Groupes

November 18, 2021

1 Groupes

Marc Lorenzi

18 novembre 2021

[1]:
```
from utils import bijections
```

1.1 1. Introduction

1.1.1 1.1 Monoïdes, groupes

Définition. Un *monoïde* est un couple $(M, \star)$ où

- M est un ensemble.
- $\star$ est une loi de composition interne sur M.
- $\star$ est associative.
- $\star$ possède un élément neutre.

Si tout élément de M est inversible pour $\star$, $(M, \star)$ est un *groupe*. Si, de plus, $\star$ est commutative, $(M, \star)$ est un *groupe abélien*.

Dans ce notebook, nous allons définir quelques fonctions permettant de manipuler des ensembles *finis* munis d'une opération, et de déterminer les propriétés de cette opération.

Certains résultats de la *théorie des groupes* permettraient d'écrire des fonctions bien plus efficaces que les nôtres : les fonctions que nous écrirons seront pour la plupart du type « force brute ». Le tableau ci-dessous donne, pour chacun des problèmes que nous allons étudier, un ordre de grandeur de la complexité de nos futures fonctions en fonction du cardinal n du monoïde. La dernière colonne donne une idée du cardinal maximal envisageable si l'on veut des réponses « quasi-immédiates ».

Propriété	Complexité	Cardinal max
Commutativité	n^2	1000
Assoctiativité	n^3	100
Neutre	n^2	1000
Inverses	n^2	1000
Sous-groupes	2^n	20
Isomorphismes	$n!$	10

1.1.2 1.2 Les monoïdes et Python

On représente un couple $(M, \star)$ formé par un ensemble M et une loi de composition interne $\star$ sur M par un objet de la classe `Monoide`. Un tel objet possède entre autres

- Un champ `elts` qui est la liste des éléments de M.
- Un champ `op` qui est une fonction de 2 variables telle que pour tous $x, y \in M$, $\mathrm{op}(x, y) = x \star y$.

Pour créer le monoide $(M, \star)$ on appelle `Monoide(elts, table)`, où `elts` est la liste des éléments de M et `table` est une matrice telle que `table[i][j]` soit le produit du $i$ème et du $j$ème élément de `elts`. Le constructeur se charge de transformer cette matrice `table` en une fonction `M.op`.

Le constructeur possède un certain nombre de paramètres optionnels :

- Si le paramètre `commut` vaut `True` (resp `False`), $\star$ est commutative (resp pas commutative). Si `commut` vaut `None`, le constructeur se charge de vérifier la commutativité.
- Si le paramètre `assoc` vaut `True` (resp `False`), $\star$ est associative (resp pas associative). Si `assoc` vaut `None`, le constructeur se charge de vérifier l'associativité.
- Si le paramètre `neutre` vaut `None`, le constructeur recherche l'éventuel neutre de M. Sinon, le neutre de M est la valeur du paramètre.
- Si le paramètre `inverses` vaut `None`, le constructeur recherche les éventuels inverses des éléments de M et place leurs valeurs dans un dictionnaire `M.inverse`. Sinon, la valeur de ce paramètre est un dictionnaire. Nous en reparlerons le moment venu.

Bien entendu, mettre l'un de ces paramètres à une valeur différente de `None` est de la responsabilité de l'utilisateur. Si l'on **sait** qu'un monoïde de cardinal n a effectivement une opération associative, alors mettre `assoc` à la valeur `True` économise n^3 opérations lors de la création de ce monoïde.

Les fonctions permettant la recherche auomatique ne sont pas encore écrites ! Pour l'instant, nous ne pouvons pas mettre les paramètres oprionnels à `None`, mais cela viendra.

[2]:
```
class Monoide:

    def __init__(self, elts, table, commut=None, assoc=None, neutre=None,␣
↪inverses=None):
        self.elts = elts
        d = {}
        for i in range(len(elts)): d[elts[i]] = i
        self.op = lambda x, y: table[d[x]][d[y]]
        self.commut = False
        self.assoc = False
        self.neutre = None
        self.inverses = dict()

        if commut == None: self.commut = tester_commut(self)
        else: self.commut = commut

        if assoc == None: self.assoc = tester_assoc(self)
        else: self.assoc = assoc
```

```
        if neutre == None: self.neutre = trouver_neutre(self)
        else: self.neutre = neutre

        if inverses == None: self.inverse = trouver_inverses(self)
        else: self.inverse = inverses
```

La fonction `afficher_table` affiche la table de l'opération. La fonction `largeurs` est une fonction auxiliaire qui calcule les largeurs des colonnes de la table.

[3]:
```
def largeurs(M):
    m = 1 + max([len(str(x)) for x in M.elts])
    fmt1 = '%' + str(m) + 's|'
    fmt2 = ''
    for y in M.elts:
        m = 1 + max([len(str(M.op(x, y))) for x in M.elts])
        fmt2 += '%' + str(m) + 's'
    return fmt1 + fmt2
```

[4]:
```
def afficher_table(M):
    fmt = largeurs(M)
    print(fmt % tuple([''] + M.elts))
    #print(fmt % tuple((len(M.elts) + 1) * ['_']))
    for x in M.elts:
        print(fmt % tuple([x] + [M.op(x,y) for y in M.elts]))
```

On aimerait bien tester tout cela, mais il nous faudrait des monoïdes. Dans la section suivante, nous allons définir un certain nombre de monoïdes. Ils nous serivront d'exemples pour le reste du notebook.

1.2 2. Les exemples

1.2.1 2.1 Les entiers modulo n - addition

Soit $n \in \mathbb{N}^*$. Posons

$$\mathbb{Z}_n = [\![0, n-1]\!]$$

Pour tous $x, y \in \mathbb{Z}_n$, posons

$$x \oplus y = (x + y) \bmod n$$

Nous avons là un groupe de neutre 0. La fonction ci-dessous prend l'entier n en paramètre et renvoie le groupe correspondant.

[5]:
```
def ZnPlus(n, **opt):
    elts = list(range(n))
```

```
    op = [[(i + j) % n for i in range(n)] for j in range(n)]
    return Monoide(elts, op, **opt)
```

Affichons la table de $(\mathbb{Z}_{12}, \oplus)$.

[6]:
```
G = ZnPlus(12, commut=True, assoc=True, neutre=0, inverses={})
afficher_table(G)
```

```
   |  0  1  2  3  4  5  6  7  8  9 10 11
  0|  0  1  2  3  4  5  6  7  8  9 10 11
  1|  1  2  3  4  5  6  7  8  9 10 11  0
  2|  2  3  4  5  6  7  8  9 10 11  0  1
  3|  3  4  5  6  7  8  9 10 11  0  1  2
  4|  4  5  6  7  8  9 10 11  0  1  2  3
  5|  5  6  7  8  9 10 11  0  1  2  3  4
  6|  6  7  8  9 10 11  0  1  2  3  4  5
  7|  7  8  9 10 11  0  1  2  3  4  5  6
  8|  8  9 10 11  0  1  2  3  4  5  6  7
  9|  9 10 11  0  1  2  3  4  5  6  7  8
 10| 10 11  0  1  2  3  4  5  6  7  8  9
 11| 11  0  1  2  3  4  5  6  7  8  9 10
```

1.2.2 2.2 Les entiers modulo n - mutliplication

Cette fois-ci, pour tout $n \in \mathbb{N}^*$, notre monoïde est $(\mathbb{Z}_n, \otimes)$, muni de l'opération

$$x \otimes y = (xy) \bmod n$$

C'est effectivement un monoïde mais ce n'est pas un groupe, sauf si $n = 1$.

[7]:
```
def ZnMult(n, **opt):
    elts = list(range(n))
    op = [[(i * j) % n for i in range(n)] for j in range(n)]
    return Monoide(elts, op, **opt)
```

[8]:
```
M = ZnMult(12, commut=True, assoc=True, neutre=1, inverses={})
afficher_table(M)
```

```
   | 0  1  2 3 4  5 6  7 8 9 10 11
  0| 0  0  0 0 0  0 0  0 0 0  0  0
  1| 0  1  2 3 4  5 6  7 8 9 10 11
  2| 0  2  4 6 8 10 0  2 4 6  8 10
  3| 0  3  6 9 0  3 6  9 0 3  6  9
  4| 0  4  8 0 4  8 0  4 8 0  4  8
  5| 0  5 10 3 8  1 6 11 4 9  2  7
  6| 0  6  0 6 0  6 0  6 0 6  0  6
  7| 0  7  2 9 4 11 6  1 8 3 10  5
```

```
 8| 0  8  4 0 8  4 0  8 4 0  8  4
 9| 0  9  6 3 0  9 6  3 0 9  6  3
10| 0 10  8 6 4  2 0 10 8 6  4  2
11| 0 11 10 9 8  7 6  5 4 3  2  1
```

1.2.3 2.3 Le groupe symétrique $\mathfrak{S}_3$

Voici le groupe $(\mathfrak{S}_3, \circ)$ des permutations de l'ensemble $\{1, 2, 3\}$, c'est à dire des bijections de $\{1, 2, 3\}$ sur lui-même. On représente la permutation σ par le triplet $(\sigma(1), \sigma(2), \sigma(3))$.

[9]:
```
def S3(**opt):
    e, p, p2, t1, t2, t3 = (1, 2, 3), (2, 3, 1), (3, 1, 2), (1, 3, 2), (3, 2,␣
 ↪1), (2, 1, 3)
    elts = [e, p, p2, t1, t2, t3]
    table = [
        [e, p, p2, t1, t2, t3],
        [p, p2, e, t3, t1, t2],
        [p2, e, p, t2, t3, t1],
        [t1, t2, t3, e, p, p2],
        [t2, t3, t1, p2, e, p],
        [t3, t1, t2, p, p2, e]]
    return Monoide(elts, table, **opt)
```

[10]:
```
G = S3(commut=False, assoc=True, neutre=(1, 2, 3), inverses={})
afficher_table(G)
```

```
          | (1, 2, 3) (2, 3, 1) (3, 1, 2) (1, 3, 2) (3, 2, 1) (2, 1, 3)
(1, 2, 3)| (1, 2, 3) (2, 3, 1) (3, 1, 2) (1, 3, 2) (3, 2, 1) (2, 1, 3)
(2, 3, 1)| (2, 3, 1) (3, 1, 2) (1, 2, 3) (2, 1, 3) (1, 3, 2) (3, 2, 1)
(3, 1, 2)| (3, 1, 2) (1, 2, 3) (2, 3, 1) (3, 2, 1) (2, 1, 3) (1, 3, 2)
(1, 3, 2)| (1, 3, 2) (3, 2, 1) (2, 1, 3) (1, 2, 3) (2, 3, 1) (3, 1, 2)
(3, 2, 1)| (3, 2, 1) (2, 1, 3) (1, 3, 2) (3, 1, 2) (1, 2, 3) (2, 3, 1)
(2, 1, 3)| (2, 1, 3) (1, 3, 2) (3, 2, 1) (2, 3, 1) (3, 1, 2) (1, 2, 3)
```

[11]:
```
print(G.op((2, 3, 1), (1, 3, 2)), G.op((1, 3, 2), (2, 3, 1)))
```

```
(2, 1, 3) (3, 2, 1)
```

Remarquons que nous avons dans tous nos exemples affecté au paramètre `inverses` une valeur erronée. Cela n'est pas grave … tant qu'on ne s'amuse pas à calculer des inverses ! Continuons dans cette voie.

1.2.4 2.4 Les groupes diédraux

Soit $n \geq 1$. L'ensemble $\mathcal{U}_n$ des racines $n$ièmes de l'unité forme un polygone régulier à n côtés (si $n = 1$ ou 2, ce polygone est un tantinet dégénéré). Rappelons que

$$\mathcal{U}_n = \{1, \omega, \dots, \omega^{n-1}\}$$

où $\omega = \exp(2i\pi/n)$. Appelons r la rotation d'angle $\frac{2\pi}{n}$ et s la symétrie par rapport à Ox. On a pour tout $z \in \mathbb{C}$,

$$r(z) = \omega z \text{ et } s(z) = \overline{z}$$

Définition. Le *groupe diédral d'ordre* n, $(\mathbb{D}_n, \circ)$, est le groupe des isométries du plan qui laissent invariant l'ensemble $\mathcal{U}_n$.

Proposition. $(D_n, \circ)$ est un groupe de cardinal $2n$. On a $\mathbb{D}_n = \{id, r, \dots, r^{n-1}, s, r\circ s, \dots, r^{n-1}\circ s\}$.

Démonstration. Laissée en exercice. □

Précisons la loi de groupe. Notons $\oplus$ et $\ominus$ l'addition et la soustraction des entiers modulo n. On a alors pour tous $i, j \in [\![0, n-1]\!]$, en remarquant que $r^n = id$,

- $r^i \circ r^j = r^{i+j} = r^{i\oplus j}$.
- $r^i \circ (r^j \circ s) = (r^i \circ r^j) \circ s = r^{i\oplus j} \circ s$.

Il nous reste à déterminer $(r^i \circ s) \circ r^j$ et $(r^i \circ s) \circ (r^j \circ s)$. Remarquons que pour tout $z \in \mathbb{C}$, $r^j(j) = \omega^j z$. De là,

$$s \circ r^j(z) = s(\omega^j z) = \omega^{-j}\overline{z} = r^{-j} \circ s(z)$$

Ainsi, $s \circ r^j = r^{-j} \circ s$. On en déduit que

$$(r^i \circ s) \circ r^j = r^i \circ r^{-j} \circ s = r^{i-j} \circ s = r^{i\ominus j} \circ s$$

et

$$(r^i \circ s) \circ (r^j \circ s) = r^i \circ r^{-j} \circ s \circ s = r^{i\ominus j}$$

Pour $i \in [\![0, n-1]\!]$ et $\alpha \in [\![0, 1]\!]$, nous représentons en Python $r^i \circ s^\alpha$ par le couple (i, α).

L'opération $\circ$ devient ainsi une opération sur $[\![0, n-1]\!] \times [\![0, 1]\!]$ que nous noterons encore $\circ$. En traduisant les résultats que nous avons obtenus, il vient

- $(i, 0) \circ (j, 0) = (i \oplus j, 0)$.
- $(i, 0) \circ (j, 1) = (i \oplus j, 1)$.
- $(i, 1) \circ (j, 0) = (i \ominus j, 1)$.
- $(i, 1) \circ (j, 1) = (i \ominus j, 0)$.

Ô miracle, le tout se résume en une seule formule : $(i, \alpha) \circ (j, \beta) = (i \oplus (-1)^\alpha j, \alpha \oplus \beta)$, où le dernier $\oplus$ est une addition modulo 2 (et pas n). Nous pouvons maintenant définir les groupes diédraux en Python.

[12]:
```
def D(n, **opt):
    elts = [(i, a) for a in range(2) for i in range(n)]
    table = []
```

```
    for a in range(2):
        eps = (-1) ** a
        for i in range(n):
            s = [((i + eps * j) % n, (a + b) % 2) for b in range(2) for j in␣
↪range(n)]
            table.append(s)
    return Monoide(elts, table, **opt)
```

[13]:
```
G = D(5, commut=False, assoc=True, neutre=(0,0), inverses={})
afficher_table(G)
```

```
      | (0, 0) (1, 0) (2, 0) (3, 0) (4, 0) (0, 1) (1, 1) (2, 1) (3, 1) (4, 1)
(0, 0)| (0, 0) (1, 0) (2, 0) (3, 0) (4, 0) (0, 1) (1, 1) (2, 1) (3, 1) (4, 1)
(1, 0)| (1, 0) (2, 0) (3, 0) (4, 0) (0, 0) (1, 1) (2, 1) (3, 1) (4, 1) (0, 1)
(2, 0)| (2, 0) (3, 0) (4, 0) (0, 0) (1, 0) (2, 1) (3, 1) (4, 1) (0, 1) (1, 1)
(3, 0)| (3, 0) (4, 0) (0, 0) (1, 0) (2, 0) (3, 1) (4, 1) (0, 1) (1, 1) (2, 1)
(4, 0)| (4, 0) (0, 0) (1, 0) (2, 0) (3, 0) (4, 1) (0, 1) (1, 1) (2, 1) (3, 1)
(0, 1)| (0, 1) (4, 1) (3, 1) (2, 1) (1, 1) (0, 0) (4, 0) (3, 0) (2, 0) (1, 0)
(1, 1)| (1, 1) (0, 1) (4, 1) (3, 1) (2, 1) (1, 0) (0, 0) (4, 0) (3, 0) (2, 0)
(2, 1)| (2, 1) (1, 1) (0, 1) (4, 1) (3, 1) (2, 0) (1, 0) (0, 0) (4, 0) (3, 0)
(3, 1)| (3, 1) (2, 1) (1, 1) (0, 1) (4, 1) (3, 0) (2, 0) (1, 0) (0, 0) (4, 0)
(4, 1)| (4, 1) (3, 1) (2, 1) (1, 1) (0, 1) (4, 0) (3, 0) (2, 0) (1, 0) (0, 0)
```

[14]:
```
G.op((3, 1), (2, 0))
```

[14]:
```
(1, 1)
```

1.2.5 2.5 Le groupe des quaternions

Notons $\mathbb{H}_8 = \{1, -1, i, -i, j, -j, k, -k\}$. On munit $\mathbb{H}_8$ d'une opération $\times$ en posant

$$i^2 = j^2 = k^2 = -1$$

$$ij = k,\ jk = i,\ ki = j$$

$$ji = -k,\ kj = -i,\ ik = -j$$

Les autres produits s'obtiennent par la règle des signes. $(\mathbb{H}_8, \times)$ est un groupe non abélien (Python saura bientôt nous le démontrer).

[15]:
```
def H8(**opt):
    elts = ['1', '-1', 'i', '-i', 'j', '-j', 'k', '-k']
    table =[
        ['1', '-1', 'i', '-i', 'j', '-j', 'k', '-k'],
```

```
        ['-1', '1', '-i', 'i', '-j', 'j', '-k', 'k'],
        ['i', '-i', '-1', '1', 'k', '-k', '-j', 'j'],
        ['-i', 'i', '1', '-1', '-k', 'k', 'j', '-j'],
        ['j', '-j', '-k', 'k', '-1', '1', 'i', '-i'],
        ['-j', 'j', 'k', '-k', '1', '-1', '-i', 'i'],
        ['k', '-k', 'j', '-j', '-i', 'i', '-1', '1'],
        ['-k', 'k', '-j', 'j', 'i', '-i', '1', '-1']]
    return Monoide(elts, table, **opt)
```

[16]:
```
G = H8(commut=False, assoc=True, neutre='1', inverses={})
afficher_table(G)
```

```
   |  1 -1  i -i  j -j  k -k
  1|  1 -1  i -i  j -j  k -k
 -1| -1  1 -i  i -j  j -k  k
  i|  i -i -1  1  k -k -j  j
 -i| -i  i  1 -1 -k  k  j -j
  j|  j -j -k  k -1  1  i -i
 -j| -j  j  k -k  1 -1 -i  i
  k|  k -k  j -j -i  i -1  1
 -k| -k  k -j  j  i -i  1 -1
```

[17]:
```
print(G.op('i', 'j'), G.op('j', 'i'))
```

```
k -k
```

1.3 3. Propriétés des monoïdes

1.3.1 3.1 Commutativité

La fonction ci-dessous renvoie `True` si les éléments x et y du monoïde $(M, \star)$ commutent.

[18]:
```
def commutent(x, y, M):
    return M.op(x, y) == M.op(y, x)
```

La fonction `tester_commut` renvoie `True` si le monoïde M est commutatif et `False` sinon. Si M possède n éléments, cette fonction nécessite $O(n^2)$ opérations.

[19]:
```
def tester_commut(M):
    for x in M.elts:
        for y in M.elts:
            if not commutent(x, y, M):
                return False
    return True
```

Dorénavant, lorsque nous créons un monoïde M, nous pouvons laisser le constructeur `Monoide` faire le travail et tester la commutativité pour nous. Le champ `M.commut` est mis à `True` ou à `False`,

selon le résultat du test.

[20]:
```
G = ZnPlus(47, assoc=True, neutre=0, inverses={})
print(G.commut)
```

```
True
```

[21]:
```
G = ZnMult(47, assoc=True, neutre=0, inverses={})
print(G.commut)
```

```
True
```

[22]:
```
G = S3(assoc=True, neutre=(1, 2, 3), inverses={})
print(G.commut)
```

```
False
```

[23]:
```
G = D(10, assoc=True, neutre=(0, 0), inverses={})
print(G.commut)
```

```
False
```

[24]:
```
G = H8(assoc=True, neutre='1', inverses={})
print(G.commut)
```

```
False
```

1.3.2 3.2 Associativité

La fonction `associent` renvoie `True` si les éléments x, y et z du monoïde $(M, \star)$ vérifient $(x \star y) \star z = x \star (y \star z)$.

[25]:
```
def associent(x, y, z, M):
    return M.op(M.op(x, y), z) == M.op(x, M.op(y, z))
```

La fonction `est_associatif` ci-dessous est coûteuse en temps de calcul. Elle effectue $O(n^3)$ opérations où n est le cardinal du monoïde.

[26]:
```
def tester_assoc(M):
    for x in M.elts:
        for y in M.elts:
            for z in M.elts:
                if not associent(x, y, z, M):
                    return False
    return True
```

Un monoïde est maintenant capable de tester sa propre associativité.

L'exemple ci-dessous effectue un million de comparaisons. N'essayons pas plus gros.

[27]:
```
M = ZnMult(101, neutre=1, inverses={})
print(M.assoc)
```

```
True
```

Comme chacun le sait, la composition des applications est associative. $\mathfrak{S}_3$ et $\mathbb{D}_n$ devraient donc passer le test d'associativité haut la main.

[28]:
```
G = S3(neutre=(1, 2, 3), inverses={})
print(G.assoc)
```

```
True
```

[29]:
```
G = D(23, neutre=(0, 0), inverses={})
print(G.assoc)
```

```
True
```

La multiplication des quaternions est associative. En effet :

[30]:
```
G = H8(neutre='1', inverses={})
print(G.assoc)
```

```
True
```

1.3.3 3.3 Élément neutre

Rappelons que si le neutre existe, il est nécessairement unique. La fonction `est_neutre` prend en paramètres un monoïde $(M, \star)$ et un élément e de M. Elle renvoie `True` si e est le neutre de M et `False` sinon.

[31]:
```
def est_neutre(e, M):
    for x in M.elts:
        if M.op(e, x) != x or M.op(x, e) != x:
            return False
    return True
```

La fonction `trouver_neutre` prend en paramètre un monoïde M. Elle recherche parmi tous les éléments de M s'il y a un élément neutre. En cas de réponse positive, le champ le champ `M.neutre` contient le neutre de M. Sinon, le champ `M.neutre` vaut `None`.

[32]:
```
def trouver_neutre(M):
    for e in M.elts:
        if est_neutre(e, M):
            return e
    return None
```

Nous en avons fini avec la fastidieuse opération de déterminer les neutres. Nous monoïdes les trouvent tout seuls …

[33]:
```
G = S3(inverses={})
print(G.neutre)
```

```
(1, 2, 3)
```

[34]:
```
G = ZnMult(53, inverses={})
print(G.neutre)
```

```
1
```

[35]:
```
G = ZnPlus(53, inverses={})
print(G.neutre)
```

```
0
```

[36]:
```
G = D(25, inverses={})
print(G.neutre)
```

```
(0, 0)
```

[37]:
```
G = H8(inverses={})
print(G.neutre)
```

```
1
```

1.3.4 3.4 Inverses

Soit $(M, \star)$ un monoïde de neutre e. Un élément x de M est inversible lorsqu'il existe $y \in M$ tel que $x \star y = y \star x = e$. Si un tel y existe, l'associativité de l'opération $\star$ assure l'unicité de y, qui est appelé *l'inverse* de x.

La fonction `sont_inverses` vérifie si x et y sont inverses l'un de l'autre dans le monoïde M. Appeler cette fonction n'a de sens que si l'on a déjà trouvé le neutre.

[38]:
```
def sont_inverses(x, y, M):
    return M.op(x,y) == M.neutre and M.op(y, x) == M.neutre
```

La fonction `inverse` renvoie l'inverse de x dans le monoïde M de neutre e, si cet inverse existe. Sinon elle renvoie `None`.

[39]:
```
def inverse(x, M):
    for y in M.elts:
        if sont_inverses(x, y, M): return y
    return None
```

La fonction `trouver_inverses` prend en paramètre un monoïde M. Elle recherche les inverses de tous les éléments de M. Le champ `M.inverse` d'un monoïde M contient un dictionnaire des inverses des éléments de M. Si `M.inverse[x]` vaut `None` cela signifie que x n'est pas inversible. Sinon, `M.inverse[x]` est l'inverse de x.

[40]:
```
def trouver_inverses(M):
    inv = {}
    for x in M.elts:
        inv[x] = inverse(x, M)
    return inv
```

Voici la liste des éléments de $(\mathbb{Z}_{15}, \otimes)$ avec leur éventuel inverse.

[41]:
```
M = ZnMult(15)
for x in M.elts:
    print(x, M.inverse[x])
```

```
0 None
1 1
2 8
3 None
4 4
5 None
6 None
7 13
8 2
9 None
10 None
11 11
12 None
13 7
14 14
```

Voici les inverses des éléments de $\mathfrak{S}_3$.

[42]:
```
G = S3()
for x in G.elts:
    print(x, G.inverse[x])
```

```
(1, 2, 3) (1, 2, 3)
(2, 3, 1) (3, 1, 2)
(3, 1, 2) (2, 3, 1)
(1, 3, 2) (1, 3, 2)
(3, 2, 1) (3, 2, 1)
(2, 1, 3) (2, 1, 3)
```

Voici les inverses des éléments de $\mathbb{D}_5$.

[43]:
```
G = D(5)
for x in G.elts:
    print(x, G.inverse[x])
```

```
(0, 0) (0, 0)
(1, 0) (4, 0)
```

```
(2, 0) (3, 0)
(3, 0) (2, 0)
(4, 0) (1, 0)
(0, 1) (0, 1)
(1, 1) (1, 1)
(2, 1) (2, 1)
(3, 1) (3, 1)
(4, 1) (4, 1)
```

Voici les inverses des éléments de $\mathbb{H}_8$.

[44]:
```
G = H8()
for x in G.elts:
    print(x, G.inverse[x])
```

```
1 1
-1 -1
i -i
-i i
j -j
-j j
k -k
-k k
```

1.3.5 3.5 Bilan

Nous y voilà. Nous n'avons plus besoin de *calculer* quoi que ce soit avant de définir un monoïde. Il se charge de vérifier tout seul quelles sont les propriétés que le monoïde vérifie. Les fonctions ci-dessous se passent de commentaires.

[45]:
```
def est_monoide(M): return M.assoc and M.neutre != None
def est_monoide_commutatif(M): return est_monoide(M) and M.commut
```

[46]:
```
def est_groupe(M):
    return est_monoide(M) and None not in [M.inverse[x] for x in M.elts]

def est_groupe_abelien(M):
    return M.commut and est_groupe(M)
```

[47]:
```
G = ZnPlus(18)
print(est_groupe_abelien(G))
```

```
True
```

[48]:
```
G = ZnMult(18)
print(est_groupe(G))
```

```
False
```

[49]:
```
print(est_groupe(S3()))
print(est_groupe_abelien(S3()))
```

```
True
False
```

[50]:
```
G = D(25)
print(est_groupe(G))
print(est_groupe_abelien(G))
```

```
True
False
```

[51]:
```
G = H8()
print(est_groupe(G))
print(est_groupe_abelien(G))
```

```
True
False
```

1.4 4. Quelques compléments

1.4.1 4.1 Groupe des éléments inversibles d'un monoïde

Proposition. Soit $(M, \star)$ un monoïde. L'ensemble M^* des éléments inversibles de M est un groupe.

Démonstration. En exercice.

La fonction `groupe_inversibles` prend en paramètre un monoïde M et renvoie le groupe de ses éléments inversibles.

[52]:
```
def groupe_inversibles(M):
    elts = [x for x in M.elts if M.inverse[x] != None]
    table = []
    for x in elts:
        table.append([M.op(x, y) for y in elts])
    G = Monoide(elts, table, assoc=True, neutre=M.neutre)
    return G
```

Voici par exemple le groupe des inversibles de $\mathbb{Z}_{48}$.

[53]:
```
M = ZnMult(48, assoc=True, commut=True, neutre=1)
G = groupe_inversibles(M)
print(est_groupe_abelien(G))
```

```
True
```

[54]:
```
print(G.elts)
```

```
[1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47]
```

[55]:
```
afficher_table(G)
```

```
   |  1  5  7 11 13 17 19 23 25 29 31 35 37 41 43 47
 1|  1  5  7 11 13 17 19 23 25 29 31 35 37 41 43 47
 5|  5 25 35  7 17 37 47 19 29  1 11 31 41 13 23 43
 7|  7 35  1 29 43 23 37 17 31 11 25  5 19 47 13 41
11| 11  7 29 25 47 43 17 13 35 31  5  1 23 19 41 37
13| 13 17 43 47 25 29  7 11 37 41 19 23  1  5 31 35
17| 17 37 23 43 29  1 35  7 41 13 47 19  5 25 11 31
19| 19 47 37 17  7 35 25  5 43 23 13 41 31 11  1 29
23| 23 19 17 13 11  7  5  1 47 43 41 37 35 31 29 25
25| 25 29 31 35 37 41 43 47  1  5  7 11 13 17 19 23
29| 29  1 11 31 41 13 23 43  5 25 35  7 17 37 47 19
31| 31 11 25  5 19 47 13 41  7 35  1 29 43 23 37 17
35| 35 31  5  1 23 19 41 37 11  7 29 25 47 43 17 13
37| 37 41 19 23  1  5 31 35 13 17 43 47 25 29  7 11
41| 41 13 47 19  5 25 11 31 17 37 23 43 29  1 35  7
43| 43 23 13 41 31 11  1 29 19 47 37 17  7 35 25  5
47| 47 43 41 37 35 31 29 25 23 19 17 13 11  7  5  1
```

1.4.2 3.2 Ordre d'un élément dans un groupe

Définition. Soit $(G, \star)$ un groupe de neutre e. Soit $x \in G$. L'ordre de x est

$$\omega(x) = \min\{p \in \mathbb{N}^*, x^p = e\}$$

Lorsque le groupe G est fini, $\omega(x)$ existe. C'est le cardinal du plus petit sous-groupe de G contenant x.

[56]:
```
def ordre(x, G):
    p = x
    c = 1
    while p != G.neutre:
        p = G.op(p, x)
        c = c + 1
    return c
```

[57]:
```
G = S3()
print([(x,ordre(x, G)) for x in G.elts])
```

```
[((1, 2, 3), 1), ((2, 3, 1), 3), ((3, 1, 2), 3), ((1, 3, 2), 2), ((3, 2, 1), 2),
((2, 1, 3), 2)]
```

[58]:
```
G = D(6)
for x in G.elts:
```

```
    print(x,ordre(x, G))
```

```
(0, 0) 1
(1, 0) 6
(2, 0) 3
(3, 0) 2
(4, 0) 3
(5, 0) 6
(0, 1) 2
(1, 1) 2
(2, 1) 2
(3, 1) 2
(4, 1) 2
(5, 1) 2
```

[59]:
```
G = groupe_inversibles(ZnMult(100, commut=True, assoc=True, neutre=1))
print([(x,ordre(x, G)) for x in G.elts])
```

```
[(1, 1), (3, 20), (7, 4), (9, 10), (11, 10), (13, 20), (17, 20), (19, 10), (21,
5), (23, 20), (27, 20), (29, 10), (31, 10), (33, 20), (37, 20), (39, 10), (41,
5), (43, 4), (47, 20), (49, 2), (51, 2), (53, 20), (57, 4), (59, 10), (61, 5),
(63, 20), (67, 20), (69, 10), (71, 10), (73, 20), (77, 20), (79, 10), (81, 5),
(83, 20), (87, 20), (89, 10), (91, 10), (93, 4), (97, 20), (99, 2)]
```

[60]:
```
G = H8()
for x in G.elts:
    print('%5s%5d' % (x, ordre(x, G)))
```

```
    1    1
   -1    2
    i    4
   -i    4
    j    4
   -j    4
    k    4
   -k    4
```

1.4.3 3.3 Groupes cycliques

Regardons les ordres des éléments du groupe des inversibles de $(\mathbb{Z}_{46}, \otimes)$.

[61]:
```
G = groupe_inversibles(ZnMult(46, commut=True, assoc=True, neutre=1))
print(len(G.elts))
print([(x,ordre(x, G)) for x in G.elts])
```

```
22
[(1, 1), (3, 11), (5, 22), (7, 22), (9, 11), (11, 22), (13, 11), (15, 22), (17,
```

```
22), (19, 22), (21, 22), (25, 11), (27, 11), (29, 11), (31, 11), (33, 22), (35,
11), (37, 22), (39, 11), (41, 11), (43, 22), (45, 2)]
```

On remarque que parmi les inversibles de $\mathbb{Z}_{46}$ il y a un élément d'ordre 22, par exemple le nombre 11. Or le groupe des inversibles de $\mathbb{Z}_{46}$ est justement de cardinal 22. Ainsi, $\mathbb{Z}_{46}^*$ est *engendré* par 11 : c'est l'ensemble des puissances de 11. On a affaire à ce que l'on appelle un *groupe cyclique*.

Définition. Soit G un groupe fini. Le groupe G est cyclique s'il existe $a \in G$ tel que $G = \{a^n, n \in \mathbb{Z}\}$.

Remarquons qu'un groupe cyclique est nécessairement commutatif.

La fonction `est_cyclique` ci-dessous prend un groupe G en paramètre et renvoie `(True, a)` si G est cyclique engendré par a et `(False, None)`sinon.

[62]:
```
def est_cyclique(G):
    n = len(G.elts)
    for x in G.elts:
        if ordre(x, G) == n: return (True, x)
    return (False, None)
```

[63]:
```
est_cyclique(groupe_inversibles(ZnMult(8)))
```

[63]:
```
(False, None)
```

[64]:
```
est_cyclique(groupe_inversibles(ZnMult(46, commut=True, assoc=True, neutre=1)))
```

[64]:
```
(True, 5)
```

Pour les entiers n entre 2 et 300, quels sont les groupe des éléments inversibles de $(\mathbb{Z}_n, \otimes)$ qui sont cycliques ?

[65]:
```
for n in range(2, 301):
    M = ZnMult(n, commut=True, assoc=True, neutre=1)
    G = groupe_inversibles(M)
    if est_cyclique(G)[0]:
        print(n, end=' ')
```

```
2 3 4 5 6 7 9 10 11 13 14 17 18 19 22 23 25 26 27 29 31 34 37 38 41 43 46 47 49
50 53 54 58 59 61 62 67 71 73 74 79 81 82 83 86 89 94 97 98 101 103 106 107 109
113 118 121 122 125 127 131 134 137 139 142 146 149 151 157 158 162 163 166 167
169 173 178 179 181 191 193 194 197 199 202 206 211 214 218 223 226 227 229 233
239 241 242 243 250 251 254 257 262 263 269 271 274 277 278 281 283 289 293 298
```

Pas très parlant, sauf si on connaît la réponse. Mais quelle est-elle ? Eh bien, c'est pour $n = 2, 4$, p^α et $2p^\alpha$ où p est un nombre premier impair et $\alpha \in \mathbb{N}^*$. Et ce n'est pas trivial …

[66]:
```
def est_premier(p):
    if p <= 1: return False
    else:
        k = 2
```

```
        while k * k <= p and p % k != 0:
            k = k +1
        return k * k > p
```

[67]:
```
print([p for p in range(2, 50) if est_premier(p)])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

[68]:
```
ps = [p for p in range(3, 301) if est_premier(p)]
alphas = list(range(1, 6))
s = []
for p in ps:
    for alpha in alphas:
        n = p ** alpha
        if n > 300: break
        if 2 * n <= 300: s.append(2 * n)
        if n <= 300: s.append(n)
s.sort()
print(s)
```

```
[3, 5, 6, 7, 9, 10, 11, 13, 14, 17, 18, 19, 22, 23, 25, 26, 27, 29, 31, 34, 37,
38, 41, 43, 46, 47, 49, 50, 53, 54, 58, 59, 61, 62, 67, 71, 73, 74, 79, 81, 82,
83, 86, 89, 94, 97, 98, 101, 103, 106, 107, 109, 113, 118, 121, 122, 125, 127,
131, 134, 137, 139, 142, 146, 149, 151, 157, 158, 162, 163, 166, 167, 169, 173,
178, 179, 181, 191, 193, 194, 197, 199, 202, 206, 211, 214, 218, 223, 226, 227,
229, 233, 239, 241, 242, 243, 250, 251, 254, 257, 262, 263, 269, 271, 274, 277,
278, 281, 283, 289, 293, 298]
```

1.4.4 3.4 Produit cartésien de groupes

Soient $(M, \star)$ et $(N, \circ)$ deux monoïdes. Le *produit cartésien* de M et N est $M \times N$. On le munit de l'opération

$$(x, y) \otimes (x', y') = (x \star x', y \circ y')$$

On vérifie facilement que $(M \times N, \otimes)$ est un monoïde. Si M et N sont des groupes, alors $M \times N$ aussi.

La fonction `produit` prend en paramètres deux monoïdes M et N et renvoie le monoïde $M \times N$.

[69]:
```
def produit(M, N, **opt):
    elts = [(x, y) for y in N.elts for x in M.elts]
    table = [[(M.op(x1, x2), N.op(y1, y2)) for y2 in N.elts for x2 in M.elts]␣
 ↪for y1 in N.elts for x1 in M.elts]
    return Monoide(elts, table, **opt)
```

[70]:
```
G = produit(ZnPlus(3), ZnPlus(2))
afficher_table(G)
print(G.commut, G.assoc, G.neutre)
print(G.inverse)
```

```
       | (0, 0) (1, 0) (2, 0) (0, 1) (1, 1) (2, 1)
 (0, 0)| (0, 0) (1, 0) (2, 0) (0, 1) (1, 1) (2, 1)
 (1, 0)| (1, 0) (2, 0) (0, 0) (1, 1) (2, 1) (0, 1)
 (2, 0)| (2, 0) (0, 0) (1, 0) (2, 1) (0, 1) (1, 1)
 (0, 1)| (0, 1) (1, 1) (2, 1) (0, 0) (1, 0) (2, 0)
 (1, 1)| (1, 1) (2, 1) (0, 1) (1, 0) (2, 0) (0, 0)
 (2, 1)| (2, 1) (0, 1) (1, 1) (2, 0) (0, 0) (1, 0)
True True (0, 0)
{(0, 0): (0, 0), (1, 0): (2, 0), (2, 0): (1, 0), (0, 1): (0, 1), (1, 1): (2, 1),
(2, 1): (1, 1)}
```

1.5 4. Sous-groupes

1.5.1 4.1 Sous-monoïde, sous-groupe

Soit $(M, \star)$ un monoïde de neutre e. Soit $H \subset M$. H est un *sous-monoïde* de M si et seulement si

- $e \in H$.
- H est stable pour l'opération $\star$.

Si $(M, \star)$ est un groupe, H est un *sous-groupe* de M si et seulement si

- H est un sous-monoïde de M.
- L'inverse de tout élément de H pour l'opération $\star$ appartient à H.

[71]:
```
def est_inclus(H, M):
    for x in H:
        if not x in M.elts: return False
    return True
```

La fonction `est_stable` prend en paramètres un ensemble H et un monoïde M. L'ensemble H est censé être inclus dans M. La fonction renvoie `True` si H est stable pour l'opération de M et `False` sinon.

[72]:
```
def est_stable(H, M):
    for x in H:
        for y in H:
            if M.op(x, y) not in H: return False
    return True
```

La fonction `est_sous_monoïde` renvoie `True` si H est un sous-monoïde de M et `False` sinon.

[73]:
```
def est_sous_monoide(H, M):
    return est_inclus(H, M) and M.neutre in H and est_stable(H, M)
```

Il est alors facile de savoir si un sous-ensemble H du groupe G est un sous-groupe de G. La fonction `est_sous_groupe` fait le travail.

[74]:
```
def est_sous_groupe(H, G):
    if not est_sous_monoide(H, G): return False
    for x in H:
        y = G.inverse[x]
        if y not in H: return False
    return True
```

1.5.2 4.2 Énumérer les parties d'un ensemble

Soit $n \in \mathbb{N}$. Tout entier $k \in [\![0, 2^n - 1]\!]$ s'écrit de façon unique

$$k = \sum_{i=0}^{n-1} a_i 2^i$$

où les a_i appartiennent à $\{0, 1\}$. L'application $\varphi : k \longmapsto (a_0, \ldots, a_{n-1})$ est une bijection de $[\![0, 2^n-1]\!]$ sur $\{0, 1\}^n$.

Soit $E = \{x_0, \ldots, x_{n-1}\}$ un ensemble de cardinal n. L'application $\chi : \{0, 1\}^n \longrightarrow \mathcal{P}(E)$ définie par $\chi(a_0, \ldots, a_{n-1}) = \{x_k, a_k = 1\}$ est elle aussi une bijection. En composant φ et χ, on obtient donc une bijection de $[\![0, 2^n - 1]\!]$ sur $\mathcal{P}(E)$.

La fonction `int_to_bin` prend en paramètres deux entiers k et n, où $k \in [\![0, 2^n - 1]\!]$. Elle renvoie la liste $[a_0, \ldots, a_{n-1}]$.

[75]:
```
def int_to_bin(k, n):
    s = []
    for i in range(n):
        s.append(k % 2)
        k = k // 2
    s.reverse()
    return s
```

[76]:
```
for k in range(2 ** 3):
    print(int_to_bin(k, 3))
```

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

La fonction `indic` prend en paramètres un ensemble E (décrit par la liste de ses éléments) de cardinal n et un n-uplet c d'entiers de $\{0,1\}$. Elle renvoie la liste des éléments de E dont l'indice est tel que $c_k = 1$.

[77]:
```
def indic(E, c):
    F = []
    for k in range(len(E)):
        if c[k] == 1:
            F.append(E[k])
    return F
```

[78]:
```
print(indic(['a', 'b', 'c', 'd', 'e', 'f'], [0, 1, 1, 0, 0, 1]))
```

```
['b', 'c', 'f']
```

Il est maintenant facile d'énumérer les parties d'un ensemble (fini) E.

[79]:
```
def parties(E):
    n = len(E)
    for k in range(2 ** n):
        yield indic(E, int_to_bin(k, n))
```

[80]:
```
for p in parties([1, 2, 3, 4]): print(p, end=' ')
```

```
[] [4] [3] [3, 4] [2] [2, 4] [2, 3] [2, 3, 4] [1] [1, 4] [1, 3] [1, 3, 4] [1, 2]
[1, 2, 4] [1, 2, 3] [1, 2, 3, 4]
```

1.5.3 4.3 Énumérer les sous-groupes d'un groupe

Pour énumérer les sous-groupes d'un groupe G :

- On énumère les parties de G.
- On garde celles qui sont des sous-groupes de G.

La fonction `sous-groupes` ci-dessous fait le travail. Pour gagner (un tout petit peu) en efficacité, elle utilise le fait que le neutre de G est dans tous les sous-groupes. Elle utilise aussi le *théorème de Lagrange*.

Proposition [Lagrange]. Soit G un groupe fini. Soit H un sous-groupe de G. Alors le cardinal de H divise le cardinal de G.

[81]:
```
def enlever(s, x):
    s1 = s[:]
    s1.remove(x)
    return s1
```

[82]:
```
def sous_groupes(G):
    n = len(G.elts)
    G1 = enlever(G.elts, G.neutre)
```

```
    for H1 in parties(G1):
        H = H1 + [G.neutre]
        if n % len(H) == 0 and est_sous_groupe(H, G):
            H.sort()
            yield(H)
```

1.5.4 4.4 Exemples

Prenons quelques exemples, mais soyons raisonnables. Un groupe de cardinal n possède 2^n parties, alors ne testons que sur des groupes ayant au plus une vingtaine d'éléments.

Quels sont les sous-groupes de $\mathfrak{S}_3$?

[83]:
```
for H in sous_groupes(S3()):
    print(H)
```

```
[(1, 2, 3)]
[(1, 2, 3), (2, 1, 3)]
[(1, 2, 3), (3, 2, 1)]
[(1, 2, 3), (1, 3, 2)]
[(1, 2, 3), (2, 3, 1), (3, 1, 2)]
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

$\mathfrak{S}_3$ possède donc 6 sous-groupes.

Quels sont les sous-groupes de $(\mathbb{Z}_{20}, \oplus)$?

[84]:
```
for H in sous_groupes(ZnPlus(20)):
    print(H)
```

```
[0]
[0, 10]
[0, 5, 10, 15]
[0, 4, 8, 12, 16]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Considérons le groupe G des inversibles de $\mathbb{Z}_{25}$ pour la multiplication. Quels sont ses sous-groupes ?

[85]:
```
G = groupe_inversibles(ZnMult(25))
for H in sous_groupes(G):
    print(H)
```

```
[1]
[1, 24]
[1, 7, 18, 24]
[1, 6, 11, 16, 21]
```

```
[1, 4, 6, 9, 11, 14, 16, 19, 21, 24]
[1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24]
```

Quels sont les sous-groupes du groupe diédral $\mathbb{D}_6$?

[86]:
```
G = D(6)
for H in sous_groupes(G):
    print(H)
```

```
[(0, 0)]
[(0, 0), (5, 1)]
[(0, 0), (4, 1)]
[(0, 0), (3, 1)]
[(0, 0), (2, 1)]
[(0, 0), (1, 1)]
[(0, 0), (0, 1)]
[(0, 0), (3, 0)]
[(0, 0), (2, 1), (3, 0), (5, 1)]
[(0, 0), (1, 1), (3, 0), (4, 1)]
[(0, 0), (0, 1), (3, 0), (3, 1)]
[(0, 0), (2, 0), (4, 0)]
[(0, 0), (1, 1), (2, 0), (3, 1), (4, 0), (5, 1)]
[(0, 0), (0, 1), (2, 0), (2, 1), (4, 0), (4, 1)]
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1), (4, 0), (4, 1),
(5, 0), (5, 1)]
```

Voici les sous-groupes de $\mathbb{H}_8$.

[87]:
```
G = H8()
for H in sous_groupes(G):
    print(H)
```

```
['1']
['-1', '1']
['-1', '-k', '1', 'k']
['-1', '-j', '1', 'j']
['-1', '-i', '1', 'i']
['-1', '-i', '-j', '-k', '1', 'i', 'j', 'k']
```

Terminons par les sous-groupes de $\mathbb{Z}_3 \times \mathbb{Z}_4$.

[88]:
```
G = produit(ZnPlus(3), ZnPlus(4))
for H in sous_groupes(G):
    print(H)
```

```
[(0, 0)]
[(0, 0), (0, 2)]
[(0, 0), (0, 1), (0, 2), (0, 3)]
[(0, 0), (1, 0), (2, 0)]
```

```
[(0, 0), (0, 2), (1, 0), (1, 2), (2, 0), (2, 2)]
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1),
(2, 2), (2, 3)]
```

1.6 5. Isomorphismes

1.6.1 5.1 Introduction

Soient $(G, \star)$ et $(H, \otimes)$ deux groupes. La fonction $f : G \longrightarrow H$ est un *morphisme de groupes* si pour tous $x, y \in G$, $f(x \star y) = f(x) \otimes f(y)$. C'est un *isomorphisme* si, de plus, f est bijective.

Les groupes G et H sont *isomorphes* s'il existe un isomorphisme de G sur H. On note alors $G \simeq H$.

Tester si une fonction est un morphisme de groupes demande naïvement $O(mn)$ opérations, où m et n sont les cardinaux de G et H. Nous allons dans ce qui suit être naïfs. Nous représenterons en Python une fonction $f : E \longrightarrow F$ par un dictionnaire (appelons-le encore f). Si $x \in E$, $f[x]$ est l'image de x par f.

1.6.2 5.2 bijections entre deux ensembles

Soient E et F deux ensembles finis. Si $|E| \neq |F|$, il n'y a évidemment aucune bijection de E sur F. Sinon, en posant $n = |E| = |F|$, il y a $n!$ bijections de E vers F. Ceci suggère une petite remarque sur les temps de calcul … Si $n = 9$, la fonction `bijections` ci-dessous met environ 3 secondes pour calculer toutes les bijections de E sur F. Si $n = 10$, le temps de calcul passe à 30 secondes. Bref, évitons de dépasser 10. Oui, je sais, 10 est un très petit nombre, mais vouloir faire mieux demanderait d'adopter une toute autre approche …

La fonction `bijections` prend en paramètres deux ensembles E et F. Elle énumère les bijections de E sur F. Cette fonction utilise la fonciton `permutations` du module `permut`, que je ne décrirai pas ici.

[89]:
```
for f in bijections([1, 2, 3], ['a', 'b', 'c']):
    print(f)
```

```
{1: 'a', 2: 'b', 3: 'c'}
{1: 'a', 2: 'c', 3: 'b'}
{1: 'b', 2: 'a', 3: 'c'}
{1: 'b', 2: 'c', 3: 'a'}
{1: 'c', 2: 'a', 3: 'b'}
{1: 'c', 2: 'b', 3: 'a'}
```

1.6.3 5.3 Tester la propriété de morphisme

La fonction `est_morphisme` prend en paramètres une fonction $f : G \longrightarrow H$ et deux groupes G et H. Elle renvoie `True` si f est un morphisme de groupes et `False` sinon.

[90]:
```
def est_morphisme(f, G, H):
    for x in G.elts:
        for y in G.elts:
            if f[G.op(x, y)] != H.op(f[x], f[y]): return False
    return True
```

1.6.4 5.3 Trouver tous les isomorphismes

La fonction `isomorphismes` renvoie la liste des isomorphismes du groupe G sur le groupe H.

[91]:
```
def isomorphismes(G, H):
    G1 = enlever(G.elts, G.neutre)
    H1 = enlever(H.elts, H.neutre)
    for f in bijections(G1, H1):
        f[G.neutre] = H.neutre
        if est_morphisme(f, G, H):
            yield(f)
```

La fonction `sont_isomorphes` prend en paramètres deux groupes G et H.

- Si G et H sont isomorphes, elle renvoie le couple `(True, f)` où f est un isomorphisme de G sur H.
- Sinon, la fonction renvoie `(False, None)`.

[92]:
```
def sont_isomorphes(G, H):
    for f in isomorphismes(G, H):
        return (True, f)
    return (False, None)
```

Il existe 4 automorphismes de $(\mathbb{Z}_{10}, \oplus)$.

[93]:
```
for f in isomorphismes(ZnPlus(10), ZnPlus(10)):
    print(f)
```

```
{1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9, 0: 0}
{1: 3, 2: 6, 3: 9, 4: 2, 5: 5, 6: 8, 7: 1, 8: 4, 9: 7, 0: 0}
{1: 7, 2: 4, 3: 1, 4: 8, 5: 5, 6: 2, 7: 9, 8: 6, 9: 3, 0: 0}
{1: 9, 2: 8, 3: 7, 4: 6, 5: 5, 6: 4, 7: 3, 8: 2, 9: 1, 0: 0}
```

Quels sont les automorphismes de $(\mathbb{H}_8, \times)$?

[94]:
```
for f in isomorphismes(H8(), H8()):
    print(f)
```

```
{'-1': '-1', 'i': 'i', '-i': '-i', 'j': 'j', '-j': '-j', 'k': 'k', '-k': '-k',
'1': '1'}
{'-1': '-1', 'i': 'i', '-i': '-i', 'j': '-j', '-j': 'j', 'k': '-k', '-k': 'k',
'1': '1'}
```

```
{'-1': '-1', 'i': 'i', '-i': '-i', 'j': 'k', '-j': '-k', 'k': '-j', '-k': 'j',
'1': '1'}
{'-1': '-1', 'i': 'i', '-i': '-i', 'j': '-k', '-j': 'k', 'k': 'j', '-k': '-j',
'1': '1'}
{'-1': '-1', 'i': '-i', '-i': 'i', 'j': 'j', '-j': '-j', 'k': '-k', '-k': 'k',
'1': '1'}
{'-1': '-1', 'i': '-i', '-i': 'i', 'j': '-j', '-j': 'j', 'k': 'k', '-k': '-k',
'1': '1'}
{'-1': '-1', 'i': '-i', '-i': 'i', 'j': 'k', '-j': '-k', 'k': 'j', '-k': '-j',
'1': '1'}
{'-1': '-1', 'i': '-i', '-i': 'i', 'j': '-k', '-j': 'k', 'k': '-j', '-k': 'j',
'1': '1'}
{'-1': '-1', 'i': 'j', '-i': '-j', 'j': 'i', '-j': '-i', 'k': '-k', '-k': 'k',
'1': '1'}
{'-1': '-1', 'i': 'j', '-i': '-j', 'j': '-i', '-j': 'i', 'k': 'k', '-k': '-k',
'1': '1'}
{'-1': '-1', 'i': 'j', '-i': '-j', 'j': 'k', '-j': '-k', 'k': 'i', '-k': '-i',
'1': '1'}
{'-1': '-1', 'i': 'j', '-i': '-j', 'j': '-k', '-j': 'k', 'k': '-i', '-k': 'i',
'1': '1'}
{'-1': '-1', 'i': '-j', '-i': 'j', 'j': 'i', '-j': '-i', 'k': 'k', '-k': '-k',
'1': '1'}
{'-1': '-1', 'i': '-j', '-i': 'j', 'j': '-i', '-j': 'i', 'k': '-k', '-k': 'k',
'1': '1'}
{'-1': '-1', 'i': '-j', '-i': 'j', 'j': 'k', '-j': '-k', 'k': '-i', '-k': 'i',
'1': '1'}
{'-1': '-1', 'i': '-j', '-i': 'j', 'j': '-k', '-j': 'k', 'k': 'i', '-k': '-i',
'1': '1'}
{'-1': '-1', 'i': 'k', '-i': '-k', 'j': 'i', '-j': '-i', 'k': 'j', '-k': '-j',
'1': '1'}
{'-1': '-1', 'i': 'k', '-i': '-k', 'j': '-i', '-j': 'i', 'k': '-j', '-k': 'j',
'1': '1'}
{'-1': '-1', 'i': 'k', '-i': '-k', 'j': 'j', '-j': '-j', 'k': '-i', '-k': 'i',
'1': '1'}
{'-1': '-1', 'i': 'k', '-i': '-k', 'j': '-j', '-j': 'j', 'k': 'i', '-k': '-i',
'1': '1'}
{'-1': '-1', 'i': '-k', '-i': 'k', 'j': 'i', '-j': '-i', 'k': '-j', '-k': 'j',
'1': '1'}
{'-1': '-1', 'i': '-k', '-i': 'k', 'j': '-i', '-j': 'i', 'k': 'j', '-k': '-j',
'1': '1'}
{'-1': '-1', 'i': '-k', '-i': 'k', 'j': 'j', '-j': '-j', 'k': 'i', '-k': '-i',
'1': '1'}
{'-1': '-1', 'i': '-k', '-i': 'k', 'j': '-j', '-j': 'j', 'k': '-i', '-k': 'i',
'1': '1'}
```

Les groupes $(\mathbb{Z}_{11}^*, \otimes)$ et $(\mathbb{Z}_{10}^*, \oplus)$ ont le même cardinal. Sont-ils isomorphes ?

[95]:
```
print(sont_isomorphes(groupe_inversibles(ZnMult(11)), ZnPlus(10)))
```

```
(True, {2: 1, 3: 8, 4: 2, 5: 4, 6: 9, 7: 7, 8: 3, 9: 6, 10: 5, 1: 0})
```

Oui, $(\mathbb{Z}_{11}^*, \otimes) \simeq (\mathbb{Z}_{10}, \oplus)$.

Les groupes $(\mathbb{D}_3, \circ)$ et $(\mathfrak{S}_3, \circ)$ ont le même cardinal. Sont-ils isomorphes ?

[96]:
```
sont_isomorphes(D(3), S3())
```

[96]:
```
(True,
 {(1, 0): (2, 3, 1),
  (2, 0): (3, 1, 2),
  (0, 1): (1, 3, 2),
  (1, 1): (2, 1, 3),
  (2, 1): (3, 2, 1),
  (0, 0): (1, 2, 3)})
```

Oui. Effectivement, $(\mathbb{D}_3, \circ) \simeq (\mathfrak{S}_3, \circ)$.

Les groupes $(\mathbb{D}_4, \circ)$ et $(\mathbb{Q}_8, \times)$ ont le même cardinal. Pourtant, ils ne sont pas isomorphes :

[97]:
```
print(sont_isomorphes(D(4), H8()))
```

```
(False, None)
```

Dernier exemple : $(\mathbb{Z}_{10}, \oplus) \simeq (\mathbb{Z}_5 \times \mathbb{Z}_2, \oplus)$.

[98]:
```
print(sont_isomorphes(ZnPlus(10), produit(ZnPlus(5), ZnPlus(2))))
```

```
(True, {1: (1, 1), 2: (2, 0), 3: (3, 1), 4: (4, 0), 5: (0, 1), 6: (1, 0), 7: (2,
1), 8: (3, 0), 9: (4, 1), 0: (0, 0)})
```

1.6.5 5.4 Perspectives

Trouver un isomorphisme entre deux groupes n'est pas une chose facile. Montrer qu'ils ne sont pas isomorphes ne l'est pas plus. Remarquons cependant que *si* deux groupes sont isomorphes, alors ils ont un certain nombre de propriétés en commun. Appelons f un isomorphisme du groupe G sur le groupe H.

- Pour tout $x \in G$, x et $f(x)$ ont le même ordre. G et H ont donc exactement le même nombre d'éléments de chaque ordre. Si ce n'est pas le cas, G et H ne sont pas isomorphes. Cette propriété peut partfois limiter fortement le nombre de bijections de G sur H qui sont des isomorphismes. À supposer que G et H ont le même nombre d'éléments de chaque ordre, une bijection admissible envoie bijectivement les éléments de G d'un ordre donné sur les éléments de H du même ordre. Nous *pourrions* écrire une fonction qui renvoie uniquement les bijections de G sur H qui vérifient cette propriété.
- L'image d'un sous-groupe de G par f est un sous-groupe de H. Ainsi, f induit une bijection de l'ensemble des sous-groupes de G sur l'ensemble des sous-groupes de H. Ceci nous donne une condition suffisante de non-isomorphisme, ainsi que des conditions sur les bijections qui pourraient être des isomorphismes.

- Enfin, un groupe fini peut être *représenté* par des *générateurs* et des *relations* Pour prendre l'exemple de $\mathbb{D}_n$, ce groupe est engendré par la rotation r et la symétrie s, avec les relations

$$r^n = id,\ s^2 = id,\ r \circ s = s \circ r^{-1}$$

Un groupe G isomorphe à $\mathbb{D}_n$ doit nécessairement posséder deux tels éléments ρ et σ et l'un des isomorphismes de G sur $\mathbb{D}_n$ envoie ρ sur r et σ sur s.

Bref, ces remarques nous montrent que nos fonctions du genre « force brute » sont bien loin d'avoir épuisé le sujet …