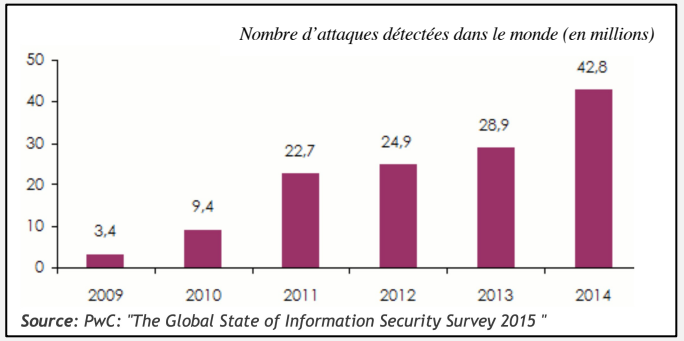
LE GROUPE DES TRESSES ET SES APPLICATIONS EN CRYPTOGRAPHIE

AXEL BENYAMINE
NUMÉRO DE CANDIDAT : 29530

INTRODUCTION

- La cryptographie en tant qu'enjeu sociétal



- Problématique :

Quelles sont les propriétés définissant le groupe des tresses ?

Dans quelle mesure permet-il de réaliser un système cryptographique performant et sûr ?

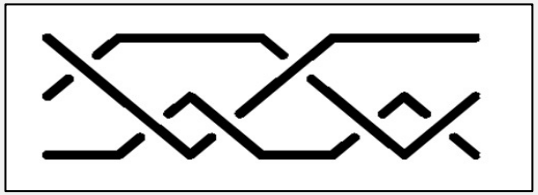
- Sommaire:

LE GROUPE DES TRESSES

NOTION D'ISOTOPIE

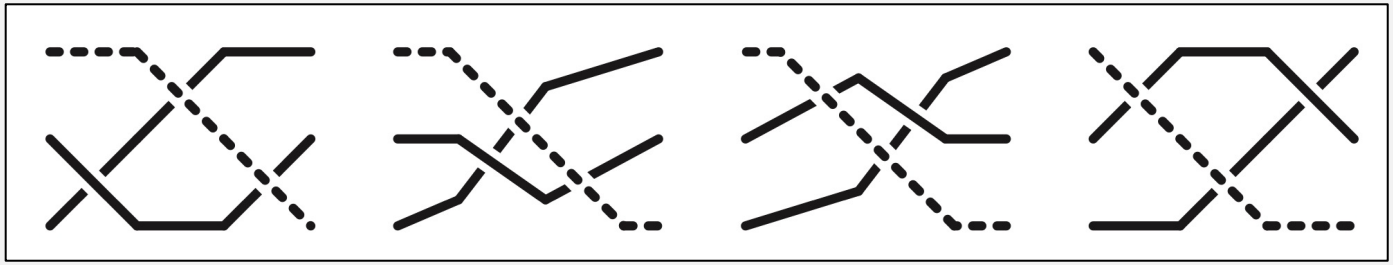
- Une tresse :

- Représentation d'une tresse à 3 brins :



- Isotopie :

a et *b* appartiennent à la même classe d'isotopie si on peut passer de *a* à *b* par déformations successives



Figures extraites de [9]

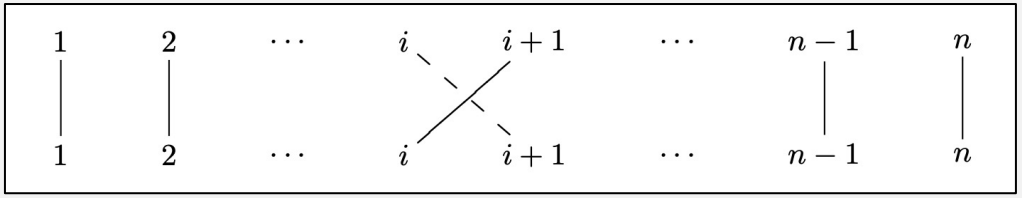
DÉFINITION DU GROUPE DES TRESSES: UNE DÉFINITION ALGÉBRIQUE

- B_n est le groupe engendré par les n-1 générateurs σ_i pour i ∈ [1, n-1]
- Propriétés des générateurs : $\begin{cases} \sigma_i\sigma_j = \sigma_j\sigma_i & si\ |i-j| > 1 \\ \sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j & si\ |i-j| = 1 \end{cases}$
- Remarque : B_n est non commutatif
- A^* : ensemble des mots sur l'alphabet $A = \{\sigma_i , i \in [1, n-1]\} \cup \{\sigma_i^{-1} , i \in [1, n-1]\}$
- A_+^* : ensemble des mots sur l'alphabet $A_+ = \{\sigma_i , i \in [1, n-1]\}$

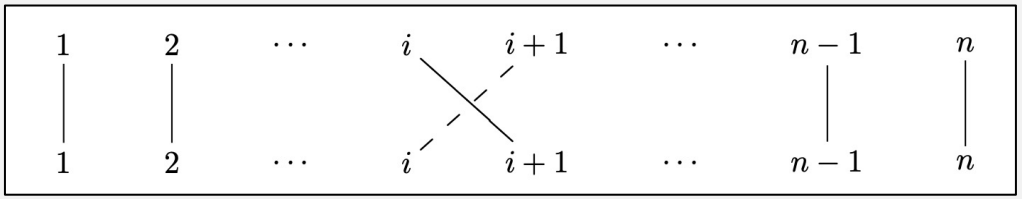
DÉFINITION DU GROUPE DES TRESSES:

COHÉRENCE DE LA DÉFINITION ALGÉBRIQUE AVEC LA GÉOMÉTRIE DES TRESSES

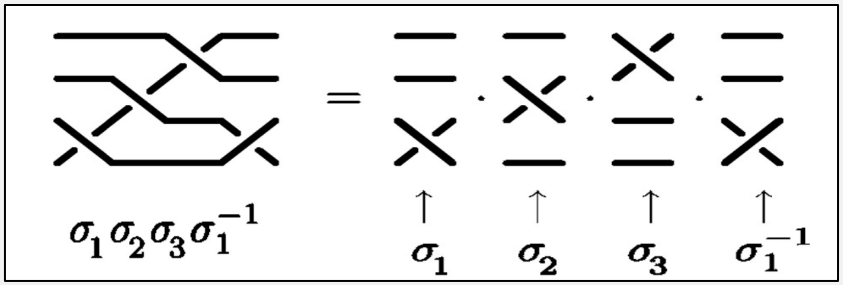
- Représentation de σ_i :



- Représentation de σ_i^{-1} :



- Un exemple de produit de tresses :

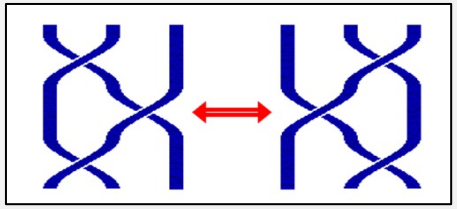


Figures extraites de [5] et [9]

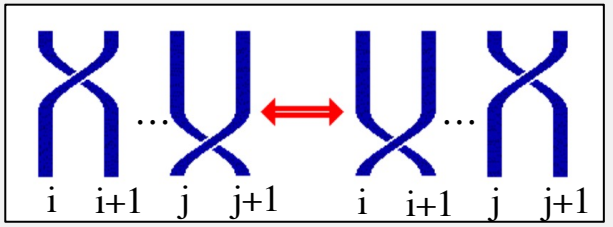
DÉFINITION DU GROUPE DES TRESSES:

COHÉRENCE DE LA DÉFINITION ALGÉBRIQUE AVEC LA GÉOMÉTRIE DES TRESSES

- Illustration de l'égalité $\sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j \quad si\ |i-j| = 1$



- Illustration de l'égalité $\sigma_i\sigma_j = \sigma_j\sigma_i \qquad si\ |i-j| > 1$



Figures extraites de [5]

IMPLÉMENTATION SUR PYTHON DU GROUPE DES TRESSES

- Implémentation récursive par les générateurs
- Caractérisation des tresses par 3 objets : tresse.gen, tresse.puiss et tresse.tail

tels que $tresse = \sigma_{tresse.gen}^{\,tresse.puiss} \times tresse.tail$

- Produit de tresses :

Si $t_1 = \sigma_d^{\,p} \times q_1$ et $t_2 = \sigma_b^{\,q} \times q_2$

Alors $t_1 \times t_2 = \sigma_d^{\,p} \times q_1 \times \sigma_b^{\,q} \times q_2 = \sigma_d^{\,p} \times (q_1 \times t_2)$

D'où
$$\begin{cases} (t_1 \times t_2).gen = t_1.gen \\ (t_1 \times t_2).puiss = t_1.puiss \\ (t_1 \times t_2).tail = t_1.tail \times t_2 \end{cases}$$

DES PROPRIÉTÉS ALGÉBRIQUES

PROBLÈME DE MOTS
PROBLÈME DE CONJUGAISON
PROBLÈME DE RECHERCHE DU CONJUGUANT

- Problème de mots :

  Déterminer si deux mots a et b représentent la même tresse

- Problème de conjugaison :

  Déterminer si x et y sont conjugués

- Problème de recherche du conjuguant :

  Déterminer a à partir de x et $a^{-1}xa$

LE RETOURNEMENT DE MOTS

- Transformation $\rightarrow$: $\forall (a,b) \in A^*$,
$$\begin{cases} a\sigma_i{}^{-1}\sigma_i b \rightarrow ab & \forall i \\ a\sigma_i{}^{-1}\sigma_j b \rightarrow a\sigma_j\sigma_i\sigma_j{}^{-1}\sigma_i{}^{-1}b & si\ |i-j| = 1 \\ a\sigma_i{}^{-1}\sigma_j b \rightarrow a\sigma_j\sigma_i{}^{-1}b & si\ |i-j| > 1 \end{cases}$$

- Si $a \rightarrow b$ Alors a et b appartiennent à la même classe d'isotopie

- En appliquant itérativement les règles de transformation $\rightarrow$ à un mot ω :

  Existence et unicité d'un mot terminal $N(\omega) \times D(\omega)^{-1}$ où $(N(\omega), D(\omega)) \in A_+^*$

  On note $\omega \rightarrow^* N(\omega)\, D(\omega)^{-1}$

- ω représente la tresse triviale ssi $D(\omega)\, N(\omega)^{-1} \rightarrow^*$ *mot vide*

UN EXEMPLE DE RETOURNEMENT DE MOTS

$$\omega = \sigma_3^{-1}\sigma_3^{-1}\sigma_2\sigma_1^{-1}\,\sigma_3^{-1}\sigma_3^{-1}\sigma_2\sigma_1^{-1}\,\sigma_3^{-1}\sigma_2\sigma_3\sigma_2^{-1}\sigma_3^{-1}\sigma_1^{-1}\,\sigma_2\sigma_3\sigma_2^{-1}\sigma_3^{-1}\sigma_3\sigma_2^{-1}\sigma_3^{-1}\sigma_1^{-1}\,\sigma_2\sigma_3\sigma_2^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_1^{-1}$$

$$N(\omega) = \sigma_2\sigma_3 \qquad D(\omega) = \sigma_1\sigma_3\sigma_2\sigma_2$$

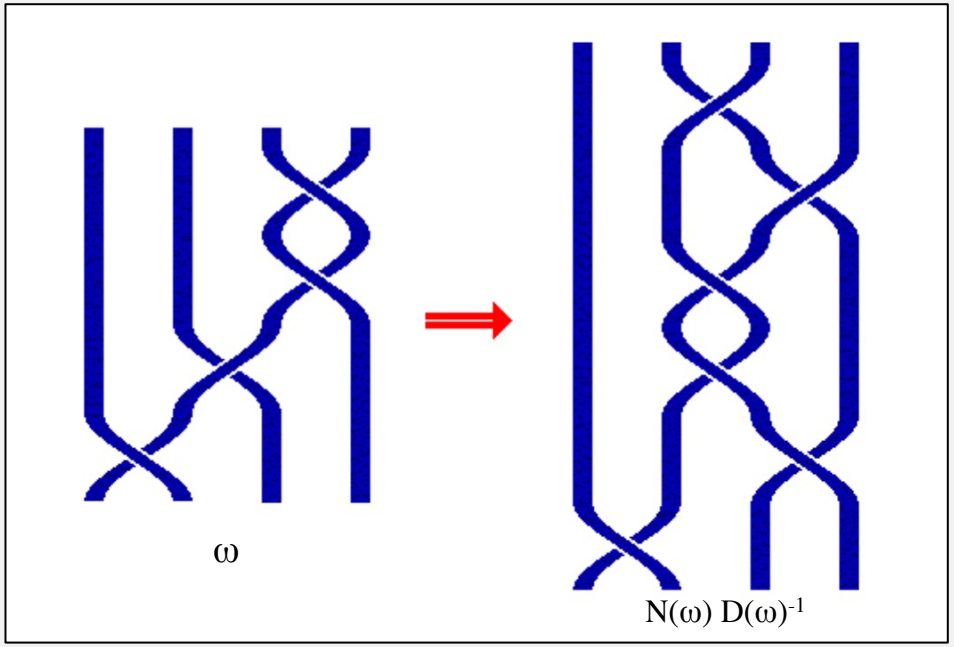


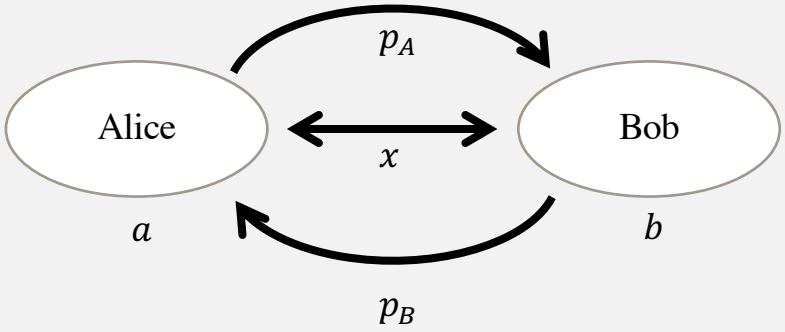
Figure extraites de [5]

SYSTÈME CRYPTOGRAPHIQUE

PRINCIPE DU SYSTÈME CRYPTOGRAPHIQUE

- Alice et Bob se munissent de $x \in B_n$
- Alice se munit de $a \in B_{1,r}$
- Bob se munit de $b \in B_{r+1,n}$

$$B_{i,j} = \langle \sigma_k, k \in [i, j-1] \rangle$$



- Alice rend public $p_A = a^{-1}xa$ (sous la forme $N(p_A)\ D(p_A)^{-1}$)
- Bob rend public $p_B = b^{-1}xb$ (sous la forme $N(p_B)\ D(p_B)^{-1}$)

$x = \sigma_6^1\sigma_5^1\sigma_6^{-1}\sigma_5^1\sigma_4^1$
$a = \sigma_2^1\sigma_4^{-1}\sigma_2^1$
$a^{-1}xa = \sigma_2^{-1}\sigma_4^1\sigma_2^{-1}\sigma_6^1\sigma_5^1\sigma_6^{-1}\sigma_5^1\sigma_4^1\sigma_2^1\sigma_4^{-1}\sigma_2^1$
$N(p_A)\ D(p_A)^{-1} = \sigma_4^1\sigma_6^1\sigma_5^1\sigma_5^1\sigma_6^1\sigma_4^1\sigma_5^1\sigma_4^{-1}\sigma_5^{-1}\sigma_6^{-1}\sigma_4^{-1}$

- Alice et Bob calculent la clef privée $K = a^{-1}p_Ba = b^{-1}p_Ab$

NÉCESSITÉ D'UNE FONCTION HACHAGE

- Comment crypter un message avec une tresse ?
- $H : B_n \rightarrow \{0,1\}^N$
- Le message M : suite de N bits
- Le message codé $M' = M \oplus H(K)$
- Pour décoder : $M' \oplus H(K) = M \oplus 2.H(K) = M$

$\oplus$ Addition bit à bit modulo 2

$M=$	01111
$H(K) =$	01100
$\oplus$	$00011 = M'$

$M' =$	00011
$H(K) =$	01100
$\oplus$	$01111 = M$

LA REPRÉSENTATION (NON RÉDUITE) DE BURAU

- Morphisme $\beta : B_n \to GL_n(\mathbb{Z}[t, t^{-1}])$
- Définition sur les σ_i :

$$\beta(\sigma_i) = \begin{pmatrix} I_{i-1} & 0 & 0 \\ 0 & \begin{matrix} 1-t & 1 \\ t & 0 \end{matrix} & 0 \\ 0 & 0 & I_{n-i-1} \end{pmatrix} = Diag\left(I_{i-1}, \begin{pmatrix} 1-t & 1 \\ t & 0 \end{pmatrix}, I_{n-i-1}\right)$$

- On retrouve bien :

$$\begin{cases} \beta(\sigma_i)\beta(\sigma_j) = \beta(\sigma_j)\beta(\sigma_i) & si\ |i-j| > 1 \\ \beta(\sigma_i)\beta(\sigma_j)\beta(\sigma_i) = \beta(\sigma_j)\beta(\sigma_i)\beta(\sigma_j) & si\ |i-j| = 1 \end{cases}$$

CONSTRUCTION DE LA FONCTION HACHAGE

- On prend arbitrairement t=10 :

$$\beta : B_n \to GL_n(\mathbb{Q})$$

- $\gamma : \left(\frac{a_{i,j}}{b_{i,j}}\right)_{1 \le i,j \le n} \longmapsto \overline{a_{1,1}b_{1,1} \dots a_{1,n}b_{1,n}a_{2,1} \dots b_{n,n}}$ où $\overline{x}$ est l'écriture (non signée) en base 10 de x

- $sha256 : Chaîne\ de\ caractères \to \{0,1\}^{256}$

- $H = sha256 \circ \gamma \circ \beta$

IMPLÉMENTATION DE CETTE REPRÉSENTATION

- Nécessité de tests d'égalité :

  Création d'une classe « rationnels »

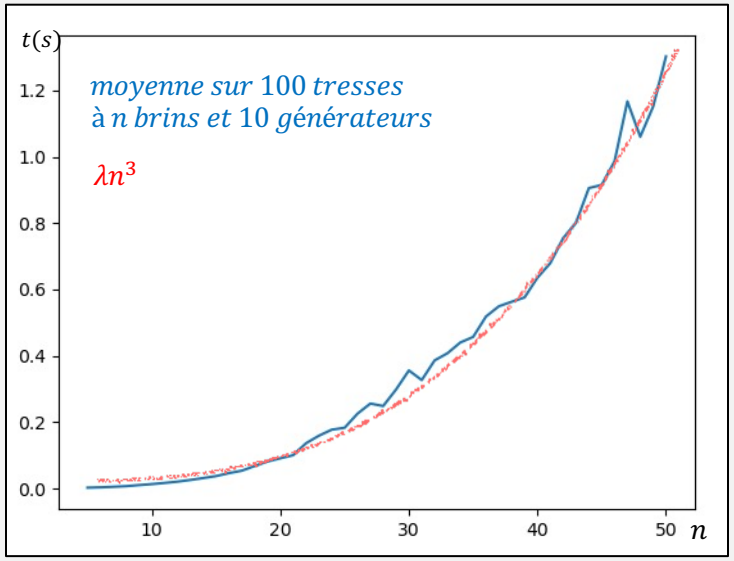
  Définition d'un élément par $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$

  $$où \begin{cases} a \wedge b = 1 \; si \; a \neq 0 \\ b = 1 \; si \; a = 0 \end{cases}$$

- Complexité de β :

  $O(l.n^3)$ où l est le nombre de générateurs

- Rapidité de H :



FIDÉLITÉ DE LA REPRÉSENTATION DE BURAU

- Un contre-exemple:

$$\psi_1 = \sigma_3^{-1}\sigma_2\sigma_1^2\sigma_2\sigma_4^3\sigma_3\sigma_2 \qquad \psi_2 = \sigma_4^{-1}\sigma_3\sigma_2\sigma_1^{-2}\sigma_2\sigma_1^2\sigma_2^2\sigma_1\sigma_4^5$$

$$a = \psi_1^{-1}\sigma_4\psi_1 \qquad b = \psi_2^{-1}\sigma_4\sigma_3\sigma_2\sigma_1^2\sigma_2\sigma_3\sigma_4\psi_2$$

$$\beta(ab) = \beta(ba)$$

- Tests d'infidélité:

  -Prendre deux tresses aléatoires t et t'

  -Vérifier l'égalité si $\beta(t) = \beta(t')$

- Pour $n = 10$: $\quad 50.10^6\ tests \rightarrow \begin{cases} 389\ égalités\ de\ représentation\ de\ Burau \\ 389\ égalités\ de\ tresse \end{cases}$

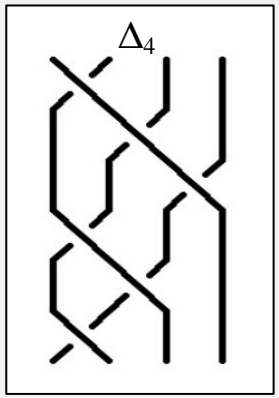
CRAQUAGE DU SYSTÈME

LA FORME NORMALE

- Le demi-tour de Garside Δ_n :

  Permutation suivant le schéma i devient n-i

  Diviseurs de Δ_n : tresses simples



- Toute tresse s'écrit de façon unique sous la forme

$$a = \Delta_n^{\,k} s_1 \ldots s_p$$

où $\forall\, i, \begin{cases} s_i \textit{ divise strictement (à gauche) } \Delta_n \\ s_i = \Delta_n \wedge s_i \ldots s_p \end{cases}$

Figure extraite de [7]

UNE SOLUTION AUX PROBLÈMES DE CONJUGAISON ET DE RECHERCHE DU CONJUGUANT

- Super Sumit Set de a : $SSS(a) = \{conjugués\ de\ a\ de\ longueur\ minimale\}$
- a et b sont conjugués ssi $SSS(a) = SSS(b)$
- Construction de $SSS(a)$:

  -Construction d'un élément e par opérations successives sur a

  -Construction de $SSS(a)$ par conjugaison successive de e

  →Résolution du problème de conjugaison

- En gardant en mémoire les conjugaisons successives pour passer de e à tout élément de SSS

  → Résolution du problème de recherche du conjuguant

RÉPONSE À LA PROBLÉMATIQUE

RÉPONSE À LA PROBLÉMATIQUE

- Rapidité de la transmission d'un message :
  - Durée de l'encodage et du décodage ≈ entre 1 sec et 1 min
- Un algorithme de craquage :
  - Très couteux : en moyenne $O(n!)$
  - Très lent : 20 min $pour\ n = 4$
- Limites
  - Transmission qui laisse entrevoir des informations
  - Complexités difficilement maîtrisable (pour la transmission et le craquage)
  - Choix d'une implémentation récursive sur python

n \ l	1	2	3	4	5	6	7	8	9
2	0,24	0,91	1,28	2,62	5,47	8,67	14,3	14,5	20,4
3	0,47	1,42	3,10	5,68	8,27	15,8	20,4	24,0	30,1
4	35,6	136	175	434	Erreur	Erreur	Erreur	Erreur	Erreur

Temps nécessaire au craquage (s)

ANNEXES

LA FORME NORMALE

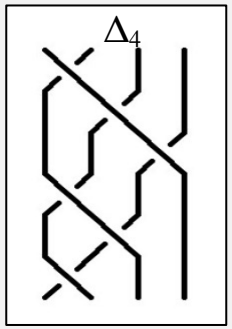
- Le demi-tour de Garside Δ_n :

  Permutation suivant le schéma i devient n-i

  $\Delta_n = (\sigma_1 \ldots \sigma_i) \ldots (\sigma_1 \ldots \sigma_{i+1})(\sigma_i \ldots \sigma_1) \ldots (\sigma_i) \quad \forall i$

  $\Delta_n \sigma_i = \sigma_{n-i} \Delta_n$

  Diviseurs de Δ_n : tresses simples



- Toute tresse s'écrit de façon unique sous la forme

$$a = \Delta_n^{\,k} s_1 \ldots s_p$$

où $\forall i$

$$\begin{cases} s_i \text{ divise strictement (à gauche) } \Delta_n \\ s_i = \Delta_n \wedge s_i \ldots s_p \end{cases}$$

- Détermination de la forme normale de a:

  -Ecriture sous la forme $a = \Delta_n^{-r} b$ où b positive (appartenant au monoïde engendré par les σ_i)

  -Ecriture de b sous la forme $b = \Delta_n^{\,l} s_1 \ldots s_p$ par calcul successif de pgcd

- Complexité en $O(l^2.n.log(n))$

Figure extraite de [7]

UNE SOLUTION AUX PROBLÈMES DE CONJUGAISON ET DE RECHERCHE DU CONJUGUANT

- Tresse a écrite sous forme normale $a = \Delta_n^k a_1 \dots a_p$
- Longueur canonique de a : $\|a\| = p$
- Classe de conjugaison de a : $C(a) = \{\sigma^{-1} a \sigma, \sigma \in B_n\}$
- Grande longueur de a : $gl(a) = min\{\|x\|, x \in C(a)\}$
- Super Summit Set de a : $SSS(a) = \{x \in C(a) : \|x\| = gl(a)\}$

- Automorphisme $\tau : \begin{cases} B_n \to B_n \\ x \longmapsto \Delta_n x \Delta_n \end{cases}$
- Cyclage $c : \begin{cases} B_n \to B_n \\ a \longmapsto \Delta_n^{-r} a_2 \dots a_p \tau^r(a_1) \end{cases}$
- Décyclage $d : \begin{cases} B_n \to B_n \\ a \longmapsto \Delta_n^{-r} \tau^{-r}(a_p) a_1 \dots a_{p-1} \end{cases}$

- a et b sont conjugués ssi *SSS(a) = SSS(b)*
- Construction de *SSS(a)* :

  -Construction d'un élément *e* par cyclage successif puis décyclage successif de *a*

  -Construction de *SSS(a)* par conjugaison successive de *e*
- En gardant en mémoire les conjugaisons successives pour passer de *e* à tout élément de SSS,

→ Résolution du problème de recherche du conjuguant

LE RETOURNEMENT DE MOTS

- Transformation $\rightarrow$: $\forall (a,b) \in A^*$,
$$\begin{cases} a\sigma_i{}^{-1}\sigma_i b \rightarrow ab & \forall i \\ a\sigma_i{}^{-1}\sigma_j b \rightarrow a\sigma_j\sigma_i\sigma_j{}^{-1}\sigma_i{}^{-1}b & si\ |i-j| = 1 \\ a\sigma_i{}^{-1}\sigma_j b \rightarrow a\sigma_j\sigma_i{}^{-1}b & si\ |i-j| > 1 \end{cases}$$

- Si $a \rightarrow b$ Alors a et b appartiennent à la même classe d'isotopie

- En appliquant itérativement les règles de transformation $\rightarrow$ à un mot ω :

  Existence et unicité d'un mot terminal $N(\omega) \times D(\omega)^{-1}$ où $(N(\omega), D(\omega)) \in A_+^*$

  On note $\omega \rightarrow^* N(\omega)\, D(\omega)^{-1}$

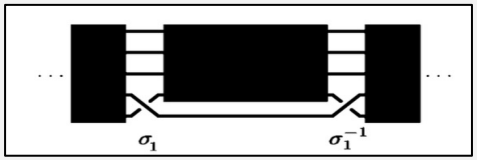
- ω représente la tresse triviale ssi $D(\omega)\, N(\omega)^{-1} \rightarrow^*$ *mot vide*
- Complexité en $O(9^n l^2)$ mais jamais atteinte en pratique

UNE SOLUTION (TRÈS RAPIDE) AU PROBLÈME DE MOTS : LA RÉDUCTION DES POIGNÉES

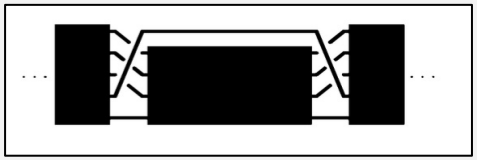
- Théorème : Si ω contient au moins un σ_1 et pas de σ_1^{-1}

  Alors ω ne représente pas la tresse triviale

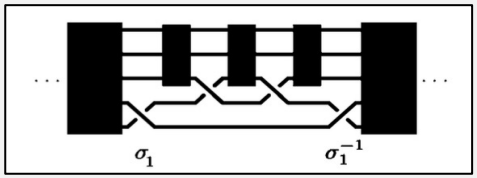
- Une poignée :

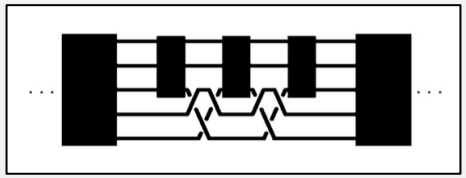


- Une solution globale ?



- Une solution locale





- Complexité théorique très élevée mais très rapide en pratique (en ms pour n=1000)

Figures extraites de [9]

SHA-256

$$A_{-3} = h_3, \quad A_2 = h_2, \quad A_{-1} = h_1, \quad A_0 = h_0$$

$$B_{-3} = h_7, \quad B_{-2} = h_6, \quad B_{-1} = h_5, \quad B_0 = h_4$$

Durant chaque étape i, les registres cibles A_{i+1} et B_{i+1} sont mis à jour par les fonctions f et g respectivement :

$$A_{i+1} = f(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i)$$

$$B_{i+1} = g(A_i, A_{i-1}, A_{i-2}, A_{i-3}, B_i, B_{i-1}, B_{i-2}, B_{i-3}, W_i, K_i)$$

Où les K_i sont des constantes prédéfinies pour chaque étape

$$W_i = \{(M_i, pour 0 \leq i \leq 15, \sigma^1(W_{i-2}) + W_{i-7} + \sigma^0(W_{i-15}) + W_{i-16}, pour 16 \leq i \leq 63) \}$$

Avec :

$$\sigma^0(x) = (x^{>>>7}) \oplus (x^{>>>18}) \oplus (x^{>>>3})$$

$$\sigma^1(x) = (x^{>>>17}) \oplus (x^{>>>19}) \oplus (x^{>>>10})$$

À la fin des 64 étapes, les mots de la sortie de la fonction de compression sont calculés par :

$$h'_0 = A_{64} + A_0, \quad h'_1 = A_{63} + A_{-1}, \quad h'_2 = A_{62} + A_{-2}, \quad h'_3 = A_{61} + A{-3}$$
$$h'_4 = B_{64} + B_0, \quad h'_5 = B_{63} + B_{-1}, \quad h_6 = B_{62} + B_{-2}, \quad h'_7 = B_{61} + B{-3}$$

Figure extraite de [4]

```python
import time
import numpy as np
import random
import hashlib as hashlib
import itertools

class Tresse:
    def __init__(self, sigma=None, eps=None, queue=None):
        '''
        Fonction pour créer une tresse
        '''

        self.gen = sigma
        self.puiss = eps
        self.tail = queue
        if queue == None:
            if sigma == None:
                self.nb_brins = 1
            else:
                self.nb_brins = sigma + 1
        else:
            self.nb_brins = max(queue.nb_brins, sigma + 1)

    def __len__(self):
        '''
        Fonction de longueur d'une Tresse
        '''
        if self.est_triv():
            return 0
        else:
            return 1 + len(self.tail)

    def representation(self, deb=True):
        if deb:
            s = '<'
        else:
            s = ''
        if self.est_triv():
            return s + '>'
        elif self.tail.est_triv():
            return s + '(' + repr(self.gen) + ',' + repr(self.puiss) + ')>'
        else:
            return s + '(' + repr(self.gen) + ',' + repr(
                self.puiss) + '),' + self.tail.representation(False)

    def __repr__(self):
        return self.representation(True)

    def est_triv(self):
        '''
        t.est_triv() renvoie si la tresse est la tresse triviale
        '''
        return (self == None or (self.puiss == None and self.tail == None and self.gen == None))

    def __mul__(self, t2):
        if self.est_triv():
            return t2
        else:
            return Tresse(self.gen, self.puiss, self.tail * t2)

    def __getitem__(self,i):
      if self.est_triv():
        raise IndexError
      if i==0:
        return (Tresse(),Tresse(self.gen,self.puiss,Tresse()),self.tail)
      else:
        a,b,c = self.tail[i-1]
        return mult(self.gen,self.puiss,a),b,c

    def __pow__(self,n):
      if n == 0:
        return Tresse()
      elif n<0:
        return inverse(self**(-n))
      elif n%2 == 0:
        return (self * self)**(n//2)
      else:
        return  self * (self**(n-1))
```

```
0079|
0080|
0081| def mult(gen, puiss, t):
0082|     '''
0083|   Retourne la tresse t concaténée avec le générateur sigma_gen à la puissance puiss
0084|   '''
0085|     return Tresse(gen, puiss, t)
0086|
0087|
0088| def prod(t1, t2):
0089|     '''
0090|   Renvoie le produit t1.t2
0091|   '''
0092|     if t1.est_triv():
0093|         return t2
0094|     else:
0095|         return mult(t1.gen, t1.puiss, prod(t1.tail, t2))
0096|
0097|
0098| def inverse(t):
0099|     """
0100|     Renvoie l'inverse de la tresse t
0101|     """
0102|     x = t
0103|     res = Tresse()
0104|     while not x.est_triv():
0105|         res = mult(x.gen, -x.puiss, res)
0106|         x = x.tail
0107|     return res
0108|
0109|
0110| def conversion(l):
0111|     '''
0112|   Convertit une liste de couples générateur,puissance en une tresse
0113|   '''
0114|     x = Tresse()
0115|     for (gen, puiss) in reversed(l):
0116|         if puiss > 0:
0117|           for i in range(puiss):
0118|             x = mult(gen, 1, x)
0119|         elif puiss < 0:
0120|           for i in range(-puiss):
0121|             x = mult(gen,-1,x)
0122|     return x
0123|
0124| ##
0125|
0126|
0127|
0128| i =1
0129| f = 50
0130| L=50
0131|
0132| def random_tresse(inf=i,sup=f,l=L):
0133|   t = Tresse()
0134|   for i in range(l):
0135|     if random.choice([True,False]):
0136|       eps = 1
0137|     else:
0138|       eps = -1
0139|     t = mult(random.randint(inf,sup),eps,t)
0140|   return t
0141|
0142| def random_tresse_brins(brins,l):
0143|   '''
0144|   Retourne une tresse aléatoire dont on choisit le nombre de brins'''
0145|   t=Tresse()
0146|   n=random.randint(1,l)
0147|   for i in range(n):
0148|     if random.choice([True,False]):
0149|       eps = 1
0150|     else:
0151|       eps = -1
0152|     t = mult(random.randint(1,brins),eps,t)
0153|   if random.choice([True,False]):
0154|     eps = 1
0155|   else:
0156|     eps = -1
```

```
  t=mult(brins,eps,t)
  for i in range(n+1,l):
    if random.choice([True,False]):
      eps = 1
    else:
      eps = -1
    t = mult(random.randint(1,brins),eps,t)
  return t

def random_tresse_brins_max(brins,m):
  '''
  Retourne une tresse aléatoire dont on choisit le nombre de brins et un majorant du nombre de gen'''
  l=random.randint(1,m)
  return random_tresse_brins(brins,l)

##

def val_abs(x):
    if x> 0:
        return x
    else:
        return -x


def simplifd(t):
    if t.est_triv() or t.tail.est_triv():
        return (t, True)
    i = t.gen
    j = t.tail.gen
    eps1 = t.puiss
    eps2 = t.tail.puiss
    if eps1 == -1 and eps2 == 1:
        if val_abs(i - j) == 0:
            return (t.tail.tail, False)
        elif val_abs(i - j) == 1:
            x = t.tail.tail
            x = mult(i, -1, x)
            x = mult(j, -1, x)
            x = mult(i, 1, x)
            x = mult(j, 1, x)
            return (x, False)
        else:
            x = t.tail.tail
            x = mult(i, -1, x)
            x = mult(j, 1, x)
            return (x, False)
    else:
        (x, b) = simplifd(t.tail)
        return (mult(i, eps1, x), b)


def retournementd(t):
    res, b = simplifd(t)
    while not b:
        res, b = simplifd(res)
    return res


def simplifg(t):
    if t.est_triv() or t.tail.est_triv():
        return (t, True)
    i = t.gen
    j = t.tail.gen
    eps1 = t.puiss
    eps2 = t.tail.puiss
    if eps1 == 1 and eps2 == -1:
        if val_abs(i - j) == 0:
            return (t.tail.tail, False)
        elif val_abs(i - j) == 1:
            x = t.tail.tail
            x = mult(i, 1, x)
            x = mult(j, 1, x)
            x = mult(i, -1, x)
            x = mult(j, -1, x)
            return (x, False)
        else:
            x = t.tail.tail
            x = mult(i, 1, x)
```

```
            x = mult(j, -1, x)
            return (x, False)
    else:
        (x, b) = simplifg(t.tail)
        return (mult(i, eps1, x), b)


def retournementg(t):
    res, b = simplifg(t)
    while not b:
        res, b = simplifg(res)
    return res


def separationd(t):
    '''
  Renvoie les parties positives et négatives d'un mot retourné
  '''
    if t.est_triv() or t.puiss == -1:
        return (Tresse(), t)
    else:
        (D, N) = separationd(t.tail)
        return (mult(t.gen, t.puiss, D), N)


def separationg(t):
    if t.est_triv() or t.puiss == 1:
        return (Tresse(), t)
    else:
        (N, D) = separationg(t.tail)
        return (mult(t.gen, t.puiss, N), D)


def parties_DN_d(t):
    return separationd(retournementd(t))


def parties_DN_g(t):
    return separationg(retournementg(t))
```

```


def retournementg_iter(t):
  deb,prem,c = t[0]
  while not c.est_triv():
     ,deuz,fin = c[0]
    if prem.puiss ==1 and  deuz.puiss==-1:
      i,j = prem.gen,deuz.gen
      if i == j:
        t = deb * fin
      elif val_abs(i-j) == 1:
        t = deb * conversion([(j,-1),(i,-1),(j,1),(i,1)]) *fin
      else:
        t = deb * conversion([(j,-1),(i,1)]) *fin
      deb,prem,c = t[0]
    else:
      deb = deb * prem
      prem = deuz
      c = fin
  return t

def retournementd_iter(t):
  deb,prem,c = t[0]
  while not c.est_triv():
     ,deuz,fin = c[0]
    if prem.puiss < deuz.puiss:
      i,j = prem.gen,deuz.gen
      if i == j:
        t = deb * fin
      elif val_abs(i-j) == 1:
        t = deb * conversion([(j,1),(i,1),(j,-1),(i,-1)]) *fin
      else:
        t = deb * conversion([(j,1),(i,-1)]) * fin
      deb,prem,c = t[0]
    else:
      deb = deb * prem
      prem = deuz
      c = fin
  return t
```

```
0313|
0314|
0315| ##
0316|
0317|
0318|
0319| def recherche_i_poignee(tresse, i):
0320|   a = Tresse()
0321|   b = Tresse()
0322|   c = tresse
0323|   def aux(at,bt,ct, isa, isb, eps1 = None, eps2 = None):
0324|     if len(ct) == 0: ##Si on est arrivé au bout de la tresse, il ne peut de toute façon pas y avoir de poignée,
puisqu'on se serait arrêté avant
0325|       return(at*bt, Tresse(), ct, i, None, None)
0326|
0327|     if isa:
0328|       if ct.gen != i:
0329|         return(aux(at*mult(ct.gen, ct.puiss, Tresse()),b,ct.tail, True, False, None,None))
0330|       else:
0331|         e1 = ct.puiss
0332|         return(aux(at, mult(i, e1, Tresse() ), ct.tail, False, True, e1, None))
0333|
0334|     if isb: # On a commencé une poignée, et on la parcourt
0335|       k = ct.gen
0336|       p = ct.puiss
0337|       if k < i:
0338|         return(aux(at*bt*mult(ct.gen, ct.puiss, Tresse()), Tresse(), ct.tail, True, False, None, None))
0339|       elif k == i :
0340|         if p == -eps1:
0341|           return(at, bt*mult(k,p, Tresse()), ct.tail, eps1, eps2)
0342|         else:
0343|           return(aux(at*bt, mult(ct.gen, ct.puiss, Tresse()), ct.tail, False, True, eps1, None))
0344|       elif (k == i + 1):
0345|         if (eps2 == None) or (p == eps2) :
0346|           return(aux(at, bt*mult(k, p, Tresse()), ct.tail, False, True, eps1, p))
0347|         else:
0348|           return(aux(at*bt*mult(k, p, Tresse()), Tresse(), ct.tail, True, False,None, None))
0349|       else:
0350|         return(aux(at, bt*mult(k, p, Tresse()), ct.tail, False, True, eps1, eps2))
0351|   r = aux(a,b,c, True, False, None, None)
0352|   ar=r[0]
0353|   br=r[1]
0354|   cr=r[2]
0355|   e1=r[3]
0356|   e2=r[4]
0357|   return(ar, br, cr, i, e1, e2)
0358|
0359|
0360| test = conversion([(7,1),(4,-1),(4,-1),(5,-1),(2,-1),(8,1)])
0361|
0362|
0363| def changement(b,i,eps1,eps2):
0364|   (b1,b2) = (Tresse(), b.tail)
0365|   if b2 == None:
0366|     l = 0
0367|   else:
0368|     l = len(b2)
0369|   for j in range(1,l):
0370|     k = b2.gen
0371|     p = b2.puiss
0372|     if k != i+1:
0373|       (b1,b2) = (b1*mult(k,p,Tresse()),b2.tail)
0374|     else:
0375|       (b1,b2) = (b1*mult(i+1,-eps1,Tresse())*mult(i,eps2,Tresse())*mult(i+1,eps1,Tresse()),b2.tail)
0376|   return(b1)
0377|
0378|
0379| def remplacement(tresse, i):
0380|   a,b,c,i,eps1,eps2 = recherche_i_poignee(tresse,i)
0381|
0382|   bn=changement(b,i,eps1,eps2)
0383|   if eps1 == None or len(b)==0:
0384|     return(tresse,None)
0385|   return(a*bn*c,i)
0386|
0387|
0388|
0389| def reduction_Patrick_Dehornoy(tresse):
```

Classe « rationnels »

```
0390|   n = tresse.nb_brins
0391|   k = 0
0392|   fini=[False]*(n-1)
0393|   while fini!=[True]*(n-1):
0394|     for i in range(1, n):
0395|
0396|       passage = False
0397|       while passage == False:
0398|         tresse,fini_pour_i = remplacement(tresse,i)
0399|         if fini_pour_i == None:
0400|           fini[i-1] = True
0401|           passage = True
0402|         else:
0403|           fini=[False]*(n-1)
0404|   return(tresse)
0405|
0406| import random as rd
0407| def crée_test(n,l):
0408|   tresse=Tresse()
0409|   for i in range (1,l):
0410|     g=rd.randint(1,n)
0411|     p=(-1)**rd.randint(1,2)
0412|     tresse=tresse*mult(g,p,Tresse())
0413|   return(tresse)
0414|
0415|
0416| def eq_rapide(t1,t2):
0417|   return reduction_Patrick_Dehornoy(inverse(t1)*t2).est_triv()
0418|
0419| ##
0420|
0421|
0422|
0423|
0424| class r:
0425|     def __init__(self, a, b):
0426|       if a == 0:
0427|         self.num = 0
0428|         self.den = 1
```

```
0429|       else:
0430|         g = pgcd_n(a,b)
0431|         self.num = a//g
0432|         self.den = b//g
0433|
0434|     def __add__(self, q):
0435|         a = self.num
0436|         b = self.den
0437|         c = q.num
0438|         d = q.den
0439|         return r(a * d + b * c, b * d)
0440|
0441|     def __mul__(self, q):
0442|         a = self.num
0443|         b = self.den
0444|         c = q.num
0445|         d = q.den
0446|         return r(a * c, b * d)
0447|
0448|     def __repr__(self):
0449|         a = self.num
0450|         b = self.den
0451|         return str(a) + '/' + str(b)
0452|
0453| def pgcd_n(a,b):
0454|   if b ==0:
0455|     return a
0456|   else:
0457|     return pgcd_n(b, a%b)
0458|
0459| def inv(x):
0460|   a=x.num
0461|   b=x.den
0462|   return r(b,a)
0463|
0464| def egal(x,y):
0465|   a = x.num
0466|   b = x.den
0467|   c = y.num
```

Hachage

```
0468|    d = y.den
0469|    return (a,b)==(c,d)
0470|
0471| ##
0472|
0473|
0474|
0475|
0476| def ajout(m,x):
0477|    if x==0:
0478|      m.update(b'0')
0479|    elif x==1:
0480|      m.update(b'1')
0481|    elif x==2:
0482|      m.update(b'2')
0483|    elif x==3:
0484|      m.update(b'3')
0485|    elif x==4:
0486|      m.update(b'4')
0487|    elif x==5:
0488|      m.update(b'5')
0489|    elif x==6:
0490|      m.update(b'6')
0491|    elif x==7:
0492|      m.update(b'7')
0493|    elif x==8:
0494|      m.update(b'8')
0495|    elif x==9:
0496|      m.update(b'9')
0497|
0498| def binaire(x):
0499|    y=int(x)
0500|    if y==1:
0501|      return '1'
0502|    if y==0:
0503|      return '0'
0504|    a=y%2
0505|    b=str(a)
0506|    return binaire(y//2)+b
```

```
0507|
0508| def hextobin2(l):
0509|    return binaire(int(l,16))
0510|
0511| def hextobin(l):
0512|    res=''
0513|    for x in l:
0514|      y=str(x)
0515|      if int(y,16)==0:
0516|        res=res+'0000'
0517|      elif int(y,16)==1:
0518|        res=res+'0001'
0519|      elif int(y,16)==2:
0520|        res=res+'0010'
0521|      elif int(y,16)==3:
0522|        res=res+'0011'
0523|      elif int(y,16)==4:
0524|        res=res+'0100'
0525|      elif int(y,16)==5:
0526|        res=res+'0101'
0527|      elif int(y,16)==6:
0528|        res=res+'0110'
0529|      elif int(y,16)==7:
0530|        res=res+'0111'
0531|      elif int(y,16)==8:
0532|        res=res+'1000'
0533|      elif int(y,16)==9:
0534|        res=res+'1001'
0535|      elif int(y,16)==10:
0536|        res=res+'1010'
0537|      elif int(y,16)==11:
0538|        res=res+'1011'
0539|      elif int(y,16)==12:
0540|        res=res+'1100'
0541|      elif int(y,16)==13:
0542|        res=res+'1101'
0543|      elif int(y,16)==14:
0544|        res=res+'1110'
0545|      elif int(y,16)==15:
```

```
0546|        res=res+'1111'
0547|    return res
0548|
0549| def hachage(A):
0550|    m=hashlib.sha256()
0551|    n,p=np.shape(A)
0552|    for i in range(n):
0553|      for j in range(n):
0554|        x,y=A[i,j].num,A[i,j].den
0555|        ajout(m,x)
0556|        ajout(m,y)
0557|    return hextobin(m.hexdigest())
0558|
0559|
0560| def messtobin(l):
0561|    res=''
0562|    for x in l:
0563|      res=res+lettrebin(str(x))
0564|    if len(res)%256!=0:
0565|      a=(256-len(res)%256)*'0'
0566|      res=a+res
0567|    return res
0568|
0569| def lettrebin(x):
0570|    y=ord(x)
0571|    res=binaire(y)
0572|    res=(7-len(res))*'0'+res
0573|    return res
0574|
0575| def sommebin(message,clef):
0576|    k=len(message)//256
0577|    clef2=k*clef
0578|    res=''
0579|    for i in range(k*256):
0580|      if int(message[i])+int(clef2[i])==0 or int(message[i])+int(clef2[i])==2:
0581|        res = res+'0'
0582|      elif int(message[i])+int(clef2[i])==1:
0583|        res = res+'1'
0584|    return res
```

```
0585|
0586|
0587| def bintomess(l):
0588|   res = ''
0589|   for i in range(len(l)%7,len(l),7):
0590|     res = res + chr(int(str(l[i:i+7]),2))
0591|   return res
0592|
0593| ##
0594|
0595|
0596|
0597|
0598| def demi_tour(n):
0599|     if n == 1:
0600|         return Tresse()
0601|     else:
0602|         x = Tresse()
0603|         for i in range(n - 1, 0, -1):
0604|             x = mult(i, 1, x)
0605|         return prod(x, demi_tour(n - 1))
0606|
0607|
0608| def pgcd(a, b):
0609|     x = prod(inverse(a), b)
0610|     x = retournementd(x)
0611|     (e, d) = parties_DN_g(x)
0612|     return retournementg(a * e)
0613|
0614|
0615|
0616|
0617| def perm(t, n):
0618|     '''
0619|   Renvoie le tableau associé à la permutation de S_n
0620|   définie par t, où n est un majorant du nombre de brins de t
0621|   '''
0622|     res = [i for i in range(1, n + 1)]
0623|     s = t
0624|     while not s.est_triv():
0625|         i = s.gen
0626|         res[i - 1], res[i] = res[i], res[i - 1]
0627|         s = s.tail
0628|     return res
0629|
0630|
0631| def eg_perm(t, s):
0632|     '''
0633|   Renvoie si les permutations associées à t et s sont égales
0634|   '''
0635|     n = max(t.nb_brins, s.nb_brins)
0636|     return perm(t, n) == perm(s, n)
0637|
0638|
0639| def forme_normale_positive(t, n):
0640|     '''
0641|   Renvoie la forme normale d'une tresse positive dans B_n
0642|   '''
0643|     s = t
0644|     r = 0
0645|     l = []
0646|     idn = [i for i in range(1, n + 1)]
0647|     deltan = demi_tour(n)
0648|     deltan_inv = inverse(deltan)
0649|     ret = perm(deltan, n)
0650|     g = pgcd(deltan, t)
0651|     while perm(g, n) == ret:
0652|         r += 1
0653|         s = deltan_inv * s
0654|         g = pgcd(s, deltan)
0655|     while not perm(g, n) == idn:
0656|         l.append(retournementd(g))
0657|         s = inverse(g) * s
0658|         g = pgcd(deltan, s)
0659|     return (r, l)
0660|
0661|
0662| def sym(t, n):
```

```
0663|     if t.est_triv():
0664|         return t
0665|     else:
0666|         return Tresse(n - t.gen, t.puiss, sym(t.tail, n))
0667|
0668|
0669| def positivation(t, n):
0670|     if t.est_triv():
0671|         return (0, t)
0672|     else:
0673|         deltan = demi_tour(n)
0674|         sym_deltan = sym(deltan, n)
0675|         i = t.gen
0676|         eps = t.puiss
0677|         if eps == 1:
0678|             nb, pos = positivation(t.tail, n)
0679|             if nb % 2 == 0:
0680|                 return (nb, Tresse(i, eps, pos))
0681|             else:
0682|                 return (nb, Tresse(n - i, eps, pos))
0683|
0684|         if eps == -1:
0685|             nb, pos = positivation(t.tail, n)
0686|             gen_i = Tresse(i, eps, Tresse())
0687|             sym_i = Tresse(n - i, eps, Tresse())
0688|             if nb % 2 == 0:
0689|                 return (nb + 1, deltan * gen_i * pos)
0690|             else:
0691|                 return (nb + 1, sym_deltan * sym_i * pos)
0692|
0693|
0694| def forme_normale(t, n=-1):
0695|     if n == -1:
0696|         n = t.nb_brins
0697|     r, pos_t = positivation(t, n)
0698|     s, l = forme_normale_positive(pos_t, n)
0699|     return (s - r, l)
0700|
0701|
```

```
0702| def eq(t1, t2):
0703|     n = max(t1.nb_brins, t2.nb_brins)
0704|     r, l1 = forme_normale(t1, n)
0705|     s, l2 = forme_normale(t2, n)
0706|     n, m = len(l1), len(l2)
0707|     if r != s or n != m:
0708|         return False
0709|     else:
0710|         for i in range(n):
0711|             if not eg_perm(l1[i], l2[i]):
0712|                 return False
0713|         return True
0714|
0715|
0716| ##
0717|
0718|
0719|
0720| def conj(x,a):
0721|   return inverse(a) *x * a
0722|
0723| def tau(t,r,n=-1):
0724|   if n==-1:
0725|     n=t.nb_brins
0726|   deltan = demi_tour(n)
0727|   if r==0:
0728|     return t
0729|   elif r > 0:
0730|     return inverse(deltan) * tau(t,r-1,n) * deltan
0731|   else :
0732|     return deltan * tau(t,r+1,n) * inverse(deltan)
0733|
0734| def cyclage(t,n=-1):
0735|   if n==-1:
0736|     n = t.nb_brins
0737|   r,l = forme_normale(t,n)
0738|   if l == []:
0739|     return t
0740|   else:
```

```
    return tau(inverse(l[0]),r,n) * t * tau(l[0],-r,n)

def decyclage(t,n=-1):
  if n ==-1:
    n = t.nb_brins
  r,l = forme_normale(t,n)
  if l == []:
    return t
  else:
    return l[-1]*t * inverse(l[-1])

def long(t,n=-1):
  if n==-1:
    n = t.nb_brins
  r,l = forme_normale(t,n)
  p = len(l)
  return (r,p+r,p)

def sup(t,n=-1):
  return long(t,n)[1]

def inf(t,n=-1):
  return long(t,n)[0]

def abs(t,n=-1):
  return long(t,n)[2]

def index(perm,x):
  n = len(perm)
  for i in range(n):
    if perm[i]==x:
      return i

def simple(perm):
  n=len(perm)
  for i in range(1,n+1):
    k=index(perm,i)
    if k!=i-1:
      perm[k],perm[k-1] = perm[k-1],perm[k]
      return Tresse(k,1,Tresse()) * simple(perm)
  return Tresse()

def ens_perm(n):
  l = [i for i in range(1,n+1)]
  res = list(itertools.permutations(l))
  for i in range(len(res)):
    res[i]=list(res[i])
  return res

def ens_tresses_simples(n):
  l = ens_perm(n)
  res = []
  for perm in l:
    res.append(simple(perm))
  return res

def el_SSS(a,n=-1,k = 0,cycle = True,r=None,p=None,l=None):
  dessiner(a)
  if n==-1:
    n = a.nb_brins
  if r == None:
    r,p,l = long(a,n)
  N = n*(n-1)//2
  if cycle:
    if k == N:
      return el_SSS(a,n,0,False,r,p,l)

    else:
      c = cyclage(a)
      inf2,liste = forme_normale(c,n)
      l2 = len(liste)
      sup2 = l2 + inf2
      if -inf2 > -r:
        el,conj = el_SSS(retournementg(retournementd(c)),n,0,True,inf2,sup2,l2)
        if liste!=[]:
          return (el,conj * tau(inverse(liste[0]),r,n))
        else:
          return (el,conj)
```

```
      else:
        el,conj = el_SSS(retournementg(retournementd(c)),n,k+1,True,inf2,sup2,l2)
        if liste!=[]:
          return (el,conj * tau(inverse(liste[0]),r,n))
        else:
          return (el,conj)
  else:
    if k == N:
      return (a,Tresse())
    else:
      d = decyclage(a,n)
      inf2,liste = forme_normale(d,n)
      l2 = len(liste)
      sup2 = l2 + inf2
      if sup2 < p:
        el,conj = el_SSS(retournementg(retournementd(d)),n,0,True,inf2,sup2,l2)
        if liste != []:
          return (el,conj * liste[-1])
        else:
          return (el,conj)
      else:
        el,conj = el_SSS(retournementg(retournementd(d)),n,k+1,False,inf2,sup2,l2)
        if liste != []:
          return (el,conj * liste[-1])
        else:
          return (el,conj)

def est_dans_SSS(ens,t,n):
  for tresse in ens:
    if eq_rapide(t,tresse):
      return True
  return False

def construction_SSS(ens,ens_conj,i,l,n,tresses_simples):
  x = ens[i]
  for tresse in tresses_simples:
    el = inverse(tresse) * x * tresse
    if abs(el,n) == l and not est_dans_SSS(ens,el,n):
      ens.append(el)
      ens_conj.append(ens_conj[i]*tresse)
  if i < len(ens) - 1:
    return construction_SSS(ens,ens_conj,i+1,l,n,tresses_simples)
  else:
    return ens,ens_conj


def SSS(a,n=-1):
  if n == -1:
    n = a.nb_brins
  simp = ens_tresses_simples(n)
  res = []
  b,conj = el_SSS(a,n)
  b = retournementd(retournementg(b))
  res.append(b)
  l = abs(b,n)
  l_conj = [conj]
  return construction_SSS(res,l_conj,0,l,n,simp)

def conjugues(a,b,n=-1):
  if n == -1:
    n = max(a.nb_brins,b.nb_brins)
  l = SSS(a,n)[0]
  c = el_SSS(b,n)[0]
  return est_dans_SSS(l,c,n)

def conjugant(a,b,n=-1):
  """
  renvoie x un tresse t telle que
  a = t^-1 * b * t si elle existe et None sinon

  """


  if n== -1:
    n = max(a.nb_brins,b.nb_brins)
  super_set,l_conj = SSS(a,n)
  x,conj = el_SSS(b,n)
  for i in range(len(super_set)):
```

Représentation de Burau

```
0897|     if eq_rapide(super_set[i],x):
0898|       return inverse(conj) * l_conj[i]
0899|
0900| def recherche_conj(y,ens,ens_conj,i,l,n,tresses_simples):
0901|   x = ens[i]
0902|   for tresse in tresses_simples:
0903|     el = inverse(tresse) * x * tresse
0904|     if eq_rapide(el,y):
0905|       return ens_conj[i] * tresse
0906|     if abs(el,n) == l and not est_dans_SSS(ens,el,n):
0907|       ens.append(el)
0908|       ens_conj.append(ens_conj[i]*tresse)
0909|   if i<len(ens)-1:
0910|     return recherche_conj(y,ens,ens_conj,i+1,l,n,tresses_simples)
0911|
0912| def conjugant_rapide(a,b,n=-1):
0913|   if n ==-1:
0914|     n = max(a.nb_brins,b.nb_brins)
0915|   x,conj_a = el_SSS(a,n)
0916|   y,conj_b = el_SSS(b,n)
0917|   ens = [x]
0918|   ens_conj = [conj_a]
0919|   l = abs(x,n)
0920|   tresses_simples = ens_tresses_simples(n)
0921|   conj =  recherche_conj(y,ens,ens_conj,0,l,n,tresses_simples)
0922|   if conj == None:
0923|     return None
0924|   return conj * inverse(conj_b)
0925|
0926| ##
0927|
0928|
0929| t = 10
0930|
0931|
0932| def I(n):
0933|   res = []
0934|   for i in range(n):
0935|     res.append([r(0,1)]* n)
```

```
0936|   A = np.array(res)
0937|   for i in range (n):
0938|     A[i,i] = r(1,1)
0939|   return A
0940|
0941| def puiss_mat(A, k):
0942|     (n, p) = np.shape(A)
0943|     if k >= 0:
0944|         if k == 0:
0945|             return I(n)
0946|         elif k % 2 == 0:
0947|             return np.dot(puiss_mat(A, k // 2), puiss_mat(A, k // 2))
0948|         elif k % 2 == 1:
0949|             return np.dot(puiss_mat(A, k - 1), A)
0950|     else:
0951|         return np.linalg.inv(puiss_mat(A, -k))
0952|
0953|
0954| def burau_gen(sigma, puiss, n):
0955|     i, j = sigma, puiss
0956|     A = I(n)
0957|     if puiss == 1:
0958|         A[i - 1, i - 1] = r(1 - t, 1)
0959|         A[i - 1, i] = r(1, 1)
0960|         A[i, i - 1] = r(t, 1)
0961|         A[i, i] = r(0, 1)
0962|         return A
0963|     if puiss == -1:
0964|         A[i - 1, i - 1] = r(0, 1)
0965|         A[i - 1, i] = r(1, t)
0966|         A[i, i - 1] = r(1, 1)
0967|         A[i, i] = r(t - 1, t)
0968|         return A
0969|
0970|
0971| def burau(psi):
0972|     n = psi.nb_brins
0973|     A = I(n)
0974|     while not psi.est_triv():
```

```
        A = np.dot(A, burau_gen(psi.gen, psi.puiss, n))
        psi = psi.tail
    return A


def burau_rec(psi, n=-1):
    if n == -1:
        n = psi.nb_brins
    if psi.est_triv():
        return I(n)
    else:
        return np.dot(burau_gen(psi.gen, psi.puiss, n), burau_rec(psi.tail, n))


def egal_burau(A,B):
  assert np.shape(A)==np.shape(B)
  n,p=np.shape(A)
  for i in range(n):
    for j in range(n):
      if not(egal(A[i,j],B[i,j])):
        return False
  return True


def test_burau(brins,l,N):
  egalburau=0
  egaltresse=0
  for i in range(N):
    a=random_tresse_brins(brins,l)
    b=random_tresse_brins(brins,l)
    A=burau(a)
    B=burau(b)
    if egal_burau(A,B):
      if eq_rapide(a,b):
        egaltresse+=1
      egalburau+=1
  return (N,egalburau,egaltresse)

def test_burau2(brins,l,N):
  '''
  On fixe la 1ere tresse'''
  egalburau=0
  egaltresse=0
  a=random_tresse_brins(brins,l)
  A=burau(a)
  for i in range(N):
    b=random_tresse_brins(brins,l)
    B=burau(b)
    if egal_burau(A,B):
      if eq_rapide(a,b):
        egaltresse+=1
      egalburau+=1
  return (N,egalburau,egaltresse)

def test_burau3(brins,l,essai_max):
  debut=time.time()
  egaltresse=0
  egalburau=0
  essai=0
  a=random_tresse_brins_max(brins,l)
  A=burau(a)
  clef=True
  while clef and essai<essai_max:
    b=random_tresse_brins_max(brins,l)
    B=burau(b)
    essai+=1
    if egal_burau(A,B):
      if eq_rapide(a,b):
        egaltresse+=1
      else:
        clef=False
      egalburau+=1
  fin=time.time()
  duree=fin-debut
  return (essai,egalburau,egaltresse,duree)


##
```

```
1053|
1054| def test_conj(nb_brins,longueur,nb_essais=1):
1055|     l=[]
1056|     for i in range(nb_essais):
1057|         x = random_tresse(1,nb_brins,longueur)
1058|         a = random_tresse(1,nb_brins,longueur)
1059|         y = conj(x,a)
1060|         while eq_rapide(x,Tresse()):
1061|           x = random_tresse(1,nb_brins,longueur)
1062|           a = random_tresse(1,nb_brins,longueur)
1063|           y = conj(x,a)
1064|         debut= time.time()
1065|         conjugant_rapide(x,y)
1066|         fin = time.time()
1067|         l.append(fin-debut)
1068|     res = 0
1069|     for el in l:
1070|         res += el
1071|     return res/len(l)
1072|
1073|
1074| def test_Burau(nb_brins,longueur,nb_essais=1000):
1075|   l=[]
1076|   for _ in range(nb_essais):
1077|     x = random_tresse(1,nb_brins,longueur)
1078|     debut = time.time()
1079|     burau(x)
1080|     fin = time.time()
1081|     l.append(fin-debut)
1082|   res = 0
1083|   for el in l:
1084|     res += el
1085|   return res/(len(l))
1086|
1087|
1088|
```