

Permutations

April 13, 2022

Ce TP est un TP intermédiaire si vous avez fini **intégralement** les deux TP précédents sur les tris et le rendu de monnaie.

On considère ici des permutations de $[0; n - 1] = \{0, 1, \dots, n - 1\}$, c'est-à-dire des éléments de $\mathfrak{S}_n$ (c'est un S en gothique, on écrira donc Sn dans les commentaires en Python) l'ensemble des bijections de $[0; n - 1]$ dans lui même. **Remarque** en mathématiques, on utilise en général des permutations de $[1; n]$ mais le fait de commencer par 0 permettra d'utiliser directement ces valeurs comme indices.

Pour représenter $\sigma \in \mathfrak{S}_n$, on va utiliser un tableau $t_\sigma = [\sigma(0), \sigma(1), \dots, \sigma(n - 1)]$, c'est-à-dire que la valeur $\sigma(i)$ est à l'indice i dans t_σ :

$$\forall i \in [0; n - 1], \sigma(i) = t_\sigma[i]$$

1 Le groupe $(\mathfrak{S}_n, \circ)$

Comme les permutations ont même ensemble de départ et d'arrivée, on peut les composer et comme elles sont bijectives, la composée l'est également. Ainsi $\mathfrak{S}_n$ est stable par $\circ$. De plus, l'identité est une permutation ainsi que la réciproque d'une permutation.

```
[2]: def identite(n):  
    # renvoie la permutation identité de Sn  
    # BEGIN SOLUTION  
    p = []  
    for i in range(n):  
        p.append(i)  
    return p  
    # END SOLUTION
```

```
[3]: def compose(sigma, tau):  
    # renvoie sigma o tau la composée des deux permutations de Sn  
    # Cette fonction sera toujours appelée avec des permutations portant sur les  
    # mêmes éléments, il n'y a rien à vérifier  
    # BEGIN SOLUTION  
    n = len(sigma)  
    p = []  
    for i in range(n):  
        p.append(sigma[tau[i]])
```

```
return p
# END SOLUTION
```

```
[6]: def inverse(sigma):
      # renvoie la permutation inverse de sigma
      # BEGIN SOLUTION
      n = len(sigma)
      p = [ 0 ] * n
      for i in range(n):
          p[sigma[i]] = i
      return p
      # END SOLUTION
```

```
[4]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

```
[11]: def iter(sigma, k):
       # renvoie la permutation  $\sigma^k = \sigma \circ \dots \circ \sigma$ 
       # on pourra faire une fonction récursive
       # BEGIN SOLUTION
       if k == 0:
           return identite(len(sigma))
       return compose(sigma, iter(sigma, k-1))
       # END SOLUTION
```

```
[ ]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

Pour vérifier si un tableau **d'entiers** est une permutation, on peut créer un tableau de $n = \text{len}(t)$ booléens valant **False** et mettre **True** à l'indice p quand $t[i] = p$ pour i parcourant les indices. **Attention** $t[i]$ peut très bien ne pas être une valeur dans $[0; n - 1]$.

```
[7]: def test_permutation(t):
      # renvoie un booléen (True ou False) indiquant si le
      # tableau t est celui d'une permutation
      # BEGIN SOLUTION
      n = len(t)
      valid = [ False ] * n
      for i in range(n):
          p = t[i]
          if 0 <= p < n:
              valid[p] = True
      for b in valid:
          if not b:
              return False
      return True
      # END SOLUTION
```

```
[ ]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

Si σ est une permutation de $\mathfrak{S}_n$ et t un tableau à n éléments, il est possible de considérer le tableau permuté de t par σ , noté $\sigma(t)$ par extension et définie par $\sigma.t[i] = t[\sigma(i)]$ pour tout indice i . Ainsi si σ est la permutation correspondant au tableau $[3, 1, 2, 0]$ et $t = ['a', 'b', 'c', 'd']$ alors $\sigma.t = ['d', 'b', 'c', 'a']$.

```
[9]: def action(sigma, t):
    # renvoie le tableau sigma . t
    # on suppose t et sigma de même longueur
    # BEGIN SOLUTION
    st = []
    n = len(t)
    for i in range(n):
        st.append(t[sigma[i]])
    return st
    # END SOLUTION
```

```
[ ]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

2 Orbite

Soit $\sigma \in \mathfrak{S}_n$ et $i \in [0; n - 1]$. L'ensemble $Orb_\sigma(i) = \{i, \sigma(i), \sigma^2(i), \dots\}$. Cet ensemble est nécessairement fini, on peut donc le représenter par un tableau d'entiers.

Par exemple, si on considère $\sigma = [5, 2, 4, 1, 3, 0]$ on a $Orb_\sigma(0) = \{0, 5\} = Orb_\sigma(5)$ et $Orb_\sigma(1) = \{1, 2, 4, 3\}$.

```
[16]: def orbite(sigma, i):
    # renvoie Orb_sigma(i) comme un tableau sans répétitions
    # BEGIN SOLUTION
    o = [i]
    k = i
    while sigma[k] != i:
        k = sigma[k]
        o.append(k)
    return o
    # END SOLUTION
```

```
[ ]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

Les orbites de σ forment une partition de $[0; n - 1]$. Cela signifie que chaque élément est présent dans une orbite et une seule.

```
[ ]: def orbites(sigma):
    # renvoie les orbites de sigma dans un tableau d'orbites sans répétitions
    # BEGIN SOLUTION
    n = len(sigma)
    traites = [ False ] * n
    orbites = []
```

```

for i in range(n):
    if not traites[i]:
        traites[i] = True
        o = [i]
        k = i
        while sigma[k] != i:
            k = sigma[k]
            traites[k] = True
            o.append(k)
        orbites.append(o)
return orbites
# END SOLUTION

```

```
[19]: orbites([5,2,4,1,3,0]) # doit renvoyer par exemple [[0, 5], [1, 2, 4, 3]]
```

```
[19]: [[0, 5], [1, 2, 4, 3]]
```

3 Génération des permutations

Il peut être intéressant de générer toutes les permutations pour des problèmes d'exploration exhaustive comme on en verra en deuxième année. Pour le moment, imaginons qu'on souhaite casser un code, une manière bête mais sûre d'y arriver et de tous les tester.

Il existe de nombreuses approches, dont certaines sont très astucieuses pour économiser l'espace. On va voir ici une approche plus simple proche de celle vue dans le TP précédent.

On va prendre un point de vue plus proche de notre représentation en parlant de tableaux sans répétitions. On note $P(E)$ l'ensemble des tableaux sans répétitions dont les éléments sont dans E . On cherche donc à calculer $P([0; n-1])$. Pour cela, on va remarquer que

$$P(E) = \bigcup_{e \in E} \{[e] + t \mid t \in P(E \setminus \{e\})\}$$

Ici $+$ est la concaténation des tableaux. On en déduit une fonction récursive d'énumération.

On remarque $P(\emptyset) = \{[]\}$.

```

[21]: def retire(t, e):
    # renvoie un NOUVEAU tableau contenant les éléments de t sauf e
    # BEGIN SOLUTION
    tp = []
    for x in t:
        if x != e:
            tp.append(x)
    return tp
    # END SOLUTION

```

```
[ ]: # écrivez vos tests ici pour vous assurer que vos fonctions fonctionnent
```

```
[26]: def permutations(t):
      # renvoie P(t)
      # BEGIN SOLUTION
      if t == []:
          return [ [] ]
      p = []
      for e in t:
          for tp in permutations(retire(t, e)):
              p.append( [e] + tp )
      return p
      # END SOLUTION
```

```
[28]: permutations([0,1,2]) # doit renvoyer [[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0],
↪ [2, 0, 1], [2, 1, 0]]
```

```
[28]: [[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```