# Backtracking - sudoku

OPTION INFORMATIQUE - TP nº 3.1bis - Olivier Reynet

**À la fin de ce chapitre, je sais :**

- ☞ Implémenter un algorithme de recherche par la force brute
- ☞ Implémenter un algorithme de recherche par retour sur trace
- ☞ garantir qu'une fonction retourne toujours le même type et éventuellement `unit ()`.
- ☞ coder une fonction sur une liste de manière récursive en utilisant le filtrage de motif et les fonctions auxiliaires
- ☞ utiliser module List pour coder des équivalents aux codes récursifs (iter, map, filter, fold)

## A Sudoku ?

Le sudoku est une grille carrée de 9x9 qu'il faut compléter à l'aide de nombres entiers compris entre 1 et 9. La grille est divisée en neuf lignes, neuf colonnes et neufs blocs comme indiqué sur la figure 1. Les blocs sont les carrés de 3x3 délimités par des traits en gras.

Les règles de complétion sont les suivantes : **on ne peut pas placer deux fois le même chiffre**

- sur une même ligne,
- sur une même colonne,
- sur un même bloc.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | · | 9 | · | · | · | · | · | 7 |
| · | · | · | 5 | · | 9 | · | · | · |
| · | 3 | 5 | · | 7 | · | 8 | 9 | · |
| · | 2 | · | 7 | · | 8 | · | 1 | · |
| 7 | · | · | · | · | · | · | · | 2 |
| · | 6 | · | 1 | · | 5 | 4 | 7 | · |
| · | 9 | 7 | 8 | 5 | 4 | 2 | 3 | · |
| · | · | · | 9 | · | 6 | 7 | · | · |
| 5 | · | · | · | · | · | · | · | 6 |

FIGURE 1 – Exemple de sudoku. Un bloc est délimité par des traits en gras.

```
8 · 9 · · · · · 7
· · · 5 · 9 · · ·
· 3 5 · 7 · 8 9 ·
· 2 · 7 · 8 · 1 ·
7 · · · · · · · 2
· 6 · 1 · 5 4 7 ·
· 9 7 8 5 4 2 3 ·
· · · 9 · 6 7 · ·
5 · · · · · · 6
```

FIGURE 2 – Résultat sur la console de la fonction `rec_show board` avec sudoku `telegram`

## B Modélisation du sudoku

On choisit de représentere simplement le sudoku par une liste de valeur avec la convention qu'on lit la grille ligne par ligne. Ainsi, le sudoku de la figure 1 est modélisé par la liste :

```
let telegram = [ 8; 0; 9; 0; 0; 0; 0; 0; 7;
                 0; 0; 0; 5; 0; 9; 0; 0; 0;
                 0; 3; 5; 0; 7; 0; 8; 9; 0;
                 0; 2; 0; 7; 0; 8; 0; 1; 0;
                 7; 0; 0; 0; 0; 0; 0; 0; 2;
                 0; 6; 0; 1; 0; 5; 4; 7; 0;
                 0; 9; 7; 8; 5; 4; 2; 3; 0;
                 0; 0; 0; 9; 0; 6; 7; 0; 0;
                 5; 0; 0; 0; 0; 0; 0; 0; 6
];;
```

---

Le zéro représente l'absence de chiffre dans une case.

Si on utilise cette modélisation, le 36e élément de `telegram` est le dernier élément de la cinquième ligne et vaut 2.

Dans tout ce qui suit, on considère que $n$ est une variable globale qui vaut 9.

B1. Écrire une fonction récursive de signature `rec_show : int list -> unit` qui affiche le sudoku sur la console comme indiqué sur la figure 2.

B2. Écrire une fonction de signature `show : int list -> unit` qui affiche le sudoku sur la console comme indiqué sur la figure 2. On utilisera les fonctions du module List.

B3. Écrire une fonction de signature `in_row int -> int -> int list -> bool` qui teste sur la ligne de la case numérotée `i` la présence du chiffre `c` sur un sudoku. Elle renvoie `true` si le chiffre est déjà présent sur la ligne.

B4. Écrire une fonction de signature `in_col int -> int -> int list -> bool` qui teste sur la colonne de la case numérotée `i` la présence du chiffre `c` sur un sudoku. Elle renvoie `true` si le chiffre est déjà présent sur la colonne.

B5. Écrire une fonction de signature `in_block int -> int -> int list -> bool` qui teste la présence d'un chiffre sur un bloc. Elle renvoie `true` si le chiffre est déjà présent sur le bloc.

B6. Écrire une fonction de signature `is_valid_number : int -> int -> int list -> bool` qui teste la validité d'un chiffre $c$ sur la grille à l'index $i$. Elle renvoie `true` si le chiffre $c$ peut-être écrit dans la case $i$.

## C Résolution du sudoku par retour sur trace

L'algorithme de retour sur trace 1 construit au fur et à mesure les solutions partielles du problème et les rejette dès qu'il découvre une impossibilité.

**Algorithme 1** Algorithme de retour sur trace

```
1 : Fonction RETOUR_SUR_TRACE(v)                    ▷ v est un nœud de l'arbre de recherche
2 :     si v est une feuille alors
3 :         renvoyer Vrai
4 :     sinon
5 :         pour chaque fils u de v répéter
6 :             si u peut compléter une solution partielle au problème 𝒫 alors
7 :                 RETOUR_SUR_TRACE(u)
8 :     renvoyer Faux
```

On utilise la modélisation du sudoku précédente, c'est à dire qu'on complète la liste des cases occupées au fur et à mesure. On procède par ligne puis par colonne en positionnant d'abord un chiffre dans la première case vide puis une autre sur la deuxième case libre... À la fin, toutes les cases vides ont été complétées.

La différence par rapport au problème des n reines est que l'on commence avec une solution partielle.

C1. Comment coder «$v$ est une feuille» en OCaml?

> **Solution :** Une feuille de l'arbre d'exploration est atteinte lorsque toute la grille est remplie : l'indice de la liste à compléter lors de l'appel récursif vaudra alors 81 : on a fini de compléter la grille.

C2. Comment coder «$u$ peut compléter une solution partielle au problème $\mathcal{P}$»?

> **Solution :** Une solution partielle peut être complétée si le chiffre que l'on souhaite ajouter à la grille est valide dans cette position.

C3. Implémenter un algorithme de retour sur trace pour le problème du sudoku pour résoudre le sudoku `telegram`.

On dispose également d'un autre sudoku `multiple` :

```
let multiple = [8; 0; 9; 0; 0; 0; 0; 0; 7;
                0; 7; 6; 0; 8; 9; 0; 0; 0;
                0; 3; 0; 0; 7; 0; 8; 9; 0;
                0; 2; 0; 7; 0; 8; 0; 1; 0;
                7; 0; 0; 0; 0; 0; 0; 0; 2;
                0; 6; 0; 1; 2; 0; 4; 7; 0;
                6; 9; 7; 8; 0; 4; 2; 3; 1;
```

```
                    0; 0; 0; 9; 0; 6; 7; 0; 0;
                    0; 0; 0; 0; 0; 7; 9; 0; 6
    ];;
```

C4. Tester le programme de résolution sur `multiple`. Combien y-a-t-il de solutions ?

**Solution :** 28

C5. Quelle est la complexité de cet algorithme de retour sur trace ? Comment pourrait-on l'améliorer ?

**Solution :** La complexité est exponentielle. Chaque nœud de l'arbre d'exploration déclenchant potentiellement 9 appels récursif, la complexité dans le pire des cas (pas de retour avant d'atteindre les feuilles) vaut $\mathcal{O}(9^9)$, si on ne compte pas les cases remplies. Néanmoins, on peut imaginer guider l'exploration de l'arbre en utilisant des heuristiques d'exploration. Par exemple, commencer l'exploration par une ligne, une colonne ou un bloc qui sont déjà bien complétés et donc bien contraints. Dans ce cas, le retour sur trace élaguera plus vite et la complexité s'en trouvera améliorée.

C6. (bonus points) Pourrait-on proposer une version du programme dont le modèle de sudoku est un `Array`. Quelles différences y-aurait-t-il avec l'implémentation sous forme de `List` ?

**Solution :** L'utilisation d'une structure de type `Array` est à la fois un avantage et un inconvénient. C'est un avantage car l'accès à l'élément en position `i` s'effectue en un temps constant $O(1)$ et on en créé pas de nouvelles structures. C'est un inconvénient car il faut bien penser à retirer le nombre que l'on a inscrit sur la grille si celui-ci ne convient pas lors du retour sur trace. Avec les `list`, étant donné qu'une structure nouvelle est créée à chaque appel récursif, ce problème n'apparaît pas.

C7. (bonus points) Implémenter cet algorithme en Python.

**Solution :**

**Code 1 – Sudoku - List**

```
let telegram = [    8; 0; 9; 0; 0; 0; 0; 0; 7;
                    0; 0; 0; 5; 0; 9; 0; 0; 0;
                    0; 3; 5; 0; 7; 0; 8; 9; 0;
                    0; 2; 0; 7; 0; 8; 0; 1; 0;
                    7; 0; 0; 0; 0; 0; 0; 0; 2;
                    0; 6; 0; 1; 0; 5; 4; 7; 0;
                    0; 9; 7; 8; 5; 4; 2; 3; 0;
                    0; 0; 0; 9; 0; 6; 7; 0; 0;
                    5; 0; 0; 0; 0; 0; 0; 0; 6
];;

let all_done = [1; 2; 3; 4; 5; 6; 7; 8; 9;
4; 5; 6; 7; 8; 9; 1; 2; 3;
7; 8; 9; 1; 2; 3; 4; 5; 6;
```

```
2; 3; 4; 5; 6; 7; 8; 9; 1;
5; 6; 7; 8; 9; 1; 2; 3; 2;
8; 9; 1; 2; 3; 4; 5; 6; 7;
3; 4; 5; 6; 7; 8; 9; 1; 2;
6; 7; 8; 9; 1; 2; 3; 2; 5;
9; 1; 2; 3; 4; 5; 6; 7; 8
];;

let multiple = [8; 0; 9; 0; 0; 0; 0; 0; 7;
     0; 7; 6; 0; 8; 9; 0; 0; 0;
     0; 3; 0; 0; 7; 0; 8; 9; 0;
     0; 2; 0; 7; 0; 8; 0; 1; 0;
     7; 0; 0; 0; 0; 0; 0; 0; 2;
     0; 6; 0; 1; 2; 0; 4; 7; 0;
     6; 9; 7; 8; 0; 4; 2; 3; 1;
     0; 0; 0; 9; 0; 6; 7; 0; 0;
     0; 0; 0; 0; 0; 7; 9; 0; 6
     ];;


let n = 9;;
let rec_show sudoku = (* VERSION RECURSIVE PURE *)
  let rec aux i b = match b with
   |  [] -> print_string "\n"
   |  v :: t -> if i mod n = 0 then print_newline (); if v = 0 then print_string
        "." else print_int v; print_string " "; aux (i + 1) t
  in aux 0 sudoku;;

let show sudoku = (* VERSION AVEC ITERI *)
  let pl i v = if i mod n = 0 then print_newline (); if v = 0 then print_string
      "." else print_int v; print_string " ";
in List.iteri pl sudoku; print_newline ();;

let show sudoku = (* VERSION IMPERATIVE *)
    let head = ref 0 and lst = ref sudoku in
    for i = 0 to n*n-1 do
        if i mod n = 0 then print_newline ();
        head := List.hd !lst;
        lst := List.tl !lst;
        if !head = 0 then print_string "." else print_int !head;
        print_string " ";
    done;
    print_newline();;

let in_row i number board = (* AVEC FILTERI *)
  let row_number = i / n in
  let row = List.filteri (fun index element -> index / n = row_number && number
      = element) board in
  row != [];;

let in_row i number board = (* RECURSIVE PURE *)
  let row_number = i / n in
  let rec aux lst index  =
    match lst with
    | [] -> false
```

```
      | element::t -> if index / n = row_number && number = element then true else
          aux t (index + 1) in
    aux board 0;;

  let in_col i number board = (* AVEC FILTERI *)
    let col_number = i mod n in
    let col =  List.filteri (fun index element -> col_number = index mod n &&
        number = element) board in
    col != [];;

  let in_col i number board = (* RECURSIVE PURE *)
    let col_number = i mod n in
    let rec aux lst index =
        match lst with
      | [] -> false
      | element::t -> if index mod n = col_number && number = element then true
        else aux t (index + 1) in
    aux board 0;;

  let in_block i number board = (* AVEC FILTERI *)
    let row_number = i / n in
    let col_number = i mod n in
    let row_block =  (row_number / 3) * 3 in
    let col_block =   (col_number / 3) * 3 in
    let check index element = index / n   >= row_block  &&
                              index / n   <  row_block + 3 &&
                              index mod n >= col_block &&
                              index mod n <  col_block +3 &&
                              element = number in
    let block = List.filteri check board in
    block != [];;

  let in_block i number board =  (* RECURSIVE PURE *)
    let row_number = i / n in
    let col_number = i mod n in
    let row_block =  (row_number / 3) * 3 in
    let col_block =   (col_number / 3) * 3 in
    let check index element = index / n   >= row_block  &&
                              index / n   <  row_block + 3 &&
                              index mod n >= col_block &&
                              index mod n <  col_block +3 &&
                              element = number in
    let rec aux lst index =
      match lst with
      | [] -> false
      | element::t -> if check index element then true else aux t (index +1) in
    aux board 0;;

  let is_valid_number number i board = not (in_row i number board) && not (in_col
      i number board) && not (in_block i number board);;

  let sudoku board =
    let sol_nb = ref 0 in
    let rec aux index b =
      match index with
```

```
        | 81 -> (incr sol_nb; print_string "Solution #"; print_int !sol_nb; show b
           )  (* condition d'arrêt *)
        | _  -> if List.nth b index > 0  (* O(n) *)
                then aux (index + 1) b
                else (for number = 1 to 9 do
                        if is_valid_number number index b   (* O(n) tel qu'on l'a
                           programmé *)
                        then (aux (index+1) (List.mapi (fun i e -> if index = i
                           then number else e) b)) (* le map est en O(n) *)
                      done)
    in aux 0 board;;

  sudoku telegram;;
  sudoku multiple;;
```

**Code 2 – Sudoku - Array**

```
  let telegram = [|  [| 8; 0; 9; 0; 0; 0; 0; 0; 7|];
                     [| 0; 0; 0; 5; 0; 9; 0; 0; 0|];
                     [| 0; 3; 5; 0; 7; 0; 8; 9; 0|];
                     [| 0; 2; 0; 7; 0; 8; 0; 1; 0|];
                     [| 7; 0; 0; 0; 0; 0; 0; 0; 2|];
                     [| 0; 6; 0; 1; 0; 5; 4; 7; 0|];
                     [| 0; 9; 7; 8; 5; 4; 2; 3; 0|];
                     [| 0; 0; 0; 9; 0; 6; 7; 0; 0|];
                     [| 5; 0; 0; 0; 0; 0; 0; 0; 6 |] |];;

  let all_done = [|[|1; 2; 3; 4; 5; 6; 7; 8; 9|];
                   [|4; 5; 6; 7; 8; 9; 1; 2; 3|];
                   [|7; 8; 9; 1; 2; 3; 4; 5; 6|];
                   [|2; 3; 4; 5; 6; 7; 8; 9; 1|];
                   [|5; 6; 7; 8; 9; 1; 2; 3; 2|];
                   [|8; 9; 1; 2; 3; 4; 5; 6; 7|];
                   [|3; 4; 5; 6; 7; 8; 9; 1; 2|];
                   [|6; 7; 8; 9; 1; 2; 3; 2; 5|];
                   [|9; 1; 2; 3; 4; 5; 6; 7; 8|]
                 |];;

  let multiple = [|[|8; 0; 9; 0; 0; 0; 0; 0; 7|];
                   [|0; 7; 6; 0; 8; 9; 0; 0; 0|];
                   [|0; 3; 0; 0; 7; 0; 8; 9; 0|];
                   [|0; 2; 0; 7; 0; 8; 0; 1; 0|];
                   [|7; 0; 0; 0; 0; 0; 0; 0; 2|];
                   [|0; 6; 0; 1; 2; 0; 4; 7; 0|];
                   [|6; 9; 7; 8; 0; 4; 2; 3; 1|];
                   [|0; 0; 0; 9; 0; 6; 7; 0; 0|];
                   [|0; 0; 0; 0; 0; 7; 9; 0; 6|]
                 |];;

  let n = 9;;

  let show sudoku =
    let print_elem e =  if e > 0 then (print_int e; print_string " ") else
        print_string ". " in
    let print_line line = print_newline (); Array.iter print_elem line in
```

```
  print_newline (); Array.iter print_line sudoku; print_newline ();;

show telegram;;

(* vérifier si la ligne de la case (de numéro [0..80]) contient le chiffre (
    number) *)
(* val in_row : int -> 'a -> 'a array array -> bool = <fun> *)
let in_row case number board = let line = case / n in Array.mem number board.(
    line);;
in_row 77 7 telegram;;
in_row 77 5 telegram;;
in_row 77 6 telegram;;

(* vérifier si la colonne de la case (de numéro [0..80]) contient le chiffre (
    number) *)
(* val in_col : int -> 'a -> 'a array array -> bool = <fun> *)
let in_col case number board =
  let j = case mod n in
  let mem_line k = board.(k).(j) = number in
  let rec aux i =
    match i with
     | i when i < n ->  if mem_line i then true else aux (i+1)
     | _ -> false in
  aux 0;;
in_col 11 3 telegram;;
in_col 11 6 telegram;;
in_col 11 7 telegram;;
in_col 11 5 telegram;;
in_col 11 9 telegram;;

(* vérifier si le block 3x3 de la case (de numéro [0..80]) contient le chiffre (
    number) *)
(* val in_block : int -> 'a -> 'a array array -> bool = <fun> *)
let in_block case number board =
  let row_number = case / n in
  let col_number = case mod n in
  let row_block =  (row_number / 3) * 3 in
  let col_block =   (col_number / 3) * 3 in
  let found = ref false and i = ref row_block  and j = ref col_block in
  while not !found && !i < row_block + 3  do
     found := (board.(!i).(!j) = number);
     if (!j - col_block) = 2 then (j := col_block; incr i) else incr j;
  done;
  !found;;
in_block 77 2 telegram;;
in_block 77 1 telegram;;
in_block 77 9 telegram;;
in_block 77 4 telegram;;

(* Peut-on placer le chiffre number dans la case case ? *)
(* val is_valid_number : int -> 'a -> 'a array array -> bool = <fun> *)
let is_valid_number case number board = not (in_row case number board)
                                        && not (in_col case number board)
                                        && not (in_block case number board);;
```

```
(* Résolution du sudoku : backtracking *)
let sudoku board =
  let sol_nb = ref 0 in
  let rec backtrack c b =
    match c with
      |  81  -> (incr sol_nb; print_string "Solution #"; print_int !sol_nb; show
          b)
      |  _   -> let row = c / n and col = c mod n in
                if b.(row).(col) > 0 then backtrack (c + 1) b
                else (for number = 1 to 9 do
                        if is_valid_number c number b
                        then (b.(row).(col) <- number; backtrack (c + 1) b; b.(
                            row).(col) <- 0;)
                      done)
  in backtrack 0 board;;

sudoku telegram;;
sudoku multiple;;
sudoku all_done;;
```

**Code 3 – Sudoku - Python**

```
# coding: utf8
# nb solution --> https://oeis.org/A000170/list
from random import randrange, sample, seed

nu = 3
N = nu * nu
solutions_number = 0
seed(66)

telegram = [8, None, 9, None, None, None, None, None, 7,
            None, None, None, 5, None, 9, None, None, None,
            None, 3, 5, None, 7, None, 8, 9, None,
            None, 2, None, 7, None, 8, None, 1, None,
            7, None, None, None, None, None, None, None, 2,
            None, 6, None, 1, None, 5, 4, 7, None,
            None, 9, 7, 8, 5, 4, 2, 3, None,
            None, None, None, 9, None, 6, 7, None, None,
            5, None, None, None, None, None, None, None, 6
            ]

all_done = [1, 2, 3, 4, 5, 6, 7, 8, 9,
            4, 5, 6, 7, 8, 9, 1, 2, 3,
            7, 8, 9, 1, 2, 3, 4, 5, 6,
            2, 3, 4, 5, 6, 7, 8, 9, 1,
            5, 6, 7, 8, 9, 1, 2, 3, 2,
            8, 9, 1, 2, 3, 4, 5, 6, 7,
            3, 4, 5, 6, 7, 8, 9, 1, 2,
            6, 7, 8, 9, 1, 2, 3, 2, 5,
            9, 1, 2, 3, 4, 5, 6, 7, 8
            ]

multiple = [8, None, 9, None, None, None, None, None, 7,
            None, 7, 6, None, 8, 9, None, None, None,
```

```
            None, 3, None, None, 7, None, 8, 9, None,
            None, 2, None, 7, None, 8, None, 1, None,
            7, None, None, None, None, None, None, None, 2,
            None, 6, None, 1, 2, None, 4, 7, None,
            6, 9, 7, 8, None, 4, 2, 3, 1,
            None, None, None, 9, None, 6, 7, None, None,
            None, None, None, None, None, 7, 9, None, 6
            ]



def show(board):
    for c in range(N * N):
        if board[c]:
            print(board[c], end=" ")
        else:
            print('.', end=" ")
        # print("L :", c // N, "C: ", c % N)
        if (c % N) % nu == nu - 1:
            print(' ', end=" ")
        if c % N == N - 1:
            print()
            if (c // N) % nu == nu - 1:
                print()
    print("--")


def in_row(board, c, n):
    row = c // N
    return n in board[row * N:row * N + N]


def in_col(board, c, n):
    col = c % N
    return n in board[col:col + N * N:N]


def in_block(board, c, n):
    row = c // N
    col = c % N
    r = (row // nu) * nu
    c = (col // nu) * nu
    block_given = set()
    for i in range(r, r + nu):
        for j in range(c, c + nu):
            if board[i * N + j]:
                block_given.add(board[i * N + j])
    return n in block_given


def is_valid(board, c, n):
    return not in_row(board, c, n) and not in_col(board, c, n) and not in_block(
        board, c, n)
```

```
    def sudoku(board, c=0):
        global solutions_number
        if c == N * N or None not in board:  # last place set or full board
            solutions_number += 1
            print("New solution !", solutions_number)
            show(board)
            return True
        else:
            if board[c] is None:
                for n in range(1, N + 1):
                    if is_valid(board, c, n):
                        board[c] = n
                        sudoku(board, c + 1)
                        board[c] = None  # Backtrack
            else:
                sudoku(board, c + 1)
        return False


    if __name__ == "__main__":
        show(telegram)
        #sudoku(telegram)
        sudoku(multiple)
```