UNIVERSITÉ PARIS VII
DENIS DIDEROT

LICENCE D'INFORMATIQUE

L1 - S2

CONCEPTS INFORMATIQUES (CI2)

Support de cours - 2007/2008

Jean-Marie Rifflet

Version du 18 Février 2008

Table des matières

1

Rappels Java : variables, types primitifs et types références

1.1 Les références de tableaux en Java : étude d'un exemple

```
--> cat Adresse1.java
/   1  / class Adresse1 {
/   2  /  public static void main(String[] args) {
/   3  /    int n = 32;
/   4  /    System.out.println("valeur de n : " + n);
/   5  /    int[] t1;
/   6  /    // System.out.println("valeur de t1 " + t1);
/   7  /    t1 = new int[3]; // allocation en mémoire d'un tableau de 3 entiers
/   8  /    System.out.println("valeur de t1 : " + t1);
/   9  /    for(int i = 0; i < t1.length; i ++) System.out.print(t1[i] + " ");
/  10  /    System.out.println();
/  11  /    int[][] t2;
/  12  /    t2 = new int[4][];
/  13  /    System.out.println("valeur de t2 : " + t2);
/  14  /    for(int i = 0; i < t2.length; i++)  System.out.print(t2[i] + " ");
/  15  /    System.out.println("\n-------------------------");
/  16  /    t2[0] = t1;
/  17  /    System.out.println("valeur de t2[0] " + t2[0]);
/  18  /    t2[2] = new int[2];
/  19  /    System.out.println("valeur de t2[2] " + t2[2]);
/  20  /    System.out.println("valeur de t2[3] " + t2[3]);
/  21  /    System.out.println("-------------------------");
/  22  /    System.out.println("valeur de t1[0] : " +  t1[0]);
/  23  /    System.out.println("valeur de t2[0][0] : " +  t2[0][0]);
/  24  /    t1[0] = 1000;
/  25  /    System.out.println("valeur de t1[0] : " +  t1[0]);
/  26  /    System.out.println("valeur de t2[0][0] : " +  t2[0][0]);
/  27  /    }
/  28  / }
--> java Adresse1
valeur de n : 32             <== resultat correspondant a la ligne 4
valeur de t1 : [I@e09713     <== resultat correspondant a la ligne 8
0 0 0                        <== resultat correspondant a la ligne 9
valeur de t2 : [[I@de6f34    <== resultat correspondant a la ligne 13
null null null null          <== resultat correspondant a la ligne 14
-------------------------
valeur de t2[0] : [I@e09713  <== resultat correspondant a la ligne 17
valeur de t2[2] : [I@6ee8e   <== resultat correspondant a la ligne 19
valeur de t2[3] : null       <== resultat correspondant a la ligne 20
-------------------------
valeur de t1[0] : 0          <== resultat correspondant a la ligne 22
valeur de t2[0][0] : 0       <== resultat correspondant a la ligne 23
valeur de t1[0] : 1000       <== resultat correspondant a la ligne 25
valeur de t2[0][0] : 1000    <== resultat correspondant a la ligne 26
-->
```

• la variable `n` (déclaration ligne 3) est de type `int` qui est un **type primitif**.
Cette déclaration entraine la réservation en mémoire de l'espace nécessaire au codage d'un `int`, c'est-à-dire 4 octets. L'identificateur `n` est interprété dans des expressions arithmétiques comme la grandeur (valeur) de type `int` stockée en mémoire à cet emplacement et comme l'adresse de cet emplacement lorsqu'il est utilisé en partie gauche d'une affectation ;

• la variable `t1` est de type tableau d'`int`. Il s'agit d'un exemple de ce qu'on appelle un **type référence** (un autre exemple de type référence déjà rencontré est le type `String`).
La déclaration de cette variable (ligne 5) entraine la réservation en mémoire de la place nécessaire au codage de la référence d'un tableau d'`int` : il n'y a pas d'allocation effective d'un tableau d'`int` mais simplement la place nécessaire pour mémoriser une adresse.
La variable `t1` n'est pas initialisée (pas plus que ne l'est une variable de type primitif lors de sa déclaration). Une demande d'accès à sa valeur (par exemple pour l'imprimer) provoquerait une erreur de compilation (ligne 6 si elle n'était pas en commentaire).
On peut lui affecter une référence de tableau d'`int`, c'est-à-dire l'adresse en mémoire d'un tableau d'`int`).

Une telle référence peut être :

a) celle d'un tableau d'`int` fraichement alloué en mémoire avec l'opérateur `new` (ligne 7) ;
b) la valeur d'une variable de même type (i.e. `int[]`) déjà affectée (ligne 16). On peut aussi lui affecter la valeur `null` : la variable est initialisée mais ne fait référence à aucun objet.

On peut, pour cela, procéder de différentes manières :

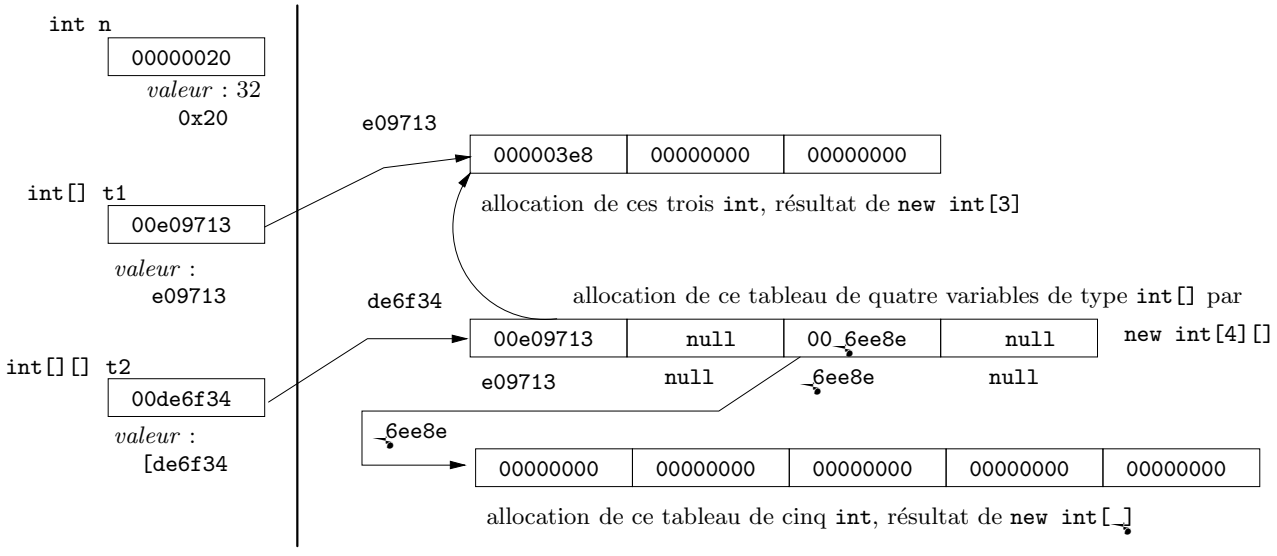
a) faire une demande d'allocation globale pour un tableau rectangulaire avec l'opérateur `new` et affecter le résultat de cette opération à `t2`. Cela donnerait par exemple :

```
t2 = new int[4][3];
```

De cette manière, 12 (4 paquets de 3) entiers sont alloués et initialisés à 0.
Ces différents éléments de type `int` sont désignés par les expressions de la forme `t[`i`][`j`]` avec $0 \leq i < 4$ et $0 \leq j < 3$;

b) procéder par étapes comme on l'a fait sur l'exemple donné (lignes 12, 16 et18) :
- dans un premier temps, allouer un tableau de références de tableau d'`int` de la bonne taille. C'est exactement ce que réalise la ligne 12 : `t2 = new int[4][]`.
  On demande ici l'allocation d'un tableau de pouvant contenir 4 références sur des tableaux d'`int`. Le résultat affiché pour la ligne 13 donne la valeur de `t2` : `[[I@de6f34`.
  Cette valeur indique qu'il s'agit d'une référence de tableau dont les éléments sont des tableaux d'`int` (`[[`) située en mémoire à l'adresse `de6f34`.
  L'affichage provoqué par l'exécution de la ligne 14 montre que les éléments de ce tableau (désignés par `t2[0]`, `t2[1]`, `t2[2]` et `t2[3]`), ont été automatiquement initialisés à une valeur particulière des variables de type référence, la valeur `null`.
  Une variable possédant cette valeur est initialisée mais ne fait référence à rien. La seule chose ayant un sens qu'on puisse avec cette valeur est la tester :
  ```
  if(t2 == null) ......
  ```
  Une variable ayant cette valeur ne fait référence à aucun tableau ;
- dans un second temps, donner une valeur à ces variables selon les besoins : cela peu se faire soit par affectation de la valeur d'une variable de type correct (c'est-à-dire référence sur tableau d'`int`), soit en allouant un tableau d'entiers.
  On a donné une valeur à `t2[0]` de la première manière et pour donner une valeur à `t2[2]` on a utilisé la seconde.
  En procédant de cette manière, il est possible de définir des tableaux non rectangulaires (ici `t2[0]` contient 3 éléments alors que `t2[2]` en contient 5).

La suite de l'exemple montre que `t1` et `t2[0]` font référence au même tableau : la modification de l'élément `t1[0]` en ligne 24 est visible à l'affichage de la valeur de `t2[0][0]`.
Le schéma suivant décrit l'état de la mémoire après exécution de la ligne 18 :



1.2 Les références d'objets quelconques en Java

1.2.1 Des exemples

Considérons la classe `Point` définie de la manière suivante :

```
--> cat Point.java
class Point {
  char nom; double x, y;
  Point(double x, double y, char c){  // constructeur de la classe Point
     this.x = x; // initialisation du membre x de l'objet crée avec la valeur du parametre x
     this.y = y; // initialisation du membre y de l'objet crée avec la valeur du parametre y
     this.nom = c; // initialisation du membre nom de l'objet crée avec la valeur de c
  }
}
```

et un programme l'utilisant :

```
--> cat UsePoint.java
class UsePoint {
  public static void main(String[] args) {
    Point p = new Point(4, -.2, 'A');
    System.out.println(p);
    System.out.println(p.x);   // acces au membre x de l'objet dont p contient la reference
    System.out.println(p.y);   // acces au membre y de l'objet dont p contient la reference
    System.out.println(p.nom); // acces au membre nom de l'objet dont p contient la reference
  }
}
```

Son exécution produit :

```
-->  java UsePoint
Point@10d448
4.0
-.2
A
-->
```

La valeur affichée pour la variable p (`Point@10d448`) indique qu'elle référence un objet de la classe `Point` en mémoire à l'adresse `10d448`.
Il est évidemment possible de définir des tableaux de `Point` et des variables les référençant comme dans ce qui suit :

```
--> cat TabPoint.java
class TabPoint {
    public static void main(String[] args) {
       Point[] t = new Point[4];
       System.out.println(t);
       for(int i = 0; i < t.length; i++)
           System.out.print(t[i] + " ");
       System.out.println(); }
}
--> java TabPoint
[LPoint;@10d448
null null null null
-->
```

1.2.2 Ce qu'il est possible de faire avec les variables de type références

Relativement peu de choses sont possibles sur les variables de types références de java qui n'ont pour seul objectif que de permettre un accès indirect à des objets.

Les seules opérations tendent donc à ce but :
a) affectation d'une nouvelle valeur correspondant à une référence sur un objet de type compatible ou de `null` ;
b) accès à un élément (indice entre crochets pour un tableau) ou un membre via l'opérateur `.` (par exemple `length` pour un tableau ou `nom` pour la classe `Point` que nous avons définie) ;
c) invocation d'une méthode de la classe via l'opérateur « `.` ».

1.3 Adresses, pointeurs, indirections, références

1.3.1 Adresses et pointeurs

Dans certains langages, par exemple Pascal, C ou C++, les choses sont un peu différentes. On peut manipuler effectivement les adresses et en particulier obtenir l'adresse d'un objet en mémoire, ce qu'on ne peut pas faire en Java.
Ainsi en langage C :

```
--> cat point.c
// definition d'un type de nom point
typedef struct {double x, y; char nom;} point;
main() {
  point p;  // declare et alloue une variable de type point
  p.x = 3;  // affecte 3 au champ x de p
  p.y = 4;
  p.nom = 'B';
  printf("%p\n", &p); // imprime l'adresse de la variable p
}
--> gcc point.c -o point  <== compilation
--> ./point               <== demande d'execution du binaire obtenu
0xbffffd10                <== adresse en memoire (sous forme hexa) de la variable p
-->
```

On a défini ici un type `point` en tout point semblable à la classe `Point` que nous avons définie en Java, du moins en ce qui concerne les informations que renferme ce qui est déclaré de ce type.
Puis après avoir défini une variable de ce type, on en a affiché l'adresse.
De fait, dans ce langage, la variable `p` désigne directement le point et non sa référence comme c'est le cas en Java pour un objet de la classe `Point`.
Par contre il est effectivement possible d'en obtenir l'adresse au travers de l'opérateur `&`.
En C et C++, il est par ailleurs possible de définir des **variables de type pointeur** pour mémoriser les adresses de variables en mémoire. Nous verrons que cela est essentiel du point de vue du passage des paramètres en C.

Inversement, disposant d'un pointeur `p` sur une entité de type T, il est possible d'accéder à cette entité au moyen d'un opérateur particulier noté `*`.
L'exemple suivant illustre l'ensemble de ces mécanismes du langage C :

```
--> cat pointeur.c
main() {
  int  ptr;  // definit une variable de type pointeur sur entier
  int i = 4;
  ptr = &i;  // la variable p contient l'adresse de i
  printf("%d\n", i);
  printf("%d\n",  ptr);  //  ptr est (physiquement) ce qui est pointé par pp
}
--> ./pointeur
4
4
-->
```

En Java il n'est pas possible, par exemple, d'obtenir une référence sur une variable de type `int` déclarée comme telle.
En résumé, on peut dire qu'en Java les entités en mémoire sont désignées et accédées de manière exclusive :

- par un identificateur pour celles d'un type primitif ;
- indirectement via une constante d'un type référence (typiquement la valeur d'un `new`) ou une variable d'un type référence contenant leur adresse pour les autres.

Remarque : il existe par ailleurs des types encapsulant les types primitifs (*wrappers*). Ainsi, par exemple, la classe `Integer` permet la définition de valeurs entières comme objets.

1.3.2 Les références en C++

En C++, il est par ailleurs possible de définir, en plus des pointeurs, des références.
Une référence constitue un synonyme d'un identificateur.
Elle permet de manipuler une variable sous un autre nom que celui avec lequel elle a été déclarée. Dans l'exemple ci-après, une référence de nom `ref` est associée à une variable `i` du programme :

```
--> cat reference.C
#include <iostream.h>
main() {
int i = 30;
int &ref = i;    // ref est une reference synonyme de i
i = ref + i;     // la valeur de i est doublee (i et ref sont des synonymes)
cout << i;       // affichage de la valeur referencee par i
cout << endl;    // affichage de la valeur de la variable n
cout << ref;     // affichage de la valeur referencee par ref
cout << endl;    // affichage d'un caractere de fin de ligne
}
--> g++ reference.C -o reference   // commande de compilation d'un source C++
--> ./reference   // execution du binaire obtenu
60
60
-->
```

2

Modes de passage des paramètres de fonctions

2.1 Principes généraux

2.1.1 Paramètres, signature, polymorphisme

À la définition d'une fonction (méthode, sous-programme, procédure, ...), l'en-tête de cette définition contient en particulier une liste de paramètres de cette fonction : on parle de **paramètres formels**.
Lors de l'appel (ou invocation) d'une fonction, cet appel est réalisé en spécifiant des **paramètres d'appel** (on dit aussi **paramètres actuels** ou **paramètres effectifs**).
Dans un certain nombre de langages (C par exemple), une fonction de nom donné ne peut avoir qu'une seule définition. Dans d'autres langages (Java ou C++ par exemple), une fonction peut avoir plusieurs définitions (on parle de **surcharge** qui est un cas particulier du concept de **polymorphisme**). Chacune de ces définitions correspond à un contexte d'utilisation : ces différentes définitions sont différentiables par le type et/ou le nombre des paramètres utilisés à l'appel qui permettent de choisir la définition à utiliser. On nomme **signature** l'ensemble constitué du nom d'une fonction et de la liste des types de ses paramètres. Ce qui est imposé est qu'il n'existe qu'une seule définition de la fonction pour une signature donnée afin que lors d'un appel donné, une seule définition soit applicable.

2.1.2 Correction d'un appel

Pour qu'un appel soit correct, il faut que les paramètres utilisés lors de l'appel correspondent à la signature (ou l'une des signatures) de cette fonction. Cela signifie que :

a) le nombre des paramètres effectifs est égal à celui des paramètres formels (sauf si le langage, comme le langage C par exemple, autorise des paramètres optionnels) d'une signature ;

b) chaque paramètre effectif est d'un type compatible avec celui du paramètre formel correspondant, c'est-à-dire en même position dans la définition de la fonction).

Le compilateur Java impose le respect total de ces règles. Toute violation est détectée et produit une erreur fatale interdisant la compilation et la production d'un code exécutable par la machine Java.
Dans l'exemple suivant, le premier paramètre `9.81` est une constante de type `double`. Or la seule définition de la fonction `puis` attend un premier paramètre de type `float`. Le type `double` n'étant pas compatible avec ce type, une erreur est signalée (des écritures correctes de cet appel sont données en commentaires :

```
--> cat Param1.java
class Param1 {
  static float puis(float x, int n) {
    float y = 1;
    for(int i = 0; i < n; i++)
        y  =  x;   // y = y   x
    return y;
    }
```

```
  public static void main(String[] args){
    System.out.println(puis(9.81, 3));
    // System.out.println(puis(9.81f, 3));
    // System.out.println(puis((float)9.81, 3));
   }
}
--> javac Param1.java
Param1.java:8: puis(float,int) in Param1 cannot be applied to (double,int)
      System.out.println(puis(9.81, 3));
                         ^
1 error
```

Remarque : les compilateurs C sont de manière générales plus laxistes. Tout au plus, provoquent-ils à la demande des avertissements (*warning*) lors d'appels folkloriques mais produisent du code dont l'exécution peut conduitre à des résultats inattendus.

2.2 Modes de passage des paramètres : la valeur et la référence

2.2.1 Introduction

Il s'agit de définir la manière dont l'information est transmise entre le module appelant et le module appelé, cest-à-dire la manière dont un paramètre effectif est substitué au paramètre formel correspondant lors d'un appel.

Il existe deux modes fondamentaux de passage (ou transmission) de paramètres (nous nous limitons volontairement à ces deux-là qui peuvent être illustrés au travers d'exemples Java, C ou C++).

a) le passage par valeur : avec ce mode de passage de paramètre, le paramètre formel est vu comme une variable locale de la fonction (c'est-à-dire créée à l'entrée dans la fonction et supprimée à la sortie de la fonction et donc uniquement visible et accessible durant l'exécution du corps de la fonction). À l'appel, le paramètre reçoit comme valeur celle du paramètre effectif correspondant. Le paramètre effectif ne sert donc, au travers de sa valeur, qu'à permettre d'initialiser le paramètre correspondant de la fonction. Il n'y a aucun lien, du point de vue physique en mémoire, entre le paramètre effectif et le paramètre formel lors de l'exécution de la fonction. Un changement de valeur du paramètre formel sera perdu au retour de la fonction : la variable correspondant au paramètre effectif ne sera pas modifiée. C'est ce mode de transmission des paramètres qui est utilisé en Java et en C (c'est le seul). Mais cependant, compte tenu de la nature des variables en Java (dont les valeurs sont soit des valeurs d'un type primitif, soit des références sur un type non primitif) ou de la possibilité de définir des pointeurs en C, nous verrons, plus loin, qu'il est possible de modifier de manière indirecte des entités dont une référence (ou un pointeur) est transmise en paramètre ;

b) le passage par référence (ou par adresse) : avec ce mode de transmission de paramètres, le paramètre effectif est une référence dont le paramètre formel devient un synonyme lors de l'appel. Le paramètre formel et le paramètre effectif sont alors deux références synonymes et désignent donc un même espace en mémoire. Toute modification de l'entité référencé par le paramètre formel de la fonction modifie donc son synonyme référencé par le paramètre effectif.

Dans les langages PL1 et Fortran, c'est ce mode de passage des paramètres qui est utilisé. En C++, Pascal, ou ADA par exemple, le mode de passage par valeur est le mode par défaut. Cependant, il est possible, lors de l'écriture d'une fonction, de demander individuellement pour chacun des paramètres qu'il soit transmis par référence plutôt que par valeur.

2.2.2 Exemples avec un paramètre de type primitif

2.2.2.1 Java : une fonction ayant un paramètre de type primitif `int`

```
--> cat Param2.java
class Param2 {
  // definition d'une fonction avec un parametre de type int
  static void inc(int a) {
     System.out.println("entree dans inc : a = " + a);
     a = a + 1;
     System.out.println("avant retour : a = " + a);
     return; }
  public static void main(String[] args) {
     int n = ↵;
     // appel de inc : ce qui importe c'est la valeur du parametre, pas sa localisation
     inc(n);
     System.out.println("retour de l'appel : a = " + n); }
  }
--> java Param2
entree dans inc : a = ↵     <== parametre formel initialise a la valeur du parametre effectif
avant retour : a = 6        <== le parametre formel est incremente
retour de l'appel : a = ↵   <== a et n ne correspondant pas a la même entite physique
-->
```

Cet exemple illustre ce que nous avons dit à propos du mode de transmission par valeur :

- le paramètre formel `i` de la méthode `inc` correspond à une variable locale créée à l'appel de la fonction et détruite à son retour ;
- la variable locale associée au paramètre formel est initialisée à la valeur du paramètre effectif : celui-ci est la valeur de la variable `n` de la méthode `main`, c'est-à-dire `5` ;
- le paramètre formel est associé à une nouvelle zone mémoire : les modifications du contenu de cette zone ne sont pas visibles de l'extérieur de la fonction. La variable `n` de la méthode `main` n'est pas touchée par la modification de la valeur de la variable `i` de la méthode `inc`.

Avec ce mode de transmisson il n'est pas possible, par exemple, d'écrire une méthode ou procédure (c'est-à-dire une fonction de type `void` en Java) dont l'effet soit d'échanger le contenu de deux variables (de type `int` par exemple) données en paramètres.

2.2.2.2 C++ : un paramètre de type primitif `int` transmis par valeur ou par référence

Cet exemple, écrit en C++ illustre la possibilité de choisir lors de la définition d'une fonction le mode de transmission d'un paramètre.
On y a défini deux fonctions qui ne diffèrent que par leur en-tête :

- celle de la fonction `inc1`

  ```
  void inc1(int i)
  ```

  spécifie que pour le paramètre `i`, le mode de passage utilisé est la valeur ;
  Son invocation conduit à des résultats semblables à ceux obtenus avec le programme Java `Param2` précédent ;
- celle de la fonction `inc2`

  ```
  void inc1(int &i)
  ```

  spécifie que pour le paramètre `i`, le mode de passage utilisé est la référence.

  Son invocation conduit à des résultats différents : les noms utilisés pour le paramètre formel et le paramètre effectif sont associés au même emplacement en mémoire. Les modifications effectuées dans la fonction au travers du nom du paramètre formel sont donc visibles au retour de la fonction au travers de la variable utilisée comme paramètre effectif.

```
--> cat param2.C
#include <stream.h>
void inc1(int i) { // i est transmis par valeur
  i = i + 1 ; return; }
void inc2(int &i) { // i est transmis par reference
  i = i + 1 ; return; }
main() {
  int n = 5;
  inc1(n); //
  cout << n; // impression de la valeur de n
  cout << endl; // impression d'un caractere de fin de ligne
  inc2(n);
  cout << n; // impression de la valeur de n
  cout << endl; // impression d'un caractere de fin de ligne
}
--> g++ param2.C -o param2   <== compilation du fichier (g++ : compilateur C++)
--> ./param2
5    // appel par valeur : parametre d'appel non modifie
6    // appel par reference : parametre d'appel modifie
-->
```

Avec le passage de paramètres par références, l'écriture d'une procédure réalisant l'échange du contenu de deux variables est possible :

```
--> cat echanger.C
#include <stream.h>
/  fonction echangeant le contenu de deux variables entieres  /
static void echanger(int &x, int &y){
  int z = x;
  x = y;
  y = z;
  }
/  programme principal  /
main() {
  int a = 5;
      b = 9;
  echanger(a, b);
  cout << "a="; cout << a;
  cout << "    ";
  cout << "b="; cout << b;
  cout << endl;
}
--> ./echanger
a=9    b=5      <== les valeurs des variables a et b ont ete echangees
-->
```

2.2.3 Java : les paramètres de type référence

2.2.3.1 Le cas général

Comme nous l'avons dit, en Java le passage des paramètres est toujours réalisé par valeur. Si les choses sont simples en ce qui concerne le passage de paramètres de types primitifs, elles méritent qu'on examine ce qui se passe en ce qui concerne les paramètres de types référence. Reprenons la classe `Point` utilisée dans la première partie du cours :

```
class Point {
  char nom;
  double x, y;
  Point(double x, double y, char c){  // constructeur de la classe Point
     this.x = x;   //initialisation du membre x de l'objet
     this.y = y;   //initialisation du membre y de l'objet
     this.nom = c; //initialisation du membre nom de l'objet
  }
}
```

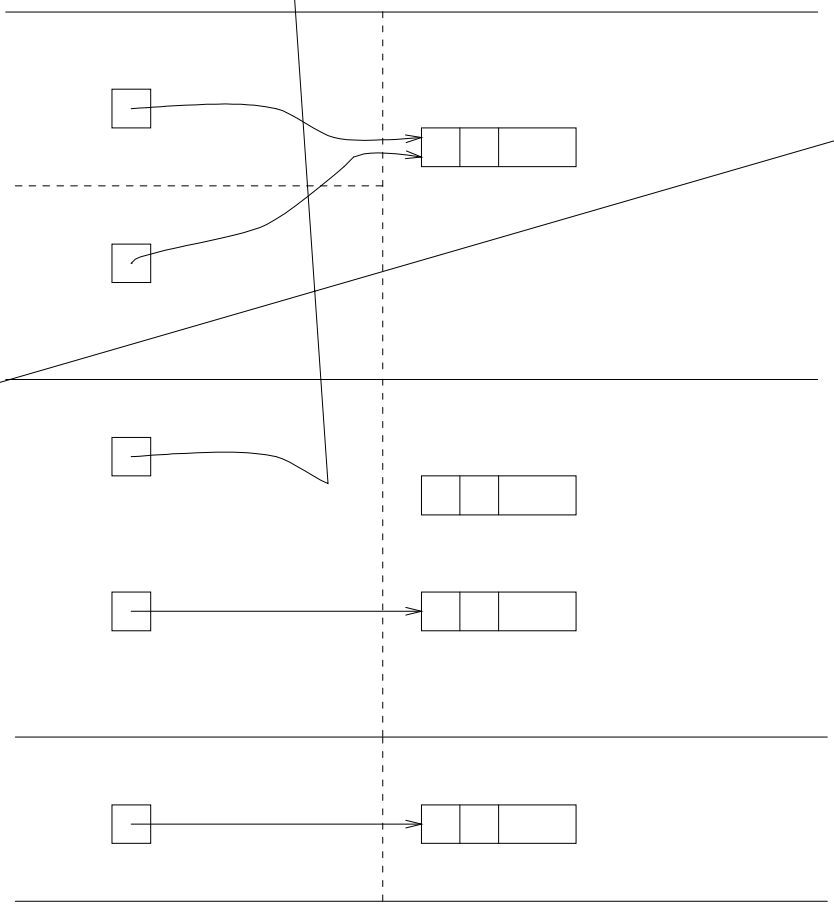
Considérons l'application suivante :

```
--> cat AppliPoint.java
class AppliPoint {
  static void fonc(Point point) {
    System.out.println("fonc : " + point + " --> " + point.x + " " + point.y);
    point.x ++; point.y ++;
    System.out.println("fonc : " + point + " --> " + point.x + " " + point.y);
    point = new Point(-1, 6, 'B');
    System.out.println("fonc : " + point + " --> " + point.x + " " + point.y);
  }
  public static void  main(String[] args) {
    Point p = new Point(1, 2, 'A');
    System.out.println("main : " + p + " --> " + p.x + " " + p.y);
    fonc(p);
    System.out.println("main : " + p + " --> " + p.x + " " + p.y);
  }
}
```

Intéressons nous à son exécution :

```
--> java AppliPoint
main : Point@8a992f --> 1.0 2.0
fonc : Point@8a992f --> 1.0 2.0
fonc : Point@8a992f --> 2.0 3.0
fonc : Point@9189e1 --> -1.0 6.0
main : Point@8a992f --> 2.0 3.0
-->
```

La figure suivante illustre ce qui se passe en mémoire lors de l'exécution de ce programme, à l'entrée dans la fonction `fonc`, juste avant le retour de la fonction et juste après le retour de la fonction.

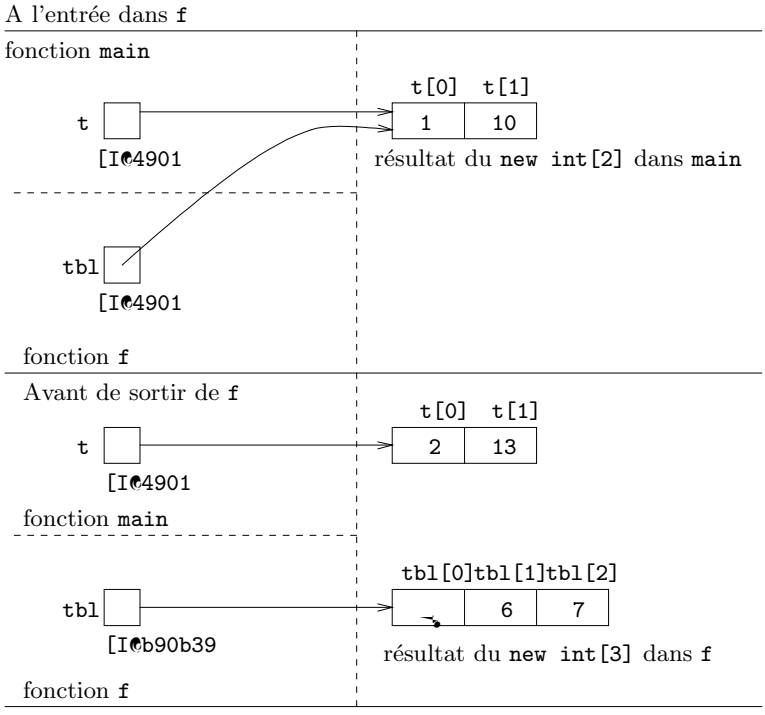


2.2.3.2 Le cas particulier des tableaux

Lorsque le paramètre est une référence de tableau, la modification de certains éléments du tableau référencé est possible dans une fonction recevant cette référence en paramètre comme l'illustre l'exemple suivant :

```
--> cat ParamTableau.java
class ParamTableau {
  static void f(int[] tbl){
    System.out.print("Entree dans f : ");
    System.out.println(tbl + " -> " + tbl[0] + " " + tbl[1]);
    tbl[0] = tbl[0] + 1; tbl[1] = tbl[1] + 3;
    tbl = new int[3];
    for(int i = 0; i < tbl.length; i++) tbl[i] = i + 7;
    System.out.print("Avant sortie de f : + tbl + " -> ");
    for(int i = 0; i < tbl.length; i++) System.out.print(tbl[i] + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    int[] t = {1, 10};
    System.out.print("Avant appel de f : ");
    System.out.println(t + " -> " + t[0] + " " + t[1]);
    f(t);
    System.out.print("Retour appel de f : ");
    System.out.println(t + " -> " + t[0] + " " + t[1]);
  }
}
--> java ParamTableau
Avant appel de f : [I@4901 -> 1 10
Entree dans f : [I@4901 -> 1 10
Avant sortie de f : [I@b90b39 -> 0 6 7
Retour appel de f : [I@4901 -> 2 13
-->
```

La figure suivante représente l'état de la mémoire lorsqu'on entre dans la fonction `f` et qu'on en sort et à quoi sont associés les différents identificateurs utilisés.



2.2.3.3 Histoire de recopie de tableaux

Pour terminer avec ce sujet, nous espérons que le lecteur aura compris que dans le code suivant, la variable `tab` de la méthode `main` fait toujours référence au même tableau au retour de l'appel de la fonction et que le contenu de ce tableau est inchangé.

```
--> cat Tableau.java
class Tableau {
   public static void f(int[] t){
      int[] t1 = new int[t.length];
      for(int i = 0; i < t.length; i++)
          t1[i] = t[i] + 1;
      t = t1;
      }
   public static void main(String[]args) {
      int[] tab = new int[4];
      for(int i = 0; i < tab.length; i++)
          tab[i] = i + 1;
      System.out.println("Avant l'appel");
      System.out.println("       valeur de tab : " + tab);
      System.out.print("       contenu de tab : ");
      for(int i = 0; i < tab.length; i++)
          System.out.print(tab[i] + " ");
      f(tab);
      System.out.println("\nApres l'appel");
      System.out.println("       valeur de tab : " + tab);
      System.out.print("       contenu de tab : ");
      for(int i = 0; i < tab.length; i++)
           System.out.print(tab[i] + " ");
      System.out.println();
      }
}
--> java Tableau
Avant l'appel
       valeur de tab : [I@f72617
       contenu de tab : 1 2 3 4
Apres l'appel
       valeur de tab : [I@f72617
       contenu de tab : 1 2 3 4
-->
```

2.2.3.4 En guise de conclusion

Pour résumer les choses, on peut dire que le paramètre transmis est ici la variable contenant la référence : la référence est donc protégée par le mode de transmission par valeur mais pas l'objet référencé (puisqu'il n'est pas recopié).

2.3 Compléments et comparaison des deux modes

2.3.1 Le passage par référence et les constantes

Le passage par référence suppose que lors de l'appel le paramètre effectif possède une adresse. Que se passe-t-il lorsque le mode de passage d'un paramètre est le passage par référence et que, lors d'un appel, le paramètre effectif correspondant est une constante et ne possède donc pas d'adresse en mémoire ?

2.3.1.1 Ce qui se passe en C++

Dans le cas du langage C++, une erreur est signalée dès la phase de compilation :

```
--> cat paramErreur.C
#include <stream.h>
void f(int &x){ x++; }
main(){ f(_); }
--> g++ paramErreur.C
paramErreur.C: In function ‘int main()’:
paramErreur.C:3: error: could not convert ‘_’ to ‘int&’
paramErreur.C:2: error: in passing argument 1 of ‘void f(int&)’
-->
```

2.3.1.2 Ce qui se passe en Fortran

Une autre approche est adoptée dans le cas du langage Fortran.
Dans ce langage où tous les paramètres sont transmis (a priori) par référence, lorsqu'une constante ou une expression est utilisée comme paramètre effectif dans un appel, une variable temporaire contenant la valeur de la constante ou de l'expression est transmise par référence. Ce mécanisme s'apparente à un mode de transmission par valeur implicite.
Cette approche rend ainsi possible, en Fortran, la « simulation » de l'appel par valeur lorsqu'une fonction est appelée avec une variable comme paramètre. Il suffit de remplacer le nom de la variable que l'on souhaite transmettre par valeur (typiquement si on veut avoir la certitude qu'elle ne pourra être modifiée pendant l'exécution d'un sous-programme ou d'une fonction) par une expression ayant comme valeur la valeur de cette variable.
Ainsi, plutôt que de faire un appel (c'est la syntaxe d'un appel dans ce langage)

```
CALL   SP(X)
```

on pourra faire un appel

```
CALL   SP(X 1)
```

2.3.2 Avantages/inconvénients. Quel mode choisir si on a le choix ?

Il doit être bien clair que le mode de transmission par référence économise la recopie en mémoire des données transmises.
Si le paramètre est une grosse structure ou un gros tableau, le gain en mémoire est évident (surtout si des appels en cascade sont réalisés).
A contrario, avec ce mode de transmission, on n'est pas, a priori, à l'abri d'effets de bord modifiant des informations du programme appelant : c'est pourquoi on utilise dans ce langage des types `const`.
La possibilité de définir des variables externes ou globales (c'est-à-dire partagées par différentes fonctions sans qu'il soit nécessaire de les transmettre en paramètre) associé au mode de transmission par valeur est une autre solution proposée par différents langages. Nous aborderons ce point dans un prochain chapitre.

2.4 Ce qui se passe en C

2.4.1 Le cas général

En C, tous les passages de paramètres sont réalisés par valeur, comme en Java. Par ailleurs, toutes les variables, à l'exception des tableaux, sont interprétées comme des valeurs, contrairement à ce qui se passe en Java.
Ainsi, une traduction brutale en C (sans se poser de question particulière) de l'application Java `AppliPoint` décrite précédemment est :

```
--> cat appliPoint.c
typedef struct {
   int a, b; char nom;} point;
void fonc(point pnt) {
   printf("fonc : %p --> %d %d\n", &pnt,  pnt.a, pnt.b);
   pnt.a ++;
   pnt.b ++;
   printf("fonc : %p --> %d %d\n", &pnt,  pnt.a, pnt.b);
}
void  main() {
   point p;
   printf("main : %p --> %d %d\n", &p, p.a, p.b);
   fonc(p);
   printf("main : %p --> %d %d\n", &p, p.a, p.b);
}
--> ./appliPoint
main : 0xbffffd20 --> 0 0   <== adresse de p
fonc : 0xbffffcf8 --> 0 0   <== adresse de pnt differente de celle de p
fonc : 0xbffffcf8 --> 1 1   <== modification de champs
main : 0xbffffd20 --> 0 0   <== champs de la structure initiale inchanges
-->
```

Son exécution fait apparaître que la valeur d'une variable sur un type T, est une grandeur de ce type et non une référence (en ce qui concerne les tableaux, nous verrons que les choses sont différentes).
Pour obtenir l'adresse en mémoire, il faut en faire une demande explicite.
On observe aussi que le paramètre est transmis par valeur : la modification d'un champ lors de l'exécution n'est pas visible au retour de la fonction sur la grandeur d'origine.

Comment, dans ces circonstances, réaliser un passage de paramètre par référence afin d'obtenir un résultat semblable à celui obtenu avec la version Java ?
La solution repose sur les mécanismes offerts par le langage C, à savoir le déréférencement lors de l'appel de la fonction et l'indirection dans le corps de la fonction elle-même.
C'est au programmeur de prendre les dispositions nécessaires, rien n'est implicite :

- dans l'écriture de la fonction : spécifier que le paramètre formel est un pointeur sur le type souhaité et manipuler la valeur pointée par indirection ;
- à l'appel de la fonction : transmettre l'adresse de la variable.

Cela conduit à la traduction suivante du programme dont les résultats sont similaires à ceux de l'application Java :

```
--> cat appliPoint2.c
typedef struct{ int a, b; char nom;} point;
void fonc(point  pnt) {
   printf("fonc : %p --> %d %d\n", pnt,  ( pnt).a, ( pnt).b);
   ( pnt).a ++; ( pnt).b ++;
   /  pnt->a est une ecriture equivalente ( pnt).a ne necessitant pas de parenthese  /
   printf("fonc : %p --> %d %d\n", pnt,  pnt->a, pnt->b);
}
void  main() {
   point p;
   printf("main : %p --> %d %d\n", &p, p.a, p.b);
   fonc(&p);
   printf("main : %p --> %d %d\n", &p, p.a, p.b);
}
--> ./appliPoint2
main : 0xbffffd20 --> 0 0
fonc : 0xbffffd20 --> 0 0
fonc : 0xbffffd20 --> 1 1
main : 0xbffffd20 --> 1 1
-->
```

2.4.2 Un cas particulier notable en C : les tableaux

Nous avons incidemment dit que les tableaux avaient un statut particulier dans le langage C. En effet, un tableau de valeurs de type T n'est rien d'autre qu'un pointeur sur le type T. Ainsi, en plus de pouvoir réaliser des opérations particulières constituant l'arithmétique des pointeurs qui dépassent le cadre de cette présentation, cela signifie que le traitement des tableaux en tant que paramètres de fonctions, s'apparente à celui des tableaux en Java.
Traduisons en C l'application `ParamTableau` écrite plus haut en Java :

```
--> cat paramTableau.c
void f(int tbl[]){  // en C : en tête equivalente a void f(int  tbl)
  printf("Entree dans f : ");
  printf("%p -> %d %d\n", tbl, tbl[0], tbl[1]);
  tbl[0] = tbl[0] + 1; tbl[1] = tbl[1] + 3;
}
main() {
  int t[] = {1, 10};
  printf("Avant appel de f : ");
  printf("%p -> %d %d\n", t, t[0], t[1]);
  f(t);
  printf("Retour appel de f : ");
  printf("%p -> %d %d\n", t, t[0], t[1]);
}
--> ./paramTableau
Avant appel de f : 0xbffffd00 -> 1 10
Entree dans f : 0xbffffd00 -> 1 10
Retour appel de f : 0xbffffd00 -> 2 13
-->
```

3

Implémentation des appels de fonctions au moyen d'une pile

Le concept de fonction et la possibilité de réaliser des appels de fonctions conduisent à la rupture de la séquentialité de l'exécution des instructions pronée par le modèle de Von Neumann.
Nous nous proposons d'étudier ici comment de tels appels sont effectivement implémentés.

3.1 Le concept de pile (stack)

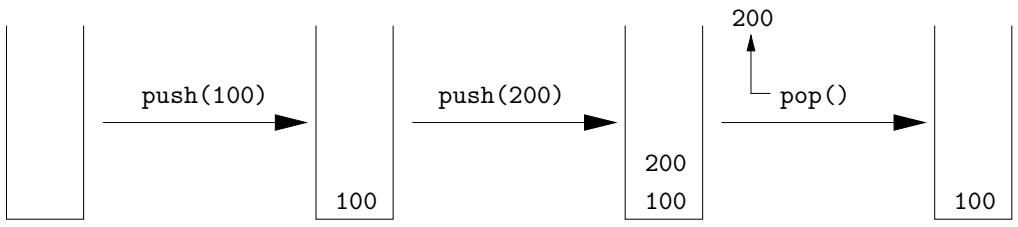
3.1.1 Définitions

Cette structure de données est centrale dans l'implantation des appels de fonctions.
Les propriétés d'une telle structure sont celles d'une pile d'assiettes sur un chauffe-assiettes : une assiette est toujours déposée au-dessus des autres et quand on en prend une, on prend celle du dessus. C'est pourquoi on leur donne couramment le nom de *LIFO* pour *Last In First Out*.

De manière plus formelle, disons qu'une pile est une structure de données pour laquelle les opérations suivantes ont un sens :

- empiler une information (*push*) ;
- extraire une information ou dépiler (*pop*) ;
- tester si elle vide.

La figure suivante illustre l'utilisation des deux fonctions *push* et *pop* sur une pile d'entiers :



une pile vide

Dans l'écriture de pseudo-code, nous supposerons que le type `Pile` existe, peu importe la manière dont une telle pile est effectivement implantée : ce qui nous importera, c'est la possibilité de déclarer des variables de type `Pile` et de disposer des opérations de manipulation. Nous présenterons et utiliserons dun peu plus loin, dans des exemples de vrais programmes, la classe `Stack` disponible dans un *package* Java.

Ainsi, si p est une variable de type `Pile`,

- un appel *`push(p, valeur)`* (dans le style impératif) ou *`p.push(valeur)`* (dans le style objet) empilera sur la pile la valeur. Ce qui est important, c'est que la pile soit modifiée par un tel appel. Cela signifie que dans un langage tel que C++, le paramètre p sera naturellement transmis par référence ;

- un appel *pop(p)* ou *p.pop()* renverra la valeur au sommet de la pile et la dépilera effectivement. La pile étant modifiée par un tel appel, le paramètre p sera transmis par référence si nécessaire ;
- un appel *vide(p)* ou *p.vide()* renverra le booléen *Vrai* ou *Faux* selon que la pile p est vide ou non.

La nature des informations empilées dépend bien évidemment de l'application l'utilisant. Sur la figure que nous avons donnée, on ne fait qu'empiler et dépiler des entiers.

Dans l'application qui nous intéresse, à savoir l'implantation du mécanisme d'appel de fonction, on empilera et dépilera des enregistrements composés d'un ensemble d'informations de natures différentes.

Cette multiplicité d'utilisation des piles dans des contextes divers suffit à justifier la possibilité de définir un type de pile générique dans lequel la nature des informations mémorisées soit paramétrable : nous verrons que c'est le cas de la classe `Stack` de Java dans ses versions récentes.

3.1.2 Exemple d'utilisation

Avant de décrire l'implantation du mécanisme d'appels de fonctions au moyen d'une pile, illustrons son utilisation sur un exemple simple : l'évaluation d'une expression arithmétique écrite en forme polonaise suffixe (ou postfixée ou inverse).

Rappelons que dans cette écriture des expressions, les opérandes précèdent l'opérateur.

Ainsi `x + y` s'écrit `x y +`

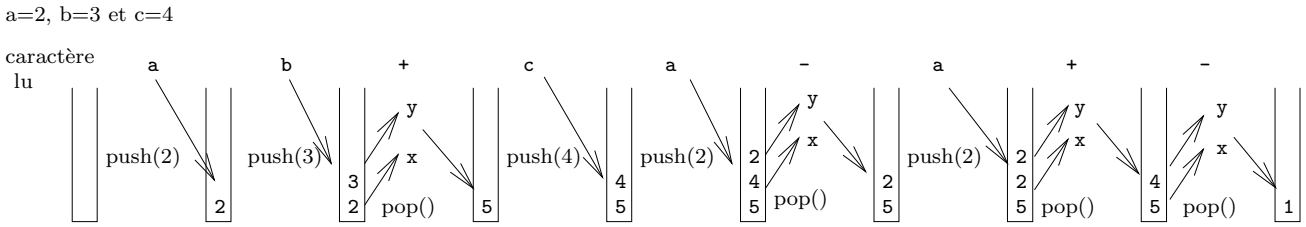
Un premier intérêt de cette écriture est qu'elle ne nécessite pas l'utilisation de parenthèses pour distinguer des expressions telles que `a + b * c` et `(a + b) * c`.

La première s'écrit `a b c * +` et la seconde `a b + c *`

Un second intérêt de cette écriture réside dans la possibilité d'évaluer une expression par lecture uniforme (sans jamais revenir en arrière) de gauche à droite en utilisant une pile.

Par exemple, supposons que les variables a, b et c aient respectivement 2, 3 et 4 comme valeurs et soit l'expression `a b + c a - a + -` dont on souhaite calculer la valeur.

La figure suivante illustre le calcul de sa valeur en utilisant une pile pour mémoriser les résultats des calculs intermédiaires :



Pour résumer :

- si le caractère lu correspond à un opérateur binaire, on extrait de la pile (par deux `pop` successifs) les deux opérandes (si l'expression est syntaxiquement correcte, cette opération est possible). La première valeur dépilée est l'opérande droite et la seconde, l'opérande gauche. Puis le résultat de l'opération est empilé (`push`) ;
- sinon la valeur de la variable est empilée (`push`) ;
- la valeur au sommet de la pile à la fin de la lecture de l'expression est la valeur de l'expression. Si l'expression est syntaxiquement correcte, il ne doit rester sur la pile qu'un seul élément.

3.2 Implémentation des appels de fonction

3.2.1 Existence d'une pile des appels

Nous allons révéler son existence en y provoquant une erreur : dans l'exemple, on appelle une fonction `inc` dont la définition entraine un appel à une fonction `dec` qui appelle elle-même la fonction `inc`.
Nous reviendrons sur ce type de définition tout à fait autorisé dans le chapitre 5 consacré à la récursion. Mais tel qu'il est écrit, le programme est mal écrit et ne peut se terminer : il provoque un débordement de la pile des appels. Chaque appel provoque en effet (avant qu'il ne soit terminé) un nouvel appel. Il y a donc un empilement continu d'informations sans jamais dépiler.

```
--> cat DebordementPile.java
class DebordementPile {
   static int inc(int n){
       if (n == 0) return 0;
       return (dec(n-1) + 1);
   }
   static int dec(int n){
       if (n == 0) return 0;
       return (inc(n+1) - 1);
   }
   public static void main(String[] args){
       System.out.println(inc(_));
   }
}
--> java DebordementPile
Exception in thread "main" java.lang.StackOverflowError
-->
```

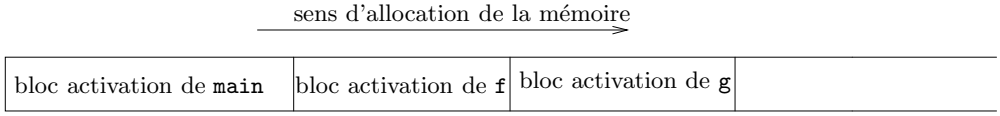
La levée de l'exception `StackOverflowError` matérialise en Java le **débordement de pile**.

3.2.2 Bloc d'activation

Un appel (ou invocation) de fonction entraine l'empilement sur la pile des appels d'un **bloc d'activation** (dans la machine Java on parle de *frame*) qui vise à permettre l'exécution (du corps) de la fonction appelée et d'assurer la relation avec le code appelant.
Précisons les caractéristiques d'un tel bloc d'activation :

a) sa durée de vie : c'est celle de l'exécution de la fonction. Il est créé à l'appel de la fonction et supprimé à la fin (au retour) de l'exécution de la fonction ;

b) compte tenu de la possibilité d'effectuer des appels de fonctions en cascade (avec ou sans récursivité), la structure de pile est la plus adaptée pour gérer les blocs d'activation.
La figure suivante (où la pile est représentée horizontalement avec le sommet à droite) illustre ce mode de gestion des blocs d'activation :



dans la fonction `main`, on a appelé la fonction `f` dans laquelle on a appelé la fonction `g`

c) un bloc d'activation est constitué des objets suivants :
- les paramètres de la fonction initialisés avec la valeur des paramètres effectifs ;
- les variables locales de la fonction ;
- un espace pour stocker la valeur de retour de la fonction ;
- l'adresse de retour (ce qui permettra au retour de la fonction de reprendre l'exécution là où elle a été suspendue).

Nous donnerons en 3.4 un exemple complet de code Java simulant des appels de fonction.

3.3 La classe `Stack`

Nous traçons maintenant les grandes lignes de la classe `Stack` de Java qui possède les caractéristiques de la structure de pile dont nous venons de parler :

- son utilisation suppose l'import de `java.util.Stack`;
- elle constitue un modèle général de pile ;
- une définition telle que :

  ```
  Stack p = new Stack();
  ```

  définit une pile vide destinée à recevoir des références d'objets de types quelconques et la variable `p` en contient la référence. Son utilisation nécessite un usage intensif de l'opération de « coercition » (*cast*) pour réaliser des transtypages vers le bas ;
- dans les dernières versions du langage, il est possible de définir des piles dont on contraint le type de ce qu'on peut y mettre et y lire, ce qui simplifie grandement l'écriture des programmes. Cela suppose évidemment que tout ce qu'on empile est de ce type ou d'un type en dérivant :
  Ainsi, avec la définition :

  ```
  Stack<Integer> p = new Stack<Integer>();
  ```

  on définit une pile vide destinée à recevoir des références d'objets de type `Integer` (enveloppant des entiers) et la variable `p` en contient la référence. Le type des informations qu'on met sur la pile (et qu'on en extrait) est en quelque sorte un paramètre du type de la pile. Attention, on ne peut pas utiliser les types primitifs (d'où l'usage du *wrapper* `Integer`) ;
- l'empilement d'une information référencée par `info` sur une pile référencée par `p` est réalisée par :

  ```
  p.push(info); // info du type utilisé pour déclarer p
  ```
- l'extraction de l'information au somment de la pile est réalisée par :

  ```
  info = p.pop(); // info du type utilisé pour déclarer p
  ```

  La référence est effectivement extraite de la pile.
  Par contre, avec l'appel

  ```
  info = p.peek(); // info du type utilisé pour déclarer p
  ```

  on obtient la référence située au sommet de la pile mais il n'y a pas dépilement.
  Du fait de la nature des variables de références en Java, cette méthode est particulièrement adaptée à ce que nous voulons faire. Un appel à cette primitive permettra de récupérer la référence du bloc courant (celui au sommet de la pile) et de travailler directement sur son contenu ;
- le test de la vacuité de la pile est réalisée par :

  ```
  if(p.empty()) .... // si la pile est vide, ....
  ```

3.4 Un exemple complet

Nous nous proposons ici de traduire un programme un programme faisant des appels de fonctions en un programme séquentiel équivalent.

3.4.1 Le programme d'origine

```
--> cat ExempleFonction.java
class ExempleFonction{
  // recherche du plus grand élément d'un tableau
  static int valMax(int[] tab) {
    // point 1 : entrée dans la fonction valMax
    int i, x = tab[0];
    for(i = 1; i < tab.length; i++)
       if(tab[i] > x) x = tab[i];
    return x;
  }
```

```
  // comptage des occurrences d'un nombre compris entre 0 et n
  // dans un tableau : tous les elements sont supposes compris entre 0 et n
  static int[] repOcc1(int[] tab, int n) {
    // point 2 : entrée dans la fonction repOcc1
    int[] occ = new int[n+1];
    int i;
    for(i = 0; i < tab.length; i++)
       occ[tab[i]]++;
    return occ;
  }
  // comptage des occurrences de tous les nombres trouvés
  // dans un tableau
  static int[] repOcc2(int[] tab) {
    // point 3 : entrée dans la fonction repOcc2
    int n = valMax(tab);
    // point 4 : au retour de l'appel a valMax, affecter la valeur renvoyée a n
    int[] occ = repOcc1(tab, n);
    // point 5 : au retour de l'appel a repOcc1, affecter la valeur renvoyée a occ
    return occ;
  }
  // écriture des valeurs contenues dans un tableau
  // sur une seule ligne et séparées par un espace
  static void ecrireTableau(int[] tab){
    // point 6 : entrée dans la fonction ecrireTableau
    int i;
    for(i = 0; i < tab.length; i++)
       System.out.print(tab[i] + " ");
    System.out.println();
  }
  public static void main(String[] args){
    // point 0 : entrée dans le main
    int i;
    int[] t = {0, 2, 1, 2, 7, 1, 3, 1, 3};
    int[] tab = repOcc2(t);
    // point 7 : au retour de l'appel a repOcc2, affecter la valeur renvoyée a tab
    ecrireTableau(tab);
    // point 8 : retour de l'appel a ecrireTableau
    // point 9 : fin du programme
  }
}

--> java ExempleFonction
1 3 2 2 0 4 0 1
-->
```

3.4.2 La traduction

3.4.2.1 Les blocs d'activation

Commençons par la définition de la classe `Bloc`, classe dont dériveront les blocs d'activation propres à chaque fonction.
Un objet de cette classe encapsule une valeur permettant de mémoriser le point de retour à la fin de l'appel de la fonction.

```
--> cat Bloc.java
class Bloc {   // information existant dans tout bloc d'activation
  int retour;  // point de retour
  Bloc(int r){ // constructeur de Bloc initialisant le membre retour
     retour = r;
  }
}
```

Les classes `Bloc0`, `Bloc1`, `Bloc2`, `Bloc3` et `Bloc4` correspondent respectivement aux blocs d'activation des méthodes `main`, `repOcc2`, `repOcc1`, `valMax` et `ecrireTableau`.

Chacune de ces classes est une sous-classe (ou classe dérivée) de la classe `Bloc` précédente (sa définition contient la spécification `extends Bloc`). Cela signifie que tout objet de l'une de ces classes sontient un objet de la classe `Bloc` et peut donc être vu de manière dégradée comme un objet de cette classe.

Dans chacune de ces classes, le constructeur qui y est défini commence par invoquer le constructeur de la classe mère (c'est-à-dire la classe `Bloc`) au moyen de l'appel `super(retour)`.

```
--> cat Bloc0.java
class Bloc0 extends Bloc{ // bloc d'activation de main
  String[] args; // la parametre args
  int i;         // la variable i
  int[] t;       // la variable t
  int[] tab;     // la variable tab
  Bloc0(int retour, String[] args){
    super(retour); this.args = args;
  }
}
--> cat Bloc1.java
class Bloc1 extends Bloc{ // bloc d'activation de repOcc2
  int[] tab;    // la reference du tableau en parametre
  int n;        // variable n locale
  int[] occ;    // variable occ locale
  Bloc1(int retour, int[] tab){
    super(retour); this.tab = tab;
  }
}
--> cat Bloc2.java
class Bloc2 extends Bloc { // bloc d'activation de repOcc1
  int[] tab;    // la reference du tableau en parametre
  int n;        // le parametre n
  int[] occ;    // la variable locale occ
  int i;        // la variable locale i
  Bloc2(int retour, int[] tab, int n){
    super(retour); this.tab = tab; this.n = n;
  }
}
--> cat Bloc3.java
class Bloc3 extends Bloc{ // bloc d'activation de valMax
  int[] tab;  // la reference du tableau en parametre
  int i;      // la variable locale i
  int x;      // la variable locale x
  Bloc3(int retour, int[] tab){
    super(retour); this.tab = tab;
  }
}
--> cat Bloc4.java
class Bloc4 extends Bloc{ // bloc d'activation de ecrireTableau
  int[] tab;  // la reference du tableau en parametre
  int i;      // la variable locale i
  Bloc4(int retour, int[] tab){
    super(retour); this.tab = tab;
  }
}
-->
```

3.4.2.2 La traduction avec une pile

Dans le code suivant on remarquera que :

- nous avons spécifié qu'on utilise une pile dont les éléments référencent des objets de la classe `Bloc`.
  Il doit être clair qu'en faisant cela, on autorise l'écriture sur la pile de références vers des objets de l'une des classes `Bloc`i quelconque (avec $i \in \{0, 1, 2, 3, 4\}$) puisque toute référence de ce type est implicitement référence vers `Bloc` (ce qu'on appelle transtypage implicite vers le haut ou *upcasting*) ;
- lorsqu'on extrait un élément de la pile, on le considère a priori du type `Bloc` et c'est sur la base de l'instruction qu'on exécute (valeur de la variable `instruction`) qu'on le regarde comme une référence vers un objet plus spécifique (un `Bloc`i particulier).
  On réalise alors une opération
  `bloc`i `= (Bloc`i`)bloc ;`
  Une telle opération est ce qu'on nomme un transtypage explicite vers le bas (`downcasting`). Elle demande à ce que la référence vers un objet de la classe `Bloc` soit vue plus finement comme une référence vers un objet de la classe `Bloc`i.
  Si cela a un sens (c'est-à-dire si l'objet référencé par `bloc` est effectivement de la classe `Bloc`i ou une de ses dérivées), il est alors possible d'accéder aux informations spécifiques encapsulées dans cet objet de la classe `Bloc`i.

```
--> cat ExemplePile.java
import java.util.Stack;
  class ExemplePile{
    public static void main(String[] args){
      int[] tt = {7, 0, 2, 1, 2, 7, 1, 3, 1, 3};
      int instruction,    // instruction a executer
          valeurInt = 0;  // valeur de retour de type int
      int[] valeurTab = null; // valeur de retour de type int[]
      Stack<Bloc> pile = new Stack<Bloc>();
      Bloc bloc; Bloc0 bloc0; Bloc1 bloc1; Bloc2 bloc2; Bloc3 bloc3; Bloc4 bloc4;
      pile.push(new Bloc0(-1, args)); // appel de la fonction rep0cc2
      instruction = 0;
      while (!pile.empty()){
          // System.out.print(instruction + " ");
          // System.out.println();
          bloc = (Bloc)pile.peek();
          switch(instruction){
             case 0 : // entreee dans le main
               bloc0 = (Bloc0)bloc;
               bloc0.t = tt;
               pile.push(new Bloc1(7, bloc0.t));
               instruction = 3; // on va executer la ligne reperee par point 3
               break;
             case 3 : // appel de la fonction rep0cc2
                      // le bloc est de type Bloc1;
                bloc1 = (Bloc1)bloc;
                pile.push(new Bloc3(4, bloc1.tab));
                instruction = 1; // on va executer la ligne reperee par point 1
                break;
             case 1 : // on entre dans la fonction valMax
                bloc3 = (Bloc3)bloc;
                bloc3.x = bloc3.tab[0];
                for(bloc3.i = 1; bloc3.i < (bloc3.tab).length; bloc3.i++)
                    if((bloc3.tab)[bloc3.i] > bloc3.x) bloc3.x = (bloc3.tab)[bloc3.i];
                instruction = bloc.retour;  // retour a l'envoyeur
                valeurInt =  bloc3.x;
                pile.pop();
                break;
```

```
            case 4 :
               bloc1 = (Bloc1)bloc;
               bloc1.n = valeurInt;
               pile.push(new Bloc2(_, bloc1.tab, bloc1.n));
               instruction =  2; // on va execute la ligne reperee par point 2
               break;
            case 2 :
               bloc2 = (Bloc2)bloc;
               bloc2.occ = new int[bloc2.n + 1];
               for(bloc2.i = 0; bloc2.i < (bloc2.tab).length; bloc2.i++)
                   bloc2.occ[(bloc2.tab)[bloc2.i]]++;
               valeurTab = bloc2.occ;
               instruction = bloc.retour; // retour a l'envoyeur
               pile.pop();
               break;
            case _:
               bloc1 = (Bloc1)bloc;
               bloc1.occ = valeurTab;
               valeurTab = bloc1.occ;
               instruction = bloc.retour; // retour a l'envoyeur
               pile.pop();
               break;
            case 7 :
               bloc0 = (Bloc0)bloc;
               bloc0.tab = valeurTab;
               pile.push(new Bloc4(8, bloc0.tab));
               instruction = 6; // on va executer la ligne reperee par point 6
             break;
            case 6 :
               bloc4 = (Bloc4)bloc;
               for(bloc4.i = 0; bloc4.i < (bloc4.tab).length; bloc4.i++)
                   System.out.print(bloc4.tab[bloc4.i] + " ");
               System.out.println();
               instruction = bloc.retour; // retour a l'envoyeur
               pile.pop();
               break;
            case 8 :
               pile.pop();
               instruction = 9; // on va executer la ligne reperee par point 9
               break;
            case 9 :
               pile.pop();
               break;
         }
      }
   }
}
--> java ExemplePile
1 3 2 2 0 4 0 1
-->
```

4 Interprétation des identificateurs

4.1 Nom et valeur

Si on considère une instruction telle que l'instruction d'affectation Java/C/C++

```
i = i + 1;   // i déclaré en tant que int
```

l'interprétation que l'on fait de l'identificateur `i` est différente dans sa partie gauche et sa partie droite :

- dans la partie droite, l'identificateur est interprété comme *valeur de la grandeur identifiée par i et du type de i*. On parle de ***R-value*** (pour *Right-value*) ;
- dans la partie gauche, l'identificateur est interprété comme *emplacement en mémoire de la grandeur identifiée par i*. On parle de ***L-value*** (pour *Left-value*).

Les deux erreurs signalées par le compilateur correspondent au fait qu'on essaie de redéfinir un identificateur qui l'est déjà (le paramètre `arg` et la variable `i`).
Une traduction possible de ce programme en C serait :

```
--> cat pasErreur.c
int main(int argc, char  arg[]) {
   int arg = 3, i = 3;
   { int i;
     for (i = 0; i < arg; i++)
         printf("%d ",i);
   }
   printf("\n%d\n", i);
}
--> gcc pasErreur.c -o pasErreur
pasErreur.c: In function ‘main’:
pasErreur.c:2: warning: declaration of ‘arg’ shadows a parameter
--> ./pasErreur
0 1 2 3 4
3
-->
```

Sa compilation provoque simplement l'affichage d'un avertissement (*warning*) indiquant une redéfinition de l'identificateur `arg` masquant le paramètre de même nom. Un fichier exécutable est cependant généré.
Nous reviendrons un peu plus loin sur les résultats qu'il fournit.
Le problème général sous-jacent est le suivant : étant donnée une occurrence d'un identificateur, cette occurrence peut-elle être interprétée, c'est-à-dire associée à une *L-value* ou une *R-value* selon le cas et si oui à laquelle si plusieurs associations sont possibles.
Les choses sont plus ou moins compliquées selon la permissivité du langage en terme de redéfinitions des identificateurs. Nous allons nous concentrer sur ce qui se passe en Java où les choses sont, somme toutes, assez simples.
L'exemple suivant illustre ce qu'il est possible de faire en Java en matière d'utilisation multiple d'un identificateur de variables (la définition multiple d'identificateurs de fonctions est un autre sujet).
Nous avons numéroté en commentaire chacune des lignes qui sera exécutée et aura une incidence sur l'interprétation et les valeurs possibles associées à l'identificateur `i`.

```
-> cat Redef.java
class Redef {
  static int i = 30;                          // 1
  static void f(int n) {                      // 2
    System.out.println("f : " + i);           // 3
    for(int i = 0; i < n; i++)                // 4
      System.out.println("f : " + i);         // 5
    n = 6;                                    // 6
    i = 10;                                   // 7
    g(i);                                     // 8
    System.out.println("f : " + i); }         // 9
  static void g(int i){                       // 10
    System.out.println("g : " + i);           // 11
    i = i + 100;                              // 12
    System.out.println("g : " + i); }         // 13
  public static void main(String[] args) {
    System.out.println("main : " + i);        // 14
    int i = 2;                                // 15
    f(i);                                     // 16
    System.out.println("main : " + i);        // 17
    System.out.println("main : " + Redef.i); } // 18
}
```

Il y a quatre définitions de l'identificateur `i` (lignes `1`, `4`, `10` et `15`). Il se trouve que pour chacune de ces définitions, on définit une variable de type `int`, mais on aurait pu imaginer de leur associer des définitions de variables de types différents.
Au travers de chacune de ces définitions, l'identificateur `i` est associée à un emplacement en mémoire : cette association (ou liaison) constitue ce qu'on appelle communément une variable.
Chacune de ces associations possède une **durée de vie** (entre sa création et sa libération) et une **visibilité**.
Il est important de noter qu'une variable peut être définie (exister) mais ne pas être visible.
On distingue :

- les variables qualifiées de **globales** qui existent en général durant toute l'exécution du programme : elles sont allouées une fois pour toutes (on parle d'allocation statique) ;
- les variables dites **locales** qui sont créées au cours de l'exécution (typiquement à l'appel d'une fonction ou à l'entrée dans un bloc d'instructions) et supprimées avant la fin de l'exécution (au retour d'un appel de fonction ou à la sortie du bloc d'instructions). Elles ont allouées de manière automatique sur la pile.
  Remarque : on peut parler d'allocation dynamique. Mais nous préférons garder ce terme pour les allocations effectuées dans le tas par exemple par l'opérateur `new` de Java ou la fonction `malloc` de C.

Pour l'exemple donné plus haut :

- la variable résultant de la définition de la ligne `1`, et que nous désignerons par $\texttt{i}_1$, est ce qu'on appelle une **variable de classe**. Elle existe dès que la classe `Redef` est chargée en mémoire. Elle est externe aux trois fonctions `f`, `g` et `main` et y est a priori visible. Le nom i pourra ne pas être suffisant pour accéder à cette variable lorsque ce nom aura été réutilisé pour désigner une autre variable. Dans ce cas il sera nécessaire d'utiliser le nom qu'on peut qualifier d'absolu de cette variable, à savoir `Redef.i` comme cela est fait en ligne 18 ;
- la variable $\texttt{i}_4$, correspondant à la définition de la ligne 4, est créée lorsqu'on entre dans la boucle et détruite lorsqu'on en sort. Elle n'est visible que dans cette boucle (instructions de contrôle et du corps). Cette définition rend la variable $\texttt{i}_1$ invisible au travers de l'identificateur `i` : on dit que cette définition masque l'autre ;
- la variable $\texttt{i}_{10}$ est la variable associée au paramètre formel lors de l'appel de la fonction `g`. Cette définition masque également celle de la variable $\texttt{i}_1$ (elle pourrait cependant être accédée en utilisant le nom `Redef.i`). Cette variable n'est visible que dans la fonction `g`. La variable $\texttt{i}_4$ n'est évidemment pas visible ici puisqu'à ce point on est à l'extérieur du bloc où elle est définie ;
- la variable $\texttt{i}_{15}$, associée à la définition de la ligne 15 est une variable locale de la fonction `main`. Elle n'existe qu'entre la ligne 14 et la sortie de la fonction après la ligne 18 donc (en l'occurrence la fin du programme puisqu'il s'agit de la fonction `main`). Elle n'est visible que lorsqu'on est dans la fonction `main` (elle ne l'est pas lorsqu'on est dans `f` ou `g`).

L'exécution de ce programme fournit les résultats suivants :

```
--> java Redef
main : 30
f : 30
f : 0
f : 1
g : 10
g : 110
f : 10
main : 2
main : 10
-->
```

Détaillons les différentes étapes de cette exécution qui conduisent à ces résultats :

Instruction	Liaison de i	Variable i_1	Variable i_4	Variable i_{10}	Variable i_{15}
Lancement du programme Appel de la méthode `main`	i_1	création `Redef.i` Valeur : `30`			
`// 14` : affichage de $i \longrightarrow$ `30`	i_1	`30`			
`// 1`␣: affectation de i	i_{15}	`30`			création Valeur : `2`
`// 16` : appel de `f`	i_{15}	`30`			`2`
`// 2` : entrée dans `f` variable locale n=`2`	i_1	`30`			`2`
`// 3` : affichage de $i \longrightarrow$ `30`	i_1	`30`			`2`
`// 4` : boucle	i_4	`30`	création valeur : `0`		`2`
`//` ␣: affichage de $i \longrightarrow$ `0`	i_4	`30`	`0`		`2`
`// 4` : boucle	i_4	`30`	`1`		`2`
`//` ␣: affichage de $i \longrightarrow$ `1`	i_4	`30`	`1`		`2`
`// 4` : sortie de boucle	i_4	`30`	destruction		`2`
`// 6` : n prend la valeur `6`	i_1	`30`			`2`
`// 7` : affectation de i	i_1	`10`			`2`

`// 8` : appel de `g`
`// 10` : entrée dans `g`
variable locale i=`10`
	i_{10}	`10`		création
				valeur : `10`
					`2`

`// 11` : affichage de $i \longrightarrow$ `10`	i_{10}	`10`		`10`	`2`
`// 12` : affectation de i	i_{10}	`10`		`110`	`2`
`// 13` : affichage de $i \longrightarrow$ `110` sortie de `g`	i_{10}	`10` `10`		`110` destruction	`2` `2`
`// 9` : affichage de $i \longrightarrow$ `10` sortie de `f`	i_1 i_1	`10` `10`			`2` `2`
`// 17` : affichage de $i \longrightarrow$ `2` `// 18` : affichage de $Redef.i \longrightarrow$ `10`	i_{15} i_{15}	`10` `10`			`2` `2`

5 La récursion

5.1 Définition

La récursion (ou récursivité) est le principe consistant à « définir un concept en fonction de lui-même ».

On peut l'utiliser par exemple :

- pour donner la définition de « quelque chose » : par exemple une expression arithmétique ou booléenne.
  Des expressions atomiques (constantes ou variables) ayant été définies, on définit une expression soit comme étant une expression atomique, soit comme étant construite par combinaison d'une ou deux (voire plus) expressions au moyen d'opérateurs ;
- pour définir des structures de données telles que les listes ou les arbres ;
- pour définir des fonctions ou des procédures. Pour définir une fonction, on fait appel à la fonction elle-même. Ce mode de définition de fonctions est fondamentale dans la programmation fonctionnelle : il s'y substitue au mécanisme d'itération (qui n'y existe pas).

5.2 Les structures de données récursives

5.2.1 Le principe

Les structures telles que les listes ou les arbres par exemple sont des structures dynamiques sur lesquelles on souhaite réaliser des opérations telles que l'extraction ou l'insertion/ajout d'un élément ou le parcours.
Une définition récursive d'un tel élément revient à définir la structure d'un élément constituant en ajoutant en plus de la valeur des informations permettant d'accéder à un ou plusieurs successeurs ou prédécesseurs de même nature.
La définition récursive d'une structure doit permettre d'énumérer tous les éléments de la structure à partir d'un de ses élémemts (la tête, la racine, ...).

5.2.2 Exemples

5.2.2.1 Liste chaînée de valeurs

Dans ce premier exemple, on définit, en première approche, un élément d'une liste de valeurs entières comme un couple constitué d'une valeur et d'un lien vers l'élément suivant dans la liste. Ce lien n'est rien d'autre que la référence en mémoire de cet élément. Ainsi, une liste privée de son premier élément est encore une liste.
Cette définition permet, à partir de la connaissance du premier élément de la liste de parcourir toute la liste en suivant le lien `suivant`.

```
class Liste {
  int valeur;
  Liste suivant;
}
```

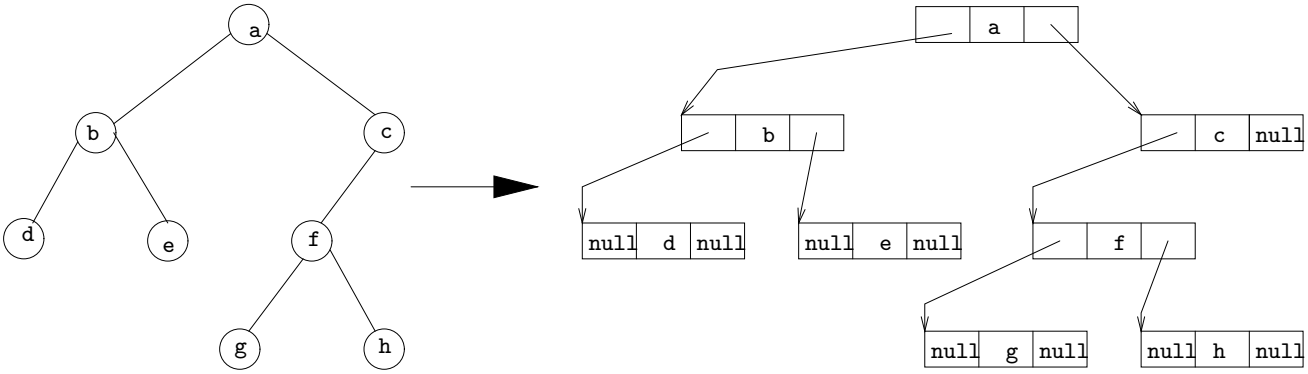
Le schéma suivant illustre une liste constituée de quatre entiers représentée selon ce principe. On voit que la connaissance du premier élément (celui situé à gauche) en permet le parcours exhaustif.

5.2.2.2 Arbre binaire

Dans ce second exemple, on définit, en première approche, un nœud élément d'un arbre binaire. Là, un élément de l'arbre a au plus deux successeurs, ses fils gauche et droit, d'où la définition suivante :

```
class Arbre {
    char etiquette; Arbre gauche, droit; }
```

À partir de la racine, il est possible en suivant les liens `gauche` et `droit` de parcourir l'ensemble des nœuds de l'arbre.
Un exemple d'arbre et sa représentation est donnée dans la figure suivante :



5.2.3 Les définitions récursives de fonctions/méthodes/procédures

Nous allons en illustrer le principe sur quelques exemples simples.

5.2.3.1 La somme des n premiers nombres

Soit $S(n)$ la somme des n premiers entiers. On a :
- $S(0) = 0$
- $\forall i > 0$, $S(i) = S(i-1) + i$

Une telle définition est fortement liée au concept de récurrence des mathématiques.
La correction d'une telle définition suppose le respect de quelques règles :
- la fonction définie récursivement doit l'être au moyen d'une condition dont au moins un cas est calculable sans devoir faire appel à la récursion. En un mot, il doit exister des cas dégénérés ou conditions d'arrêt ;
- quelle que soit la valeur ou les valeurs pour lesquelles la fonction est calculée, il faut qu'une condition d'arrêt puisse être atteinte en un nombre fini d'étapes (c'est-à-dire d'appels récursifs).

Sur l'exemple précédent, la condition d'arrêt est que le paramètre est nul (la valeur de la fonction est alors `0`). Par ailleurs, partant d'une valeur positive, on finira par atteindre la condition d'arrêt en un nombre fini d'étapes. Par contre, en appelant la fonction avec un nombre négatif (on ne devrait pas), ce ne sera pas le cas.
Cette définition de la fonction se traduit brutalement en Java :

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

5.2.3.2 La factorielle $n!$ d'un nombre n

Elle peut être définie de la manière suivante :
◦ $0! = 1$
◦ $\forall i > 0,\ i! = (i-1)! \times i$
Cette définition conduit également à une expression simple de la fonction en Java :

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

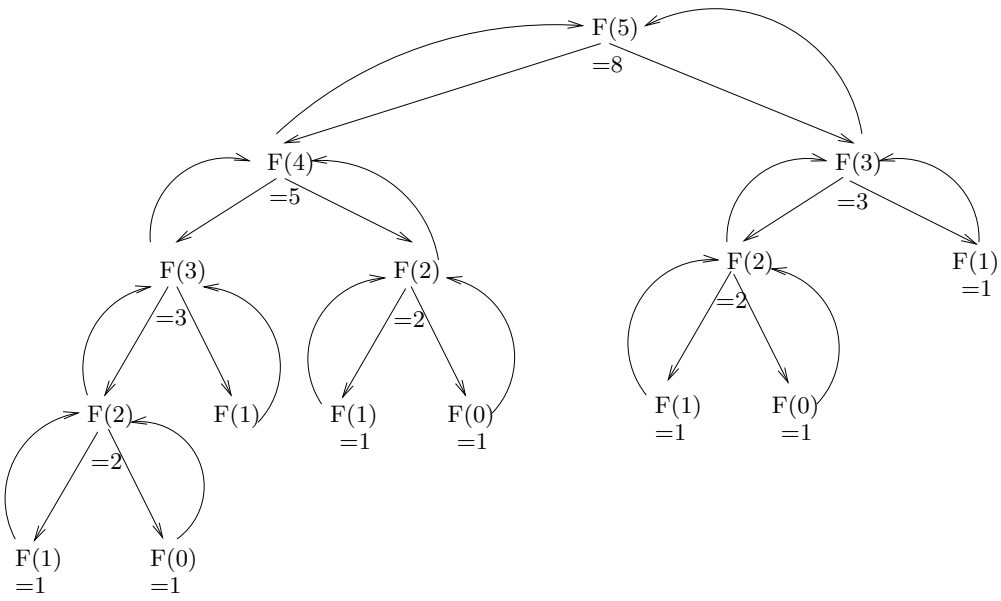
5.2.3.3 Calcul du n-ième élément d'une suite définie par récurrence

Considérons par exemple la suite de Fibonacci dont le terme F_n est défini par :
• $F_0 = F_1 = 1$
• $\forall i \geq 2,\ F_i = F_{i-1} + F_{i-2}$

Le code suivant contient la définition de la fonction et d'une application l'utilisant :

```
-> cat Fibo1.java
class Fibo1 {
  static long fibo(int n) {
     if (n == 0 || n == 1) return 1;
     return fibo(n-1) + fibo(n-2); }
  public static void main(String[] args) {
     System.out.println("f4␣ = " + fibo(4␣));
     System.out.println("f␣ = " + fibo(␣));
     System.out.println("f10 = " + fibo(10)); }
}
--> java Fibo1
f4␣ = 1836311903
f␣ = 8
f10 = 89
-->
```

La suite des appels successifs de la fonction pour l'appel initial de `f(5)` peut être commodément représentée par l'arbre suivant (dont les arêtes sont construites dans l'ordre préfixe) :



Pour calculer $F(5)$, il faut calculer $F(4)$ et $F(3)$.
Pour calculer $F(4)$, il faut calculer $F(3)$ et $F(2)$.
...

5.3 La récursion illustrée par des exemples

5.3.1 Partage (ou partition) d'un nombre entier positif

Un partage (ou une partition) d'un entier n est une décomposition de l'entier n comme somme d'entiers strictement positifs (appelés ses sommants).
Si on considère les entiers de 1 à 7, ils peuvent s'écrire (aux commutations près) :

1 : `1`

2 : `2, 1+1`

3 : `3, 2+1, 1+1+1`

4 : `4, 3+1, 2+2, 2+1+1, 1+1+1+1`

5 : `5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1`

6 : `6, 5+1, 4+2, 4+1+1, 3+2+1, 3+1+1+1, 2+2+2, 2+2+1+1, 2+1+1+1+1, 1+1+1+1+1+1`

7 : `7, 6+1, 5+2, 5+1+1, 4+3, 4+2+1, 4+1+1+1, 3+3+1, 3+2+2, 3+2+1+1, 3+1+1+1+1, 2+2+2+1, 2+2+1+1+1, 2+1+1+1+1+1, 1+1+1+1+1+1+1`

Ils possèdent donc respectivement exactement 1, 2, 3, 5, 7, 10 et 15 partages.

Adoptons les notations suivantes :

- $p(n,k)$ le nombre de partages de l'entier n constitués d'au plus k nombres.
  On a par exemple $p(5,3) = 5$, $p(7,3) = 8$ et $p(7,4) = 11$;
- $p'(n,k)$ le nombre de partages de n constitués d'exactement k nombres.
  On a par exemple $p'(5,3) = 2$ et $p'(7,4) = 3$;
- $P(n)$ le nombre de partages de n. On a bien évidemment $P(n) = p(n,n)$ et $P(n) = \sum_{i=1}^{n} p'(n,i)$.

On souhaite calculer $p(n,k)$ pour des valeurs données de n et k.
Nous allons voir qu'une approche récursive du problème permet d'en donner une solution simple.
Intéressons nous d'abord aux cas simples pour lesquels il est facile de donner la valeur de $p(n,k)$ ou de se ramener à une valeur $p(n',k')$ avec $n' < n$ et/ou $k' < k$

- $\forall k \geq 0$, $p(0,k) = 1$ (le nombre 0 s'écrit d'une seule manière, comme somme de 0 nombre)
- $\forall n > 0$, $p(n,0) = 0$ et $p(n,1) = 1$
- si $k > n$, $p(n,k) = p(n,n)$

Intéressons nous maintenant au cas général.
Un partage de n en au plus k nombres est soit un partage de n en exactement k nombres, soit un partage de n en au plus $k-1$ nombres, et donc :

$$p(n,k) = p'(n,k) + p(n,k-1).$$

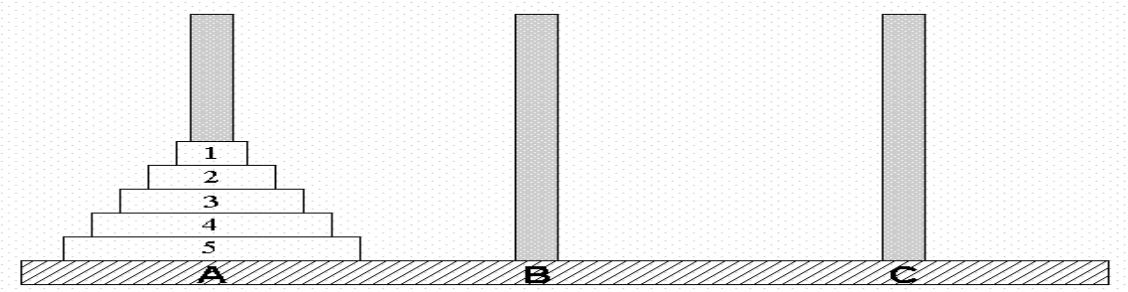
Intéressons nous à $p'(n,k)$.
Chacun des k sommants d'une décomposition de n comportant k nombres est un nombre strictement positif (c'est-à-dire ≥ 1). En retranchant 1 de chacun de ces k nombres, on obtient une suite d'au plus k nombres ≥ 1 dont la somme est $n-k$. Donc $p'(n,k) \leq p(n-k,k)$.
Inversement, à partir d'un partage de $n-k$ en au plus k nombres, on peut construire un partage de n en exactement k nombres.
Soit en effet un tel partage : $n-k = c_1 + c_2 + \ldots + c_l$ avec $l \leq k$
On a aussi $n-k = c_1 + c_2 + \ldots + c_l + c_{l+1} + \ldots c_k$ avec $c_{l+1} = \ldots = c_k = 0$.
Soit, pour chaque entier $1 \leq i \leq k$, $c'_i = c_i + 1$.
On a : $c'_1 + c'_2 + \ldots c'_k = n$.

Donc, la suite $c'_1, \ldots, c'_k$ est un partage de n en k nombres.
Par suite, $p(n-k,k) \leq p'(n,k)$.
Finalement, on tire des deux inégalités, il découle que $p'(n,k) = p(n-k,k)$.
On a ainsi un procédé de calcul récursif de $p(n,k)$ dont la valeur se déduit de valeurs de p(n',k') avec $n' < n$ ou $k' < k$. Ainsi, en partant de valeurs positives de n et k on atteindra en un nombre fini d'étapes une des conditions d'arrêt.
Cela conduit directement au programme Java suivant :

```
--> cat Partage.java
class Partage {
  static int part(int n, int k) {
    if (n == 0 || k == 1) return  1;
    if (k == 0) return 0;
    if (k > n) return part(n, n) ;
    return (part(n - k, k) + part(n, k - 1));
  }
  public static void main(String[] args) {
    System.out.println("p(_, 3) = " + part(_, 3));
    System.out.println("p(7, 3) = " + part(7, 3));
    System.out.println("p(7, 4) = " + part(7, 4));
    System.out.println("p'(_, 3) = " + part(2, 3));
    System.out.println("p'(7, 4) = " + part(3, 4));
    System.out.println("p(20, 7) = " + part(20, 7));
    System.out.println("P(10) = " + part(10, 10));
    System.out.println("P(_0) = " + part(_0, _0));
  }
}
--> java Partage
p(_, 3) = _
p(7, 3) = 8
p(7, 4) = 11
p'(_, 3) = 2
p'(7, 4) = 3
p(20, 7) = 364
P(10) = 42
P(_0) = 204226
-->
```

5.3.2 Les tours de Hanoi

Problème on ne peut plus classique illustrant l'approche récursive de résolution d'un problème. Ici, il ne s'agit pas de calculer la valeur d'une fonction, mais de décrire une suite d'actions à réaliser pour résoudre un casse-tête : on est donc en mode procédural (on définit des suites d'actions plutôt que des valeurs).
Ce jeu est composé de trois tours nommées respectivement A, B et C et d'anneaux enfilés sur ces tours. Ces anneaux doivent obligatoirement être posés les uns sur les autres par taille décroissante. Le jeu consiste à déplacer (en un minimum de temps) tous les anneaux placés par exemple sur la tour A vers la tour B, en utilisant la tour C comme intermédiaire.



Pour réaliser ce transfert complet, on doit respecter les règles suivantes :

- on ne peut déplacer qu'un seul anneau à la fois ;
- on ne peut placer un anneau que sur un plus grand que lui ou sur une tour vide.

Une approche récursive pour résoudre le problème va suggérer :

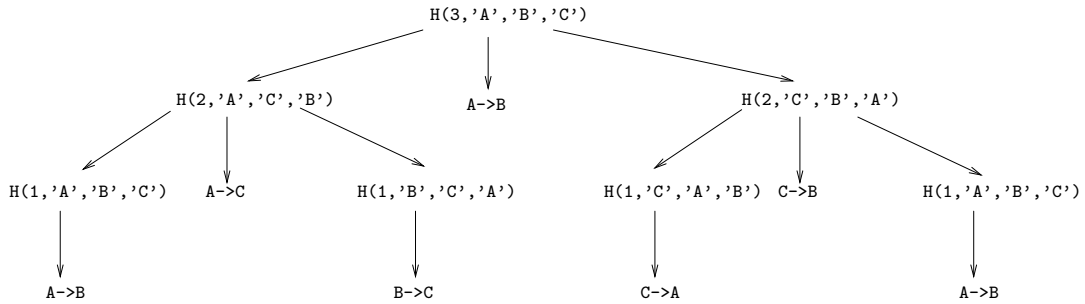
- de considérer le problème de taille $n-1$ consistant à déplacer les $n-1$ anneaux supérieurs sur la tour A vers la tour C en utilisant la tour B comme intermédiaire ;
- une fois cela fait, de transférer le dernier anneau de la tour A vers la tour B ;
- de résoudre le problème de taille $n-1$ consistant à déplacer les $n-1$ anneaux sur la tour C vers la tour B en utilisant la tour A comme intermédiaire.

Cela fonctionne car le dernier anneau de A est plus grand que tous les autres et peut donc être utilisé comme intermédiaire.
Cela conduit au programme Java suivant :

```
--> cat Hanoi.java
class Hanoi {
  static void hanoi(int n, char origine, char destination, char intermediaire) {
   // fonction pour d\'epace n disques d'une origine \`a une destination
    switch(n)  {
        case 0  : return ;  // pas atteint sauf pour un appel initial avec n = 0
        case 1  : System.out.println(origine + " --> " + destination);
                  break;
        default : hanoi(n - 1, origine, intermediaire, destination);
                  System.out.println(origine + " --> " + destination);
                  hanoi(n - 1, intermediaire, destination, origine);
        }

  }
  public static void main(String[] arg) {
        hanoi(3, 'A', 'B' ,'C');
        System.out.println("============================");
        hanoi(4, 'A', 'B' ,'C');
  }
}
```

L'arbre suivant décrit la suite des appels réalisés pour l'appel initial `hanoi(3, 'A', 'B' ,'C')`. Cet arbre doit être lu en profondeur (c'est-à-dire par un parcours préfixe) :



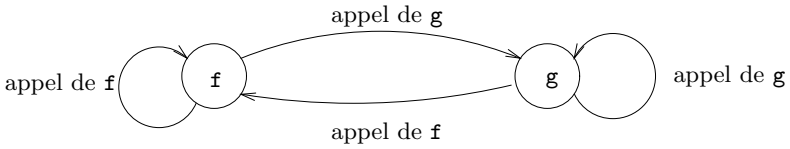
L'exécution du programme Java fournit les résultats correspondant à ce parcours pour l'appel `hanoi(3, 'A', 'B' ,'C')`.

```
--> java Hanoi
A --> B
A --> C
B --> C
A --> B
C --> A
C --> B
A --> B
============================
```

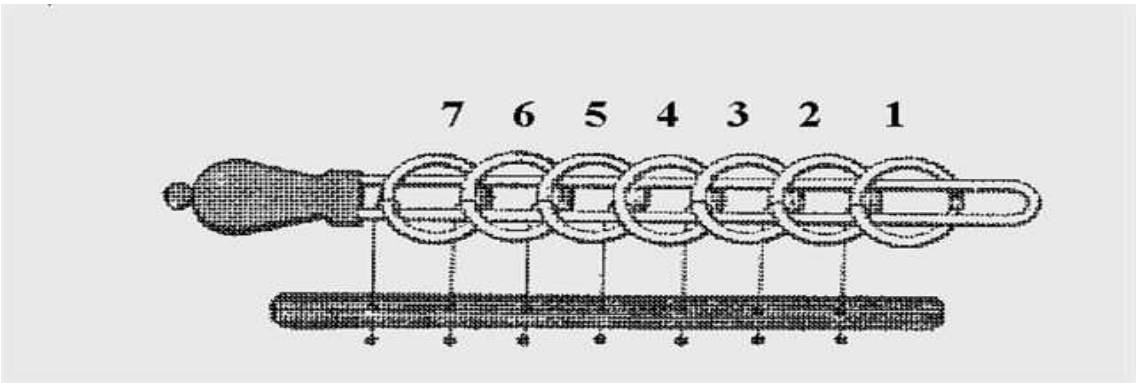
```
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
A --> B
C --> B
C --> A
B --> A
C --> B
A --> C
A --> B
C --> B
-->
```

5.3.3 Un exemple de récursion croisée : le baguenaudier

Ce problème est intéressant d'un point de vue informatique car une solution simple en est donnée en utilisant une récursion croisée. Une fonction `f` y est définie en appelant `f` elle-même et une fonction `g` alors que cette fonction `g` est définie en fonction d'elle-même et de la fonction `f`.
La figure suivante illustre le graphe des appels correspondant à cette situation :



Le baguenaudier est un curieux appareil d'origine chinoise, appelé aussi « anneaux chinois » ou « anneaux de Cardan », dont l'invention remonte aux années 200. Il est formé d'un jeu d'anneaux qu'il faut enfiler puis retirer, dans un certain ordre, sur un morceau d'os ou de métal plat, percé de trous dans chacun desquels se trouve un fil de fer. Une des extrémités de chaque tige est attachée à un anneau et l'autre se termine par une tête qui empêche la tige de s'échapper du trou.



Afin de chercher une solution, nous allons donner d'un tel casse-tête, supposé constitué de n anneaux, une description sous forme de damier et dévoiler le secret de son fonctionnement :

- le baguenaudier est une règle comportant n cases numérotées de gauche à droite de 1 à n et sur chacune des cases est placé un pion. le but de jeu est d'enlever les n pions. Au cours du temps, une case a deux états possibles, occupé (un pion dessus) ou libre (pas de pion dessus) ;
- l'état de la case 1 peut être changé n'importe quand ;
- l'état de la case i ($i \geq 2$) peut être changé si la case $i-1$ est occupée et chacune des cases antérieures (c'est-à-dire $1, \ldots, i-2$) est libre.

Si on représente l'état du baguenaudier par une suite de n caractères 0 ou 1 dans laquelle un caractère 0 en position i indique que la case i est libre (ordre inversé par rapport à l'image) et un caractère 1 en cette même position indique que la case est occupée, le jeu consiste à passer de la configuration :

$\underbrace{1,\ldots,1}_{n fois}$, notée symboliquement 1^n

à la configuration

$\underbrace{0,\ldots,0}_{n fois}$ notée symboliquement 0^n

par une suite d'opérations autorisées.
Ce changement d'état est noté $1^n \rightarrow 0^n$ et `bag(n)` est la suite des opérations pour le réaliser. Inversement, `debag(n)` est la suite des opérations permettant de passer de l'état 0^n à l'état 1^n (changement d'états noté $0^n \rightarrow 1^n$).
En utilisant le même formalisme, les opérations autorisées correspondent aux changements d'états suivants :

- $x\alpha \rightarrow (1-x)\alpha$ pour toute suite α de longueur $n-1$;
- $0^{i-2}1x\beta \rightarrow 0^{i-2}1(1-x)\beta$, pour tout $i \geq 2$ et toute suite β de longueur $n-i$.

Pour résoudre le problème pour un entier $n \geq 2$ on peut procéder de la manière suivante :

- résoudre le baguenaudier plus petit constitué des $n-2$ premières cases. La suite des opérations est ce que nous avons appelé `bag(n-2)`.
  Partant de l'état 1, on arrive alors en l'état $0^{n-2}11$;
- il est alors possible de modifier l'état de la case n. En enlevant le pion qui s'y trouve, on passe en l'état $0^{n-2}10$;
- l'idée est alors de reconstituer un baguenaudier de taille $n-1$ dans son état initial. Pour cela, il suffit de reposer les pions sur les $n-2$ premières cases. Cela est fait par la séquence `debag(n-2)`.
  Après cette séquence, le baguenaudier est dans l'état $1^{n-1}0$;
- il suffit alors de réaliser la séquence `bag(n-1)` pour enlever les $n-1$ premiers pions et se retrouver ainsi en l'état 0^n.

Ce procédé et la définition des deux cas $n = 0$ et $n-1$ fournissent une solution, si tant est que la suite `debag(n)` soit elle même définie. Pour cela, il suffit de reprendre la séquence précédente à l'envers en inversant les opérations réalisées : au lieu d'enlever un pion on le pose, au lieu de faire une suite d'opérations `bag(x)` on fera la suite d'opérations `debag(x)` ou l'inverse.
Ainsi, la suite d'opérations `debag(n)` avec $n \geq 2$ peut être définie par :

- faire la séquence d'opérations `debag(n-1)` ;
- faire la séquence d'opérations `bag(n-2)` ;
- poser le pion sur la case n ;
- faire la séquence d'opérations `debag(n-2)`.

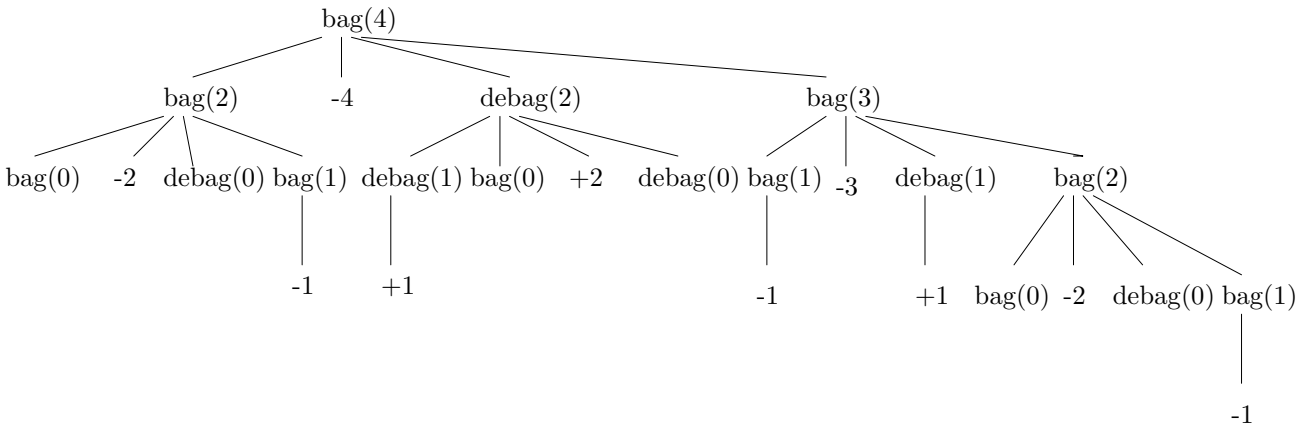
Pour résumer, les deux séquences d'opérations `bag(n)` et `debag(n)` (pour un entier $n \geq 2$) font passer par les états suivants :

- `bag(n)` : $1^n \rightarrow 0^{n-2}11 \rightarrow 0^{n-2}10 \rightarrow 1^{n-2}10 = 1^{n-1}0 \rightarrow 0^n$
- `debag(n)` : $0^n \rightarrow 1^{n-1}0 = 1^{n-2}10 \rightarrow 0^{n-2}10 \rightarrow 0^{n-2}11 \rightarrow 1^{n-2}11 = 1^n$

Cela conduit au programme Java suivant, dans lequel une opération `+n` signifie « poser un pion sur la case n » et une opération `-n` signifie « enlever le pion sur la case n » :

```
--> cat Baguenaudier.java
class Baguenaudier {
  static void bag(int n) { // enlever les n pions
    if(n == 0)
      return;
    if(n == 1) {
      System.out.print("-1  "); // enlever le seul pion
      return;
    }
    bag(n-2); // enlever les n-2 premiers pions
    System.out.print("-" +  n + "  "); // enlever le pion n
    debag(n-2); // poser les n-2 premiers pions
    bag(n-1); // enlever les n-1 premiers pions
  }
  static void debag(int n) { // poser les n pions
    if(n == 0)
      return;
    if(n == 1) {
      System.out.print("+1  "); // poser le seul pion
      return;
    }
    debag(n-1); // poser les n-1 premiers pions
    bag(n-2); // enlever les n-2 premiers pions
    System.out.print("+" + n + "  "); // poser le pion n
    debag(n-2); // poser les n-2 premiers pions
  }
public static void main(String[] arg) {
  bag(4);
  System.out.println();
  }
}
--> java Baguenaudier
-2  -1  -4  +1  +2  -1  -3  +1  -2  -1
-->
```

L'arbre des appels de fonctions au cours de cette exécution est le suivant :



La suite de mouvements affichée lors de l'exécution du programme est obtenue par parcours préfixe (ou en profondeur) de cet arbre :

```
-2  -1  -4  +1  +2  -1  -3  +1  -2  -1
```

5.4 Performances des applications récursives. Programmation dynamique

5.4.1 Le coût de la récursion

Si nous examinons l'arbre des appels d'un programme tel que celui calculant le n^e terme de la suite de Fibonacci ou de celui calculant le nombre $p(n,k)$ de partages d'un entier n en au plus k sommants, on observe que la formulation récursive de la solution conduit, lors de son exécution, à refaire un certain nombre de fois les mêmes calculs. Plus le paramètre est grand et plus on refera de fois les mêmes calculs.
Cela a évidemmment une incidence importante sur les performances de ces programmes.
Afin de quantifier l'observation relative au temps d'exécution d'une application, nous lançons les applications au travers de la commande `time` des systèmes de la famille Unix (par exemple Linux). Lorsque l'application lancée se termine, trois temps sont visualisés :

- le temps réel (`real`) : il correspond au temps absolu, tel que vécu par l'utilisateur attendant l'affichage des résultats, qui s'est écoulé entre le lancement de l'application et sa terminaison. Pour un même programme, ce temps peut différer d'une exécution à l'autre, en fonction de la charge du système. Si l'application est seule à s'exécuter, ce temps sera très voisin de la somme des deux suivants ;
- le temps CPU normal, on dit utilisateur (`user`) : il correspond au temps CPU consommé (temps d'utilisation effective du processeur) pour faire du calcul. C'est ce temps qui reflète ce que coûte un programme. Pour un même programme, il est globalement identique pour toutes les exécutions ;
- le temps CPU système (`sys`) : il correspond au temps CPU consommé par l'application pour réaliser des opérations particulières (entrées-sorties par exemple).

Pour l'application Fibonacci, on a obtenu :

```
--> time java Fibo1
f4 = 1836311903
f5 = 8
f10 = 89

real    0m48.917s
user    0m48.810s
sys     0m0.080s
```

Dans le temps CPU consommé par le programme précédent, c'est le calcul du 45^e terme de la suite qui est le gros consommateur. Après l'affichage de sa valeur, l'affichage de celles du 5^e et du 10^e est immédiat (bien que le programme que nous ayons écrit n'ait pas conservé leurs valeurs). Comparons ce temps d'exécution avec celui du programme itératif qu'on peut considérer comme naturel pour faire la même chose, c'est-à-dire conservant en mémoire les deux dernières valeurs calculées :

```
--> cat FiboIteratif.java
class FiboIteratif {
  static long fibo(int n) {
     int a = 1; // u(n-2)
     int b = 1; // u(n-1)
     int x = 0; // pour calculer u(n). Initialisation obligatoire pour le compilateur
     if (n == 0 || n == 1) return a;
     for( int i = 2; i <= n ; i++) {
         x = a + b;  // somme des deux dernieres valeurs calculees
         a = b;   // l'ancienne derniere valeur calculee devient l'avant derniere
         b = x;   // la valeur qu'on vient de calculer
         }
     return x;
  }
```

```
  public static void main(String[] args) {
     System.out.println("f4_ = " + fibo(4_));
     System.out.println("f_ = " + fibo(_));
     System.out.println("f10 = " + fibo(10));
  }
}
--> time java FiboIteratif
f4_ = 1836311903
f_ = 8
f10 = 89

real    0m0.186s
user    0m0.080s
sys     0m0.080s
-->
```

Les résultats sont suffisamment parlants pour qu'on s'interroge sur le bien-fondé de la solution récursive.

5.4.2 La programmation dynamique

Cette approche s'appuie sur l'approche récursive mais cherche à éviter de refaire des calculs déjà faits et donc coupe l'arbre des appels.
Le principe est simple : on mémorise les valeurs au fur et à mesure de leur calcul et avant de se lancer dans un nouveau calcul, on vérifie que ce qu'on doit calculer ne l'a pas encore été.
Cette approche conduit au programme suivant où un tableau externe est utilisé pour mémoriser les termes de la suite de Fibonacci au fur et à mesure qu'ils sont calculés.
Cela conduit au programme suivant :

```
--> cat Fibo2.java
class Fibo2 {
  static long[] valeurs; // tableau pour memoriser les valeurs
  static long fibo2(int n) {
    if (n == 0 || n == 1)
        return 1;
    if(valeurs[n] == 0)
        valeurs[n] = (long)(fibo2(n-1) + fibo2(n-2));
    return valeurs[n];
  }
  public static void main(String[] args) {
    valeurs = new long[100];
    System.out.println("f4_ = " + fibo2(4_));
    System.out.println("f_ = " + fibo2(_));
    System.out.println("f10 = " + fibo2(10));
  }
}
--> time java Fibo2
f4_ = 1836311903
f_ = 8
f10 = 89

real    0m0.193s
user    0m0.100s
sys     0m0.070s
-->
```

Le temps d'exécution du programme devient raisonnable et se rapproche de celui obtenu avec la version itérative. Il est néanmoins un peu plus lent : on peut attribuer la différence au coût de la gestion des appels de fonctions.

La même approche a été utilisée pour le programme de calcul des partages d'entiers dont l'exécution pour calculer `p(120, 120)` dans la version que nous avons donnée conduit au temps d'exécution suivant :

```
--> cat Partage1.java
    .......
    System.out.println("p(120, 120) = " + part(120, 120));
    .......
--> time java Partage1
p(120, 120) = 1844349_60

real    1m4.200s
user    1m4.110s
sys     0m0.090s
-->
```

Une version, dans laquelle les valeurs calculées sont mémorisées (pour simplifier, on a utilisé un tableau carré), est donnée ci-dessous et le temps d'exécution se passe de commentaires :

```
--> cat Partage2.java
class Partage2 {
  static long[][] tab;
  static long part(int n, int k) {
    if(tab[n][k] != 0) return tab[n][k];
    if (n == 0)
         return (tab[n][k] = 1);
    if (k == 0)
         return (tab[n][k] = 0);
    if (k > n)
         return (tab[n][k] = part(n, n));
    return (tab[n][k] = part(n - k, k) + part(n, k - 1));
  }
  public static void main(String[] args) {
    tab = new long[1_0][1_0];
    System.out.println("p(120, 120) = " + part(120, 120));
  }
}
--> time java Partage2
p(120, 120) = 1844349_60

real    0m0.184s
user    0m0.110s
sys     0m0.040s
-->
```

6

Implémentation et élimination de la récursion

Il est temps de voir comment tout cela marche et par là répondre à au moins deux questions :
a) récursion et itération ont-elles la même puissance d'expression ? En d'autres termes, tout ce qui peut être écrit en utilisant l'une peut-il l'être en utilisant l'autre ?
b) comment un programme exprimé de manière récursive fonctionne-t-il ?

6.1 Exprimer les itérations au moyen de la récursion

6.1.1 Principe général

Il n'y a rien de plus simple : quelques transformations du code permettent d'éliminer les itérations et de leur substituer des appels récursifs.

a) Supposons qu'un programme contienne une itération de la forme :

```
     ........
for(int i = 0; i < n; i++) corps;
     ........
```

Cette séquence peut être remplacée par un appel à une méthode récursive :

```
static void boucle(int i, int n) {
   if(i < n) { corps; boucle(i+1, n); }
}
     ........
// appel se substituant a la boucle
boucle(0,n);
    ........
```

b) Les choses peuvent cependant être légèrement compliquées par l'imbrication de boucles comme dans l'exemple suivant, mais quelques transformations simples fourniront une solution :

```
        ....  // avant les iterations
for(int i = i1; i < m; i++)
   for(int j = j1; j < n; j++)
        corps;
      ....  // apres les iterations
```

On commence par supprimer la boucle extérieure par le procédé décrit précédemment :

```
static void boucle1(int i, int m) {
  if(i < m) {
    for(int j = j1; j < n; j++) {
        corps; boucle1(i+1, n); }
  }
}
      ....  // avant
  boucle1(i1, m);    // appel equivalent a la boucle externe
      ....  // apres
```

La suppression de la boucle interne est globalement réalisée de la même manière si ce n'est que du fait de l'utilisation de la valeur de variable i dans le corps de la boucle, il faut transmettre la valeur de i en paramètre.
Cela conduit au code suivant :

```
// les variables i1, m, j1 et n sont externes
static int i1 = ....
static int m = ....
static int j1 = ....
static int n = ....
static void boucle2(int j, int n, int i) {
   if(j < n) {
       corps;
       boucle2(j+1, n, i); }
}
static void boucle1(int i, int m) {
   if(i < m) {
       boucle2(j1, n, i);
       boucle1(i+1, n); }
}
       .... // avant
   boucle1(i1, m);    // appel equivalent a la boucle externe
       .... // apres
```

6.1.2 Exemple complet

Considérons le programme Java suivant :

```
--> cat Boucle1.java
// programme avec boucles
class Boucle1 {
  static int f(int i, int j) {
     return 2 i + 3 j;
  }
  public static void main(String[] args) {
     for(int i = 2; i < 4; i++)
        for(int j = 1; j < 3; j++)
           System.out.print(f(i, j) + " ");
     System.out.println();
  }
}
-->
```

Par application des règles précédentes, on obtient le programme :

```
--> cat Boucle2.java
class Boucle2 {
  // programme equivalent a Boucle1 mais ecrit sans boucle
  static int f(int i, int j) {
    return 2 i + 3 j; }
  static void boucle2(int j, int n, int i) {
    if(j < n) {
      System.out.print(f(i, j) + " ");
      boucle2(j+1, n, i); }
  }
  static void boucle1(int i, int m) {
    if(i < m) {
      boucle2(1, 3, i);
      boucle1(i+1, m); }
  }
  public static void main(String[] args) {
    boucle1(2, 4);
    System.out.println();
  }
}
```

On peut vérifier que l'exécution de ces deux programmes conduit effectivement aux mêmes résultats :

```
--> java Boucle1
7 10 13 16 9 12 1 18
--> java Boucle2
7 10 13 16 9 12 1 18
-->
```

6.2 Récursion et pile

6.2.1 Introduction

Ce que nous avons fait pour implémenter les appels de fonctions simples en 3.2 s'applique aux définitions récursives.
Nous allons l'illustrer en reprenant un grand nombre des exemples que nous avons utilisés pour étudier le principe de la récursion.
Dans le cas d'une fonction définie récursivement, tous les blocs d'activation ont la même structure. C'est pourquoi nous utiliserons les piles typées `Stack<BlocActivation>` où `BlocActivation` est le type du bloc d'activation de la fonction.

6.2.2 La fonction factorielle

6.2.2.1 La définition récursive

Rappelons tout d'abord la définition de la fonction :

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

Nous utiliserons par ailleurs dans la traduction les repères suivants pour les instructions :

1. entrée dans la fonction ;
2. multiplication de `n` par la valeur renvoyée par `fact(n-1)` ;
3. sortie de la fonction en renvoyant sa valeur.

6.2.2.2 Le bloc d'activation

```
--> cat BlocFact.java
// Bloc d'activation utilisé pour la factorielle
class BlocFact {
  int n;        // le parametre
  int adresse;  // adresse (repere) de l'instruction
  long valeur;  // la valeur renvoyée
  BlocFact(int n, int adresse){
     this.n = n; this.adresse = adresse;
     }
}
```

6.2.2.3 Traduction

```
--> cat FactPile.java
import java.util.Stack;
// traduction avec une pile de la fonction récursive factorielle
class FactPile {
  static long factPile(int n) {
    Stack<BlocFact> pile = new Stack<BlocFact>(); // pile vide de références vers BlocFact
    long valeur = 0; // la dernière valeur retournée
    BlocFact bloc = new BlocFact(n, 1); // bloc d'activation de l'appel initial
    pile.push(bloc); // empilement de ce bloc
```

```
    while(! pile.empty()){
      bloc = pile.peek();  // on travaille sur le bloc au somment de la pile
      switch(bloc.adresse){
        case 1:  // un appel de la fonction
           if(bloc.n == 0){
               bloc.valeur = 1;   // on renvoie 1
               bloc.adresse = 3;   // on va sortir de la fonction
           }
           else {
               bloc.adresse = 2;  // il faudra multiplier n par la valeur renvoyée
               // on appelle f(n-1)  => nouveau bloc d'activation
               pile.push(new BlocFact(bloc.n -1, 1));}
           break;
        case 2:  // il faut multiplier le n courant par la valeur renvoyée
           bloc.valeur = bloc.n   valeur; // calcul de la valeur de retour
           bloc.adresse = 3; // on reprendre sur return
           break;
        case 3:
           valeur = bloc.valeur; // on renvoie la valeur
           pile.pop(); // on sort de la fonction => dépilement
      } // fin switch
    } // fin while
    return valeur;
  } // fin fonction factPile
  public static void main(String[] args){
    System.out.println("fact(0) = " + factPile(0));
    System.out.println("fact(_) = " + factPile(_));
    System.out.println("fact(10) = " + factPile(8));
  } // fin main
}  // fin class FactPile
--> java FactPile
fact(0) = 1
fact(_) = 120
fact(8) = 40320
-->
```

6.2.3 Somme de deux entiers

6.2.3.1 La forme récursive

Nous considérons la fonction de calcul récursif de la somme de deux nombres entiers suivante (basée sur l'idée que $n+m = [n+1]+[m-1]$), qui provoquereait un débordement pile pour un appel avec un second paramètre trop grand :

```
  static int add2Entiers(int x, int y){
    if(y == 0) return x;
    return add2Entiers(x + 1, y -1);
  }
```

6.2.3.2 Le bloc d'activation

Il correspond à la classe `BlocSomme` suivante :

```
--> cat BlocSomme.java
class BlocSomme {
  int x, y;     // les deux parametres
  int adresse;  // adresse (repere) de l'instruction
  long valeur;  // la valeur renvoyée
  BlocSomme(int x, int y, int adresse){
     this.x = x;  this.y = y; this.adresse = adresse;
  }
}
-->
```

Nous utiliserons par ailleurs les repères suivants (nous n'avons pas cherché à optimiser la traduction) :

1. entrée dans la fonction ;
2. retour de l'appel récursif ;
3. rerour de la fonction avc renvoi de la valeur.

6.2.3.3 La traduction

```
--> cat Somme2Entiers.java
import java.util.Stack;
class Somme2Entiers{
  static long somme(int x, int y) {
    Stack<BlocSomme> pile = new Stack<BlocSomme>();
    BlocSomme bloc = new BlocSomme(x, y, 1);
    pile.push(bloc);
    long valeur = 0;   // la derniere valeur renvoyee
    while(! pile.empty()) {
       bloc = pile.peek();
       switch(bloc.adresse) {
          case 1 : // appel de la fonction
             if(bloc.y == 0){ // y nul ==> on renvoie la valeur x
                bloc.valeur = bloc.x;
                bloc.adresse = 3; }
             else {  // y #0 ==> on fait un appel recursif avec x+1 et y-1
                bloc.adresse = 2; // il faudra renvoyer la valeur calculee
                // preparer l'appel recursif
                pile.push(new BlocSomme(bloc.x+1, bloc.y-1, 1));}
             break;
          case 2 :
             bloc.valeur = valeur; // sauvegarder la valeur
             bloc.adresse = 3;
             break;
          case 3 : // preparer la valeur de retour et depiler
             valeur = bloc.valeur;
             pile.pop();
             break;
        }
      }
      return valeur;
    }
    public static void main(String[] args) {
    System.out.println("3␣ + 24 = " + somme(3␣, 24));
  }
}
--> java Somme2Entiers
3␣ + 24 = ␣9
-->
```

6.2.4 La suite de Fibonacci

Avec cet exemple, il est nécessaire de combiner des valeurs calculées antérieurement.

6.2.4.1 La version récursive

```
static long fibo(int n) {
   if (n == 0 || n == 1) return 1;
   return fibo(n-1) + fibo(n-2);
}
```

6.2.4.2 Le bloc d'activation

Le bloc d'activation de la fonction est de la forme :

```
--> cat BlocFibo.java
class BlocFibo {
  int n;       // le parametre
  long valeur; // la valeur de la fonction
  int adresse;
  BlocFibo (int n, int adresse) { this.n = n; this.adresse = adresse; }
}
-->
```

Les repères d'instructions que nous utiliserons sont les suivants :

1. entrée dans la fonction ;
2. mémoriser la valeur renvoyée par $fibo(n-1)$ et appeler $fibo(n-2)$;
3. ajouter et mémoriser la somme des deux appels ;
4. sortie de la fonction avec renvoi de la valeur.

6.2.4.3 Traduction

```
--> cat FiboPile.java
import java.util.Stack;
class FiboPile {
  static long fibo(int n){
     Stack<BlocFibo> pile = new Stack<BlocFibo>();
     BlocFibo bloc = new BlocFibo(n, 1);
     long valeur = 0; pile.push(bloc);
     while(! pile.empty()){
       bloc = pile.peek();
       switch(bloc.adresse){
         case 1 :  // entrée dans la fonction
           if(bloc.n == 0 || bloc.n == 1) { bloc.valeur = 1; bloc.adresse = 4; }
           else { bloc.adresse = 2; pile.push(new BlocFibo(bloc.n-1, 1)); }
           break;
         case 2 : // retour du calcul de fibo(n-1)
             bloc.valeur = valeur; bloc.adresse = 3;
             pile.push(new BlocFibo(bloc.n-2, 1)); // calculer fibo(n-2)
             break;
         case 3 : // retour du calcul de fibo(n-2)
             bloc.valeur = bloc.valeur + valeur;  // somme de fibo(n-1) et fibo(n-2)
             bloc.adresse = 4;
             break;
         default :  // case 4 : renvoyer la valeur
             valeur = bloc.valeur;
             pile.pop();
        } // fin du switch
    } // fin du while
    return valeur;
  } // fin de la fonction fibo
  public static void main(String[] args) {
     System.out.println(fibo(1)); System.out.println(fibo(2));
     System.out.println(fibo(4)); System.out.println(fibo(6));
     System.out.println(fibo(1_)); }
}
--> java FiboPile
1
2
_
13
987
-->
```

6.2.5 Les tours de Hanoi

6.2.5.1 La version récursive

```
static void hanoi(int n, char origine, char destination, char intermediaire) {
// fonction pour deplace n disques d'une origine a une destination
   switch(n)  {
      case 0  : return ;  // pas atteint sauf pour un appel initial avec n = 0
      case 1  : System.out.println(origine + " --> " + destination);
                break;
      default : hanoi(n - 1, origine, intermediaire, destination);
                System.out.println(origine + " --> " + destination);
                hanoi(n - 1, intermediaire, destination, origine);
   }
}
```

6.2.5.2 Le bloc d'activation

```
--> cat BlocHanoi.java
public class BlocHanoi {
  int n; char a, b, c; int adresse;
  public BlocHanoi(int n, char a, char b, char c, int adresse) {
      this.n = n; this.a = a; this.b = b; this.c = c; this.adresse = adresse;}
}
```

Les repères utilisés pour les instructions sont :

1. appel de la fonction ;
2. retour du premier appel récursif ;
3. retour du second appel récursif et donc, en fait, retour de l'appel courant.

6.2.5.3 La traduction

```
--> cat HanoiPile.java
import java.util.Stack;
class HanoiPile {
  static void hanoiPile(int n, char a, char b, char c) {
     Stack<BlocHanoi> pile = new Stack<BlocHanoi>();
     BlocHanoi bloc;
     pile.push(new BlocHanoi(n,a,b,c,1));
     while(! pile.empty()) {
        bloc = pile.peek();
        switch(bloc.adresse) {
          case 1 :
            if(bloc.n == 1) { System.out.print(bloc.a + " -> " + bloc.b + "    "); bloc.adresse = 3; }
            else { bloc.adresse = 2; pile.push(new BlocHanoi(bloc.n - 1, bloc.a, bloc.c, bloc.b, 1)); }
            break;
          case 2 :
             System.out.print(bloc.a + " -> " + bloc.b + "    ");
             bloc.adresse = 3;
             pile.push(new BlocHanoi(bloc.n - 1, bloc.c, bloc.b, bloc.a, 1));
             break;
          default : // case 3 (en fait return)
             pile.pop();
        } // fin switch
     } // fin while
     System.out.println();
  } // fin fonction
  public static void main(String[]  args) {
     hanoiPile(3, 'A', 'B', 'C');
  }
}
```

```
--> java HanoiPile
A -> B    A -> C    B -> C    A -> B    C -> A    C -> B    A -> B
-->
```

Les états successifs de la pile et les instructions/appels correspondant à cette exécution sont décrits ci-après :

états successifs de la pile	appel ou instruction
`[`	
	appel initial : entrée dans la fonction
`[(3,'A','B','C',1)`	
	au retour du 1er appel $f(3-1,...)$
`[(3,'A','B','C',2)`	
	appel de $f(3-1,'A','C','B')$
`[(3,'A','B','C',2)(2,'A','C','B',1)`	
	au retour du 1er appel $f(2-1,...)$
`[(3,'A','B','C',2)(2,'A','C','B',2)`	
	appel de $f(2-1,'A','B','C')$
`[(3,'A','B','C',2)(2,'A','C','B',2)(1,'A','B','C',1)`	
	affichage de `A --> B`
`[(3,'A','B','C',2)(2,'A','C','B',2)(1,'A','B','C',3)`	
	`return`
`[(3,'A','B','C',2)(2,'A','C','B',2)`	
	affichage de `A --> C`
	au retour du 2^{nd} appel $f(2-1,...)$
`[(3,'A','B','C',2)(2,'A','C','B',3)`	
	appel de $f(2-1,'B','C','A')$
`[(3,'A','B','C',2)(2,'A','C','B',3)(1,'B','C','A',1)`	
	affichage de `B --> C`
`[(3,'A','B','C',2)(2,'A','C','B',3)(1,'B','C','A',3)`	
	`return`
`[(3,'A','B','C',2)(2,'A','C','B',3)`	
	`return`
`[(3,'A','B','C',2)`	
	affichage de `A --> B`
	au retour du 2^{nd} appel $f(3-1,...)$
`[(3,'A','B','C',3)`	
	appel de $f(3-1,'C','B','A')$
`[(3,'A','B','C',3)(2,'C','B','A',1)`	
	au retour du 1er appel $f(2-1,...)$
`[(3,'A','B','C',3)(2,'C','B','A',2)`	
	appel de $f(2-1,'C','A','B')$
`[(3,'A','B','C',3)(2,'C,'B','A',2)(1,'C','A','B',1)`	
	affichage de `C --> A`
`[(3,'A','B','C',3)(2,'A','C','B',2)(1,'C','A','B',3)`	
	`return`
`[(3,'A','B','C',3)(2,'C','B','A',2)`	
	affichage de `C --> B`
	au retour du 2^{nd} appel $f(2-1,...)$
`[(3,'A','B','C',3)(2,'C','B','A',3)`	
	appel de $f(2-1,'A','B','C')$
`[(3,'A','B','C',3)(2,'C','B','A',3)(1,'A','B','C',1)`	
	affichage de `A --> B`
`[(3,'A','B','C',3)(2,'C','B','A',3)(1,'A','B','C',3)`	
	`return`
`[(3,'A','B','C',3)(2,'C','B','A',3)`	
	`return`
`[(3,'A','B','C',3))`	
`[`	pile vide : c'est fini

6.2.6 Le baguenaudier

6.2.6.1 La solution récursive

```
static void bag(int n) {
  if(n == 0) return;
  if(n == 1) { System.out.print("-1  "); return; }
  bag(n-2); System.out.print("-" +  n + "  ");
  debag(n-2); bag(n-1);
}
static void debag(int n) { poser les n pions
  if(n == 0) return;
  if(n == 1) { System.out.print("+1  "); return; }
  debag(n-1); bag(n-2);
  System.out.print("+" + n + "  "); debag(n-2);
}
```

6.2.6.2 Le bloc d'activation

Nous allons remplacer le jeu de deux procédures par une seule dont le bloc d'activation aura la structure suivante :

```
--> cat BlocBag.java
class BlocBag {
  int n, adresse;
  BlocBag(int n, int adresse) { this.n = n; this.adresse = adresse; }
}
```

Les valeurs de *adresse* utilisées dans la traduction sont les suivantes :

1. correspond à un appel de la fonction `bag` ;
2. retour de l'appel `bag(n-2)` dans `bag` ;
3. retour de l'appel `debag(n-2)` dans `bag` ;
4. correspond à un appel de la fonction `debag` ;
5. retour de l'appel `debag(n-1)` dans `debag` ;
6. retour de l'appel `bag(n-2)` dans `debag` ;
7. `return` dans l'une des deux fonctions (soit sur l'une des conditions d'arrêt ou un `return` implicite après `bag(n-1)` dans `bag` ou après `debag(n-2)` dans `debag`).

6.2.6.3 La traduction

```
--> cat BagPile.java
import java.util.Stack;
class BagPile {
  static void bagPile(int n) {
    Stack<BlocBag> pile = new Stack<BlocBag>();
    BlocBag bloc = new BlocBag(n, 1);
    pile.push(bloc);
    while(! pile.empty()){
      bloc = pile.peek();
      switch(bloc.adresse) {
        case 1 : // appel de bag
          if(bloc.n == 0) bloc.adresse = 7;
          else if (bloc.n == 1){
            System.out.print("-1 "); bloc.adresse = 7;}
          else {
            bloc.adresse = 2;
            pile.push(new BlocBag(bloc.n - 2, 1));}
          break;
        case 2 : // au retour de bag(n-2) dans bag
          System.out.print("-" + bloc.n + " ");
          bloc.adresse = 3;
          pile.push(new BlocBag(bloc.n - 2, 4));  // fin switch
          break;
```

```
        case 3 : // au retour de debag(n-2) dna sbag
          bloc.adresse = 7;
          pile.push(new BlocBag(bloc.n - 1, 1));  // fin switch
          break;
        case 4 : // appel de debag
          if(bloc.n == 0) bloc.adresse = 7;
          else if (bloc.n == 1){
            System.out.print("+1 "); bloc.adresse = 7;}
          else {
            bloc.adresse = ¬;
            pile.push(new BlocBag(bloc.n - 1, 4));}
          break;
        case ¬: // au retour de bag(n-2) dans debag
          bloc.adresse = 6;
          pile.push(new BlocBag(bloc.n - 2, 1));  // fin switch
          break;
        case 6 : // au retour de debag(n-2) dans debag
          System.out.print("+" + bloc.n + " ");
          bloc.adresse = 7;
          pile.push(new BlocBag(bloc.n - 2, 4));  // fin switch
        case 7 : // return dans bag ou debag
          pile.pop();
      } // fin switch
    } // fin while
  } // fin bagNR
  public static void main(String[] arg) {
     bagPile(4); System.out.println(); }
}
--> java BagNonRec
-2 -1 -4 +1 +2 -1 -3 +1 -2 -1
-->
```

6.2.7 Réduire la taille de la pile

Nous allons commencer par voir comment il est possible de transformer certains programmes récursifs en des programmes dans lesquels il n'y a aucun empilement des données.
On ne va pas éliminer complètement la récursion mais remplacer les appels avec paramètres par des appels sans param'etres.
Ce faisant, on va économiser de la place mémoire.
Supposons qu'une fonction `fRec` comportant un appel récursif soit définie de la manière suivante :

```
static T1 fRec(T2 x) {
   T1 val;
   .....  // sequence A de code quelconque
   ... fRec(f(x));
   .....  // sequence B de code quelconque
   return val; }
```

dans laquelle la fonction f est inversible, c'est-à-dire telle qu'il existe une fonction g telle que $\forall x \in T2,\ g(f(x)) = x$.
La définition de la fonction peut être remplacée par la suivante :

```
static int n;  // variable externe (initialisee avant l'appel)
static T1 fRec2() {  // fonction sans parametre
   T1 val;
   .....  // sequence A de code quelconque
   n = f(n); // preparer la valeur du parametre
   ... fRec(); // occurrences de n remplacees par g(n) sur cette ligne
   n = g(n); // retablir la valeur
   .....  // sequence B de code quelconque
   return val; }
```

Pour illustrer ce principe, reprenons tout d'abord la définition récursive de la fonction factorielle :

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

Nous commençons par la modifier pour qu'elle ait la forme voulue :

```
static long fact1(int n){
long val;
if(n == 0)
   val = 1;
else
   val = n   fact1(n - 1); // ligne concernee
return val; }
```

En appliquant la transformation sur la ligne repérée, on obtient le code suivant :

```
--> cat Fact2.java
class Fact2 {
  static int n; // n est une variable externe
  static long fact2() { // fonction sans parametre
    long val;
    if(n == 0)
       val = 1;
    else {
       n = n-1; // calcul nouvelle valeur de valeur de n pour l'appel recursif
       val = (n+1)   fact2(); // n remplacee par (n+1) inverse (n-1)
       n = n + 1; // retablissement de la valeur de n
    }
    return val;
  }
  public static void main(String[] arg) {
  n = _; // initialisation a _ de la variable externe
  System.out.println("fact(_) = " + fact2());
  n = 10; // initialisation a 10 de la variable externe
  System.out.println("fact(10) = " + fact2());
  }
}
--> java Fact2
fact(_) = 120
fact(10) = 3628800
-->
```

Par un procédé de même nature, on peut transformer le programme des tours de Hanoi en éliminant les paramètres dans les appels récursifs. L'élimination du paramètre n et son remplacement par une variable externe est simple :

```
--> cat Hanoi1.java
static int n;
static void hanoi1(char a, char b, char c) {
  if (n > 0) {
    n = n - 1; hanoi1(a, c, b);
    n = n + 1;   // peut être omis
    System.out.println(a + " --> " + b);
    n = n - 1;   // omis en même temps que le n = n + 1 du dessus
    hanoi1(c, b, a); n = n + 1;
}
-->
```

Les paramètres a, b et c peuvent également être omis en procédant à des échanges corrects de valeurs des variables a, b et c externes :

- il faut échanger b et c avant le premier appel et refaire la même opération après cet appel (l'opération est sa propre inverse) ;
- il faut échanger a et c avant le second appel et refaire la même opération après cet appel (l'opération est sa propre inverse).

Cela conduit au code suivant :

```
--> cat Hanoi2.java
class Hanoi2 {
  static int n;
  static char a, b, c;
  static char x; // pour les échanges de valeurs
  static void hanoi2() {
    if (n > 0) {
      n = n - 1;
      x = b; b = c; c =x; // échanger les valeurs de b et c
      hanoi2();
      x = b; b = c; c =x; // échanger les valeurs de b et c
      // on omet n = n+1
      System.out.println(a + " --> " + b);
      // on omet n = n - 1;
      x = a; a = c; c =x; // échanger les valeurs de a et c
      hanoi2();
      x = a; a = c; c =x; // échanger les valeurs de a et c
      n = n + 1;
    }
  }
  public static void main(String[] arg) {
    n = 3;
    a = 'A'; b = 'B'; c = 'C';
    hanoi2();
  }
}
--> java Hanoi2
A --> B
A --> C
B --> C
A --> B
C --> A
C --> B
A --> B
-->
```

6.2.8 Récursion terminale. Se passer de la pile

Nous avons vu comment simuler la récursion au moyen d'une pile et comment, sous certaines conditions limiter la quantité d'informations à empiler.
On peut se demander s'il est possible dans certaines applications de remplacer la récursion par des itérations simples sans utiliser de pile .
C'est le cas pour les **récursions terminales** (*tail-recursion*). Une récursion est dite terminale si l'appel récursif est la dernière instruction exécutée dans la fonction ou procédure appelante. Autrement dit, au retour d'un tel appel récursif, on sort immédiatement de la fonction ou procédure ayant fait cet appel.

6.2.8.1 Exemple introductif : la factorielle

Reprenons la version récursive classique de la fonction factorielle :

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

Cette définition comporte un appel récursif qui n'est pas en position terminale (bien qu'il apparaisse en dernière position dans le code) : en effet au retour de l'appel $f(n-1)$, on doit encore multiplier la valeur renvoyée par cet appel par la valeur de n avant de sortir de la fonction.

Le calcul de fact(4) correspond à la séquence :

```
fact(4)
= 4   fact(3)
= 4   (3   fact(2))
= 4   (3   (2   fact(1))
= 4   (3   (2   (1   fact(0))
= 4   (3   (2   (1   1)
= 4   (3   (2   1)
= 4   (3   2)
= 4   6
= 24
```

Ainsi que nous l'avons vu, ce calcul impose l'empilement des environnements (les valeurs de n, les valeurs et adresses de retour).
La fonction factorielle peut être récrite récursivement mais de telle sorte qu'elle ne comporte que des récursions terminales (en fait il n'y en a qu'une seule).
Pour cela, on définit une fonction `factAuxiliaire` comportant deux paramètres : le premier est l'entier n dont on veut calculer la factorielle et le second est la valeur de (n-1) !.

```
--> cat FactRecTerm.java
class FactRecTerm {
  // la fonction factorielle de service cachee a l'exterieur
  // de la classe par la specification private
  private static long factAuxiliaire(int n, long valeur) {
    if (n == 0)
      return valeur;
    return factAuxiliaire(n - 1, n   valeur);
  }
  // la fonction factorielle qui sera appelable par un utilisateur
  // sous le nom FactRecTerm.fact
  public static long fact(int n) {
    return factAuxiliaire(n, 1);
  }
}
--> cat CalculFact.java
class Calculfact {
  public static void main(String[] arg) {
    // un appel a factAuxiliaire provoquerait une erreur de compilation
    System.out.println("fact(0) = " + FactRecTerm.fact(0));
    System.out.println("fact(10) = " + FactRecTerm.fact(10));
  }
}
--> java CalculFact
fact(0) = 1
fact(10) = 3628800
-->
```

Le calcul de fact(4) correspondrait maintenant à la séquence :

```
fact(4)
= fAuxiliaire(4, 1)
= fAuxiliaire(3, 4)
= fAuxiliaire(2, 12)
= fAuxiliaire(1, 24)
= fAuxiliaire(0, 24)
= 24
```

Il n'est donc pas nécessaire a priori d'empiler tous les environnements : une fois qu'on a fait l'appel $f(n-1, n*valeur)$, on sait qu'on n'aura plus besoin des valeurs que n et $valeur$ avaient au moment de cet appel. Il suffit donc a priori de disposer d'un seul couple de variables n et val pour exécuter ce programme.

Il est cependant intéressant de noter que les langages impératifs tels par exemple que C ou Java n'utilisent pas cette optimisation : une récursion terminale y utilise inutilement la pile. Nous avons déjà donné la preuve avec la version récursive de l'addition de deux entiers. La fonction utilise une récursion terminale, donc a priori ne nécessitant pas de pile mais l'addition d'un entier avec par exemple 10000000 provoque un débordement de pile.
Par contre les langages fonctionnels tels que Scheme ou CAML reconnaissent ce type de récursion et optimisent leur exécution.

6.2.8.2 Une généralisation est-elle possible ?

Il est bien évidemment tentant d'essayer de généraliser cette transformation d'une récursion non terminale en une récursion terminale.
Cela est souvent possible, mais pas toujours.
Le principe consiste à ajouter un ou plusieurs paramètres utilisés pour construire le résultat. L'idée est donc d'incorporer les opérations pendantes dans la fonction appelante (et qui devront être exécutées au retour de l'appel récursif) dans la valeur du paramètre auxiliaire. La définition d'une fonction auxiliaire appelée par la fonction permet de cacher aux utilisateurs ces détails.

6.2.8.3 Fibonacci en récursion terminale

Pour finir, transformons la version récursive de la fonction calculant le n-ème terme de la suite de Fibonacci en une version récursive terminale.
Compte tenu du fait que le calcul de $f(n)$ nécessite les valeurs de $f(n-1)$ et de $f(n-2)$, nous allons utiliser une fonction auxiliaire possédant, en plus du paramètre n, deux paramètres permettant l'accumulation des valeurs au fur et à mesure :

```
--> cat FiborecTerm.java
class FiborecTerm {
  private static long fibAuxiliaire(int n, long suivant, long valeur) {
    if (n == 0) return valeur;
    else return fibAuxiliaire(n - 1, suivant + valeur, suivant); }
  public static long fibo(int n) {
    return fibAuxiliaire(n, 1, 1);
  }
}
--> CalculFibo.java
class CalculFibo {
  public static void main(String[] args) {
    System.out.println("fibo(␣) = " + FiborecTerm.fibo(␣));
    System.out.println("fibo(10) = " + FiborecTerm.fibo(10));
  }
}
--> java CalculFibo
fibo(␣) = 8
fibo(10) = 89
-->
```

7 Backtracking

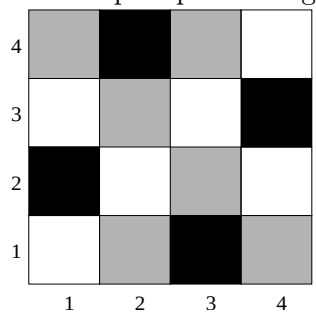
7.1 Introduction

Il s'agit d'une technique plutôt brutale de résolution de problèmes de nature combinatoire. Elle consiste en une exploration systématique de toutes les configurations possibles d'un espace fini totalement ordonné, donc représentable par un arbre fini mais en procédant de telle sorte que le rejet d'une configuration entraine automatiquement celui d'un ensemble d'autres.
Ce n'est rien d'autre que l'approche adoptée pour sortir d'un labyrinthe : avancer aussi loin que possible et revenir sur ses pas quand on est bloqué et essayer alors un autre chemin.
Un champ d'applications privilégié (mais il y en a d'autres) en est la résolution de jeux.
Après un exemple introductif choisi parmi les grands classiques, nous proposerons un modèle de problèmes et de leurs solutions auxquels cette technique s'applique, en donnerons des schémas généraux récursif et itératif et enfin traiterons quelques exemples.

7.2 Exemple introductif

7.2.1 Le problème

On souhaite placer sur un échiquier de $n \times n$ cases (nous illustrerons pour le cas $n = 4$) n reines sans qu'aucune d'elles ne soit en prise : une reine peut en prendre une autre placée sur la même ligne, la même colonne ou sur une même diagonale qu'elle.
La figure suivante donne une solution du problème pour $n = 4$ (les cases noires sont celles où on place une reine) et la numérotation adoptée pour les lignes et les colonnes de l'échiquier :



Une position sur l'échiquier correspond à un couple (l, c) où $1 \leq l, c \leq n$, l désignant une ligne et c désignant une colonne.
Si on place n reines sur un tel échiquier (à des emplacements différents et pas nécessairement dans une configuration constituant une solution du problème), on peut représenter la disposition des n reines par un ensemble de n couples.
Ainsi la configuration des 4 reines de la figure peut être représentée par :

```
{(2,1), (4,2), (1,3), (3,4)}
```

Puisque dans le problème qui nous intéresse, pour une solution il ne peut y avoir deux reines sur la même colonne, on peut adopter une représentation simplifiée en ne gardant que les indices de lignes, avec la convention qu'ils apparaissent par numéro de colonne croissant.
Pour la configuration donnée en exemple, on obtient `[2 4 1 3]`.

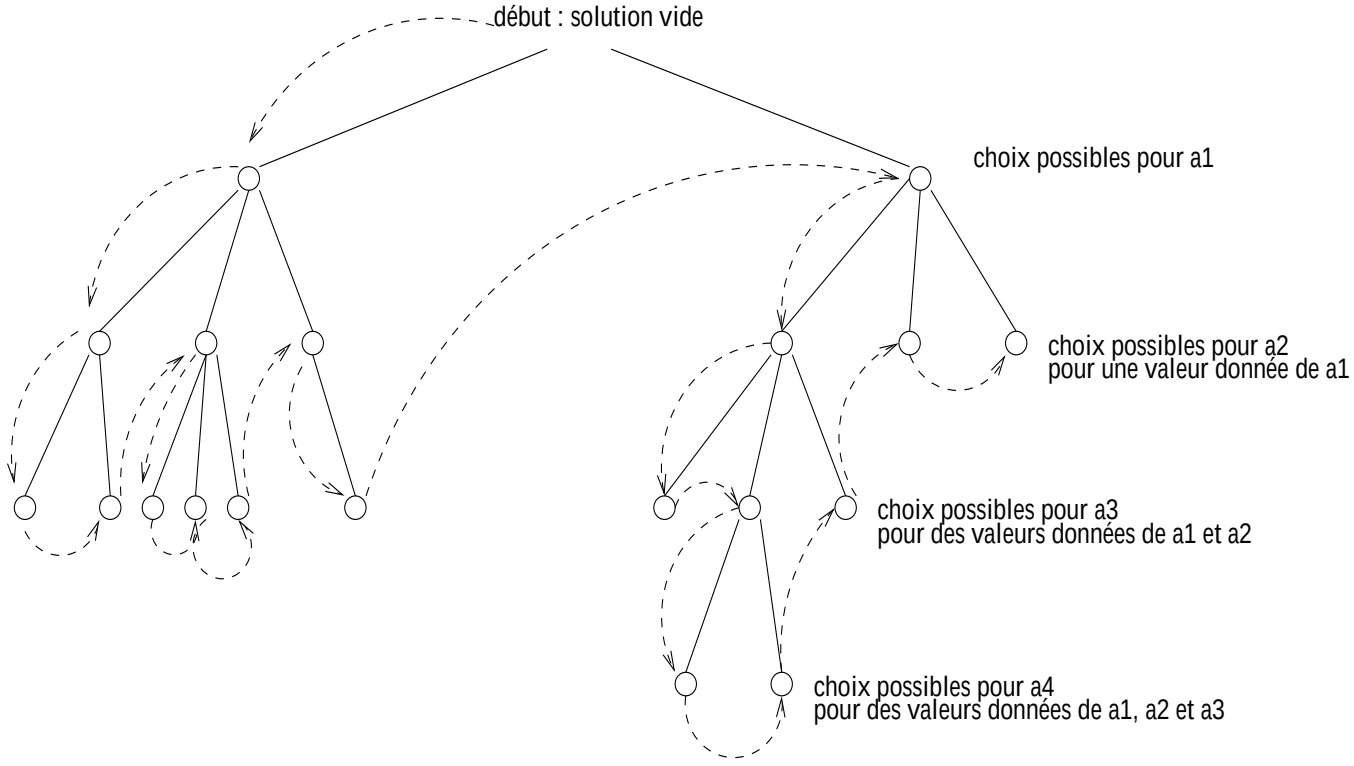
Une solution correspond nécessairement à une permutation de l'ensemble `[1 2 ...n]`

Le problème des n reines précédent en est un cas particulier : la longueur de la solution fait partie de la condition (la solution est de longueur n). Le reste de la condition est que tous les éléments du vecteur sont différents et que la condition *diag* est satisfaite.

Pour résoudre le problème on part du vecteur vide `[ ]` qui constitue donc la solution partielle initiale. Les contraintes imposées par la satisfaction de la condition $\mathcal{C}$ définissent un sous-ensemble $\mathtt{S}_1$ de $\mathtt{A}_1$ contenant toutes les valeurs acceptables pour a_1 (on a très souvent $\mathtt{S}_1 = \mathtt{A}_1$). On choisit le plus petit élément de $\mathtt{S}_1$ (il y en a un puisque $\mathtt{A}_1$ est totalement ordonné). Cela donne la solution partielle $[a_1]$ de rang 1. Si elle n'est pas une solution totale, on va essayer de l'allonger en une solution de rang 2. Pour cela, on calcule le sous-ensemble $\mathtt{S}_2$ de $\mathtt{A}_2$ contenant toutes les valeurs acceptables pour a_2 et on construit la solution partielle $[a_1 a_2]$ en prennant pour a_2 la plus petit élément de $\mathtt{S}_2$.

De manière générale, étant donnée une solution partielle $[a_1 a_2 \dots a_k]$, de rang k et supposée non solution totale du problème, les contraintes imposées par les valeurs a_1, ..., a_k entrant dans la composition de cette solution partielle conduisent à un ensemble $\mathtt{S}_{k+1} \subset \mathtt{A}_{k+1}$ de valeurs candidates pour a_{k+1} de telle sorte que $[a_1 a_2 \dots a_k a_{k+1}]$ soit une solution (partielle ou totale) de rang $k+1$.
Si à instant donné l'ensemble $\mathtt{S}_{k+1}$ est l'ensemble vide $\emptyset$, le principe est de revenir en arrière (*backtrack*) : on essaie alors pour la composante a_k la valeur suivante (dans $\mathtt{S}_k$) de celle qu'on vient d'essayer et s'il n'y en a pas on continue de revenir en arrière.

Ce principe général peut être représenté par un arbre de recherche des solutions qui est parcouru en profondeur (ordre préfixe) :



7.3.2 Notations utilisées

Dans les différentes descriptions du schéma général de la recherche d'une solution par la technique de *backtracking*, nous utiliserons les notations suivantes :

- $\mathtt{V} = [\mathtt{V}_1 \ldots]$ est un vecteur représentant une solution partielle de rang k et on note :
  - ◇ $\mathtt{A}_i$ l'ensemble totalement ordonné des valeurs théoriquement possibles pour $\mathtt{V}_i$. On notera $\mathtt{A}_i[j]$ le j-ème élément de $\mathtt{A}_i$ dans l'ordre sur cet ensemble ;
  - ◇ si $\mathtt{V} = [\mathtt{V}_1 \ldots \mathtt{V}_m]$ alors $\mathtt{V} \circ x = [\mathtt{V}_1 \ldots \mathtt{V}_m\ x]$;
  - ◇ $\overline{\mathtt{V}}$ est la solution partielle de longueur $|\mathtt{V}| - 1$ obtenue en supprimant la dernière composante de $\mathtt{V}$. Donc si $\mathtt{V} = [\mathtt{V}_1 \ldots \mathtt{V}_{n-1}\ \mathtt{V}_n]$, alors $\overline{\mathtt{V}} = [\mathtt{V}_1 \ldots \mathtt{V}_{n-1}]$. Par convention $\overline{\emptyset} = \emptyset$
- $\mathcal{C}$ désigne la condition correspondant aux solutions totales du problème : donc, $[\mathtt{V}_1 \ldots \mathtt{V}_n]$ est une solution si et seulement si $\mathcal{C}([\mathtt{V}_1 \ldots \mathtt{V}_n])$ est vraie.
  Nous nous limiterons aux cas où une solution totale ne peut être solution partielle d'une autre solution totale (c'est-à-dire que si $([\mathtt{V}_1 \ldots \mathtt{V}_n])$ est solution totale alors il n'existe pas de solution totale de rang $> n$ de la forme $[\mathtt{V}_1 \ldots \mathtt{V}_n \ldots]$).
  On peut aussi noter que pour certains problèmes, toutes les solutions totales ont même longueur (c'est le cas pour le problème des n reines), ce n'est pas nécessairement le cas. Un exemple en est le partage comme somme d'entiers d'un nombre.
  Une solution partielle de rang m ne satisfait la condition $\mathcal{C}$ que si elle est une solution totale et sinon elle satisfait une certaine condition $\mathcal{C}_m$. Ainsi, pour le problème des n reines, ce qui distingue une solution partielle d'une solution totale est qu'une solution partielle est de longueur $< n$ et une solution totale est de longueur n (sinon elle vérifie des conditions de même nature qu'une solution totale [condition *diag* et pas deux éléments identiques]) ;
- $\mathtt{S}_m$ est le sous-ensemble (totalement ordonné) de $\mathtt{A}_m$ contenant les valeurs acceptables pour $\mathtt{V}_m$ compte tenu des valeurs actuelles de $\mathtt{V}_1$, …, $\mathtt{V}_{m-1}$ et les contraintes fixées par la condition $\mathcal{C}$.
  Ces ensembles pourront ou non être effectivement calculés mais le plus souvent ils ne le seront pas : un algorithme de calcul du suivant d'un élément dans un tel ensemble est suffisant.

7.3.3 Forme récursive du backtracking

7.3.3.1 Description

Les algorithmes de recherche par *backtracking* s'exprime de manière très simple sous forme récursive. Le *backtracking* est alors caché par la récursion : il est réalisé automatiquement par l'implantation de cette récursion (au moyen d'une pile comme nous l'avons vu).
La version donnée ci-dessous calcule toutes les solutions d'un problème (en supposant qu'une solution totale n'est partielle d'aucune autre). Si on n'en veut qu'une seule, il suffit évidemment de terminer l'algorithme après avoir trouvé la première.

Réaliser l'appel initial *backtrack(* `[], 1` *)*

backtrack(`V, rang` *)* est défini par :

si `V` est une solution
  l'exploiter (par exemple l'imprimer ou l'enregistrer)

sinon
  calculer $\mathtt{S}_{\mathtt{rang}}$
  pour chaque $x \in \mathtt{S}_{\mathtt{rang}}$ *faire*
    backtrack(`V` $\circ\ x$, `rang+1` *)*

Afin de minimiser le coût en espace dû en particulier à la transmission des paramètres et contourner la difficulté (par exemple en Java) de transmettre un paramètre de la forme $\mathtt{V} \circ x$, on peut utiliser une variable globale comme nous l'avons déjà fait, ce qui conduit à la forme suivante :

V ← []
Réaliser l'appel *backtrack(1)*
*backtrack(*rang*)* est défini par :
si V est une solution
    l'exploiter (par exemple l'imprimer ou l'enregistrer)
sinon
    calculer S_{rang}
    pour chaque $x \in S_{rang}$ *faire*
        V ← V ∘ x
        *backtrack(*rang+1*)*
        V ← $\overline{V}$

7.3.3.2 Exemple : les n reines sur un échiquier

Dans le programme suivant, on utilise la classe `Vector` du *package* `java.util.Vector` dans sa forme générique pour construire la solution. Un tel objet est une liste d'objets.
Dans la séquence suivante,

```
static Vector<Integer> solution;
      .....
solution = new Vector<Integer>();
```

la première ligne déclare une variable de référence pour un vecteur destiné à contenir des objets de type `Integer` et la dernière crée un tel vecteur vide.
On utilise par ailleurs les méthodes suivantes applicables à un objet de type `Vector` :
- `size()` : renvoie le nombre d'éléments de la liste définie par l'objet ;
- `contains(x)` : renvoie `true` si x appartient à la liste ;
- `get(i)` : renvoie l'élément en position i dans la liste (le premier élément est en position 0) ;
- `removeElement(i)` : supprime l'élément en position i dans la liste ;
- `add(x)` : ajoute l'objet x en fin de liste.

Enfin, la fonction `println` affiche les objets de la classe `Vector` de manière agréable (entre crochets et séparés par des virgules).
Dans le programme suivant, on calcule effectivement les ensembles des positions possibles dans une colonne au moyen de la fonction `positions`.

```
    }
    return lignes;
  }
  static void reinesRec(int n, int col) {
    Vector<Integer> valeurs;
    if(col == n){
      nombreSol ++;
      if (first){
        System.out.println("     Premiere solution : " + solution);
        first = false; }
    }
    else {
      valeurs = positions(n, col);
         for(int i = 0; i < valeurs.size(); i++) {
         solution.add(valeurs.get(i));
         reinesRec(n, col + 1);
         solution.removeElementAt(solution.size() - 1);
      }
    }
  }
  public static void main(String[] args) {
    for(int taille = 4; taille <= 9; taille++)  {
      nombreSol = 0;
      first = true;
      solution = new Vector<Integer>();
      System.out.println("reines(" + taille + ") : ");
      reinesRec(taille, 0);
      System.out.println("     " + nombreSol + " solutions");
      System.out.println("--------------------------");
    }
  }
}
--> java ReinesRec
reines(4) :
     Premiere solution : [2, 4, 1, 3]
     2 solutions
--------------------------
reines(5) :
     Premiere solution : [1, 3, 5, 2, 4]
     10 solutions
--------------------------
reines(6) :
     Premiere solution : [2, 4, 6, 1, 3, 5]
     4 solutions
--------------------------
reines(7) :
     Premiere solution : [1, 3, 5, 7, 2, 4, 6]
     40 solutions
--------------------------
reines(8) :
     Premiere solution : [1, 5, 8, 6, 3, 7, 2, 4]
     92 solutions
--------------------------
reines(9) :
     Premiere solution : [1, 3, 6, 8, 2, 4, 9, 7, 5]
     352 solutions
--------------------------
reines(10) :
     Premiere solution : [1, 3, 6, 8, 10, 5, 9, 2, 4, 7]
     724 solutions
--------------------------
-->
```

7.3.4 Forme non récursive du backtracking

7.3.4.1 Description

Dans la première expression récursive du *backtracking* les appels sont en position terminale et par conséquent il est donc naturel d'en obtenir une version non récursive.

```
rang ← 1
Srang ← A1   // en supposant que toutes les valeurs de A1 sont acceptables
nombreSol ← 0
tant que rang > 0 faire
    tant que Srang ≠ ∅ faire
        x ← Srang[1] // le 1er élément de Srang
        Srang ← Srang − {x}
        V ← V ∘ x        // le rajouter à la fin de V
        si C(V)
            // V est une solution : l'exploiter (l'imprimer par exemple)
            nombreSol ← nombreSol + 1
            si on veut toutes les solutions
                rang ← rang - 1 // retour en arrière
                V ← V̄
                continuer
            sinon
                arrêter
        sinon
            // on va essayer d'allonger la solution partielle au rang supérieur
            rang ← rang + 1 // passage au rang supérieur
            Calculer Srang
    rang ← rang - 1 // retour en arrière
    V ← V̄
```

7.3.4.2 Exemple : les n reines sur un échiquier

Dans la version non récursive écrite en Java donnée ici, on utilise pour représenter la solution partielle la classe `Stack` que nous avons déjà utilisée pour implanter les apples de fonctions et simuler la récursion.

Mais ce qui nous intéresse en plus ici, c'est qu'un objet de la classe `Stack` est de fait un objet de la classe `Vector` : la classe `Stack` est définie comme une sous-classe de la classe `Vector`. Cela signifie qu'en plus des méthodes qui sont propres à la classe `Stack` (`push`, `pop` et `empty`), les méthodes de la classe `Vector` (par exemple `contains` ou `get`) sont utilisables sur un objet de la classe `Stack`. Par ailleurs, la fonction `println` affiche agréablement le contenu d'un objet de la classe `Stack`.

Il faut noter que les ensembles de positions possibles ne sont pas effectivement construits. On a utilisé une fonction `suivant` qui détermine la première position possible supérieure à une

valeur donnée sur une colonne donnée.

```
--> cat Reines.java
import java.util.Stack;
class Reines {
  static Stack<Integer> solution;
  static int n;
  static Integer suivant(int col, Integer suiv) {
    int j;
    boolean b = true;
    if (col == n + 1) return null;
    int x = suiv + 1;  // x = 1 + suiv.intValue()
    while(x <= n) {
      if(!solution.contains(x)) {
        b = true;
        for(j = 0; j < col - 1; j++) {
          if(Math.abs(col -1 - j) == Math.abs(x - solution.get(j))) {
            b = false; break; }
        }
        if (b) return x;
      }
      x++;
    }
    return null;
  }
  static int reines(int n) {
    solution = new Stack<Integer>();
    int nombreSol = 0;
    int col = 1; // numero de la prochaine colonne traitee
    Integer suiv = 0;
    while(col > 0) {
      suiv =  suivant(col, suiv);
      while(suiv != null) {
        solution.push(suiv);
        if(col == n){   // c'est  une solution complete
           nombreSol++;
           if(nombreSol == 1) System.out.println(solution);
         }
         col = col + 1; suiv = suivant(col, 0);
      }
      if(!solution.empty()) suiv = solution.pop();
      col--;
      }
    return nombreSol;
  }
  public static void main(String[] args) {
    for(n = 4; n <= 10; n++)  {
      System.out.println("reines(" + n + ") : ");
      System.out.print("     Premiere solution : ");
      System.out.println("     " + reines(n) + " solutions");
      System.out.println("--------------------------");
    }
  }
}
--> java Reines
reines(4) :
     Premiere solution : [2, 4, 1, 3]
     2 solutions
--------------------------
reines(10) :
     Premiere solution : [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
     724 solutions
--------------------------
reines(6) :
```

```
     Premiere solution : [2, 4, 6, 1, 3, ...]
     4 solutions
-------------------------
reines(7) :
     Premiere solution : [1, 3, ... 7, 2, 4, 6]
     40 solutions
-------------------------
reines(8) :
     Premiere solution : [1, ... 8, 6, 3, 7, 2, 4]
     92 solutions
-------------------------
reines(9) :
     Premiere solution : [1, 3, 6, 8, 2, 4, 9, 7, ...]
     3...2 solutions
-------------------------
reines(10) :
     Premiere solution : [1, 3, 6, 8, 10, ... 9, 2, 4, 7]
     724 solutions
-------------------------
-->
```

7.4 Pour finir : résolution brutale d'une grille de sudoku

7.4.1 Le problème

Il s'agit de compléter une grille carrée de côté 9 (9 lignes, 9 colonnes et 9 petits carrés de côté 3) contenant initialement un certain nombre de cases remplies de nombres entre 1 et 9 de telle sorte que chaque ligne, chaque colonne et chaque petit carré contienne exactement une fois chacun des nombres de 1 à 9.
Un exemple d'un telle grille à remplir est :

	0	1	2	3	4	5	6	7	8
0	7	8	1						
1								3	
2	9				2	5			
3				3		1	8		
4		4	7				1	6	
5			5	6		9			
6				4	8				7
7		6							
8							2	9	5

La classe `Position` permet d'identifier une case de la grille :

```
--> cat Position.java
class Position {
  int lig, col;
  Position(int i, int j) { lig = i; col = j;}
}
```

7.4.2 La solution proposée

Dans cette solution, il n'y a aucune intelligence : on explore de manière systématique toutes les possibilités. Un joueur de sudoku procédant de la sorte remplit la grille avec un crayon et fait un usage intensif de la gomme pour effacer des tentatives infructueuses et revenir en arrière et en explorer de nouvelles. Une recherche de solution à la main de ce type est évidemment peu réaliste.

Nous traçons les grandes lignes de la solution basée sur le backtracking :

- la grille à compléter est fournie sous la forme d'un tableau carré d'entiers de côté 9 : les cases non remplies contiennent la valeur 0 ;
- la fonction `aRemplir` calcule le vecteur des positions des cases à remplir en les ordonnant selon l'ordre défini de la manière suivante :
  $(lig1, col1) < (lig2, col2)$ si et seulement si :
  - ou bien $lig1 < lig2$
  - ou bien $lig1 = lig2$ et $col1 < col2$

  Ainsi, pour la grille donnée en exemple, la fonction renvoie le vecteur :
  `[(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),`
  `(1,8),(2,1),(2,2),(2,3),(2,6),(2,7),(2,8),(3,0),(3,1),(3,2),(3,4),(3,7),(3,8),`
  `(4,0),(4,3),(4,4),(4,5),(4,8),(5,0),(5,1),(5,4),(5,6),(5,7),(5,8),(6,0),(6,1),`
  `(6,2),(6,5),(6,6),(6,7),(7,0),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),(7,8),(8,0),`
  `(8,1),(8,2),(8,3),(8,4),(8,5)]`
- la fonction `pasDansLigne` renvoie `true` si la valeur `val` ne figure pas sur la ligne `lig` de la grille `gr` ;
- la fonction `pasDansColonne` renvoie `true` si la valeur `val` ne figure pas sur la colonne `col` de la grille `gr` ;
- la fonction `pasDansPetitCarre` renvoie `true` si la valeur `val` ne figure pas dans le petit carré de la grille `gr` auquel appartient la position `(lig,col)` ;
- la fonction `possibles` construit un vecteur de valeurs possibles pour une position (lig, col) donnée encore non remplie donnée (on peut y mettre les valeurs ne figurant pas sur la ligne, sur la colonne ou le petit carré de la position). Cette fonction calcule donc l'ensemble $\mathtt{S}_i$ où i est l'indice (compté à partir de 0) de (lig, col) dans le vecteur construit par la fonction `aRemplir` compte tenu des valeurs posées sur les cases dont les positions sont aux indices 0, 1, ..., $i-1$ dans ce vecteur ;
- dans la solution proposée sous la forme de la fonction `resoudre`, on explore les possibilités de remplissage de la grille par une technique de *backtracking*.
  On y mémorise dans un objet de la classe `Stack` la liste des nombres écrits successivement sur la grille : ces nombres ont été extraits des $\mathtt{S}_i$ correctement construits au fur et à mesure du remplissage de la grille.

7.4.3 Code Java

```
--> cat Sudoku.java
import java.util.Vector;
import java.util.Stack;
class Sudoku {
  // determine la liste ordonnee des positions a remplir sur la grille gr
  static Position[] aRemplir (int[][] gr) {
    // dans un premier temps on compte le nombre de cases non remplies
    int cpt = 0;
    for(int i = 0; i < 9; i++)
      for(int j = 0; j < 9; j++)
         if(gr[i][j] == 0) cpt++;
    // on alloue le tableau pour memoriser les positions a remplir
    Position[] p = new Position[cpt];
    // on remplit effectivement le tableau des positions
    cpt = 0;
    for(int i = 0; i < 9; i++)
      for(int j = 0; j < 9; j++)
         if(gr[i][j] == 0) // une case non remplie
             p[cpt++] = new Position(i,j);
    return p;
  }
```

```
// teste si val est dans la ligne lig de la grille gr
static boolean pasDansLigne(int val, int[][]gr, int lig){
  for(int col = 0; col < 9; col++)
    if(gr[lig][col] == val) return false;
  return true;
}

// teste si val est dans la colonne col de la grille gr
static boolean pasDansColonne(int val, int[][]gr, int col){
  for(int lig = 0; lig < 9; lig++)
    if(gr[lig][col] == val) return false;
  return true;
}

// teste si val est dans le petit carre (lig,col) de la grille gr
static boolean pasDansPetitCarre(int val, int[][]gr, int lig, int col){
  int a = lig/3; int b = col/3;
  for(int i = 3   a; i <  3   (a + 1); i++)
    for(int j = 3   b; j < 3   (b + 1); j++)
      if(gr[i][j] == val) return false;
  return true;
}

// determine la liste des valeurs qu'on peut poser en (lig,col)
// compte tenu des nombres deja dans la grille
static Vector possibles(int[][]gr, int lig, int col) {
  Vector v = new Vector();
    for (int val = 1; val <= 9; val++)
      if(   pasDansLigne(val, gr, lig)
         && pasDansColonne(val, gr, col)
         && pasDansPetitCarre(val, gr, lig, col)
         )
         v.add(new Integer(val));
  return v;
}

// affiche la grille gr donnee en parametre
static void afficher(int[][] gr) {
  for(int i = 0; i < 9; i++){
    for(int j = 0; j < 9; j++)
      System.out.print(gr[i][j] + " ");
    System.out.println(); }
}

// resoud une grille donnee
static int resoudre(int[][] gr) {
  boolean first = true;   // pour afficher la premiere solution
  Position[] p = aRemplir(gr);  // liste des positions a remplir
  Vector[] S = new Vector[p.length]; // vecteur de vecteurs de valeurs possibles
  // valeurs acceptables dans premiere case libre
  S[0] = possibles(gr, p[0].lig, p[0].col);
  Stack<Integer>solution = new Stack<Integer>(); // positions deja remplies
  int x;
  int rang = 0; // indice de la position ou poser prochain nombre
  int nombreSol = 0; // le nombre de solutions deja trouvees
  while(rang > -1){
    while(S[rang] . size() > 0){
      x = ((Integer)S[rang].get(0)).intValue(); // 1-er nombre de liste des possibles
      solution.push(x);  // on le met sur la pile
      gr[p[rang].lig][p[rang].col] = x;  // on l'ecrit effectivement sur la grille
      S[rang].removeElementAt(0);  // l'enlever de la liste des valeurs a essayer
```

```
        if (solution.size() == p.length) { // si on a rempli toutes les cases
          nombreSol ++;  // une nouvelle solution a ete trouvee
          if(first) { // si c'est la premiere, on l'affiche
              afficher(gr); first = false; }
          solution.pop(); // on revient en arriere, si jamais il y en avait une autre
          gr[p[rang].lig][p[rang].col] = 0; // effacer valeur dans case courante de grille
          rang--; // la solution partielle est de rang inferieur
        }
        else { // pas une solution (toutes les cases n'ont pas ete remplies)
          rang ++;  // indice de la prochaine position a remplir
          S[rang] = possibles(gr, p[rang].lig, p[rang].col); // valeurs possibles
        }
      }
      solution.pop(); // plus de valeur a essayer. On revient en arriere
      gr[p[rang].lig][p[rang].col] = 0; // effacer derniere valeur ecrite dans gr
      rang--;
    }
    return nombreSol;
  }

  public static void main(String[] args) {
    int[][] grille = {
           {7, 8, 1, 0, 0, 0, 0, 0 ,0},
           {0, 0, 0, 0, 0, 0, 0, 3 ,0},
           {9, 0, 0, 0, 2, 0, 0, 0 ,0},
           {0, 0, 0, 3, 0, 1, 8, 0 ,0},
           {0, 4, 7, 0, 0, 0, 1, 6 ,0},
           {0, 0, 0, 6, 0, 9, 0, 0 ,0},
           {0, 0, 0, 4, 8, 0, 0, 0 ,7},
           {0, 6, 0, 0, 0, 0, 0, 0 ,0},
           {0, 0, 0, 0, 0, 0, 2, 9, 0}};
    System.out.println(resoudre(grille) + " solution(s) trouvee(s)");
  }
}
--> java Sudoku
7 8 1 9 3 4 0 2 6
2 0 4 7 1 6 9 3 8
9 3 6 8 2 0 7 4 1
6 2 9 3 7 1 8 0 4
3 4 7 2 0 8 1 6 9
8 1 0 6 4 9 3 7 2
0 9 3 4 8 2 6 1 7
1 6 2 0 9 7 4 8 3
4 7 8 1 6 3 2 9 0
1 solution(s) trouvee(s)
-->
```

7.4.4 Ce qui s'est passé

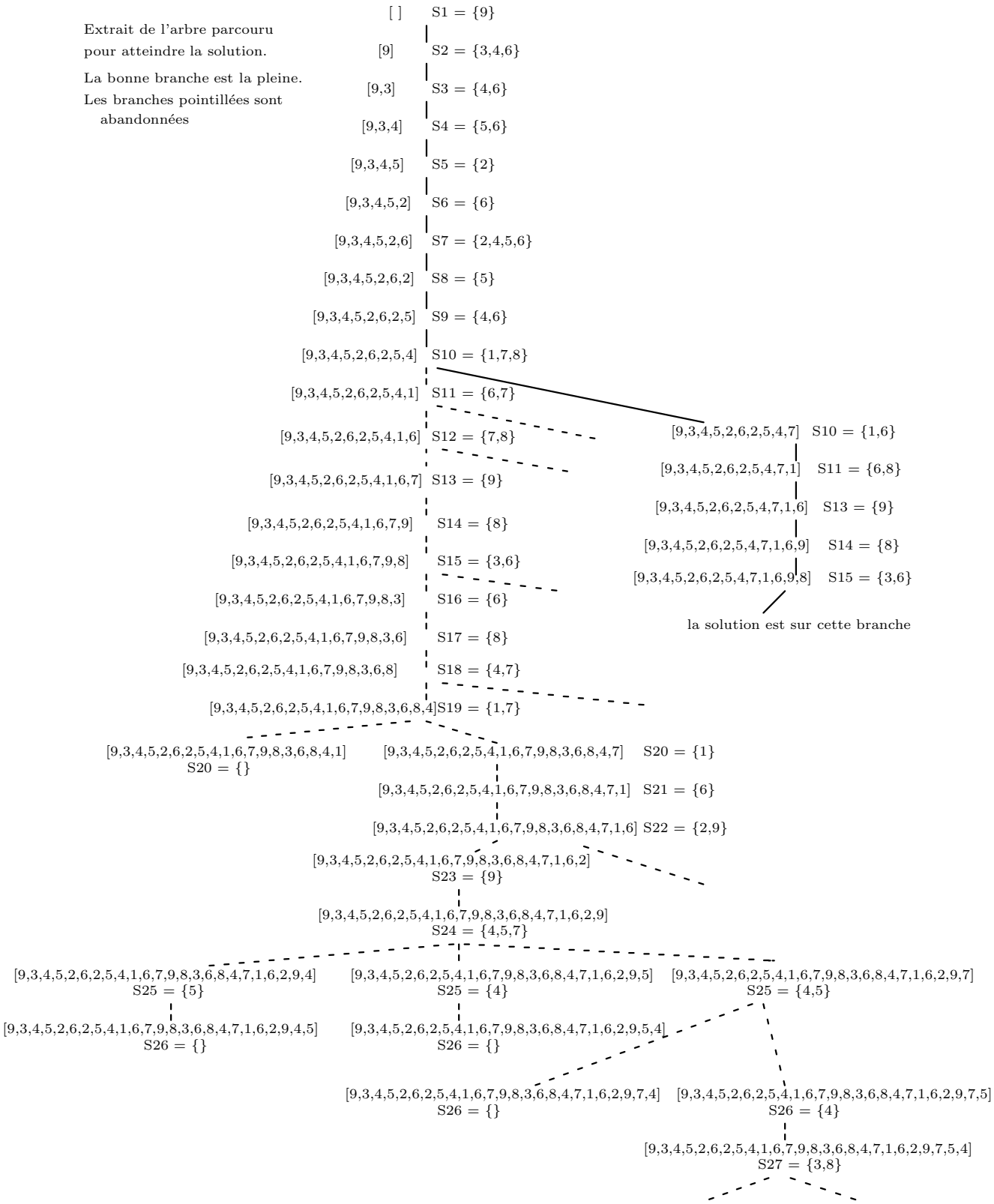
	0	1	2	3	4	5	6	7	8
0	7	8	1	9	3	4	5	2	6
1	2	5	4	7	1	6	9	3	8
2	9	3	6	8	2	5	7	4	1
3	6	2	9	3	7	1	8	5	4
4	3	4	7	2	5	8	1	6	9
5	8	1	5	6	4	9	3	7	2
6	5	9	3	4	8	2	6	1	7
7	1	6	2	5	9	7	4	8	3
8	4	7	8	1	6	3	2	9	5

La grille complétée : les cases encerclées sont celles qui ont été remplies en appliquant l'algorithme.

À la fin du programme, la variable `solution` (de type `Stack`) contient la suite des valeurs encerclées de gauche à droite et de haut en bas :

[9 3 4 5 2 6 2 5 4 7 1 6 9 8 3 6 8 7 4 1 6 2 9 7 5 4 3 2 5 8 9 8 1 4 9 3 7 2 5 9 3 2 6 1 1 2 5 9 7 4 8 3 4 7 8 1 6 3]

Un extrait de l'arbre exploré :



8

Codage de textes. Introduction à la compression

8.1 Codage de données textuelles

8.1.1 La notion de code

Outre la simple représentation de l'information dont c'est la fonction première en informatique sous la forme d'une suite de 0 et de 1, les codes sont conçus pour répondre éventuellement à un certain nombre d'exigences :

- diminuer la taille du codage : c'est l'objet de la compression des données ;
- rendre les données difficilement compréhensibles et incompréhensibles par des intrus : c'est l'objet de la cryptographie ;
- permettre de détecter des altérations des données, voire les corriger : c'est l'objet de la détection et correction d'erreurs.

Étant donné un ensemble de caractères (communément appelés lettres) $X = \{x_1, x_2, \ldots, x_n\}$, on note $X^\star$ l'ensemble de toutes les suites (appelées **mots**) qu'on peut écrire avec ces caractères (un mot de longueur 0 appelé mot vide, n mots formés d'un seul caractère et donc de longueur 1, n^2 formés de deux caractères et donc de longueur 2, ..., n^p mots formés de p caractères et donc de longueur p, ...).

Un codage de X sur un alphabet $Y = \{y_1, \ldots, y_q\}$ consiste à associer à chaque caractère x_i un mot $\tau(x_i)$ sur l'alphabet Y. Le codage $\tau(w)$ d'un mot w de $X^\star$ est alors obtenu en mettant bout à bout (en concaténant) les images de chacune des lettres composant ce mot.
Ce qui est important est que le résultat obtenu soit réversible et ne puisse pas correspondre au codage d'un autre mot construit sur l'alphabet X. Du point de vue mathématique, il faut donc que l'application τ (définie sur tous les mots de $X^\star$) soit injective (deux mots différents ont des images différentes par τ).

Par exemple, si l'alphabet X est $\{a, b, c, d, e\}$ et l'alphabet Y est $\{0, 1\}$ (celui qui est utilisé en informatique),

- l'application τ_1 suivante est acceptable :
  $\tau_1(a) = 00$, $\tau_1(b) = 10$, $\tau_1(c) = 100$, $\tau_1(d) = 110$ et $\tau_1(e) = 11$.
  Le mot *baace* est codé (10)(00)(00)(100)(11) et le mot 10000010011 n'admet pas d'autre factorisation avec les 5 mots précédents autre que celle qui a conduit à sa construction (cela peut ne pas paraître évident mais il existe des algorithmes pour le vérifier) ;
- l'application τ_2 suivante n'est pas acceptable pour définir un codage :
  $\tau_2(a) = 0$, $\tau_2(b) = 10$, $\tau_2(c) = 01$, $\tau_2(d) = 110$ et $\tau_2(e) = 1011$.
  En effet le codage de *ab* conduit à (0)(10), c'est-à-dire 010 et ce mot admet une autre décomposition : (01)(0) qui correspond au codage du mot *ca*.

8.1.2 Codes à longueur constante

8.1.2.1 Définitions

Si tous les mots associés aux lettres de l'alphabet X par l'applicaton τ sont de même longueur, disons k, le code est dit à longueur constante (dans le cas contraire, on parle de codes à longueur variable). Pour définir un tel code pour une valeur donnée de k, il suffit de considérer une application τ associant un mot (de longueur k) différent à chacune des lettres.
Un mot écrit sur l'alphabet Y ne peut être l'image d'un mot sur X que si sa longueur est un multiple de k. Son décodage paut alors être réalisé en le découpant en blocs de k lettres consécutives.
Rappelons que de manière générale, les textes sont codés en machine en utilisant un code (sur l'alphabet binaire $\{0, 1\}$) de longueur constante pour chacun des caractères qui le compose : on peut citer les codes ASCII 7 bits (permettant le codage de 128 caractères différents) ou étendus sur 8 bits (permettant le codage de 256 caractères), voire le code UNICODE sur 16 bits (permettant le codage de 64K caractères) utilisé par exemple en Java (ou dans un certain nombre de langages conçus pour Internet tels que XML et HTML4 par exemple) pour le codage des informations de type caractère (le type primitif `char` de Java).
Par exemple un fichier Unix contenant un texte constitué de n caractères occupera n octets, soit $8n$ bits comme l'illustre la séquence suivante :

```
--> echo ABCDE > toto
--> ls -l toto
-rw-r--r--  1 rifflet  rifflet  6  4 Apr 10:00 toto
--> od -x toto   <== contenu de toto sous forme hexadecimale
0000000     4142    4344    4_0a
0000006
-->
```

Le fichier `toto` se termine par un caractère de fin de ligne (de code ASCII décimale 10) et correspond à la suite de 48 bits suivante (afin de faciliter la lecture, un espace sépare chacune des séquences de 4 bits) :

```
0100 0001 0100 0010 0100 0011 0100 0100 0100 0101 0000 1010
```

8.1.2.2 Remarques

- un codage de longueur constante k utilisant un alphabet Y de n lettres permet le codage de n^k symboles (cas particulier informatique : un mot machine de k bits permet le codage de 2^k symboles) ;
- si on prend le problème dans l'autre sens, c'est-à-dire si on dispose de N symboles qu'on souhaite coder au moyen d'un alphabet de n lettres, quelle est longueur des mots utilisés pour réaliser ce codage dans le cas d'un code de longueur fixe ?
  La réponse est $Entier(log_n(N)) + 1$ où $Entier(x)$ a pour valeur la partie entière de x, c'est-à-dire le plus petit entier inférieur ou égal à x.
  Ainsi, pour coder 10 symboles avec un alphabet de deux caractères, typiquement 0 et 1, il faudra utiliser 4 caractères (bits dans le cas de 0 et 1). En effet $log_2(10) = 3.32$ (10 est compris entre 2^3 et 2^4). Avec 3 bits on ne peut coder que 8 symboles et avec 4 bits on peut en coder 16. En utilisant 4 bits, on n'a pas un codage optimal : 6 configurations ne sont pas utilisées.
  De même, le codage des 26 lettres de l'alphabet nécessite l'utilisation de 5 bits : en effet $2^4 = 16 < 26 < 2^5 = 32$.

8.1.3 Codes à longueur variable

Supposons qu'un texte soit composé uniquement des 4 caractères a, b, c et d.

Différents codages binaires sont évidemment possibles parmi lesquels :

- des codages de longueur fixe au moins égale à 2 comme par exemple celui associée à l'application τ_3 suivante :
  $\tau_3(a) = 00$, $\tau_3(b) = 01$, $\tau_3(c) = 10$ et $\tau_3(d) = 11$;
- des codages de longueur variable (les images des différents caractères dans le code ne sont pas toutes de même longueur) comme par exemple celui associée à l'application τ_4 suivante :
  $\tau_4(a) = 0$, $\tau_4(b) = 10$, $\tau_3(c) = 110$ et $\tau_4(d) = 111$

Intéressons nous au coût de ces codages du point de vue de la longueur de la séquence obtenue. Si la fréquence d'apparition de chacun ces caractères est la même (caractères équiprobables), le codage τ_3 est meilleur que τ_4 du point de vue de la taille de l'image du texte dans ce codage.

Par exemple, si chacun des caractères apparaît par exemple 16 fois dans le texte (c'est-à-dire a 16 occurrences dans le texte), le codage du texte nécessitera $16 \times 2 \times 4 = 128$ bits avec le codage τ_3 alors qu'avec le codage τ_4 il en nécessitera $16 \times (1 + 2 + 3 + 3) = 144$.

Par contre, si un texte de 64 caractères est constitué de 32 occurrences de a, de 16 occurrences de b et de 8 occurrences de chacun des caractères c et d, son codage nécessitera toujours 128 caractères avec le codage τ_3 alors qu'avec τ_4 il n'en demandera que 112.

De manière générale, considérons un texte (une source d'information), de longueur N, constitué des caractères c_1, c_2, ..., c_q et dans lequel le caracère c_i possède n_i occurrences ($N = n_1 + n_2 + \ldots + n_q$).

La probabilité p_i d'apparition du caractère c_i est n_i/N. On définit :

- la **quantité d'information propre** l_i apportée par une occurrence de c_i par :
  $l_i = log_2(1/p_i) = -log_2(p_i)$.
  Dans le cas particulier où les caractères sont équiprobables (on parle de source d'informations sans mémoire), on a $p_i = 1/N$ et donc $N = 1/p_i$. Ce qui correspond à $l(c_i) = log_2(N)$. Cette mesure de l'information (dont l'unité est le bit) a la propriété attendue que l'information apportée par l'occurrence d'un caractère en ayant peu apporte beaucoup d'information ;
- l'**entropie** d'un message m est définie par :
  $$H(m) = \sum_{i=1}^{q} p_i \times l_i = \sum_{i=1}^{q} p_i \times log_2(1/p_i)$$
- la valeur $H(m)$ ainsi calculée est de fait une moyenne. Reprenons l'exemple d'un message m de longueur 62 sur un alphabet de 4 caractères a, b, c et d avec respectivement 32, 16, 8 et 8 occurrences de chacun d'eux. On a :
  - probabilités des caractères : $p_a = 1/2$, $p_b = p_c = 1/4$ et $p_d = 1/8$;
  - les quantités d'information propres sont :
    $l_a = log_2(1/1/2) = log_2(2) = 1$, $l_b = log_2(4)2$ et $l_c = log_2(8) = 3$;
  - la moyenne des informations est :
    $(32 \times l_a + 16 \times l_b + 8 \times l_c + 8 \times l_d)/64$
    $= 1/2 \times l_a + 1/4 \times l_b + 1/8 \times l_c + 1/8 \times l_d$
    $= p_a \times l_a + p_b \times l_b + p_c \times l_c + p_d \times l_d$
    $= H(m) = 112/64 = 1.75$

    Les travaux de Shannon permettent d'assurer que quel que soit le code adopté, la longueur moyenne en bits des codages ne pourra être inférieure à l'entropie du message.
    Sur notre exemple, on pourra l'atteindre en codant a par un mot de longueur 1 (par exemple 0), b par un mot de longueur 2 (par exemple 10) et c et d par des mots de longueur 3 (par exemple 110 et 111).

8.2 Compression

8.2.1 Introduction

Les codes à longueur fixe sont définis sans aucune propriété statistique de la composition des textes à coder. Tous les caractères sont supposés équiprobables.
Mais lorsque ceci n'est plus le cas, il devient intéressant d'adopter des codes de longueur variable. Dans un tel code, on affectera à chaque caractère un mot d'autant plus court que ce caractère a beaucoup d'occurrences dans le texte à coder. Ce faisant, on diminuera le nombre de bits nécessaires pour le coder. C'est un exemple de ce qu'on appelle la compression.

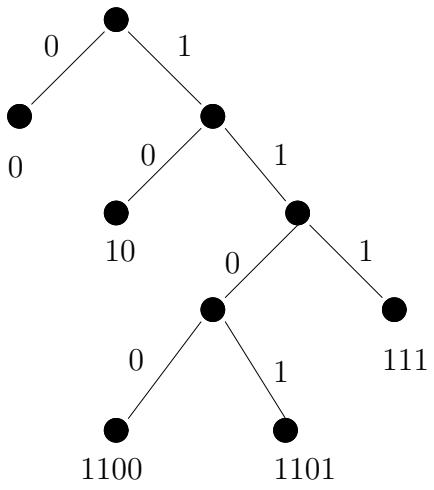
Il existe deux grandes catégories de méthodes de compression :

- les techniques de compression conservatives (sans pertes ou de compactage). Elles réduisent la taille des données sans les dégrader. À partir de la forme compactée il est possible de reconstituer exactement la donnée d'origine ;
- les techniques de compression non conservatives (ou avec pertes) : ces techniques sont utilisées par exemple pour la compression des sons ou des images.

8.2.2 La technique RLE

Cette technique (*Run Length Encoding*), utilisée à la base sur les séquences de caractères, consiste à rechercher les séquences d'a35.465(c)4.44458(t)36.8905(`)478(e)4.44458(r)-4.33716(e)4.44458(s)

Les caractères e, t et SP sont ceux qui ont le plus d'occurrences : on va les coder avec des mots aussi courts que possible. Par contre, les caractères b, l, n, s et $\hat{\imath}$ n'ayant qu'une occurrence, il sera peu pénalisant de les coder avec des mots plus longs.
Les deux algorithmes présentés ci-après conduisent à la construction de **codes préfixes** possédant cette propriété. Un code préfixe a la propriété qu'aucun mot n'y est le début d'un autre mot : on dit qu'aucun mot n'est facteur gauche ou préfixe d'un autre.
Tout ensemble péfixe (en général on considère des ensembles finis) est un code et le décodage d'un mot formé par concaténation de mots lui appartenant est réalisable par lecture de gauche à droite (sans faire de retour en arrière).
Enfin un tel ensemble est représentable par un arbre dont il est l'ensemble des feuilles (nœuds sans successeurs). Par exemple, l'ensemble $\{0, 10, 111, 1100, 1101\}$ est représentable par l'arbre :



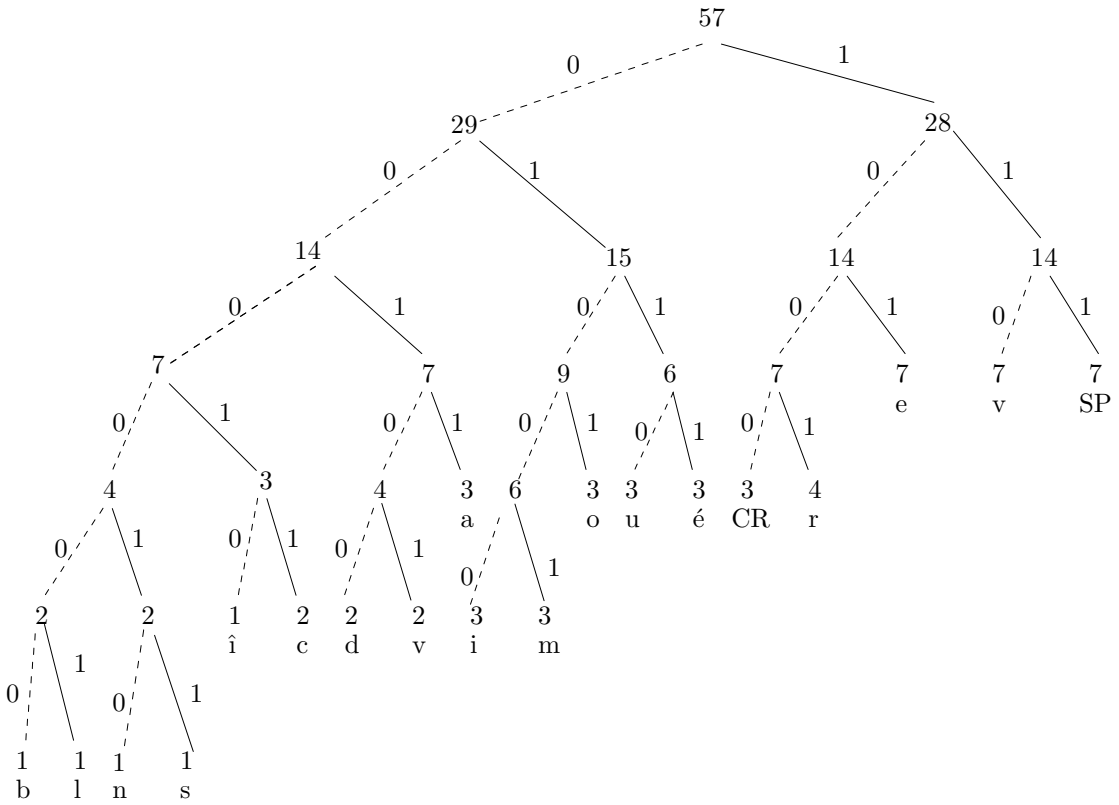
Les deux algorithmes que nous présentons maintenant sont les plus connus pour construire un tel arbre conduisant à un code préfixe pour les caractères composant un texte.

8.2.3.2 Approche top-down : algorithme de Shannon-Fano

Le premier de ces deux algorithmes (Shannon-Fano) construit l'arbre à partir de la racine (*top-down*).

On part du nombre 57 somme de toutes les valeurs du tableau trié des nombres d'occurrences. On écrit ce nombre comme somme de deux sous-sommes d'éléments du tableau : cela revient à découper le tableau de nombres en deux sous-tableaux dont les sommes des valeurs seront aussi proches que possible l'une de l'autre (et donc de la moitié de la valeur).

On itère ensuite récursivement ce principe sur les tableaux gauche et droit ainsi obtenus.
Sur notre exemple, le tableau d'occurrences conduit à la construction de l'arbre binaire suivant par l'algorithme *top-down* de Shannon-Fano :



ce qui correspondant au codage de chacune des lettres par la séquence correspondante (la dernière ligne de ce tableau donne la longueur du code de chacune des lettres) :

a	b	c	d	e	i	l	m	n	o
0011	000000	00011	00100	101	01000	000001	01001	000010	0101
4	6	5	5	3	5	6	5	6	4

r	s	t	u	v	î	é	SP	CR	
1001	000011	110	0110	00101	00010	0111	111	1000	
4	6	3	4	5	5	4	3	4	

En utilisant ce codage, le texte peut être écrit comme une suite de 228 bits alors que le codage ASCII en demande 456, soit un gain de 50%. Il faut cependant prendre en compte la nécessité de mémoriser le codage. Ce type de codage sera évidemment intéressant pour des textes longs pour lesquels l'espace nécessaire pour le dictionnaire sera négligeable par rapport au reste.

Un code Java pour cet algorithme est proposé ci-après (il ne s'agit que d'un prototype qui peut être, et ne demande qu'à l'être, grandement amélioré).

```
--> cat Fano.java
class Fano {
  static int indice = 0;
  static void fano (int pos, int taille, int valeur, int niveau,
                    String st, int[] occ, char[] dico) {
    int i, somme, nb, max;
    for(i = 0; i < niveau; i++)
        System.out.print("     ");
    System.out.print(valeur + " [" + st + "]");
    if(taille == 1){
        System.out.println(" --> " + dico[indice++]); return; }
    else
        System.out.println();
```

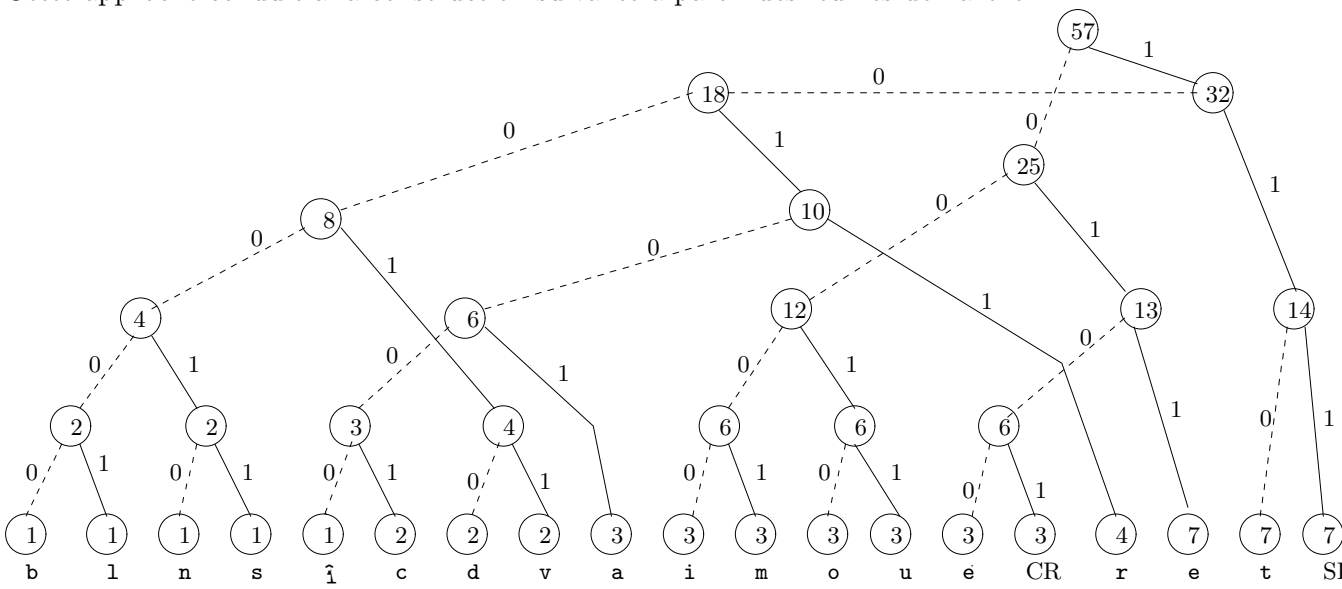
```
    somme = occ[pos];
    i = pos + 1;
    nb = 1;
    max = (valeur / 2) + (valeur % 2);
    while(nb < taille - 1 && somme < max){
        somme  = somme + occ[i]; i++; nb++; }
    if (nb != 1 && valeur + occ[i - 1] < 2   somme) {
        nb --; somme -= occ[i - 1]; }
    fano(pos, nb, somme, niveau + 1, st + "0", occ, dico);
    fano(pos + nb, taille - nb, valeur - somme, niveau + 1, st + "1", occ, dico);
  }
  public static void main(String[] args){
   int[] occ = {1,1,1,1,1,2,2,2,3,3,3,3,3,3,3,4,7,7,7};
   char[] dico = {'b','l','n','s','î','c','d','v','a','i','m','o','u','e','C','r',
                  'e','t','S'};
   int somme = 0;
   for(int i = 0; i < occ.length; i++)
        somme += occ[i];
   fano(0, occ.length, somme, 0, "", occ, dico);
  }
}
--> java Fano
7 []
    29 [0]
        14 [00]
            7 [000]
                4 [0000]
                    2 [00000]
                        1 [000000] --> b
                        1 [000001] --> l
                    2 [00001]
                        1 [000010] --> n
                        1 [000011] --> s
                3 [0001]
                    1 [00010] --> î
                    2 [00011] --> c
            7 [001]
                4 [0010]
                    2 [00100] --> d
                    2 [00101] --> v
                3 [0011] --> a
        1 [01]
            9 [010]
                6 [0100]
                    3 [01000] --> i
                    3 [01001] --> m
                3 [0101] --> o
            6 [011]
                3 [0110] --> u
                3 [0111] --> e
    28 [1]
        14 [10]
            7 [100]
                3 [1000] --> CR    <== le caractere RETURN
                4 [1001] --> r
            7 [101] --> e
        14 [11]
            7 [110] --> t
            7 [111] --> S   <== le caractere SPACE
-->
```

8.2.3.3 Approche bottom-up : algorithme de Huffman

Avec cette approche, on commence par les feuilles et on remonte vers la racine :

> à partir de deux nœuds associés aux nombres les plus petits, créer un nœud père associé à la somme de ces deux nombre.

Cette approche conduit à la construction suivante à partir des feuilles de l'arbre :



ce qui donne le codage suivant des lettres utilisées dans le texte :

a	b	c	d	e	i	l	m	n	o
10101	100000	101001	10010	011	0000	100001	0001	100010	0010
5	6	6	5	3	4	6	4	6	4

r	s	t	u	v	î	é	SP	CR	
1011	100011	110	0011	10011	101000	0100	111	0101	
4	6	3	4	5	6	4	3	4	

9 Introduction à la cryptographie

9.1 Introduction

9.1.1 Cryptographie, cryptanalyse, cryptologie

La **cryptographie** étudie les principes, méthodes et techniques mathématiques liées aux aspects de sécurité de l'information et répond aux besoins suivants :

- intégrité des données transmises : vérifier que les données n'ont pas été altérées (que ce soit frauduleusement ou accidentellement) ;
- contrôle d'accès : authentifier les utilisateurs afin de limiter l'accès aux données ;
- confidentialité : rendre les données incompréhensibles aux personnes non autorisées ;
- identification : authentifier l'émetteur et le destinataire des données ;
- non répudiation : ne pas permettre à l'émetteur de nier qu'il a effectivement émis les données ni au destinataire qu'il les a effectivement reçues.

Elle vise donc à fournir des techniques permettant l'échange d'informations entre un émetteur et un destinataire dans un environnement a priori non sûr (par exemple Internet) sans que des intrus puissent s'immiscer dans cet échange, que ce soit pour y lire des informations, soit pour en modifier ou en transmettre.
En résumé, la cryptographie est donc l'art ou la science de garder secret les messages.

La **cryptanalyse** s'intéresse à la sécurité des procédés de chiffrement élaborés dans le cadre de la cryptographie. On essaie d'y casser les fonctions cryptographiques par des méthodes autres que la force brute qui consiste, pour un algorithme donné, à essayer systématiquement toutes les clés. Il s'agit donc là de rechercher les faiblesses mathématiques de l'algorithme.

La **cryptologie** recouvre les domaines de la cryptographie et de la cryptanalyse.

9.1.1.1 Terminologie

Dans le jargon cryptographique, étant donné un message (*plain-text* ou *clear-text*), l'opération de **chiffrement** (ou **cryptage**) consiste à produire un message (*cipher-text*/chiffre-texte) cryptant ce message originel et visant à le rendre illisible.
L'opération inverse consistant à reconstruire le message originel à partir du message chiffré est le **déchiffrement** (ou **décryptage**).
Un chiffre est constitué d'une méthode de cryptage et d'une méthode de décryptage.

Si quelques algorithmes reposent sur le secret des algorithmes, les algorithmes modernes illustrent le principe de Kerchkhoffs : la sécurité d'un système de chiffrement n'est pas fondée sur le secret de la procédure qu'il utilise mais sur le paramètre qu'il utilise dans sa mise en œuvre et qui en est la **clé**.
Pour illustrer la différence des deux approches, prenons l'exemple de l'algorithme de chiffrement **ROT13** (méthode de César) pour un alphabet de 26 caractères (typiquement les lettres A-Z) qui décale simplement les lettres d'un texte de 13 positions. La lettre `A` est chiffrée en `N` qui est elle-même chiffrée par `A` :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
NOPQRSTUVWXYZABCDEFGHIJKLM
```

Chaque lettre est invariablement remplacée par la même : il s'agit d'un chiffrement par *substitution monoalphabétique*. Si deux entités décident de communiquer en chiffrant les messages par cette méthode, le secret entre ces deux entités est que le chiffrage est réalisé au moyen de cet algorithme.
Avec la seconde approche, l'algorithme **ROT13** n'est qu'un cas particulier d'un algorithme général paramétré par une permutation de l'alphabet utilisé. La technique d'encryptage consiste alors à substituer à un caractère du message son image par la permutation. La technique est connue mais le secret détenu par les deux entités communiquant entre elles est la permutation : c'est cette permutation qui constitue la clé.

9.1.2 Cryptage symétrique/cryptage asymétrique

9.1.2.1 Cryptage symétrique

Dans les **algorithmes symétriques**, la même clé (**clé secrète**) est utilisée pour le chiffrement et le déchiffrement (ou la clé de déchiffrement est différente mais est calculable facilement à partir de la clé de chiffrement). La clé doit donc être échangée et connue des deux entités. Le standard de chiffrement **D.E.S.** (*Data Encryption Standard*, 1975) en est un exemple. La clé y est constituée de 64 bits, donc une succession de 64 chiffres `0` ou `1`.
Nous verrons plus loin des exemples de transformations permettant la construction du message chiffré à partir du message en clair et de la clé.

9.1.2.2 Cryptage asymétrique

Dans les **algorithmes asymétriques**, les clés de chiffrement et de déchiffrement sont différentes. Une **clé publique** est utilisée pour le chiffrement : cette clé est largement diffusée. Par contre le déchiffrement est réalisé au moyen d'une **cle privée** que seul le destinataire est supposé connaître. Un exemple en est le standard de chiffrement **R.S.A.** (du nom de leurs auteurs, *Rivest, Shamir, Adleman*, 1977).
Le schéma suivant résume ce principe n

Pour l'extrait du texte de Verlaine, cela conduit au message chiffré suivant :

```
QLUCZJOJTXVWIMRSNUCKNTBBRQPYOVDKGCSIBX'XORO'EJYEIUA'PWWAGYFXWFFD'WFBTJPKCFXWFOSUXURVN
```

Cette technique de chiffrement a résisté près de 3 siècles à la cryptanalyse. Il n'a été cassé que vers 1850 par Babbage (parallèlement avec Kasiski).
On pourra consulter

`http://www.apprendre-en-ligne.net/crypto/vigenere/decodevig.html`.

Un tel chiffrement peut être réalisé par exemple avec le programme suivant (le programme est appelé avec deux paramètres, le premier étant l'alphabet et le second le message à chiffrer) :

```
--> cat Chiffre2.java
class Chiffre2 {
  public static void main(String[] args) {
    int i,         // pour parcourir le message
        pos,       // position du caractere dans l'alphabet
        decal = 0; // pour savoir de combien de positions on d\'ecale;
    int[] decalage = {3, 7, 2, 4, 7, 9, 1, 3, 8, 9, 2, 4, 10, 1, 3};
    if(args.length < 2) {
       System.err.println("Erreur parametres"); System.exit(0);
       }
    // pour construire le message crypté
    StringBuffer message = new StringBuffer();
    for(i = 0; i < args[1].length(); i++) {
        // position dans l'alphabet du caractere courant du message en clair
        pos = args[0].indexOf(args[1].charAt(i));
        // position du prochain caractere a ajouter au message chiffré
        pos = (pos + decalage[decal]) % args[0].length();
        // on rajoute le caractere au message chiffré
        message = message.append(args[0].charAt(pos));
        // on avance circulairement dans decalage
        decal = (decal + 1) % decalage.length;
    }
    System.out.println();
    System.out.println(message);
  }
}
--> java Chiffre2 "ABCDEFGHIJKLMNOPQRSTUVWXYZ '" "LES SANGLOTS LONGS DES VIOLONS DE L' AUTO
MNE BERCENT MON COEUR D'UNE LANGUEUR MONOTONE"

QLUCZJOJTXVWIMRSNUCKNTBBRQPYOVDKGCSIBX'XORO'EJYEIUA'PWWAGYFXWFFD'WFBTJPKCFXWFOSUXURVN
-->
```

9.2 Chiffrement par flot et par blocs

9.2.1 Chiffrement par flot

Dans un tel chiffrement, le message est découpé en blocs réguliers de petite taille : il s'agit d'un bit ou d'un octet. Un exemple en est le chiffrage de Vigenère que nous venons de présenter.
Le cryptage est réalisé au moyen d'une clé qui, pour garantir la sureté, doit satisfaire les conditions suivantes (satisfaites en particulier par le système de chiffrement élaboré par Vernam pour la communication via le téléphone rouge entre la Maison Blanche et le Kremlin) :

- la clé doit être de taille égale au nombre de blocs (bits ou octets) à crypter. Ainsi, si le message M est composé des n blocs M_1, M_2, $\ldots M_n$ ($M = M_1 M_2 \ldots M_n$), la clé K doit être elle-même constituée de n blocs, c'est-à-dire s'écrit $K = k_1 k_2 \ldots k_n$;
- les composants de la clé (les k_i) doivent être choisis de façon totalement aléatoire ;
- la clé ne doit servir qu'une seule fois.

Le chiffrement d'un message M avec la clé K est réalisé en appliquant composant par composant, c'est-à-dire entre m_i et k_i, une loi $*$, ce qui conduit à sa forme chiffrée $m_i * c_i$.
Il est important que la loi $*$ soit une loi de groupe, c'est-à-dire satisfasse la propriété qu'étant donné un couple (a, b), il existe un et un seul x tel que $a = b * x$.

Un exemple simple de fonction utilisée pour le cryptage dans le cas d'un bloc d'un seul bit consiste à réaliser un *OU exclusif* (XOR) entre chacun des bits du message et le bit correspondant de la clé (rappelons que cette opération souvent notée $\oplus$ est telle que $0 \oplus 0 = 1 \oplus 1 = 0$ et $0 \oplus 1 = 1 \oplus 0 = 1$).
Ainsi, si le message à transmettre est 1100011, une clé aléatoire de même longueur sera choisie, par exemple 0100110, ce qui conduira au chiffrage

$1100011 \oplus 0100110 = 1000101$

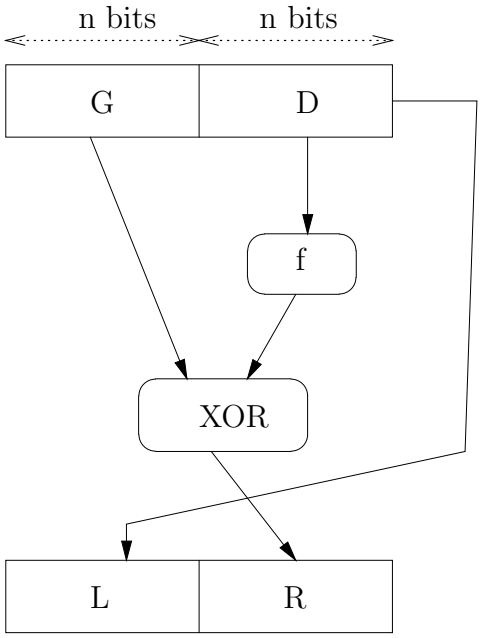
À la réception du message, le déchiffrement est réalisé par l'opération inverse du XOR, c'est-à-dire le XOR lui-même :

$1000101 \oplus 0100110 = 1100011$

Ce système apporte une grande sécurité mais nécessite d'une part de générer des clés de taille gigantesque, ce qui suppose une grande puissance de calcul et pose par ailleurs le problème de la transmission de ces clés. Cela a entrainé l'élaboration de solutions alternatives.

9.2.2 Chiffrement par blocs

Plutôt que d'utiliser une clé immense à usage unique de la taille du message à chiffrer, on va utiliser une clé plus petite et les algorithmes utiliseront cette clé de manière apparemment complexe.
Après avoir été numérisé (par exemple en utilisant les codes ASCII des caractères le composant ou tout autre codage conduisant à une suite de 0 et de 1), le message est découpé en blocs de longueur donnée, généralement une puissance de 2 comprise entre 32 et 512 bits. Ce qui différencie ce type de chiffrement du chiffrement par flot est la taille du bloc. Chaque bloc va être codé séparément au moyen d'une fonction bijective produisant un code paraissant aléatoire (le chiffrage d'un bloc peut être réalisé dans différents modes, mais cela dépasse le stade introductif auquel nous nous plaçons).
Nous allons simplement présenter le principe général proposé dans les années 1950 par Feistel :



La fonction f est supposée transformer une chaîne de n bits en une autre de même longueur.
L'algorithme de chiffrement va traiter des blocs de $2n$ bits selon le schéma précédent.
Le bloc est donc découpé en une partie gauche G et une partie droite D de n bits.
L'image du bloc (G, D) est le bloc (L, R) dans lequel $L = D$ et $R = G \oplus f(D)$.

Cette transformation est bijective. À partir du couple (L, R), il est facile de retrouver le couple $(G, D) : D = L$ et $G = R \oplus f(L)$ ($R \oplus f(L) = G \oplus f(D) \oplus f(L) = G \oplus f(L) \oplus f(L) = G \oplus 0 = G$).
La partie droite D du message initial n'a pas été transformée, elle a juste été déplacée.
D'où l'idée d'itérer le processus un certain nombre de fois (tours ou rondes).

↪ **L'algorithme de chiffrement DES** (*Data Encryption Standard*)
Il a été élaboré chez IBM puis largement adopté à partir de 1976 et a été particulièrement utilisé durant les années 1980 et 1990.
Il s'agit d'un algorithme de chiffrement symétrique utilisant une clé secrète de 64 bits dans laquelle les bits 8, 16, 24, 32, 40, 48, 56 et 64 sont des bits de parité. Le bit 8 sert ainsi à assurer que le nombre de bits valant `1` dans les bits 1 à 8 est impair : si les bits 1 à 7 ont comme valeur `1001100`, le huitième bit aura comme valeur `0`.
L'algorithme consiste en un réseau de Feistel à 16 tours : le message à chiffrer est découpé en blocs de 64 bits, chacun d'eux étant divisé en sous-blocs de 32 bits. Il a été cassé en 1998 et a été remplacé en 2000 par l'AES (*Advanced Encryption Standard*) dont les grandes lignes et caractéristiques sont :

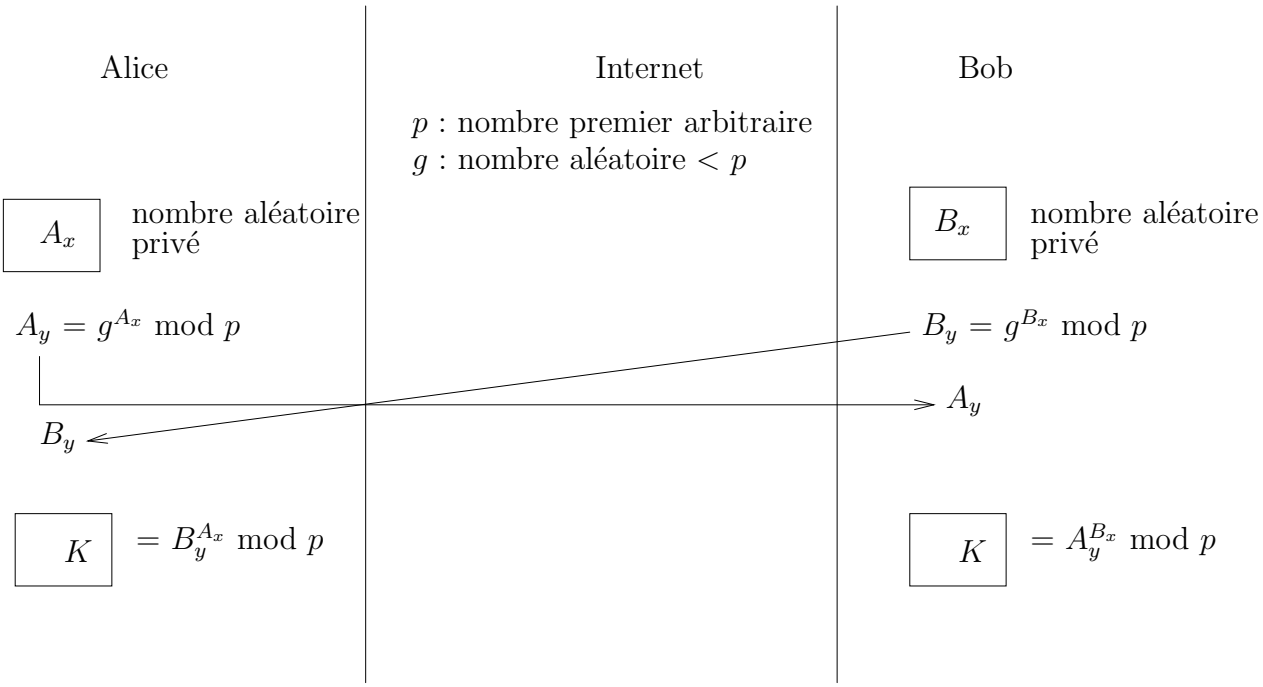
- algorithme de type symétrique comme le DES ;
- chiffrement par blocs (également comme le DES) ;
- différentes combinaisons de couples longueur de clé/longueur de bloc (128-128, 192-128 ou 256-128 bits) ;
- l'algorithme crypte des blocs de 128 bits de données et utilise pour cela :
  (a) une fonction non linéaire de subsitution d'octet à partir d'une table de substitution ;
  (b) une opération de décalages sur des lignes ;
  (c) un brouillage de colonnes ;
  (d) une addition par XOR ;
  (e) 10, 12 ou 14 tours selon que la clé est de taille 128, 192 ou 256.

9.3 Algorithme de Diffie et Hellman

L'objectif de cet algorithme est de fournir à deux entités Alice et Bob un moyen de se mettre d'accord sur un nombre utilisable comme clé de chiffrement sans qu'une troisième entité puisse découvrir ce nombre à partir des messages échangés par Alice et Bob.

9.3.1 Principe général

Un nombre premier p arbitraire est choisi ainsi qu'un entier aléatoire g inférieur à p : ces deux nombres seront publics et donc connus a priori de tous.
Le schéma suivant décrit le protocole de construction du secret K partagé entre Alice et Bob :



Comme on le voit, Alice calcule en fait $(g^{A_x})^{B_x}$ et Bob calcule $(g^{B_x})^{A_x}$. Les opérations sont réalisées modulo le nombre premier p , c'est-à-dire sur l'ensemble de nombres $Z_p = \{0, 1, \ldots, p-1\}$. Cet ensemble a la structure de corps fini et on a $(g^a)^b = (g^b)^a$.
Alice et Bob calculent donc le même secret K.
La sécurité du mécanisme repose sur la difficulté (à l'heure actuelle) de calculer, dans un corps fini, le logarithme discret, c'est-à-dire d'y inverser l'exponentiation et donc d'y déterminer la valeur de a à partir de celle de g^a.
Cependant le point faible du mécanisme est qu'il ne permet pas l'authentification des deux entités Bob et Alice : les échanges ne portent pas de signature. Il est donc possible à un adversaire d'intercepter les valeurs échangées entre Alice et Bob et de leur substituer des valeurs de son choix.

9.3.2 Un exemple

Nous donnons un exemple d'utilisation du protocole de Diffie-Hellman avec de petits nombres. Il va de soi qu'une utilisation normale s'appuie sur des nombres plus grands (typiquement un nombre premier de plus de 200 chiffres et des nombres A_x et B_x d'au moins 100 chiffres).

- Alice choisit le nombre pemier $p = 317$ et le nombre $g = 7$. Ces deux nombres sont transmis à Bob ;
- Alice choisit le nombre $A_x = 11$ et calcule $A_y = g^{A_x} \bmod p$, c'est-à-dire $7^{11} \bmod 317$.

n	1	2	3	4	5	6	7	8	9	10	11
$7^n \bmod 317$	7	49	26	182	6	42	294	156	141	36	252

  Alice obtient donc la valeur $A_y = 252$ qu'elle transmet à Bob ;
- Bob choisit le nombre $B_x = 14$ et calcule $B_y = g^{B_x} \bmod p$, c'est-à-dire $7^{14} \bmod 317$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$7^n \bmod 317$	7	49	26	182	6	42	294	156	141	36	252	179	302	212

  Bob obtient la valeur $B_y = 212$ qu'il transmet à Alice ;

- Alice calcule la valeur $K = 212^{11} \bmod 317$:

n	1	2	3	4	5	6	7	8	9	10	11
$212^n \bmod 317$	212	247	59	145	308	311	313	103	280	81	54

  Alice obtient la valeur 54 ;

- Bob calcule la valeur $K = 252^{14} \bmod 317$:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$252^n \bmod 317$	252	104	214	38	66	148	207	176	289	235	258	31	204	54

  Bob obtient également la valeur 54.

9.4 Cryptographie à clé publique : RSA

9.4.1 Fondements et bases mathématiques

La sécurité de ce système de chiffrement à clé publique, proposé en 1977 par Rivest, Shamir et Adleman repose sur la difficulté de factoriser de grands nombres : il est impossible, étant donné un entier n obtenu comme produit de deux grands nombres premiers (d'au moins 100 chiffres), de retrouver en un temps raisonnable ces deux nombres premiers.
Ainsi, étant donné le nombre de 155 chiffres suivant, le problème posé est de retrouver sa décomposition comme produit de deux nombres premiers :

10941738641_70_2742180970733220403_761200373294_44920_99091384213147634998428893478471799727_789126733249762_7_28997818337

c'est-à-dire retrouver les deux nombres

102639_9282974110_7720_4196_7399167_900716_67808038066803341933_21790711307779

et

10660348838012684_4820927220360012878679207 9_8_7_989291_22270608237193062808643

Il fait par ailleurs appel à l'arithmétique modulaire et repose sur le théorème d'Euler : nous rappelons tout d'abord le concept d'*arithmétique modulaire* et donnons, sans démonstrations, les résultats fondamentaux sur lesquels repose le chiffrement RSA :

- étant donné un nombre entier naturel n, on réalise toutes les opérations modulo n, c'est-à-dire qu'un nombre entier est identifié par le reste de division euclidienne (entière) par n.
  Cela revient donc à ne considérer que l'ensemble $Z_n = \{0, 1, \ldots, n-1\}$;
- étant donné un nombre entier naturel n, on note $\phi(n)$ (fonction phi d'Euler ou indicateur d'Euler), le nombre d'entiers compris entre 1 et n qui sont premiers avec n (c'est -à-dire dont le PGCD avec n est égal à 1) ;
- un nombre premier p est premier avec tous les nombres strictement plus petits que lui et donc p premier $\Rightarrow \phi(p) = p - 1$;
- p et q premiers $\Rightarrow \phi(p \times q) = \phi(p) \times \phi(q) = (p-1) \times (q-1)$. Par exemple $\phi(35) = 4 \times 6 = 24$ car 35 est le produit des deux nombres premiers 5 et 7 ;
- le théorème d'Euler : n entier naturel et a nombre premier avec $n \Rightarrow a^{\phi(n)} \equiv 1 \bmod n$;
- si un nombre x est premier avec n, il existe un nombre y tel que $x \times y \equiv 1 \bmod n$. Le nombre y est appelé inverse de x et est noté x^{-1} (ce nombre peut être calculé par l'algorithme d'Euclide étendu présenté en annexe) ;
- le petit théorème de Fermat est un corollaire du précédent dans le cas où n est premier : n entier naturel premier et $a < n \Rightarrow a^{n-1} \equiv 1 \bmod n$ (ou encore $a^n \equiv a$).

9.4.2 Le choix des clés

L'entité Alice, qui souhaite recevoir des messages chiffrés, va choisir deux grands nombres entiers premiers (aussi grands que possible, d'au moins 100 chiffres) distincts.
Elle calcule leur produit $n = p \times q$ et choisit un entier e premier avec $(p-1) \times (q-1)$.
La clé publique RSA d'Alice, qui sera publiée, par exemple, dans un annuaire, est le couple (n, e).
Cette clé sera utilisée par Bob lorsqu'il enverra un message à Alice pour crypter ce message.
Les messages cryptés seront déchiffrés par Alice au moyen d'une clé différente calculée à partir des entiers p, q et e qu'elle est la seule à connaître. Les seules informations qui circulent et que peut posséder une autre entité sont en effet n (sans connaitre p et q) et e.
Cette clé est un couple (n, d) dans lequel l'entier d est tel que $e \times d = 1[(p-1) \times (q-1)]$: un tel entier d existe puisque e est premier avec $(p-1) \times (q-1)$ (d est l'inverse e^{-1} de e modulo $(n-1) \times (q-1)$).
Autrement dit, $e \times d - 1$ est un multiple de $(p-1) \times (q-1)$.
L'algorithme d'Euclide (voir en annexe) permet à Alice de construire d à partir des valeurs p, q et e.
À titre d'exemple, travaillons avec deux petits nombres premiers : $p = 37$ et $q = 41$.
On obtient $n = 37 \times 41 = 1517$.
Choisissons par exemple $e = 91$, nombre qui est premier avec $36 \times 40 = 1440$.
La clé publique qui sera publiée et utilisée pour chiffrer les messages est $(1517, 91)$.
En ce qui concerne la clé privée, l'inverse de 91 est 1171. La clé privée associée est donc $(1517, 1171)$.

9.4.3 Le chiffrement

Reprenons le message que nous avons déjà utilisé en exemple :

```
LES SANGLOTS LONGS DES VIOLONS DE L'AUTOMNE BERCENT MON COEUR D'UNE LANGUEUR MONOTONE
```

Commençons par le numériser. Nous remplaçons chacun des 85 caractères qui le composent par son code ASCII. Ces codes ASCII ont tous 2 caractères décimaux : 39 pour `'`, 65 à 90 pour les lettres `A` à `Z` et 95 pour `_`.
Cela conduit à un message formé de $85 \times 2 = 170$ chiffres et on a rajouté un chiffre `0` en tête pour avoir une longueur multiple de 3 (en l'occurrence 171) :

```
076698332836_787176798483327679787183326869833286737976797883326869327639 6_8_84797778693266
698267697884327779783267796 98_823268398_786932766_78718_698_82327779787984797869
```

Le message ainsi obtenu va être crypté au moyen de la clé publique $(1517, 91)$: cela signifie en particulier qu'on va donc travailler avec de l'arithmétique modulo 1517 et qu'il faut donc constituer des blocs correspondant à des nombres inférieurs à 1517. On peut crypter des blocs de taille 3 (correspondant à des nombres compris entre 0 et 999 et donc tous inférieurs à 1517). Il ne faut pas crypter des blocs de taille 2 car dans ce cas on ferait simplement un chiffrement par substitutions facilement attaquable par analyse de fréquences.
On obtient la suite de blocs suivants à chiffrer :

```
076 698 332 836 _78 717 679 848 332 767 978 718 332 686 983 328 673 797 679 788 332 686 932
786 9.049_9(8)3.2-4.33268(e)403-2.1(l)27e doi7 oili68(l)2.7794-_32.276(6)3.04.049_9(7)3.0_09_(8)-_32.2_(8)-_32.2
```

```
0997 0042 0332 038_ 086_ 0800 1171 0293 0332 0138 0423 0799 0332 1090 0983 1189 1487 0387
1171 0196 0332 1090 0229 10_0 1483 1080 1482 0740 0648 0289 1224 1381 0933 0196 09_0 0740
0423 0289 0738 0042 0730 0289 0321 0734 0648 04_0 0073 093_ 1332 0042 0730 04_0 0738 093_
0_33 0387 0648
```

Le destinataire du message va procéder à son déchiffrement au moyen de sa clé privée, à savoir 1171. Il va constituer un message numérique en déchiffrant chacun des blocs de 4 chiffres.
Un tel bloc de 4 chiffres de valeur y sera déchiffré par la formule $y^{1171} \bmod 1517$. Ainsi le déchiffrement du premier bloc (de valeur 0997) donnera lieu au calcul de $997^{1171} \bmod 1517$, c'est-à-dire 76, puis le second bloc 42 conduira à $42^{1171} \bmod 1517$ c'est-à-dire 698, soit finalement la suite de valeurs suivantes :

```
 76 698 332 836 _78 717 679 848 332 767 978 718 332 686 983 328 673 797 679 788 332 686 932
763 96_ 8_8 479 777 869 326 669 826 769 788 432 777 978 326 779 698 _82 326 839 8_7 869 327
66_ 787 18_ 698 _82 327 779 787 984 797 869
```

qui sous forme de message numérisé sans espaces donne :

```
76698332836_787176798483327679787183326869833286737976797883326869327639 6_8_847977786932666
98267697884327779783267796 98_823268398_786932766_78718_698_82327779787984797869
```

Il ne reste plus qu'à reconstituer les codes sur 2 chiffres décimaux :

```
76[L] 69[E] 83[S] 32[ ] 83[S] 6_[A] 78[N] 71[G] 76[L] 79[O] 84[T] 83[S] 32[ ] 76[L] 79[O]
78[N] 71[G] 83[S] 32[ ] 68[D] 69[E] 83[S] 32[ ] 86[V] 73[I] 79[O] 76[L] 79[O] 78[N] 83[S]
32[ ] 68[D] 69[E] 32[ ] 76[L] 39['] 6_[A] 8_[U] 84[T] 79[O] 77[M] 78[N] 69[E] 32[ ] 66[B]
69[E] 82[R] 67[C] 69[E] 78[N] 84[T] 32[ ] 77[M] 79[O] 78[N] 32[ ] 67[C] 79[O] 69[E] 8_[U]
82[R] 32[ ] 68[D] 39['] 8_[U] 78[N] 69[E] 32[ ] 76[L] 6_[A] 78[N] 71[G] 8_[U] 69[E] 8_[U]
82[R] 32[ ] 77[M] 79[O] 78[N] 79[O] 84[T] 79[O] 78[N] 69[E]
```

9.5 Signature

9.5.1 Principe général

Il s'agit de fournir un moyen permettant à un destinataire de vérifier qu'un message provient effectivement de l'expéditeur attendu et non d'un intrus.
Pour ce faire, chacune des deux entités utilisera une clé privée et une clé publique. L'émetteur d'un message constituera une *empreinte* (ou *condensat*) de ce message. Pour construire une tel empreinte on utilise une *fonction de hachage*. L'objectif est d'obtenir un résumé significatif du message à partir duquel elle est construite.
Le mécanisme général correspond à la séquence suivante :

1. l'expéditeur calcule l'empreinte du message ;
2. l'expéditeur chiffre l'empreinte avec sa clé privée ;
3. l'expéditeur ajoute l'empreinte ainsi cryptée au message en clair et chiffre l'ensemble ainsi obtenu (message + empreinte cyptée) avec la clé publique du destinataire ;
4. l'expéditeur envoie le message au destinataire ;
5. le destinataire déchiffre le message avec sa clé privée ;
6. le destinataire déchiffre l'empreinte extraite du message avec la clé publique de l'expéditeur ;
7. le destinataire calcule l'empreinte du texte tel qu'il a décrypté en utilisant la même fonction que celle qu'a utilisée l'expéditeur du message et compare les deux empreintes qui doivent être identiques.

9.5.2 Un exemple

Nous allons reprendre le texte déjà utilisé et en constituer une empreinte simple en ne conservant que les 3 premiers caractères et les 3 derniers caractères du message : l'empreinte ainsi obtenue est donc « `LESONE` ».

Nous supposons que l'émetteur en est Bob et qu'il possède :

- la clé publique d'Alice, c'est-à-dire $(1517, 91)$;
- une clé privée : $(1073, 71)$. Cette clé a été calculée à partir des nombres premiers 29 et 37, et donc $n = 1073$. Le nombre $e = 71$ (premier avec $1008 = 28 \times 36$) a été choisi, ce qui conduit à la valeur $d = 71$ utilisée dans la clé privée ($71 \times 71 \equiv 1 \bmod 1008$). On se trouve dans un cas où la clé privée et la clé publique sont identiques (ce qui n'est pas souhaitable, mais nous ferons avec) et elles sont égales à $(1073, 71)$.

De son côté, Alice, la destinatrice du message, possède :

- la clé publique de Bob, c'est-à-dire $(1073, 71)$;
- sa clé privée, c'est-à-dire $(1517, 1171)$.

La forme numérique de l'empreinte (suite des codes ASCII des caractères) donne (les espaces servent simplement à séparer les codes) :

```
76 69 83 79 78 69
```

que nous regroupons par blocs de 3 chiffres en vue de leur chiffrement (cela donne donc des nombres inférieurs à 1073) :

```
766 983 797 869
```

Le chiffrement de l'empreinte par Bob en utilisant sa clé privée donne une suite de 4 nombres écrits avec 4 chiffres :

```
0713 0844 0827 040
```

En vue de chiffrement avec la clé publique d'Alice, des nombres de 3 chiffres (donc inférieurs à 1517) sont constitués :

```
713 084 408 270 40
```

Le chiffrement de cette séquence est réalisé avec la clé publique d'Alice est réalisé en même temps que le chiffrement du message (tel qu'il est décrit précédemment et nous n'y revenons pas) :

```
0713 0121 0371 0973 1112
```

Alice va recevoir dans le message crypté ce double cryptage de l'empreinte construite par Bob à partir du message originel. Elle le décrypte avec sa clé privée, ce qui lui donne un message en clair et une empreinte encore cryptée :

```
713 084 408 270 40
```

Le décryptage avec la clé publique de Bob permet à Alice de reconstituer l'empreinte que Bob avait construite avec la fonction de hachage :

```
766 983 797 869
```

ou sous forme de codes de caractères :

```
76[L] 69[E] 83[S] 79[O] 78[N] 69[E]
```

Le dernier travail qu'Alice doit réaliser est le calcul de l'empreinte du message qu'elle a obtenu par le décryptage, puis elle doit vérifier que cette empreinte est la même que celle qu'elle a reçue de Bob sous forme doublement cryptée.

9.6 Annexes

9.6.1 Algorithme d'Euclide

Il permet le calcul du PGCD (plus grand commun diviseur) de deux nombres par la méthode suivante :

- $pgcd(n, 0) = n$ pour tout entier $n > 0$;
- si a et b sont des entiers > 0, $pgcd(a, b) = pgcd(b, a\%b)$ où $a\%b$ est le reste de la division euclidienne de a par b.

Ainsi, pour les deux nombres 91 et 1440 que nous avons utilisés pour illustrer le chiffrement RSA, l'algorithme d'Euclide permet de calculer leur PGCD qui est égal à 1 et donc de vérifier qu'ils sont premiers entre eux .
Le déroulement de l'algorithme pour ces deux nombres donne :

- pgcd(91, 1440) = pgcd(1440, 91) car $91 = 1440 \times 0 + 91$;
- pgcd(1440, 91) = pgcd(91, 75) car $1440 = 91 \times 15 + 75$;
- pgcd(91, 75) = pgcd(75, 16) car $91 = 75 \times 1 + 16$;
- pgcd(75, 16) = pgcd(16, 11) car $75 = 16 \times 4 + 11$;
- pgcd(16, 11) = pgcd(11, 5) car $16 = 11 \times 1 + 5$;
- pgcd(11, 5) = pgcd(5, 1) car $11 = 5 \times 2 + 1$;
- pgcd(5, 1) = pgcd(1,0) car $5 = 1 \times 5 + 1$;
- pgcd(1,0) = 1.

Cet algorithme s'écrit sous forme récursive de manière simple en Java :

```
static int pgcd(int a, int b) {
   if(b == 0)
       return a;
   return pgcd(b, a % b);
}
```

9.6.2 Théorème de Bezout et algorithme d'Euclide étendu

Le théorème de Bezout assure que le PGCD d de deux entiers a et b est une combinaison linéaire à coefficients entiers de a et b et donc qu'il existe des entiers u et v tels que $d = a \times u + b \times v$.

L'algorithme d'Euclide peut être modifié (algorithme d'Euclide étendu) pour déterminer ces deux coefficients u et v.
Le principe est simple : on exprime tout d'abord a et b comme combinaision linéaire de a et b :

- $a = a \times (1) + b \times (0)$
- $b = a \times (0) + b \times (1)$

Puis on exprime les restes successifs des divisons effectuées par l'algorithme d'Euclide, comme combinaisons linéaires de a et b. Par exemple de l'égalité $a = b \times q_1 + r_1$, on déduit $r_1 = a - b \times q$ et en reportant les expressions de a et b, on trouve une expression linéaire de r_1 en fonction de a et b.
En continuant, on obtient $b = r_1 \times q_2 + r_2$, d'où on tire $r_2 = b - r_1 \times q_2$. En remplaçant b et r_1 par leurs expressions en fonction de a et b on obtiendra une expression de r_2 en fonction de a et b.
En itérant ce processus, on obtiendra une expression du PGCD de a et b sous forme de combinaison linéaire de a et b.

Appliquons ce principe aux deux entiers 1440 et 91 :

- $1440 = 1440 \times (1) + 91 \times (0)$
- $91 = 1440 \times (0) + 91 \times (1)$
- $75 = 1440 \times (1) - 91 \times (15) = 1440 \times (1) + 91 \times (-15)$
- $16 = 91 \times (1) - 75 \times (1) = 91 \times (1) - 1440 \times (1) - 91 \times (-15) = 1440 \times (-1) + 91 \times (16)$
- $11 = 75 \times (1) - 16 \times (4) = 1440 \times (1) + 91 \times (-15) - 1440 \times (-4) - 91 \times (64) = 1440 \times (5) + 91 \times (-79)$
- $5 = 16 \times (1) - 11 \times (1) = 1440 \times (-1) + 91 \times (16) - 1440 \times (5) - 91 \times (-79) = 1440 \times (-6) + 91 \times (95)$
- $1 = 11 \times (1) - 5 \times (2) = 1440 \times (5) + 91 \times (-79) - 1440 \times (-12) - 91 \times (190) = 1440 \times (17) + 91 \times (-269)$

En travaillant modulo 1440, la valeur -269 est égale à 1171 $(1440 - 269)$, valeur que nous avons adoptée pour la clé privée associée à la clé publique $(1517, 91)$.
Cette méthode de calcul des coefficients u et v dans l'expression linéaire du PGCD de a et b est implantée dans le code Java suivant :

```
--> cat Bezout.java
class Couple {
  int c1, c2;
  Couple(int a, int b) { c1 = a; c2 = b; }
  // redefinition de la fonction toString pour imprimer un couple de maniere agreable
  public String toString() {
      return("(" + c1 + "," + c2 +")");
  }
}
class Bezout {
  static Couple bezout(int a, int b) {
    int q, r;
    Couple cpl1 = new Couple (1, 0);
    Couple cpl2 = new Couple (0, 1);
    Couple tmp;
    System.out.println(a + " " + b);
    while (b != 0) {
       q = a  / b; r = a % b;
       a = b; b = r;
       tmp = new Couple(cpl1.c1 - q   cpl2.c1, cpl1.c2 - q   cpl2.c2);
       cpl1 = cpl2; cpl2 = tmp;
       System.out.println(cpl1 + " " + cpl2);
       System.out.println("================");
       System.out.println(a + " " + b);
    }
    return cpl1;
  }
  public static void main(String[] args){
    System.out.println(bezout(1440, 91));
  }
}
--> java Bezout
1440 91
(0,1) (1,-15)
================
91 75
(1,-15) (-1,16)
================
75 16
(-1,16) (16,-79)
================
```

```
16 11
(-79) (-6,9)
================
11
(-6,9) (17,-269)
================
1
(17,-269) (-91,1440)
========
1 0
(17,-269)
-->
```

9.6.3 Du code Java pour le chiffrement RSA

```
--> cat Cle.java
class Couple {
  int c1, c2;
  Couple(int a, int b) { c1 = a; c2 = b;}
  // redefinition de la fonction toString pour imprimer un couple de maniere agreable
  public String toString() { return("(" + c1 + "," + c2 +")"); }
}

class Cle {
  int valeur, modulo;
  // constructeur d'une cle pour un modulo et une valeur donnee
  Cle(int valeur, int modulo){ this.valeur = valeur; this.modulo = modulo;}
  public String toString() { return("(" + valeur + "," + modulo +")"); }

  static Couple bezout(int a, int b) {
    int q, r;
    Couple cpl1 = new Couple (1, 0);
    Couple cpl2 = new Couple (0, 1);
    Couple tmp;
    while (b != 0) {
      q = a  / b; r = a % b;
      a = b; b = r;
      tmp = new Couple(cpl1.c1 - q   cpl2.c1, cpl1.c2 - q   cpl2.c2);
      cpl1 = cpl2; cpl2 = tmp;
      }
    return cpl1;
  }

// construction d'une cle publique et d'une cle privee a partir
// de deux nombres premiers p et q et un nombre e premier avec (p-1).(q-1)
static Cle[] creerCles(int p, int q, int e) {
  Cle[] keys = new Cle[2]; // le tableau des deux cles
  keys[0] = new Cle(e, p   q);  // la cle publique
  int d = bezout((p-1) (q-1), e).c2;
  if(d < 0)
    d = (p-1)   (q-1) + d;
  keys[1] = new Cle(d, p   q);
  return keys;
  }
}
--> cat RSA.java
class RSA {
  // impression d'un tableau de bytes
  static void imprimer(byte[] tab){
    for(int i = 0; i <tab.length; i++) System.out.print(tab[i]);
    System.out.println();
  }
```

```
// tranformation d'un message (suite de caracteres) en une suite de chiffres :
//      les caracteres du message sont supposes tous codes par un nombre compris
//      entre 10 et 99 (2 chiffres decimaux, le premier non nul).
//      Le tableau de chiffres construit doit avoir une taille multiple de n
static byte[] numeriser(String message, int n) {
  int lg = 2   message.length();
  int pos = 0;
  if(lg % n != 0) {pos = n - lg % n; lg = lg +  pos;}
  byte[] tab = new byte[lg];
  for(int i = 0; i < message.length(); i++){
    tab[pos++] = (byte) (message.charAt(i) / 10);
    tab[pos++] = (byte) (message.charAt(i) % 10);
    }
  return tab;
}

// transformation d'une suite de bytes contenant des chiffres en une chaîne de caracteres
// correspondant aux codes ASCII : les codes ASCII sont supposes compris entre 10 et 99
static StringBuffer construireMsg(byte[] tab) {
  StringBuffer message = new StringBuffer();
  int premier,
      taille = tab.length / 2;
  if(tab[0] != 0)
      premier = 0;
  else if(tab [1] != 0)
       premier = 1;
  else {
       premier = 2; taille --;}
  for(int i = 0; i < taille; i++)
     message.append((char)(10   tab[premier + 2   i] + tab[premier + 2   i + 1]));
  return message;
}

// fonction qui crypte le nombre bloc avec la cle key
static long crypter(long bloc, Cle key){
  long res = 1L;
  for (int i = 0; i < key.valeur; i++)
    res = (res   bloc) % key.modulo;
  return res;
}

// chiffrage d'un tableau t decoupe en blocs de taille lg1 (taille supposee mutiple de lg1 avec la cle key.
// Chaque bloc de taille lg1 sera encrypte en un bloc taille lg2
static byte[] chiffrer(byte[]t, int lg1, int lg2, Cle key) {
  int n = t.length / lg1; // nombre de blocs de taille lg1
  long bloc; // bloc exprime comme nombre
  int pos;
  // allocation du message chiffre
  long chiffre;
  byte[] msgChiffre = new byte[n   lg2];
  for(int i = 0; i < n; i++) {
    bloc = 0;
    for(int j = 0; j < lg1; j++)
        bloc = bloc   10 + t[lg1   i + j];
    chiffre = crypter(bloc, key);
    pos = lg2   (i + 1) -1;
    for(int j = 0; j < lg2; j++) {
        msgChiffre[pos--] = (byte)(chiffre % 10); chiffre /= 10;}
    }
  return msgChiffre;
 }
```

```
    public static void main(String[] args){
      if(args.length == 0){
        System.err.println("Parametre absent");
        System.exit(2);
        }
      Cle[] keys = Cle.creerCles(37, 41, 91);
      System.out.println("Cle publique : " + keys[0]);
      System.out.println("Cle privee : " + keys[1]);
      System.out.println("================================");
      System.out.println("Message en clair :");
      System.out.println("  " + args[0]);
      byte[] msgNum = numeriser(args[0], 3);  // suite des chiffres du message numerise
      System.out.println("Message numerise :");
      System.out.print("  ");
      imprimer(msgNum);
      byte[] chiffre = chiffrer(msgNum, 3, 4, keys[0]);
      System.out.println("Message chiffre :");
      System.out.print("  ");
      imprimer(chiffre);
      byte[] dechiffre = chiffrer(chiffre, 4, 3, keys[1]);
      System.out.println("Message dechiffre sous forme numerique :");
      System.out.print("  ");
      imprimer(dechiffre);
      System.out.println("Message dechiffre :");
      System.out.print("  ");
      System.out.println(construireMsg(dechiffre));
    }
  }
  --> java RSA "LES SANGLOTS LONGS DES VIOLONS DE L'AUTOMNE BERCENT MON COEUR D'UNE LANGUEUR MONOTONE"
Cle publique : (91,1_17)
Cle privee : (1171,1_17)
================================
Message en clair :
  LES SANGLOTS LONGS DES VIOLONS DE L'AUTOMNE BERCENT MON COEUR D'UNE LANGUEUR MONOTONE
Message numerise :
  076698332836_787176798483327679787183326869833286737976797883326869327639 6_8_847977786932666982676 9
  788432777978326779698_823268398_786932766_78718_698_82327779787984797869
Message chiffre :
  099700420332038_086_0800117102930332013804230799033210900983118914870387117101960332109002291 0_0148
  3108014820740064802891224138109330196 09_0074004230289073800420730028903210734064804_00073093_133200
  42073004_00738093_0_3303870648
Message dechiffre sous forme numerique :
  076698332836_787176798483327679787183326869833286737976797883326869327639 6_8_847977786932666982676 9
  788432777978326779698_823268398_786932766_78718_698_82327779787984797869
Message dechiffre :
  LES SANGLOTS LONGS DES VIOLONS DE L'AUTOMNE BERCENT MON COEUR D'UNE LANGUEUR MONOTONE
```