldots$ *(nombre d'or),* $\phi' = (1-\sqrt{5})/2 \approx -0.618\ldots$ *:*

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \phi'^n\right) = O(\phi^n).$$

(La notation $O()$ sera rappelée au chapitre suivant.)

On fait donc un nombre exponentiel d'appels à la fonction.

Une façon de calculer F_n qui ne coûte que n appels est la suivante. On calcule de proche en proche les valeurs du couple (F_i, F_{i+1}). Voici le programme :

```
public static int fib(int n){
    int i, u, v, w;

    // u = F(0); v = F(1)
    u = 0; v = 1;
    for(i = 2; i <= n; i++){
        // u = F(i-2); v = F(i-1)
        w = u+v;
        u = v;
        v = w;
    }
    return v;
}
```

De meilleures solutions pour calculer F_n vous seront données en TD.

Exercice. (Fonction d'Ackerman) On la définit de la façon suivante :

$$Ack(m,n) = \begin{cases} n+1 & \text{si } m = 0, \\ Ack(m-1,1) & \text{si } n = 0, \\ Ack(m-1, Ack(m,n-1)) & \text{sinon.} \end{cases}$$

Montrer que

$$Ack(1,n) = n+2,$$
$$Ack(2,n) = 2n+3,$$
$$Ack(3,n) = 8\cdot 2^n - 3,$$
$$Ack(4,n) = 2^{\left.2^{2^{\cdots^2}}\right\}n},$$
$$Ack(4,4) > 2^{65536} > 10^{80}$$

nombre bien plus grand que le nombre estimé de particules dans l'univers.

6.4 Fonctions mutuellement récursives

Rien n'empêche d'utiliser des fonctions qui s'appellent les unes les autres, du moment que le programme termine. Nous allons en donner maintenant des exemples.

6.4.1 Pair et impair sont dans un bateau

Commençons par un exemple un peu artificiel : nous allons écrire une fonction qui teste la parité d'un entier n de la façon suivante : 0 est pair ; si $n > 0$, alors n est pair si et seulement si $n - 1$ est impair. De même, 0 n'est pas impair, et $n > 1$ est impair si et seulement si $n - 1$ est pair. Cela conduit donc à écrire les deux fonctions :

```
// n est pair ssi (n-1) est impair
public static boolean estPair(int n){
    if(n == 0) return true;
    else return estImpair(n-1);
}

// n est impair ssi (n-1) est pair
public static boolean estImpair(int n){
    if(n == 0) return false;
    else return estPair(n-1);
}
```

qui remplissent l'objectif fixé.

6.4.2 Développement du sinus et du cosinus

Supposons que nous désirions écrire la formule donnant le développement de $\sin nx$ sous forme de polynôme en $\sin x$ et $\cos x$. On va utiliser les formules

$$\sin nx = \sin x \cos(n-1)x + \cos x \sin(n-1)x,$$

$$\cos nx = \cos x \cos(n-1)x - \sin x \sin(n-1)x$$

avec les deux cas d'arrêt : $\sin 0 = 0$, $\cos 0 = 1$. Cela nous conduit à écrire deux fonctions, qui retournent des chaînes de caractères écrites avec les deux variables `S` pour $\sin x$ et `C` pour $\cos x$.

```
public static String DeveloppperSin(int n){
    if(n == 0) return "0";
    else{
        String g = "S*(" + DeveloppperCos(n-1) + ")";
        return g + "+C*(" + DeveloppperSin(n-1) + ")";
    }
}

public static String DeveloppperCos(int n){
    if(n == 0) return "1";
    else{
        String g = "C*(" + DeveloppperCos(n-1) + ")";
        return g + "-S*(" + DeveloppperSin(n-1) + ")";
    }
}
```

L'exécution de ces deux fonctions nous donne par exemple pour $n = 3$:

```
sin(3*x)=S*(C*(C*(1)-S*(0))-S*(S*(1)+C*(0)))+C*(S*(C*(1)-S*(0))+C*(S*(1)+C*(0)))
```

Bien sûr, l'expression obtenue n'est pas celle à laquelle nous sommes habitués. En particulier, il y a trop de 0 et de 1. On peut écrire des fonctions un peu plus compliquées, qui donnent un résultat simplifié pour $n = 1$:

```
public static String DeveloppperSin(int n){
    if(n == 0) return "0";
    else if(n == 1) return "S";
    else{
        String g = "S*(" + DeveloppperCos(n-1) + ")";
        return g + "+C*(" + DeveloppperSin(n-1) + ")";
    }
}

public static String DeveloppperCos(int n){
    if(n == 0) return "1";
    else if(n == 1) return "C";
    else{
        String g = "C*(" + DeveloppperCos(n-1) + ")";
        return g + "-S*(" + DeveloppperSin(n-1) + ")";
    }
}
```

ce qui fournit :

```
sin(3*x)=S*(C*(C)-S*(S))+C*(S*(C)+C*(S))
```

On n'est pas encore au bout de nos peines. Simplifier cette expression est une tâche complexe, qui sera traitée au cours d'Informatique fondamentale.

6.5 Le problème de la terminaison

Nous avons vu combien il était facile d'écrire des programmes qui ne s'arrêtent jamais. On aurait pu rêver de trouver des algorithmes ou des programmes qui prouveraient cette terminaison à notre place. Hélas, il ne faut pas rêver.

Théorème 1 (Gödel) *Il n'existe pas de programme qui décide si un programme quelconque termine.*

Expliquons pourquoi de façon informelle, en trichant avec JAVA. Supposons que l'on dispose d'une fonction `Termine` qui prend un programme écrit en JAVA et qui réalise la fonctionnalité demandée : `Termine(fct)` retourne `true` si `fct` termine et `false` sinon. On pourrait alors écrire le code suivant :

```
public static void f(){
    while(Termine(f))
        ;
}
```

C'est un programme bien curieux. En effet, termine-t-il ? Ou bien `Termine(f)` retourne `true` et alors la boucle `while` est activée indéfiniment, donc il ne termine pas. Ou bien `Termine(f)` retourne `false` et alors le programme ne termine pas, alors que la boucle `while` n'est jamais effectuée. Nous venons de rencontrer un problème *indécidable*, celui de l'arrêt. Classifier les problèmes qui sont ou pas décidables représente une part importante de l'informatique théorique.

Chapitre 7

Introduction à la complexité des algorithmes

L'utilisateur d'un programme se demande souvent combien de temps mettra son programme à s'exécuter sur sa machine. Ce problème est concret, mais mal défini. Il dépend de la machine, du système d'exploitation, de ce qu'on fait en parallèle, etc.

D'un point de vue abstrait, il faut se demander comment fonctionne le programme, quel modèle de calcul il utilise (séquentiel, parallèle, vectoriel), etc.. On peut également se fixer un problème et se demander quelle est la méthode la plus rapide pour le résoudre. Nous allons voir dans ce chapitre comment on peut commencer à s'attaquer au problème, en s'intéressant à la complexité des algorithmes.

7.1 Complexité des algorithmes

La complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille n des données. On s'intéresse au coût exact quand c'est possible, mais également au coût moyen (que se passe-t-il si on moyenne sur toutes les exécutions du programme sur des données de taille n), au cas le plus favorable, ou bien au cas le pire. On dit que la complexité de l'algorithme est $O(f(n))$ où f est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Ceci reprend la notation mathématique classique, et signifie que le nombre d'opérations effectuées est borné par $cf(n)$, où c est une constante, lorsque n tend vers l'infini.

Considérer le comportement à l'infini de la complexité est justifié par le fait que les données d'entrée des algorithmes sont souvent de grande taille et qu'on se préoccupe surtout de la croissance de cette complexité en fonction de la taille des données. Une question systématique à se poser est : que devient le temps de calcul si on multiplie la taille des données par 2 ? De cette façon, on peut également comparer des algorithmes entre eux.

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

– Les algorithmes sous-linéaires, dont la complexité est en général en $O(\log n)$. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal n.

- Les algorithmes en complexité $O(n)$ (algorithmes linéaires) ou en $O(n \log n)$ sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de n symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

La recherche de l'algorithme ayant la plus faible complexité, pour résoudre un problème donné, fait partie du travail régulier de l'informaticien. Il ne faut toutefois pas tomber dans certains excès, par exemple proposer un algorithme excessivement alambiqué, développant mille astuces et ayant une complexité en $O(n^{1,99})$, alors qu'il existe un algorithme simple et clair de complexité $O(n^2)$. Surtout, si le gain de l'exposant de n s'accompagne d'une perte importante dans la constante multiplicative : passer d'une complexité de l'ordre de $n^2/2$ à une complexité de $10^{10} n \log n$ n'est pas vraiment une amélioration. Les critères de clarté et de simplicité doivent être considérés comme aussi importants que celui de l'efficacité dans la conception des algorithmes.

7.2 Calculs élémentaires de complexité

Donnons quelques règles simples concernant ces calculs. Tout d'abord, le coût d'une suite de deux instructions est la somme des deux coûts :

$$T(P; Q) = T(P) + T(Q).$$

Plus généralement, si l'on réalise une itération, on somme les différents coûts :

$$T(\text{for(i = 0 ; i < n ; i++) P(i) ;}) = \sum_{i=0}^{n-1} T(P(i)).$$

Si f et g sont deux fonctions positives réelles, on écrit

$$f = O(g)$$

si et seulement si le rapport f/g est borné à l'infini :

$$\exists n_0, \quad \exists K, \quad \forall n \geqslant n_0, \quad 0 \leqslant f(n) \leqslant K g(n).$$

Autrement dit, f ne croît pas plus vite que g.

Autres notations : $f = \Theta(g)$ si $f = O(g)$ et $g = O(f)$.

Les règles de calcul simples sur les O sont les suivantes (n'oublions pas que nous travaillons sur des fonctions de coût, qui sont à valeur positive) : si $f = O(g)$ et $f' = O(g')$, alors

$$f + f' = O(g + g'), \quad f f' = O(g g').$$

On montre également facilement que si $f = O(n^k)$ et $h = \sum_{i=1}^{n} f(i)$, alors $h = O(n^{k+1})$ (approximer la somme par une intégrale).

7.3 Quelques algorithmes sur les tableaux

7.3.1 Recherche du plus petit élément

Reprenons l'exemple suivant :

```
public static int plusPetit(int[] x){
    int k = 0, n = x.length;

    for(int i = 1; i < n; i++)
        // invariant : k est l'indice du plus petit
        // {\'e}l{\'e}ment de x[0..i-1]
        if(x[i] < x[k])
            k = i; // (1)
    return k;
}
```

Dans cette fonction, on exécute $n-1$ tests de comparaison. La complexité est donc $n-1 = O(n)$.

On peut également se demander combien de fois on passe dans la ligne (1) du programme. Cette question est mal posée, car elle dépend fortement des données d'entrée. Si le tableau `x` est trié dans l'ordre croissant, 0 fois ; s'il est classé dans l'ordre décroissant, n fois. On a donc prouvé le cas le meilleur et le cas le pire. On peut également prouver, mais c'est plus difficile, que la complexité en moyenne (c'est-à-dire si on fait la moyenne sur le nombre d'exécutions sur toutes les données – toutes les permutations de taille n) est $O(\log n)$.

7.3.2 Recherche dichotomique

Si `t` est un tableau d'entiers de taille n, dont les éléments sont triés (par exemple par ordre croissant) on peut écrire une fonction qui cherche si un entier donné se trouve dans le tableau. Comme le tableau est trié, on peut procéder par dichotomie : cherchant à savoir si x est dans `t[g..d[`, on calcule $m = (g+d)/2$ et on compare x à `t[m]`. Si $x = t[m]$, on a gagné, sinon on réessaie avec `t[g..m[` si $t[m] > x$ et dans `t[m+1..d[` sinon. Voici la fonction JAVA correspondante :

```
// on cherche si x est dans t[g..d[; si oui on retourne ind
// tel que t[ind] = x; si non, on retourne -1.
public static int rechercheDichotomique(long[] t, long x,
                                        int g, int d){
  int ind = -1;

  while(g < d){
      // tant que [g..d[ n'est pas vide
      int m = (g+d)/2;

      if(t[m] == x){
          ind = m;
          break; // on sort
      }
```

```
        else if(t[m] > x)
            // on cherche dans [g..m[
            d = m;
        else
            // on cherche dans [m+1..d[
            g = m+1;
    }
    return ind;
}

public static int rechercheDicho(long[] t, long x){
    return rechercheDichotomique(t, x, 0, t.length);
}
```

Notons que l'on peut écrire cette fonction sous forme récursive, ce qui la rapproche de l'idée de départ :

```
// recherche de x dans t[g..d[
public static int dichoRec(long[] t, long x, int g, int d){
    int m;

    if(g >= d) // l'intervalle est vide
        return -1;
    m = (g+d)/2;
    if(t[m] == x)
        return m;
    else if(t[m] > x)
        return dichoRec(t, x, g, m);
    else
        return dichoRec(t, x, m+1, d);
}
```

Le nombre maximal de comparaisons à effectuer pour un tableau de taille n est :

$$T(n) = 1 + T(n/2).$$

Pour résoudre cette récurrence, on écrit $n = 2^t$, ce qui conduit à

$$T(2^t) = T(2^{t-1}) + 1 = \cdots = T(1) + t$$

d'où un coût en $O(t) = O(\log n)$.

On verra dans les chapitres suivants d'autres calculs de complexité, temporelle ou bien spatiale.

7.3.3 Recherche simultanée du maximum et du minimum

L'idée est de chercher simultanément ces deux valeurs, ce qui va nous permettre de diminuer le nombre de comparaisons nécessaires. La remarque de base est qu'étant donnés deux entiers a et b, on les classe facilement à l'aide d'une seule comparaison, comme programmé ici. La fonction retourne un tableau de deux entiers, dont le premier s'interprète comme une valeur minimale, le second comme une valeur maximale.

```
// SORTIE: retourne un couple u = (x, y) avec
// x = min(a, b), y = max(a, b)
public static int[] comparerDeuxEntiers(int a, int b){
    int[] u = new int[2];

    if(a < b){
        u[0] = a; u[1] = b;
    }
    else{
        u[0] = b; u[1] = a;
    }
    return u;
}
```

Une fois cela fait, on procède récursivement : on commence par chercher les couples min-max des deux moitiés, puis en les comparant entre elles, on trouve la réponse sur le tableau entier :

```
// min-max pour t[g..d[
public static int[] minMaxAux(int[] t, int g, int d){
    int gd = d-g;

    if(gd == 1){
        // min-max pour t[g..g+1[ = t[g], t[g]
        int[] u = new int[2];

        u[0] = u[1] = t[g];
        return u;
    }
    else if(gd == 2)
        return comparerDeuxEntiers(t[g], t[g+1]);
    else{ // gd > 2
        int m = (g+d)/2;
        int[] tg = minMaxAux(t, g, m); // min-max de t[g..m[
        int[] td = minMaxAux(t, m, d); // min-max de t[m..d[
        int[] u = new int[2];

        if(tg[0] < td[0])
            u[0] = tg[0];
        else
            u[0] = td[0];
        if(tg[1] > td[1])
            u[1] = tg[1];
        else
            u[1] = td[1];
        return u;
    }
}
```

Il ne reste plus qu'à écrire la fonction de lancement :

```
public static int[] minMax(int[] t){
    return minMaxAux(t, 0, t.length);
}
```

Examinons ce qui se passe sur l'exemple

`int[] t = {1, 4, 6, 8, 2, 3, 6, 0}.`

On commence par chercher le couple min-max sur $t_g = \{1, 4, 6, 8\}$, ce qui entraîne l'étude de $t_{gg} = \{1, 4\}$, d'où $u_{gg} = (1, 4)$. De même, $u_{gd} = (6, 8)$. On compare 1 et 6, puis 4 et 8 pour finalement trouver $u_g = (1, 8)$. De même, on trouve $u_d = (0, 6)$, soit au final $u = (0, 8)$.

Soit $T(k)$ le nombre de comparaisons nécessaires pour $n = 2^k$. On a $T(1) = 1$ et $T(2) = 2T(1) + 2$. Plus généralement, $T(k) = 2T(k-1) + 2$. D'où

$$T(k) = 2^2T(k-2) + 2^2 + 2 = \cdots = 2^uT(k-u) + 2^u + 2^{u-1} + \cdots + 2$$

soit

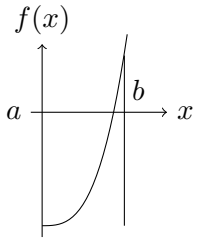
$$T(k) = 2^{k-1}T(1) + 2^{k-1} + \cdots + 2 = 2^{k-1} + 2^k - 2 = n/2 + n - 2 = 3n/2 - 2.$$

7.4 Diviser pour résoudre

C'est là un paradigme fondamental de l'algorithmique. Quand on ne sait pas résoudre un problème, on essaie de le couper en morceaux qui seraient plus faciles à traiter. Nous allons donner quelques exemples classiques, qui seront complétés par d'autres dans les chapitres suivants du cours.

7.4.1 Recherche d'une racine par dichotomie

On suppose que $f : [a, b] \to \mathbb{R}$ est continue et telle que $f(a) < 0$, $f(b) > 0$:



Il existe donc une racine x_0 de f dans l'intervalle $[a, b]$, qu'on veut déterminer de sorte que $|f(x_0)| \leqslant \varepsilon$ pour ε donné. L'idée est simple : on calcule $f((a + b)/2)$. En fonction de son signe, on explore $[a, m]$ ou $[m, b]$.

Par exemple, on commence par programmer la fonction f :

```
public static double f(double x){
    return x*x*x-2;
}
```

puis la fonction qui cherche la racine :

```
    // f(a) < 0, f(b) > 0
    public static double racineDicho(double a, double b,
                                     double eps){
        double m = (a+b)/2;
        double fm = f(m);

        if(Math.abs(fm) <= eps)
            return m;
        if(fm < 0) // la racine est dans [m, b]
            return racineDicho(m, b, eps);
        else // la racine est dans [a, m]
            return racineDicho(a, m, eps);
    }
```

7.4.2 Exponentielle binaire

Cet exemple va nous permettre de montrer que dans certains cas, on peut calculer la complexité dans le meilleur cas ou dans le pire cas, ainsi que calculer le comportement de l'algorithme en moyenne.

Supposons que l'on doive calculer x^n avec x appartenant à un groupe quelconque. On peut calculer cette quantité à l'aide de $n-1$ multiplications par x, mais on peut faire mieux en utilisant les formules suivantes :

$$x^0 = 1, x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ est pair,} \\ x(x^{(n-1)/2})^2 & \text{si } n \text{ est impair.} \end{cases}$$

Par exemple, on calcule

$$x^{11} = x(x^5)^2 = x(x(x^2)^2)^2,$$

ce qui coûte 5 multiplications (en fait 3 carrés et 2 multiplications).

La fonction évaluant x^n avec x de type `long` correspondant aux formules précédentes est :

```
public static long Exp(long x, int n){
    if(n == 0) return 1;
    else{
        if((n%2) == 0){
            long y = Exp(x, n/2);
            return y * y;
        }
        else{
            long y = Exp(x, n/2);
            return x * y * y;
        }
    }
}
```

Soit $E(n)$ le nombre de multiplications réalisées pour calculer x^n. En traduisant directement l'algorithme, on trouve que :

$$E(n) = \begin{cases} E(n/2) + 1 & \text{si } n \text{ est pair,} \\ E(n/2) + 2 & \text{si } n \text{ est impair.} \end{cases}$$

Écrivons $n > 0$ en base 2, soit $n = 2^{t-1} + \sum_{i=0}^{t-2} b_i 2^i = b_{t-1}b_{t-2}\cdots b_0 = 2^{t-1} + n'$ avec $t \geqslant 1$, $b_i \in \{0, 1\}$. On récrit donc :

$$\begin{aligned} E(n) &= E(b_{t-1}b_{t-2}\cdots b_1 b_0) = E(b_{t-1}b_{t-2}\cdots b_1) + b_0 + 1 \\ &= E(b_{t-1}b_{t-2}\cdots b_2) + b_1 + b_0 + 2 = \cdots = E(b_{t-1}) + b_{t-2} + \cdots + b_0 + (t-1) \\ &= \sum_{i=0}^{t-2} b_i + t \end{aligned}$$

On pose $\nu(n') = \sum_{i=0}^{t-2} b_i$. On peut se demander quel est l'intervalle de variation de $\nu(n')$. Si $n = 2^{t-1}$, alors $n' = 0$ et $\nu(n') = 0$, et c'est donc le cas le plus favorable de l'algorithme. À l'opposé, si $n = 2^t - 1 = 2^{t-1} + 2^{t-2} + \cdots + 1$, $\nu(n') = t - 1$ et c'est le cas le pire.

Reste à déterminer le cas moyen, ce qui conduit à estimer la quantité :

$$\overline{\nu}(n') = \frac{1}{2^{t-1}} \sum_{b_0 \in \{0,1\}} \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{t-2} \in \{0,1\}} \left(\sum_{i=0}^{t-2} b_i \right).$$

On peut récrire cela comme :

$$\overline{\nu}(n') = \frac{1}{2^{t-1}} \left(\sum b_0 + \sum b_1 + \cdots + \sum b_{t-2} \right)$$

où les sommes sont toutes indexées par les 2^{t-1} $(t-1)$-uplets formés par tous les $(b_0, b_1, \ldots, b_{t-2})$ possibles dans $\{0, 1\}^{t-1}$. Toutes ces sommes sont égales par symétrie, d'où :

$$\overline{\nu}(n') = \frac{t-1}{2^{t-1}} \sum_{b_0, b_1, \ldots, b_{t-2}} b_0.$$

Cette dernière somme ne contient que les valeurs pour $b_0 = 1$ et les b_i restant prenant toutes les valeurs possibles, d'où finalement :

$$\overline{\nu}(n') = \frac{t-1}{2^{t-1}} 2^{t-2} = \frac{t-1}{2}.$$

Autrement dit, un entier de $t-1$ bits a en moyenne $(t-1)/2$ chiffres binaires égaux à 1.

En conclusion, l'algorithme a un coût moyen

$$\overline{E}(n) = t + (t-1)/2 = \frac{3}{2}t + c$$

avec $t = \lfloor \log_2 n \rfloor$ et c une constante.

Deuxième partie

Structures de données classiques

Chapitre 8

Listes chaînées

Pour le moment, nous avons utilisé des structures de données *statiques*, c'est-à-dire dont la taille est connue à la compilation (tableaux de taille fixe) ou bien en cours de route, mais fixée une fois pour toute, comme dans

```
public static int[] f(int n){
    int[] t = new int[n];
    return t;
}
```

Cette écriture est déjà un confort pour le programmeur, confort qu'on ne trouve pas dans tous les langages.

Parfois, on ne peut pas prévoir la taille des objets avant l'exécution, et il est dans ce cas souhaitable d'avoir à sa disposition des moyens d'extension de la place mémoire qu'ils occupent. Songez par exemple à un système de fichiers sur disque. Il est hors de question d'imposer à un fichier d'avoir une taille fixe, et il faut prévoir de le construire par morceaux, en ayant la possibilité d'en rajouter si besoin est.

C'est là qu'on trouve un intérêt à introduire des structures de données dont la taille augmente (ou diminue) de façon *dynamique*, c'est-à-dire à l'exécution (par rapport à statique, au moment de la compilation). C'est le cas des *listes*, qui vont être définies récursivement de la façon imagée suivante : une liste est soit vide, soit contient une information, ainsi qu'une autre liste. Pour un fichier, on pourra allouer un bloc, qui contiendra les caractères du fichier, ainsi que la place pour un lien vers une autre zone mémoire potentielle qu'on pourra rajouter si besoin pour étendre le fichier.

Dans certains langages, comme Lisp, tous les objets du langage (y compris les méthodes) sont des listes, appelées dans ce contexte des S-expressions.

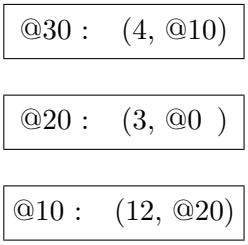
En Java, le lien entre *cellules* (ces blocs contenant l'information) ne sera rien d'autre que la référence du bloc suivant en mémoire. Nous allons dans ce chapitre utiliser essentiellement des listes d'entiers pour simplifier notre propos, et nous donnerons quelques exemples d'utilisation à la fin du chapitre.

8.1 Opérations élémentaires sur les listes

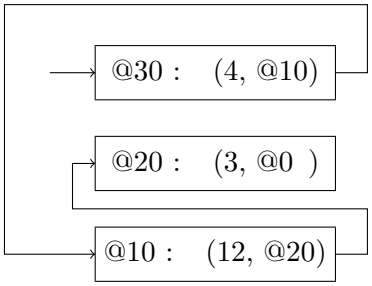
8.1.1 Création

Une *liste* d'entiers est une structure abstraite qu'on peut voir comme une suite de *cellules* contenant des informations et reliées entre elles. Chaque cellule contient un couple formé d'une donnée, et de la référence à la case suivante, comme dans un jeu de piste. C'est le système d'exploitation de l'ordinateur qui retourne les adresses des objets.

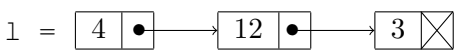
Considérons l'exemple suivant : le début de la liste se trouve à l'adresse @30, à laquelle on trouve le couple (4, @10) où 4 est la donnée, @10 l'adresse de la cellule suivante. On continue en trouvant à l'adresse @10 le couple (12, @20) qui conduit à (3, @0) où l'adresse spéciale @0 sert d'adresse de fin :



Cette représentation est très proche de ce qui se passe en mémoire, mais elle n'est pas très visuelle. Aussi a-t-on l'habitude de rajouter des flèches pour lier les cellules, la première flèche désignant le point d'entrée dans le jeu de piste :



Il est parfois commode d'utiliser ainsi les adresses explicites dans la mémoire pour bien comprendre les mécanismes qui sont à l'œuvre. Dans la plupart des cas, et quand on a bien saisi le mécanisme, on peut abstraire les listes jusqu'à obtenir :



ou de façon encore plus stylisée :

```
l = @30:4 -> @10:12 -> @20:3 -> @0
```

ou sans les adresses :

```
l = 4 -> 12 -> 3 -> null
```

avec la convention JAVA que **null** représente l'adresse @0. On dit que le case contenant 4 pointe sur la case contenant 12, etc.

Faire des dessins est primordial quand on veut comprendre les structures de données récursives !

Comment déclare-t-on une liste chaînée en JAVA ? Très simplement : une liste est la référence d'une cellule, cellule qui contient la donnée et la référence de la cellule suivante. On donne ci-dessous la syntaxe et la réalisation de la structure dessinée ci-dessus :

```
public class Liste{
    int contenu;
    Liste suivant;

    public Liste(int c, Liste suiv){
        this.contenu = c;
        this.suivant = suiv;
    }
    public static void main(String[] args){
        Liste l; // l = null = @0

        l = new Liste(3, null); // l = @20       (1)
        l = new Liste(12, l);   // l = @10       (2)
        l = new Liste(4, l);    // l = @30       (3)
    }
}
```

La liste `l` est construite très simplement en rajoutant des éléments en tête par l'utilisation de **new**. Comme dans l'écriture des méthodes récursives, il est impératif de ne pas oublier le cas de base, ici de bien démarrer à partir de la cellule vide.

Examinons ce qui se passe ligne à ligne. Quand on crée la première liste (ligne 1), on obtient :

```
l = @20:3 -> @0
```

La valeur de `l` est la référence en mémoire de la cellule construite, qui vaut ici `@20`. Après la ligne (2), on obtient :

```
l = @10:12 -> @20:3 -> @0
```

et `l` vaut `@10`. À la fin du programme, on a :

```
l = @30:4 -> @10:12 -> @20:3 -> @0
```

et `l` contient `@30`.

8.1.2 Affichage

Comment faire pour afficher une liste à l'écran ? Il faut reproduire le parcours du jeu de piste dans la mémoire : tant qu'on n'a pas atteint la case spéciale, on affiche le contenu de la case courante, puis on passe à la case suivante :

```
public static void afficherContenu(Liste l){
    Liste ll = l;

    while(ll != null){
        TC.print(ll.contenu + " -> ");
        ll = ll.suivant;
    }
    TC.println("null");
}
```

Dans ce premier programme, nous avons été ultra-conservateurs, en donnant l'impression qu'il ne faut pas toucher à la variable `l`. Bien sûr, il n'en est rien, puisqu'on passe une référence à une liste, *par valeur* et non par variable. Ainsi, le code suivant est-il plus idiomatique et compact encore :

```
public static void afficherContenu(Liste l){
    while(l != null){
        TC.print(l.contenu + " -> ");
        l = l.suivant;
    }
    TC.println("null");
}
```

La liste originale n'est pas perdue, puisqu'on a juste recopié son adresse dans le `l` de la méthode `afficherContenu`.

8.1.3 Longueur

Nous pouvons maintenant utiliser le même schéma pour calculer la longueur d'une liste, c'est-à-dire le nombre de cellules de la liste. Par convention, la liste vide a pour longueur 0. Nous pouvons écrire itérativement :

```
public static int longueur(Liste l){
    int lg = 0;

    while(l != null){
        lg++;
        l = l.suivant;
    }
    return lg;
}
```

On peut écrire également ce même calcul de façon récursive, qui a l'avantage d'être plus compacte, mais également de coller de plus près à la définition récursive de la liste : une liste est ou bien vide, ou bien elle contient au moins une cellule, sur laquelle on peut effectuer une opération avant d'appliquer la méthode au reste de la liste. Cela donne la fonction

```
public static int longueurRec(Liste l){
    if(l == null) // test d'arrêt
```

```
            return 0;
        else
            return 1 + longueurRec(l.suivant);
    }
```

8.1.4 Le i-ème élément

Le schéma de la méthode précédente est très général. On accède aux éléments d'une liste de façon itérative, en commençant par la tête. Si on veut renvoyer l'entier contenu dans la i-ème cellule (avec $i \geqslant 0$, 0 pour l'élément en tête de liste), on est obligé de procéder comme suit :

```
    // on suppose i < longueur de la liste
    public static int ieme(Liste l, int i){
        for(int j = 0; j < i; j++)
            l = l.suivant;
        return l.contenu;
    }
```

On peut arguer que le code précédent n'est pas très souple, puisqu'il demande à l'utilisateur de savoir par avance que l'indice est plus petit que la longueur de la liste, qui n'est pas toujours connue. Une manière plus prudente de procéder est la suivante : on décide que le i-ème élément d'une liste est **null** dans le cas où i est plus grand que la longueur de la liste. On écrit alors :

```
    // on suppose i < longueur de la liste
    public static int ieme(Liste l, int i){
        if(l == null)
            return null;
        if(i == 0) // on a demandé la tête de la liste
            return l.contenu;
        else
            // le i ème élément de la liste
            // est le (i-1) de l.suivant
            return ieme(l.suivant, i-1);
    }
```

8.1.5 Ajouter des éléments en queue de liste

Le principe de la méthode est le suivant : si la liste est vide, on crée une nouvelle cellule que l'on retourne ; sinon, on cherche la fin de la liste et on rajoute une nouvelle cellule à la fin.

```
    public static Liste ajouterEnQueue(Liste l, int c){
        Liste ll;

        if(l == null)
            return new Liste(c, null);
        // l a un suivant
```

```
        ll = l; // cherchons la dernière cellule
        while(ll.suivant != null)
            ll = ll.suivant;
        ll.suivant = new Liste(c, null);
        return l; // pas ll...
    }
```

Regardons ce qui se passe quand on ajoute 5 à la fin de la liste

```
l = @30:4 -> @10:12 -> @20:3 -> null
```

L'exécution pas à pas donne :

```
ll = @30; // ll.suivant = @10
ll = ll.suivant // valeur = @10; ll.suivant = @20
```

comme `ll.suivant` vaut **null**, on remplace alors cette valeur par la référence d'une nouvelle cellule contenant 5 :

```
ll.suivant = @60
```

et le schéma final est :

```
l = @30:4 -> @10:12 -> @20:3 -> @60:5 -> null
```

La version itérative de la méthode est complexe à lire. La version récursive est beaucoup plus compacte :

```
    public static Liste ajouterEnQueueRec(Liste l, int c){
        if(l == null)
            return new Liste(c, null);
        else{
            // les modifications vont avoir lieu dans la
            // partie "suivante" de la liste
            l.suivant = ajouterEnQueueRec(l.suivant, c);
            return l;
        }
    }
}
```

8.1.6 Copier une liste

Une première version de la copie paraît simple à écrire :

```
    public static Liste copier(Liste l){
        Liste ll = null;

        while(l != null){
            ll = new Liste(l.contenu, ll);
            l = l.suivant;
        }
        return ll;
    }
```

mais cela copie à l'envers. Si

```
l = 1 -> 2 -> 3 -> null
```

la méthode crée :

```
ll = 3 -> 2 -> 1 -> null
```

Pour copier à l'endroit, il est beaucoup plus compact de procéder récursivement : si on veut copier `l=(c, suivant)`, on copie à l'endroit la liste `suivant` à laquelle on rajoute `c` en tête :

```
public static Liste copierALEndroit(Liste l){
    if(l == null)
        return null;
    else
        return
          new Liste(l.contenu, copierALEndroit(l.suivant));
}
```

Ex. Écrire la méthode de façon itérative.

8.1.7 Suppression de la première occurrence

Le problème ici est de fabriquer une liste `l2` avec le contenu de la liste `l1` dont on a enlevé la première occurrence d'une valeur donnée, dans le même ordre que la liste de départ. On peut subdiviser le problème en deux sous-cas. Le premier correspond à celui où la cellule concernée est en tête de liste. Ainsi :

```
l1 = @10:4 -> @30:12 -> @20:3 -> null
```

dont on enlève 4 va donner :

```
l2 = @50:12 -> @60:3 -> null
```

Dans le cas général, l'élément se trouve au milieu :

```
l1 = @10:4 -> @30:12 -> @20:3 -> null
```

si on enlève 3 :

```
l2 = @50:4 -> @60:12 -> @20 -> null
```

Le principe est de copier le début de liste, puis d'enlever la cellule, puis de copier la suite.

On peut programmer récursivement ou itérativement (bon exercice !). La méthode récursive peut s'écrire :

```
public static Liste supprimerRec(Liste l, int c){
    if(l == null)
        return null;
    if(l.contenu == c)
        return copierALEndroit(l.suivant);
```

```
        else
            return
              new Liste(l.contenu, supprimerRec(l.suivant, c));
    }
```

Ex. Modifier la méthode précédente de façon à enlever toutes les occurrences de `c` dans la liste.

8.2 Interlude : tableau ou liste ?

On utilise un tableau quand :
- on connaît la taille (ou un majorant proche) à l'avance ;
- on a besoin d'accéder aux différentes cases dans un ordre quelconque (accès direct à `t[i]`).

On utilise une liste quand :
- on ne connaît pas la taille *a priori* ;
- on n'a pas besoin d'accéder souvent au i-ème élément ;
- on ne veut pas gaspiller de place.

Bien sûr, quand on ne connait pas la taille à l'avance, on peut se contenter de faire évoluer la taille du tableau, comme dans l'exemple caricatural ci-dessous où on agrandit le tableau d'une case à chaque fois, en créant un nouveau tableau dans lequel on transvase le contenu du tableau précédent :

```
    public static int[] creerTableau(int n){
        int[] t, tmp;

        t = new int[1];
        t[0] = 1;
        for(int i = 2; i <= n; i++){
            tmp = new int[i];
            for(int j = 0; j < t.length; j++)
                tmp[j] = t[j];
            tmp[i-1] = i;
            t = tmp;
        }
        return t;
    }
```

Il est clair qu'on doit faire là n allocations de tableaux, et que les recopies successives vont coûter environ n^2 opérations. On peut faire un peu mieux, par exemple en allouant un tableau de taille $2n$ à chaque fois.

8.3 Partages

Les listes sont un moyen abstrait de parcourir la mémoire. On peut imaginer des jeux subtils qui n'altèrent pas le contenu des cases, mais la manière dont on interprète le parcours. Donnons deux exemples.

8.3.1 Insertion dans une liste triée

Il s'agit ici de créer une nouvelle cellule, qu'on va intercaler entre deux cellules existantes. Considérons la liste :

```
l = @10:4 -> @30:12 -> @20:30 -> null
```

qui contient trois cellules avec les entiers 4, 12, 30. On cherche à insérer l'entier 20, ce qui conduit à créer la liste

```
ll = @50:20 -> null
```

On doit insérer cette liste entre la deuxième et la troisième cellule. Graphiquement, on sépare le début et la fin de la liste en :

```
l = @10:4 -> @30:12 ->                    @20:30 -> null
```

décrochant chacun des wagons qu'on raccroche à la nouvelle cellule

```
l = @10:4 -> @30:12 ->      [ @50:20 -> ]      @20:30 -> null
```

d'où :

```
l = @10:4 -> @30:12 -> @50:20 -> @20:30 -> null
```

On programme selon ce principe une méthode qui insère un entier dans une liste déjà triée dans l'ordre croissant. Donnons le programme JAVA correspondant :

```
// on suppose l triée dans l'ordre croissant
public static Liste insertion(Liste l, int c){
    if(l == null)
        return new Liste(c, null);
    // l = [contenu, suivant]
    if(l.contenu < c){
        // on doit insérer c dans suivant
        l.suivant = insertion(l.suivant, c);
        return l;
    }
    else
        // on met c en tête car <= l.contenu
        return new Liste(c, l);
}
```

Ex. Écrire une méthode de tri d'un tableau en utilisant la méthode précédente.

8.3.2 Inverser les flèches

Plus précisément, étant donnée une liste

```
l = @10:(4, @30) -> @30:(12, @20) -> @20:(3, @0) -> @0
```

on veut modifier les données de sorte que `l` désigne maintenant :

```
l = @20:(3, @30) -> @30:(12, @10) -> @10:(4, @0) -> @0
```

Le plus simple là encore, est de penser en termes récursifs. Si `l` est **`null`**, on retourne **`null`**. Sinon, `l = (c, l.suivant)`, on retourne `l.suivant` et on met `c` à la fin. Le code est assez complexe, et plus simple, une fois n'est pas coutume, sous forme itérative :

```
    public static Liste inverser(Liste l){
        Liste lnouv = null, tmp;

        while(l != null){
            // lnouv contient le début de la liste inversée
            tmp = l.suivant; // on protège
            l.suivant = lnouv; // on branche
            lnouv = l; // on met à jour
            l = tmp; // on reprend avec la suite de la liste
        }
        return lnouv;
    }
```

8.4 Exemple de gestion de la mémoire au plus juste

Le problème est le suivant. Dans les logiciels de calcul formel comme MAPLE, on travaille avec des polynômes, parfois gros, et on veut gérer la mémoire au plus juste, c'est-à-dire qu'on ne veut pas stocker les coefficients du polynôme $X^{10000}+1$ dans un tableau de 10001 entiers, dont deux cases seulement serviront. Aussi adopte-t-on une représentation *creuse* avec une liste de monômes qu'on interprète comme une somme.

On utilise ainsi une liste de monômes par ordre décroissant du degré. Le type de base et son constructeur explicite sont alors :

```
public class PolyCreux{
    int degre, valeur;
    PolyCreux suivant;

    PolyCreux(int d, int v, PolyCreux p){
        this.degre = d;
        this.valeur = v;
        this.suivant = p;
    }
}
```

Pour tester et déboguer, on a besoin d'une méthode d'affichage :

```
    public static void Afficher(PolyCreux p){
        while(p != null){
            System.out.print("("+p.valeur+")*X^"+p.degre);
            p = p.suivant;
            if(p != null) System.out.print("+");
        }
        System.out.println();
```

```
    }
```

On peut alors tester via :

```
public class TestPolyCreux{
    public static void main(String[] args){
        PolyCreux p, q;

        p = new PolyCreux(0, 1, null);
        p = new PolyCreux(17, -1, p);
        p = new PolyCreux(100, 2, p);
        PolyCreux.Afficher(p);

        q = new PolyCreux(1, 3, null);
        q = new PolyCreux(17, 1, q);
        q = new PolyCreux(50, 4, q);
        PolyCreux.Afficher(q);
    }
}
```

Et on obtient :

```
(2)*X^100+(-1)*X^17+(1)*X^0
(4)*X^50+(1)*X^17+(3)*X^1
```

Ex. Écrire une méthode `Copier` qui fabrique une copie d'un polynôme creux dans le même ordre.

Comment procède-t-on pour l'addition ? On doit en fait créer une troisième liste représentant la fusion des deux listes, en faisant attention au degré des monômes qui entrent en jeu, et en retournant une liste qui préserve l'ordre des degrés des monômes. La méthode implantant cette idée est la suivante :

```
  public static PolyCreux Additionner(PolyCreux p,PolyCreux q){
      PolyCreux r;

      if(p == null) return Copier(q);
      if(q == null) return Copier(p);
      if(p.degre == q.degre){
          r = Additionner(p.suivant, q.suivant);
          return new PolyCreux(p.degre, p.valeur + q.valeur,r);
      }
      else{
          if(p.degre < q.degre){
              r = Additionner(p, q.suivant);
              return new PolyCreux(q.degre, q.valeur, r);
          }
          else{
              r = Additionner(p.suivant, q);
              return new PolyCreux(p.degre, p.valeur, r);
          }
```

```
        }
    }
```

Sur les deux polynômes donnés, on trouve :

```
(2)*X^100+(4)*X^50+(0)*X^17+(3)*X^1+(1)*X^0
```

Ex. Écrire une méthode qui nettoie un polynôme, c'est-à-dire qui enlève les monômes de valeur 0, comme dans l'exemple donné ci-dessus.

Chapitre 9

Arbres

Les arbres sont une structure très classique utilisée en informatique, pour ses propriétés de représentation et de compaction. Nous allons donner les propriétés de base de ces objets.

9.1 Arbres binaires

Un *arbre binaire* permet de stocker des informations hiérarchiques. Un arbre binaire est défini comme étant soit vide, soit muni d'une racine et d'éventuels fils gauche et droit, qui sont eux-mêmes des arbres. Ce sera le cas de l'arbre de la figure 9.1.

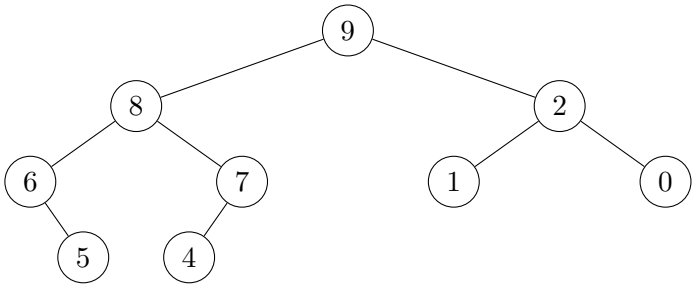


Fig. 9.1 – Exemple d'arbre.

Les éléments cerclés sont appelés des *nœuds* de l'arbre. Le nœud initial est appelé *racine*. Les nœuds terminaux sont des *feuilles*. Tous les nœuds sauf la racine ont un père, les feuilles n'ont pas de fils.

9.1.1 Représentation en machine

Dans un tableau

Une des représentations possibles est celle d'un tableau des pères, ou à chaque nœud on associe son père (qui est nécessairement unique). L'arbre de la figure 9.1 serait ainsi

codé par

0	1	2	4	5	6	7	8	9
2	2	9	7	6	8	8	9	−1

où −1 est un codage du fait que 9 est la racine de l'arbre. Ce type d'implantation ne respecte pas l'ordre gauche/droite et n'est pas utilisable dans tous les cas de figure. De plus, il faudrait connaître le nombre de nœuds de l'arbre.

De façon dynamique

Pour économiser de la place, il est plus pratique de définir une structure dynamique qui implante un arbre. La classe correspondante est

```
public class Arbre{
    private int racine;
    private Arbre gauche, droit;

    public Arbre(int r, Arbre g, Arbre d){
        this.racine = r;
        this.gauche = g;
        this.droit = d;
    }
}
```

9.1.2 Complexité

On appelle *hauteur* d'un arbre le nombre maximal de nœuds trouvés dans un chemin entre la racine et une feuille. Si l'arbre contient n nœuds, sa hauteur est majorée par n (cas d'un arbre filiforme – une liste). La borne inférieure est $\log_2 n$, car c'est la hauteur d'un arbre *complet*, dont toutes les feuilles sont à la même hauteur. Il existe des techniques pour *équilibrer* les arbres, mais nous ne les détaillerons pas ici. Il faut savoir qu'en moyenne, une permutation aléatoire de n éléments est stockable dans un arbre de hauteur $O(\log_2 n)$.

9.1.3 Les trois parcours classiques

On définit classiquement trois *parcours* d'arbre, qui permettent de considérer chaque nœud dans un ordre particulier. L'*ordre préfixe* considère d'abord la racine, puis les deux sous-arbres gauche et droit dans cet ordre ; l'*ordre infixe* parcourt le sous-arbre gauche, la racine, le sous-arbre droit ; l'*ordre postfixe* considère le sous-arbre gauche, le droit, puis la racine. Le code Java correspondant est :

```
    public static void afficherPrefixe(Arbre a){
        if(a != null){
            System.out.print(a.racine);
            afficherPrefixe(a.gauche);
            afficherPrefixe(a.droit);
        }
    }
```

```
    public static void afficherInfixe(Arbre a){
        if(a != null){
            afficherInfixe(a.gauche);
            System.out.print(a.racine);
            afficherInfixe(a.droit);
        }
    }

    public static void afficherPostfixe(Arbre a){
        if(a != null){
            afficherPostfixe(a.gauche);
            afficherPostixe(a.droit);
            System.out.print(a.racine);
        }
    }

}
```

La classe

```
public class TesterArbre{

    public static void main(String[] args){
        Arbre a = new Arbre(1,
                            new Arbre(0,
                                      new Arbre(3, null, null),
                                      null),
                            new Arbre(2, null, null));
        Arbre.afficherInfixe(a); System.out.println();
        Arbre.afficherPrefixe(a); System.out.println();
        Arbre.afficherPostfixe(a); System.out.println();
    }
}
```

nous affichera :

```
1032
3012
3021
```

9.2 Arbres généraux

Dans certaines applications, il y a plus que deux fils associés à un nœud.

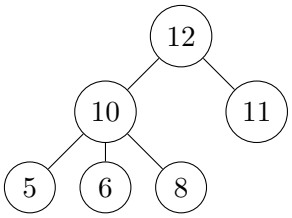
9.2.1 Définitions

Un arbre général est défini comme étant $\emptyset$ ou bien une structure contenant une racine, ainsi qu'un ensemble d'arbres $(S_1, S_2, \ldots, S_n)$ attachés à la racine r. On peut le noter symboliquement : $A = (r, S_1, S_2, \ldots, S_n)$.

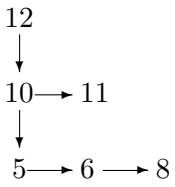
9.2.2 Représentation en machine

On peut par exemple gérer un tableau de fils en lieu et place de gauche et droit, ou encore une liste de fils.

On peut également utiliser la représentation *fils gauche – frère droit* qui *de facto* utilise un arbre binaire. Par exemple,



sera codé en machine sous la forme de l'arbre binaire :



Nous laissons en exercice l'écriture des fonctions qui permettent de coder les arbres n-aires de cette façon, en particulier les parcours.

9.3 Exemples d'utilisation

9.3.1 Expressions arithmétiques

On considère ici des expressions arithmétiques faisant intervenir des variables $a..z$, des entiers, des opérateurs binaires $+$, $-$, $*$, $/$, comme par exemple l'expression $x + y/(2z + 1) + t$. Toute expression de ce type peut être représentée par un arbre binaire (de façon non unique), cf. figure 9.2, ou un arbre n-aire (comme dans MAPLE, cf. figure 9.3.

Objets de base, première opérations

Le début de la classe est

```
public class Expression{
    char type;
    int n;
    Expression filsg, filsd;

    public Expression(char type, int n,
                      Expression fg, Expression fd){
        this.type = type;
        this.n = n;
```

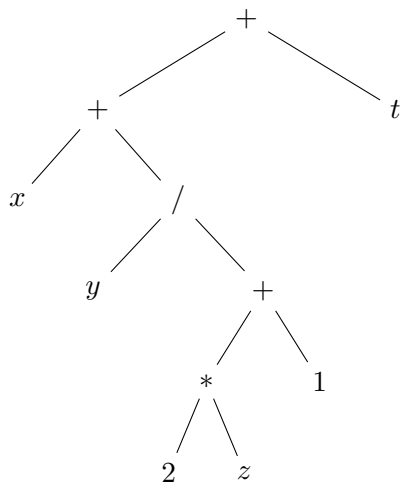


FIG. 9.2 – Arbre binaire pour l'expression $x + y/(2z + 1) + t$.

```
        this.filsg = fg;
        this.filsd = fd;
    }

    public static void afficherPrefixe(Expression e){
        if(e != null){
            System.out.print("(");
            if(e.type == 'I')
                System.out.print(e.n + " ");
            else
                System.out.print(e.type + " ");
            afficherPrefixe(e.filsg);
            afficherPrefixe(e.filsd);
            System.out.print(")");
        }
    }
}
```

Passons aux quatre opérations :

```
    public static Expression additionner(Expression e1,
                                         Expression e2){
        return new('+', 0, e1, e2);
    }

    public static Expression soustraire(Expression e1,
                                        Expression e2){
        return new('-', 0, e1, e2);
    }
```

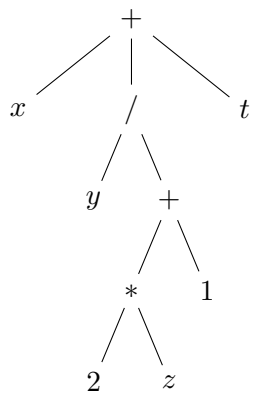


FIG. 9.3 – Arbre n-naire pour l'expression $x + y/(2z + 1) + t$.

```
    public static Expression multiplier(Expression e1,
                                        Expression e2){
        return new('*', 0, e1, e2);
    }

    public static Expression diviser(Expression e1,
                                     Expression e2){
        return new('/', 0, e1, e2);
    }
```

Un exemple d'utilisation sera :

```
public class TestExpr{
    public static void main(String[] args){
        Expression e, e1;

        e1 = new Expression('I', 2, null, null);
        e = new Expression('z', 0, null, null);
        e = Expression.multiplier(e1, e);
        e1 = new Expression('I', 1, null, null);
        e = Expression.additionner(e, e1);
        e1 = new Expression('y', 0, null, null);
        e = Expression.diviser(e1, e);
        e1 = new Expression('x', 0, null, null);
        e = Expression.additionner(e1, e);
        e1 = new Expression('t', 0, null, null);
        e = Expression.additionner(e, e1);
        Expression.afficherPrefixe(e);
  }
}
```

ce qui nous donne

```
(+ (+ (x )(/ (y )(+ (* (2 )(z ))(1 ))))(t ))
```

Substitution d'expressions

L'intérêt des arbres apparaît clairement dès qu'on veut substituer une variable par une expression quelconque. En clair, il suffit de brancher l'arbre de substitution partout où la variable apparaît.

```
public static void substituer(Expression e, char v,
                              Expression f){
    if(e != null){
        if(e.type == v){
            e.type = f.type;
            e.n = f.n;
            e.filsg = f.filsg;
            e.filsd = f.filsd;
        }
        substituer(e.filsg, v, f);
        substituer(e.filsd, v, f);
    }
}
```

L'exemple

```
e1 = new Expression('x', 0, null, null);
e1 = Expression.multiplier(e1, e1);
Expression.substituer(e, 'z', e1);
```

nous donne

```
(+ (+ (x )(/ (y )(+ (* (2 )(* (x )(x )))(1 ))))(t ))
```

Remplacer une variable par une valeur s'appelle *instantiation*. C'est une variante de la substitution générale :

```
public static void instancier(Expression e,
                              char v, int n){
    if(e != null){
        if(e.type == v){
            e.type = 'I';
            e.n = n;
        }
        instancier(e.filsg, v, n);
        instancier(e.filsd, v, n);
    }
}
```

L'exécution de

```
Expression.instancier(e, 'x', 5);
Expression.instancier(e, 'y', 2);
Expression.instancier(e, 't', 3);
Expression.afficherPrefixe(e);
```

donne

```
(+ (+ (5 )(/ (2 )(+ (* (2 )(* (5 )(5 )))(1 ))))(3 ))
```

Il ne nous reste plus qu'à écrire le code d'évaluation d'une expression numérique :

```
    public static int evaluer(Expression e){
        if(e == null)
            return 0;
        switch(e.type){
        case 'I':
            return e.n;
        case '+':
            return evaluer(e.filsg) + evaluer(e.filsd);
        case '*':
            return evaluer(e.filsg) * evaluer(e.filsd);
        case '-':
            return evaluer(e.filsg) - evaluer(e.filsd);
        case '/':
            return evaluer(e.filsg) / evaluer(e.filsd);
        default:
            System.out.println("Erreur");
        }
        return 0;
    }
```

9.3.2 Arbres binaires de recherche

Nous allons traiter un exemple important, celui de l'*arbre binaire* de recherche, soit $A = (r, G, D)$ vérifiant la propriété que tout élément du sous-arbre gauche G est plus petit que r, lui-même plus petit que tout élément du sous-arbre droit D. Un parcourt infixe d'un tel arbre fournit la liste des valeurs des nœuds des arbres dans l'ordre croissant. Si l'arbre est équilibré, le temps d'insertion ou de recherche sera en $O(\log_2 n)$, c'est-à-dire rapide.

Nous allons modifier notre classe de base pour gérer de tels arbres.

```
public class ABR{
    private int racine;
    private ABR gauche, droit;

    public ABR(int r, ABR g, ABR d){
        this.racine = r;
        this.gauche = g;
        this.droit = d;
    }

    public static ABR inserer(ABR a, int x){
        if(a == null)
            // on crée un nouvel ABR
            return new ABR(x, null, null);
```

```
        else if(x <= a.racine){
            // on insère dans le sous-arbre gauche
            a.gauche = inserer(a.gauche, x);
            return a;
        }
        else // x > a.racine;
            // on insère dans le sous-arbre droit
            a.droit = inserer(a.droit, x);
        return a;
    }
}
```

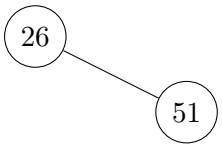
La seule nouveauté est la fonction d'insertion, qui est plus complexe. Celle-ci cherche à insérer en respectant la notion d'ABR. Le cas de base de la récursion est la création d'une feuille. Quand l'arbre a une racine, on effectue l'insertion du côté où x doit se trouver pour respecter la propriété. Cette fonction s'écrit beaucoup plus facilement de façon récursive que de manière itérative.

À titre d'exemple, nous allons créer un ABR à partir du tableau

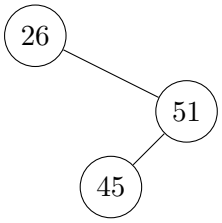
```
int[] t = {26, 51, 45, 57, 95, 87, 1, 67, 96, 91};
```

qui va nous permettre de simuler une insertion dynamique de ses éléments, l'un après l'autre. Le premier nœud est

On insère alors 51 dans le sous-arbre droit de 26 :



Pour insérer le 45, on part dans le sous-arbre droit de 26, puis dans le sous-arbre gauche de 51 :



On procède ainsi de proche en proche pour aboutir finalement à la figure 9.4.

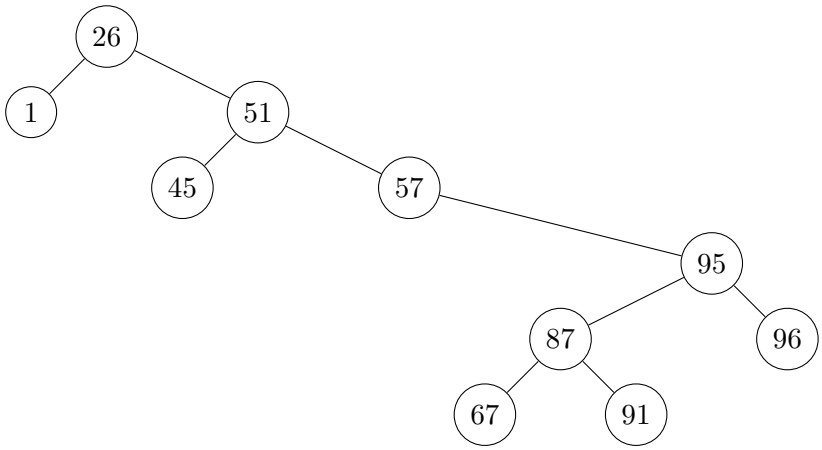


FIG. 9.4 – Exemple d'arbre binaire de recherche.

9.3.3 Les tas

Nous allons étudier ici une structure de données particulière qui *garantit* que le temps de recherche du plus grand élément d'un tableau de n éléments se fasse en temps $O(\log_2 n)$.

On dit qu'un tableau `t[0..TMAX]` possède la *propriété de tas* si pour tout $i >$ 0, `t[i]` (un parent) est plus grand que ses deux enfants gauche `t[2*i]` et droit `t[2*i+1]`. Nous supposons ici que les tas sont des tas d'entiers, mais il serait facile de modifier cela.

Le tableau `t = {0, 9, 8, 2, 6, 7, 1, 0, 3, 5, 4}` (rappelons que `t[0]` ne nous sert à rien ici) a la propriété de tas, ce que l'on vérifie à l'aide du dessin suivant :

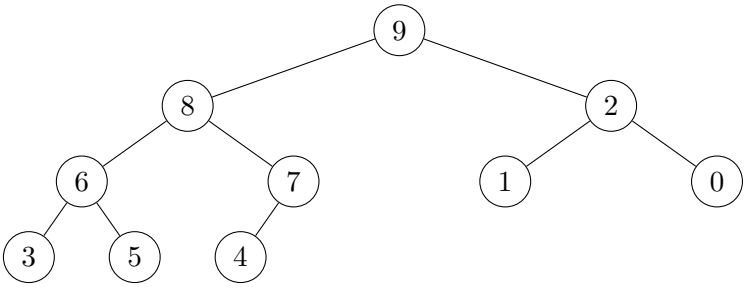


FIG. 9.5 – Exemple de tas.

Bien que nous puissions nous contenter d'utiliser un tableau ordinaire, il est plus intéressant d'utiliser une classe spéciale, que nous appelerons `Tas`, et qui nous permettra de ranger les éléments de façon dynamique en gérant un indice `n`, qui désignera le nombre d'éléments présents dans le tas :

```
public class Tas{
    private int[] t; // la partie utile est t[1..tmax]
```

```
    private int n; // indice du dernier élément

    public static Tas creer(int tmax){
        Tas tas = new Tas();

        tas.t = new int[tmax+1];
        tas.n = 0;
        return tas;
    }

    public boolean estVide(){
        return tas.n == 0;
    }
}
```

La première fonction que l'on peut utiliser est celle qui teste si un tas a bien la propriété qu'on attend :

```
    public static boolean estTas(Tas tas){
        for(int i = tas.n; i > 1; i--)
            if(tas.t[i] > tas.t[i/2])
                return false;
        return true;
    }
```

Proposition 2 *Soit $n \geqslant 1$ et t un tas. On définit la* hauteur *du tas (ou de l'arbre) comme l'entier h tel que $2^h \leqslant n < 2^{h+1}$. Alors*

(i) L'arbre a $h+1$ niveaux, l'élément $t[1]$ se trouvant au niveau 0.

(ii) Chaque niveau, $0 \leqslant \ell < h$, est stocké dans $t[2^\ell..2^{\ell+1}[$ et comporte ainsi 2^ℓ éléments. Le dernier niveau ($\ell = h$) contient les éléments $t[2^h..n]$.

(iii) Le plus grand élément se trouve en `t[1]`.

Exercice. Écrire une fonction qui à l'entier $i \leqslant n$ associe son niveau dans l'arbre.

On se sert d'un tas pour implanter facilement une *file de priorité*, qui permet de gérer des clients qui arrivent, mais avec des priorités qui sont différentes, contrairement au cas de la poste. À tout moment, on sait qu'on doit servir le client `t[1]`. Il reste à décrire comment on réorganise le tas de sorte qu'à l'instant suivant, le client de plus haute priorité se retrouve en `t[1]`. On utilise de telles structures pour gérer les impressions en Unix, ou encore dans l'ordonnanceur du système.

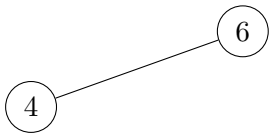
Dans la pratique, le tas se comporte comme un lieu de stockage dynamique où entrent et sortent des éléments. Pour simuler ces mouvements, on peut partir d'un tas déjà formé `t[1..n]` et insérer un nouvel élément x. S'il reste de la place, on le met temporairement dans la case d'indice $n+1$. Il faut vérifier que la propriété est encore satisfaite, à savoir que le père de `t[n+1]` est bien supérieur à son fils. Si ce n'est pas le cas, on les permute tous les deux. On n'a pas d'autre test à faire, car au cas où `t[n+1]` aurait eu un frère, on savait déjà qu'il était inférieur à son père. Ayant permuté père et fils, il se peut que la propriété de tas ne soit toujours pas vérifiée, ce qui fait que l'on doit remonter vers l'ancêtre du tas éventuellement.

Illustrons tout cela sur un exemple, celui de la création d'un tas à partir du tableau :

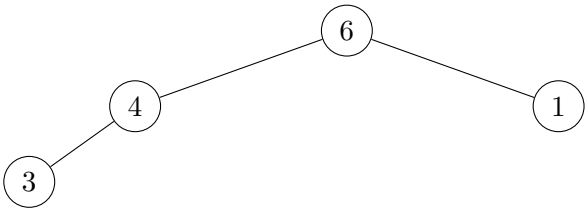
```
int[] a = {6, 4, 1, 3, 9, 2, 0, 5, 7, 8};
```

Le premier tas est facile :

L'élément 4 vient naturellement se mettre en position comme fils gauche de 6 :



et après insertion de 1 et 3, on obtient :

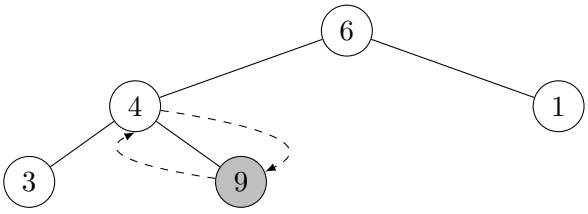


Ces éléments sont stockés dans le tableau

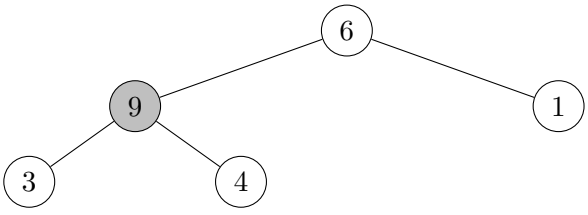
i	1	2	3	4
$t[i]$	6	4	1	3

Pour s'en rappeler, on balaie l'arbre de haut en bas et de gauche à droite.

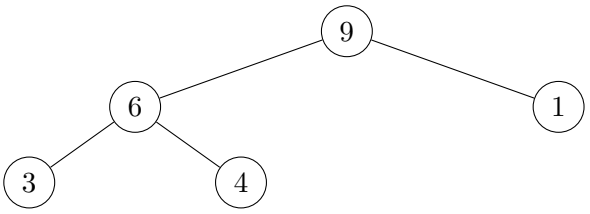
On doit maintenant insérer 9, ce qui dans un premier temps nous donne



On voit que 9 est plus grand que son père 4, donc on les permute :



Ce faisant, on voit que 9 est encore plus grand que son père, donc on le permute, et cette fois, la propriété de tas est bien satisfaite :



Après insertion de tous les éléments de `t`, on retrouve le dessin de la figure 9.5.

Le programme JAVA d'insertion est le suivant :

```
public static boolean inserer(Tas tas, int x){
    if(tas.n >= tas.t.length)
        // il n'y a plus de place
        return false;
    // il y a encore au moins une place
    tas.n += 1;
    tas.t[tas.n] = x;
    monter(tas, tas.n);
    return true;
}
```

et utilise la fonction de remontée :

```
// on vérifie que la propriété de tas est
// vérifiée à partir de tas.t[k]
public static void monter(Tas tas, int k){
    int v = tas.t[k];

    while((k > 1) && (tas.t[k/2] <= v)){
        // on est à un niveau > 0 et
        // le père est <= fils
        // le père prend la place du fils
        tas.t[k] = tas.t[k/2];
        k /= 2;
    }
    // on a trouvé la place de v
    tas.t[k] = v;
}
```

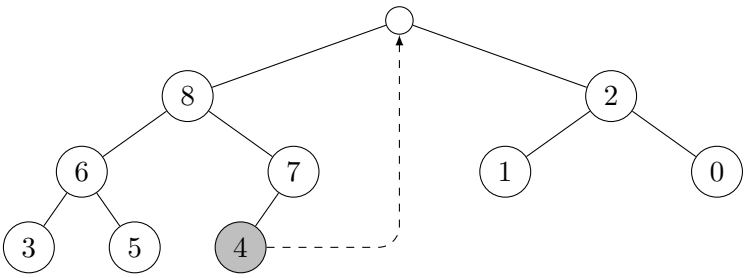
Pour transformer un tableau en tas, on utilise alors :

```
public static Tas deTableau(int[] a){
    Tas tas = creer(a.length);

    for(int i = 0; i < a.length; i++)
        inserer(tas, a[i]);
    return tas;
}
```

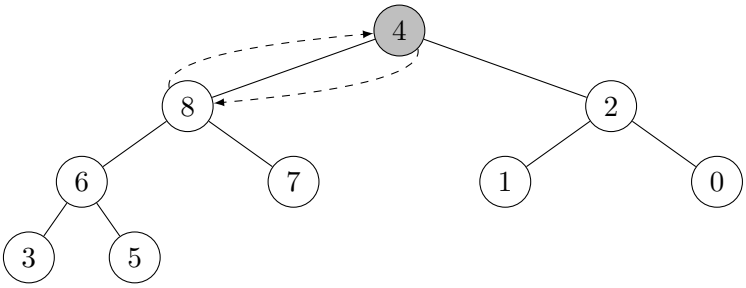
Pour parachever notre travail, il nous faut expliquer comment servir un client. Cela revient à retirer le contenu de la case `t[1]`. Par quoi la remplacer ? Le plus simple est

de mettre dans cette case `t[n]` et de vérifier que le tableau présente encore la propriété de tas. On doit donc descendre dans l'arbre.

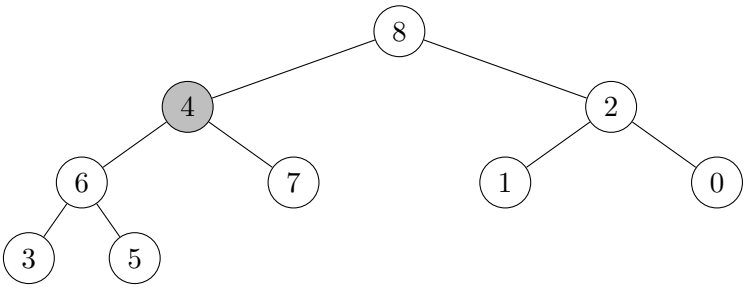
Reprenons l'exemple précédent. On doit servir le premier client de numéro 9, ce qui conduit à mettre au sommet le nombre 4 :



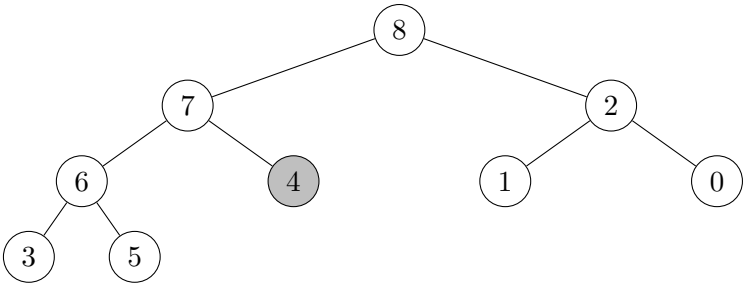
On doit maintenant faire redescendre 4 :



ce qui conduit à l'échanger avec son fils gauche :



puis on l'échange avec 7 pour obtenir finalement :



La fonction de "service" est :

```
public static int tacheSuivante(Tas tas){
    int tache = tas.t[1];

    tas.t[1] = tas.t[tas.n];
    tas.n -= 1;
    descendre(tas, 1);
    return tache;
}
```

qui appelle :

```
public static void descendre(Tas tas, int k){
    int v = tas.t[k], j;

    while(k <= tas.n/2){
        // k a au moins 1 fils gauche
        j = 2*k;
        if(j < tas.n)
            // k a un fils droit
            if(tas.t[j] < tas.t[j+1])
                j++;
        // ici, tas.t[j] est le plus grand des fils
        if(v >= tas.t[j])
            break;
        else{
            // on échange père et fils
            tas.t[k] = tas.t[j];
            k = j;
        }
    }
    // on a trouvé la place de v
    tas.t[k] = v;
}
```

Notons qu'il faut gérer avec soin le problème de l'éventuel fils droit manquant. De même, on n'échange pas vraiment les cases, mais on met à jour les cases pères nécessaires.

Proposition 3 *La complexité des procédures* `monter` *et* `descendre` *est* $O(h)$ *ou encore* $O(\log_2 n)$.

Démonstration : on parcourt au plus tous les niveaux de l'arbre à chaque fois, ce qui fait au plus $O(h)$ mouvements. □

Pour terminer cette section, nous donnons comme dernier exemple d'application un nouveau tri rapide, appelé *tri par tas* (en anglais, *heapsort*). L'idée est la suivante : quand on veut trier le tableau `t`, on peut le mettre sous la forme d'un tas, à l'aide de la procédure `deTableau` déjà donnée. Celle-ci aura un coût $O(n \log_2 n)$, puisqu'on doit insérer n éléments avec un coût $O(\log_2 n)$. Cela étant fait, on permute le plus

grand élément `t[1]` avec `t[n]`, puis on réorganise le tas `t[1..n-1]`, avec un coût $O(\log_2(n-1))$. Finalement, le coût de l'algorithme sera $O(nh) = O(n \log_2 n)$. Ce tri est assez séduisant, car son coût moyen est égal à son coût le pire : il n'y a pas de tableaux difficiles à trier. La procédure JAVA correspondante est :

```
public static void triParTas(int[] a){
    Tas tas = deTableau(a);

    for(int k = tas.n; k > 1; k--){
        // a[k..n[ est déjà trié,
        // on trie a[0..k-1]
        // t[1] contient max t[1..k] = max a[0..k-1]
        a[k-1] = tas.t[1];
        tas.t[1] = tas.t[k];
        tas.n -= 1;
        descendre(tas, 1);
    }
    a[0] = tas.t[1];
}
```

Nous verrons d'autres tris au chapitre 11.

Cette utilisation d'un tableau comme représentant un arbre complet est couramment utilisée en calcul formel, par exemple dans les arbres de produit. Nous renvoyons à [GG99] pour cela.

Chapitre 10

Graphes

Les graphes sont un moyen commode de représenter des liens entre des données. Nous allons donner ici un aperçu de ces objets. Les algorithmes élaborés seront vus dans la suite des cours.

10.1 Définitions

Un *graphe* $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est donné par un ensemble $\mathcal{S}$ de *sommets* et un ensemble $\mathcal{A}$ d'*arcs*, $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$.

On visualise en général un graphe par un dessin dans le plan. Les sommets sont représentés par des points, les arcs par des flèches. Le graphe $\mathcal{G}_1$ de la figure 10.1 a pour ensemble de sommets $\mathcal{S} = \{a, b, c, d, e, f, g, h, i, j, k, l, m\}$, et ensemble d'arêtes $\mathcal{A} = \{(a, b),\ (b, c),\ (c, b),\ (c, d),\ (d, a),\ (a, e),\ (e, f),\ (f, b),\ (f, g),\ (b, g),\ (g, h),\ (h, c),\ (h, i),\ (g, i),\ (i, c),\ (i, d),\ (d, j),\ (j, e),\ (k, f),\ (k, g),\ (k, i),\ (k, j),\ (m, l)\}$.

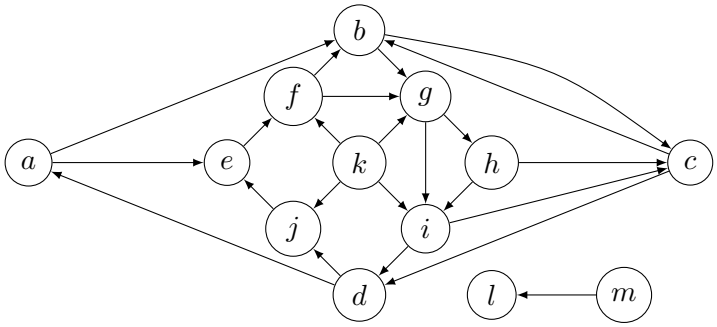


FIG. 10.1 – Le graphe $\mathcal{G}_1$.

Il est important de noter qu'il n'existe pas de dessin canonique d'un graphe. Par exemple, les deux dessins de la figure 10.2 correspondent au même graphe.

Un arc $\alpha = (a, b)$ est *orienté* de a vers b ; α a pour *origine* a et *destination* b ; l'arc α est *incident* à a ainsi qu'à b. On dira également que a est un *prédécesseur* de b, et b un *successeur* de a ; a et b seront également dits *adjacents*. L'arc (x, x) est une *boucle*.

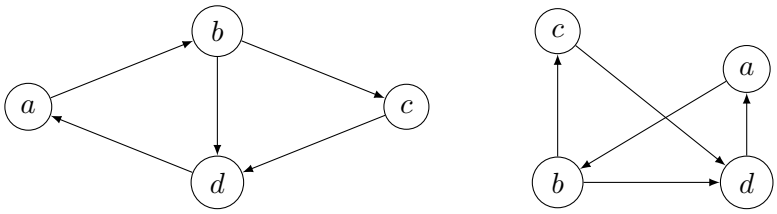


FIG. 10.2 – Deux dessins d'un même graphe.

Un graphe est *simple* s'il ne comporte pas de boucles. Le graphe de la figure 10.1 est simple. Le graphe $\mathcal{G}_2$ de la figure 10.3 ne l'est pas.

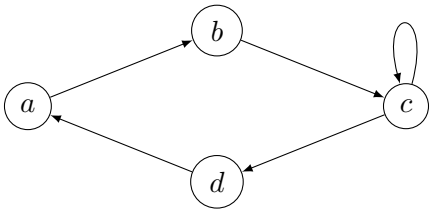


FIG. 10.3 – Le graphe $\mathcal{G}_2$.

On peut enrichir un graphe en considérant des quantités associées à chaque arc, ou à chaque sommet. Penser par exemple à la carte de circulation d'une ville, avec ses sens interdits et les longueurs des rues. On parlera d'*arcs valués*, c'est-à-dire qu'on utilisera une fonction $\nu : \mathcal{A} \to \mathbb{N}$ qui donnera le poids ou la valeur associée à un arc. C'est le cas du graphe $\mathcal{G}_3$ de la figure 10.4 dans lequel on a ajouté des poids sur chaque arc.

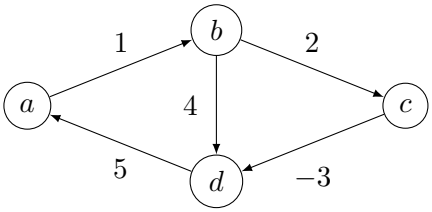


FIG. 10.4 – Le graphe valué $\mathcal{G}_3$.

Graphes non orientés

Les graphes les plus généraux que nous considérerons sont *a priori orientés*, ce qui veut dire qu'un arc indique un sens de lecture ou de dessin. Autrement dit, si l'arc (a, b) existe, cela n'implique pas l'existence de l'arc (b, a) dans le graphe. Il peut arriver que pour tout arc (a, b), son inverse (b, a) existe également. On dit que le graphe $\mathcal{G}$ est *non orienté*. Dans ce cas, on utilise le terme *arête* au lieu d'arc et on note parfois

$\{a, b\}$ l'arête reliant a à b[1]. On dessine alors des traits au lieu de flèches, comme dans l'exemple du graphe $\mathcal{G}_4$ de la figure 10.5.

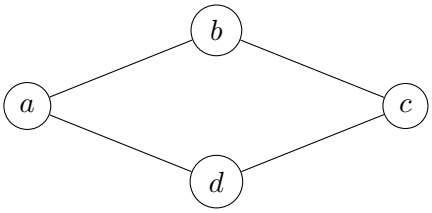


FIG. 10.5 – Le graphe $\mathcal{G}_4$.

10.2 Représentation en machine

Les dessins que nous avons faits dans la section précédente sont commodes pour faire des exemples et des raisonnements à la main, mais ils ne sont pas utilisables en machine. Dessiner dans le plan un graphe donné est en soi une tâche algorithmique et pratique relativement ardue. Donc nous devons trouver une autre représentation plus commode.

Le problème se réduit à représenter $\mathcal{S}$ et $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ en machine. Sans perte de généralité, on peut se ramener au cas où $\mathcal{S} = \{0, 1, \ldots, n-1\}$ si $n = |\mathcal{S}|$. Dans la pratique, il sera toujours possible de coder les éléments du graphe par un tel ensemble d'entiers, quitte à utiliser du hachage par exemple (voir les polys des cours précédents ou [CLR90]). Passons au problème de la représentation de $\mathcal{A}$. Avec notre définition de graphe, il est clair que le cardinal de $\mathcal{A}$ est borné par n^2. Si $m = |\mathcal{A}|$ est proche de n^2, on parlera de graphe *dense*. Si $m = o(n^2)$, on parlera de graphe *creux*. Il est très fréquent en pratique d'avoir à traiter des graphes creux (éléments finis, carte routière, graphe d'Internet, etc.). Les algorithmes que nous étudierons auront une complexité qui pourra bien souvent être décrite en terme de n et m, ce qui est toujours intéressant quand m est loin de sa valeur maximale théorique $O(n^2)$.

10.2.1 Représentation par une matrice

C'est la façon la plus simple de représenter l'ensemble $\mathcal{A}$. On se donne donc une matrice $\mathcal{M}$ de taille $n \times n$. Une telle matrice est appelée *matrice d'adjacence* du graphe. Il existe essentiellement deux choix pour le type de la matrice.

Le premier choix consiste à prendre une matrice de booléens tels que $\mathcal{M}_{i,j}$ est vrai si et seulement si $(i, j) \in \mathcal{A}$. Pour le graphe $\mathcal{G}_2$ de la figure 10.3, on numérote les sommets a, b, c, d selon la correspondance

a	b	c	d
0	1	2	3

[1] Nous considérerons des paires au sens propre, c'est-à-dire qu'un graphe non orienté ne possède pas de boucles.

puis on construit la matrice booléenne (où nous avons mis T pour vrai et omis les valeurs fausses) :

i,j	0	1	2	3
0		T		
1			T	
2			T	T
3	T			

Le deuxième choix conduit à utiliser un tableau d'entiers (ou de flottants) quand les arcs de $\mathcal{G}$ sont valués. On pose alors $\mathcal{M}_{i,j} = \nu((i,j))$ et on utilise une valeur par défaut quand l'arc (i,j) n'existe pas (par exemple une valeur plus grande que toutes les valeurs de ν). Pour le graphe $\mathcal{G}_3$ de la figure 10.4, on trouve :

i,j	0	1	2	3
0		1		
1			2	4
2				-3
3	5			

Dans le cas où $\mathcal{G}$ est non orienté, la matrice $\mathcal{M}$ est une matrice symétrique.

10.2.2 Représentation par un tableau de listes

Le stockage à l'aide de matrices prend une place proportionnelle à n^2. Quand $m = |\mathcal{A}|$ est petit par rapport à n^2, cela représente une perte de place parfois importante (ainsi qu'une perte de temps). D'où l'idée d'utiliser une représentation *creuse*, par exemple sous la forme d'un tableau de listes L. La liste L_i pour $i \in \mathcal{S}$, contiendra la liste des successeurs de i dans le graphe, c'est-à-dire

$$L_i = (j \in \mathcal{S}, (i,j) \in \mathcal{A}).$$

En utilisant une représentation "graphique" des listes, l'exemple du graphe $\mathcal{G}_2$ pourra conduire à :

```
L[0] = (1)
L[1] = (2)
L[2] = (2, 3)
L[3] = (0)
```

Il est important de noter, encore une fois, qu'une telle représentation n'est nullement canonique, car on peut décider de ranger les voisins dans n'importe quel ordre. De manière plus générale, on peut tout à fait représenter notre graphe sous forme creuse comme un tableau d'ensembles, les ensembles pouvant avoir une représentation et des propriétés variées.

La place mémoire nécessitée par une telle représentation est de $|S|$ ensembles. Dans le cas de listes chaînées, chaque ensemble demande une place $O(|L[i]|)$ (par exemple le contenu d'une cellule et d'un pointeur sur la suivante). En tout, on aura un stockage en $O(n+m)$, donc linéaire. En fonction des problèmes, on pourra trouver des représentations parfois plus adaptées.

10.3 Recherche des composantes connexes

La décomposition d'un graphe non orienté en composantes connexes permet de ramener un problème général à un problème composante par composante. Deux sommets sont dans la même composante connexe si et seulement s'il existe un chemin menant de l'un à l'autre.

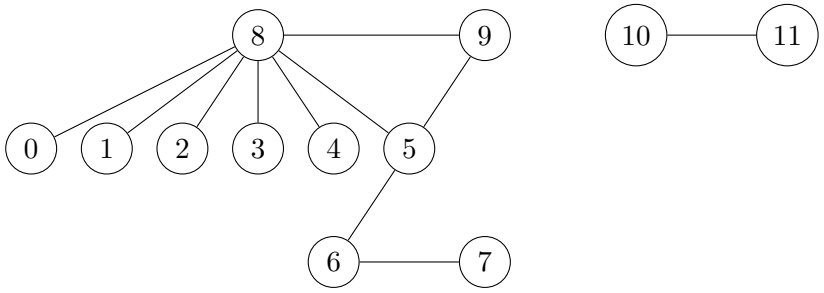


FIG. 10.6 – Le graphe exemple.

Considérons le graphe de la figure 10.6. L'idée de recherche des composantes est simple : on part d'un sommet et on explore tous les sommets possibles à partir de ce premier sommet. Pour le graphe exemple, on part de 0, qui a un voisin 8. Les voisins de 8 sont $\{0, 1, 2, 3, 4, 5, 9\}$. On connaît déjà 0, donc ce n'est pas la peine de le visiter à nouveau. Les sommets 1, 2, 3, 4 n'ont pas de voisins autre que 8, donc ils n'apportent rien à la composante. Le sommet 5 a pour voisins 6 et 9 ; 9 n'a pas de voisins non visités ; 6 a pour voisin 7, ce qui termine l'examen de la composante.

Commençons par le programme de test, qui va nous guider dans le choix des autres classes. Ce programme doit créer un graphe à partir d'un tableau d'arêtes, puis afficher le graphe, puis afficher ses composantes connexes :

```
public class TestGraphe{
    public static void main(String[] args){
        int[][] aretes = {{10, 11}, {0, 8}, {8, 1}, {8, 2},
                          {8, 3}, {8, 4}, {8, 5}, {8, 9},
                          {8, 5}, {9, 5}, {5, 6}, {6, 7}};
        int n = 12;
        Graphe G = new Graphe(n);

        for(int i = 0; i < aretes.length; i++){
            G.ajouterArc(aretes[i][0], aretes[i][1]);
            G.ajouterArc(aretes[i][1], aretes[i][0]);
        }
        System.out.println("Voici le graphe\n"+G);
        System.out.println("Voici les composantes connexes");
        G.afficherComposantesConnexes();
    }
}
```

Ceci nous suggère que la classe `Graphe` n'a pas besoin d'être très grosse, et qu'on n'a pas besoin de connaître son implantation pour s'en servir. Nous allons choisir ici une représentation par tableau de listes, une liste étant semblable à celles déjà vues au chapitre 8.

```
public class Liste {
    public int i;
    public Liste suivant;

    public Liste(Liste l, int ii){
        this.i = ii;
        this.suivant = l;
    }

    public static Liste enlever(Liste l, int ii){
        if(l == null)
            return null;
        if(l.i == ii)
            return l.suivant;
        l.suivant = enlever(l.suivant, ii);
        return l;
    }

    public String toString(){
        if(this == null)
            return "";
        else
            return this.i + " -> " + this.suivant;
    }
}
```

Les fonctions sont classiques, et il est à noter que la fonction qui enlève un entier est couteuse, car proportionnelle à la longueur de la liste. Le début de la classe `Graphe` est alors :

```
public class Graphe {
    private Liste[] voisins;

    public Graphe(int n){
        this.voisins = new Liste[n];
    }

    public void ajouterArc(int s, int t){
        this.voisins[s] = new Liste(this.voisins[s], t);
    }

    public String toString(){
        String str = "";

        for(int i = 0; i < this.voisins.length; i++)
```

```
            if(this.voisins[i] != null){
                str += "S" + i + " = ";
                str += this.voisins[i].toString() + "\n";
            }
        return str;
    }
}
```

Il nous reste à programmer la recherche de composantes connexes. Pour ce faire, nous allons introduire une classe `Ensemble` qui permet les opérations suivantes : ajout (`add`), retrait (`remove`), retrait du "premier élément" (`removeFirst`), test de vacuité (`isEmpty`), contient (`has`). On procède composante par composante, en prenant dans la liste des sommets du graphe, un sommet non encore utilisé. Une composante sera également de type `Ensemble`. L'itération principale porte sur la `Liste avisiter` qui gère de façon dynamique les nouveaux amis qui apparaissent lors de l'exploration.

```
    public void afficherComposantesConnexes(){
        int n = this.voisins.length;
        Ensemble sommets = new Ensemble(n);
        int ncc = 0;

        // on remplit la liste des sommets
        for(int i = 0; i < n; i++)
            sommets.add(i);
        while(! sommets.isEmpty()){
            Ensemble cc = new Ensemble(n); // la composante
            int s = sommets.removeFirst();
            Liste avisiter = new Liste(null, s);
            while(avisiter != null){
                // tant qu'il reste des sommets à explorer
                int t = avisiter.i;
                avisiter = avisiter.suivant;
                if(! cc.has(t)){
                    // t est nouveau dans la composante
                    cc.add(t);
                    // on rajoute ses voisins non
                    // encore visités
                    Liste l = this.voisins[t];
                    while(l != null){
                        if(sommets.has(l.i)){
                            // l.i n'a pas encore été visité
                            avisiter = new Liste(avisiter, l.i);
                            sommets.remove(l.i);
                        }
                        l = l.suivant;
                    }
                }
            }
            ncc++;
```

```
            System.out.println("Composante "+ncc+": "+cc);
        }
    }
```

Il nous reste à coder la classe `Ensemble` de façon que les opérations soient les plus rapides possibles (typiquement en $O(1)$). Cela suggère d'utiliser un tableau de booléens qui indique si un sommet est déjà utilisé ou non. Mais cela ne suffit pas, car on doit pouvoir extraire un sommet non encore utilisé. D'où l'utilisation conjointe d'une liste, même avec un temps de gestion qui est parfois long (à cause de `remove`). Cela nous donne :

```
public class Ensemble {
    private boolean[] utilise;
    private Liste li;

    public Ensemble(int n){
        boolean[] b = new boolean[n];

        for(int i = 0; i < n; i++)
            b[i] = false;
        this.utilise = b;
        this.li = null;
    }

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

    public boolean has(int i){
        return this.utilise[i];
    }

    public void add(int i){
        if(! this.utilise[i]){
            this.utilise[i] = true;
            this.li = new Liste(this.li, i);
        }
    }

    public void remove(int i){
        this.utilise[i] = false;
        // cette etape est couteuse
        this.li = Liste.enlever(this.li, i);
    }

    public int removeFirst(){
        int f = this.li.i;

        this.li = this.li.suivant;
```

```
        this.utilise[f] = true;
        return f;
    }

    public String toString(){
        String str = "{";
        Liste l = this.li;

        while(l != null){
            str += l.i;
            l = l.suivant;
            if(l != null)
                str += ", ";
        }
        str += "}";
        return str;
    }
}
```

Exercice. Exécuter les programmes ci-dessous à la main sur l'exemple donné à la figure 10.6.

Pour disposer d'une gestion pour laquelle `remove` est plus rapide, il faut utiliser d'autres structures de données moins intuitives, qui permettent d'itérer sur `sommets` rapidement. Nous en donnons un exemple ici (d'autres existent).

Exercice (*). On introduit la structure de données suivante (inspirée des travaux de Mairson [Mai77]), qui permet de faire des opérations sur les entiers entre 1 et n en gérant une liste du type

```
0 <-> 1 <-> 2 <-> ... <-> n <-> n+1
```

avec 0 et $n+1$ des sentinelles. On définit deux tableaux `rlink` et `llink`, tels que $rlink[i] = i+1$ et $llink[i] = i-1$ au départ du programme. Quand on veut supprimer l'élément i de la structure, on est dans la situation

```
i1 <-> i <-> i2
```

et la suppression de i va entraîner

```
i1 <-> i2
```

Montrer que cette structure de données peut être utilisée pour faire rapidement les opérations nécessaires au parcours des sommets dans `afficherComposantesConnexes` (variable `sommets`). Écrire la classe `Ensemble2` correspondante (avec les mêmes fonctions que pour `Ensemble` sans la fonction `add` inutile) et en déduire la fonction `afficherComposantesConnexes2`.

10.4 Conclusion

Nous venons de voir comment stocker des informations présentant des liens entre elles. Ce n'est qu'une partie de l'histoire. Dans les bases de données, on stocke des

informations pouvant avoir des liens compliqués entre elles. Pensez à une carte des villes de France et des routes entre elles, par exemple. Ceci fera l'objet d'autres cours du département dans les années qui suivent.

Chapitre 11

Ranger l'information... pour la retrouver

L'informatique permet de traiter des quantités gigantesques d'information et déjà, on dispose d'une capacité de stockage suffisante pour archiver tous les livres écrits. Reste à ranger cette information de façon efficace pour pouvoir y accéder facilement. On a vu comment construire des blocs de données, d'abord en utilisant des tableaux, puis des objets. C'est le premier pas dans le stockage. Nous allons voir dans ce chapitre quelques-unes des techniques utilisables pour aller plus loin. D'autres manières de faire seront présentées dans le cours INF 421.

11.1 Recherche en table

Pour illustrer notre propos, nous considérerons deux exemples principaux : la correction d'orthographe (un mot est-il dans le dictionnaire ?) et celui de l'annuaire (récupérer une information concernant un abonné).

11.1.1 Recherche linéaire

La manière la plus simple de ranger une grande quantité d'information est de la mettre dans un tableau, qu'on aura à parcourir à chaque fois que l'on cherche une information.

Considérons le petit dictionnaire contenu dans la variable `dico` du programme ci-dessous :

```
public class Dico{

    public static boolean estDans(String[] dico, String mot){
        boolean estdans = false;

        for(int i = 0; i < dico.length; i++)
            if(mot.compareTo(dico[i]) == 0)
                estdans = true;
        return estdans;
    }
```

```
    public static void main(String[] args){
        String[] dico = {"maison", "bonjour", "moto",
                         "voiture", "artichaut", "Palaiseau"};

        if(estDans(dico, args[0]))
            System.out.println("Le mot est présent");
        else
            System.out.println("Le mot n'est pas présent");
    }
}
```

Pour savoir si un mot est dans ce petit dictionnaire, on le passe sur la ligne de commande par

```
unix% java Dico bonjour
```

On parcourt tout le tableau et on teste si le mot donné, ici pris dans la variable `args[0]` se trouve dans le tableau ou non. Le nombre de comparaisons de chaînes est ici égal au nombre d'éléments de la table, soit n, d'où le nom de *recherche linéaire*.

Si le mot est dans le dictionnaire, il est inutile de continuer à comparer avec les autres chaînes, aussi peut-on arrêter la recherche à l'aide de l'instruction `break`, qui permet de sortir de la boucle `for`. Cela revient à écrire :

```
        for(int i = 0; i < dico.length; i++)
            if(mot.compareTo(dico[i]) == 0){
                estdans = true;
                break;
            }
```

Si le mot n'est pas présent, le nombre d'opérations restera le même, soit $O(n)$.

11.1.2 Recherche dichotomique

Dans le cas où on dispose d'un ordre sur les données, on peut faire mieux, en réorganisant l'information suivant cet ordre, c'est-à-dire en triant, sujet qui formera la section suivante. Supposant avoir trié le dictionnaire (par exemple avec les méthodes de la section 11.2), on peut maintenant y chercher un mot par dichotomie, en adaptant le programme déjà donné au chapitre 7, et que l'on trouvera à la figure 11.1. Rappelons que l'instruction `x.compareTo(y)` sur deux chaînes `x` et `y` retourne 0 en cas d'égalité, un nombre négatif si `x` est avant `y` dans l'ordre alphabétique et un nombre positif sinon. Comme déjà démontré, le coût de la recherche dans le cas le pire passe maintenant à $O(\log n)$.

Le passage de $O(n)$ à $O(\log n)$ peut paraître anodin. Il l'est d'ailleurs sur un dictionnaire aussi petit. Avec un vrai dictionnaire, tout change. Par exemple, le dictionnaire de P. Zimmermann[1] contient 260688 mots de la langue française. Une recherche d'un mot ne coûte que 18 comparaisons au pire dans ce dictionnaire.

[1] http://www.loria.fr/~zimmerma/

```
// recherche de mot dans dico[g..d[
public static boolean dichoRec(String[] dico, String mot,
                               int g, int d){
    int m, cmp;

    if(g >= d) // l'intervalle est vide
        return false;
    m = (g+d)/2;
    cmp = mot.compareTo(dico[m]);
    if(cmp == 0)
        return true;
    else if(cmp < 0)
        return dichoRec(dico, mot, g, m);
    else
        return dichoRec(dico, mot, m+1, d);
}

public static boolean estDansDico(String[] dico,
                                  String mot){
    return dichoRec(dico, mot, 0, dico.length);
}
```

FIG. 11.1 – Recherche dichotomique.

11.1.3 Utilisation d'index

On peut repérer dans le dictionnaire les zones où on change de lettre initiale ; on peut donc construire un *index*, codé dans le tableau `ind` tel que tous les mots commençant par une lettre donnée sont entre `ind[i]` et `ind[i+1]-1`. Dans l'exemple du dictionnaire de P. Zimmermann, on trouve par exemple que le mot `a` est le premier mot du dictionnaire, les mots commençant par `b` se présentent à partir de la position 19962 et ainsi de suite.

Quand on cherche un mot dans le dictionnaire, on peut faire une dichotomie sur la première lettre, puis une dichotomie ordinaire entre `ind[i]` et `ind[i+1]-1`.

Nous laissons la programmation d'index en exercice.

11.2 Trier

Nous avons montré l'intérêt de trier l'information pour pouvoir retrouver rapidement ce que l'on cherche. Nous allons donner dans cette section quelques algorithmes de tri des données. Nous ne serons pas exhaustifs sur le sujet, voir par exemple [Knu73] pour plus d'informations.

Deux grandes classes d'algorithmes existent pour trier un tableau de taille n. Ceux dont le temps de calcul est $O(n^2)$ et ceux de temps $O(n \log n)$. Nous présenterons quelques exemples de chaque. On montrera en INF 421 que $O(n \log n)$ est la meilleure complexité possible pour la classe des algorithmes de tri procédant par comparaison.

Pour simplifier la présentation, nous trierons un tableau de n entiers `t` par ordre

croissant. Si nous devions trier un tableau d'éléments non entiers, il nous suffirait de procéder par indirection. Si `TO` est un tableau d'objets, on lui associerait un tableau auxiliaire `t` de même taille, et on comparerait `t[i]` et `t[j]` en comparant en fait `TO[t[i]]` et `TO[t[j]]`.

11.2.1 Tris élémentaires

Nous présentons ici deux tris possibles, le tri sélection et le tri par insertion. Nous renvoyons à la littérature pour d'autres algorithmes, comme le tri à bulles par exemple.

Le tri sélection

Le premier tri que nous allons présenter est le *tri par sélection*. Ce tri va opérer *en place*, ce qui veut dire que le contenu du tableau `t` va être remplacé par le contenu trié. Le tri consiste à chercher le plus petit élément de `t[0..n[`, soit `t[m]`. À la fin du calcul, cette valeur devra occuper la case 0 de `t`. D'où l'idée de permuter la valeur de `t[0]` et de `t[m]` et il ne reste plus ensuite qu'à trier le tableau `t[1..n[`. On procède ensuite de la même façon.

L'esquisse du programme est la suivante :

```
public static void triSelection(int[] t){
    int n = t.length, m, tmp;

    for(int i = 0; i < n; i++){
        // invariant: t[0..i[ contient les i plus petits
        //            éléments du tableau
        //            de départ
        m = indice du minimum de t[i..n[
        // on échange t[i] et t[m]
        tmp = t[i]; t[i] = t[m]; t[m] = tmp;
    }
}
```

On peut remarquer qu'il suffit d'arrêter la boucle à $i = n - 2$ au lieu de $n - 1$, puisque le tableau `t[n-1..n[` sera automatiquement trié.

Notons le rôle du commentaire de la boucle `for` qui permet d'indiquer une sorte de propriété de récurrence toujours satisfaite au moment où le programme repasse par cet endroit pour chaque valeur de l'indice de boucle.

Reste à écrire le morceau qui cherche l'indice du minimum de `t[i..n[`, qui n'est qu'une adaptation d'un algorithme de recherche du minimum global d'un tableau. Finalement, on obtient la fonction suivante :

```
public static void triSelection(int[] t){
    int n = t.length, m, tmp;

    for(int i = 0; i < n-1; i++){
        // invariant: t[0..i[ contient les i plus petits
        //            éléments du tableau de départ
        // recherche de l'indice du minimum de t[i..n[
        m = i;
```

```
            for(int j = i+1; j < n; j++)
                if(t[j] < t[m])
                    m = j;
            // on échange t[i] et t[m]
            tmp = t[i]; t[i] = t[m]; t[m] = tmp;
        }
    }
```

qu'on utilise par exemple dans :

```
    public static void main(String[] args){
        int[] t = {3, 5, 7, 3, 4, 6};

        triSelection(t);
    }
```

Analysons maintenant le nombre de comparaisons faites dans l'algorithme. Pour chaque valeur de $i \in [0, n-2]$, on effectue $n-1-i$ comparaisons à l'instruction `if(t[j] < t[m])`, soit au total :

$$(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$$

comparaisons. L'algorithme fait donc $O(n^2)$ comparaisons. De même, on peut compter le nombre d'échanges. Il y en a 3 par itération, soit $3(n-1) = O(n)$.

Le tri par insertion

Ce tri est celui du joueur de cartes qui veut trier son jeu (c'est une idée farfelue en général, mais pourquoi pas). On prend en main sa première carte (`t[0]`), puis on considère la deuxième (`t[1]`) et on la met devant ou derrière la première, en fonction de sa valeur. Après avoir classé ainsi les $i-1$ premières cartes, on cherche la place de la i-ième, on décale alors les cartes pour insérer la nouvelle carte.

Regardons sur l'exemple précédent, la première valeur se place sans difficulté :

3					

On doit maintenant insérer le 5, ce qui donne :

3	5				

puisque $5 > 3$. De même pour le 7. Arrive le 3. On doit donc décaler les valeurs 5 et 7

3		5	7		

puis insérer le nouveau 3 :

3	**3**	5	7		

Et finalement, on obtient :

3	3	4	5	6	7

Écrivons maintenant le programme correspondant. La première version est la suivante :

```
public static void triInsertion(int[] t){
    int n = t.length, j, tmp;

    for(int i = 1; i < n; i++){
        // t[0..i-1] est déjà trié
        j = i;
        // recherche la place de t[i] dans t[0..i-1]
        while((j > 0) && (t[j-1] > t[i]))
            j--;
        // si j = 0, alors t[i] <= t[0]
        // si j > 0, alors t[j] > t[i] >= t[j-1]
        // dans tous les cas, on pousse t[j..i-1]
        // vers la droite
        tmp = t[i];
        for(int k = i; k > j; k--)
            t[k] = t[k-1];
        t[j] = tmp;
    }
}
```

La boucle `while` doit être écrite avec soin. On fait décroître l'indice `j` de façon à trouver la place de `t[i]`. Si `t[i]` est plus petit que tous les éléments rencontrés jusqu'alors, alors le test sur $j-1$ serait fatal, j devant prendre la valeur 0. À la fin de la boucle, les assertions écrites sont correctes et il ne reste plus qu'à déplacer les éléments du tableau vers la droite. Ainsi les éléments précédemment rangés dans `t[j..i-1]` vont se retrouver dans `t[j+1..i]` libérant ainsi la place pour la valeur de `t[i]`. Il faut bien programmer en faisant décroître k, en recopiant les valeurs dans l'ordre. Si l'on n'a pas pris la précaution de garder la bonne valeur de `t[i]` sous le coude (on dit qu'on l'a *écrasée*), alors le résultat sera faux.

Dans cette première fonction, on a cherché d'abord la place de `t[i]`, puis on a tout décalé après-coup. On peut condenser ces deux phases comme ceci :

```
public static void triInsertion(int[] t){
    int n = t.length, j, tmp;

    for(int i = 1; i < n; i++){
        // t[0..i-1] est déjà trié
        tmp = t[i];
        j = i;
        // recherche la place de tmp dans t[0..i-1]
        while((j > 0) && (t[j-1] > tmp)){
            t[j] = t[j-1]; j = j-1;
        }
        // ici, j = 0 ou bien tmp >= t[j-1]
        t[j] = tmp;
    }
}
```

On peut se convaincre aisément que ce tri dépend assez fortement de l'ordre initial

du tableau `t`. Ainsi, si `t` est déjà trié, ou presque trié, alors on trouve tout de suite que `t[i]` est à sa place, et le nombre de comparaisons sera donc faible. On montre qu'en moyenne, l'algorithme nécessite un nombre de comparaisons moyen égal à $n(n+3)/4-1$, et un cas le pire en $(n-1)(n+2)/2$. C'est donc encore un algorithme en $O(n^2)$, mais avec un meilleur cas moyen.

Exercice. Pour quelle permutation le maximum de comparaisons est-il atteint ? Montrer que le nombre moyen de comparaisons de l'algorithme a bien la valeur annoncée ci-dessus.

11.2.2 Un tri rapide : le tri par fusion

Il existe plusieurs algorithmes dont la complexité atteint $O(n \log n)$ opérations, avec des constantes et des propriétés différentes. Nous avons choisi ici de présenter uniquement le tri par fusion.

Ce tri est assez simple à imaginer et il est un exemple classique d'algorithme de type diviser pour résoudre. Pour trier un tableau, on le coupe en deux, on trie chacune des deux moitiés, puis on interclasse les deux bleaux. On peut déjà écrire simplement la fonction implantant cet algorithme :

```
public static int[] triFusion(int[] t){
    if(t.length == 1) return t;
    int m = t.length / 2;
    int[] tg = sousTableau(t, 0, m);
    int[] td = sousTableau(t, m, t.length);

    // on trie les deux moitiés
    tg = triFusion(tg);
    td = triFusion(td);
    // on fusionne
    return fusionner(tg, td);
}
```

en y ajoutant la fonction qui fabrique un sous-tableau à partir d'un tableau :

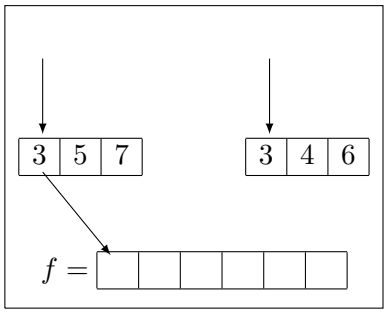
```
// on crée un tableau contenant t[g..d[
public static int[] sousTableau(int[] t, int g, int d){
    int[] s = new int[d-g];

    for(int i = g; i < d; i++)
        s[i-g] = t[i];
    return s;
}
```

On commence par le cas de base, c'est-à-dire un tableau de longueur 1, donc déjà trié. Sinon, on trie les deux tableaux `t[0..m[` et `t[m..n[` puis on doit recoller les deux morceaux. Dans l'approche suivie ici, on retourne un tableau contenant les éléments du tableau de départ, mais dans le bon ordre. Cette approche est couteuse en allocation mémoire, mais suffit pour la présentation. Nous laissons en exercice le codage de cet algorithme par effets de bord.

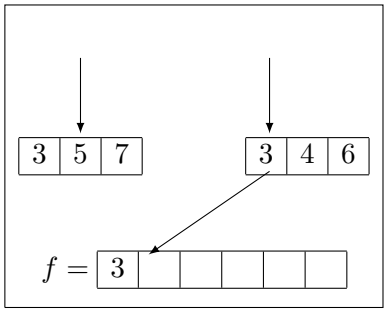
Il nous reste à expliquer comment on fusionne deux tableaux triés dans l'ordre. Reprenons l'exemple du tableau :

```
int[] t = {3, 5, 7, 3, 4, 6};
```

Dans ce cas, la moitié gauche triée du tableau est `tg = {3, 5, 7}`, la moitié droite est `td = {3, 4, 6}`. Pour reconstruire le tableau fusionné, noté `f`, on commence par comparer les deux valeurs initiales de `tg` et `td`. Ici elles sont égales, on décide de mettre en tête de `f` le premier élément de `tg`. On peut imaginer deux pointeurs, l'un qui pointe sur la case courante de `tg`, l'autre sur la case courante de `td`. Au départ, on a donc :



À la deuxième étape, on a déplacé les deux pointeurs, ce qui donne :



Pour programmer cette fusion, on va utiliser deux indices `g` et `d` qui vont parcourir les deux tableaux `tg` et `td`. On doit également vérifier que l'on ne sort pas des tableaux. Cela conduit au code suivant :

```
public static int[] fusionner(int[] tg, int[] td){
    int[] f = new int[tg.length + td.length];
    int g = 0, d = 0; // indices de parcourt de tg et td

    for(int k = 0; k < f.length; k++){
        // f[k] est la case à remplir
        if(g >= tg.length) // g est invalide
            f[k] = td[d++];
        else if(d >= td.length) // d est invalide
            f[k] = tg[g++];
        else // g et d sont valides
```

```
                if(tg[g] <= td[d])
                    f[k] = tg[g++];
                else // tg[g] > td[d]
                    f[k] = td[d++];
        }
        return f;
    }
```

Le code est rendu compact par utilisation systématique des opérateurs de post-incrémentation. Le nombre de comparaisons dans la fusion de deux tableaux de taille n est $O(n)$.

Appelons $T(n)$ le nombre de comparaisons de l'algorithme complet. On a :

$$T(n) = \underbrace{2T(n/2)}_{\text{appels récursifs}} + \underbrace{2n}_{\text{recopies}}$$

qui se résout en écrivant :

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 2.$$

Si $n = 2^k$, alors $T(2^k) = 2k2^k = O(n \log n)$ et le résultat reste vrai pour n qui n'est pas une puissance de 2. C'est le coût, quelle que soit le tableau `t`.

Que reste-t-il à dire ? Tout d'abord, la place mémoire nécessaire est $2n$, car on ne sait pas fusionner en place deux tableaux. Il existe d'autres tris rapides, comme heapsort et quicksort, qui travaillent en place, et ont une complexité moyenne en $O(n \log n)$, avec des constantes souvent meilleures.

D'autre part, il existe une version non récursive de l'algorithme de tri par fusion qui consiste à trier d'abord des paires d'éléments, puis des quadruplets, etc. Nous laissons cela à titre d'exercice.

11.3 Hachage

Revenons maintenant au stockage de l'information, dans un esprit proche de celui de la recherche en table. On peut également voir cela comme une approche dynamique du rangement, les informations à ranger arrivant l'une après l'autre. Nous allons présenter ici le hachage dit *ouvert*.

Dans le cas du dictionnaire, on pourrait aussi remplacer un tableau unique par 26 tableaux, un par lettre de début de mot, ou encore un tableau pour les mots commençant par `aa`, etc. On aurait ainsi $26^2 = 676$ tableaux, idéalement se partageant tout le dictionnaire, chaque tableau contenant $260688/676 \approx 385$ mots. Une recherche de mot couterait le coût de localisation du tableau auxiliaire, puis une dichotomie dans un tableau de taille 385.

Supposons qu'on ait à stocker N éléments. On peut essayer de fabriquer M tables ayant à peu près N/M éléments, à condition de savoir localiser facilement la table dans laquelle chercher. Arrivé là, pourquoi ne pas passer à la limite ? Cela revient à fabriquer N tables ayant 1 élément, et il ne reste plus qu'à localiser la table. Remarquons en passant qu'une table de tables à 1 élément serait avantageusement remplacée par un tableau tout court.

Pour localiser l'élément dans sa table, il faut donc savoir comment calculer une fonction de l'élément qui donne un entier qui sera l'indice dans le tableau. Expliquons comment ce miracle est possible. Si `c` est un caractère, on peut le représenter par son caractère unicode, qu'on obtient simplement par `(int)c`. Si `s` est une chaîne de caractères de longueur n, on calcule une fonction de tous ces caractères. En JAVA, on peut tricher et utiliser le fait que si `x` est un objet, alors `x.hashCode()` retourne un entier.

Une fois qu'on dispose d'un nombre $h(z)$ pour un objet z, on peut s'en servir comme indice de stockage dans un tableau `t`. On peut fixer la taille de `t`, soit T, si l'on dispose d'une borne sur le nombre d'éléments à stocker. Nous verrons plus loin comment fixer cette borne.

Considérons un exemple simple, celui où on veut ranger un ensemble $\mathcal{Z}$ d'entiers strictement positifs dans une *table de hachage*, représentée ici par le tableau `t[0..T[` initialisé à 0. Comment insère-t-on z dans le tableau ? Idéalement, on le place à l'indice $h(z)$ de `t`.

Essayons avec l'ensemble $\mathcal{Z} = \{11, 59, 32, 44, 31, 26, 19\}$. On va prendre une table de taille 10 et $h(z) = z \bmod 10$.

i	0	1	2	3	4	5	6	7	8	9
$t[i]$										

Le premier élément à mettre est 11, qu'on met dans la case $h(11) = 1$:

i	0	1	2	3	4	5	6	7	8	9
$t[i]$		11								

On continue avec 59, 32, 44 :

i	0	1	2	3	4	5	6	7	8	9
$t[i]$		11	32		44					59

Avec 31 arrive le problème : la case $h(31) = 1$ est déjà occupée. Qu'à cela ne tienne, on cherche la première case vide à droite, ici la case 3 :

i	0	1	2	3	4	5	6	7	8	9
$t[i]$		11	32	**31**	44					59

On met alors 26 dans la case 6 :

i	0	1	2	3	4	5	6	7	8	9
$t[i]$		11	32	31	44		26			59

Pour 19, on doit faire face à un nouveau problème : la case 9 est occupée, et on doit mettre 19 ailleurs, on la met dans la case 0, en considérant que le tableau `t` est géré de façon circulaire.

i	0	1	2	3	4	5	6	7	8	9
$t[i]$	**19**	11	32	31	44		26			59

Les fonctions implantant cet algorithme sont :

```
public class HachageEntier{
    final static int T = 10;
    static int[] t = new int[T];

    public static void initialiser(){
        for(int i = 0; i < T; i++)
            t[i] = 0;
    }

    public static int hash(int z){
        return (z % T);
    }

    public static void inserer(int z){
        int hz = hash(z);

        while(t[hz] != 0)
            hz = (hz+1) % T;
        t[hz] = z;
    }

    public static void main(String[] args){
        int[] u = {11, 59, 32, 44, 31, 26, 19};

        initialiser();
        for(int i = 0; i < u.length; i++)
            inserer(u[i]);
    }
}
```

Si l'on doit chercher un élément dans la table, on procède comme pour l'insertion, mais en s'arrêtant quand on trouve la valeur cherchée ou bien un 0.

```
    public static boolean estDans(int z){
        int hz = hash(z);

        while((t[hz] != 0) && (t[hz] != z))
            hz = (hz+1) % T;
        return (t[hz] == z);
    }
```

Exercice. Comment programmer la suppression d'un élément dans une table de hachage ?

Donnons maintenant un cas un plus réaliste, celui d'un annuaire, composé d'abonnés ayant un nom et un numéro de téléphone :

```
public class Abonne{
    String nom;
```

```
    int tel;

    public static Abonne creer(String n, int t){
        Abonne ab = new Abonne();

        ab.nom = n;
        ab.tel = t;
        return ab;
    }
}
```

Le hachage se fait comme précédemment :

```
public class HachageAnnuaire{
    final static int T = 101;
    static Abonne[] t = new Abonne[T];

    public static int hash(String nom){
        return Math.abs(nom.hashCode()) % T;
    }

    public static void inserer(String nom, int tel){
        int h = hash(nom);

        while(t[h] != null)
            h = (h+1) % T;
        t[h] = Abonne.creer(nom, tel);
    }

    public static int rechercher(String nom){
        int h = hash(nom);

        while((t[h] != null) && (! nom.equals(t[h].nom)))
            h = (h+1) % T;
        if(t[h] == null)
            return -1;
        else
            return t[h].tel;
    }

    public static void initialiser(){
        for(int i = 0; i < t.length; i++)
            t[i] = null;
        inserer("dg", 4001);
        inserer("dgae", 4002);
        inserer("de", 4475);
        inserer("cdtpromo", 5971);
        inserer("kes", 4822);
        inserer("bobar", 4824);
```

```
        inserer("scola", 4154);
        inserer("dix", 3467);
    }

    public static void main(String[] args){
        initialiser();
        System.out.println(rechercher("bobar"));
        System.out.println(rechercher("dg"));
    }
}
```

On peut montrer le résultat suivant :

Théorème 2 *On appelle N le nombre d'éléments déjà présents dans la table et on pose $\alpha = N/T$. Alors le nombre d'opérations à faire est :*
$1/2 + 1/(2(1-\alpha))$ *pour une recherche avec succès,*
$1/2 + 1/(2(1-\alpha)^2)$ *pour une recherche avec échec.*

Par exemple, si $\alpha = 2/3$, on fait 2 ou 5 opérations. Cela permet de stocker un ensemble de M éléments à l'aide d'un tableau de $3/2M$ entiers et le test d'appartenance se fait en $O(1)$ opérations.

Il existe d'autres techniques de hachage. Par exemple celle qui consiste à gérer les collisions en utilisant des listes chaînées.

Terminons avec quelques applications du hachage. Dans le logiciel de calcul formel MAPLE, il faut calculer une adresse dans la mémoire pour n'importe quel type d'objet. La comparaison est très simple par calcul d'adresse, même sur de gros objets. Dans les navigateurs, le hachage permet de repérer les URL parcourues récemment (donc mises en grisé). De la même façon, les moteurs de recherche comme GOOGLE, doivent stocker plusieurs milliards de chaînes de caractères (correspondant à $> 500 \times 10^6$ pages), ce qui n'est possible qu'avec du hachage.

Troisième partie

Introduction au génie logiciel

Chapitre 12

Comment écrire un programme

Ce chapitre a pour but de dégager de grandes constantes dans l'écriture de petits ou de gros programmes, en commençant par les petits. Il vaut mieux prendre de bonnes habitudes tout de suite. Quels sont les buts à atteindre : on cherche toujours la concision, la modularité interne et externe, la réutilisation éventuelle.

Ajoutons qu'un programme évolue dans le temps, qu'il n'est pas figé, et on doit donc prévoir qu'il va évoluer, que ce soit sous la main du programmeur originel, ou de ces successeurs qui vont devoir en modifier quelques lignes.

12.1 Pourquoi du génie logiciel ?

Le code source de Windows XP représente 50 millions de lignes de code, Linux environ 30 millions. Comment peut-on gérer autant de lignes de code, et autant de programmeurs supposés ?

Dans un article des *Communications of the ACM* (septembre 2006), des données rassemblées par le *Quantitative Software Management* sont présentées. L'étude a porté sur 564 projets récents, réalisés dans 31 entreprises dans 16 branches dans 16 pays. Il en ressort qu'un projet moyen requiert moins de 7 personnes pour une durée inférieure à 8 mois, pour un coût moyen inférieur à 58 homme-mois, avec comme langage encore majoritaire COBOL, en passe d'être détrôné par Java, représentant moins de $9,200$ lignes de code.

Depuis l'avènement de l'informatique, de nombreux chercheurs et praticiens s'interrogent sur les aspects d'organisation des gros programmes en grosses (?) équipes. Une des bibles de référence est toujours [Bro95].

12.2 Principes généraux

12.2.1 La chaîne de production logicielle

Le schéma de la figure 12.1 permet de comprendre les différentes phases de la création d'un logiciel conséquent.

Comme on peut le constater, *un logiciel ne se résume pas à la programmation.* Malgré tout, la phase de programmation reste l'endroit où on a le plus de prise sur le produit.

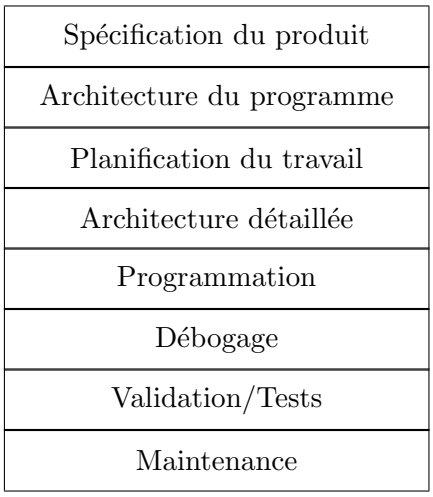


FIG. 12.1 – La chaîne de production logicielle.

La spécification et la documentation

La phase de spécification est importante et conditionne le reste. Les problèmes que l'on doit se poser sont généralement (la liste n'est pas exhaustive) :

- Sur quelle machine (avec quel système d'exploitation) le programme doit-il tourner ? Y a-t-il des intéractions avec d'autres programmes existant ?
- Que doit faire le programme ? Qui doit l'utiliser ?
- Quelles sont les opérations spécifiques ?
- Quel doit être le temps de réponse ?
- À quelles erreurs le programme doit-il pouvoir résister ?

La première question est assez bien réglée par Java, qui est *portable*, c'est-à-dire tourne sur toutes les machines.

La documentation d'un programme (petit ou gros) est fondamentale. Il faut commencer à l'écrire dès le début, avec mise à jour à chaque fois qu'on écrit une fonction.

L'architecture du programme

Une architecture excellente peut être gâtée par une programmation médiocre ; une excellente programmation ne peut pas rattraper complètement une architecture désastreuse.

La *division en sous-systèmes* permet de se mettre d'accord sur l'interface (ici pris au sens d'entrée des données, formatage des sorties) en liaison avec le moteur du programmes (fabriquant les données de sortie). Cette phase recense également les bases de données, les problèmes de communications, l'aspect graphique, etc.

C'est dans cette phase que la modularité s'exprime le mieux. On cherche toujours la simplicité, mais aussi une forme de protection contre les changements (surtout dans l'interface). Cela permet de réaliser une bonne division du travail et de minimiser les interactions.

Le chapitre 14 reviendra sur les structures de données et la réalisation de modules réutilisables.

12.2.2 Architecture détaillée

Étudiez l'architecture soigneusement et vérifiez qu'elle marche avant de continuer. La méthode généralement employée est celle de l'analyse descendante et du raffinage. Cette phase doit rester la plus abstraite possible, en particulier elle doit être la plus indépendante possible du langage de programmation choisi. De même, les détails de programmation sont renvoyés au plus tard possible.

Prenons l'exemple simple du comptage du nombre de mots dans un fichier. Les actions nécessaires sont :

- ouvrir le fichier ;
- aller au début du fichier ;
- tant que la fin du fichier n'est pas atteinte
  - lire un mot ;
  - incrémenter le nombre de mots ;
- fermer le fichier ;
- afficher le nombre de mots.

On peut encore raffiner : comment lire un mot, etc.

Rajoutons quelques règles :

- ne pas descendre de niveau tant que vous n'êtes pas convaincu(e)s que le niveau actuel est satisfaisant ;
- si un problème apparaît, c'est sans doute qu'il trouve sa source au niveau immédiatement supérieur. Remonter et régler le problème.

12.2.3 Aspects organisationnels

Planification du travail

Dans son bestseller, *The mythical Man-Month*, F. P. Brooks (qui a conçu le système d'exploitation de l'IBM 360, au début des années 1970), donne quelques règles empiriques pour un "bon" projet.

La répartition du temps devrait être celle-ci :

- 1/3 de spécification ;
- 1/6 de programmation ;
- 1/4 de test (alpha) ;
- 1/4 d'intégration et test (beta).

Il faut également garder en tête la fameuse de la figure 12.2, qui décrit l'état d'avancement vers le but final.

Des outils

Au fil du temps, des outils de programmation confortables et efficaces ont vu le jour, ce sont des *Integrated Development Environments* (IDE), comme eclipse et netbeans. Ces outils permettent de simplifier le travail sur un logiciel, en intégrant éditeur (intuitif, avec aide en ligne notamment), compilateur, tests et documentation automatiques. Cela permet entre autres de respecter les bonnes habitudes de programmation sans effort, de s'intégrer à des produits plus complexes. Leur utilisation est recommandée dans un projet moyen ou gros, et même dans les petits, ils aident à la compréhension et à l'initiation du langage (documentation en ligne, etc.). Ces IDE sont souvent disponibles sur toutes les machines et tous les systèmes.

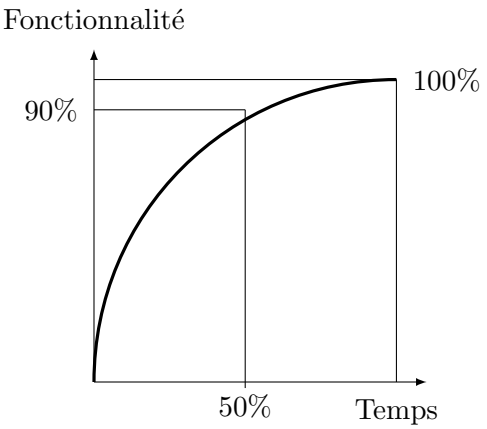


FIG. 12.2 – Fonctionnalité en fonction du temps.

Outre les IDE, il est relativement facile d'utiliser des outils de travail collaboratif, qui permettent de décentraliser la programmation et d'assurer les copies de sauvegardes. L'un des plus connus et des plus faciles à utiliser est SVN. Notons qu'ils sont également souvent integrés aussi dans les IDE, ce qui facilite encore plus la vie du programmeur.

Profitons-en pour insister sur le fait que ces outils n'ont pas que des finalités informatiques pures et dures. Ce poly est par exemple mis dans SVN, ce qui facilite la synchronisation, où que l'auteur se trouve, et même si celui-ci n'a à se coordonner qu'avec lui-même...

Programmer

La programmation est d'autant plus simple qu'elle découle logiquement de l'architecture détaillée. S'étant mis d'accord sur un certain nombre de tâches, le programme principal est facile à écrire : il se *contente* d'appeler les différentes actions prévues. On peut donc écrire cette fonction en appelant des fonctions qui ne font rien pour le moment. On parle de programmation par stubs (ou *bouchons*), voir figure 12.3. À tout instant, une maquette du programme tourne, et il ne *reste plus* qu'à programmer les bouchons les uns après les autres. Certains IDE permettent de facilement programmer par bouchons, en écrivant directement les prototypes des fonctions, libérant ainsi le programmeur de tâches fastidieuses.

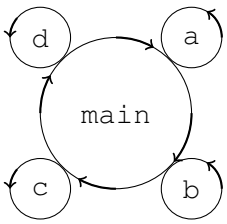


FIG. 12.3 – Programmation par bouchons.

```
public class MaClasse{
    static int a(int i){
        return 1;
    }
    static void b(int n){
    }
    static void c(int k){
    }
    static void d(){
    }
    public static void main(String[] args){
        int n = a(1);
        b(n);
        c(n-1);
        d(); // démo
    }
}
```

Cela illustre également un principe fondamental : le test et le débug doivent accompagner l'écriture du programme à chaque instant. Les bugs doivent être corrigés le plus vite possible, car ils peuvent révéler très tôt des problèmes de conception irratrappables par la suite.

Règle d'or : il est absurde d'écrire 1000 lignes de code et de les déboguer d'un seul coup.

Déboguer

La programmation fait intervenir trois acteurs : le programme, le programmeur et le bogue.

Déboguer est un art qui demande patience, ingéniosité, expérience, un temps non borné et... du sommeil !

Il est bon de se répéter les lois du débogage dès que tout va mal :

- tout logiciel complexe contient des bogues ;
- le bogue est probablement causé par la dernière chose que vous venez de modifier ;
- si le bogue n'est pas là où vous pensez, c'est qu'il est ailleurs ;
- un bogue algorithmique est beaucoup plus difficile à trouver qu'un bogue de programmation pure.
- on ne débogue pas un programme qui marche !

Au-delà de ces boutades, déboguer relève quand même d'une démarche scientifique : il faut isoler le bogue et être capable de le reproduire. Déboguer est donc très difficile dans les programmes non déterministes (attention aux générateurs aléatoires – mieux vaut les débrancher au départ ; parallélisme, calculs distribués, etc.).

Une façon de procéder classique est d'afficher ou de faire des tests dans le programme, tests qui ne seront activés que dans certains cas, par exemple si une variable de classe `debug` est mise à **`true`**. Des tests de condition (assertions) à l'intérieur des programmes peuvent être utilisés avec profit.

Il existe aussi des outils qui permettent de détecter certains bugs (profileurs, etc.). Notons encore une fois que les IDE contiennent un débogueur intégré, qui permet d'exécuter pas à pas un programme pour localiser les problèmes.

Valider et tester

D'après G. J. Myers[Myc04] : *tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts.*

Le test d'un programme est une activité prenante, et nécessaire, du moins quand on veut réaliser un programme correct, et non pas un produit à vendre au client, à qui on fera payer les corrections et les mises à jour.

Règle d'or : il faut penser à écrire son programme de façon qu'il soit facile à tester.

On sera amené à écrire des programmes fabriquant des jeux de test automatiquement (en Java ou langage de script). On doit écrire des tests *avant d'écrire le programme principal*, ce qui facilitera l'écriture de celui-ci, et permettra de contrôler et valider la spécification du programme. Écrire un programme de test permet de le réutiliser à chaque modification du programme principal.

Les tests de fonctionnalité du programme sont impératifs, font partie du projet et sont de la responsabilité immédiate du programmeur. Plus généralement, on parle d'*alpha test* pour désigner les tests faits par l'équipe de développement ; le code est alors gelé, seuls les bogues corrigés. En phase de *beta test*, les tests sont réalisés par des testeurs sélectionnés et extérieurs à l'équipe de développement.

Écrire des tests n'est pas toujours facile. Ils doivent couvrir tous les cas normaux ou anormaux (boîte de verre, boîte noire, etc.). Souvent, tester toutes les branches d'un programme est tout simplement impossible.

On procède également à des tests de non régression, pour contrer la célèbre maxime : *la modification d'un logiciel : un pas en avant, deux pas en arrière.* Ce sont des tests que l'on pratique dès la réussite de la compilation, pour vérifier que des tests (rapides) passent encore. Ils contiennent souvent les instances ayant conduit à des bugs dans le passé. Les tests lourds sont souvent regroupés à un autre moment, par exemple la nuit (ce sont les fameux *night-build*).

Les *tests de validation* permettent de vérifier que l'on a bien fait le logiciel et on valide que l'on a fait le bon logiciel. Ce sont ceux que l'on passe en dernier quand tout le reste est près.

Le test structurel statique (boîte de verre) Il s'agit là d'analyser le code source sans faire tourner le programme. Parmi les questions posées, on trouve :

- **Revues de code :** individuelles ou en groupe ; permet de s'assurer du respect de certains standards de codage.
  - y a-t-il suffisamment de commentaires ?
  - les identifieurs ont-ils des noms pertinents ?
  - le code est-il structuré ?
  - y a-t-il trop de littéraux (variables) ?
  - la taille des fonctions est-elle acceptable ?
  - la forme des décisions est-elle assez simple ?
- **Jeux de tests :** pour toutes les branches du programme (si possible).

– **Preuve formelle :** cf. deuxième partie du cours + cours année 3 (notamment analyse statique, preuve de programme).

Heureusement, des outils existent pour faire tout cela, et représentent des sujets de recherche très actifs en France.

Le test fonctionnel (boîte noire) On s'attache ici au comportement fonctionnel d'un programme, sans regarder le contenu du programme. On pratique ainsi souvent des tests aléatoires, pratiques et faciles à programmer, mais qui ne trouvent pas souvent les bogues dures.

Bien sûr, dans certains cas, on peut tester le comportement du programme sur toutes les données possibles. Pour de plus gros programmes, on pratique souvent l'analyse partitionnelle, qui revient à établir des classes d'équivalence de comportement pour les tests et comparer sur une donnée représentative de chaque classe que le comportement est bien celui attendu, incluant le test aux limites.

Donnons quelques exemples :

– Si la donnée $x \in [a, b]$:
  – une classe de valeurs pour $x < a$ (resp. $x > b$) ;
  – n valeurs valides, dont a, b.
– Si x est un ensemble avec $|x| \leqslant X$:
  – cas invalides : $x = \emptyset$, $|x| > X$;
  – n valeurs valides.
– Si $x = \{x_0, x_1, \ldots, x_r\}$ (avec r petit) :
  – une classe valide pour chaque x_i ;
  – des classes invalides, comprenant une classe correcte sauf pour un des x_i, pour tous les i.
– Si x est une obligation ou contrainte (forme, syntaxe, sens) : une classe contrainte respectée, une non respectée.

Illustrons ceci par la vérification d'une fonction qui calcule $F(x) = \sqrt{1/x}$. Les classes utilisées pourront être :

– réels négatifs ;
– $x = 0$;
– réels strictement positifs.

Analyser les performances (benchmarks) Mesurer la vitesse de son programme est également une bonne idée. Même si tous les programmes du monde ne peuvent se terminer instantément, essayer de comprendre où on passe son temps est primordial, et un programme rapide est plus vendeur.

On peut écrire un programme de test qui affiche les paramètres pertinents. On peut tester 2 fonctions et produire deux courbes de temps, qu'il reste à afficher et commenter (`xgraphic` ou `gnuplot` en Unix).

Si l'algorithme théorique est en $O(n^2)$, on teste avec n, $2n$, $3n$ et on regarde si le temps varie par un facteur 4, 9, ... Si le temps est inférieur, c'est qu'il y a une erreur ; s'il est supérieur, c'est qu'on passe du temps à faire autre chose et il faut comprendre pourquoi. Si le comportement dépend trop de la valeur initiale de n, il y a matière à problème.

Il ne reste plus qu'à commenter, déduire, etc. C'est le côté expérimental de l'informatique.

12.2.4 En guise de conclusion provisoire...

Un programme ressemble à un pont :
- Plus le projet est grand, plus il faut soigner l'architecture et le planning. Les problèmes humains ne peuvent être négligés.
- Découvrir les erreurs très vite est essentiel (ou *la découverte tardive est catastrophique*).
- Les erreurs peuvent être désastreuses (Ariane 5 – 1 milliard de dollars).
- Utiliser des préfabriqués permet de gagner du temps.

Un programme n'est pas un pont :
- Le logiciel est purement abstrait ; il est *invisible*, car il n'est vu que par son action sur un matériel physique.
- Le logiciel est écrit pour être changé, amélioré.
- Le logiciel est en partie réutilisable.
- Le logiciel peut être testé à tout moment de sa création et de sa vie.

12.3 Un exemple détaillé

12.3.1 Le problème

On cherche à calculer le jour de la semaine correspondant à une date donnée.

Face à n'importe quel problème, il faut établir une sorte de cahier des charges, qu'on appelle *spécification du programme*. Plus la spécification sera précise, plus le programme final sera conforme aux attentes. Ici, on entre la date en chiffres sous la forme agréable `jj mm aaaa` et on veut en réponse le nom du jour écrit en toutes lettres.

Avons-nous tout dit ? Non. En particulier, comment les données sont-elles fournies au programme ? Par exemple, le programme doit prendre en entrée au terminal trois entiers j, m, a séparés par des espaces, va calculer J et afficher le résultat sous une forme agréable compréhensible par l'humain qui regarde. Que doit faire le programme en cas d'erreur sur les données en entrée ? Nous indiquons une erreur à l'utilisateur, mais nous ne voulons pas que le programme "plante". Finalement, quel format voulons-nous pour la réponse ? Est-ce que nous voulons que la réponse soit nécessairement en français, ou bien pouvons-nous choisir la langue de la réponse ? Nous allons commencer avec le français, et afficher

```
Le j/m/a est un xxx.
```

Nous allons aussi imaginer que nous pourrons faire évoluer le programme pour donner la réponse dans d'autres langues (cf. section 13.3).

12.3.2 Architecture du programme

Quelle architecture pour ce programme ? Nous devons garder en tête que le programme doit pouvoir être testé dans toutes ses parties. Il doit également avoir une interface conviviale, à la fois pour entrer les données, mais également pour afficher le résultat du calcul. Si l'on veut un programme le plus générique possible, il faut que chacune de ses parties soit la plus indépendante possible des autres. Une analyse descendante possible est la suivante :
- l'utilisateur entre les données ;

– le programme vérifie la validité des données d'entrée et avertit l'utilisateur si tel n'est pas le cas ;
– le programme calcule le jour de la semaine correspondant aux paramètres ;
– le programme affiche ce jour de façon conviviale.

Ce découpage est simple, et il laisse pour plus tard le modèle d'entrée des données et le modèle de sortie. Ceux-ci pourront être changés sans que le cœur du calcul n'ait besoin d'être modifié, c'est l'intérêt de la modularité. De même, nous pourrions remplacer la primitive de calcul par une autre basée sur un algorithme différent, sans avoir à changer les autres morceaux.

Très généralement, les entrées seront des chaînes de caractères, qu'il faudra vérifier. Cette vérification prend deux étapes : dispose-t-on de trois entiers ? Si oui, représentent-ils une date valide ? Ces derniers calculs sont plus faciles à faire sur des entiers. Ainsi, on peut rafiner l'architecture :

– l'utilisateur entre les données sous forme de trois chaînes de caractères ;
– le programme vérifie que les trois chaînes de caractères représentent des entiers ;
– si c'est le cas, on récupère les trois entiers (trois **int** suffiront) et on vérifie qu'ils correspondent bien à une date ;
– le programme calcule le jour de la semaine correspondant aux paramètres ;
– le programme affiche ce jour de façon conviviale.

12.3.3 Programmation

Appliquons le principe de la programmation par stubs. Nous devons progresser incrémentalement, avec à chaque fois des progrès mesurables dans l'utilisation du programme. Le premier squelette qu'on peut écrire et qui permet de démarrer peut être le suivant :

```
public class Jour{

    static boolean donneesCorrectes(int j, int m, int a){
        return true; // stub
    }

    public static String jourDeLaSemaine(int j, int m, int a){
        return "vendredi"; // stub
    }

    public static String calculerJour(String sj, String sm,
                                      String sa){

        int j, m, a;

        j = Integer.parseInt(sj);
        m = Integer.parseInt(sm);
        a = Integer.parseInt(sa);
        if(donneesCorrectes(j, m, a))
            return jourDeLaSemaine(j, m, a);
        else
            return null;
```

```
    }

    public static void afficherJour(String sj, String sm,
                                    String sa, String s){
        System.out.print("Le "+sj+"/"+sm+"/"+sa);
        System.out.println(" est un "+s+".");
    }

    public static void main(String[] args){
        String s, sj, sm, sa;

        sj = "18";
        sm = "3";
        sa = "2011";
        s = calculerJour(sj, sm, sa);
        if(s != null)
            afficherJour(sj, sm, sa, s);
        else
            System.out.println("Données incorrectes");
    }
}
```

Ce programme ne fait pas grand chose, mais il faut bien commencer par le commencement. Il compile, il s'exécute, et affiche ce qu'on veut pour au moins un exemple connu. Nous avons également décidé qu'en cas de problème, la fonction `calculerJour` retourne **null**. Noter la convention adoptée pour différencier (dans la tête du programmeur) l'entier `j` et la chaîne `sj`. D'autre part, nous avons converti les chaînes de caractères en entiers à l'aide de la fonction `Integer.parseInt` (nous y reviendrons plus loin).

Armés de ce squelette, nous pouvons commencer à écrire un programme de test, que l'on fera évoluer en même temps que le programme :

```
public class TesterJour{

    static boolean testerJour(String t){
        String resultat, s;
        String[] tab;

        tab = t.split(" ");
        resultat = tab[3];
        s = Jour.calculerJour(tab[0], tab[1], tab[2]);
        if(s == null)
            return resultat.equals("erreur");
        else
            return s.equals(resultat);
    }

    public static void recette(){
        String[] tests = {"18 3 2011 vendredi",
```

```
                          "1 1 1 erreur"};

        for(int i = 0; i < tests.length; i++){
            boolean ok = testerJour(tests[i]);

            System.out.print("Test "+i+" : ");
            System.out.println(ok);
        }
    }

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

Le programme principal de cette fonction utilise un tableau de chaînes qui contiennent une entrée et la sortie attendue. La fonction `testerJour` appelle la fonction idoine de la classe `jour` et compare le résultat obtenu à celui qu'elle attend et affiche le résultat du test de comparaison. Une fois cela fait, il sera facile de rajouter des chaînes de test au fur et à mesure de l'écriture du programme `Jour`. Noter l'utilisation de l'instruction bien pratique

```
tab = t.split(" ");
```

qui décompose la chaîne `t` en chaînes de caractères séparées par un blanc et retourne un tableau formé de ces chaînes. Nous avons également anticipé sur un résultat de calcul qui donnerait **`null`**.

Le cœur du programme est le calcul du jour de la semaine basé sur le théorème 3 donné à la fin du chapitre. Seul nous intéresse ici la spécification de cette fonction. Il n'utilise que des données numériques et retourne une donnée numérique. La fonction correspondante est facile à écrire :

```
  // ENTRÉE: 1 <= j <= 31, 1 <= m <= 12, 1584 < a
  // SORTIE: entier J tel que 0 <= J <= 6, avec 0 pour
  //         dimanche, 1 pour lundi, etc.
  // ACTION: J est le jour de la semaine correspondant à
  //         la date donnée sous la forme j/m/a
  public static int jourZeller(int j, int m, int a){
      int mz, az, e, s, J;

      // calcul des mois/années Zeller
      mz = m-2;
      az = a;
      if(mz <= 0){
          mz += 12;
          az--;
      }
      // az = 100*s+e, 0 <= e < 100
      s = az / 100;
      e = az % 100;
```

```
        // la formule du révérend Zeller
        J = j + (int)Math.floor(2.6*mz-0.2);
        J += e + (e/4) + (s/4) - 2*s;
        // attention aux nombres négatifs
        if(J >= 0)
            J %= 7;
        else{
            J = (-J) % 7;
            if(J > 0)
                J = 7-J;
        }
        return J;
    }
```

Les commentaires indiquent des propriétés supposées satisfaites en entrée et en sortie. Cette fonction suffit-elle à nos besoins ? Il paraît logique d'introduire une fonction un peu générale qui va cacher l'utilisation de la méthode de Zeller, qui ne parle pas nécessairement au lecteur. Par exemple :

```
    final static String[] JOUR = {"dimanche", "lundi", "mardi",
                                  "mercredi", "jeudi",
                                  "vendredi", "samedi"};

    public static String jourDeLaSemaine(int j, int m, int a){
        int jz = jourZeller(j, m, a);

        return JOUR[jz];
    }
```

Ici, nous avons fait le choix de définir des constantes globales (champs statiques) de la classe, que l'on peut exporter et réutiliser dans d'autres contextes. Nous aurions aussi pu garder la correspondance numérique/texte des jours de la semaine à l'intérieur de la fonction `jourDeLaSemaine`. Nous pouvons en parallèle ajouter d'autres tests simples dans `TesterJour.java` :

```
        String[] tests = {"18 3 2011 vendredi",
                          "1 1 1 erreur",
                          "19 3 2011 samedi",
                          "10 5 2011 mardi"
        };
```

Il nous faut maintenant remplir les fonctions qui ne faisaient rien, parce que par exemple nous n'avions pas besoin de contrôler nos entrées. Dans le monde réel, où un utilisateur n'est pas nécessairement le programmeur lui-même, il faut prévoir beaucoup de cas d'erreurs, ne serait-ce que pour ne pas perturber l'ordinateur (ou le système).

Ainsi, que doit tester la fonction `donneesCorrectes` ? Que les valeurs de `j`, `m`, et `a` sont correctes. Pour le mois et l'année, c'est facile, mais pour le jour, c'est plus compliqué car on doit faire intervenir le fait que l'année peut-être bissextile. On va également utiliser un tableau pour stocker le nombre de jours de chaque mois, ainsi qu'une fonction qui retourne le nombre de jours dans un mois. Cela nous donne

```
    final static int[] JOURS_DANS_MOIS = {31, 28, 31, 30, 31,
                                          30, 31, 31, 30, 31,
                                          30, 31};

    static boolean estBissextile(int a){
        if((a % 4) != 0)
            return false;
        if((a % 100) != 0)
            return true;
        return ((a % 400) == 0);
    }

    static int nbJoursDansMois(int m, int a){
        if((m != 2) || !estBissextile(a))
            return JOURS_DANS_MOIS[m-1];
        else
            return 29;
    }

    static boolean donneesCorrectes(int j, int m, int a){
        if(a <= 1584)
            return false;
        if((m < 1) || (m > 12))
            return false;
        if((j < 1) || (j > nbJoursDansMois(m, a)))
            return false;
        return true;
    }
```

À ce point, nous devons tester les nouvelles fonctions ajoutées. Nous allons donner les fonctions de test pour l'année bissextile, laissant les autres en exercices. Ici, quatre cas sont suffisants pour couvrir tous les branchements du code. La fonction d'appel du test s'écrit simplement :

```
    static void tester_estBissextile(){
        String[] tests = {"1600 true", "1604 true",
                          "1700 false", "1911 false"};

        System.out.println("Tests de Bissextile");
        for(int i = 0; i < tests.length; i++){
            boolean ok = tester_estBissextile(tests[i]);

            System.out.print("Test "+i+" : ");
            System.out.println(ok);
        }
    }
```

La fonction de test elle-même récupère les entrées, appelle la fonction testée et retourne un booléen qui exprime que le test est réussi ou pas :

```
    static boolean tester_estBissextile(String t){
        String[] tab = t.split(" ");
        int a = Integer.parseInt(tab[0]);
        boolean res = tab[1].equals("true");
        boolean estb = Jour.estBissextile(a);

        return estb == res;
    }
```

Une bonne habitude à prendre est de choisir les noms des fonctions de test de façon canonique en fonction du nom de la fonction testée.

Nous allons maintenant décrire comment on peut mettre à jour notre programme de Test, simplement en modifiant légèrement le tableau de test :

```
        String[] tests = {"18 3 2011 vendredi",
                          "1 1 1 erreur",
                          "19 3 2011 samedi",
                          "10 5 2011 mardi",
                          "32 1 1 erreur",
                          "1 32 1 erreur",
                          "1 1 32 erreur",
                          "-1 1 1 erreur",
                          "28 2 2011 lundi",
                          "29 2 2011 erreur",
                          "29 2 2000 mardi",
                          "29 2 2100 erreur",
                          "29 2 2008 vendredi",
                          "1 1 1500 erreur"
        };
```

Nous avons essayé d'être assez exhaustifs dans le test des cas d'erreurs. Nous avons décidé qu'il n'y avait qu'un seul type d'erreur. Quel que soit le problème dans les données, nous savons seulement qu'il y a eu une erreur, mais ça ne suffit sans doute pas à l'utilisateur. Il est conseillé en général de retourner le maximum d'information sur l'erreur rencontrée, ce qui peut permettre au programme de corriger tout seul, ou bien de renseigner suffisament l'utilisateur sur le problème.

Il existe au moins deux façons de signaler des erreurs : dans notre cas, modifier `donneesCorrectes` pour qu'elle retourne une chaîne de caractères suffit et nous laissons cela comme exercice. L'autre, plus générique, consiste à *lever* une exception, mais cela nous entraînerait trop loin pour le moment.

Cette solution suffit-elle ? Et que se passe-t-il quand l'utilisateur entre des données qui ne sont pas des nombres ? Ou pas assez de données ? Ce dernier cas est facile à traiter par modification de la fonction principale, qui devient plus réaliste :

```
    public static void main(String[] args){
        String s, sj, sm, sa;

        if(args.length < 3){
            System.out.println("Pas assez de données");
            return;
```

```
        }
        sj = args[0];
        sm = args[1];
        sa = args[2];
        s = calculerJour(sj, sm, sa);
        if(s != null)
            afficherJour(sj, sm, sa, s);
        else
            System.out.println("Données incorrectes");
        return;
    }
```

Le premier problème est résolu de manière différente. Nous avons essentiellement deux choix : ou bien nous vérifions que chaque chaîne d'entrée ne contient que des chiffres, ou bien nous laissons Java essayer de lire des entiers et nous renvoyer une erreur si tel n'est pas le cas. Par exemple, l'appel

```
unix% java Jour a b c
```

va provoquer

```
Exception in thread "main" java.lang.NumberFormatException: For input string
at java.lang.NumberFormatException.forInputString(NumberFormatException.java
at java.lang.Integer.parseInt(Integer.java:447)
at java.lang.Integer.parseInt(Integer.java:497)
at Jour.calculerJour(Jour.java:73)
at Jour.main(Jour.java:97)
```

Comment interprète-t-on cela ? La fonction `Integer.parseInt` a *lancé une exception* car la chaîne de caractères en entrée n'a pu être interprétée par Java comme un nombre entier. Le programme appelant n'a pu que renvoyer l'exception au niveau au-dessus et la machine virtuelle de Java a alors stoppé le programme en redonnant la main au système. Ce que nous voyons affiché est la pile d'exécution.

Un programme utilisateur ne saurait échouer ainsi (imaginer que ça se passe sur un satellite inaccessible... ou sur votre téléphone). Java offre la possibilité de *rattraper cette erreur* de manière plus conviviale. Cela se fait de la manière suivante :

```
    public static void main(String[] args){
        String s = null, sj, sm, sa;

        if(args.length < 3){
            System.out.println("Pas assez de données");
            return;
        }
        sj = args[0];
        sm = args[1];
        sa = args[2];
        try{
            s = calculerJour(sj, sm, sa);
        } catch(Exception e){
            System.err.println("Exception: "
```

```
                                + e.getMessage());
        }
        if(s != null)
            afficherJour(sj, sm, sa, s);
        else
            System.out.println("Données incorrectes");
        return;
    }
```

L'instruction **try ... catch** permet en quelque sorte de protéger l'exécution de la fonction qui nous intéresse. La seule façon pour la fonction d'échouer est qu'il y ait une erreur de lecture des données, ici qu'une donnée ne soit pas un entier. Dans notre cas, l'exception sera de type `NumberFormatException` et `parseInt` lève une exception de ce type qui est propagée par `calculerJour`. Cet objet contient un message qui informe sur l'erreur et nous décidons de l'afficher. Remarquez que la variable `s` n'est affectée que si aucune erreur n'est déclenchée. Dans ce cas, elle a la valeur qu'elle avait lors de son initialisation, c'est-à-dire ici **null**.

Le bon emploi des exceptions est assez complexe. Nous renvoyons aux pages web du cours pour plus d'information (page écrite par J. Cervelle).

12.3.4 Tests exhaustifs du programme

Dans certains cas, on peut rêver de faire des tests exhaustifs sur le programme. Ici, rien n'empêche de tester toutes les dates correctes possibles entre 1586 et l'an 5000 par exemple. Le seul problème à résoudre est d'écrire ce programme de test, une fois que l'on "sait" que le 1er janvier 1586 était un mercredi.

Un programme de test doit être le plus indépendant possible du programme à tester. En particulier, il ne faut pas que le programme de test utilise des fonctions du programme testé alors que c'est justement ce que l'on veut tester...

Ici, nous allons seulement utiliser le fait que les fonctions `estBissextile` et `nbJoursDansMois` ont déjà été testées (ce sont les plus faciles à tester) et qu'elles sont correctes. Ensuite, nous allons écrire une fonction qui teste tous les jours d'une année donnée, celle-ci itérant sur les mois et le nombre de jours des mois.

```
// ENTRÉE: j11 est le jour de la semaine qui commence
//         l'année a.
// SORTIE: le jour de la semaine qui suit le 31/12 de
//         l'année a.
public static int testerAnnee(int a, int j11){
    int jj = j11;

    for(int m = 1; m <= 12; m++){
        int jmax = Jour.nbJoursDansMois(m, a);
        for(int j = 1; j <= jmax; j++){
            // on construit la chaîne de test
            String t = j + " " + m + " " + a;
            t += " " + JourF.jour[jj];
            if(! testerJour(t))
                System.out.println("PB: "+ t);
            // on avance d'un jour
```

```
                jj++;
                if(jj == 7)
                  jj = 0;
            }
        return jj;
    }

    public static void testsExhaustifs(){
        int j11 = 3; // mercredi 1er janvier 1586

        for(int a = 1586; a < 2500; a++){
            System.out.println("Test de l'année "+a);
            j11 = testerAnnee(a, j11);
        }
    }
```

La fonction `testerAnnee` prend en entrée une année et le jour de la semaine correspondant au 1er janvier de l'année. En sortie, la fonction retourne le jour de la semaine du 1er janvier de l'année qui suit. À l'intérieur de la fonction, on boucle sur les mois et les jours, ainsi que sur le jour de la semaine.

12.3.5 Est-ce tout ?

Nous avons passé du temps à expliquer comment écrire un programme raisonnable basé sur le calcul du jour de la semaine correspondant à une date. La solution est satisfaisante, mais nous avons figé des choses qui pour le moment empêchent la mise à jour et l'extension du programme, notamment dans l'utilisation des entrées/sorties. Nous continuerons le développement de notre programme au chapitre 13.

12.3.6 Calendrier et formule de Zeller

Nous allons d'abord donner la preuve de la formule due au Révérend Zeller et qui résout notre problème.

Théorème 3 *Le jour J (un entier entre 0 et 6 avec dimanche codé par 0, etc.) correspondant à la date $j/m/a$ est donné par :*

$$J = (j + \lfloor 2.6m' - 0.2 \rfloor + e + \lfloor e/4 \rfloor + \lfloor s/4 \rfloor - 2s) \bmod 7$$

où

$$(m', a') = \begin{cases} (m-2, a) & \text{si } m > 2, \\ (m+10, a-1) & \text{si } m \leqslant 2, \end{cases}$$

et s (resp. e) est le quotient (resp. reste) de la division euclidienne de a' par 100, c'est-à-dire $a' = 100s + e$, $0 \leqslant e < 100$.

Commençons d'abord par rappeler les propriétés du calendrier grégorien, qui a été mis en place en 1582 par le pape Grégoire XIII : l'année est de 365 jours, sauf quand

elle est bissextile, i.e., divisible par 4, sauf les années séculaires (divisibles par 100), qui ne sont bissextiles que si elles sont divisibles par 400.

Si j et m sont fixés, et comme $365 = 7 \times 52 + 1$, la quantité J avance de 1 chaque année, sauf quand la nouvelle année est bissextile, auquel cas J progresse de 2. Il faut donc déterminer le nombre d'années bissextiles inférieures à a.

Détermination du nombre d'années bissextiles

Lemme 1 *Le nombre d'entiers de* $[1, N]$ *qui sont divisibles par* k *est* $\delta(N, k) = \lfloor N/k \rfloor$.

Démonstration : les entiers m de l'intervalle $[1, N]$ divisibles par k sont de la forme $m = kr$ avec $1 \leqslant kr \leqslant N$ et donc $1/k \leqslant r \leqslant N/k$. Comme r doit être entier, on a en fait $1 \leqslant r \leqslant \lfloor N/k \rfloor$. □

Proposition 4 *Le nombre d'années bissextiles dans* $]1600, A]$ *est*

$$\begin{aligned} B(A) &= \delta(A-1600, 4) - \delta(A-1600, 100) + \delta(A-1600, 400) \\ &= \lfloor A/4 \rfloor - \lfloor A/100 \rfloor + \lfloor A/400 \rfloor - 388. \end{aligned}$$

Démonstration : on applique la définition des années bissextiles : toutes les années bissextiles sont divisibles par 4, sauf celles divisibles par 100 à moins qu'elles ne soient multiples de 400. □

Pour simplifier, on écrit $A = 100s + e$ avec $0 \leqslant e < 100$, ce qui donne :

$$B(A) = \lfloor e/4 \rfloor - s + \lfloor s/4 \rfloor + 25s - 388.$$

Comme le mois de février a un nombre de jours variable, on décale l'année : on suppose qu'elle va de mars à février. On passe de l'année (m, a) à l'année-Zeller (m', a') comme indiqué ci-dessus.

Détermination du jour du 1er mars

Ce jour est le premier jour de l'année Zeller. Posons $\mu(x) = x \bmod 7$. Supposons que le 1er mars 1600 soit n, alors il est $\mu(n+1)$ en 1601, $\mu(n+2)$ en 1602, $\mu(n+3)$ en 1603 et $\mu(n+5)$ en 1604. De proche en proche, le 1er mars de l'année a' est donc :

$$\mathcal{M}(a') = \mu(n + (a' - 1600) + B(a')).$$

Maintenant, on détermine n à rebours en utilisant le fait que le 1er mars 2011 était un mardi. On trouve $n = 3$.

Le premier jour des autres mois

On peut précalculer le décalage entre le jour du 1er mars et le jour du 1er des mois suivants :

1er avril	1er mars+3
1er mai	1er avril+2
1er juin	1er mai+3
1er juillet	1er juin+2
1er août	1er juillet+3
1er septembre	1er août+3
1er octobre	1er septembre+2
1er novembre	1er octobre+3
1er décembre	1er novembre+2
1er janvier	1er décembre+3
1er février	1er janvier+3

Ainsi, si le 1er mars d'une année est un vendredi, alors le 1er avril est un lundi, et ainsi de suite.

On peut résumer ce tableau par la formule $\lfloor 2.6m' - 0.2 \rfloor - 2$, d'où :

Proposition 5 *Le 1er du mois m' est :*

$$\mu(1 + \lfloor 2.6m' - 0.2 \rfloor + e + \lfloor e/4 \rfloor + \lfloor s/4 \rfloor - 2s)$$

et le résultat final en découle.

Chapitre 13

Introduction au génie logiciel en Java

13.1 Modularité

Le concept de modularité est fondamental en informatique, et c'est une des seules façons de pouvoir gérer les gros programmes ou systèmes. Nous ne nous occupons pas ici de comment on découpe les programmes en modules, mais plutôt quels outils nous pouvons utiliser et comment.

Le concept de classe en Java permet déjà une certaine forme de modularité. Une classe permet de regrouper ensemble toutes les fonctions, données, algorithmes qui permettent de résoudre une tâche donnée. Bien sûr, une classe peut utiliser d'autres classes, être utilisées par d'autres, etc. L'élaboration d'une architecture de programme peut souvent être traduite en une suite (ou un arbre) de classes, qui elles-mêmes pourront être implantées. Dans ce cours, nous n'irons pas plus loin, laissant le concept d'héritage à plus tard dans le cursus.

Notons que Java permet de regrouper des classes au sein d'un paquetage (**`package`**). L'intérêt des paquetages est de limiter les conflits de noms de fonctions, de classes, en créant des *espaces de nom* et des *espace d'accessibilité*. Les fonctions publiques de ces paquetages peuvent être importées par l'intermédiaire de l'instruction **`import`**.

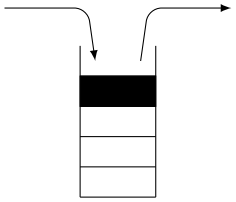
Si nous avons découpé le programme en modules, il est logique de tester chaque module de façon séparée, c'est ce qu'on appelle les *tests unitaires*. Quand on a terminé, on passe aux *tests d'intégration* qui eux testent tout le programme.

13.2 Les interfaces de Java

Les interfaces de Java sont un mécanisme qui permet de définir le comportement d'une classe par l'intermédiaire des fonctions qui doivent y être programmées. Il s'agit d'une sorte de contrat passé par l'interface avec une classe qui l'implantera. Un programme pourra utiliser une interface, sans réellement savoir quelle classe réalisera l'implantation, du moment que celle-ci est conforme à ce qui est demandé. Nous allons voir deux exemples classique, ceux des piles et des files.

13.2.1 Piles

Nous avons déjà croisé les piles dans le chapitre 4. Les informations sont stockées dans l'ordre de leur arrivée. Le premier élément arrivé se trouve dans le fond. Le dernier arrivé peut sortir, ainsi qu'il est montré dans le dessin qui suit :



Avant de donner des implantations possibles des piles, nous pouvons nous demander de quoi exactement nous avons besoin. Nous avons besoin de créer ou détruire une pile, empiler l'élément au-dessus de la pile, dépiler l'élément qui est au-dessus. Il est logique de pouvoir tester si une pile est vide ou non. Nous avons en fait défini une pile par le comportement qu'elle doit avoir, pas par sa représentation. On parle classiquement de *type de données abstrait*. On voit que Java nous fournit un moyen de traiter ce concept à l'aide d'**interface**. Dans le cas présent, en nous limitant pour l'instant à une pile contenant des entiers de type **int** :

```
public interface Pile{
    public boolean estVide();
    public void empiler(int x);
    public int depiler();
}
```

Un programme de test pour une telle interface commencera par :

```
public class TesterPile{

    public static void testerPile(Pile p){
        for(int n = 0; n < 10; n++)
            p.empiler(n);
        while(! p.estVide())
            System.out.println(p.depiler());
    }
```

et on pourra tester toute classe implantant `Pile`.

Arrêtons-nous un instant sur cette fonction. Nous voyons que nous pouvons écrire du code qui dépend juste des propriétés des piles, pas de leur implantation. C'est justement ce dont on a besoin pour travailler à plusieurs : une fois les interfaces spécifiées, les uns peuvent travailler à leur utilisation, les autres à la réalisation concrète des choses.

Nous allons donner deux implantations possibles de cette classe, la première à l'aide d'un tableau, la seconde à l'aide d'une liste. Dans les deux cas, nous cachons la gestion mémoire à l'intérieur des deux classes. De même, nous ne rendons pas accessibles la structure de données utilisée.

Nous pouvons définir la classe `PileTab` en représentant celle-ci par un tableau dont la taille sera créée à la construction et éventuellement agrandie (modulo recopie) lors d'une insertion.

```
public class PileTab implements Pile{

    private int taille, hauteur;
    private int[] t;

    public PileTab(){
        this.taille = 16;
        this.t = new int[this.taille];
        this.hauteur = -1;
    }

    public boolean estVide(){
        return this.hauteur == -1;
    }

    public void empiler(int x){
        this.hauteur += 1;
        if(this.hauteur > this.taille){
            int[] tmp = new int[2 * this.taille];

            for(int i = 0; i < this.hauteur; i++)
                tmp[i] = this.t[i];
            this.taille = 2 * this.taille;
            this.t = tmp;
        }
        this.t[this.hauteur] = x;
    }

    public int depiler(){
        return this.t[this.hauteur--];
    }
}
```

Par convention, la pile vide sera caractérisée par une hauteur égale à -1. Si la hauteur est positive, c'est l'indice où le dernier élément arrivé a été stocké.

On peut également utiliser des listes, l'idée étant de remplacer un tableau de taille fixe par une liste dont la taille varie de façon dynamique. La seule restriction sera la taille mémoire globale de l'ordinateur. Dans ce qui suit, on va utiliser notre classe `ListeEntier` et modifier le type de la pile, qui va devenir simplement :

```
public class PileListe implements Pile{
    private ListeEntier l;

    public PileListe(){
        this.l = null;
    }

    public boolean estVide(){
        return this.l == null;
```

```
    }

    public void empiler(int x){
        this.l = new ListeEntier(x, this.l);
    }

    public int depiler(){
        int c = this.l.contenu;

        this.l = this.l.suivant;
        return c;
    }
}
```

Le programme de test pourra alors contenir la fonction principale suivante

```
    public static void main(String[] args){
        System.out.println("Avec un tableau");
        testerPile(new PileTab());
        System.out.println("Avec une liste");
        testerPile(new PileListe());
    }
```

13.2.2 Files d'attente

Le premier exemple est celui d'une file d'attente à la poste. Là, je dois attendre au guichet, et au départ, je suis à la fin de la file, qui avance progressivement vers le guichet. Je suis derrière un autre client, et il est possible qu'un autre client entre, auquel cas il se met derrière moi.

D'un point de vue utilisation, nous n'avons besoin que de l'interface suivante[1] :

```
public interface FIFO{
    public boolean estVide();
    public void ajouter(int x);
    public int supprimer();
}
```

Nous n'allons donner ici que l'implantation de la FIFO par une liste, laissant le cas du tableau et du programme de test en exercice. Le principe est de gérer une liste dont on connaît la tête (appelée ici `debut`) et la référence de la prochaine cellule à utiliser. Les fonctions qui suivent mettent à jour ces deux variables.

```
public class FIFOListe implements FIFO{
    private ListeEntier debut, fin;

    public FIFOListe(){
        this.debut = this.fin = null;
    }
```

[1] Nous avons pris le nom anglais FIFO, car File est un mot-clef réservé de Java.

```
    public boolean estVide(){
        return this.debut == this.fin;
    }

    public void ajouter(int x){
        if(this.debut == null){
            this.debut = new ListeEntier(x, null);
            this.fin = this.debut;
        }
        else{
            this.fin.suivant = new ListeEntier(x, null);
            this.fin = this.fin.suivant;
        }
    }

    public int supprimer(){
        int c = this.debut.contenu;

        this.debut = this.debut.suivant;
        return c;
    }
}
```

13.2.3 Les génériques

Nous avons utilisé jusqu'à présent des types d'entiers pour simplifier l'exposition. Il n'y a aucun problème à utiliser des piles d'objets si le cas s'en présente.

Il peut être intéressant de chercher à faire un type de pile qui ne dépende pas de façon explicite du type d'objet stocké. En Java, on peut réaliser cela avec des types génériques dont nous allons esquisser l'usage.

L'exemple le plus simple est celui d'une liste, qui *a priori* n'a pas besoin de connaître le type de ses éléments. Pour des raisons techniques, faire une pile d'`Object` n'est pas suffisant, car il faut que la liste puisse travailler sur des objets de *même type*, même si ce type n'est pas connu. On peut utiliser la classe `LinkedList` de Java de la façon suivante :

```
    LinkedList<Integer> l = new LinkedList<Integer>();
```

qui déclare `l` comme étant une liste d'`Integer`. On peut remplacer `Integer` par n'importe quel type. On peut alors utiliser les fonctions classiques :

```
    l.add(new Integer("111111"));
    Integer n = l.getFirst();
```

Utiliser les classes génériques donne accès à une grande partie de la richesse des bibliothèques de Java, offrant ainsi un réel confort au programmeur. À titre d'exemple, la syntaxe

```
    for(Integer a : l)
```

```
        System.out.println(a);
```

nous permet d'itérer sur tous les composants de la liste de façon agréable. Nous verrons à la section 14.5.1 comment ce mécanisme nous simplifie la vie.

Notez que la classe `LinkedList` définit une méthode `get(int index)` mais qu'il serait extrémement maladroit de vouloir écrire l'itération sous la forme

```
    for(int i = 0; i < l.size(); i++){
        Integer a = l.get(i);
        System.out.println(a);
    }
```

La classe `LinkedList` implante l'interface `List`

```
    LinkedList<Integer> implements List<Integer>{...}
```

correspond à la définition du modèle "collection séquentielle indexée". Une autre implantation de `List` est `ArrayList` qui correspond à la notion de tableau dynamique (sa taille est doublée automatiquement quand il est plein).

La classe `LinkedList` implante aussi l'interface `Queue` (terme anglais pour file) et on peut écrire ainsi :

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

Cela dispense les concepteurs de la librairie Java d'écrire une implantation particulière de `Queue` alors que `LinkedList` est très bien pour cela. Pour l'utilisateur, cela apporte plus de lisibilité et de sûreté de son code car il ne peut appliquer à `q` que les quelques fonctions définies dans `Queue`.

13.3 Retour au calcul du jour de la semaine

Nous poursuivons ici l'exemple du chapitre 12, en donnant un exemple de création de paquetage.

Nous allons décider de créer un paquetage `calendrier` qui va regrouper une classe traitant le calendrier grégorien (et donc appelée `Gregorien`), puis une classe `AfficherDate` qui affiche la réponse dans un terminal. C'est à ce moment qu'on peut se poser la question de l'internationalisation. Ce n'est pas à classe `Gregorien` de s'occuper de la traduction en langage de sortie. Ceci nous amène à modifier le type de la fonction `calculerJour` et à propager cette nouvelle sémantique.

En Unix, Java nous impose de créer un répertoire `calendrier`, qui va contenir les deux classes `Gregorien` (dans un fichier `Gregorien.java`) et `AfficherDate` (dans `AfficherDate.java`). Chacun de ces fichiers doit commencer par la commande

```
package calendrier;
```

qui les identifie comme appartenant à ce paquetage. Le fichier `Gregorien.java` contient donc

```
package calendrier;

public class Gregorien{
```

```
public static int[] JOURS_DANS_MOIS = {31, 28, 31, 30, 31,
                                       30, 31, 31, 30, 31,
                                       30, 31};

static boolean estBissextile(int a){
    if((a % 4) != 0)
        return false;
    if((a % 100) != 0)
        return true;
    return ((a % 400) == 0);
}

public static int nbJoursDansMois(int m, int a){
    if((m != 2) || !estBissextile(a))
        return JOURS_DANS_MOIS[m-1];
    else
        return 29;
}

public static boolean donneesCorrectes(int j, int m,
                                       int a){
    if(a <= 1584)
        return false;
    if((m < 1) || (m > 12))
        return false;
    if((j < 1) || (j > nbJoursDansMois(m, a)))
        return false;
    return true;
}

// Calcul du jour de la semaine correspondant
// à la date j / m / a, sous la forme d'un entier
// J tel que 0 <= J <= 6, avec 0 == dimanche, etc.
static int jourZeller(int j, int m, int a){
    int mz, az, e, s, J;

    // calcul des mois/années Zeller
    mz = m-2;
    az = a;
    if(mz <= 0){
        mz += 12;
        az--;
    }
    // az = 100*s+e, 0 <= e < 100
    s = az / 100;
    e = az % 100;
    // la formule du révérend Zeller
```

```
        J = j + (int)Math.floor(2.6*mz-0.2);
        J += e + (e/4) + (s/4) - 2*s;
        // attention aux nombres négatifs
        if(J >= 0)
            J %= 7;
        else{
            J = (-J) % 7;
            if(J > 0)
                J = 7-J;
        }
        return J;
    }

    // SORTIE: un jour entre 0 et 6 ou -1 en cas d'erreur
    public static int calculerJour(String sj, String sm,
                                   String sa){

        int j, m, a;

        j = Integer.parseInt(sj);
        m = Integer.parseInt(sm);
        a = Integer.parseInt(sa);
        if(donneesCorrectes(j, m, a))
            return jourZeller(j, m, a);
        else
            return -1;
    }
}
```

Le nom de la classe est désormais `calendrier.Gregorien`.

Le fichier `AfficherDate.java` contient

```
package calendrier;

public class AfficherDate{

    public final static String[] JOUR = {"dimanche", "lundi",
                                         "mardi", "mercredi",
                                         "jeudi", "vendredi",
                                         "samedi"};

    public static void afficherJour(String sj, String sm,
                                    String sa, int jZ,
                                    String lg){
        if(lg.equals("fr")){
            String s = JOUR[jZ];
            System.out.print("Le "+sj+"/"+sm+"/"+sa);
            System.out.println(" est un "+s+".");
        }
```

```
    }

}
```

Nous avons ébauché un début d'internationalisation du programme en rajoutant un paramètre d'affichage (le choix par la langue `lg`), ainsi que les noms de jours en français. Notons que pour aller plus loin, il faut écrire des fonctions d'affichage par langue (la grammaire n'est pas la même en anglais, etc.).

On les utilise alors comme suit, dans le fichier `Jour.java`, au même niveau que le répertoire `calendrier`

```
// importation de toutes les classes du paquetage
import calendrier.*;

public class Jour{

    public static void main(String[] args){
        String sj, sm, sa;
        int jZ = -1;

        if(args.length < 3){
            System.out.println("Pas assez de données");
            return;
        }
        sj = args[0];
        sm = args[1];
        sa = args[2];
        try{
            jZ = Gregorien.calculerJour(sj, sm, sa); // (*)
        } catch(Exception e){
            System.err.println("Exception: "
                               + e.getMessage());
        }
        if(jZ != -1) // (*)
            AfficherDate.afficherJour(sj, sm, sa, jZ, "fr");
        else
            System.out.println("Données incorrectes");
        return;
    }
}
```

Les appels aux classes sont modifiées dans les lignes marquées d'une étoile (*) ci-dessus. Et nous avons demandé un affichage en français.

Nous laissons au lecteur le soin de mettre à jour les fonctions de test de ce programme pour tenir compte du changement de type de certaines fonctions, ainsi que pour l'internationalisation.

Maintenant, nous avons isolé chacune des phases du programme dans des classes bien définies, que l'on peut réutiliser dans d'autres contextes, par exemple celle d'une application X11, ou bien encore dans une application de téléphone android, une application de type réseau, etc.

Chapitre 14

Modélisation de l'information

Dans ce chapitre[1], nous allons passer en revue différentes façons de modéliser, stocker et traiter l'information, à travers de nombreux exemples, en utilisant les objets et structures définies dans les parties précédentes.

14.1 Modélisation et réalisation

14.1.1 Motivation

Nous considérons ici que de l'information consiste en des données (ou de l'information plus parcellaire) et des relations entre ces données. Certains modèles d'information peuvent aussi préciser des contraintes que les données ou les relations doivent vérifier. Enfin, un modèle inclut une définition des opérations possibles et permettant de maintenir les contraintes.

Pour un même modèle, il existe souvent plusieurs structures de données qui permettent de le réaliser. La structure idéale n'existe pas toujours, et on doit souvent choisir de privilégier l'efficacité de telle ou telle opération au détriment d'autres. Il est même possible de faire cohabiter plusieurs réalisations du même modèle. Séparer modélisation et réalisation est donc fondamental.

Dans un langage comme Java, la séparation entre modèle et réalisation est à peu près décrite par le tandem « interface/implémentation ». Il manque pourtant le moyen de spécifier les contraintes et d'exiger qu'une réalisation les implante. Cela reste souvent sous la format d'un « contrat », exprimé dans la documentation et que le programmeur de la structure de données, d'une part, et l'utilisateur, d'autre part, doivent s'efforcer de respecter chacun à son niveau de responsabilité.

Alors qu'il est généralement facile d'identifier les données qu'un programme devra manipuler, il est généralement plus difficile de recenser de manière exhaustive les relations et les contraintes. Comme on l'a vu au chapitre 12, il est pourtant impératif de faire cela dès la première étape de la conception car se rendre compte d'éventuels oublis ou imprécisions, lors de la programmation ou lors des tests, peut conduire à d'énormes pertes de temps. Il est alors rassurant et avantageux de pouvoir facilement relier son problème à un modèle type, quand c'est possible. Cela permet ensuite, sans trop se poser de questions, de prendre "sur l'étagère" les bons composants pour le réaliser.

[1] chapitre écrit avec P. Chassignet

14.1.2 Exemple : les données

Un exemple particulier est la notion de multiplet (élément d'un produit cartésien) qui permet d'associer des données de types disparates mais connus et en nombre fini. La traduction en Java est une classe qui est une représentation du produit cartésien, chaque objet de cette classe pouvant représenter un multiplet particulier. Néanmoins la modélisation par multiplet n'implique pas nécessairement une correspondance directe entre les composants du multiplet et les champs de l'objet.

Par exemple, si on considère la représentation des nombres complexes, on peut considérer qu'un nombre complexe associe quatre données qui sont sa partie réelle, sa partie imaginaire, son module et son argument. Néanmoins, il existe les règles bien connues qui relient ces quatre données et il serait maladroit de représenter un nombre complexe comme un objet ayant quatre champs.

Là aussi la séparation entre modèle et réalisation nous permet de concilier les deux points de vue. On peut ainsi dire qu'un nombre complexe est défini par une interface Java qui comporte huit méthodes, une pour obtenir la valeur, une pour la modifier et ce pour les quatre grandeurs. Ensuite, on peut envisager une réalisation basée sur la représentation cartésienne et une autre sur la représentation polaire (ie. deux classes qui implantent l'interface). Chacune de ces classes ne définit que deux champs (private), avec quatre méthodes qui y accèdent directement et quatre autres méthodes qui font le changement de représentation requis à chaque accès. Le choix d'instancier un objet plutôt de l'une ou de l'autre de ces classes dépend ensuite de l'application. Par exemple, si l'on a majoritairement des multiplications de nombres complexes à traiter, on priviligiera la représentation polaire.

Sans changer de modèle, on peut également réaliser une classe pour définir des constantes complexes où les méthodes pour modifier ne font rien ou lance une exception.

Nous allons maintenant considérer des associations de données de même type. Ce type peut être un type élémentaire, un produit cartésien déjà défini ou un super-type (comme une interface Java) qui permet de manipuler de manière uniforme des types a priori disparates.

14.2 Conteneurs, collections et ensembles

Nous introduisons ici un type abstrait très général qui est celui de conteneur pour lequel sont définies l'ajout et la suppression d'un élément, le test d'appartenance d'un élément et l'itération c'est-à-dire un mécanisme pour considérer un à un tous les éléments du conteneur. On ajoute généralement un accès direct au nombre d'éléments contenus, pour ne pas avoir à calculer cela par une itération, ainsi que la possibilité de vider le conteneur en une seule opération.

La collection est le type de conteneur le plus vague dans lequel il est permis de placer plusieurs occurrences d'une même donnée. Lors d'une itération de la collection, cette donnée sera alors considérée plusieurs fois.

L'ensemble, correspondant à la définition classique en mathématiques, est une collection avec une contrainte d'unicité. Cette contrainte est généralement assurée dans la méthode d'ajout qui doit procéder à l'équivalent d'un test d'appartenance avant d'ajouter effectivement. Avec des réalisations naïves, par exemple, un tableau ou une liste chaînée, le test d'appartenance et la contrainte d'unicité coûtent cher car il faut itérer sur tout le conteneur pour vérifier l'absence d'un élément. Des structures d'arbre

particulières permettent de réaliser ce test en temps logarithmique et le hachage permet de le faire en temps quasi-constant.

A priori, une collection ou un ensemble ne sont pas ordonnés, c'est-à-dire que l'ordre d'énumération de leurs éléments n'est pas défini. En fait la collection non ordonnée n'existe pas. Selon la structure de données qui est utilisée pour réaliser le stockage, il existe toujours un ordre déterministe d'énumération, sauf à ajouter explicitement de l'aléatoire lors de l'itération. Une exception notable est le cas où la structure sous-jacente utilise le hachage et l'ordre d'énumération, bien que déterministe, est alors difficilement prédictible.

On va maintenant considérer des conteneurs ordonnés. Selon les procédés les plus courants, l'ordre des éléments dans le conteneur est défini soit par l'ordre ou la position de leur ajout, soit par une relation d'ordre définie sur les données.

14.2.1 Collections séquentielles

Il s'agit des collections où l'ordre des éléments est principalement défini par l'ordre de leur ajout. Notons que cela permet de considérer de nouvelles opérations d'ajout, d'accès ou de suppression qui utilisent le numéro d'ordre (la position) dans le conteneur pour désigner où opérer. On parle alors de collection séquentielle indexée. Des réalisations particulièrement simples sont possibles à partir de tableaux ou de listes chaînées mais il faut faire attention aux fonctions que l'on tient à privilégier car elles n'ont pas toutes la même efficacité selon la structure sous-jacente.

Il existe des cas où on peut restreindre les opérations permises à un élément particulier. Par exemple, on veut que l'accès et la suppression ne soient possibles que sur le dernier ou le premier ajouté. Ce sont respectivement les piles et les files. Le fait de les identifier comme des modèles à part permet de clarifier l'expression des algorithmes qui les utilisent. Le fait de leur dédier des réalisations particulières qui limitent les opérations permises permet d'éviter les erreurs et parfois d'optimiser.

On retrouve ainsi les piles et les files présentées au chapitre 13.

14.2.2 Collections ordonnées

Il s'agit des collections où l'ordre des éléments est défini par une relation d'ordre total sur ces éléments et donc indépendant de l'ordre dans lequel ils sont ajoutés dans la collection. Une réalisation efficace utilise généralement des structures d'arbres équilibrés.

Si la collection est relativement statique, c'est-à-dire qu'elle est constituée au départ et qu'ensuite, on se contente de la consulter, une alternative consiste à la former en triant les éléments d'une collection séquentielle.

De même qu'une file peut être vue comme une collection séquentielle particulière, on peut définir la file de priorité comme une collection ordonnée dont on restreint les opérations permises.

Exemple : file de priorité

Dans certains cas, une file d'attente ne suffit pas à nos besoins. Les données peuvent arriver avec une *priorité*. Par exemple, dans une file d'impression, on peut décider qu'un utilisateur est prioritaire. Un ordonnanceur de système d'exploitation a également

plusieurs files de priorité à gérer. C'est comme ça qu'un bon chef doit également gérer ses affaires courantes...

Que demandons-nous à notre modèle ? Il suffit de deux actions notables, la première de stocker un nouvel élément avec sa priorité, la seconde de pouvoir demander quelle est la tâche prioritaire suivante à traiter. Accessoirement, nous pourrions insister pour que cette gestion soit rapide, mais le typage ne suffit pas pour cela. Il est clair que si le nombre de tâches est fixé une fois pour toutes, on peut trier le tableau des tâches en fonction de leur priorité et nous n'avons plus rien à faire. Ce qui nous intéresse ici est le cas où des tâches arrivent de façon dynamique, et sont traitées de façon dynamique (par exemple l'impression de fichiers avec priorité). L'interface désirée, qui ressemble à celle d'une file d'attente normale est

```
public interface FileDePriorite{
    public boolean estVide();
    public void ajouter(String x, int p);
    public int tacheSuivante();
}
```

Nous laissons au lecteur le soin d'inventer une classe qui implante cette interface à l'aide d'un tableau, en s'inspirant de ce qui a été fait pour les piles et les files. Nul doute que le résultat aura une complexité proche de $O(n^2)$ si n est le nombre de tâches traitées. L'utilisation d'un tas (cf. 9.3.3) nous permet de faire mieux, avec un tableau de taille n et des complexités en $O(n \log n)$.

On peut implanter une file de priorité opérant sur des fichiers et des priorités en raisonnant sur des couples `(String, int)` et il est facile de modifier la classe `Tas` (cf. section 9.3.3) en `TasFichier`. On pourrait alors créer

```
public class Impression implements FileDePriorite{
    TasFichier tas;

    public Impression(){
        this.tas = new TasFichier();
    }

    public boolean estVide(){
        return this.tas.estVide();
    }

    public void ajouter(String f, int priorite){
        this.tas.ajouter(f, priorite);
    }

    public String tacheSuivante(){
        return this.tas.tacheSuivante();
    }
}
```

Nous laissons au lecteur le soin de terminer.

14.3 Associations

Formellement une table d'association peut être vue comme une fonction qui, a un ensemble fini de données dites les clefs, associe d'autres données. On peut aussi considérer cela comme un ensemble de couples (clef, données) avec une fonction particulière qui consiste à retrouver les données associées à une clef.

Il y a diverses réalisations possibles. Celle qui consisterait à utiliser la représentation naïve par une liste de couples est maladroite puisqu'elle conduit à programmer la recherche séquentielle du couple ayant la clef considérée. La solution la plus efficace est une table de hachage organisée suivant les clefs.

Si l'on a besoin de maintenir un ordre particulier sur les clefs, par exemple un annuaire trié par ordre alphabétique, le concept de table est précisé comme étant ordonné. Son interprétation au niveau modélisation comme "ensemble ordonné de couples (clef, données) avec une fonction de recherche particulière par la clef" doit alors suggérer au programmeur que sa bonne réalisation est l'arbre de couples (clef, données) organisé en un arbre binaire de recherche sur les clefs qui permet une recherche relativement rapide.

14.4 Information hiérarchique

14.4.1 Exemple : arbre généalogique

Une personne `p` a deux parents (une mère et un père), qui ont eux-mêmes deux parents. On aimerait pouvoir stocker facilement une telle relation. Une structure de donnée qui s'impose naturellement est celle d'arbre. Il est à noter que la relation père-fils dans cette application est à l'inverse de celle qui est employée par les informaticiens pour décrire la structure de données en arbre. Cet exemple particulièrement frappant de conflit de terminologies illustre combien il est important de bien séparer les étapes conceptuelles entre la définition et la réalisation. Illustrons notre propos par un dessin, construit grâce aux bases de données utilisées dans le logiciel GeneWeb réalisé par Daniel de Rauglaudre[2]. On remarquera qu'en informatique, on a tendance à dessiner les arbres la racine en haut.

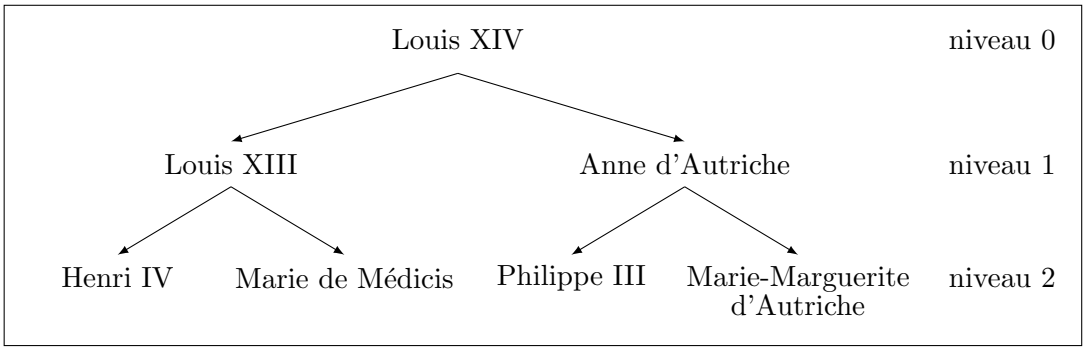


Fig. 14.1 – Un arbre généalogique.

[2] `http ://cristal.inria.fr/~ddr/GeneWeb/`

Une réalisation possible reprend la structure de tas vue en 9.3.3. Pour mémoire, on utilise un tableau `a`, de telle sorte que `a[1]` (au niveau 0) soit la personne initiale et on continue de proche en proche, en décidant que `a[i]` aura pour père `a[2*i]`, pour mère `a[2*i+1]`, et pour enfant (si $i > 1$) la case `a[i/2]`. Néanmoins, une telle structure de données ne permet pas de répondre facilement à une question comme "qui est le père d'Anne d'Autriche ?". Une modélisation par des tables d'association, une pour chaque relation "père", "mère", "enfant", est alors préférable. Notons aussi que ce que l'on cherche à faire avec de l'information hiérarchique ressemble beaucoup à ce que l'on va voir en 14.5. Ce n'est pas étonnant puisque les arbres peuvent être vus comme des graphes avec des contraintes particulières.

14.4.2 Autres exemples

On trouve également ce type de représentation hiérarchique dans la classification des espèces ou par exemple un organigramme administratif.

14.5 Quand les relations sont elles-mêmes des données

Que faire quand les liens sont plus complexes, comme par exemple quand on doit représenter la carte d'un réseau routier, un circuit électronique, la liste de ses amis ? On utilise alors souvent des graphes, qui permettent de coder de telles informations. Par exemple, on peut imaginer gérer une carte du réseau ferré en stockant les gares de France et les différentes distances entre ces gares.

14.5.1 Un exemple : un réseau social

Nous allons prendre comme exemple directeur le cas de réseaux[3] d'amis[4]. De fait, nous allons donner une autre implantation d'un graphe avec des applications légèrement différentes.

Spécification

Un ami est un membre du réseau. Un membre du réseau a un nom et une collections d'amis. Nous allons choisir d'implanter la relation "je suis l'ami de". Nous supposerons que si Claude a pour ami Dominique, alors la réciproque est vraie (la relation est *symétrique*).

Que demander au réseau ? D'abord de pouvoir rajouter de nouveaux membres. Un nouvel arrivant peut s'incrire comme membre isolé, ou bien arriver comme étant ami d'un membre. On considère que l'adresse électronique est suffisante pour indentifier un membre de façon unique.

Choix des structures de données

Comme nous voulons garder l'identification unique des membres, il nous faut un moyen de tester si un membre existe déjà, et ce test doit être rapide. Il nous faut donc un ensemble de membres qui supporte les opérations d'ajout et d'appartenance (ainsi

[3]Toute ressemblance avec des réseaux sociaux existant ne pourrait être que fortuite.

[4]Nous prenons ici ami dans un sens neutre.

que la suppression, mais nous laissons cela de côté pour le moment), ce qui tend vers une solution à base de hachage.

Programmation

On commence par les plus classiques, qui codent les membres et les listes d'amis, en utilisant la classe `LinkedList` déjà présentée à la section 13.2.3.

```
import java.util.*; // nécessaire pour utiliser LinkedList

public class Membre{
    String nom;
    LinkedList<Membre> lamis;

    public Membre(String n){
        this.nom = n;
        this.lamis = new LinkedList<Ami>();
    }

    public String toString(){
        return this.nom;
    }
}
```

Nous en avons profité pour définir des méthodes pratiques, comme `toString` qui permet d'afficher simplement un membre. La gestion d'une amitié unidirectionnelle (ajout d'un membre dans la liste d'amis d'un autre membre) est simple :

```
    public void ajouterAmitie(Membre b){
        this.lamis.add(b);
    }
```

Passons à l'implantation de la classe `ReseauSocial`. Nous allons utiliser une table de hachage pour stocker les membres déjà présents, ce qui va nous simplifier la gestion.

```
import java.util.*;

public class ReseauSocial{
    private String nom;
    private HashMap<String, Membre> hm;

    public ReseauSocial(String n){
        this.nom = n;
        this.hm = new HashMap<String, Membre>();
    }

    // retourne le membre dont n est le nom
    public Membre deNom(String n){
        return this.hm.get(n);
    }
```

```
    public boolean estMembre(String nom){
        return this.hm.containsKey(nom);
    }

    public void creerMembre(String nom){
        if(! this.estMembre(nom))
            this.hm.put(nom, new Membre(nom));
    }

    public void ajouterAmitie(String nom_a, String nom_b){
        Membre a = this.hm.get(nom_a);
        Membre b = this.hm.get(nom_b);
        a.ajouterAmitie(b);
        b.ajouterAmitie(a);
    }
}
```

Un membre n'est créé que s'il n'existe pas déjà. On ajoute une amitié de manière symétrique (notons que nous pourrions provoquer une exception dans le cas où `a` ou `b` ne seraient pas membres, mais nous simplifions ici).

Pour afficher tous les membres présents dans le réseau, on utilise simplement

```
    public void afficherMembres(){
        for(Membre a : this.hm.values())
            System.out.println(a);
    }
```

De même, nous pouvons afficher toutes les amitiés :

```
    public void afficherAmities(){
        for(Membre a : this.hm.values()){
            System.out.print("Les amis de "+a+" :");
            a.afficherAmities();
            System.out.println();
        }
    }
```

à condition d'avoir implanté dans la classe `Membre`.

```
    public void afficherAmities(){
        for(Membre a : this.lamis)
            System.out.print(a + " ");
        System.out.println();
    }
```

Nous avons tout ce qu'il faut pour tester nos classes et créer un réseau à nous

```
public class FB311{

    public static void main(String[] args){
        ReseauSocial RS = new ReseauSocial("FB311");
```

```
        String[] inf311 = {"s@", "d@", "r@", "p@", "2@"};

        RS.creerMembre("m@");
        RS.creerMembre("g@");
        RS.creerMembre("c@");
        RS.creerMembre("A@");
        RS.creerMembre("Z@");
        RS.creerMembre("E@");
        RS.creerMembre("F@");

        RS.ajouterAmitie("m@", "g@");
        RS.ajouterAmitie("m@", "c@");
        RS.ajouterAmitie("c@", "g@");
        RS.ajouterAmitie("c@", "A@");
        RS.ajouterAmitie("A@", "Z@");

        RS.ajouterAmitie("E@", "F@");

        for(int i = 0; i < inf311.length; i++){
            RS.creerMembre(inf311[i]);
            RS.ajouterAmitie("m@", inf311[i]);
        }

        System.out.println("Voici tous les membres en stock");
        RS.afficherMembres();

        RS.afficherAmities();
    }
}
```

qui va nous fournir

```
Voici tous les membres en stock
E@
s@
F@
r@
2@
p@
d@
m@
Z@
A@
c@
g@
Les amis de E@ : F@
Les amis de s@ : m@
Les amis de F@ : E@
Les amis de r@ : m@
Les amis de 2@ : m@
```

```
Les amis de p@ : m@
Les amis de d@ : m@
Les amis de m@ : g@ c@ s@ d@ r@ p@ 2@
Les amis de Z@ : A@
Les amis de A@ : c@ Z@
Les amis de c@ : m@ g@ A@
Les amis de g@ : m@ c@
```

ce qui correspond au dessin de la figure 14.2, où chaque trait reliant deux membres symbolise une amitié.

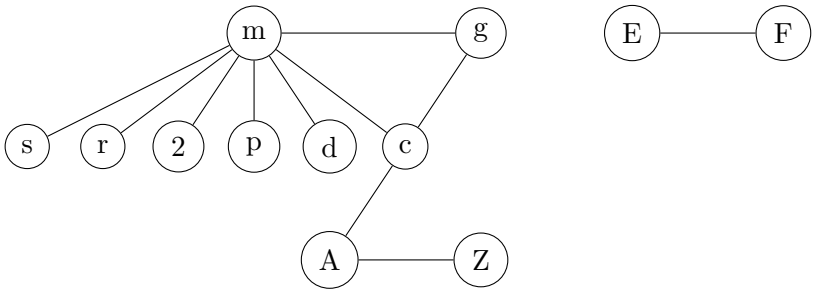


FIG. 14.2 – Le graphe exemple.

Passons à quelque chose de plus compliqué. Nous souhaitons maintenant afficher les amis des amis, au sens où nous faisons l'union ensembliste. La première fonction qui vient à l'esprit est la suivante (dans la classe `Membre`) :

```
    public void afficherAmisDeMesAmis(){
        for(Ami a : this.lamis)
            a.afficherAmities();
    }
```

Les lignes

```
        Membre m = RS.deNom("m@");
        System.out.println("Voici les amis des amis de "+m);
        m.afficherAmisDeMesAmis();
        System.out.println();
```

nous donnent

```
Voici les amis des amis de m@
g@ m@ c@
c@ m@ g@ A@
s@ m@
d@ m@
r@ m@
p@ m@
2@ m@
```

La réponse est très redondante et à peu près sans intérêt (en plus de ne pas donner l'ensemble voulu). Pour résoudre ce problème, nous allons regarder sur un dessin le comportement que nous souhaiterions. Les amis de `m@` sont faciles à trouver, nous les avons cerclés sur le dessin de la figure 14.3 (avec la convention que `m@` est son propre ami).

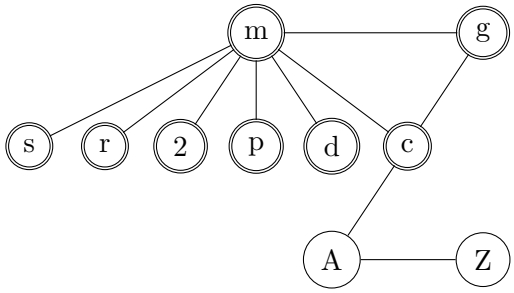


FIG. 14.3 – Le graphe exemple avec les amis à distance 1.

Pour trouver les amis des amis, il suffit maintenant d'ajouter à cet ensemble tous les amis non encore visités. Autrement dit, dans un premier temps, on construit la liste des amis à *distance 1* de `m@`, puis on parcourt cette liste pour rajouter des membres non encore vus. Pour tester si un membre a déjà été vu ou non, on utilise une table de hachage.

```
    public void afficherAmisDeMesAmis2(){
        HashSet<Membre> dejavus = new HashSet<Membre>();
        LinkedList<Membre> l = new LinkedList<Membre>();

        dejavus.add(this);
        // on traite le cas des amis
        for(Membre a : this.lamis){
            l.add(a);
            System.out.print(" " + a);
            dejavus.add(a);
        }
        // cas des amis des amis
        for(Membre a : l)
            for(Membre b : a.lamis)
                if(! dejavus.contains(b)){
                    System.out.print(" " + b);
                    dejavus.add(b);
                }
    }
```

qui donne le résultat voulu

```
Voici les amis des amis de m@
 g@ c@ s@ d@ r@ p@ 2@ A@
```

14.6 Automates

Pour le moment, nous avons considéré des données figées dans le temps. On peut aussi modéliser facilement des processus plus compliqués, par exemple le fonctionnement d'une machine, d'un programme, etc.. Les *automates* permettent de modéliser des fonctionnements simples.

14.6.1 L'exemple de la machine à café

Regardons une machine à café. La partie de son fonctionnement qui nous intéresse est celui où l'on doit vérifier qu'un client a bien mis la somme demandée, sachant qu'un café vaut 30 centimes, et qu'il a droit à des pièces de 10 ou 20 centimes.

Dans l'état initial, la machine attend un client, et est prête à lui servir un café. En fonction de la première pièce entrée par l'utilisateur, l'état de la machine change : si un utilisateur entre une pièce de 20 centimes, cet état devient "j'ai reçu une pièce de 20". Si maintenant la machine reçoit une pièce de 10, la somme totale est 30, la machine peut lancer la préparation du café, puis revenir à l'état initial où elle attend le prochain client. Si la machine reçoit 10 centimes, elle passe dans l'état "j'ai reçu pour l'instant 20 centimes". Si elle reçoit maintenant 10 centimes, elle passe dans l'état de service comme en 10.

On modélise ce fonctionnement par le graphe de la figure 14.4 : l'état initial est le sommet i ; l'état 1 est codé par le sommet 1, et on rajouté sur l'arc $(i, 1)$ l'information "j'ai reçu 20 centimes". On trace un arc de 1 vers le sommet 2, qui symbolise l'état de délivrance du café avant retour à l'état i. On complète sans difficulté les autres états et arcs.

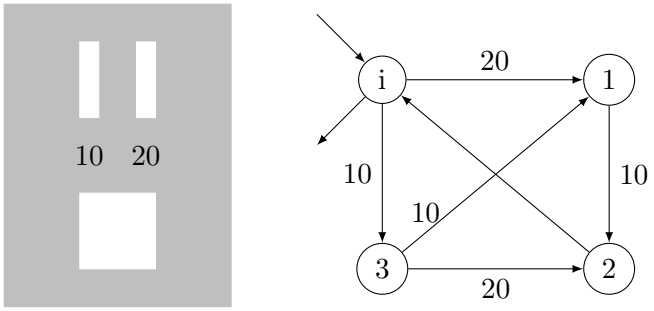


FIG. 14.4 – Un automate.

14.6.2 Définitions

Un *automate fini* A est un quintuplet $A = (Q, \Sigma, \Delta, I, F)$ où :
Σ est un alphabet donné,
Q est un ensemble *fini* d'états,
$I \subset Q$ est l'ensemble des *états initiaux*,
$F \subset Q$ est l'ensemble des *états finaux*,
Δ est un sous-ensemble de $Q \times \Sigma \times Q$, *ensemble des transitions.*

Il est commode de représenter un automate par un graphe, c'est-à-dire un ensemble de nœuds (sommets) reliés par des flèches (arcs). Les états initiaux (resp. finaux) sont symbolisés par une flèche entrante (resp. sortante).

On dit que l'automate est *déterministe* si $\#I = \#F = 1$ (on a un seul point d'entrée, un seul point de sortie), et il n'y a pas d'arcs doubles (deux flèches ayant mêmes point de départ et d'arrivée).

Soit (p, a, q) une transition : p est l'*origine*, a l'*étiquette*, q l'*extrémité* ; on note $p \xrightarrow{a} q$.

Un *chemin* est une suite de transitions successives :

$$c = (q_0, a_1, q_1)(q_1, a_2, q_2) \cdots (q_{n-1}, a_n, q_n).$$

On le note

$$c : q_0 \xrightarrow{a_1} q_1 \to \cdots \to q_{n-1} \xrightarrow{a_n} q_n.$$

Un chemin est *réussi* (ou *acceptant*) si $q_0 \in I$, $q_n \in F$.

Exemple, pour l'automate de la figure 14.4, on a $\sigma = \{10, 20\}$, $Q = \{i, 1, 2, 3\}$, $F = \{i\}$.

14.6.3 Représentations d'automate déterministe

Si l'automate est simple, on peut le programmer sous forme de switch :

```
  char c = in.read(); int q = q0;
  switch(q){
  case 0:
    if (c == 'a') ... else if (c == 'b')  ...;
    break;
  case 1:
    if (c == 'b') ... else if (c == 'c') ... ;
    break;
  ...
  case 10:
    if ...  break;
}
```

Dans un cas plus général, tout codage de graphe convient. Un avantage est que l'automate devient une donnée du programme, plutôt que d'être "codé en dur" par des instructions comme switch.

Quatrième partie

Problématiques classiques en informatique

Chapitre 15

Recherche exhaustive

Ce que l'ordinateur sait faire de mieux, c'est traiter très rapidement une quantité gigantesque de données. Cela dit, il y a des limites à tout, et le but de ce chapitre est d'expliquer sur quelques cas ce qu'il est raisonnable d'attendre comme temps de résolution d'un problème. Cela nous permettra d'insister sur le coût des algorithmes et sur la façon de les modéliser.

15.1 Rechercher dans du texte

Commençons par un problème pour lequel de bonnes solutions existent. Rechercher une phrase dans un texte est une tâche que l'on demande à n'importe quel programme de traitement de texte, à un navigateur, un moteur de recherche, etc. C'est également une part importante du travail accompli régulièrement en bio-informatique.

Vues les quantités de données gigantesques que l'on doit parcourir, il est crucial de faire cela le plus rapidement possible. Dans certains cas, on n'a même pas le droit de lire plusieurs fois les données. Le but de cette section est de présenter quelques algorithmes qui accomplissent ce travail.

Pour modéliser le problème, nous supposons que nous travaillons sur un texte T (un tableau de caractères `char[]`, plutôt qu'un objet de type `String` pour alléger un peu les programmes) de longueur n dans lequel nous recherchons un motif M (un autre tableau de caractères) de longueur m que nous supposerons plus petit que n. Nous appelerons *occurence en position* $i \geqslant 0$ la propriété que `T[i]=M[0]`, ..., `T[i+m-1]=M[m-1]`.

Recherche naïve

C'est l'idée la plus naturelle : on essaie toutes les positions possibles du motif en dessous du texte. Comment tester qu'il existe une occurrence en position i ? Il suffit d'utiliser un indice j qui va servir à comparer `M[j]` à `T[i+j]` de proche en proche :

```
public static boolean occurrence(char[] T, char[] M, int i){
    for(int j = 0; j < M.length; j++)
        if(T[i+j] != M[j]) return false;
    return true;
}
```

Nous utilisons cette primitive dans la fonction suivante, qui teste toutes les occurrences possibles :

```
public static void naif(char[] T, char[] M){
    System.out.print("Occurrences en position :");
    for(int i = 0; i < T.length-M.length; i++)
        if(occurrence(T, M, i))
            System.out.print(" "+i+",");
    System.out.println("");
}
```

Si `T` contient les caractères de la chaîne `"il fait beau aujourd'hui"` et `M` ceux de `"au"`, le programme affichera

```
Occurrences en position: 10, 13,
```

Le nombre de comparaisons de caractères effectuées est au plus $(n-m)m$, puisque chacun des $n-m$ tests en demande m. Si m est négligeable devant n, on obtient un nombre de l'ordre de nm. Le but de la section qui suit est de donner un algorithme faisant moins de comparaisons. Notons que l'on peut adapter cet algorithme au cas où `M` est lu à la volée (et donc pas stocké en mémoire).

Algorithme linéaire de Karp-Rabin

Supposons que S soit une fonction (non nécessairement injective) qui donne une valeur numérique à une chaîne de caractères quelconque, que nous appelerons *signature* : nous en donnons deux exemples ci-dessous. Si deux chaînes de caractères C_1 et C_2 sont identiques, alors $S(C_1) = S(C_2)$. Réciproquement, si $S(C_1) \neq S(C_2)$, alors C_1 ne peut être égal à C_2. Insistons lourdement sur le fait que $S(C_1) = S(C_2)$ n'implique pas $C_1 = C_2$.

Le principe de l'algorithme de Karp-Rabin utilise cette idée de la façon suivante : on remplace le test d'occurrence $T[i..i+m-1] = M[0..m-1]$ par $S(T[i..i+m-1]) = S(M[0..m-1])$. Le membre de droite de ce test est constant, on le précalcule donc et il ne reste plus qu'à effectuer $n-m$ calculs de S et comparer la valeur $S(T[i..i+m-1])$ à cette constante. En cas d'égalité, on soupçonne une occurrence et on la vérifie à l'aide de la fonction `occurrence` présentée ci-dessus. Le nombre de calculs à effectuer est simplement $1+n-m$ évaluations de S.

Voici la fonction qui implante cette idée. Nous préciserons la fonction de signature S plus loin (codée ici sous la forme d'une fonction `signature`) :

```
public static void KR(char[] T, char[] M){
    int n, m;
    long hT, hM;

    n = T.length;
    m = M.length;
```

```
    System.out.print("Occurrences en position :");
    hM = signature(M, m, 0);
    for(int i = 0; i < n-m; i++){
        hT = signature(T, m, i);
        if(hT == hM){
            if(occurrence(T, M, i))
                System.out.print(" "+i+",");
            else
                System.out.print(" ["+i+"],");
        }
    }
    System.out.println("");
}
```

La fonction de signature est critique. Il est difficile de fabriquer une fonction qui soit à la fois injective et rapide à calculer. On se contente d'approximations. Soit `X` un texte de longueur `m`. En JAVA ou d'autres langages proches, il est généralement facile de convertir un caractère en nombre. Le codage unicode représente un caractère sur 16 bits et le passage du caractère `c` à l'entier est simplement `(int)c`. La première fonction à laquelle on peut penser est celle qui se contente de faire la somme des caractères représentés par des entiers :

```
    public static long signature(char[] X, int m, int i){
        long s = 0;

        for(int j = i; j < i+m; j++)
            s += (long)X[j];
        return s;
    }
```

Avec cette fonction, le programme affichera :

```
Occurrences en position: 10, 13, [18],
```

où on a indiqué les fausses occurrences par des crochets. On verra plus loin comment diminuer ce nombre.

Pour accélérer le calcul de la signature, on remarque que l'on peut faire cela de manière incrémentale. Plus précisément :

$$S(X[1..m]) = S(X[0..m-1]) - X[0] + X[m],$$

ce qui remplace m additions par 1 addition et 1 soustraction à chaque étape (on a confondu $X[i]$ et sa valeur en tant que caractère).

Une fonction de signature qui présente moins de collisions s'obtient à partir de ce qu'on appelle une fonction de hachage, dont la théorie sera présentée à la section 11.3. On prend p un nombre premier et B un entier. La signature est alors :

$$S(X[0..m-1]) = (X[0]B^{m-1} + \cdots + X[m-1]B^0) \bmod p.$$

On montre que la probabilité de collisions est alors $1/p$. Typiquement, $B = 2^{16}$, $p = 2^{31} - 1 = 2147483647$.

L'intérêt de cette fonction est qu'elle permet un calcul incrémental, puisque :

$$S(X[i+1..i+m]) = BS(X[i..i+m-1]) - X[i]B^m + X[i+m],$$

qui s'évalue d'autant plus rapidement que l'on a précalculé $B^m \bmod p$.
Le nombre de calculs effectués est $O(n+m)$, ce qui représente une amélioration notable par rapport à la recherche naïve.

Les fonctions correspondantes sont :

```
public static long B = ((long)1) << 16, p = 2147483647;

// calcul de S(X[i..i+m-1])
public static long signature2(char[] X, int i, int m){
    long s = 0;

    for(int j = i; j < i+m; j++)
        s = (s * B + (int)X[j]) % p;
    return s;
}

// S(X[i+1..i+m]) = B S(X[i..i+m-1])-X[i] B^m + X[i+m]
public static long signatureIncr(char[] X, int m, int i,
                                 long s, long Bm){
    long ss;

    ss = ((int)X[i+m]) - ((((int)X[i]) * Bm)) % p;
    if(ss < 0) ss += p;
    ss = (ss + B * s) % p;
    return ss;
}

public static void KR2(char[] T, char[] M){
    int n, m;
    long Bm, hT, hM;

    n = T.length;
    m = M.length;
    System.out.print("Occurrences en position :");
    hM = signature2(M, 0, m);
    // calcul de Bm = B^m mod p
    Bm = B;
    for(int i = 2; i <= m; i++)
        Bm = (Bm * B) % p;
    hT = signature2(T, 0, m);
```

```
        for(int i = 0; i < n-m; i++){
            if(i > 0)
                hT = signatureIncr(T, m, i-1, hT, Bm);
            if(hT == hM){
                if(occurrence(T, M, i))
                    System.out.print(" "+i+",");
                else
                    System.out.print(" ["+i+"],");
            }
        }
        System.out.println("");
    }
```

Cette fois, le programme ne produit plus de collisions :

```
Occurrences en position : 10, 13,
```

Remarques complémentaires

Des algorithmes plus rapides existent, comme par exemple ceux de Knuth-Morris-Pratt ou Boyer-Moore. Il est possible également de chercher des chaînes proches du motif donné, par exemple en cherchant à minimiser le nombre de lettres différentes entre les deux chaînes.

La recherche de chaînes est tellement importante qu'Unix possède une commande `grep` qui permet de rechercher un motif dans un fichier. À titre d'exemple :

```
unix% grep int Essai.java
```

affiche les lignes du fichier `Essai.java` qui contiennent le motif `int`. Pour afficher les lignes ne contenant pas `int`, on utilise :

```
unix% grep -v int Essai.java
```

On peut faire des recherches plus compliquées, comme par exemple rechercher les lignes contenant un 0 ou un 1 :

```
unix% grep [01] Essai.java
```

Le dernier exemple est :

```
unix% grep "int .*[0-9]" Essai.java
```

qui est un cas d'expression régulière. Elles peuvent être décrites en termes d'automates, qui sont étudiés en cours INF421 et INF431. Pour plus d'informations sur la commande `grep`, tapez `man grep`.

15.2 Le problème du sac-à-dos

Considérons le problème suivant, appelé *problème du sac-à-dos* : on cherche à remplir un sac-à-dos avec un certain nombre d'objets de façon à le remplir exactement. Comment fait-on ?

On peut modéliser ce problème de la façon suivante : on se donne n entiers strictement positifs a_i et un entier S. Existe-t-il des nombres $x_i \in \{0, 1\}$ tels que

$$S = x_0a_0 + x_1a_1 + \cdots + x_{n-1}a_{n-1} \,?$$

Si x_i vaut 1, c'est que l'on doit prendre l'objet a_i, et on ne le prend pas si $x_i = 0$.

Un algorithme de recherche des solutions doit être capable d'énumérer rapidement tous les n uplets de valeurs des x_i. Nous allons donner quelques algorithmes qui pourront être facilement modifiés pour chercher des solutions à d'autres problèmes numériques : équations du type $f(x_0, x_1, \ldots, x_{n-1}) = 0$ avec f quelconque, ou encore $\max f(x_0, x_1, \ldots, x_{n-1})$ sur un nombre fini de x_i.

15.2.1 Premières solutions

Si n est petit et fixé, on peut s'en tirer en utilisant des boucles `for` imbriquées qui permettent d'énumérer les valeurs de x_0, x_1, x_2. Voici ce qu'on peut écrire :

```
// Solution brutale
public static void sacADos3(int[] a, int S){
    int N;

    for(int x0 = 0; x0 < 2; x0++)
        for(int x1 = 0; x1 < 2; x1++)
            for(int x2 = 0; x2 < 2; x2++){
                N = x0 * a[0] + x1 * a[1] + x2 * a[2];
                if(N == S)
                    System.out.println(""+x0+x1+x2);
            }
}
```

Cette version est gourmande en calculs, puisque N est calculé dans la dernière boucle, alors que la quantité $x_0a_0 + x_1a_1$ ne dépend pas de x_2. On écrit plutôt :

```
public static void sacADos3b(int[] a, int S){
    int N0, N1, N2;

    for(int x0 = 0; x0 < 2; x0++){
        N0 = x0 * a[0];
        for(int x1 = 0; x1 < 2; x1++){
            N1 = N0 + x1 * a[1];
            for(int x2 = 0; x2 < 2; x2++){
                N2 = N1 + x2 * a[2];
                if(N2 == S)
```

```
                        System.out.println(""+x0+x1+x2);
                }
            }
        }
    }
```

On peut encore aller plus loin, en ne faisant aucune multiplication, et remarquant que deux valeurs de N_i diffèrent de a_i. Cela donne :

```
public static void sacADos3c(int[] a, int S){
    for(int x0 = 0, N0 = 0; x0 < 2; x0++, N0 += a[0])
       for(int x1 = 0, N1 = N0; x1 < 2; x1++, N1 += a[1])
          for(int x2 = 0, N2 = N1; x2 < 2; x2++, N2 += a[2])
             if(N2 == S)
                System.out.println(""+x0+x1+x2);
  }
```

Arrivé ici, on ne peut guère faire mieux. Le problème majeur qui reste est que le programme n'est en aucun cas évolutif. Il ne traite que le cas de $n = 3$. On peut bien sûr le modifier pour traiter des cas particuliers fixes, mais on doit connaître n à l'avance, au moment de la compilation du programme.

15.2.2 Deuxième approche

Les x_i doivent prendre toutes les valeurs de l'ensemble $\{0, 1\}$, soit 2^n. Toute solution peut s'écrire comme une suite de bits $x_0x_1 \dots x_{n-1}$ et donc s'interpréter comme un entier unique de l'intervalle $I_n = [0, 2^n[$, à savoir

$$x_0 2^0 + x_1 2^1 + \cdots + x_{n-1} 2^{n-1}.$$

Parcourir l'ensemble des x_i possibles ou bien cet intervalle est donc la même chose.

On connait un moyen simple de passer en revue tous les éléments de I_n, c'est l'addition. Il nous suffit ainsi de programmer l'addition binaire sur un entier représenté comme un tableau de bits pour faire l'énumération. On additionne 1 à un registre, en propageant à la main la retenue. Pour simplifier la lecture des fonctions qui suivent, on a introduit une fonction qui affiche les solutions :

```
  // affichage de i sous forme de sommes de bits
  public static afficher(int i, int[] x){
      System.out.print("i="+i+"=");
      for(int j = 0; j < n; j++)
          System.out.print(""+x[j]);
      System.out.println("");
  }

  public static void parcourta(int n){
```

```
        int retenue;
        int[] x = new int[n];

        for(int i = 0; i < (1 << n); i++){
            afficher(i, x);
            // simulation de l'addition
            retenue = 1;
            for(int j = 0; j < n; j++){
                x[j] += retenue;
                if(x[j] == 2){
                    x[j] = 0;
                    retenue = 1;
                }
                else break; // on a fini
            }
        }
    }
```

(L'instruction `1<<n` calcule 2^n.) On peut faire un tout petit peu plus concis en gérant virtuellement la retenue : si on doit ajouter 1 à $x_j = 0$, la nouvelle valeur de x_j est 1, il n'y a pas de retenue à propager, on s'arrête et on sort de la boucle ; si on doit ajouter 1 à $x_j = 1$, sa valeur doit passer à 0 et engendrer une nouvelle retenue de 1 qu'elle doit passer à sa voisine. On écrit ainsi :

```
    public static void parcourtb(int n){
        int[] x = new int[n];

        for(int i = 0; i < (1 << n); i++){
            afficher(i, x);
            // simulation de l'addition
            for(int j = 0; j < n; j++){
                if(x[j] == 1)
                    x[j] = 0;
                else{
                    x[j] = 1;
                    break; // on a fini
                }
            }
        }
    }
```

La boucle centrale étant écrite, on peut revenir à notre problème initial, et au programme de la figure 15.1.

Combien d'additions fait-on dans cette fonction ? Pour chaque valeur de i, on fait

```
// a[0..n[ : existe-t-il x[] tel que
// somme(a[i]*x[i], i=0..n-1) = S ?
public static void sacADosn(int[] a, int S){
    int n = a.length, N;
    int[] x = new int[n];

    for(int i = 0; i < (1 << n); i++){
        // reconstruction de N = somme x[i]*a[i]
        N = 0;
        for(int j = 0; j < n; j++)
            if(x[j] == 1)
                N += a[j];
        if(N == S){
            System.out.print("S="+S+"=");
            for(int j = 0; j < n; j++)
                if(x[j] == 1)
                    System.out.print("+"+a[j]);
            System.out.println("");
        }
        // simulation de l'addition
        for(int j = 0; j < n; j++){
            if(x[j] == 1)
                x[j] = 0;
            else{
                x[j] = 1;
                break; // on a fini
            }
        }
    }
}
```

FIG. 15.1 – Version finale.

au plus n additions d'entiers (en moyenne, on en fait d'ailleurs $n/2$). Le corps de la boucle est effectué 2^n fois, le nombre d'additions est $O(n2^n)$.

15.2.3 Code de Gray*

Théorie et implantations

Le code de Gray permet d'énumérer tous les entiers de $I_n = [0, 2^n - 1]$ de telle sorte qu'on passe d'un entier à l'autre en changeant la valeur d'un seul bit. Si k est un entier de cet intervalle, on l'écrit $k = k_0 + k_1 2 + \cdots + k_{n-1}2^{n-1}$ et on le note $[k_{n-1}, k_{n-2}, \ldots, k_0]$.

On va fabriquer une suite $G_n = (g_{n,i})_{0 \leqslant i < 2^n}$ dont l'ensemble des valeurs est $[0, 2^n - 1]$, mais dans un ordre tel qu'on passe de $g_{n,i}$ à $g_{n,i+1}$ en changeant *un seul chiffre* de l'écriture de $g_{n,i}$ en base 2.

Commençons par rappeler les valeurs de la fonction ou exclusif (appelé XOR en anglais) et noté $\oplus$. La table de vérité de cette fonction logique est

	0	1
0	0	1
1	1	0

En JAVA, la fonction $\oplus$ s'obtient par `^` et opère sur des mots : si `m` est de type `int`, `m` représente un entier signé de 32 bits et `m ^ n` effectue l'opération sur tous les bits de `m` et `n` à la fois. Autrement dit, si les écritures de m et n en binaire sont :

$$m = m_{31}2^{31} + \cdots + m_0 = [m_{31}, m_{30}, \ldots, m_0],$$

$$n = n_{31}2^{31} + \cdots + n_0 = [n_{31}, n_{30}, \ldots, n_0]$$

avec m_i, n_i dans $\{0, 1\}$, on a

$$m \oplus n = \sum_{i=0}^{31} (m_i \oplus n_i) 2^i.$$

On définit maintenant $g_n : [0, 2^n - 1] \to [0, 2^n - 1]$ par $g_n(0) = 0$ et si $i > 0$, $g_n(i) = g_n(i-1) \oplus 2^{b(i)}$ où $b(i)$ est le plus grand entier j tel que $2^j \mid i$. Cet entier existe et $b(i) < n$ pour tout $i < 2^n$. Donnons les valeurs des premières fonctions :

$$g_1(0) = [0], g_1(1) = [1],$$

$$g_2(0) = [00], g_2(1) = [01], g_2(2) = g_2([10]) = g_2(1) \oplus 2^1 = [01] \oplus [10] = [11],$$

$$g_2(3) = g_2([11]) = g_2(2) \oplus 2^0 = [11] \oplus [01] = [10].$$

Écrivons les valeurs de g_3 sous forme d'un tableau qui souligne la symétrie de celles-ci :

i	$g(i)$	i	$g(i)$
0	$000 = 0$	7	$100 = 4$
1	$001 = 1$	6	$101 = 5$
2	$011 = 3$	5	$111 = 7$
3	$010 = 2$	4	$110 = 6$

Cela nous conduit naturellement à prouver que la fonction g_n possède un comportement "miroir" :

Proposition 6 *Si* $2^{n-1} \leqslant i < 2^n$, *alors* $g_n(i) = 2^{n-1} + g_n(2^n - 1 - i)$.

Démonstration : Notons que $2^n - 1 - 2^n < 2^n - 1 - i \leqslant 2^n - 1 - 2^{n-1}$, soit $0 \leqslant 2^n - 1 - i \leqslant 2^{n-1} - 1 < 2^{n-1}$.

On a

$$g_n(2^{n-1}) = g_n(2^{n-1} - 1) \oplus 2^{n-1} = 2^{n-1} + g_n(2^{n-1} - 1) = 2^{n-1} + g_n(2^n - 1 - 2^{n-1}).$$

Supposons la propriété vraie pour $i = 2^{n-1} + r > 2^{n-1}$. On écrit :

$$\begin{aligned} g_n(i+1) &= g_n(i) \oplus 2^{b(r+1)} \\ &= (2^{n-1} + g_n(2^n - 1 - i)) \oplus 2^{b(r+1)} \\ &= 2^{n-1} + (g_n(2^n - 1 - i) \oplus 2^{b(r+1)}). \end{aligned}$$

On conclut en remarquant que :

$$g_n(2^n - 1 - i) = g_n(2^n - 1 - i - 1) \oplus 2^{b(2^n - 1 - i)}$$

et $b(2^n - i - 1) = b(i+1) = b(r+1)$. □

On en déduit par exemple que $g_n(2^n - 1) = 2^{n-1} + g_n(0) = 2^{n-1}$.

Proposition 7 *Si* $n \geqslant 1$, *la fonction* g_n *définit une bijection de* $[0, 2^n - 1]$ *dans lui-même.*

Démonstration : nous allons raisonner par récurrence sur n. Nous venons de voir que g_1 et g_2 satisfont la propriété. Supposons-la donc vraie au rang $n \geqslant 2$ et regardons ce qu'il se passe au rang $n+1$. Commençons par remarquer que si $i < 2^n$, g_{n+1} coïncide avec g_n car $b(i) < n$.

Si $i = 2^n$, on a $g_{n+1}(i) = g_n(2^n - 1) \oplus 2^n$, ce qui a pour effet de mettre un 1 en bit $n+1$. Si $2^n < i < 2^{n+1}$, on a toujours $b(i) < n$ et donc $g_{n+1}(i)$ conserve le $n+1$-ième bit à 1. En utilisant la propriété de miroir du lemme précédent, on voit que g_{n+1} est également une bijection de $[2^n, 2^{n+1} - 1]$ dans lui-même. □

Quel est l'intérêt de la fonction g_n pour notre problème ? Des propriétés précédentes, on déduit que g_n permet de parcourir l'intervalle $[0, 2^n - 1]$ en passant d'une valeur d'un entier à l'autre en changeant seulement un bit dans son écriture en base 2. On trouvera à la figure 15.2 une première fonction JAVA qui réalise le parcours.

On peut faire un peu mieux, en remplaçant les opérations de modulo par des opérations logiques, voir la figure 15.3.

Revenons au sac-à-dos. On commence par calculer la valeur de $\sum x_i a_i$ pour le n-uplet $[0, 0, \ldots, 0]$. Puis on parcourt l'ensemble des x_i à l'aide du code de Gray. Si à l'étape i, on a calculé

$$N_i = x_{n-1}a_{n-1} + \cdots + x_0 a_0,$$

avec $g(i) = [x_{n-1}, \ldots, x_0]$, on passe à l'étape $i+1$ en changeant un bit, mettons le j-ème, ce qui fait que :

$$N_{i+1} = N_i + a_j$$

si $g_{i+1} = g_i + 2^j$, et

$$N_{i+1} = N_i - a_j$$

si $g_{i+1} = g_i - 2^j$. On différencie les deux valeurs en testant la présence du j-ème bit après l'opération sur `gi` :

```
public static void gray(int n){
  int gi = 0;

  affichergi(0, n);
  for(int i = 1; i < (1 << n); i++){
    // on écrit i = 2^j*k, 0 <= j < n, k impair
    int k = i, j;

    for(j = 0; j < n; j++){
      if((k % 2) == 1)
        // k est impair, on s'arrête
        break;
      k /= 2;
    }
    gi ^= (1 << j);
    affichergi(gi, n);
  }
}

public static void afficherAux(int gi, int j, int n){
  if(j >= 0){
    afficherAux(gi >> 1, j-1, n);
    System.out.print((gi & 1));
  }
}

public static void affichergi(int gi, int n){
  afficherAux(gi, n-1, n);
  System.out.println("="+gi);
}
```

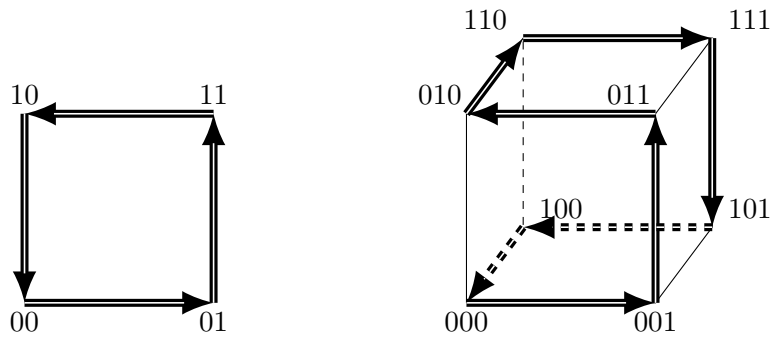
FIG. 15.2 – Affichage du code de Gray.

```
public static void gray2(int n){
  int gi = 0;

  affichergi(0, n);
  for(int i = 1; i < (1 << n); i++){
    // on écrit i = 2^j*k, 0 <= j < n, k impair
    int k = i, j;

    for(j = 0; j < n; j++){
      if((k & 1) == 1)
        // k est impair, on s'arrête
        break;
      k >>= 1;
    }
    gi ^= (1 << j);
    affichergi(gi, n);
  }
}
```

FIG. 15.3 – Affichage du code de Gray (2è version).

Remarques

Le code de Gray permet de visiter chacun des sommets d'un hypercube. L'hypercube en dimension n est formé précisément des sommets $(x_0, x_1, \ldots, x_{n-1})$ parcourant tous les n-uplets d'éléments formés de $\{0, 1\}$. Le code de Gray permet de visiter tous les sommets du cube une fois et une seule, en commençant par le point $(0, 0, \ldots, 0)$, et en s'arrêtant juste au-dessus en $(1, 0, \ldots, 0)$. Remarquons que ce parcours ne visite pas nécessairement toutes les arêtes de l'hypercube. C'est pour cela que nous avons dessiné des flèches épaisses pour bien indiquer quelles arêtes sont parcourues.



```
public static void sacADosGray(int[] a, int S){
  int n = a.length, gi = 0, N = 0, deuxj;

  if(N == S)
    afficherSolution(a, S, 0);
  for(int i = 1; i < (1 << n); i++){
    // on écrit i = 2^j*k, 0 <= j < n, k impair
    int k = i, j;

    for(j = 0; j < n; j++){
      if((k & 1) == 1)
        // k est impair, on s'arrête
        break;
      k >>= 1;
    }
    deuxj = 1 << j;
    gi ^= deuxj;
    if((gi & deuxj) != 0)
      N += a[j];
    else
      N -= a[j];
    if(N == S)
      afficherSolution(a, S, gi);
  }
}

public static void afficherSolution(int[] a, int S, int gi){
  System.out.print("S="+S+"=");
  for(int i = 0; i < a.length; i++){
    if((gi & 1) == 1)
      System.out.print("+"+a[i]);
    gi >>= 1;
  }
  System.out.println();
}
```

FIG. 15.4 – Code de Gray pour le sac-à-dos.

15.2.4 Retour arrière (backtrack)

L'idée est de résoudre le problème de proche en proche. Supposons avoir déjà calculé $S_i = x_0a_0 + x_1a_1 + \ldots + x_{i-1}a_{i-1}$. Si $S_i = S$, on a trouvé une solution et on ne continue pas à rajouter des $a_j > 0$. Sinon, on essaie de rajouter $x_i = 0$ et on teste au cran suivant, puis on essaie avec $x_i = 1$. On fait ainsi des calculs et si cela échoue, on retourne en arrière pour tester une autre solution, d'où le nom *backtrack*.

L'implantation de cette idée est donnée ci-dessous :

```
// on a déjà calculé Si = sum(a[j]*x[j], j=0..i-1)
public static void sacADosrec(int[] a, int S, int[] x,
                              int Si, int i){
    nbrec++;
    if(Si == S)
        afficherSolution(a, S, x, i);
    else if(i < a.length){
        x[i] = 0;
        sacADosrec(a, S, x, Si, i+1);
        x[i] = 1;
        sacADosrec(a, S, x, Si+a[i], i+1);
    }
}
```

On appelle cette fonction avec :

```
public static void sacADos(int[] a, int S){
    int[] x = new int[a.length];

    nbrec = 0;
    sacADosrec(a, S, x, 0, 0);
    System.out.print("# appels=" + nbrec);
    System.out.println(" // " + (1 << (a.length + 1)));
}
```

et le programme principal est :

```
public static void main(String[] args){
    int[] a = {1, 4, 7, 12, 18, 20, 30};

    sacADos(a, 11);
    sacADos(a, 12);
    sacADos(a, 55);
    sacADos(a, 14);
}
```

On a ajouté une variable `nbrec` qui mémorise le nombre d'appels effectués à la fonction `sacADosrec` et qu'on affiche en fin de calcul. L'exécution donne :

```
S=11=+4+7
# appels=225 // 256
S=12=+12
S=12=+1+4+7
# appels=211 // 256
S=55=+7+18+30
S=55=+1+4+20+30
S=55=+1+4+12+18+20
# appels=253 // 256
# appels=255 // 256
```

On voit que dans le cas le pire, on fait bien 2^{n+1} appels à la fonction (mais seulement 2^n additions).

On remarque que si les a_j sont tous strictement positifs, et si $S_i > S$ à l'étape i, alors il n'est pas nécessaire de poursuivre. En effet, on ne risque pas d'atteindre S en ajoutant encore des valeurs strictement positives. Il suffit donc de rajouter un test qui permet d'éliminer des appels récursifs inutiles :

```
    // on a déjà calculé Si = sum(a[j]*x[j], j=0..i-1)
    public static void sacADosrec(int[] a, int S, int[] x,
                                  int Si, int i){
        nbrec++;
        if(Si == S)
            afficherSolution(a, S, x, i);
        else if((i < a.length) && (Si < S)){
            x[i] = 0;
            sacADosrec(a, S, x, Si, i+1);
            x[i] = 1;
            sacADosrec(a, S, x, Si+a[i], i+1);
        }
    }
```

On constate bien sur les exemples une diminution notable des appels, dans les cas où S est petit par rapport à $\sum_i a_i$:

```
S=11=+4+7
# appels=63 // 256
S=12=+12
S=12=+1+4+7
# appels=71 // 256
S=55=+7+18+30
S=55=+1+4+20+30
S=55=+1+4+12+18+20
# appels=245 // 256
# appels=91 // 256
```

Terminons cette section en remarquant que le problème du sac-à-dos est le prototype des *problèmes difficiles* au sens de la théorie de la complexité, et que c'est là l'un des sujets traités en Année 3.

15.3 Permutations

Une *permutation* des n éléments 1, 2, ..., n est un n-uplet $(a_1, a_2, \ldots, a_n)$ tel que l'ensemble des valeurs des a_i soit exactement $\{1, 2, \ldots, n\}$. Par exemple, $(1, 3, 2)$ est une permutation sur 3 éléments, mais pas $(2, 2, 3)$. Il y a $n! = n \times (n-1) \times 2 \times 1$ permutations de n éléments.

15.3.1 Fabrication des permutations

Nous allons fabriquer toutes les permutations sur n éléments et les stocker dans un tableau. On procède récursivement, en fabriquant les permutations d'ordre $n-1$ et en rajoutant n à toutes les positions possibles :

```
public static int[][] permutations(int n){
    if(n == 1){
        int[][] t = {{0, 1}};

        return t;
    }
    else{
        // tnm1 va contenir les (n-1)!
        // permutations à n-1 éléments
        int[][] tnm1 = permutations(n-1);
        int factnm1 = tnm1.length;
        int factn = factnm1 * n; // vaut n!
        int[][] t = new int[factn][n+1];

        // recopie de tnm1 dans t
        for(int i = 0; i < factnm1; i++)
            for(int j = 1; j <= n; j++){
                // on recopie tnm1[][1..j[
                for(int k = 1; k < j; k++)
                    t[n*i+(j-1)][k] = tnm1[i][k];
                // on place n à la position j
                t[n*i+(j-1)][j] = n;
                // on recopie tnm1[][j..n-1]
                for(int k = j; k <= n-1; k++)
                    t[n*i+(j-1)][k+1] = tnm1[i][k];
            }
        return t;
    }
```

15.3.2 Énumération des permutations

Le problème de l'approche précédente est que l'on doit stocker les $n!$ permutations, ce qui peut finir par être un peu gros en mémoire. Dans certains cas, on peut vouloir se contenter d'énumérer sans stocker.

On va là aussi procéder par récurrence : on suppose avoir construit une permutation `t[1..i0-1]` et on va mettre dans `t[i0]` les $n-i_0+1$ valeurs non utilisées auparavant, à tour de rôle. Pour ce faire, on va gérer un tableau auxiliaire de booléens `utilise`, tel que `utilise[j]` est vrai si le nombre j n'a pas déjà été choisi. Le programme est alors :

```
// approche en O(n!)
public static void permrec2(int[] t, int n,
                      boolean[] utilise, int i0){
    if(i0 > n)
        afficher(t, n);
    else{
        for(int v = 1; v <= n; v++){
            if(! utilise[v]){
                utilise[v] = true;
                t[i0] = v;
                permrec2(t, n, utilise, i0+1);
                utilise[v] = false;
            }
        }
    }
}

public static void permrec2(int n){
    int[] t = new int[n+1];
    boolean[] utilise = new boolean[n+1];

    permrec2(t, n, utilise, 1);
}
```

Pour $n = 3$, on fabrique les permutations dans l'ordre :

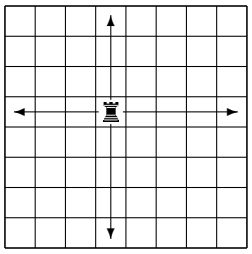
```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

15.4 Les n reines

Nous allons encore voir un algorithme de backtrack pour résoudre un problème combinatoire. Dans la suite, nous supposons que nous utilisons un échiquier $n \times n$.

15.4.1 Prélude : les n tours

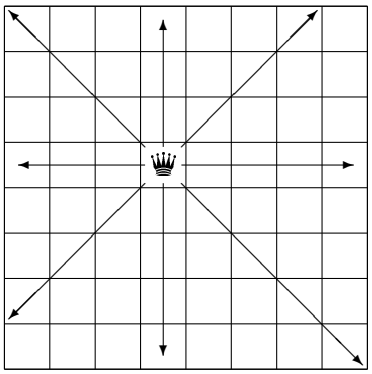
Rappelons quelques notions du jeu d'échecs. Une tour menace toute pièce adverse se trouvant dans la même ligne ou dans la même colonne.



On voit facilement qu'on peut mettre n tours sur l'échiquier sans que les tours ne s'attaquent. En fait, une solution correspond à une permutation de $1..n$, et on sait déjà faire. Le nombre de façons de placer n tours non attaquantes est donc $T(n) = n!$.

15.4.2 Des reines sur un échiquier

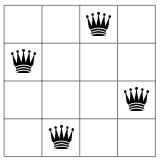
La reine se déplace dans toutes les directions et attaque toutes les pièces (adverses) se trouvant sur les même ligne ou colonne ou diagonales qu'elle.



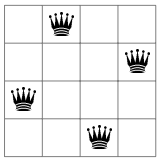
Une reine étant une tour avec un peu plus de pouvoir, il est clair que le nombre maximal de reines pouvant être sur l'échiquier sans s'attaquer est au plus n. On peut montrer que ce nombre est n pour $n = 1$ ou $n \geqslant 4$[1]. Reste à calculer le nombre de solutions possibles, et c'est une tâche difficile, et non résolue.

Donnons les solutions pour $n = 4$:

[1]Les petits cas peuvent se faire à la main, une preuve générale est plus délicate et elle est due à Ahrens, en 1921.

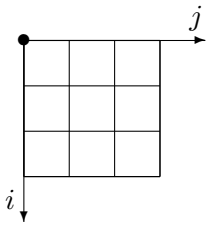


(2413)



(3142)

Expliquons comment résoudre le problème de façon algorithmique. On commence par chercher un codage d'une configuration. Une configuration admissible sera codée par la suite des positions d'une reine dans chaque colonne. On oriente l'échiquier comme suit :



Avec ces notations, on démontre :

Proposition 8 *La reine en position* (i_1, j_1) *attaque la reine en position* (i_2, j_2) *si et seulement si* $i_1 = i_2$ *ou* $j_1 = j_2$ *ou* $i_1 - j_1 = i_2 - j_2$ *ou* $i_1 + j_1 = i_2 + j_2$.

Démonstration : si elles sont sur la même diagonale nord-ouest/sud-est, $i_1 - j_1 = i_2 - j_2$; ou encore sur la même diagonale sud-ouest/nord-est, $i_1 + j_1 = i_2 + j_2$. □

On va procéder comme pour les permutations : on suppose avoir construit une solution approchée dans `t[1..i0[` et on cherche à placer une reine dans la colonne `i0`. Il faut s'assurer que la nouvelle reine n'attaque personne sur sa ligne (c'est le rôle du tableau `utilise` comme pour les permutations), et personne dans aucune de ses diagonales (fonction `pasDeConflit`). Le code est le suivant :

```
// t[1..i0[ est déjà rempli
public static void reinesAux(int[] t, int n,
                             boolean[] utilise, int i0){
    if(i0 > n)
        afficher(t);
    else{
        for(int v = 1; v <= n; v++)
            if(! utilise[v] && pasDeConflit(t, i0, v)){
                utilise[v] = true;
                t[i0] = v;
                reinesAux(t, n, utilise, i0+1);
                utilise[v] = false;
            }
    }
}
```

La programmation de la fonction `pasDeConflit` découle de la proposition 8 :

```
// t[1..i0[ est déjà rempli
public static boolean pasDeConflit(int[] t, int i0, int j){
    int x1, y1, x2 = i0, y2 = j;

    for(int i = 1; i < i0; i++){
        // on récupère les positions
        x1 = i;
        y1 = t[i];
        if((x1 == x2) // même colonne
           || (y1 == y2) // même ligne
           || ((x1-y1) == (x2-y2))
           || ((x1+y1) == (x2+y2)))
            return false;
    }
    return true;
}
```

Notons qu'il est facile de modifier le code pour qu'il calcule le nombre de solutions. Terminons par un tableau des valeurs connues de $R(n)$:

n	$R(n)$	n	$R(n)$	n	$R(n)$	n	$R(n)$
4	2	9	352	14	365596	19	4968057848
5	10	10	724	15	2279184	20	39029188884
6	4	11	2680	16	14772512	21	314666222712
7	40	12	14200	17	95815104	22	2691008701644
8	92	13	73712	18	666090624	23	24233937684440

Vardi a conjecturé que $\log R(n)/(n \log n) \to \alpha > 0$ et peut-être que $\alpha = 1$. Rivin & Zabih ont d'ailleurs mis au point un algorithme de meilleur complexité pour résoudre le problème, avec un temps de calcul de $O(n^2 8^n)$.

15.5 Les ordinateurs jouent aux échecs

Nous ne saurions terminer un chapitre sur la recherche exhaustive sans évoquer un cas très médiatique, celui des ordinateurs jouant aux échecs.

15.5.1 Principes des programmes de jeu

Deux approches ont été tentées pour battre les grands maîtres. La première, dans la lignée de Botvinik, cherche à programmer l'ordinateur pour lui faire utiliser la démarche humaine. La seconde, et la plus fructueuse, c'est utiliser l'ordinateur dans ce qu'il sait faire le mieux, c'est-à-dire examiner de nombreuses données en un temps court.

Comment fonctionne un programme de jeu ? En règle général, à partir d'une position donnée, on énumère les coups valides et on crée la liste des nouvelles positions. On tente alors de déterminer quelle est la meilleure nouvelle position possible. On fait cela sur

plusieurs tours, en parcourant un arbre de possibilités, et on cherche à garder le meilleur chemin obtenu.

Dans le meilleur des cas, l'ordinateur peut examiner tous les coups et il gagne à coup sûr. Dans le cas des échecs, le nombre de possibilités en début et milieu de partie est beaucoup trop grand. Aussi essaie-t-on de programmer la recherche la plus profonde possible.

15.5.2 Retour aux échecs

Codage d'une position

La première idée qui vient à l'esprit est d'utiliser une matrice 8×8 pour représenter un échiquier. On l'implante généralement sous la forme d'un entier de type `long` qui a 64 bits, un bit par case. On gère alors un ensemble de tels entiers, un par type de pièce par exemple.

On trouve dans la thèse de J. C. Weill un codage astucieux :

- les cases sont numérotées de 0 à 63 ;
- les pièces sont numérotées de 0 à 11 : pion blanc = 0, cavalier blanc = 1, ..., pion noir = 6, ..., roi noir = 11.

On stocke la position dans le vecteur de bits

$$(c_1, c_2, \ldots, c_{768})$$

tel que $c_{64i+j+1} = 1$ ssi la pièce i est sur la case j.

Les positions sont stockées dans une table de hachage la plus grande possible qui permet de reconnaître une position déjà vue.

Fonction d'évaluation

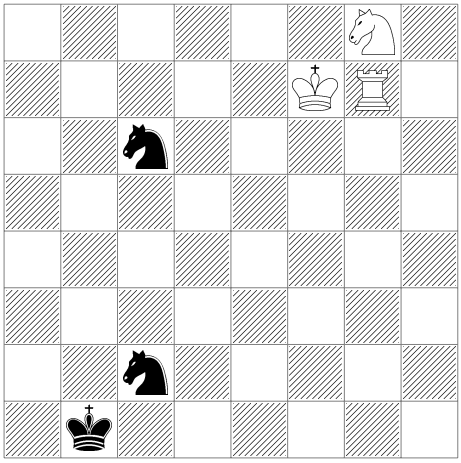
C'est un des secrets de tout bon programme d'échecs. L'idée de base est d'évaluer la force d'une position par une combinaison linéaire mettant en œuvre le poids d'une pièce (reine = 900, tour= 500, etc.). On complique alors généralement la fonction en fonction de stratégies (position forte du roi, etc.).

Bibliothèques de début et fin

Une façon d'accélérer la recherche est d'utiliser des bibliothèque d'ouvertures pour les débuts de partie, ainsi que des bibliothèques de fins de partie.

Dans ce dernier cas, on peut tenter, quand il ne reste que peu de pièces d'énumérer toutes les positions et de classifier les perdantes, les gagnantes et les nulles. L'algorithme est appelé analyse rétrograde et a été décrite par Ken Thompson (l'un des créateurs d'Unix).

À titre d'exemple, la figure ci-dessous décrit une position à partir de laquelle il faut **243 coups** (contre la meilleure défense) à Blanc (qui joue) pour capturer une pièce sans danger, avant de gagner (Stiller, 1998).



Deep blue contre Kasparov (1997)

Le projet a démarré en 1989 par une équipe de chercheurs et techniciens : C. J. Tan, Murray Campbell (fonction d'évaluation), Feng-hsiung Hsu, A. Joseph Hoane, Jr., Jerry Brody, Joel Benjamin. Une machine spéciale a été fabriquée : elle contenait 32 nœuds avec des RS/6000 SP (chip P2SC) ; chaque nœud contenait 8 processeurs spécialisés pour les échecs, avec un système AIX. Le programme était écrit en C pour le IBM SP Parallel System (MPI). La machine était capable d'engendrer $200,000,000$ positions par seconde (ou 60×10^9 en 3 minutes, le temps alloué). Deep blue a gagné 2 parties à 1 contre Kasparov[2].

Deep Fritz contre Kramnik (2002 ; 2006)

C'est cette fois un ordinateur plus raisonnable qui affronte un humain : 8 processeurs à 2.4 GHz et 256 Mo, qui peuvent calculer 3 millions de coups à la seconde. Le programmeur F. Morsch a soigné la partie algorithmique. Kramnik ne fait que match nul (deux victoires chacun, quatre nulles), sans doute épuisé par la tension du match.

Lors de la rencontre de 2006, **Deep Fritz** bat Kramnik 4 à 2, ce qui semble mettre un terme au débat pour le moment, les ordinateurs battant les grands maîtres.

Conclusion

Peut-on déduire de ce qui précède que les ordinateurs sont plus intelligents que les humains ? Certes non, ils *calculent* plus rapidement sur certaines données, c'est tout. Pour la petite histoire, les joueurs d'échec peuvent s'adapter à l'ordinateur qui joue face à lui et trouver des positions qui le mettent en difficulté. Une manière de faire est de jouer systématiquement de façon à maintenir un grand nombre de possibilités à chaque étape.

Nous renvoyons aux articles correspondant de wikipedia pour plus d'informations.

[2] `www.research.ibm.com/deepblue`

Chapitre 16

Polynômes et transformée de Fourier

Nous allons donner quelques idées sur la réalisation de bibliothèques de fonctions s'appliquant à un domaine commun, en l'illustrant sur un exemple, celui des calculs sur les polynômes à coefficients entiers. Une bonne référence pour les algorithmes décrits dans ce chapitre est [Knu81].

Comment écrit-on une bibliothèque ? On commence d'abord par choisir les objets de base, puis on leur adjoint quelques prédicats, des primitives courantes (fabrication, entrées sorties, test d'égalité, etc.). Puis dans un deuxième temps, on construit des fonctions un peu plus complexes, et on poursuit en assemblant des fonctions déjà construites.

16.1 La classe `Polynome`

Nous décidons de travailler sur des polynômes à coeffients entiers, que nous supposerons ici être de type `long`[1]. Un polynôme $P(X) = p_d X^d + \cdots + p_0$ a un *degré* d, qui est un entier positif ou nul si P n'est pas identiquement nul et -1 sinon (par convention).

16.1.1 Définition de la classe

Cela nous conduit à définir la classe, ainsi que le constructeur associé qui fabrique un polynôme dont tous les coefficients sont nuls :

```
public class Polynome{
    int deg;
    long[] coeff;

    public Polynome(int d){
        this.deg = d;
        this.coeff = new long[d+1];
```

[1] *stricto sensu*, nous travaillons en fait dans l'anneau des polynômes à coefficients définis modulo 2^{64}.

```
    }
}
```

Nous faisons ici la convention que les arguments d'appel d'une fonction correspondent à des polynômes dont le degré est exact, et que la fonction retourne un polynôme de degré exact. Autrement dit, si `P` est un paramètre d'appel d'une fonction, on suppose que `P.deg` contient le degré de P, c'est-à-dire que `P` est nul si `P.deg ==-1` et `P.coeff[P.deg]` n'est pas nul sinon.

16.1.2 Création, affichage

Quand on construit de nouveaux objets, il convient d'être capable de les créer et manipuler aisément. Nous avons déjà écrit un constructeur, mais nous pouvons avoir besoin par exemple de copier un polynôme :

```
public static Polynome copier(Polynome P){
    Polynome Q = new Polynome(P.deg);

    for(int i = 0; i <= P.deg; i++)
        Q.coeff[i] = P.coeff[i];
    return Q;
}
```

On écrit maintenant une fonction `toString()` qui permet d'afficher un polynôme à l'écran. On peut se contenter d'une fonction toute simple :

```
public String toString(){
    String s = "";

    for(int i = this.deg; i >= 0; i--){
        s = s.concat("+"+this.coeff[i]+"*X^"+i);
    }
    if(s == "") return "0";
    else return s;
}
```

Si on veut tenir compte des simplifications habituelles (pas d'affichage des coefficients nuls de P sauf si $P = 0$, $1X^1$ est généralement écrit X), il vaut mieux écrire la fonction de la figure 16.1.

16.1.3 Prédicats

Il est commode de définir des prédicats sur les objets. On programme ainsi un test d'égalité à zéro :

```
public static boolean estNul(Polynome P){
    return P.deg == -1;
}
```

```
public String toString(){
    String s = "";
    long coeff;
    boolean premier = true;

    for(int i = this.deg; i >= 0; i--){
        coeff = this.coeff[i];
        if(coeff != 0){
            // on n'affiche que les coefficients non nuls
            if(coeff < 0){
                s = s.concat("-");
                coeff = -coeff;
            }
            else
                // on n'affiche "+" que si ce n'est pas
                // premier coefficient affiché
                if(!premier) s = s.concat("+");
            // traitement du cas spécial "1"
            if(coeff == 1){
                if(i == 0)
                    s = s.concat("1");
            }
            else{
                s = s.concat(coeff+"");
                if(i > 0)
                    s = s.concat("*");
            }
            // traitement du cas spécial "X"
            if(i > 1)
                s = s.concat("X^"+i);
            else if(i == 1)
                s = s.concat("X");
            // à ce stade, un coefficient non nul
            // a été affiché
            premier = false;
        }
    }
    // le polynôme nul a le droit d'être affiché
    if(s == "") return "0";
    else return s;
}
```

FIG. 16.1 – Fonction d'affichage d'un polynôme.

De même, on ajoute un test d'égalité :

```
    public static boolean estEgal(Polynome P, Polynome Q){
        if(P.deg != Q.deg) return false;
        for(int i = 0; i <= P.deg; i++)
            if(P.coeff[i] != Q.coeff[i])
                return false;
        return true;
    }
```

16.1.4 Premiers tests

Il est important de tester le plus tôt possible la bibliothèque en cours de création, à partir d'exemples simples et maîtrisables. On commence par exemple par écrire un programme qui crée le polynôme $P(X) = 2X + 1$, l'affiche à l'écran et teste s'il est nul :

```
public class TestPolynome{
  public static void main(String[] args){
      Polynome P;

      // création de 2*X+1
      P = new Polynome(1);
      P.coeff[1] = 2;
      P.coeff[0] = 1;
      System.out.println("P="+P);
      System.out.println("P == 0 ? " + Polynome.estNul(P));
  }
}
```

L'exécution de ce programme donne alors :

```
P=2*X+1
P == 0? false
```

Nous allons avoir besoin d'entrer souvent des polynômes et il serait souhaitable d'avoir un moyen plus simple que de rentrer tous les coefficients les uns après les autres. On peut décider de créer un polynôme à partir d'une chaîne de caractères formattée avec soin. Un format commode pour définir un polynôme est une chaîne de caractères `s` de la forme `"deg s[deg] s[deg-1] ... s[0]"` qui correspondra au polynôme $P(X) = s_{deg}X^{deg} + \cdots + s_0$. Par exemple, la chaîne `"1 1 2"` codera le polynôme $X + 2$. La fonction convertissant une chaîne au bon format en polynôme est alors :

```
  public static Polynome deChaine(String s){
      Polynome P;
      long[] tabi = TC.longDeChaine(s);

      P = new Polynome((int)tabi[0]);
```

```
    for(int i = 1; i < tabi.length; i++)
        P.coeff[i-1] = tabi[i];
    return P;
}
```

(la fonction `TC.longDeChaine` appartient à la classe `TC` décrite en annexe) et elle est utilisée dans `TestPolynome` de la façon suivante :

```
P = Polynome.deChaine("1 1 2"); // c'est X+2
```

Une fois définis les objets de base, il faut maintenant passer aux opérations plus complexes.

16.2 Premières fonctions

16.2.1 Dérivation

La dérivée du polynôme 0 est 0, sinon la dérivée de $P(X) = \sum_{i=0}^{d} p_i X^i$ est :

$$P'(X) = \sum_{i=1}^{d} i p_i X^{i-1}.$$

On écrit alors la fonction :

```
public static Polynome deriver(Polynome P){
    Polynome dP;

    if(estNul(P)) return copier(P);
    dP = new Polynome(P.deg - 1);
    for(int i = P.deg; i >= 1; i--)
        dP.coeff[i-1] = i * P.coeff[i];
    return dP;
}
```

16.2.2 Évaluation ; schéma de Horner

Passons maintenant à l'évaluation du polynôme $P(X) = \sum_{i=0}^{d} p_i X^i$ en la valeur x. La première solution qui vient à l'esprit est d'appliquer la formule en calculant de proche en proche les puissances de x. Cela s'écrit :

```
// évaluation de P en x
public static long evaluer(Polynome P, long x){
    long Px, xpi;

    if(estNul(P)) return 0;
    // Px contiendra la valeur de P(x)
```

```
    Px = P.coeff[0];
    xpi = 1;
    for(int i = 1; i <= P.deg; i++){
        // calcul de xpi = x^i
        xpi *= x;
        Px += P.coeff[i] * xpi;
    }
    return Px;
}
```

Cette fonction fait $2d$ multiplications et d additions. On peut faire mieux en utilisant le schéma de Horner :

$$P(x) = (\cdots((p_d x + p_{d-1})x + p_{d-2})x + \cdots)x + p_0.$$

La fonction est alors :

```
public static long Horner(Polynome P, long x){
    long Px;

    if(estNul(P)) return 0;
    Px = P.coeff[P.deg];
    for(int i = P.deg-1; i >= 0; i--){
        // à cet endroit, Px contient:
        // p_d*x^(d-i-1) + ... + p_{i+1}
        Px *= x;
        // Px contient maintenant
        // p_d*x^(d-i) + ... + p_{i+1}*x
        Px += P.coeff[i];
    }
    return Px;
}
```

On ne fait plus désormais que d multiplications et d additions. Notons au passage que la stabilité numérique serait meilleure, si x était un nombre flottant.

16.3 Addition, soustraction

Si $P(X) = \sum_{i=0}^{n} p_i X^i$, $Q(X) = \sum_{j=0}^{m} q_j X^j$, alors

$$P(X) + Q(X) = \sum_{k=0}^{\min(n,m)} (p_k + q_k)X^k + \sum_{i=\min(n,m)+1}^{n} p_i X^i + \sum_{j=\min(n,m)+1}^{m} q_j X^j.$$

Le degré de $P + Q$ sera inférieur ou égal à $\max(n, m)$ (attention aux annulations).

Le code pour l'addition est alors :

```
  public static Polynome plus(Polynome P, Polynome Q){
      int maxdeg = (P.deg >= Q.deg ? P.deg : Q.deg);
      int mindeg = (P.deg <= Q.deg ? P.deg : Q.deg);
      Polynome R = new Polynome(maxdeg);

      for(int i = 0; i <= mindeg; i++)
          R.coeff[i] = P.coeff[i] + Q.coeff[i];
      for(int i = mindeg+1; i <= P.deg; i++)
          R.coeff[i] = P.coeff[i];
      for(int i = mindeg+1; i <= Q.deg; i++)
          R.coeff[i] = Q.coeff[i];
      trouverDegre(R);
      return R;
  }
```

Comme il faut faire attention au degré du résultat, qui est peut-être plus petit que prévu, on a dû introduire une nouvelle primitive qui se charge de mettre à jour le degré de P (on remarquera que si P est nul, le degré sera bien mis à -1) :

```
  // vérification du degré
  public static void trouverDegre(Polynome P){
      while(P.deg >= 0){
          if(P.coeff[P.deg] != 0)
              break;
          else
              P.deg -= 1;
      }
  }
```

On procède de même pour la soustraction, en recopiant la fonction précédente, les seules modifications portant sur les remplacements de $+$ par $-$ aux endroits appropriés.

Il importe ici de bien tester les fonctions écrites. En particulier, il faut vérifier que la soustraction de deux polynômes identiques donne 0. Le programme de test contient ainsi une soustraction normale, suivie de deux soustractions avec diminution du degré :

```
public static void testerSous(){
    Polynome P, Q, S;

    P = Polynome.deChaine("1 1 1"); // X+1
    Q = Polynome.deChaine("2 2 2 2"); // X^2+X+2
    System.out.println("P="+P+" Q="+Q);
    System.out.println("P-Q="+Polynome.sous(P, Q));
    System.out.println("Q-P="+Polynome.sous(Q, P));

    Q = Polynome.deChaine("1 1 0"); // X
    System.out.println("Q="+Q);
    System.out.println("P-Q="+Polynome.sous(P, Q));
    System.out.println("P-P="+Polynome.sous(P, P));
```

```
}
```

dont l'exécution donne :

```
P=X+1 Q=2*X^2+2*X+2
P-Q=-2*X^2-X-1
Q-P=2*X^2+X+1
Q=1
P-Q=X
P-P=0
```

16.4 Deux algorithmes de multiplication

16.4.1 Multiplication naïve

Soit $P(X) = \sum_{i=0}^{n} p_i X^i$, $Q(X) = \sum_{j=0}^{m} q_j X^j$, alors

$$P(X)Q(X) = \sum_{k=0}^{n+m} \left(\sum_{i+j=k} p_i q_j \right) X^k.$$

Le code correspondant en Java est :

```
public static Polynome mult(Polynome P, Polynome Q){
    Polynome R;

    if(estNul(P)) return copier(P);
    else if(estNul(Q)) return copier(Q);
    R = new Polynome(P.deg + Q.deg);
    for(int i = 0; i <= P.deg; i++)
        for(int j = 0; j <= Q.deg; j++)
            R.coeff[i+j] += P.coeff[i] * Q.coeff[j];
    return R;
}
```

16.4.2 L'algorithme de Karatsuba

Nous allons utiliser une approche diviser pour résoudre de la multiplication de polynômes.

Comment fait-on pour multiplier deux polynômes de degré 1 ? On écrit :

$$P(X) = p_0 + p_1 X, \quad Q(X) = q_0 + q_1 X,$$

et on va calculer

$$R(X) = P(X)Q(X) = r_0 + r_1 X + r_2 X^2,$$

avec

$$r_0 = p_0 q_0, \quad r_1 = p_0 q_1 + p_1 q_0, \quad r_2 = p_1 q_1.$$

Pour calculer le produit $R(X)$, on fait 4 multiplications sur les coefficients, que nous appelerons *multiplication élémentaire* et dont le coût sera l'unité de calcul pour les comparaisons à venir. Nous négligerons les coûts d'addition et de soustraction.

Si maintenant P est de degré $n-1$ et Q de degré $n-1$ (ils ont donc n termes), on peut écrire :

$$P(X) = P_0(X) + X^m P_1(X), \quad Q(X) = Q_0(X) + X^m Q_1(X),$$

où $m = \lceil n/2 \rceil$, avec P_0 et Q_0 de degré $m-1$ et P_1, Q_1 de degré $n-1-m$. On a alors :

$$R(X) = P(X)Q(X) = R_0(X) + X^m R_1(X) + X^{2m} R_2(X),$$

$$R_0 = P_0 Q_0, \quad R_1 = P_0 Q_1 + P_1 Q_0, \quad R_2 = P_1 Q_1.$$

Notons $\mathcal{M}(d)$ le nombre de multiplications élémentaires nécessaires pour calculer le produit de deux polynômes de degré $d-1$. On vient de voir que :

$$\mathcal{M}(2^1) = 4\mathcal{M}(2^0).$$

Si $n = 2^t$, on a $m = 2^{t-1}$ et :

$$\mathcal{M}(2^t) = 4\mathcal{M}(2^{t-1}) = O(2^{2t}) = O(n^2).$$

L'idée de Karatsuba est de remplacer 4 multiplications élémentaires par 3, en utilisant une approche dite évaluation/interpolation. On sait qu'un polynôme de degré n est complètement caractérisé soit par la donnée de ses $n+1$ coefficients, soit par ses valeurs en $n+1$ points distincts (en utilisant par exemple les formules d'interpolation de Lagrange). L'idée de Karatsuba est d'évaluer le produit PQ en trois points 0, 1 et ∞. On écrit :

$$R_0 = P_0 Q_0,\ R_2 = P_1 Q_1,\ R_1 = (P_0 + P_1)(Q_0 + Q_1) - R_0 - R_1$$

ce qui permet de ramener le calcul des R_i à une multiplication de deux polynômes de degré $m-1$, et deux multiplications en degré $n-1-m$ plus 2 additions et 2 soustractions. Dans le cas où $n = 2^t$, on obtient :

$$\mathcal{K}(2^t) = 3\mathcal{K}(2^{t-1}) = O(3^t) = O(n^{\log_2 3}) = O(n^{1.585}).$$

La fonction $\mathcal{K}$ vérifie plus généralement l'équation fonctionnelle :

$$\mathcal{K}(n) = 2\mathcal{K}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{K}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

et son comportement est délicat à prédire (on montre qu'elle a un comportement fractal).

Première implantation

Nous allons implanter les opérations nécessaires aux calculs précédents. On a besoin d'une fonction qui récupère P_0 et P_1 à partir de P. On écrit donc une fonction :

```
// crée le polynôme
//     P[début]+P[début+1]*X+...+P[fin]*X^(fin-début)
public static Polynome extraire(Polynome P,
                                 int début, int fin){
    Polynome E = new Polynome(fin-début);

    for(int i = début; i <= fin; i++)
        E.coeff[i-début] = P.coeff[i];
    trouverDegre(E);
    return E;
}
```

Quel va être le prototype de la fonction de calcul, ainsi que les hypothèses faites en entrée ? Nous décidons ici d'utiliser :

```
// ENTRÉE: deg(P) = deg(Q) <= n-1,
//     P.coeff et Q.coeff sont de taille >= n;
// SORTIE: R tq R = P*Q et deg(R) <= 2*(n-1).
public static Polynome Karatsuba(Polynome P, Polynome Q,
                                 int n){
```

Nous fixons donc arbitrairement le degré de P et Q à $n-1$. Une autre fonction est supposée être en charge de la normalisation des opérations, par exemple en créant des objets de la bonne taille.

On remarque également, avec les notations précédentes, que P_0 et Q_0 sont de degré $m-1$, qui est toujours plus grand que le degré de P_1 et Q_1, à savoir $n-m-1$. Il faudra donc faire attention au calcul de la somme $P_0 + P_1$ (resp. $Q_0 + Q_1$) ainsi qu'au calcul de R_1.

La fonction complète est donnée dans la table 16.2.

Expliquons la remarque 1. On décide pour l'instant d'arrêter la récursion quand on doit multiplier deux polynômes de degré 0 (donc $n = 1$).

La remarque 2 est justifiée par notre invariant de fonction : les degrés de `SP` et `SQ` (ou plus exactement la taille de leurs tableaux de coefficients), qui vont être passés à `Karatsuba` doivent être $m-1$. Il nous faut donc modifier l'appel `plus(P0, P1);` en celui `plusKara(P0, P1, m-1);` qui retourne la somme de `P0` et `P1` dans un polynôme dont le nombre de coefficients est toujours m, quel que soit le dégré de la somme (penser que l'on peut tout à fait avoir $P_0 = 0$ et P_1 de degré $m-2$).

```
// ENTREE: deg(P), deg(Q) <= d.
// SORTIE: P+Q dans un polynôme R tel que R.coeff a taille
//         d+1.
public static Polynome plusKara(Polynome P, Polynome Q,
                                int d){
    int mindeg = (P.deg <= Q.deg ? P.deg : Q.deg);
    Polynome R = new Polynome(d);

    //PrintK("plusKara("+d+", "+mindeg+"): "+P+" "+Q);
    for(int i = 0; i <= mindeg; i++)
```

```
public static Polynome Karatsuba(Polynome P, Polynome Q,
                                 int n){
    Polynome P0, P1, Q0, Q1, SP, SQ, R0, R1, R2, R;
    int m;

    if(n <= 1)                // (cf. remarque 1)
        return mult(P, Q);
    m = n/2;
    if((n % 2) == 1) m++;
    // on multiplie P = P0 + X^m * P1 avec Q = Q0 + X^m * Q1
    // deg(P0), deg(Q0) <= m-1
    // deg(P1), deg(Q1) <= n-1-m <= m-1
    P0 = extraire(P, 0, m-1);
    P1 = extraire(P, m, n-1);
    Q0 = extraire(Q, 0, m-1);
    Q1 = extraire(Q, m, n-1);

    // R0 = P0*Q0 de degré 2*(m-1)
    R0 = Karatsuba(P0, Q0, m);

    // R2 = P2*Q2 de degré 2*(n-1-m)
    R2 = Karatsuba(P1, Q1, n-m);

    // R1 = (P0+P1)*(Q0+Q1)-R0-R2
    // deg(P0+P1), deg(Q0+Q1) <= max(m-1, n-1-m) = m-1
    SP = plusKara(P0, P1, m-1);       // (cf. remarque 2)
    SQ = plusKara(Q0, Q1, m-1);
    R1 = Karatsuba(SP, SQ, m);
    R1 = sous(R1, R0);
    R1 = sous(R1, R2);
    // on reconstruit le résultat
    // R = R0 + X^m * R1 + X^(2*m) * R2
    R = new Polynome(2*(n-1));
    for(int i = 0; i <= R0.deg; i++)
        R.coeff[i] = R0.coeff[i];
    for(int i = 0; i <= R2.deg; i++)
        R.coeff[2*m + i] = R2.coeff[i];
    for(int i = 0; i <= R1.deg; i++)
        R.coeff[m + i] += R1.coeff[i];
    trouverDegre(R);
    return R;
  }
```

FIG. 16.2 – Algorithme de Karatsuba.

```
        R.coeff[i] = P.coeff[i] + Q.coeff[i];
    for(int i = mindeg+1; i <= P.deg; i++)
        R.coeff[i] = P.coeff[i];
    for(int i = mindeg+1; i <= Q.deg; i++)
        R.coeff[i] = Q.coeff[i];
    return R;
  }
```

Comment teste-t-on un tel programme ? Tout d'abord, nous avons de la chance, car nous pouvons comparer `Karatsuba` à `mul`. Un programme test prend en entrée des couples de polynômes (P, Q) de degré n et va comparer les résultats des deux fonctions. Pour ne pas avoir à rentrer des polynômes à la main, on construit une fonction qui fabrique des polynômes (unitaires) "aléatoires" à l'aide d'un générateur créé pour la classe :

```
public static Random rd = new Random();

public static Polynome aleatoire(int deg){
    Polynome P = new Polynome(deg);

    P.coeff[deg] = 1;
    for(int i = 0; i < deg; i++)
        P.coeff[i] = rd.nextLong();
    return P;
}
```

La méthode `rd.nextLong()` retourne un entier "aléatoire" de type `long` fabriqué par le générateur `rd`.

Le programme test, dans lequel nous avons également rajouté une mesure du temps de calcul est alors :

```
// testons Karatsuba sur n polynômes de degré deg
public static void testerKaratsuba(int deg, int n){
    Polynome P, Q, N, K;
    long tN, tK, totN = 0, totK = 0;

    for(int i = 0; i < n; i++){
        P = Polynome.aleatoire(deg);
        Q = Polynome.aleatoire(deg);
        TC.demarrerChrono();
        N = Polynome.mult(P, Q);
        tN = TC.tempsChrono();

        TC.demarrerChrono();
        K = Polynome.Karatsuba(P, Q, deg+1);
        tK = TC.tempsChrono();

        if(! Polynome.estEgal(K, N)){
```

```
            System.out.println("Erreur");
            System.out.println("P*Q(norm)=" + N);
            System.out.println("P*Q(Kara)=" + K);
            for(int i = 0; i <= N.deg; i++){
                if(K.coeff[i] != N.coeff[i])
                    System.out.print(" "+i);
            }
            System.out.println("");
            System.exit(-1);
        }
        else{
            totN += tN;
            totK += tK;
        }
    }
    System.out.println(deg+" N/K = "+totN+" "+totK);
}
```

Que se passe-t-il en pratique ? Voici des temps obtenus avec le programme précédent, pour $100 \leqslant deg \leqslant 1000$ par pas de 100, avec 10 couples de polynômes à chaque fois :

```
Test de Karatsuba
100 N/K = 2 48
200 N/K = 6 244
300 N/K = 14 618
400 N/K = 24 969
500 N/K = 37 1028
600 N/K = 54 2061
700 N/K = 74 2261
800 N/K = 96 2762
900 N/K = 240 2986
1000 N/K = 152 3229
```

Cela semble frustrant, Karatsuba ne battant jamais (et de très loin) l'algorithme naïf sur la plage considérée. On constate cependant que la croissance des deux fonctions est à peu près la bonne, en comparant par exemple le temps pris pour d et $2d$ (le temps pour le calcul naïf est multiplié par 4, le temps pour Karatsuba par 3).

Comment faire mieux ? L'astuce classique ici est de décider de repasser à l'algorithme de multiplication classique quand le degré est petit. Par exemple ici, on remplace la ligne repérée par la remarque 1 en :

```
if(n <= 16)
```

ce qui donne :

```
Test de Karatsuba
100 N/K = 1 4
200 N/K = 6 6
300 N/K = 14 13
```

```
400 N/K = 24 17
500 N/K = 38 23
600 N/K = 164 40
700 N/K = 74 69
800 N/K = 207 48
900 N/K = 233 76
1000 N/K = 262 64
```

Le réglage de cette constante est critique et dépend de la machine sur laquelle on opère.

Remarques sur une implantation optimale

La fonction que nous avons implantée ci-dessus est gourmande en mémoire, car elle alloue sans cesse des polynômes auxiliaires. Diminuer ce nombre d'allocations (il y en a $O(n^{1.585})$ également...) est une tâche majeure permettant de diminuer le temps de calcul. Une façon de faire est de travailler sur des polynômes définis par des extraits compris entre des indices de début et de fin. Par exemple, le prototype de la fonction pourrait devenir :

```
public static Polynome Karatsuba(Polynome P, int dP, int fP,
                                 Polynome Q, int dQ, int fQ, int n){
```

qui permettrait de calculer le produit de $P' = P_{fP}X^{fP-dP} + \cdots + P_{dP}$ et $Q' = P_{fQ}X^{fQ-dQ} + \cdots + Q_{dQ}$. Cela nous permettrait d'appeler directement la fonction sur P'_0 et P'_1 (resp. Q'_0 et Q'_1) et éviterait d'avoir à extraire les coefficients.

Dans le même ordre d'idée, l'addition et la soustraction pourraient être faites *en place*, c'est-à-dire qu'on implanterait plutôt $P := P - Q$.

16.5 Multiplication à l'aide de la transformée de Fourier*

Quel est le temps minimal requis pour faire le produit de deux polynômes de degré n ? On vient de voir qu'il existe une méthode en $O(n^{1.585})$. Peut-on faire mieux ? L'approche de Karatsuba consiste à couper les arguments en deux. On peut imaginer de couper en 3, voire plus. On peut démontrer qu'asymptotiquement, cela conduit à une méthode dont le nombre de multiplications élémentaires est $O(n^{1+\varepsilon})$ avec $\varepsilon > 0$ aussi petit qu'on le souhaite.

Il existe encore une autre manière de voir les choses. L'algorithme de Karatsuba est le prototype des méthodes de multiplication par évaluation/interpolation. On a calculé $R(0)$, $R(1)$ et $R(+\infty)$ et de ces valeurs, on a pu déduire la valeur de $R(X)$. L'approche de Cooley et Tukey consiste à interpoler le produit R sur des racines de l'unité bien choisies.

16.5.1 Transformée de Fourier

Définition 1 *Soit $\omega \in \mathbb{C}$ et N un entier. La transformée de Fourier est une application*

$$\begin{array}{rccc} \mathcal{F}_\omega : & \mathbb{C}^N & \to & \mathbb{C}^N \\ & (a_0, a_1, \ldots, a_{N-1}) & \mapsto & (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1}) \end{array}$$

où

$$\hat{a}_i = \sum_{j=0}^{N-1} \omega^{ij} a_j$$

pour $0 \leqslant i \leqslant N-1$.

Proposition 9 *Si* ω *est une racine primitive* N*-ième de l'unité, (i.e.,* $\omega^N = 1$ *et* $\omega^i \neq 1$ *pour* $1 \leqslant i < N$*), alors* $\mathcal{F}_\omega$ *est une bijection et*

$$\mathcal{F}_\omega^{-1} = \frac{1}{N} \mathcal{F}_{\omega^{-1}}.$$

Démonstration : Posons

$$\alpha_i = \frac{1}{N} \sum_{k=0}^{N-1} \omega^{-ik} \hat{a}_k.$$

On calcule

$$N\alpha_i = \sum_k \omega^{-ik} \sum_j \omega^{kj} a_j = \sum_j a_j \sum_k \omega^{k(j-i)} = \sum_j a_j S_{i,j}.$$

Si $i = j$, on a $S_{i,j} = N$ et si $j \neq i$, on a

$$S_{i,j} = \sum_{k=0}^{N-1} (\omega^{j-i})^k = \frac{1 - (\omega^{j-i})^N}{1 - \omega^{j-i}} = 0. \square$$

16.5.2 Application à la multiplication de polynômes

Soient

$$P(X) = \sum_{i=0}^{n-1} p_i X^i, \quad Q(X) = \sum_{i=0}^{n-1} q_i X^i$$

deux polynômes dont nous voulons calculer le produit :

$$R(X) = \sum_{i=0}^{2n-1} r_i X^i$$

(avec $r_{2n-1} = 0$). On utilise une transformée de Fourier de taille $N = 2n$ avec les vecteurs :

$$p = (p_0, p_1, \ldots, p_{n-1}, \underbrace{0, 0, \ldots, 0}_{n \text{ termes}}),$$

$$q = (q_0, q_1, \ldots, q_{n-1}, \underbrace{0, 0, \ldots, 0}_{n \text{ termes}}).$$

Soit ω une racine primitive $2n$-ième de l'unité. La transformée de p

$$\mathcal{F}_\omega(p) = (\hat{p}_0, \hat{p}_1, \ldots, \hat{p}_{2n-1})$$

n'est autre que :

$$(P(\omega^0), P(\omega^1), \ldots, P(\omega^{2n-1})).$$

De même pour q, de sorte que le *produit terme à terme* des deux vecteurs :

$$\mathcal{F}_\omega(p) \otimes \mathcal{F}_\omega(q) = (\hat{p}_0\hat{q}_0, \hat{p}_1\hat{q}_1, \ldots, \hat{p}_{2n-1}\hat{q}_{2n-1})$$

donne en fait les valeurs de $R(X) = P(X)Q(X)$ en les racines de l'unité, c'est-à-dire $\mathcal{F}_\omega(R)$! Par suite, on retrouve les coefficients de P en appliquant la transformée inverse.

Un algorithme en pseudo-code pour calculer R est alors :

- $N = 2n$, $\omega = \exp(2i\pi/N)$;
- calculer $\mathcal{F}_\omega(p)$, $\mathcal{F}_\omega(q)$;
- calculer $(\hat{r}_0, \hat{r}_1, \ldots, \hat{r}_{2n-1}) = \mathcal{F}_\omega(p) \otimes \mathcal{F}_\omega(q)$;
- récupérer les r_i par

$$(r_0, r_1, \ldots, r_{2n-1}) = (1/N)\mathcal{F}_{\omega^{-1}}(\hat{r}_0, \hat{r}_1, \ldots, \hat{r}_{2n-1}).$$

16.5.3 Transformée rapide

Transformée multiplicative

Si l'on s'y prend naïvement, le calcul des $\hat{x}_i$ définis par

$$\hat{x}_k = \sum_{m=0}^{N-1} x_m \omega^{mk}, 0 \leqslant k \leqslant N-1$$

prend N^2 multiplications[2].

Supposons que l'on puisse écrire N sous la forme d'un produit de deux entiers plus grands que 1, soit $N = N_1N_2$. On peut écrire :

$$m = N_1m_2 + m_1, \quad k = N_2k_1 + k_2$$

avec $0 \leqslant m_1, k_1 < N_1$ et $0 \leqslant m_2, k_2 < N_2$. Cela conduit à récrire :

$$\hat{x}_k = \sum_{m_1=0}^{N_1-1} \omega^{N_2m_1k_1}\omega^{m_1k_2} \sum_{m_2=0}^{N_2-1} x_{N_1m_2+m_1}\omega^{N_1m_2k_2}.$$

On peut montrer sans grande difficulté que $\omega_1 = \omega^{N_2}$ est racine primitive N_1-ième de l'unité, $\omega_2 = \omega^{N_1}$ est racine primitive N_2-ième. On se ramène alors à calculer :

$$\hat{x}_k = \sum_{m_1=0}^{N_1-1} \omega_1^{m_1k_1}\omega^{m_1k_2} \sum_{m_2=0}^{N_2-1} x_{N_1m_2+m_1}\omega_2^{m_2k_2}.$$

La deuxième somme est une transformée de longueur N_2 appliquée aux nombres

$$(x_{N_1m_2+m_1})_{0 \leqslant m_2 < N_2}.$$

Le calcul se fait donc comme celui de N_1 transformées de longueur N_2, suivi de multiplications par des facteurs $\omega^{m_1k_2}$, suivies elles-mêmes de N_2 transformées de longueur N_1.

Le nombre de multiplications élémentaires est alors :

$$N_1(N_2^2) + N_1N_2 + N_2(N_1^2) = N_1N_2(N_1 + N_2 + 1)$$

ce qui est en général plus petit que $(N_1N_2)^2$.

[2] On remarque que $\omega^{mk} = \omega^{(mk) \bmod N}$ et le précalcul des ω^i pour $0 \leqslant i < N$ coûte N multiplications élémentaires.

Le cas magique $N = 2^t$

Appliquons le résultat précédent au cas où $N_1 = 2$ et $N_2 = 2^{t-1}$. Les calculs que nous devons effectuer sont :

$$\begin{aligned}\hat{x}_k &= \sum_{m=0}^{N/2-1} x_{2m}(\omega^2)^{mk} + \omega^k \sum_{m=0}^{N/2-1} x_{2m+1}(\omega^2)^{mk} \\ \hat{x}_{k+N/2} &= \sum_{m=0}^{N/2-1} x_{2m}(\omega^2)^{mk} - \omega^k \sum_{m=0}^{N/2-1} x_{2m+1}(\omega^2)^{mk}\end{aligned}$$

car $\omega^{N/2} = -1$. Autrement dit, le calcul se divise en deux morceaux, le calcul des moitiés droite et gauche du signe + et les résultats sont réutilisés dans la ligne suivante.

Pour insister sur la méthode, nous donnons ici le pseudo-code en Java sur des vecteurs de nombres réels :

```
public static double[] FFT(double[] x, int N, double omega){
    double[] X = new double[N], xx, Y0, Y1;
    double omega2, omegak;

    if(N == 2){
        X[0] = x[0] + x[1];
        X[1] = x[0] - x[1];
        return X;
    }
    else{
        xx = new double[N/2];
        omega2 = omega*omega;
        for(m = 0; m < N/2; m++) xx[m] = x[2*m];
        Y0 = FFT(xx, N/2, omega2);
        for(m = 0; m < N/2; m++) xx[m] = x[2*m+1];
        Y1 = FFT(xx, N/2, omega2);
        omegak = 1.; // pour omega^k
        for(k = 0; k < N/2; k++){
            X[k]     = Y0[k] + omegak*Y1[k];
            X[k+N/2] = Y0[k] - omegak*Y1[k];
            omegak = omega * omegak;
        }
        return X;
    }
}
```

Le coût de l'algorithme est alors $F(N) = 2F(N/2) + N/2$ multiplications élémentaires. On résout la récurrence à l'aide de l'astuce suivante :

$$\frac{F(N)}{N} = \frac{F(N/2)}{N/2} + 1/2 = F(1) + t/2 = t/2$$

d'où $F(N) = \frac{1}{2} N \log_2 N$. Cette variante a ainsi reçu le nom de *transformée de Fourier rapide* (*Fast Fourier Transform* ou FFT).

À titre d'exemple, si $N = 2^{10}$, on fait 5×2^{10} multiplications au lieu de 2^{20}.

Remarques complémentaires

Nous avons donné ici une brève présentation de l'idée de la FFT. C'est une idée très importante à utiliser dans tous les algorithmes basés sur les convolutions, comme par exemple le traitement d'images, le traitement du signal, etc.

Il y a des milliards d'astuces d'implantation, qui s'appliquent par exemple aux problèmes de précision. C'est une opération tellement critique dans certains cas que du hardware spécifique existe pour traiter des FFT de taille fixe. On peut également chercher à trouver le meilleur découpage possible quand N n'est pas une puissance de 2. Le lecteur intéressé est renvoyé au livre de Nussbaumer [Nus82].

Signalons pour finir que le même type d'algorithme (Karatsuba, FFT) est utilisé dans les calculs sur les grands entiers, comme cela est fait par exemple dans la bibliothèque multiprécision GMP (cf. `http://www.swox.com/gmp/`).

Cinquième partie

Annexes

Annexe A

Compléments

A.1 Exceptions

Les exceptions sont des objets de la classe `Exception`. Il existe aussi une classe `Error` moins utilisée pour les erreurs système. Toutes les deux sont des sous-classes de la classe `Throwable`, dont tous les objets peuvent être appliqués à l'opérateur `throw`, comme suit :

```
throw e ;
```

Ainsi on peut écrire en se servant de deux constructeurs de la classe `Exception` :

```
throw new Exception();
throw new Exception ("Accès interdit dans un tableau");
```

Heureusement, dans les classes des bibliothèques standard, beaucoup d'exceptions sont déjà pré-définies, par exemple `IndexOutOfBoundsException`. On récupère une exception par l'instruction `try ... catch`. Par exemple

```
try {
  // un programme compliqué
} catch ( IOException e) {
  // essayer de réparer cette erreur d'entrée/sortie
}
catch ( Exception e) {
  //  essayer de réparer cette erreur plus générale
}
```

Si on veut faire un traitement uniforme en fin de l'instruction `try`, que l'on passe ou non par une exception, que le contrôle sorte ou non de l'instruction par une rupture de séquence comme un `return`, `break`, etc, on écrit

```
try {
  // un programme compliqué
} catch ( IOException e) {
  //  essayer de réparer cette erreur d'entrée/sortie
```

```
}
catch ( Exception e) {
  //essayer de réparer cette erreur plus générale
}
finally {
  // un peu de nettoyage
}
```

Il y a deux types d'exceptions. On doit déclarer les *exceptions vérifiées* derrière le mot-clé `throws` dans la signature des fonctions qui les lèvent. Ce n'est pas la peine pour les *exceptions non vérifiées* qui se reconnaissent en appartenant à une sous-classe de la classe `RuntimeException`. Ainsi

```
static int lire () throws IOException, ParseException {
  int n;
  BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));

  System.out.print ("Taille du carré magique, svp?::  ");
  n = Integer.parseInt (in.readLine());
  if ((n <= 0) || (n > N) || (n % 2 == 0))
      erreur ("Taille impossible.");
  return n;
}
```

A.2 La classe MacLib

Cette classe est l'œuvre de Philippe Chassignet, et elle a survécu à l'évolution de l'enseignement d'informatique à l'X. Jusqu'en 1992, elle permettait de faire afficher sur un écran de Macintosh des dessins produits sur un Vax (via TGiX, autre interface du même auteur). Elle a ensuite été adaptée en 1994 à **ThinkPascal** sur Macintosh, puis **TurboPascal** sous PC-Windows ; puis à **ThinkC** et **TurboC**, X11 ; **DelphiPascal**, **Borland C** (nouveau Windows) ; **CodeWarrior Pascal et C**, tout ça dans la période 1996–1998. En 1998 elle a commencé une nouvelle adaptation, avec JAVA, qui a constitué une simplification énorme du travail, la même version marchant sur toutes les plateformes ! Elle est désormais interfacée avec l'AWT (*Abstract Windowing Toolkit*), avec un usage souple de la boucle d'événement.

A.2.1 Fonctions élémentaires

Les fonctions sont inspirées de la libraire QuickDraw du Macintosh. La méthode `initQuickDraw()` – dont l'utilisation est impérative et doit précéder toute opération de dessin – permet d'initialiser la fenêtre de dessin. Cette fenêtre *Drawing* créée par défaut permet de gérer un écran de 1024×768 points. L'origine du système de coordonnées est en haut et à gauche. L'axe des x va classiquement de la gauche vers la droite, l'axe des y va plus curieusement du haut vers le bas (c'est une vieille tradition de l'informatique, dure à remettre en cause). En QuickDraw, x et y sont souvent appelés `h` (horizontal) et `v` (vertical). Il y a une notion de point courant et de crayon avec une

taille et une couleur courantes. On peut déplacer le crayon, en le levant ou en dessinant des vecteurs par les fonctions suivantes

moveTo (x, y) Déplace le crayon aux coordonnées absolues `x`, `y`.

move (dx, dy) Déplace le crayon en relatif de `dx`, `dy`.

lineTo (x, y) Trace une ligne depuis le point courant jusqu'au point de coordonnées `x`, `y`.

line (dx, dy) Trace le vecteur (`dx`, `dy`) depuis le point courant.

penSize(dx, dy) Change la taille du crayon. La taille par défaut est (1, 1). Toutes les opérations de tracé peuvent se faire avec une certaine épaisseur du crayon.

penMode(mode) Change le mode d'écriture : `patCopy` (mode par défaut qui remplace ce sur quoi on trace), `patXor` (mode Xor, i.e. en inversant ce sur quoi on trace).

A.2.2 Rectangles

Certaines opérations sont possibles sur les rectangles. Un rectangle `r` a un type prédéfini `Rect`. Ce type est une classe qui a le format suivant

```
public class Rect {

  short left, top, right, bottom;
}
```

Fort heureusement, il n'y a pas besoin de connaître le format internes des rectangles, et on peut faire simplement les opérations graphiques suivantes sur les rectangles

setRect(r, g, h, d, b) fixe les coordonnées (gauche, haut, droite, bas) du rectangle `r`. C'est équivalent à faire les opérations `r.left := g;`, `r.top := h;`, `r.right := d;`, `r.bottom := b`. Le rectangle `r` doit déjà avoir été construit.

unionRect(r1, r2, r) définit le rectangle `r` comme l'enveloppe englobante des rectangles `r1` et `r2`. Le rectangle `r` doit déjà avoir été construit.

frameRect(r) dessine le cadre du rectangle `r` avec la largeur, la couleur et le mode du crayon courant.

paintRect(r) remplit l'intérieur du rectangle `r` avec la couleur courante.

invertRect(r) inverse la couleur du rectangle `r`.

eraseRect(r) efface le rectangle `r`.

drawChar(c), drawString(s) affiche le caractère `c` ou la chaîne `s` au point courant dans la fenêtre graphique. Ces fonctions diffèrent de `write` ou `writeln` qui écrivent dans la fenêtre texte.

frameOval(r) dessine le cadre de l'ellipse inscrite dans le rectangle `r` avec la largeur, la couleur et le mode du crayon courant.

paintOval(r) remplit l'ellipse inscrite dans le rectangle `r` avec la couleur courante.

invertOval(r) inverse l'ellipse inscrite dans `r`.

eraseOval(r) efface l'ellipse inscrite dans `r`.

frameArc(r,start,arc) dessine l'arc de l'ellipse inscrite dans le rectangle `r` démarrant à l'angle `start` et sur la longueur définie par l'angle `arc`.

frameArc(r,start,arc) peint le camembert correspondant à l'arc précédent .... Il y a aussi des fonctions pour les rectangles avec des coins arrondis.

button() est une fonction qui renvoie la valeur vraie si le bouton de la souris est enfoncé, faux sinon.

getMouse(p) renvoie dans `p` le point de coordonnées (`p.h`, `p.v`) courantes du curseur.

A.2.3 La classe `Maclib`

```
public class Point {
  short h, v;

  Point(int h, int v) {
    h = (short)h;
    v = (short)v;
  }
}
public class MacLib {

  static void setPt(Point p, int h, int v) {..}
  static void addPt(Point src, Point dst) {...}
  static void subPt(Point src, Point dst) {...}
  static boolean equalPt(Point p1, Point p2) {...}
  ...
}
```

Et les fonctions correspondantes (voir page 235)

```
static void setRect(Rect r, int left, int top, int right, int bottom)
static void unionRect(Rect src1, Rect src2, Rect dst)

static void frameRect(Rect r)
static void paintRect(Rect r)
static void eraseRect(Rect r)
static void invertRect(Rect r)

static void frameOval(Rect r)
static void paintOval(Rect r)
static void eraseOval(Rect r)
static void invertOval(Rect r)

static void frameArc(Rect r, int startAngle, int arcAngle)
static void paintArc(Rect r, int startAngle, int arcAngle)
static void eraseArc(Rect r, int startAngle, int arcAngle)
static void invertArc(Rect r, int startAngle, int arcAngle)
static boolean button()
static void getMouse(Point p)
static void getClick(Point p)
```

Toutes ces définitions sont aussi sur les stations de travail, dans le fichier

`/usr/local/lib/MacLib-java/MacLib.java`

On veillera à avoir cette classe dans l'ensemble des classes chargeables (variable d'environnement `CLASSPATH`).

A.2.4 Jeu de balle

Le programme suivant fait rebondir une balle dans un rectangle, première étape vers un jeu de *pong*.

```
class Pong{

  static final int  C  =  5, // Le rayon de la balle
      X0 =  5, X1 =  250,
      Y0 =  5, Y1 =  180;

  public static void main(String args[]) {
      int   x, y, dx, dy;
      Rect  r = new Rect();
      Rect  s = new Rect();
      Point p = new Point();
      int  i;

      // Initialisation du graphique
      MacLib.initQuickDraw();
      MacLib.setRect(s, 50, 50, X1 + 100, Y1 + 100);
      MacLib.setDrawingRect(s);
      MacLib.showDrawing();
      MacLib.setRect(s, X0, Y0, X1, Y1);

      // le rectangle de jeu
      MacLib.frameRect(s);
      // on attend un click et on note les coordonnées
      // du pointeur

      MacLib.getClick(p);
      x = p.h; y = p.v;
      // la vitesse initiale de la balle
      dx = 1;
      dy = 1;
      while(true){
           MacLib.setRect(r, x - C, y - C, x + C, y + C);
           // on dessine la balle en x,y
           MacLib.paintOval(r);
```

```
            x = x + dx;
            if(x - C <= X0 + 1 || x + C >= X1 - 1)
                dx = -dx;
            y = y + dy;
            if(y - C <= Y0 + 1 || y + C >= Y1 - 1)
                dy = -dy;
            // On temporise
            for(i = 1; i <= 2500; ++i)
                ;
            // On efface la balle
            MacLib.invertOval(r);
        }
    }
}
```

A.3 La classe TC

Le but de cette classe écrite spécialement pour le cours par l'auteur du présent poly (F. Morain) est de fournir quelques fonctions pratiques pour les TP, comme des entrées-sorties faciles, ou encore un chronomètre. Les exemples qui suivent sont presque tous donnés dans la classe `TestTC` qui est dans le même fichier que `TC.java` (qui se trouve lui-même accessible à partir des pages web du cours).

A.3.1 Fonctionnalités, exemples

Gestion du terminal

Le deuxième exemple de programmation consiste à demander un entier à l'utilisateur et affiche son carré. Avec la classe `TC`, on écrit :

```
public class TCex{
    public static void main(String[] args){
        int n;

        System.out.print("Entrer n=");
        n = TC.lireInt();
        System.out.println("n^2=" + (n*n));
    }
}
```

La classe `TC` contient d'autres primitives de ce type, comme `TC.lireLong`, ou encore `lireLigne`.

Si l'on veut lire plusieurs nombres[1], ou bien si on veut les lire sur l'entrée standard, par exemple en exécutant

[1] Ce passage peut être sauté en première lecture.

```
unix% java prgm < fichier
```

il convient d'utiliser la syntaxe :

```
do{
    System.out.print("Entrer n=");
    n = TC.lireInt();
    if(! TC.eof())
        System.out.println("n^2=" + (n*n));
} while(! TC.eof());
```

qui teste explicitement si l'on est arrivé à la fin du fichier (ou si on a quitté le programme).

Lectures de fichier

Supposons que le fichier `fic.int` contienne les entiers suivants :

```
1 2 3 4
5
6
6
7
```

et qu'on veuille récupérer un tableau contenant tous ces entiers. On utilise :

```
int[] tabi = TC.intDeFichier("fic.int");

for(int i = 0; i < tabi.length; i++)
    System.out.println(""+tabi[i]);
```

On peut lire des `double` par :

```
double[] tabd = TC.doubleDeFichier("fic.double");

for(int i = 0; i < tabd.length; i++)
    System.out.println(""+tabd[i]);
```

La fonction

```
static char[] charDeFichier(String nomfichier)
```

permet de récupérer le contenu d'un fichier dans un tableau de caractères. De même, la fonction

```
static String StringDeFichier(String nomfichier)
```

retourne une chaîne qui contient le fichier `nomfichier`.

On peut également récupérer un fichier sous forme de tableau de lignes à l'aide de :

```
static String[] lignesDeFichier(String nomfichier)
```

La fonction

```
static String[] motsDeFichier(String nomfichier)
```

retourne un tableau de chaînes contenant les mots contenus dans un fichier, c'est-à-dire les chaînes de caractères séparées par des blancs, ou bien des caractères de tabulation, etc.

Sortie dans un fichier

Il s'agit là d'une fonctionnalité spéciale. En effet, on souhaite pouvoir faire à la fois une sortie d'écran normale, ainsi qu'une sortie simultanée dans un fichier. L'utilisation typique en est la pale machine.

Pour bénéficier de cette fonctionnalité, on utilise :

```
TC.sortieFichier("nom_sortie");
TC.print(3);
TC.println(" allo");
```

ce qui a pour effet d'afficher à l'écran

```
3 allo
```

ainsi que d'en faire une copie dans le fichier `nom_sortie`. Les fonctions `TC.print` et `TC.println` peuvent être utilisées en lieu et place de `System.out.print` et `System.out.println`.

Si on oublie de faire l'initialisation, pas de problème, les résultats s'affichent à l'écran comme si de rien n'était.

Conversions à partir des `String`

Souvent, on récupère une chaîne de caractères (par exemple une ligne) et on veut la couper en morceaux. La fonction

```
static String[] motsDeChaine(String s)
```

retourne un tableau contenant les mots de la chaîne.

Si on est sûr que `s` ne contient que des entiers séparés par des blancs (cf. le chapitre sur les polynômes), la fonction

```
static int[] intDeChaine(String s)
```

retourne un tableau d'entiers contenus dans `s`. Par exemple si `s="1 2 3"`, on récupèrera un tableau contenant les trois entiers 1, 2, 3 dans cet ordre. Si l'on a affaire à des entiers de type `long`, on utilisera :

```
static long[] longDeChaine(String s)
```

Utilisation du chronomètre

La syntaxe d'utilisation du chronomètre est la suivante :

```
long t;

TC.demarrerChrono();
N = 1 << 10;
t = TC.tempsChrono();
```

La variable t contiendra le temps écoulé pendant la suite d'opérations effectuées depuis le lancement du chronomètre.

A.3.2 La classe Efichier

Cette classe rassemble des primitives de traitement des fichiers en entrée. Les explications qui suivent peuvent être omises en première lecture.

L'idée est d'encapsuler le traitement des fichiers dans une structure nouvelle, appelée Efichier (pour fichier d'entrées). Cette structure est définie comme :

```
public class Efichier{
    BufferedReader buf;
    String ligne;
    StringTokenizer tok;
    boolean eof, eol;
}
```

On voit qu'elle contient un tampon d'entrée à la JAVA, une ligne courante (ligne), un tokeniser associé (tok), et gère elle-même les fins de ligne (variable eol) et la fin du fichier (variable eof). On a défini deux constructeurs :

```
    Efichier(String nomfic){
        try{
            buf = new BufferedReader(new FileReader(new File(nomfic)));
        }
        catch(IOException e){
            System.out.println(e);
        }
        ligne = null;
        tok = null;
        eof = false;
        eol = false;
    }

    Efichier(InputStreamReader isr){
        buf = new BufferedReader(isr);
        ligne = null;
        tok = null;
        eof = false;
        eol = false;
    }
```

le second permettant d'utiliser l'entrée standard sous la forme :

```
static Efichier STDIN = new Efichier(new InputStreamReader(System.in));
```

La lecture d'une nouvelle ligne peut alors se faire par :

```
    String lireLigne(){
        try{
            ligne = buf.readLine();
        }
        catch(EOFException e){
            eol = true;
            eof = true;
            return null;
        }
        catch(IOException e){
            erreur(e);
        }
        eol = true;
        if(ligne == null)
            eof = true;
        else
            tok = new StringTokenizer(ligne);
        return ligne;
    }
```

où nous gérons tous les cas, et en particulier `eof` "à la main". On définit également `lireMotSuivant`, etc.

Les fonctionnalités de gestion du terminal sont encapsulées à leur tour dans la classe `TC`, ce qui permet d'écrire entre autres :

```
public class TC{
    static boolean eof(){
        return Efichier.STDIN.eof;
    }

    static int lireInt(){
        return Efichier.STDIN.lireInt();
    }
...
}
```

Bibliographie

[AS85] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[Bro95] F. P. Brooks. *The Mythical Man-Month : Essays on Software Engineering, Anniversary Edition (2nd Edition)*. Addison-Wesley Professional, 1995.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[GG99] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[Kle71] Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1971. 6ème édition (1ère en 1952).

[Knu73] Donald E. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison Wesley, 1973.

[Knu81] D. E. Knuth. *The Art of Computer Programming : Seminumerical Algorithms*. Addison-Wesley, 2nd edition, 1981.

[Mai77] H. Mairson. Some new upper bounds on the generation of prime numbers. *Comm. ACM*, 20(9) :664–669, September 1977.

[Mye04] G. L. Myers. *The Art of Software Testing*. Wiley, 2nd edition, 2004.

[Nus82] H. J. Nussbaumer. *Fast Fourier transform and convolution algorithms*, volume 2 of *Springer Series in Information Sciences*. Springer-Verlag, 2 edition, 1982.

[Rog87] Hartley Rogers. *Theory of recursive functions and effective computability*. MIT press, 1987. Édition originale McGraw-Hill, 1967.

[Sed88] Bob Sedgewick. *Algorithms*. Addison-Wesley, 1988. En français : *Algorithmes en langage C*, traduit par Jean-Michel Moreau, InterEditions, 1991.

Index

Table des figures