Taquin

May 21, 2019

1 Table of Contents

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2016-17

- *Date* : 29 mai 2017
- *Auteur* : Lilian Besson
- *Texte*: Taquin (pub2008-D2)

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un texte d'annale de l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [1]: Sys.command "ocaml -version";;
```

10	6	4	12
1	14	3	7
5	15	11	13
8	0	2	9

table T_1

0	1	2	4
3	6	10	12
5	7	14	11
8	9	15	13

table T_2

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

table finale T_f

FIG. 1. Différentes tables du jeu de taquin pour $n = 4$.

images/taquin.png

```
The OCaml toplevel, version 4.04.2
```

```
Out[1]: - : int = 0
```

> Notez que certaines fonctions des modules usuels `List` et `Array` ne sont pas disponibles en OCaml 3.12. J'essaie autant que possible de ne pas les utiliser, ou alors de les redéfinir si je m'en sers.

2.3 Question de programmation

La question de programmation pour ce texte était donnée en page 5, et était assez courte :

> "Implanter l'algorithme qui, à partir d'une table T du jeu de taquin, calcule les coordonnées du trou et la valeur de $\pi_2(T)$."

2.4 Réponse à l'exercice requis

2.4.1 Structures de données

On doit pouvoir représenter T, une table du jeu de taquin.

On utilisera la structure de données suggérée par l'énoncé :

> "Une table T du jeu de taquin est codée par un tableau carré (encore noté T) où, pour $i, j \in [|0, n-1|]$, $T[i, j]$ désigne le numéro de la pièce située en ligne i et colonne j, le trou étant codé par l'entier 0."

La figure 1 présente trois tables du jeu pour $n = 4$; la première, notée T_1 , est aléatoire, la troisième est la table finale T_f et la deuxième, notée T_2, peut être qualifiée d'intermédiaire : il est possible de passer en un certain nombre de coups de T_1 à T_2, puis de T_2 à T_f.

Par exemple, dans la table T_1 de la figure 1, $T_1[2, 0] = 5$ et le trou est situé à la position $[3, 1]$.

```
In [2]: type taquin = int array array;;

Out[2]: type taquin = int array array
```

2.4.2 Exemples de grilles

On reproduit les trois exemples ci-dessous :

```
In [3]: let t1 : taquin = [|
            [| 10;  6;  4; 12 |];
            [|  1; 14;  3;  7 |];
            [|  5; 15; 11; 13 |];
            [|  8;  0;  2;  9 |]
        |];;

Out[3]: val t1 : taquin =
          [|[|10; 6; 4; 12|]; [|1; 14; 3; 7|]; [|5; 15; 11; 13|]; [|8; 0; 2; 9|]|]


In [4]: let t2 : taquin = [|
            [|  0;  1;  2;  4 |];
            [|  3;  6; 10; 12 |];
            [|  5;  7; 14; 11 |];
            [|  8;  9; 15; 13 |]
        |];;

Out[4]: val t2 : taquin =
          [|[|0; 1; 2; 4|]; [|3; 6; 10; 12|]; [|5; 7; 14; 11|]; [|8; 9; 15; 13|]|]


In [5]: let tf : taquin = [|
            [|  0;  1;  2;  3 |];
            [|  4;  5;  6;  7 |];
            [|  8;  9; 10; 11 |];
            [| 12; 13; 14; 15 |]
        |];;

Out[5]: val tf : taquin =
          [|[|0; 1; 2; 3|]; [|4; 5; 6; 7|]; [|8; 9; 10; 11|]; [|12; 13; 14; 15|]|]
```

2.4.3 Permutations

On peut essayer d'obtenir les deux permutations, $\sigma(T)$ et $\sigma^B(T)$, pour une table T donnée.

Une permutation sera un simple tableau, de tailles n^2 (pour σ) et $n^2 - 1$ (pour σ^B), qui stocke en case i la valeur $\sigma(T)[i]$.

```
In [6]: type permutation = int array ;;
```

```
Out[6]: type permutation = int array
```

```
In [7]: let sigma (t : taquin) : permutation =
            (* Initialisation *)
            let n = Array.length t in
            let sigm = Array.make (n * n) 0 in
            (* Remplissons sigm *)
            for i = 0 to n - 1 do
                for j = 0 to n - 1 do
                    sigm.( i * n + j ) <- t.(i).(j)
                done;
            done;
            sigm
            (* On aurait aussi pu faire
            Array.init (n * n) (fun ij -> t.(ij mod n).(j / n))
            *)
        ;;
```

```
Out[7]: val sigma : taquin -> permutation = <fun>
```

Exemples :

```
In [8]: sigma t1;;
        sigma t2;;
        sigma tf;;
```

```
Out[8]: - : permutation = [|10; 6; 4; 12; 1; 14; 3; 7; 5; 15; 11; 13; 8; 0; 2; 9|]
```

```
Out[8]: - : permutation = [|0; 1; 2; 4; 3; 6; 10; 12; 5; 7; 14; 11; 8; 9; 15; 13|]
```

```
Out[8]: - : permutation = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15|]
```

C'était facile. Maintenant pour la permutation de Boustrophédon, $\sigma^B(T)$. On va quand même stoquer le 0, en position 0.

```
In [9]: let print = Printf.printf;;
```

```
Out[9]: val print : ('a, out_channel, unit) format -> 'a = <fun>
```

```
In [10]: let sigmaB (t : taquin) : permutation =
             (* Initialisation *)
             let n = Array.length t in
             let sigm = Array.make ((n * n) - 1) 0 in
             let nbzero = ref 0 in
```

```
        (* Remplissons sigm *)
        for i = 0 to n - 1 do
            for j = 0 to n - 1 do
                if i mod 2 = 0
                then (* gauche à droite *)
                    if t.(i).(j) = 0 then
                        incr nbzero
                    else
                        sigm.( i * n + j - !nbzero ) <- t.(i).(j)
                else (* droite à gauche *)
                    if t.(i).(n - j - 1) = 0 then
                        incr nbzero
                    else
                        sigm.( i * n + j - !nbzero ) <- t.(i).(n - j - 1)
            done;
        done;
        sigm
    ;;
```

```
Out[10]: val sigmaB : taquin -> permutation = <fun>
```

Exemples :

```
In [11]: sigmaB t1;;
         sigmaB t2;;
         sigmaB tf;;
```

```
Out[11]: - : permutation = [|10; 6; 4; 12; 7; 3; 14; 1; 5; 15; 11; 13; 9; 2; 8|]
```

```
Out[11]: - : permutation = [|1; 2; 4; 12; 10; 6; 3; 5; 7; 14; 11; 13; 15; 9; 8|]
```

```
Out[11]: - : permutation = [|1; 2; 3; 7; 6; 5; 4; 8; 9; 10; 11; 15; 14; 13; 12|]
```

2.4.4 Un déplacement

On a 4 déplacements possibles, $\{N, E, S, O\}$.

```
In [12]: type deplacement = Nord | Est | Sud | Ouest ;;
```

```
Out[12]: type deplacement = Nord | Est | Sud | Ouest
```

On va avoir besoin d'une fonction qui trouve la position (i, j) du trou :

```
In [13]: let ou_est (x : int) (t : taquin) : (int * int) =
             let n = Array.length t in
             let ij = ref (0, 0) in
             for i = 0 to n - 1 do
                 for j = 0 to n - 1 do
                     if t.(i).(j) = x then
                         ij := (i, j)
                 done;
             done;
             !ij
         ;;

         let ou_est_trou = ou_est 0 ;;

Out[13]: val ou_est : int -> taquin -> int * int = <fun>


Out[13]: val ou_est_trou : taquin -> int * int = <fun>


In [14]: let copie (t : taquin) : taquin =
             Array.map (Array.copy) t
         ;;

Out[14]: val copie : taquin -> taquin = <fun>


In [15]: let unmouvement (t : taquin) (dir : deplacement) : taquin =
             let n = Array.length t in
             let i, j = ou_est_trou t in
             let tsuivant = copie t in
             match dir with
             | Nord ->
                 if i = 0
                 then
                     failwith "Can't go north here"
                 else begin
                     tsuivant.(i).(j) <- tsuivant.(i - 1).(j);
                     tsuivant.(i - 1).(j) <- 0;
                     tsuivant
                 end;
             | Est ->
                 if j = n - 1
                 then failwith "Can't go east here"
                 else begin
                     tsuivant.(i).(j) <- tsuivant.(i).(j + 1);
                     tsuivant.(i).(j + 1) <- 0;
                     tsuivant
                 end;
```

```
            | Sud ->
                if i = n - 1
                then failwith "Can't go south here"
                else begin
                    tsuivant.(i).(j) <- tsuivant.(i + 1).(j);
                    tsuivant.(i + 1).(j) <- 0;
                    tsuivant
                end;
            | Ouest ->
                if j = 0
                then failwith "Can't go west here"
                else begin
                    tsuivant.(i).(j) <- tsuivant.(i).(j - 1);
                    tsuivant.(i).(j - 1) <- 0;
                    tsuivant
                end;
        ;;
```

```
Out[15]: val unmouvement : taquin -> deplacement -> taquin = <fun>
```

```
In [16]: let t1' = unmouvement t1 Nord ;;

         sigma t1';;
```

```
Out[16]: val t1' : taquin =
           [|[|10; 6; 4; 12|]; [|1; 14; 3; 7|]; [|5; 0; 11; 13|]; [|8; 15; 2; 9|]|]


Out[16]: - : permutation = [|10; 6; 4; 12; 1; 14; 3; 7; 5; 0; 11; 13; 8; 15; 2; 9|]
```

Ça semble fonctionner comme dans l'exemple du texte.

2.4.5 Test de la parité de σ^B

Le critère suivant permet de savoir si une table de taquin est jouable, i.e, si on peut la résoudre :

T est jouable si et seulement si $\sigma^B(T)$ est paire.

```
In [17]: let nb_inversions (sigm : permutation) : int =
             let nb = ref 0 in
             let m = Array.length sigm in
             for i = 0 to m - 1 do
                 for j = i + 1 to m - 1 do
                     if sigm.(i) > sigm.(j) then
                         incr nb
                 done;
             done;
             !nb
         ;;
```

```
Out[17]: val nb_inversions : permutation -> int = <fun>
```

```
In [18]: let est_paire (sigm : permutation) : bool =
             ((nb_inversions sigm) mod 2) = 0
         ;;

Out[18]: val est_paire : permutation -> bool = <fun>
```

On peut vérifier que les trois tables de l'énoncé ont bien une permutation de Boustrophédon paire :

```
In [19]: est_paire (sigmaB t1);;
         est_paire (sigmaB t2);;
         est_paire (sigmaB tf);;

Out[19]: - : bool = true


Out[19]: - : bool = true


Out[19]: - : bool = true
```

Et l'exemple de l'énonce qui n'est plus *jouable* :

```
In [20]: let tnof = copie tf in
         tnof.(0).(1) <- 2;
         tnof.(0).(2) <- 1;
         est_paire (sigmaB tnof);;

Out[20]: - : bool = false
```

```
In [21]: let est_jouable (t : taquin) : bool =
             est_paire (sigmaB t)
         ;;

Out[21]: val est_jouable : taquin -> bool = <fun>
```

2.4.6 Fonctions demandées

- On a déjà écrit la fonction $\pi_1(T)$, `nb_inversions`.
- Pour $\pi_2(T)$, on doit réfléchir un peu plus.

```
In [22]: let pi_1 (t : taquin) : int =
             nb_inversions (sigma t)
         ;;
```

```
Out[22]: val pi_1 : taquin -> int = <fun>
```

Pour $\pi_2(T)$, on peut être inquiet de voir dans la définition de cette distance la table finale, qui est l'objectif de la résolution du problème, mais en fait les tables finales T_f ont toutes la même forme : en case (i, j) se trouve $i \times n + j$!

On commence par définir la norme ℓ_1, sur deux couples (i, j) et (x, y) :

$$\ell_1 : (i, j), (x, y) \mapsto |i - x| + |j - y|$$

```
In [23]: let norme_1 (ij : int * int) (xy : int * int) : int =
             let i, j = ij and x, y = xy in
             abs(i - x) + abs(j - y)
         ;;

Out[23]: val norme_1 : int * int -> int * int -> int = <fun>
```

Puis la distance définie $d(T[i, j])$ dans l'énoncé :

```
In [24]: let distance (t : taquin) (i : int) (j : int) : int =
             let n = Array.length t in
             let valeur = t.(i).(j) in
             let ifin, jfin = valeur / n, valeur mod n in
             norme_1 (i, j) (ifin, jfin)
         ;;

Out[24]: val distance : taquin -> int -> int -> int = <fun>
```

Et enfin la fonction $\pi_2(T)$ est facile à obtenir :

```
In [25]: let pi_2 (t : taquin) : int =
             let n = Array.length t in
             let d = ref 0 in
             for i = 0 to n - 1 do
                 for j = 0 to n - 1 do
                     if t.(i).(j) != 0
                     then
                         d := !d + (distance t i j)
                 done;
             done;
             !d
         ;;

Out[25]: val pi_2 : taquin -> int = <fun>
```

2.4.7 Exemples

On prend l'exemple du texte avec T_1.

- $d(T_1[0,3]) = 6$?

```
In [26]: distance t1 0 3;;

Out[26]: - : int = 6
```

- $\pi_2(T) = 38$?

```
In [27]: pi_2 t1;;

Out[27]: - : int = 38
```

Ça semble bon !
Avec T_2 :

```
In [28]: distance t2 0 3;;

Out[28]: - : int = 4

In [29]: pi_2 t2;;

Out[29]: - : int = 26
```

Avec T_f, évidemment $\pi_2(T) = 0$ puisque T_f est résolue :

```
In [30]: distance tf 0 3;;

Out[30]: - : int = 0

In [31]: pi_2 tf;;

Out[31]: - : int = 0
```

2.5 Bonus ?

2.5.1 Complexité

- La fonction $\pi_1(T)$ est en $\mathcal{O}(N)$ en temps et mémoire, si $N = n^2$ est le nombre d'éléments dans le tableau.
- La fonction $\pi_2(T)$ est aussi en $\mathcal{O}(N)$ en temps et mémoire, si $N = n^2$ est le nombre d'éléments dans le tableau.

2.5.2 Autres idées

- On pourrait faire deux versions améliorées des fonctions π_1 et π_2 pour calculer $\pi(s_a(T))$ efficacement en fonction de $\pi(t)$ et $a \in \{N, E, S, O\}$. Sans écrire le code, elles seraient en temps constant, puisqu'il faut enlever et rajouter une (ou deux) valeurs dans une somme.

- On pourrait implémenter l'algorithme "ligne à ligne".

- On pourrait implémenter d'autres algorithmes de résolution, et les vérifier sur un exemple non trivial.

2.6 Conclusion

Voilà pour les deux questions obligatoires de programmation :

- on a décomposé le problème en sous-fonctions,
- on a essayé d'être fainéant, en réutilisant les sous-fonctions,
- on a fait des exemples et *on les garde* dans ce qu'on présente au jury,
- on a testé la fonction exigée sur un exemple venant du texte,
- et on a essayé d'en faire un peu plus (au début).

  Bien-sûr, ce petit notebook ne se prétend pas être une solution optimale, ni exhaustive.