

Backtracking

Il s'agit d'une technique de résolution de problèmes adaptée à des 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 l'exploration complète de l'espace des candidats-solutions ne soit pas nécessaire. C'est donc une amélioration de la technique de la force brute qui consiste à générer toutes les combinaisons possibles et à tester pour chacune d'elles si elle constitue ou non une solution.

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 (*rebrousser chemin*) quand on est bloqué et essayer alors un autre chemin.

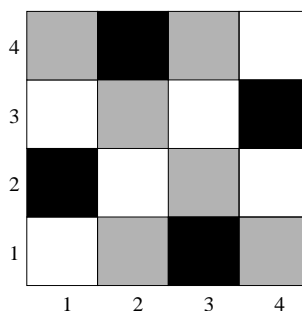
Un champ d'applications privilégié (mais il y en a d'autres) en est la résolution de jeux.

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.

↪ Un exemple introductif

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 désigne une ligne et c désigne 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 de l'exemple, on obtient [2 4 1 3].

Une solution est nécessairement une permutation de l'ensemble $[1\ 2\ \dots\ n]$, c'est-à-dire une suite de longueur n où chacun des nombres $1, 2, \dots, n$ apparaît (et donc apparaît une seule fois).

Si on considère une telle permutation $[l_1\ l_2\ \dots\ l_n]$, elle sera solution si et seulement si

$$\forall i, j = 1, \dots, n, \text{ si } i \neq j \text{ alors } |l_i - l_j| \neq |i - j|$$

ce qui exprime qu'il n'y a pas deux reines sur une même diagonale (condition *diag*).

Pour trouver une configuration satisfaisante des n reines (ou les trouver toutes), on peut envisager de générer les permutations de $1, 2, \dots, n$, après tout il n'y en a que $n!$ (pour $n = 8$ cela fait 40320 et pour $n = 10$, 3628800) et pour chacune d'elles tester si elle satisfait la condition.

Cela n'est cependant pas raisonnable.

On sait par exemple qu'aucune configuration de la forme $[1\ 2\ \dots]$ n'est solution : les positions $(1,1)$ et $(2,2)$ sont en effet sur la diagonale principale. Les $(n-2)!$ permutations de cette forme peuvent être rejetées globalement : il est inutile de les générer toutes afin de tester chacune d'elles. L'information initiale $\underline{1\ 2}$ est suffisante pour savoir que la configuration n'est pas une solution.

L'idée du *backtracking* est là. Générer des solutions partielles, c'est-à-dire correspondant au placement de m reines (avec $m < n$), puis vérifier qu'elles ne violent pas la condition. Une solution partielle satisfaisant le condition est susceptible de conduire à une solution et mérite d'être étendue, alors que si elle ne la satisfait pas, elle peut être abandonnée.

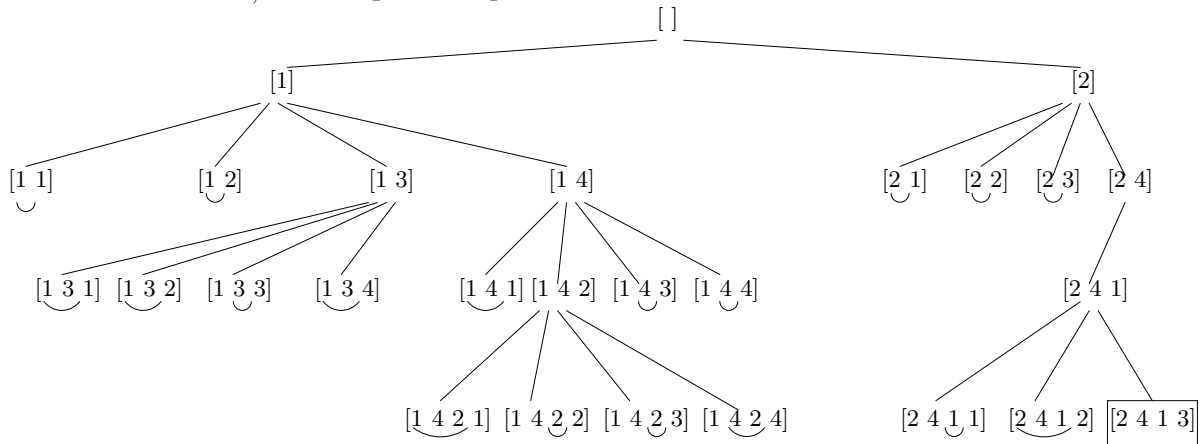
Ainsi, on part de la configuration initiale vide $[\]$ dans laquelle aucune reine n'est placée.

On commence par poser une reine dans la première colonne sur la première ligne.

Il faut ensuite placer une reine compatible sur la deuxième colonne. Cela exclut les lignes 1 et 2. On peut poser la reine de la colonne 2 sur la ligne 3.

À partir de cette solution partielle $[1\ 3]$, on essaie ensuite de poser une reine sur la colonne 3. On constate qu'aucune position ne convient. Cela impose de revenir en arrière et de déplacer la reine de la colonne 2. On essaie de la placer sur la ligne suivante, c'est-à-dire sur la ligne 4.

On continue ce processus consistant essentiellement d'extension de solutions partielles et de retour en arrière, ce qui conduit finalement à l'exploration de l'arbre suivant de configurations sur l'échiquier avant d'atteindre une solution, à savoir $[2\ 4\ 1\ 3]$:



↔ **Le problème général et le principe général du backtracking**

Dans le cas le plus général il s'agit de trouver la solution à un problème s'exprimant comme un vecteur $a = [a_1 a_2 \dots]$ de longueur finie mais quelconque et où chacun des a_i appartient à un ensemble A_i fini et totalement ordonné.

Une solution a doit satisfaire une certaine condition \mathcal{C} .

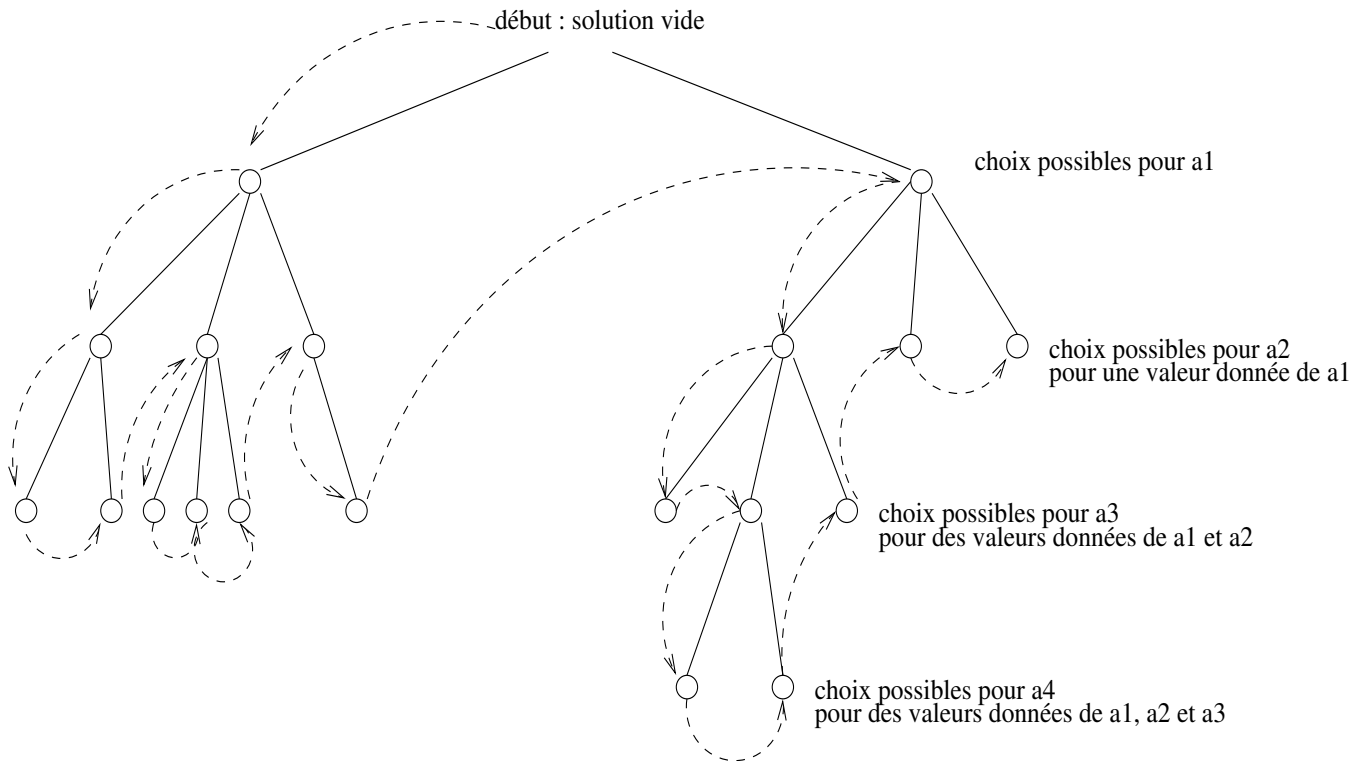
Le problème des n reines précédent en est un cas particulier : la longueur de la solution fait partie de la condition (la solution est de longueur n). Le reste de la condition est que tous les éléments du vecteur sont différents et que la condition *diag* est satisfaite.

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 S_1 de A_1 contenant toutes les valeurs acceptables pour a_1 (on a très souvent $S_1 = A_1$). On choisit le plus petit élément de S_1 (il y en a un puisque 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 S_2 de A_2 contenant toutes les valeurs acceptables pour a_2 et on construit la solution partielle $[a_1 a_2]$ en prenant pour a_2 la plus petit élément de 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, \dots, a_k entrant dans la composition de cette solution partielle conduisent à un ensemble $S_{k+1} \subset A_{k+1}$ 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 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 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) :



↔ Notation utilisée

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

- $V = [V_1 \dots]$ est un vecteur représentant une solution partielle de rang k et on note :
 - ◇ A_i l'ensemble totalement ordonné des valeurs théoriquement possibles pour V_i .
On notera $A_i[j]$ le j -ème élément de A_i dans l'ordre sur cet ensemble ;
 - ◇ si $V = [V_1 \dots V_m]$ alors $V \circ x = [V_1 \dots V_m x]$;
 - ◇ \bar{V} est la solution partielle de longueur $|V| - 1$ obtenue en supprimant la dernière composante de V . Donc si $V = [V_1 \dots V_{n-1} V_n]$, alors $\bar{V} = [V_1 \dots V_{n-1}]$.
Par convention $\bar{\emptyset} = \emptyset$

- \mathcal{C} désigne la condition correspondant aux solutions totales du problème : donc, $[V_1 \dots V_n]$ est une solution si et seulement si $\mathcal{C}([V_1 \dots V_n])$ 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 $([V_1 \dots V_n])$ est solution totale alors il n'existe pas de solution totale de rang $> n$ de la forme $[V_1 \dots V_n \dots]$).

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]) ;

- S_m est le sous-ensemble (totalement ordonné) de A_m contenant les valeurs acceptables pour V_m compte tenu des valeurs actuelles de V_1, \dots, 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.

↔ **Expression réursive du backtracking**

Les algorithmes de recherche par *backtracking* s'exprime de manière très simple sous forme réursive. 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  $S_{rang}$ 
    | pour chaque  $x \in S_{rang}$  faire
    | | backtrack(V ◦ 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 $V \circ x$, 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 ←  $\bar{V}$ 

```

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`.

```
--> cat ReinesRec.java
import java.util.Vector;
class ReinesRec {
    static Vector<Integer> solution; // solution
    static int nombreSol; // nombre de solutions
    static boolean first; // pour imprimer la 1-ère solution trouvée

    static Vector<Integer> positions(int n, int col) {
        Vector<Integer> lignes = new Vector<Integer>();
        boolean b;
        int m = solution.size();
        for(int lig = 1; lig <=n; lig++){
            b = true;
            if(!solution.contains(lig)) {
                for(int j = 0; j < m; j++)
                    if (Math.abs(col - j) == Math.abs(lig - solution.get(j))) {
                        b = false; break;
                    }
                if (b) lignes.add(lig);
            }
        }
        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
-----
-->

```

↔ **Expression non réursive du backtracking**

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

```

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 1-er 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

```

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 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 `suisant` qui détermine la première position possible supérieure à une valeur donnée sur sur une colonne donnée.

```

--> cat Reines.java
import java.util.Stack;
class Reines {
    static Stack<Integer> solution;
    static int n;
    static Integer suisant(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; // numéro de la prochaine colonne traitée
    Integer suiv = 0;
    while(col > 0) {
        suiv = suivant(col, suiv);
        while(suiv != null) {
            solution.push(suiv);
            if(col == n){ // c'est une solution complète
                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(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
-----
-->

```

↔ Un autre exemple : résolution brutale d'une grille de sudoku

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'une 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;}
}

```

Dans la solution donnée ici 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. Une recherche de solution à la main de ce type est évidemment peu réaliste.

Nous dégageons les grandes lignes de la solution donnée :

- 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 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 S_i correctement construits au fur et à mesure du remplissage de la grille.

```
--> cat Sudoku.java
import java.util.Vector;
import java.util.Stack;
class Sudoku {
    // détermine la liste ordonnée des positions à 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 mémoriser les positions à 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 carré (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;
}
// détermine la liste des valeurs qu'on peut poser en (lig,col)
// compte tenu des nombres déjà 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 donnée en paramètre
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();
    }
}
// résoud une grille donnée
static int resoudre(int[][] gr) {
    boolean first = true; // pour afficher la première solution
    Position[] p = aRemplir(gr); // liste des positions à remplir
    Vector[] S = new Vector[p.length]; // vecteur de vecteurs de valeurs possibles
    // valeurs possibles dans première case libre
    S[0] = possibles(gr, p[0].lig, p[0].col);
    Stack<Integer>solution = new Stack<Integer>(); // positions déjà remplies
    int x;
    int rang = 0; // indice de la position où poser prochain nombre
    int nombreSol = 0; // le nombre de solutions déjà trouvées
    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'écrit effectivement sur la grille
    S[rang].removeElementAt(0); // l'enlever de liste des valeurs à essayer
    if (solution.size() == p.length) { // si on a rempli toutes les cases
        nombreSol ++; // une nouvelle solution a été trouvée
        if(first) { // si c'est la première, on l'affiche
            afficher(gr);
            first = false;
        }
        solution.pop(); // on revient en arrière, 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 inférieur
    }
    else { // pas une solution (toutes les cases n'ont pas été remplies)
        rang ++; // indice de la prochaine position à remplir
        S[rang] = possibles(gr, p[rang].lig, p[rang].col); // valeurs possibles
    }
}
solution.pop(); // plus de valeur à essayer. On revient en arrière
gr[p[rang].lig][p[rang].col] = 0; // effacer dernière valeur écrite 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, 5, 0, 0 ,0},
        {0, 0, 0, 3, 0, 1, 8, 0 ,0},
        {0, 4, 7, 0, 0, 0, 1, 6 ,0},
        {0, 0, 5, 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, 5}};
    System.out.println(resoudre(grille) + " solution(s) trouvee(s)");
}
}
--> java Sudoku
7 8 1 9 3 4 5 2 6
2 5 4 7 1 6 9 3 8
9 3 6 8 2 5 7 4 1
6 2 9 3 7 1 8 5 4
3 4 7 2 5 8 1 6 9
8 1 5 6 4 9 3 7 2
5 9 3 4 8 2 6 1 7
1 6 2 5 9 7 4 8 3
4 7 8 1 6 3 2 9 5
1 solution(s) trouvee(s)
-->

```

	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 encadrées 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 encadrées 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]

Extrait de l'arbre parcouru pour atteindre la solution.
La bonne branche est la pleine.
Les branches pointillées sont abandonnées

