Module Permutation

1. Module de gestion de permutations et de conversions Tresses simples ↔ Permutation (d'après la bijection entre ces deux ensembles)
Type décrivant une permutation :

```
type permutation = int array
```

2. Gestion de permutations élémentaires.
Retourne la permutation identité.

```
let make_id n =
    let id = Array.make n 0 in
    for i = 0 to n − 1 do
        id.(i) ← i
    done;
    id
```

Teste si une permutation est l'identité.

```
let is_id p =
  let n = Array.length p in
  let ok = ref true and i = ref 0 in
  while !ok ∧ !i < n do
    ok := (p.(!i) = !i);
    i := !i + 1
  done;
  !ok
```

Retourne la transposition (i, j).

```
let make_transpose i j n =
    let t = make_id n in
    transpose t i j;
    t
```

Retourne la permutation correspondant à la tresse simple Δ.

```
let make_delta n =
    let delta = Array.make n 0 in
    for i = 0 to n − 1 do
        delta.(i) ← n − 1 − i
    done;
    delta
```

Teste si une permutation est Δ.

```
let is_delta p =
  let n = Array.length p in
  let ok = ref true and i = ref 0 in
  while !ok ∧ !i < n do
    ok := (p.(!i) = n − 1 − !i);
    i := !i + 1
  done;
  !ok
```

3. Inverse une permutation.

```
let inv permut =
    let n = Array.length permut in
    let inv = Array.make n 0 in
    for i = 0 to n − 1 do
        inv.(permut.(i)) ← i
    done;
    inv
```

4. Compose les permutations p1 et p2 : pour des questions d'optimisation il est possible de fournir un tableau (dest) qui sera rempli de manière à contenir la composée; dans le cas contraire un nouveau tableau rempli correctement sera retourné.

```
let compose ?dest p1 p2 =
    let n = Array.length p1 in
    let c = (
    match dest with
        None → Array.make n 0
      | Some c → c
    ) in
    for i = 0 to n − 1 do
        c.(i) ← p1.(p2.(i))
    done;
    c
```

5. Conjugué par Δ : $\tau(b) = \Delta^{-1}b\Delta$. On a également $\tau(\sigma_i) = \sigma_{n-i}$.

```
let tau p =
    let n = Array.length p in
    let q = Array.make n 0 in
    for i = 0 to n − 1 do
```

```
      q.(n − 1 − i) ← n − 1 − p.(i)
    done;
    q
```

6. Composition par une permutation.
Compose la permutation fournie par la transposition (i, j) (à droite) avec mutation de la permutation fournie (en O(1)).

```
let transpose permut i j =
    let tmp = permut.(i) in
    permut.(i) ← permut.(j);
    permut.(j) ← tmp
```

Compose à gauche par la transposition (i, j) sans mutation de la permutation fournie (en O(n)).

```
let compose_transpose_left permut i j =
  let n = Array.length permut in
  let res = Array.make n 0 in
  for k = 0 to n − 1 do
    let pk = permut.(k) in
    res.(k) ← (if pk = i then j
               else if pk = j then i
               else pk)
  done;
  res
```

Compose à droite par la transposition (i, j) sans mutation de la permutation fournie (en 0(n)).

```
let compose_transpose_right permut i j =
  let res = Array.copy permut in
  transpose res i j;
  res
```

7. Fonction renvoyant la permutation correspondant à une tresse fournie en argument. (Antimorphisme)

Fonctionnement : on part de la permutation identité et on applique successivement les transpositions correspondant aux générateurs : comme on compose à droite (en O(1)) et qu'on réalise un antimorphisme il est nécessaire de retourner la liste des générateurs.

```
let braid_to_permut (b : Braid.braid) =
    let permut = make_id b.Braid.size in
    List.iter (fun x → transpose permut ((abs x) − 1) (abs x)) (List.rev b.Braid.word);
    permut
```

8. Starting Set et Finishing Set pour une permutation :

- Le Starting Set correspond aux générateurs que l'on peut factoriser à gauche d'une tresse positive (qui divisent la tresse à gauche dans B_n^+). On a par ailleurs le résultat suivant : $i \in$ S(B) (où S(B) désigne le starting set de B) $\Leftrightarrow$ les brins i et $i+1$ se croisent dans la permutation correspondante. (c'est géométrique)

- Si F(B) désigne le Finishing Set de B, on a F(B) = S(rev(B)).

Renvoie la liste des inversions d'une permutation σ : il s'agit d'indices i tels que $\sigma(i+1) < \sigma(i)$.

La fonction renvoie une liste **triée** des inversions.

```
let consecutive_inversions permut =
  let n = Array.length permut in
  if n ≤ 1 then [] else (
    let l = ref [] in
    for i = 0 to n − 2 do
      if permut.(i) > permut.(i + 1)
      then l := i ::!l;
    done;
    List.rev !l
  )
```

Les listes retournées sont triées pour les deux fonctions ci-dessous.

```
let starting_set p = List.map ((+) 1) (consecutive_inversions p)
```

On a bien S(inv de permutation) = F(permutation) à la place d'utiliser le rev() de la tresse correspondante.

```
let finishing_set p = starting_set (inv p)
```

9. Opérations ensemblistes, où les ensembles sont représentés par des des **listes triées**.
Cherche si e est un sous-ensemble de f.

```
let rec is_subset e f = match (e, f) with
  | ([], _) → true
  | (_, []) → false
  | (x :: xs, y :: ys) → if x < y then false
                          else if x = y then is_subset xs ys
                               else is_subset (x :: xs) ys
```

Renvoie $e \backslash f$.

```
let rec set_difference e f = match (e, f) with
    | ([], _) → []
    | (_, []) → e
    | (x :: xs, y :: ys) → if x < y then x :: (set_difference xs (y :: ys))
                            else if x = y then set_difference xs ys
                                 else set_difference (x :: xs) ys
```

10. Mélange aléatoire d'un tableau avec l'algo de Knuth-Fisher-Yates, appliqué à la génération d'une permutation aléatoire.
Mélange sur place, modifie le tableau.

```
let shuffle t =
  Random.self_init ();
  let swap i j = let temp = t.(i) in
                 t.(i) ← t.(j);
                 t.(j) ← temp
  in
  let n = Array.length t in
  for i = n − 1 downto 0 do
    swap i (Random.int (i + 1));
  done;
  t
```

Renvoie une permutation aléatoire.

```
let random_permutation n = shuffle (make_id n)
```

11. Fonction d'affichage d'une permutation.

```
let print_permutation p =
  print_string "[";
  print_int p.(0);
  for i = 1 to Array.length p − 1 do
    Printf.printf "␣%d" p.(i);
  done;
  print_string "]"
```