L'algorithme de Robinson, Schensted et Knuth

Option informatique — Toussaint 1997

Gilles Peskine

10 novembre 1997

1 Permutations

1.2 Représentation en Caml

On choisit de représenter une permutation par une liste d'entiers : l'élément d'ordre k de la liste est l'image de k par la permutation. La fonction `image` qui à une permutation σ et un entier k associe $\sigma(k)$ n'est autre que la fonction $n^{\text{ème}}$ élément.

```
let rec image = fun
    [] _ -> raise (Invalid_argument "image")
  | (t :: _) 0 -> t
  | (_ :: q) n -> image q (pred n);;
```

Pour tester si une liste est une permutation, on peut trier la liste et vérifier que la liste résultat est une liste d'entiers consécutifs en ordre croissant commençant à 0.

```
let est_permutation sigma =
  let tau = sort__sort (prefix <=) sigma
  in let rec vérifie i = function
        [] -> true
      | n :: q -> let j = succ i in n = j && vérifie j q
  in vérifie (-1) tau;;
```

Cette solution est purement fonctionnelle ; malheureusement, son temps d'exécution est de l'ordre de $n \lg n$. Voici une autre solution, moins élégante, mais dont le temps d'exécution est linéaire en la taille de la liste. Pour tester si une liste d'entiers est bien une permutation, on commence par déterminer le nombre n d'éléments de la liste. S'il s'agit d'une permutation, son image est $\{0, \ldots, n-1\}$. La liste est alors une permutation si et seulement si son image est contenue dans l'ensemble $\{0, \ldots, n-1\}$ et si la permutation est injective. La fonction `est_permutation` teste ces deux conditions. Un invariant de fin de boucle est : l'élément `atteint.(i)` est vrai si et seulement si i appartient à l'image de $\{0, \ldots, k\}$.

```
let est_permutation sigma =
  let s = vect_of_list sigma
  in let n = vect_length s
  in let atteint = make_vect n false
  in try for k = 0 to pred n do
        if s.(k) >= n || s.(k) < 0 || atteint.(s.(k)) then raise Exit;
        atteint.(s.(k)) <- true
      done; true
     with Exit -> false;;
```

La composée des permutations σ et τ associe à l'entier i l'entier $\sigma(\tau(i))$.

```
let compose sigma tau = map (image sigma) tau;;
```

La concise expression ci-dessus a malheureusement un temps d'exécution maximal quadratique. On peut rendre le temps d'exécution linéaire en utilisant un tableau.

```
let compose sigma tau =
  let s = vect_of_list sigma
  in map (vect_item s) tau;;
```

Une permutation est involutive si et seulement si sa composée avec elle-même est l'identité.

```
let involutive sigma =
  let tau = compose sigma sigma
  in let rec vérifie i = function
        [] -> true
      | n :: q -> let j = succ i in n = j && vérifie j q
  in vérifie (-1) tau;;
```

Pour former σ^{-1} (la réciproque de σ), on peut déterminer successivement $\sigma^{-1}(0), \ldots, \sigma^{-1}(n-1)$. La fonction auxiliaire `aux` détermine $\sigma^{-1}(n)$ et se rappelle récursivement pour calculer $(\sigma^{-1}(m))_{m>n}$; la fonction auxiliaire `indice_de` renvoie l'indice de n dans σ.

```
let inverse sigma =
  let rec indice_de i n = function
      [] -> raise (Invalid_argument "inverse")
    | t :: q -> if n = t then i else indice_de (succ i) n q
  and aux n = function
      [] -> []
    | _ :: q as l -> indice_de 0 n sigma :: aux (succ n) q
  in aux 0 sigma;;
```

L'algorithme naïf de calcul de l'inverse a un temps d'exécution quadratique en la taille de la liste dans le pire des cas. Une deuxième approche consiste à rendre la représentation de la liste plus symétrique en σ et σ^{-1} : on associe facilement à la liste `[`$\sigma(0)$ `; ... ;` $\sigma(n-1)$`]` la liste `[0,` $\sigma(0)$ `; ... ;` $n-1$`,` $\sigma(n-1)$`]`. Un tri sur la deuxième composante et l'échange des composantes échange les rôles de σ et σ^{-1} ; il ne reste plus qu'à supprimer la première composante. (En fait, le programme ne fait pas l'échange des composantes et supprime directement la deuxième composante). Cette méthode de calcul de l'inverse a l'efficacité du tri : de l'ordre de $n \lg n$ où n est la taille de la liste.

```
let rec numérote n = function
    [] -> []
  | t :: q -> (n, t) :: numérote (succ n) q;;
let inverse sigma =
  map fst (sort__sort (fun (_, x) (_, y) -> x <= y) (numérote 0 sigma));;
```

Voici enfin une troisième solution au calcul de l'inverse. Au lieu de calculer successivement les $\sigma^{-1}(i)$, on place successivement les i à la position $\sigma(i)$ dans l'inverse. Cette implémentation est linéaire, elle repose sur l'accès direct dans un tableau.

```
let inverse sigma =
  let n = list_length sigma
  in let inv = make_vect n 0
  in let rec aux i = function
      [] -> ()
    | t :: q -> inv.(t) <- i; aux (succ i) q
  in aux 0 sigma; list_of_vect inv;;
```

1.3 Recherche des inversions

La fonction `est_inversion` se passe de commentaire.

```
let est_inversion sigma i j = i < j && image sigma i > image sigma j;;
```

On fait la liste des inversions de $\sigma \in S_n$ en déterminant, pour chaque entier i compris entre 0 et $n-1$, la liste des $j > i$ tels que $\sigma(i) > \sigma(j)$. On obtient la liste des inversions triée en ordre lexicographique. Temps d'exécution et occupation en mémoire sont quadratiques ; c'est le mieux qu'on puisse faire puisque, par exemple, la permutation $(i \mapsto n-1-i) \in S_n$ a $n(n-1)/2$ inversions.

```
let rec ceux_qui pred = function
    [] -> []
  | t :: q -> if pred t then (t :: ceux_qui pred q) else (ceux_qui pred q);;
let liste_inversions sigma =
  let rec aux = function
      [] -> []
    | (i,i') :: q -> map (function (j,_) -> i,j)
                         (ceux_qui (function (_,j') -> i' > j') q)  @ aux q
  in aux (numérote 0 sigma);;
```

Pour compter les inversions, on peut lister les inversions puis en compter le nombre.

```
let nb_inversions sigma = list_length (liste_inversions sigma);;
```

On peut aussi réécrire `liste_inversions` en ne mémorisant que le nombre d'inversions et non la liste. On n'a plus non plus besoin de disposer des indices (i, j), seulement des images (i', j'), car $i < j$ si et seulement si j' est après i' dans la liste `sigma`. Le temps d'exécution reste quadratique, mais l'occupation en mémoire est linéaire.

```
let rec nombre_qui pred = function
    [] -> 0
  | t :: q -> if pred t then succ (nombre_qui pred q) else (nombre_qui pred q);;
let rec nb_inversions = function
    [] -> 0
  | k :: q -> nombre_qui (gt_int k) q + nb_inversions q;;
```

2 Tableaux de Young

2.1 Définitions

2.1.1 Exemple

La figure 1 présente tous les tableaux de Young de taille 4, contenant les entiers de 0 à 3. Il y en a 10. Quand aux involutions de S_4, ce sont :

$$\begin{matrix} (1,0,2,3) & (2,1,0,3) & (3,1,2,0) & (1,0,3,2) & (2,3,0,1) \\ (0,2,1,3) & (0,3,2,1) & (0,1,3,2) & (3,2,1,0) & (0,1,2,3) \end{matrix}$$

2.1.2 Analyse d'un tableau de Young

La forme d'un tableau est la liste des longueurs des lignes ; sa taille est la somme des longueurs des lignes.

```
let forme = map list_length;;
let taille y = it_list (prefix +) 0 (forme y);;
```

FIG. 1: Tableaux de Young de taille 4

0
1
2
3

0	1
2	
3	

0	1
2	3

0	2
1	
3	

0	2
1	3

0	3
1	
2	

0	1	2	3

0	1	2
3		

0	1	3
2		

0	2	3
1		

Je définis aussi une fonction vérifiant si une liste de listes d'entiers est un tableau de Young. Je précise qu'un tableau de Young a une forme $(n_1, \dots, n_p)$ avec $p \geq 0$ et $n_1 \geq \cdots \geq n_p \geq 1$; il faut tester $n_p \neq 0$ si $p \neq 0$.

Je vérifie, successivement, l'absence de ligne vide, la croissance de chaque ligne, et la croissance de chaque colonne. La croissance des longueurs de lignes est contenue dans cette deuxième partie.

```
let est_young y =
  let rec croissante = function
      [] | [_] -> true
    | a :: (b :: q as r) -> a <= b && croissante r
  and existe_non_vide = function
      [] -> false
    | [] :: q -> existe_non_vide q
    | (_ :: _) :: _ -> true
  and isole_têtes = function
      [] -> []
    | (t :: q) :: r -> (t, q) :: isole_têtes r
    | [] :: r -> if existe_non_vide r then raise Exit else []
  in let rec colonnes_croissantes y =
    y = [] ||
      try let colonne0, autres_colonnes = split (isole_têtes y)
        in croissante colonne0 && colonnes_croissantes autres_colonnes
      with Exit -> false (* il y a une ligne plus courte que la suivante *)
  in for_all (function [] -> false | _ -> true) y
    && for_all croissante y  && colonnes_croissantes y;;
```

2.2 Insertion dans un tableau de Young

L'insertion de i dans un tableau vide résulte en un tableau à une ligne `[i]`. L'insertion de i dans un tableau non vide consiste à bousculer par i un élément b de la première ligne du tableau et à insérer b dans le tableau privé de sa première ligne.

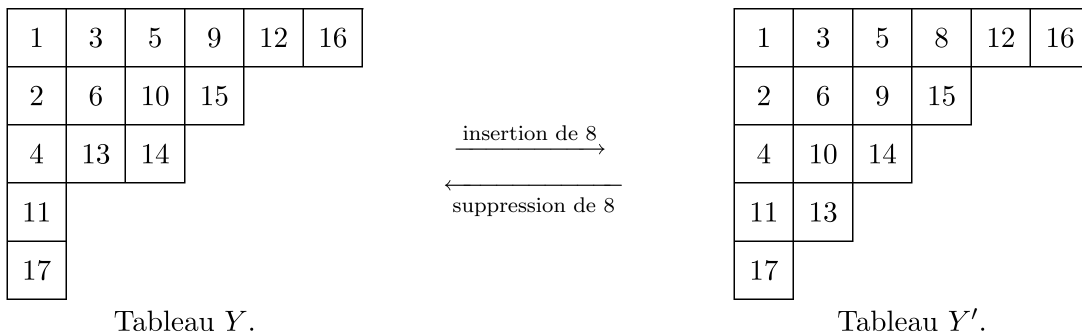
Bousculer par un entier i une liste triée sans double, c'est insérer i dans la liste à la place du plus petit élément supérieur à i. Cet élément est dit bousculé. Si i est plus grand que tous les éléments de la liste, on ajoute i en fin de liste et il n'y a pas d'élément bousculé.

```
let rec bouscule i = function
    [] -> [i], None
  | t :: q -> if t > i then (i :: q, Some t)
```

```
                       else (let r, b = bouscule i q in t :: r, b);;
let rec insertion y i = match y with
    [] -> [[i]]
  | l :: q -> match bouscule i l with
      l', None -> l' :: q
    | l', Some b -> l' :: insertion q b;;
```

2.3 L'algorithme inverse : suppression dans un tableau de Young

2.3.1 Étude d'un exemple



1	3	5	9	12	16
2	6	10	15		
4	13	14			
11					
17					

1	3	5	8	12	16
2	6	9	15		
4	10	14			
11	13				
17					

Tableau Y.

Tableau Y'.

Je commence par décrire l'algorithme de façon informelle, dans le cas du tableau Y' auquel je retire l'élément qui a entraîné l'apparition d'une colonne 1 à la ligne 3 pour retrouver Y.

Le 13 monte dans la troisième ligne, il y bouscule le 10. Le 10 monte dans la deuxième ligne, il y bouscule le 9. Le 9 monte dans la première ligne, il y bouscule le 8. On en était à la première ligne : le 8 disparaît.

2.3.2 Cas général et implémentation

Pour supprimer la colonne k de la ligne j, on retire l'élément x de la dernière colonne de la ligne j, il vient bousculer le plus grand élément de la ligne $j-1$ qui est inférieur à x, soit x' (x' existe toujours parce que l'élément en colonne k convient). x' bouscule un élément de la ligne $j-2$, et ainsi de suite jusqu'à avoir bousculé un élément de la ligne 1, qui disparaît du tableau.

La fonction `supprime` supprime le dernier élément de la ligne donnée comme argument et renvoie le couple (s, Y') où Y' est le tableau après suppression et s l'élément supprimé. La fonction `suppression` ne renvoie que Y'.

```
let supprime y j =
  let rec bouscule x = function
      a :: (b :: _ as q) -> if b >= x then a, x :: q
                                else let x', q' = bouscule x q in x', a :: q'
    | [a] when a <= x -> a, [x]
    | _ -> failwith "suppression" (* pas un tableau de Young *)
  and isole_dernier = function
      [] -> failwith "suppression" (* pas un tableau de Young *)
    | [d] -> d, []
    | t :: q -> match isole_dernier q with d, l -> d, t :: l
```

```
  and aux = fun
      _ [] -> failwith "suppression" (* pas un tableau de Young *)
    | 0 (l :: q) -> let d, l' = isole_dernier l
                    in d, ( if l' = [] then [] else l' :: q )
    | j (l :: q) -> let x, y' = aux (pred j) q
                    in let x', l' = bouscule x l
                    in x', l' :: y'
  in aux j y;;
let suppression y j = snd (supprime y j);;
```

3 L'algorithme rsk (Robinson, Schensted et Knuth)

3.1 L'algorithme direct

Prouvons (avec les notations de l'énoncé) que Q est un tableau de Young à chaque étape (donc, en particulier, à la fin). C'est vrai au départ (le tableau vide est un tableau de Young). Et chaque p_k est inséré à la fin d'une ligne (donc à la fin d'une colonne puisque P reste de Young), donc chaque q_k est inséré à la fin d'une ligne et d'une colonne : ceci assure que la croissance des lignes et des colonnes est maintenue dans Q ; et la contrainte de forme est aussi maintenue puisque Q et P ont même forme.

Pour implémenter l'algorithme RSK, on se sert d'une version modifiée de la fonction d'insertion, qui parcourt en même temps Q que P pour pouvoir y insérer l'élément q_k à la même place que p_k dans P.

```
let rsk qp =
  let rec mets_en_dernier a = function
      [] -> [a]
    | t :: q -> t :: mets_en_dernier a q
  and insertion (yp, yq) (qk, pk) = match yp with
      [] -> [[pk]], [[qk]]
    | l :: zp -> match bouscule pk l with
        l', None -> l' :: zp, ( match yq with [] -> [[qk]]
                                         | l0 :: zq -> mets_en_dernier qk l0 :: zq )
      | l', Some b -> match yq with
            [] -> failwith "rsk" (* ceci ne peut pas arriver *)
          | tq :: zq -> let yp', yq' = insertion (zp, zq) (qk, b)
                        in l' :: yp', tq :: yq'
  and aux yy = function
      [] -> yy
    | qkpk :: r -> aux (insertion yy qkpk) r
  in aux ([], []) qp;;
let RSK q p = rsk (combine (q, p));;
```

3.2 L'algorithme réciproque

Pour retrouver le couple de listes (q, p) qui a servi à fabriquer un couple (P, Q) de tableaux de Young, on retire de Q son plus grand élément (il est bien en fin de ligne et de colonne, donc le résultat est encore de Young) : c'est le dernier élément de q, et on supprime de P la position correspondante, ce qui donne le dernier élément de p. On récurse ensuite sur les tableaux (P', Q') obtenus après suppression, sachant que si P (et Q) est vide, alors p et q sont vides.

```
let ligne_du_maximum y =
  let rec dernier = function
      [] -> failwith "rsk_inverse" (* Q n'a pas une forme valide *)
    | [d] -> d
    | _ :: q -> dernier q
  in let rec indice_du_maximum j j_max max_courant = function
      [] -> j_max
    | l :: z -> let d = dernier l
                in indice_du_maximum (succ j) (if d > max_courant then j else j_max)
                                     (max d max_courant) z
  in match y with [] -> failwith "ligne_du_maximum"
    | l :: z -> indice_du_maximum 1 0 (dernier l) z;;
let retire_dernier_ligne j y =
  let rec isole_dernier = function
      [] -> failwith "rsk_inverse" (* pas un tableau de Young *)
    | [d] -> d, []
    | t :: q -> match isole_dernier q with d, l -> d, t :: l
  and aux = fun
      _ [] -> failwith "rsk_inverse" (* pas un tableau de Young *)
    | 0 (l :: z) -> let d, l' = isole_dernier l
                    in d, (if l' = [] then [] else l' :: z)
    | j (l :: z) -> let d, z' = aux (pred j) z in d, l :: z'
  in aux j y;;
let rsk_inverse yp yq =
  let rec aux = fun
      [] [] qp -> qp
    | yp yq qp -> let j = ligne_du_maximum yq
                  in let pk, yp' = supprime yp j
                  and qk, yq' = retire_dernier_ligne j yq
                  in aux yp' yq' ((qk, pk) :: qp)
  in aux yp yq [];;
let RSK_inverse (yp, yq) = split (rsk_inverse yp yq);;
```

4 Algorithme rsk et permutations

4.1 Minimums à gauche

Pour déterminer les minimums à gauche d'une liste, j'explore cette liste en maintenant accessible le minimum m desnombres précédents. Un élément t est minimum à gauche si et seulement si il est inférieur au m courant.

```
let minima_à_gauche liste =
  let rec minimums_à_gauche m = function
      [] -> []
    | t :: q -> if t <= m then t :: minimums_à_gauche t q
                  else minimums_à_gauche m q
  in match liste with [] -> [] | t :: q -> t :: minimums_à_gauche t q;;
```

4.2 Classification

4.2.1 Classes dans une liste de couples

Pour déterminer les classes dans une liste de couples, on reprend l'algorithme de recherche des minimums à gauche. En plus de former la liste des minimums à gauche, on forme la liste complémentaire. Si les minimums à gauche obtenus sont les couples de classe n, les minimums à gauche de la liste complémentaire sont les couples de classe $n+1$. On a fini quand la liste complémentaire est vide. On est assuré que l'algorithme se termine puisqu'une liste non vide a toujours un minimum à gauche, sa tête.

```
let classification q p =
  let rec extrait_minimums_à_gauche m = function
      [] -> [], []
    | (_, p as t) :: r ->
        let màg, autres = extrait_minimums_à_gauche (min p m) r
        in if p <= m then (t :: màg), autres else màg, t :: autres
  and classes = function
      [] -> []
    | (_, p as t) :: r -> let màg, autres = extrait_minimums_à_gauche p r
                          in (t :: màg) :: classes autres
  in classes (combine (q, p));;
```

4.2.2 Application à l'algorithme rsk

Un couple (q_k, p_k) est de classe 1 si et seulement si p_k est inférieur à tous ses prédécesseurs, c'est-à-dire si et seulement si p_k vient bousculer l'élément en colonne 0 lorsqu'il est inséré dans P à l'étape k de l'algorithme RSK. Un couple (q_k, p_k) est de classe 2 si et seulement si p_k est inférieur à tous ses prédécesseurs qui ne sont pas de classe 1 ; p_k est alors placé à l'étape k dans la colonne 1 de la première ligne de P. Plus généralement, (q_k, p_k) est de classe c si et seulement si p_k est placé dans la colonne $c-1$ de la première ligne de P à l'étape k.

Considérons maintenant la chaîne des insertions dans les lignes à partir de la deuxième. À une étape donnée, il n'y a que la première ligne de P et Q qui soit affectée si et seulement si l'étape a consisté à ajouter une colonne à la première ligne, c'est-à-dire si et seulement si le couple (q, p) inséré est le premier élément de sa classe. Sinon, dans P privé de sa première ligne, on insère l'élément bousculé hors de la première ligne.

Regardons l'ordre dans lequel les éléments sont bousculés hors de la première ligne. Chaque élément est bousculé par un élément de même classe. L'ordre des insertions dans P privé de sa première ligne est donc obtenu de la façon suivante : étant donné un élément p_k de classe c, on lui associe le précédent dans sa classe, ou rien s'il est le premier de sa classe ; l'image de la liste des (p_k) par cette transformation est la liste qui sert à former les lignes suivantes de P.

Étudions ensuite la formation de la première ligne de Q. Elle augmente de longueur à chaque fois que le couple inséré (q_k, p_k) est tel que p_k soit supérieur à tous les éléments dèjà présents dans la première ligne de P. En termes de classe, ceci signifie que (q_k, p_k) est le premier élément d'une classe, car p_k étant plus grand que tous ses prédécesseurs ne peut être minimum à gauche d'une liste extraite que s'il en est la tête. Ainsi, la première ligne de Q est formée des premiers éléments de chaque classe, nécessairement par classes croissantes puisque la ligne est croissante. Et Q privé de sa première ligne est formé de la même manière que Q, mais en n'insérant que les éléments qui ne sont pas premiers de classe, et ce en ordre croissant.

On tire de toutes ces observations une nouvelle manière de construire le couple de tableaux d'Young associé à deux listes q et p. On forme les classes de couples (q, p). La première ligne de Q est la suite

des premières composantes des couples qui sont premiers de leur classe, et la première ligne de P est la suite des deuxièmes composantes des couples qui sont derniers de leur classe. Il reste à former les tableaux Q' et P' égaux respectivement à Q et P privé de leur première ligne. On les obtient en appliquant le même algorithme à la nouvelle liste q' formée des éléments non insérés de Q et à la nouvelle liste p' formée des éléments non insérés de P dans l'ordre approprié comme expliqué deux paragraphes plus haut. Bien sûr, la récursion se termine lorsque p et q sont vides (elles le sont en même temps), auquel cas P et Q sont vides.

La fonction `varRSK` implémente cette nouvelle méthode.

```
let rec dernier = function
    [d] -> d
  | _ :: q -> dernier q
  | _ -> failwith "dernier";;
let varRSK lq lp =
  let rec arrange_p = fun
      [] _ -> []
    | (t :: q) c -> match trouve_p t c with
        None, c' -> arrange_p q c'
      | Some x, c' -> x :: arrange_p q c'
  and trouve_p = fun
      t ((t' :: _) :: _ as c) when t' = t -> None, c
    | t ((s' :: (t' :: _ as q)) :: r) when t' = t -> Some s', q :: r
    | t (l :: r) -> let résultat, reste = trouve_p t r in résultat, l :: reste
    | _ _ -> failwith "arrange_p"
  in let rec aux = fun
      [] [] -> [], []
    | (_::_ as lq) (_::_ as lp) ->
        let classes = classification lq lp
        in let yq0 = map (function classe -> fst (hd classe)) classes
           and yp0 = map (function classe -> snd (dernier classe)) classes
        in let lq' = ceux_qui (function qk -> not (mem qk yq0)) lq
           and lp' = arrange_p lp (snd (split (map split classes)))
        in let yp', yq' = aux lq' lp' in yp0 :: yp', yq0 ::yq'
    | _ _ -> failwith "varrsk" (* lq et lp n'ont pas la même longueur *)
  in aux lq lp;;
```

4.3 Pour terminer...

4.3.1 Classification symétrique

Soit $E = \{(q_0, p_0), \ldots, (q_r, p_r)\}$ un ensemble de couples d'entiers tels que si $i \neq j$ alors $p_i \neq p_j$ et $q_i \neq q_j$. On dira que (q_k, p_k) est de classe c si et seulement si c est le plus grand entier tel qu'il existe une suite d'indices $i_1, \ldots, i_c$ tel que $p_{i_1} < \cdots < p_{i_c}$ et $q_{i_1} < \cdots < q_{i_c}$.

Par exemple, (q, p) est de classe 1 si et seulement si $p' < p$ implique $q' > q$, ce qui est équivalent à ce que $q' < q$ implique $p' > p$. En conséquence directe de la définition, si on retire de E les couples de classe 1, la classe des couples restant diminue d'une unité.

Si on numérote les éléments de E de telle sorte que $q_0 < \cdots < q_r$, on voit que, d'une part, un couple (p_k, q_k) est de classe 1 si et seulement si p_k est minimum à gauche de la liste $[p_0, \ldots, p_k]$; et d'autre part, si on retire ces couples de classe 1, la classe des couples restant diminue d'une unité. La définition des classes correspond donc à la définition donnée dans ce cas particulier par l'énoncé.

4.3.2 Symétrisation de l'algorithme rsk

Sachant qu'on sait déterminer les classes d'un couple de listes sans faire intervenir l'ordre de la première liste, il ne reste qu'une étape de l'algorithme décrit en 4.2.2 qui dépende de l'ordre de la liste des (q_k), à savoir l'ordre dans lequel on insère les éléments de $Q' = (Q$ privé de sa première ligne). Or cet ordre peut aussi être déterminé d'une manière similaire à ce qu'on fait pour P : étant donné un élément q_k de classe c, on lui associe le suivant dans sa classe, ou rien s'il est le dernier de sa classe.

On s'aperçoit alors que l'algorithme ainsi obtenu traite les listes p et q de la même manière : l'expression « premier de sa classe » pour Q signifie « minimum de sa classe », ce qui est le sens de l'expression « dernier de sa classe » pour P. Si on échange p et q, les tableaux P et Q sont donc échangés.

4.3.3 Application aux permutations

On va utiliser une représentation des permutations plus symétrique entre une permutation et son inverse, évoquée en 1.2 (deuxième méthode de calcul de l'inverse) : on représente une permutation par n'importe quelle liste contenant chaque élément de son graphe une fois et une seule. Alors, si $\rho = [(a_0, b_0); \ldots; (a_r, b_r)]$ est une représentation de σ, $\rho' = [(b_0, a_0); \ldots; (b_r, a_r)]$ est une représentation de σ^{-1}.

L'application de l'algorithme RSK à ρ produit un couple de tableaux (P, Q) et son application à ρ' produit un couple de tableaux (Q, P). À une permutation, on associe ainsi un couple de tableaux (P, Q) tel que (Q, P) soit associé à la permutation réciproque.

En particulier, une permutation de S_n est involutive si et seulement si le couple de tableaux associé est de la forme (P, P) où P est un tableau de Young contenant les entiers $\{0, \ldots, n-1\}$. Comme tout tel tableau est l'image d'une permutation (on a vu que la fonction calculée par l'algorithme RSK est bijective), on dispose d'une bijection entre l'ensemble des tableaux de Young contenant les entiers $\{0, \ldots, n-1\}$ et l'ensemble des involutions de $\{0, \ldots, n-1\}$.