# 1

# *Parcours de tableaux*

« *Ordre, Permutations, Jeux.* »

## *1.1 Jeu d'échecs*

Sur un échiquier, on représentera chaque case par ses coordonnées $(i, j)$, la case en bas à gauche étant de coordonnées $(0, 0)$. Sur un tel échiquier, en un coup, un cavalier peut se déplacer de la case $(i, j)$ vers celles d'entre les 8 positions suivantes qui correspondent effectivement à une case de l'échiquier (abscisse et ordonnée comprises entre 0 et 7) : $(i-2, j+1)$, $(i-1, j+2)$, $(i+1, j+2)$, $(i+2, j+1)$, $(i+2, j-1)$, $(i+1, j-2)$, $(i-1, j-2)$ et $(i-2, j-1)$.

Question 1

Écrire une fonction OCaml qui donne toutes les cases accessibles en $p$ coups au plus à partir d'une case $(i_0, j_0)$.

Question 2

Écrire une fonction OCaml qui indique si toutes les cases sont accessibles à partir d'une case $(i_0, j_0)$ donnée, et si oui, quel est le plus petit nombre de coups permettant d'atteindre à partir de cette case n'importe quelle autre case de l'échiquier.

CORRIGÉ

Question 1

Choisissions déjà la structure de données. Définissions un type

```
type case = int * int
```

et donnons-nous une fonction `est_valide (c : case) : bool` qui renvoie `true` si $c$ est bien une case de l'échiquier (i.e $0 \leq$ `fst c`, `snd c` $\leq 7$), et `false` sinon. Réalisons tout de suite que quel que soit le nombre de coups, on ne pourra jamais atteindre plus de 64 cases et définissons des tableaux

```
let nombre_max_coups = 256 (* nombre max de coups *)
let coups = Array.make_matrix nombre_max_coups 64 (0,0)
let ncases = Array.make nombre_max_coups 0
```

tel que `coups[i,j]` désigne la $j$ème des cases atteignables en exactement $i$ coups et `ncases[i]` le nombre de cases atteignables en exactement $i$ coups - et non atteignables en moins de coups. La seule difficulté de la fonction et de penser à « faire le ménage »en vérifiant, à chaque fois qu'on ajoute une case, si elle n'a pas déjà été atteinte. Il est à noter qu'il suffit de vérifier ceci pour des valeurs de $i$ (numéros des coups) de même parité que le numéro courant : si on part d'une case blanche, en un nombre par de coups, on sera forcément sur une case blanche, et en un nombre impair sur une case noire.

```
(* Retourne la k-ième case atteignable a partir de *)
(* la case c selon l'ordre arbitraire de l'énoncé *)
let case_suivante (c:case) (k:int) : case =
  assert (1 <= k && k <= 8);
  let i, j = c in
  let positions = [|
    (i-2, j+1); (i-1, j+2); (i+1, j+2); (i+2, j+1);
    (i+2, j-1); (i+1, j-2); (i-1, j-2); (i-2, j-1)
  |] in
  positions.(k-1)

(* Vérifie si la case c a déjà été atteinte. On est à *)
(* l'étape pcour, et on cherche c parmi les cases *)
(* atteintes à un coup i de même parité que pcour, *)
(* pcour compris *)
let existe_deja (c:case) (pcour:int) : bool =
  let i = ref (pcour mod 2) in
  try
    while !i <= pcour do
      for j=0 to ncases.(!i)-1 do
        if coups.(!i).(j) = c then raise Exit
      done;
      i := !i + 2
    done;
    false
  with Exit -> true

let accessibles (i0:int) (j0:int) (p:int) : unit =
  assert (p < nombre_max_coups);
  coups.(0).(0) <- (i0, j0);
  ncases.(0) <- 1;
```

```
for p_courant=1 to p do
  (* on ajoute les cases atteignables en p coups *)
  (* i.e en 1 coup depuis les cases atteignables en p-1 coups *)
  for idx = 0 to ncases.(p_courant-1)-1 do
    let case_courante = coups.(p_courant-1).(idx) in
    for k=1 to 8 do
      let cs = case_suivante case_courante k in
      if est_valide cs && not (existe_deja cs p_courant) then (
        (* On insère la nouvelle case *)
        let nb_cases = ncases.(p_courant) in
        coups.(p_courant).(nb_cases) <- cs;
        ncases.(p_courant) <- nb_cases + 1;
      )
    done
  done
done
```

Question 2

Il n'y a pas grand chose à modifier : il suffit de remarquer qu'on a forcément atteint toutes les cases atteignables dès que l'ajout d'un nouveau coup n'aporte rien. Il suffit de modifier très légèrement la procédure précédente pour remplacer la boucle « for » sur la variable `p_courant` par une boucle « while » où l'on s'arrête dès que `ncases[p_courant]` vaut zéro. Il est alors facile de voir si toutes les cases sont atteintes (il faut simplement additionner les valeurs des `ncases[i]` pour voir si on trouve 64, surtout ne pas tester pour toute case si elle est dans le tableau `coups`, ceci serait trop long). Le lecteur pourra essayer d'évaluer le coût de ces fonctions. Sans réfléchir, on trouve quelque chose d'exponentiel en le nombre de coups, sauf si on se souvient que l'échiquier n'a que 64 cases !

⋆⋆⋆⋆⋆⋆⋆⋆

## 1.2 Médiane d'un tableau

On dispose d'un tableau $A$ de $n$ entiers distincts.

Question 1

Écrire une fonction OCaml `echange (a:int array) (i:int) (j:int) : unit` qui échange les éléments d'indices $i$ et $j$ du tableau $A$.

Question 2

Soient $g$ et $d$ deux entiers, $1 \leq g \leq d \leq n$. Posons $\alpha = A[g]$. On désire effectuer une permutation des éléments de $A$ d'indice compris entre $g$ et $d$, qui soit telle qu'après la permutation il existe un entier *pivot*, $(g \leq pivot \leq d)$ vérifiant :

- $A[pivot] = \alpha$,
- pour tout $i$ compris entre $g$ et $pivot$, $A[i] \leq \alpha$,
- pour tout $i$ compris entre $pivot + 1$ et $d$, $A[i] > \alpha$,
- les éléments de $A$ d'indice strictement inférieur à $g$ ou strictement supérieur à $d$ restent inchangés.

Par exemple, si $n = 6$, si les éléments de $A$ sont 5, 8, 7, 3, 9, 15, si $g = 1$ et $d = 5$, les éléments de $A$ après la permutation seront 5, 3, 7, 8, 9, 15 ou 5, 7, 3, 8, 15, 9, ou... (il n'y a pas unicité des permutations possibles), et *pivot* sera égal à 3. Écrire une fonction OCaml qui effectue la permutation et donne la valeur de *pivot* sans utiliser un autre tableau que $A$. Combien d'affectations (c'est à dire d'instructions « `<-` ») et de tests nécessite-t-elle ?

### Question 3

On appelle *médiane* de $A$ un couple $(i, \alpha)$ tel que $1 \leq i \leq n$, $A[i] = \alpha$, et $E(n/2)$ éléments de $A$ sont inférieurs strictement à $\alpha$ ($E(x)$ est la partie entière de $x$). Proposer une fonction de calcul de la médiane de $A$ utilisant la fonction de la question 2.

## CORRIGÉ

### Question 1

La question 1 est élémentaire.

### Question 2

Pour la fonction donnant la permutation (c'est une fonction dite de *partition*), on maintient deux indices $i$ et $j$ tels qu'à tout moment les éléments d'indice compris entre $g$ et $i$ sont tous inférieurs ou égaux à $\alpha$, tandis que les éléments d'indice compris entre $j$ et $d$ sont tous supérieurs ou égaux à $\alpha$.

```
let partition (a:int array) (g:int) (d:int) : int =
  let alpha = a.(g) in
  let i = ref g and j = ref d in
  while !i < !j do (
    while a.(!j) >= alpha && !j > g do
      decr j
    done;
    while a.(!i) <= alpha && !i < d do
      incr i;
    done;
    if (!i < !j)
      then echange a !i !j
  ) done;
  echange a !j g;
  !j (* on renvoie le pivot *)
```

Question 3

Partitionnons le tableau $A$ complet à l'aide de la fonction précédente. Si $pivot < n/2$, c'est-à-dire s'il y a plus d'éléments de $A$ supérieurs au pivot que d'éléments inférieurs au pivot, alors la médiane est à droite du pivot, il faut poursuivre la recherche à droite. Dans le cas contraire, il faut poursuivre la recherche à gauche du pivot.

```
let mediane (a:int array) : int =
  let n = Array.length a in
  let g, d = ref 0, ref (n-1) in
  while !d - !g > 1 do
    let pivot = partition a !g !d in
    if pivot < n/2
      then g := pivot+1
      else d := pivot-1
  done;
  !g
```

⋆⋆⋆⋆⋆⋆⋆⋆

## 1.3 Travail sur des tableaux

On se donne un tableau $T$ d'entiers de taille $N$.

Question 1

On se donne un entier $p < N$ et un entier $s$, et on recherche dans le tableau $T$ les indices $k$ tels que $\sum_{i=k}^{p+k} T[i] \geq s$. Écrire une fonction OCaml qui effectue cette recherche. Donner en fonction de $N$ et $p$ un ordre de grandeur du nombre de tests et d'opérations arithmétiques effectuées lors de l'exécution de votre fonction. Pouvez-vous l'améliorer ?

Question 2

On suppose maintenant que $T[0] < T[1] < T[2] < ... < T[N-1]$. Pouvez-vous tenir compte de cette information pour obtenir une fonction plus rapide ?

Question 3

Proposer une fonction qui affiche les triplets d'entiers $i, j, k$ avec $j < i < N$ tels que $i^2 + j^2 = k^2$. Pouvez-vous l'améliorer ?

CORRIGÉ

Question 1

Première idée : une fonction intuitive, mais quelque peu idiote

```
let recherche (t:int array) (p:int) (s:int) : unit =
  let n = Array.length t in
  for k=0 to n-p do
    let somme = ref 0 in
    for i=k to p+k do
      somme := !somme + t.(i)
    done;
    if !somme >= s then
      (print_int k; print_newline ())
  done
```

Si on réalise que lorsque l'on passe de l'étape $k$ à l'étape $k+1$ dans l'algorithme précédent, deux termes seulement de la somme changent, on obtient la fonction suivante, plus rapide si $p$ est plus grand que 2.

```
let recherche (t:int array) (p:int) (s:int) : unit =
  let n = Array.length t in
  let somme = ref 0 in
  for i=0 to p do
    somme := !somme + t.(i)
  done;
  if !somme >= s then print_endline "0";

  for k=1 to n-1-p do
    somme := !somme - t.(k-1) + t.(k+p);
    if !somme >= s then
      (print_int k; print_newline ())
  done
```

### Question 2

L'idée est bien sûr de procéder par dichotomie pour savoir à partir de quel rang on a $\sum_{i=k}^{p+k} T[i] \geq s$. On peut procéder comme suit :

```
(* Renvoie l'indice k à partir duquel on a la propriété *)
let dichotomie (t: int array) (p:int) (s:int) : int =
  let n = Array.length t in
  let min = ref 0 and max = ref (n-p-1) in
  while !min <= !max do
    let m = (!min + !max) / 2 in
    if somme t m p >= s
      then max := m - 1
      else min := m + 1
  done;
  !max + 1
```

Plusieurs variantes permettant de faire moins d'additions en utilisant le « truc » de la question 1 sont possibles.

Question 3

La première idée (idiote !) est de faire une boucle sur $i$, $j$ et $k$ en utilisant le fait que $i^2 < k^2 = i^2 + j^2 < 2 \times i^2$ donc $i < k < \sqrt{2} \times i < 1,5 \times i$ d'où $i + 1 \leq k \leq \lfloor 1,5 \times i \rfloor$ :

```
let triplets (n:int) =
  for i=1 to n-1 do
    for j=1 to i-1 do
      let i' = float_of_int i in
      let sup = int_of_float (1.5 *. i') in
      for k=i+1 to sup do
        if i*i + j*j = k*k then
          Printf.printf "%d %d %d\n" i j k
      done
    done
  done
```

on peut ensuite se dire que le seul $k$ qui a une chance de convenir est $\sqrt{i^2 + j^2}$ (si c'est un entier !), ce qui donne :

```
let triplets (n:int) =
  for i=1 to n-1 do
    for j=1 to i-1 do
      let r = float_of_int (i*i + j*j) in
      let k = Float.round (sqrt r) in
      if k*.k = r then
        Printf.printf "%d %d %d\n" i j (int_of_float k)
    done
  done
```

⋆⋆⋆⋆⋆⋆⋆⋆

## *1.4 Deuxième plus grand élément*

Soit $T$ un tableau unidimensionnel d'entiers de taille $n \geq 1$. On cherche à déterminer l'indice du deuxième plus grand élément de $T$ (celui qui viendrait en deuxième position si on rangeait les éléments de $T$ par ordre décroissant).

Question 1

Écrire une fonction OCaml qui calcule l'indice du deuxième plus grand élément de $T$ en parcourant une fois le tableau $T$.
QUel nombre de comparaisons entre éléments du tableau effectue votre fonction ?

Question 2

Pour imaginer une meilleure solution, penser à un tournoi de tennis. Le deuxième meilleur joueur n'est pas forcément le finaliste mais figure parmi les adversaires du

gagnant. Pourquoi ?
Donner un algorithme qui utilise cette analogie, et déterminer le nombre de comparaisons entre élémentts du tableau qu'il effectue.
Écrire alors la fonction OCaml correspondante.

CORRIGÉ

Question 1

Première solution évidente :

```
let deuxieme (t:int array) : int =
  let n = Array.length t in
  let max_1 = ref (max t.(0) t.(1)) in
  let max_2 = ref (min t.(0) t.(1)) in
  for i=2 to n-1 do
    if t.(i) > !max_1 then
      (max_2 := !max_1; max_1 := t.(i))
    else if t.(i) > !max_2 then
      max_2 := t.(i)
  done;
  !max_2
```

Les $n-2$ comparaisons `t.(i) > !max_1` sont toujours effectuées. Les $n-2$ comparaisons `t.(i) > !max_2` le sont aussi dans le pire cas d'un tableau décroissant. Avec les comparaisons implicites de `min` et `max` (une seule suffirait mais serait moins lisible), on arrive à $2n-3$, ce qui est intuitif : il faut $n-1$ comparaisons pour trouver le plus grand parmi $n$ (chacun sauf le plus grand doit avoir été comparé à plus grand que lui) et $n-2$ pour trouver le plus grand parmi les $n-1$ éléments restants (même argument).

Question 2

C'est clair : tout autre joueur que le gagnant et ses adversaires malheureux est de classement $\geq 3$ puisqu'on connaît deux meilleurs joueurs que lui.

Pour l'algorithme, on simule un tournoi de tennis. On transforme les éléments du tableau en feuilles puis on les fusionne en mettant l'indice du plus grand élément des deux à la racine. En itérant jusqu'à n'obtenir qu'un seul arbre, on crée effectivement un arbre de tournoi dont l'indice du vainqueur est la racine.

On effectuera $n-1$ comparaisons pour trouver le gagnant (maximum) car chaque match élimine un joueur. On cherche alors le maximum parmi les $\lceil \log_2 n \rceil$ joueurs battus par le gagnant.

On commence par définir un type d'arbre et une fonction utilitaire :

```
type tree = Node of (tree * int * tree) | Leaf of int

let etiquette tree = match tree with
  | Node(_, x, _) | Leaf x -> x
```

```
let deuxieme (t:int array) : int =
  let n = Array.length t in
  (* liste de tous les joueurs *)
  let joueurs = List.init n (fun i -> Leaf i) in

  (* simulation des duels, retourne un arbre de tournoi *)
  let rec jouer jou gagn = match jou with
    | j1::j2::q ->
      let i1, i2 = etiquette j1, etiquette j2 in
      let g = if t.(i1) > t.(i2) then i1 else i2 in
      let res = Node (j1, g, j2) in
      jouer q (res::gagn)
    | [g] when gagn = [] -> g
    | l -> jouer (l@gagn) [] (* un seul joueur ou liste vide *)
  in
  let matchs = jouer joueurs [] in

  (* on descend dans l'arbre pour trouver *)
  (* le plus grand adversaire du premier *)
  let premier = etiquette matchs in
  let rec traverser res m = match res with
    | Leaf _ -> m
    | Node (g, gagn, d) ->
      if etiquette g = premier then
        let idx = etiquette d in
        let m = max m t.(idx) in
        traverser g m
      else
        let idx = etiquette g in
        let m = max m t.(idx) in
        traverser d m
  in traverser matchs min_int
```

⋆⋆⋆⋆⋆⋆⋆⋆

# *1.5 Tri d'un petit nombre d'éléments*

Soit $n$ un entier $\geq 2$. Le but de cet exercice est d'étudier les algorithmes de tri d'un tableau de n éléments entiers, pour $n = 3, 4, 5$ et 10. On ne compte que les comparaisons entre éléments du tableau, et on note $Comp(n)$ leur nombre.

Question 1

Donner un algorithme, et écrire la fonction OCaml correspondante, pour trier un tableau de taille $n = 3$ avec $Comp(n) = 3$.
Même question avec $n = 4$ et $Comp(n) = 5$.
Même question avec $n = 5$ et $Comp(n) = 7$.

### Question 2

On suppose que $n = 10$. On demande de préciser le nombre de comparaisons pour chacun des deux algorithmes basés sur les principes suivants (mais on ne demande aucune fonction OCaml dans cette question) :

Algorithme 1. Faire deux listes de 5 éléments, utiliser deux fois l'algorithme précédent pour $n = 5$ et fusionner les deux listes triées obtenues.

Algorithme 2. Faire 5 paires d'éléments et trier les 5 plus grands de chaque paire à l'aide de l'algorithme pour $n = 5$. Puis insérer les éléments restants dans la liste formée.

## CORRIGÉ

Commençons par définir une fonction qui range dans l'ordre deux éléments d'un tableau `t` en positions `i` et `j` :

```
let ranger (t:int array) (i:int) (j:int) : unit =
  if t.(i) > t.(j) then (
    let tmp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- tmp
  )
```

### Question 1

Très facile avec 3 éléments : on met le plus petit élément en position 1 en le comparant aux deux autres, puis on classe les deux éléments restants :

```
let tri_3 (t:int array) : unit =
  ranger t 0 1;
  ranger t 0 2;
  ranger t 1 2
```

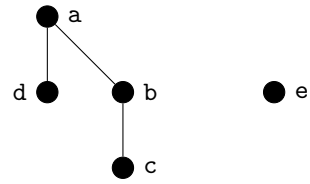
Pour 4 éléments : on classe les paires en position (0, 1) et (2, 3), puis on met les deux plus petits en position 0 et le plus grand des deux plus grands en position 3. Enfin, reste à classer les éléments en positions 1 et 2 :

```
let tri_4 (t:int array) : unit =
  ranger t 0 1;
  ranger t 2 3;
  ranger t 0 2;
  ranger t 1 3;
  ranger t 1 2
```

Une autre solution est de trier les 3 premiers éléments (3 comparaisons) puis d'insérer le quatrième, ce qui coute 2 comparaisons supplémentaires en procédant par dichotomie, c'est-à-dire en commençant par l'élément du milieu.

Avec 5 éléments : trier les 4 premiers éléments puis insérer le cinquième par dichotomie est ici trop coûteux : il faut 3 comparaisons pour insérer un élément dans une liste de taille 4. Cette solution nous coûterait $5 + 3 = 8$ comparaisons.

Voici une solution en 7 comparaisons : on prend les 4 premiers éléments, on classe les paires (0, 1) et (2, 3) et on compare les deux plus grands éléments. La situation se résume à l'aide du diagramme (où $\texttt{a} \geq \texttt{b} \geq \texttt{c}$ et $\texttt{a} \geq \texttt{d}$) :




On insère alors le cinquième élément $e$ dans la chaîne $a, b, c$ de longueur 3, ce qui coûte 2 comparaisons. Enfin, on insère le quatrième élément $d$ parmi les éléments inférieurs à $a$ dans l'ensemble $a, b, c, e$ ordonné, ce qui coûte deux comparaisons.

On commence par définir une fonction utilitaire :

```
let remplir_tableau (t:int array) (v,w,x,y,z) =
  t.(0) <- v; t.(1) <- w; t.(2) <- x; t.(3) <- y; t.(4) <- z

let tri_5 (t:int array) : unit =
  (* construction du diagramme *)
  ranger t 0 1; ranger t 2 3;
  let a, b, c, d = if t.(1) > t.(3)
    then t.(1), t.(3), t.(2), t.(0)
    else t.(3), t.(1), t.(0), t.(2)
  and e = t.(4) in

  (* insertion de e dans la liste, puis de d *)
  if e > b then
    if e > a then (* ordre c,b,a,e *)
      if d > b then
        remplir_tableau t (c,b,d,a,e)
      else
        if d > c then
          remplir_tableau t (c,d,b,a,e)
        else
          remplir_tableau t (d,c,b,a,e)
    else (* ordre c,b,e,a *)
      if d > b then
        if d > e then
          remplir_tableau t (c,b,e,d,a)
        else
          remplir_tableau t (c,b,d,e,a)
      else
        if d > c then
          remplir_tableau t (c,d,b,e,a)
        else
          remplir_tableau t (d,c,b,e,a)
  else
```

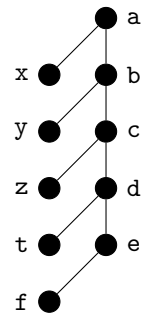
```
if e > c then (* ordre c,e,b,a *)
  if d > e then
    if d > b then
      remplir_tableau t (c,e,b,d,a)
    else
      remplir_tableau t (c,e,d,b,a)
  else
    if d > c then
      remplir_tableau t (c,d,e,b,a)
    else
      remplir_tableau t (d,c,e,b,a)
else (* ordre e,c,b,a *)
  if d > c then
    if d > b then
      remplir_tableau t (e,c,b,d,a)
    else
      remplir_tableau t (e,c,d,b,a)
  else
    if d > e then
      remplir_tableau t (e,d,c,b,a)
    else
      remplir_tableau t (d,e,c,b,a)
```

La fonction est volumineuse mais ne pose pas de difficulté particulière.

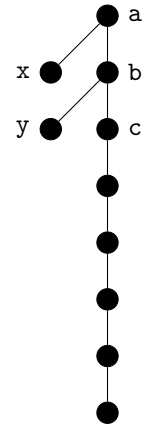
Question 2

Pour l'algorithme 1, le décompte est facile : on utilise deux fois la fonction `tri_5`, ce qui conduit à 14 comparaisons. Reste à fusionner deux listes triées de 5 éléments. Or la fusion de deux listes triées de tailles respectives $m$ et $n$ requiert $m+n-1$ comparaisons, soit ici 9 comparaisons. On obtient un total de 23 comparaisons.

Pour l'algorithme 2, après les 5 premières comparaisons et la fonction `tri_5` appliquée aux 5 plus grands, on a la situation suivante :



Reste à insérer les 4 éléments $x$, $y$, $z$ et $t$ dans la chaîne ordonnée de longueur 6 $f, e, d, c, b, a$. On commence par insérer $z$ dans $d, e, f$ (2 comparaisons) puis on insère $t$ dans la liste qui comprend $f$, $e$ et éventuellement $z$ (encore deux comparaisons). On a :




Insérer $x$ dans la chaîne de longueur 7 dont le plus grand élément est b requiert 3 comparaisons. Enfin, insérer $y$ dans une chaîne de longueur au plus 7 (longueur 6 si $x > b$, longueur 7 sinon) requiert également 3 comparaisons.

Le nombre final de comparaisons est donc $5 + 7 + 2 + 2 + 3 + 3 = 22 = Comp(10)$.

Il se trouve que l'algorithme 2 est optimal. Bien sûr, c'est difficile à prouver ! D'une manière générale, la fonction $Comp\_min(n)$ qui donne le nombre minimal de comparaisons nécessaires pour trier $n$ éléments est très mal connue : on ne connait pas d'expression exacte, juste un équivalent asymptotique en $O(n \log_2 n)$.

⋆⋆⋆⋆⋆⋆⋆⋆

## *1.6 Représentation de permutations*

On dit qu'une permutation $p$ des entiers 0, 1, ..., $n$ est représentée sous forme normale si elle est stockée dans un tableau `p` tel que `p.(i)` contienne l'image de `i` par la permutation.

Question 1

Écrire une fonction qui prend comme entrée une permutation $p$ sous forme normale et calcule le nombre de points fixes de $p$.

Question 2

Écrire une fonction qui prend comme entrée une permutation $p$ sous forme normale et calcule le nombre de cycles de $p$.

Question 3

On dit qu'une permutation est stockée sous forme de cycles si elle est stockée dans un tableau `c` construit comme suit : on stocke la permutation cycle par cycle, on regarde le plus petit élément de chaque cycle, on ordonne les cycles par ordre décroissant de leur plus petit élément. Par exemple, la permutation :

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| p[i] | 6 | 1 | 0 | 7 | 3 | 4 | 2 | 5 |

a trois cycles, l'un réduit à 1, qui est un point fixe, l'autre de longueur 3, avec 0 qui a pour image 6 qui a pour image 2 qui a pour image 0, et le troisième de longueur 4, avec 4 qui a pour image 3 qui a pour image 7 qui a pour image 5 qui a pour image 4. Les plus petits éléments de ces cycles sont respectivement 1, 0 et 3. En écrivant d'abord le cycle qui contient 3, puis celui qui contient 1, puis celui qui contient 0, on obtient le tableau `c` :

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| c[i] | 3 | 7 | 5 | 4 | 1 | 0 | 6 | 2 |

Écrire une fonction qui construit `c` à partir de `p`.

Question 4

Montrer que l'application qui a un tableau `p` associe le tableau `c` est bijective. Proposer un algorithme qui prend comme entrée une permutation sous forme de cycles et donne en sortie la même permutation représentée sous forme normale. Écrire la fonction correspondante.

CORRIGÉ

Question 1

Fonction élémentaire.

```
let nb_points_fixes (p:int array) : int =
  let pts = ref 0 in
  Array.iteri (fun i e -> if i = e then incr pts) p;
  !pts
```

Question 2

On peut par exemple parcourir les cycles un à un, en marquant les nombres comme « vus » au fur et à mesure ; on repère qu'un cycle se termine lorsqu'on retombe sur le premier élément du cycle courant. Pour trouver le cycle suivant, on cherche simplement un nombre qui n'a pas encore été vu. La fonction requiert de manipuler deux boucles imbriquées, la boucle intérieure parcourant le cycle et la boucle extérieure allant de cycle en cycle.

```
let nb_cycles (p:int array) : int =
  let n = Array.length p in
  let vu = Array.make n false in
  let nb_cycles = ref 0 in

  for i=0 to n-1 do
    if not vu.(i) then ( (* nouveau cycle *)
      incr nb_cycles;
      vu.(i) <- true;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        j := p.(!j)
      done;
    )
  done;
  !nb_cycles
```

Question 3

Un algorithme possible est de parcourir les cycles comme dans la question 2, en remplissant en même temps un tableau « min » tel que `min[j]` contienne le plus petit élément du j-ième cycle. Puis on fait un deuxième parcours des cycles pour remplir le tableau de sortie : le premier cycle à parcourir sera celui tel que `min[j]` soit maximal, et ainsi de suite jusqu'à avoir écrit tous les cycles.

```
let transforme (p:int array) : int array =
  let n = Array.length p in
  let nb_cycles = ref 0 in
  let min = Array.make n 0 in
  let vu = Array.make n false in

  for i=0 to n-1 do
    if not vu.(i) then ( (* nouveau cycle *)
      min.(!nb_cycles) <- i;
      incr nb_cycles;
      vu.(i) <- true;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        j := p.(!j)
      done;
    )
  done;

  let sortie = Array.make n 0 in
  let k = ref 0 in (* première case libre du tableau sortie *)
  for j=0 to !nb_cycles-1 do (* écrire les cycles successivement *)
```

```
    (* recherche du max *)
    let idx_max = ref 0 in
    for i=1 to !nb_cycles-1 do
      if min.(i) > min.(!idx_max) then
        idx_max := i
    done;

    let max = min.(!idx_max) in
    min.(!idx_max) <- 0;
    sortie.(!k) <- max;
    incr k;
    let j = ref p.(max) in
    while !j <> max do (* parcours du cycle *)
      sortie.(!k) <- !j;
      incr k;
      j := p.(!j)
    done;
  done;
  sortie
```

Question 4

La seule question pour savoir si la transformation de la question 3 est réversible est de décider comment séparer le tableau en cycles : ensuite, chaque cycle est écrit dans l'ordre ($i$, $p(i)$, $p(p(i))$, etc.), donc il est facile de reconstruire $p$. Pour faire la séparation des cycles, il suffit de remarquer que l'ordre dans lequel on a écrit les cycles, chaque cycle commençant par son plus petit élément et les cycles étant écrits dans l'ordre décroissant, assure que dans un tableau `c` donnant une permutation sous forme de cycles, $j = c(i)$ est le début d'un nouveau cycle si et seulement si $j$ est minimal parmi $c(0)$, $c(1)$, ..., $c(i)$. Donc l'application est bien inversible.

La fonction de passage à une forme normale est en fait nettement plus simple que celle de la question 3. Il suffit de parcourir `c` en gardant en mémoire le minimum des valeurs vues jusqu'à présent, et en remarquant que $p(c(i)) = c(i+1)$ sauf en bout de cycle.

```
let inverse (c:int array) : int array =
  let n = Array.length c in
  let p = Array.make n 0 in
  let min = ref c.(0) in
  for i=0 to n-2 do (* trouver l'image de c(i) *)
    if c.(i+1) > !min
      then p.(c.(i)) <- c.(i+1)
      else (
        p.(c.(i)) <- !min;
        min := c.(i+1)
      )
  done;
  p.(c.(n-1)) <- !min; p
```

⋆⋆⋆⋆⋆⋆⋆⋆

# 1.7 Permutations

On considère un ensemble $E = \{0, ..., N-1\}$. On représente une permutation $p$ de $E$ par un tableau `p` de taille $N$ tel que l'image d'un élément $i$ de $E$ est `p.(i)`.

### Question 1

Écrire un algorithme qui, étant donné un tableau `p`, vérifie que `p` représente effectivement une permutation de $E$.

### Question 2

Écrire un algorithme qui décompose une permutation en cycles.

### Question 3

On ordonne les permutations par ordre lexicographique (l'ordre du dictionnaire). Par exemple, si $N = 4$, $0123 < 0213 < 1230 < 2013$. Écrire un algorithme qui associe à chaque permutation `p` la permutation suivante dans l'ordre lexicographique (quand elle existe). On pourra utiliser le plus grand entier $i$ tel que $p(i) < p(i+1)$.

### Question 4

Écrire un algorithme qui énumère toutes les permutations de $E$.

## CORRIGÉ

### Question 1

Il serait particulièrement maladroit de tester successivement si tous les nombres de 0 à $N-1$ sont dans `p`. Une solution simple consiste à marquer les images rencontrées.

```
let est_permutation (p:int array) : bool =
  let n = Array.length p in
  let vu = Array.make n false in
  Array.iter (fun e -> if 0 <= e && e < n then vu.(e) <- true) p;
  Array.fold_left (&&) true vu
```

### Question 2

Comme d'habitude, la bonne idée est de procéder comme on le ferait « à la main ». On part de 0 et on regarde ses images successives par la permutation en marquant tous les entiers rencontrés. On s'arrête lorsque l'on retombre sur 0. Puis on recommence avec le premier entier non marqué.

Exemple : Soit la permutation $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 0 & 5 & 2 & 3 \end{pmatrix}$. On trouve un premier cycle (0 4 2). Le premier élément non marqué est 1, on trouve un second cycle (1). Le premier élément non marqué est 3, on trouve le troisième et dernier cycle (3 5).

```
let afficher_cycles (p:int array) : unit =
  let n = Array.length p in
  let vu = Array.make n false in

  for i=0 to n-1 do (* recherche du premier élément non vu *)
    if not vu.(i) then (
      vu.(i) <- true;
      Printf.printf "(%d" i;
      let j = ref p.(i) in
      while !j <> i do (* parcours du cycle *)
        vu.(!j) <- true;
        Printf.printf " %d" !j;
        j := p.(!j)
      done;
      print_endline ")"
    )
  done
```

Question 3

La difficulté est de comprendre comment on calcule le suivant d'une permutation dans l'orde lexicographique.

Exemple : le suivant de 0 1 3 2 est 0 2 1 3
le suivant de 0 2 1 3 est 0 2 3 1
le suivant de 1 2 3 0 est 1 3 0 2

Il faut donc chercher le plus grand entier $i$ tel que $p(i)$, ..., $p(N-1)$ contienne un élément plus grand que $p(i)$. Il est très facile de voir que cela revient à chercher, comme indiqué dans l'énoncé, le plus grand entier $i$ tel que $p(i) < p(i+1)$. Le suivant est alors obtenu en remplaçant $p(i)$ par le plus petit élément qui lui soit supérieur parmi $p.(i+1)$, ..., $p.(N-1)$. On complète la permutation en triant par ordre croissant les éléments restants.

Exemple : Pour $p = 0\ 1\ 3\ 2$, le plus grand élément tel que $p(i) < p(i+1)$ est 1. On remplace 1 par le plus petit élément plus grand que 1 parmi 3 et 2, soit 2. On trie les nombres restants (2 et 3) par ordre croissant. On obtient ainsi 0 2 1 3.

Pour éviter d'avoir une fonction de tri à écrire, on peut également marquer les éléments recontrés dans la recherche du plus grand i tel que $p(i) < p(i+1)$.

Le programme de calcul de la permutation $q$ suivant dans l'ordre lexicographique une permutation $p$ fixée va suivre la méthode décrite ci-dessus. La fonction `max_croit` renvoie le plus grand entier $i$ tel que $p(i) < p(i+1)$ si un tel entier existe, $-1$ sinon.

```
let max_croit (p:int array) : int =
  let n = Array.length p in
  let i = ref (n-2) in
  while !i >= 0 && p.(!i) >= p.(!i+1) do
    decr i;
  done;
  !i
```

La fonction `inf` renvoie le plus petit entier parmi `p.(k+1)`, ..., `p.(N-1)` qui est plus grand que `p.(k)`.

```
let inf (p:int array) (k:int) : int =
  let n = Array.length p in
  let min = ref max_int in
  for i=k+1 to n-1 do
    if p.(i) < !min && p.(i) > p.(k) then
      min := p.(i)
  done;
  !min
```

On a finalement la fonction calculant la permutation suivant `p` :

```
let suivante (p:int array) : (int array) =
  assert (est_permutation p);

  let n = Array.length p in
  let q = Array.make n 0 in
  let seuil = max_croit p in
  if (seuil < 0) then [||] (* pas de permutation suivante *)
  else

  (* recopier les éléments avant le seuil *)
  (* en les marquant comme vus *)
  let vu = Array.make n false in
  for i=0 to seuil-1 do
    q.(i) <- p.(i);
    vu.(p.(i)) <- true
  done;

  let inf = inf p seuil in
  q.(seuil) <- inf;
  vu.(inf) <- true;

  (* complèter q avec les éléments non vus *)
  let idx = ref (seuil+1) in
  for i=0 to n-1 do
    if not vu.(i) then (
      q.(!idx) <- i;
      vu.(i) <- true;
      incr idx
    )
  done;
  q
```

Question 4

Il suffit d'appliquer l'algorithme de la question précédente de manière itérative à partir de la permutation identité.

```
let affiche_perm (p:int array) : unit =
  let n = Array.length p in
  print_string "(";
  Array.iteri (fun i e -> if i = n-1
    then Printf.printf "%d)\n" e
    else Printf.printf "%d " e
  ) p

let enumere (n:int) : unit =
  (* Initialisation de la permutation identité *)
  let identite = Array.init n (fun i -> i) in
  let cur = ref identite in

  while !cur <> [||] do
    affiche_perm !cur;
    cur := suivante !cur
  done
```

⋆⋆⋆⋆⋆⋆⋆⋆

# 2

# *Jouer avec les mots*

« *Reconnaissance, Cosntruction, Codage.* »

# 3

# *Stratégies gloutonnes*

« *Le meilleur du moment, pour trouver le meilleur.* »

# 4

# *Arborescences*

« *Un père, des fils. Une racine, des noeuds, des feuilles.* »

# 5

# *Graphes*

« *Des arêtes, des sommets, des poids.* »

# 6

# *Géométrie et Images*

« *Des représentations. Des figures. Des intersections. Des pavages.* »

# 7

# *Arithmétique et calculs numériques*

« *Écrire. Calculer. Résoudre.* »

# 8

# *Vers la récursivité*

« *Diviser pour régner. Faire moins pour faire plus.* »

# A

# *Un peu d'Algèbre*

## *A.1 Anneaux Booléens*

$F_2$ désignera le corps à deux éléments $\mathbb{Z}/2\mathbb{Z}$. Un *anneau Booléen* est un anneau commutatif unitaire $(A, +, *)$ dans lequel, pour tout $x \in A, x * x = x$ ($F_2$ est un exemple d'anneau booléen).

Si $E$ est un ensemble, on note $\mathcal{P}(E)$ l'ensemble des sous-ensembles finis de $E$ et $\mathcal{AB}[E]$ l'ensemble des fonctions de $\mathcal{P}(E)$ dans $F_2$ qui sont nulles sauf pour un nombre fini d'éléments de $\mathcal{P}(E)$. $\mathcal{AB}[E]$ est muni de l'addition et de la multiplication : $(f+g)(x) = f(x) + g(x)$ et $(fg)(x) = f(x)g(x)$. $\mathcal{AB}[E]$ muni de ces opérations est aussi un anneau booléen : c'est l'anneau booléen *engendré* par $E$.

Dans tout le problème, $E$ sera supposé fini et de cardinal $n$.