MT10 – TP1

Antoine COLLAS, Tran Thi Son Nu

5 février 2018

1 Test d'une loi :

Question 1

Tout d'abord nous commençons par coder l'ensemble $\mathbb{Z}/4\mathbb{Z}$ à l'aide d'une liste :

```
Z4=[0,1,2,3]
```

Ensuite, nous formons le tableau correspondant à la loi $(\mathbb{Z}/4\mathbb{Z}, +, 0)$:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Nous codons la loi + sur $(\mathbb{Z}/4\mathbb{Z}, +, 0)$ en utilisant des listes de listes de la manière suivante :

```
t1=[[0,1,2,3],[1,2,3,0],[2,3,0,1],[3,0,1,2]]
```

Nous codons l'ensemble $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$ à l'aide d'une liste de tuples pour pouvoir coder plus facilement la loi +. En effet, dans le tableau de la loi + nous remplacerons (0,0) par 0, (1,0) par 1, (0,1) par 2 et (1,1) par 3. Ainsi pour tous les ensembles nous numérotons les éléments pour par la suite programmer plus facilement des programmes génériques. Cette liste permet de faire la correspondance entre ces deux notations :

```
Z2=[(0,0),(1,0),(0,1),(1,1)]
```

Ainsi Z2[0] nous renvoie (0,0).
Tableau correspondant à la loi $(\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}, +, 0)$:

+	(0,0)	(1,0)	(0,1)	(1,1)
(0,0)	(0,0)	(1,0)	(0,1)	(1,1)
(1,0)	(1,0)	(0,0)	(1,1)	(0,1)
(0,1)	(0,1)	(1,1)	(0,0)	(1,0)
(1,1)	(1,1)	(0,1)	(1,0)	(0,0)

Nous codons la loi + sur $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$ de la manière suivante :

```
t2=[[0,1,2,3],[1,0,3,2],[2,3,0,1],[3,2,1,0]]
```

Pour la loi $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$, les

Question 2

Une autre manière de former les tables de loi est d'exécuter les instructions Sage suivantes :

```
Z4=CyclicPermutationGroup(4)
Z4.cayley_table()
```

Ce qui nous renvoie le résultat suivant :

*	a	b	c	d
a	a	b	c	d
b	b	c	d	a
c	c	d	a	b
d	d	a	b	c

Question 3

Le programme suivant permet de connaitre si une loi, modélisée par sa table, admet un élément neutre. Si la loi admet un élément neutre, le programme renvoie le numéro de l'élément dans l'ensemble. Si la loi n'admet pas d'élément neutre, le programme renvoie 'NULL'.

```
def neutre(t):
    alpha=0
    neutre=0
    while (neutre==0) and (alpha<len(t)):
        neutre=1
        for i in range(len(t)):
                    if (t[i][alpha]!=i) or (t[alpha][i]!=i):
                                neutre=0
                alpha=alpha+1
    if neutre==1:
                return(alpha-1)
    else:
        return 'NULL'
```

Z4[neutre(t1)] renvoie 0. Donc 0 est le neutre de $\mathbb{Z}/4\mathbb{Z}$.
Z2[neutre(t2)] renvoie (0,0). Donc (0,0) est le neutre de $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$.
Une table prise au hasard :
t3=[[1,2,0,3], [3,2,1,0],[1,0,2,3],[1,2,3,0]]
neutre(t3) renvoie 'NULL' donc la loi modélisée par t3 n'admet pas d'élément neutre.

Question 4

Le programme ci-dessous permet de trouver le symétrique de chaque élément, s'il existe. Nous parcourons la matrice et testons la condition donnée dans l'énoncé : E[i] et E[j] sont symétriques $sit[i,j] = alphaett[j,i] = alpha$. Le programme affiche les éléments symétriques au fur et à mesure puis renvoie la tables des symétriques.

```
def symetrique(t):
    alpha=neutre(t)
    sym=[-1]*len(t) #on remplace les -1 par les differents
       symetriques pour former la table des symetriques
    for i in range(len(t)):
        for j in range(i,en(t)):
            if (t[i][j]==alpha) and (t[j][i]==alpha):
                print str(i)+" et "+str(j)+" sont symetriques"
                    sym[i]=j
                sym[j]=i
    return sym
```

symetrique(t1) permet d'obtenir les symétriques des éléments de $\mathbb{Z}/4\mathbb{Z}$:
0 et 0 sont symétriques
1 et 3 sont symétriques
2 et 2 sont symétriques
$[0, 3, 2, 1]$

Donc la table des symétriques de $\mathbb{Z}/4\mathbb{Z}$ est :

0	0
1	3
2	2
3	1

La commande symetrique(t2) permet d'obtenir les symétriques des éléments de $\mathbb{Z}/4\mathbb{Z}$:
0 et 0 sont symétriques
1 et 1 sont symétriques
2 et 2 sont symétriques
3 et 3 sont symétriques
$[0, 1, 2, 3]$

En utilisant la numérotation des éléments nous obtenons la table des symétriques de $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$:

(0,0)	(0,0)
(1,0)	(1,0)
(0,1)	(0,1)
(1,1)	(1,1)

Question 5

Nous utilisons la condition donnée dans l'énoncé pour tester si une loi est associative. Pour cela nous utilisons trois boucles imbriquées et testons la condition à chaque itération. Nous obtenons le programme suivant :

```
def associativite(t):
    for i in range(len(t)):
        for j in range(len(t)):
            for k in range(len(t)):
                if t[t[i][j]][k]!=t[i][t[j][k]]:
                    return false
    return true
```

associative(t1) renvoie true donc la loi + sur $\mathbb{Z}/4\mathbb{Z}$ est associative.
associative(t2) renvoie true donc la loi + sur $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$ est associative.

Question 6

TestSuite est une classe qui appartient à la super classe "unittest". Elle permet de réaliser des tests sur un groupe ou bien un objet de Sagemath. Si un test affiche "pass" alors le test a été passé avec succès. La commande "TestSuite??" nous donne des détails sur le fonctionnement de TestSuite. La structure de TestSuite a un constructeur et une méthode run. Le constructeur prend le paramètre l'objet sur lequel on va effectuer des tests. La méthode run() normalement se charge d'appeler tous les tests commençant par "_test_" dans l'ordre alphabétique. En fait, la méthode run() se base sur la fonction "built-in" dir() de Python permettant d'indiquer les noms de structure de l'objet. Grâce à la fonction "built-in" getattr() de Python, run() peut définir les méthodes effectuant les tests, et les appeler. Si des tests ont échoué, la méthode run() se charge également de les afficher. "verbose=True" va afficher tous les résultats des tests.

```
sage: Z4=CyclicPermutationGroup(4)
sage: TestSuite(Z4).run(verbose=True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . .
 pass
  running ._test_eq() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_inverse() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

2 Test d'un morphisme :

Question 7

Pour tester si une application entre deux groupes est un morphisme il n'y a qu'une condition à vérifier : $(\forall x, y \in G) f(x * y) = f(x) *' f(y)$
Nous vérifions cette condition à l'aide de deux boucles imbriquées :

```
def test_morphisme(t,t_prime,f):
    for i in range(len(t)):
        for j in range(len(t)):
            if f[t[i][j]]!=t_prime[f[i]][f[j]]:
                return false
    return true
```

Nous cherchons à construire toutes les applications f bijectives de G, groupe d'ordre n, dans G possibles, ce qui revient à faire toutes les permutations de l'ensemble 0,...,n-1. pour cela nous utilisons le module itertools et l'instruction
f=list(permutations(range(len(t)))) où t est la table de loi de G
Cette instructions génère une liste de toutes les permutations de l'ensemble $0, ..., n-1$. Le programme suivant teste une à une les applications. Si une application est un morphisme alors c'est un automorphisme puisque les applications vont de G dans G et sont bijectives.

```
from itertools import permutations

def auto(t):
    f=list(permutations(range(len(t))))
    #pour chaque application bijective on teste si on a un morphisme de G dans G
    automorphisme=[] #cette variable va contenir tous les automorphismes de G ds G
    for k in range(len(f)):
        if test_morphisme(t,t,f[k])==true:
            automorphisme.append(f[k])
    return automorphisme
```

auto(t1) renvoie deux automorphismes : $[(0,1,2,3),(0,3,2,1)]$
auto(t2) renvoie six automorphismes : $[(0,1,2,3),(0,1,3,2),(0,2,1,3),(0,2,3,1),(0,3,1,2),(0,3,2,1)]$

On cherche à construire toutes les applications f bijectives de $\mathbb{Z}/4\mathbb{Z}$ dans $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$ possibles, ce qui revient une nouvelle fois à construire toutes les permutations de l'ensemble 0,...,n-1. Pour chaque application bijective nous testons si on a un morphisme de groupe.

```
def iso(t,t_prime):
    f=list(permutations(range(len(t))))
    isomorphisme=[] #cette variable va contenir tous les isomorphismes de G ds G'
    for k in range(len(f)):
        if test_morphisme(t,t_prime,f[k])==true:
            return true
    return false
```

iso(t1,t2) renvoie False. Nous ne testons pas iso(t2,t1) parce que nous savons que l'application réciproque d'un isomorphisme de groupe est un isomorphisme de groupe donc s'il y a un ismorphisme de Z/2Z x Z/2Z -> Z/4Z on aurait du obtenir son application réciproque dans iso(t1,t2).

3 Où l'on identifie tous les groupes d'ordre 4 :

Question 8

Supposons qu'il existe un élément $y \in E$ qui apparaisse au moins 2 fois sur une ligne donc

$$\exists x1, x2, x3 \in E \; tels \; que \; x1 * x2 = y \; et \; x1 * x3 = y \; avec \; x2 \neq x3 \tag{1}$$

$$\Rightarrow x1 * x2 = x1 * x3 \tag{2}$$

En appliquant le symétrique (on souhaite que (E,*,e) soit un groupe) de x1 de chaque coté on obtient $x2 = x3$ ce qui est contradictoire avec l'hypothèse de départ.
De même pour les colonnes :
Supposons qu'il existe un élément $y \in E$ qui apparaisse au moins 2 fois sur une colonne donc

$$\exists x1, x2, x3 \in E \; tels \; que \; x2 * x1 = y \; et \; x3 * x1 = y \; avec \; x2 \neq x3 \tag{3}$$

$$\Rightarrow x2 * x1 = x3 * x1 \tag{4}$$

En appliquant le symétrique (on souhaite que (E,*,e) soit un groupe) de x1 de chaque coté on obtient $x2 = x3$ ce qui est contradictoire avec l'hypothèse de départ.

Donc une condition nécessaire pour obtenir un groupe est que chaque élément apparaisse au maximum une fois sur chaque colonne et chaque ligne. Or comme nous souhaitons que (E,*,e) soit un groupe il faut que * soit un loi de composition interne donc $\forall x, x' \in E \Rightarrow x * x' \in E$ donc chaque case de la table doit avoir un élément. Comme il y a autant de lignes/colonnes que d'éléments dans E, chaque élément doit apparaitre une et une seule fois dans chaque ligne et chaque colonne.

Question 9

Nous modélisons l'ensemble E ainsi que les lois de la manière suivante : E=['a0','a1','a2','a3']

```
E=['a0','a1','a2','a3']
l1=[[0,1,2,3],[1,0,3,2],[2,3,1,0],[3,2,0,1]]
l2=[[0,1,2,3],[1,0,3,2],[2,3,0,1],[3,2,1,0]]
l3=[[0,1,2,3],[1,2,3,0],[2,3,0,1],[3,0,1,2]]
l4=[[0,1,2,3],[1,3,0,2],[2,0,3,1],[3,2,1,0]]
```

Les 4 tables représentent des lois de compostions internes, en effet, $\forall x, x' \in E \Rightarrow x * x' \in E$.
Nous vérifions que chaque loi admet un élément neutre :

```
E[neutre(l1)]
```

'a0'

```
E[neutre(l2)]
```

'a0'

```
E[neutre(l3)]
```

'a0'

```
E[neutre(l4)]
```

'a0'
Donc chaque table admet un élément neutre qui est $e = a_0 \in E$

Nous vérifions que chaque loi est associative :

```
associativite(l1)
```

TRUE

```
associativite(l2)
```

TRUE

```
associativite(l3)
```

TRUE

```
associativite(l4)
```

TRUE
Donc toutes les lois sont associatives.

Enfin nous vérifions que tous les éléments de E ont chacun un symétrique dans les différentes lois. En utilisant l'algorithme de la question 4 nous obtenons les tables des symétriques suivantes :

Loi I :

a0	a0
a1	a1
a2	a3
a3	a2

Loi II :

a0	a0
a1	a1
a2	a2
a3	a3

Loi III :

a0	a0
a1	a3
a2	a2
a3	a1

Loi IV :

a0	a0
a1	a2
a2	a1
a3	a3

Les 4 tables sont les seules tables de groupes à 4 éléments. En effet, en utilisant les deux conditions $e = a_0$ et former un carré latin, on ne peut former que ces tables. Former un carré latin est une condition nécessaire pour obtenir une structure de groupe. Si nous choisissons un autre élément neutre, alors il est possible de trouver de nouvelles tables. Cependant ces nouvelles tables sont isomorphes à moins une des tables construites avec $e = a_0$.

Ensuite nous cherchons les tables qui sont isomorphes à $(\mathbb{Z}/4\mathbb{Z}, +, 0)$. Nous rappelons que la loi $+$ sur $\mathbb{Z}/4\mathbb{Z}$ est modélisée par la table t1.

```
iso(l1,t1)
```

True

```
iso(l2,t1)
```

False

```
iso(l3,t1)
```

True

```
iso(l4,t1)
```

True

Donc les tables 1,3 et 4 sont isomorphes à $\mathbb{Z}/4\mathbb{Z}$.
Nous cherchons maintenant les tables isomorphes à $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$

```
iso(l1,t2)
```

False

```
iso(l2,t2)
```

True

```
iso(l3,t2)
```

False

```
iso(l4,t2)
```

False

Seule la table 2 est isomorphe à $\mathbb{Z}/2\mathbb{Z}$ x $\mathbb{Z}/2\mathbb{Z}$.

4 Utiliser les fonctionnalités de Sage

Question 10

Sage permet de travailler sur des groupes d'ordres supérieurs : par exemple :

```
C6=CyclicPermutationGroup(6)
C6.cayley_table()
```

*	a	b	c	d	e	f
a	a	b	c	d	e	f
b	b	c	d	e	f	a
c	c	d	e	f	a	b
d	d	e	f	a	b	c
e	e	f	a	b	c	d
f	f	a	b	c	d	e

```
TestSuite(C6).run(verbose=True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_inverse() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

Nous voyons que le groupe des permutations d'ordre 6 est bien un groupe puisque la loi dispose d'un élément neutre, et elle est associative, et symétrique.

```
C6.cayley_graph().show3d()
```