



# TD 7 - Structures de données : Piles & Files

## 1 Les incontournables

**Exercice 1** Créer un type `Pierre`, `Feuille`, `Ciseaux` et créer une fonction `gagne` qui reçoit 2 gestes et donne le gagnant d'un jeu (0, 1 ou 2).

**Exercice 2** Écrire une fonction `liste_to_pile` : `'a list -> 'a Stack.t` qui "transforme une liste en pile" et où la tête de la liste arrive au sommet de la pile. Par exemple :

`liste_to_pile [1;2;3]` renvoie

1
2
3

**Exercice 3** On souhaite écrire quelques fonctions sur les piles :

- Écrire une fonction `remplis` : `int -> int -> int -> int Stack`, telle que `remplis n a b` renvoie une pile de `n` entiers choisis aléatoirement dans l'intervalle `[a, b]`. (On pourra utiliser la fonction `int` du module `Random` :

```
val int : int -> int Random.int bound returns a random integer between 0 (inclusive)
and bound (exclusive). bound must be greater than 0 and less than 230.
```

- On propose la fonction suivante pour afficher le contenu d'une pile :

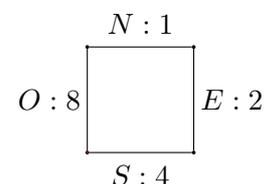
```
let affiche p =
  for i = 1 to Stack.length p do
    print_int (Stack.pop p);
    print_string "<- "
  done
;;
```

Quels sont les inconvénients de cette fonction ?

- Modifier cette fonction pour corriger ce problème et afficher le contenu de gauche à droite (sommet en bout d'affichage).

**Exercice 4** On se propose dans cet exercice de résoudre un labyrinthe de taille  $16 \times 16$  par un **parcours en largeur**.

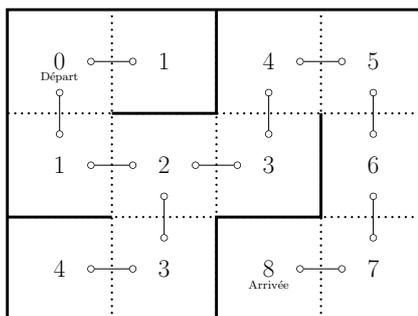
Notre labyrinthe sera modélisé par une liste de 256 entiers **téléchargeable sur le site**. Chaque nombre représentant une case, celle-ci étant entourée de 0 à 4 murs notés N, S, E, O. A chaque direction, on associe un entier, comme l'explique le schéma ci-contre. Pour déterminer le code d'une case, on ajoutera les valeurs des murs présents. Par exemple, la case  $\begin{array}{|c|} \hline \square \\ \hline \end{array}$  sera codée par le nombre  $1 + 8 = 9$ .



- On se propose d'écrire une fonction `mur` : `int -> int -> bool` telle que `mur c d` retourne `true` si le code `c` indique la présence d'un mur `d`  $\in \{1; 2; 4; 8\}$  et `false` sinon. Après avoir étudié l'écriture binaire de quelques valeurs de cases, écrire cette fonction.
- OCaml permet d'effectuer des opérations bit à bit sur les entiers. Par exemple les opérateurs `land` et `lor` permettent d'effectuer très rapidement les opérations ET et OU. Pour mieux comprendre : `22 land 51` donne 18 alors que `22 lor 51` donne 55 car  $22 = \overline{010110}^2$  et  $51 = \overline{110011}^2$ . En déduire une version très simple et rapide de `mur`.

On cherche à écrire une fonction `solve` : `int -> int -> string` qui trouve le chemin entre deux cases (dans notre cas, le chemin existe toujours et est unique). Par exemple `solve 146 196` donne "DBDBDBG"

3. Méthode naïve : On crée un tableau de 256 cases remplies de -1. On met la case de départ à 0. Tant que la valeur de la case d'arrivée vaut -1 : on met les cases voisines de 0 qui n'ont pas encore été visitées à 1 et recommence avec les cases voisines de 1, puis de 2....



4. Améliorer le précédent algorithme en stockant dans une file les cases à visiter.  
 5. Terminer le travail demandé en affichant le chemin.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

## 2 Pour s'entraîner

**Exercice 5** Écrire une fonction `split_parite : int Stack.t -> int Stack.t * int Stack.t` qui sépare une pile d'entiers en deux piles : la première ne contient que les nombres pairs de la pile de départ, la seconde que les nombres impairs (l'ordre n'est pas important). Par exemple :

$$\text{split\_parite} \begin{array}{|c|} \hline 1 \\ \hline 3 \\ \hline 4 \\ \hline 6 \\ \hline 5 \\ \hline \end{array} \text{ donne : } \left( \begin{array}{|c|} \hline 6 \\ \hline 4 \\ \hline \end{array}, \begin{array}{|c|} \hline 5 \\ \hline 3 \\ \hline 1 \\ \hline \end{array} \right)$$

**Exercice 6** Écrire une fonction `alterne : int Stack.t -> int Stack.t` qui reçoit une pile contenant autant de nombres pairs que de nombres impairs et renvoie une pile où les nombres pairs et impairs sont alternés une fois sur deux. Que se passe-t-il si la pile contient plus de nombres pairs que de nombres impairs ? Modifier la fonction pour que dans ce cas, le "surplus" soit stocké en haut de la pile renvoyée.

**Correction 1** On peut se limiter à 5 cas dans le filtrage :

```
type chifumi = Pierre | Feuille | Ciseaux;;

let gagne c1 c2 = match c1, c2 with
| a, b when a = b -> 0
| Pierre, Feuille -> 2
| Feuille, Ciseaux -> 2
| Ciseaux, Pierre -> 2
| _, _ -> 1
;;
```

**Correction 2** Il faut penser à retourner la liste, on crée une fonction auxiliaire pour remplir la pile, par exemple :

```
let liste_to_pile l =
  let p = Stack.create() in
  let rec remplis = function
    [] -> p;
  | h::t -> Stack.push h p; remplis t
  in remplis (List.rev l)
;;
```

ou ainsi,

```
let liste_to_pile l =
  let rec remplis p = function
    [] -> p;
  | h::t -> Stack.push h p; remplis p t
  in remplis (Stack.create()) (List.rev l)
;;
```

**Correction 3** 1. En utilisant la fonction proposée :

```
let remplis n a b =
  let random_int a b = a + Random.int (b-a+1) in
  let p = Stack.create() in
  for i = 1 to n do
    Stack.push (random_int a b) p
  done;
  p
;;
```

ou une solution récursive proposée par Léo ROYON :

```
let remplis n a b =
  let nombre a b = a + Random.int (b-a+1) and p = Stack.create() in
  let rec remplisRec = function
    | 0 -> p
    | n -> Stack.push (nombre a b) p ; remplisRec (n-1)
  in remplisRec n
;;
```

2. Le problème de cette fonction, c'est qu'elle vide la pile en même temps qu'elle l'affiche.

3. Il faut mémoriser l'objet avant de l'afficher de manière à pouvoir le remettre dans la pile.

```
let affiche p =
  let aux = Stack.create () and x = ref 1 in
  for i = 1 to Stack.length p do
    Stack.push (Stack.pop p) aux
  done;
  print_string "|";
  for i = 1 to Stack.length aux do
    x := Stack.pop aux;
    Stack.push !x p;
    print_string "└->└";
    print_int !x;
  done
;;
```

#### Correction 4

1. Une solution récursive : on décale les bits du mot jusqu'au mur voulu :

```
let rec mur c = function
  | 1 -> c mod 2 = 1
  | d -> mur (c/2) (d/2)
;;
```

De manière plus proche de ce qui vient d'être expliqué avec la fonction `lsr` qui décale les bits d'un nombre vers la droite : `9 lsr 1` retourne 4 :

```
let rec mur c = function
  | 1 -> c mod 2 = 1
  | d -> mur (c lsr 1) (d lsr 1)
;;
```

ou encore bien plus rapidement :

```
let mur c d = c / d mod 2 = 1;;
```

2. Avec la remarque donnée, on peut faire ainsi :

```
let mur c d = not (c land d = 0);;
```

3. Une solution "naïve" :

```
let remplis l debut fin =
  let dist = Array.make 256 (-1) and pas = ref 0 in
  dist.(debut) <- 0;
  while dist.(fin) = -1 do
    for i = 0 to 255 do
      if dist.(i) = !pas then begin
        if not (mur l.(i) 1) && dist.(i-16) = -1
          then dist.(i-16) <- !pas + 1;
        if not (mur l.(i) 2) && dist.(i+1) = -1
          then dist.(i+1) <- !pas + 1;
        if not (mur l.(i) 4) && dist.(i+16) = -1
          then dist.(i+16) <- !pas + 1;
        if not (mur l.(i) 8) && dist.(i-1) = -1
          then dist.(i-1) <- !pas + 1;
      end;
    done;
  pas := !pas + 1;
```

```

        done;
    dist
;;

```

4. A faire ...

5. Dans le code ci dessous, next permet de trouver pour une case donnée, la case suivante où aller pour rejoindre le départ.

```

let next l t = fonction
    | c when not (mur l.(c) 1) && t.(c-16) = t.(c)-1 -> (c-16,"B")
    | c when not (mur l.(c) 2) && t.(c+1) = t.(c)-1 -> (c+1,"G")
    | c when not (mur l.(c) 4) && t.(c+16) = t.(c)-1 -> (c+16,"H")
    | c when not (mur l.(c) 8) && t.(c-1) = t.(c)-1 -> (c-1,"D")
;;

let solve l debut fin =
    let r = remplis l debut fin and
        c = ref fin and chem = ref "" in
        while !c <> debut do
            let a, b = next l r !c in
                c := a;
                chem := b ^ !chem
        done;
    !chem
;;

```

**Correction 5** Une solution :

```

let split_parite p =
    let x = ref 0
    and pairs = Stack.create ()
    and impairs = Stack.create () in
    while not (Stack.is_empty p) do
        x := Stack.pop p;
        if !x mod 2 = 0 then
            Stack.push !x pairs
        else
            Stack.push !x impairs
    done;
    pairs, impairs;;

```

**Correction 6**

1. Proposition de solution :

```

let alterne p =
    let pairs, impairs = split_parite p in
    while not (Stack.is_empty pairs) do
        Stack.push (Stack.pop pairs) p;
        Stack.push (Stack.pop impairs) p
    done;
    p
;;

```

2. S'il n'y a pas autant la pile n'est pas vide à la fin.

3. Une solution pour remédier au problème :

```
let alterne p =
  let pairs, impairs = split_parite p in
  let nb_pairs = Stack.lenght pairs in
  let nb_impairs = Stack.lenght impairs in
  while not (Stack.is_empty pairs) && not (Stack.is_empty impairs) do
    if nb_pairs >= nb_impairs then
      begin
        Stack.push (Stack.pop pairs) p;
        Stack.push (Stack.pop impairs) p
      end
    else
      begin
        Stack.push (Stack.pop impairs) p;
        Stack.push (Stack.pop pairs) p
      end
    end
  done;
  while not (Stack.is_empty pairs) do
    Stack.push (Stack.pop pairs) p
  done;
  while not (Stack.is_empty impairs) do
    Stack.push (Stack.pop impairs) p
  done;
  p
;;
```