

Permutations

April 1, 2023

1 Permutations

Marc Lorenzi

31 mars 2023

```
[1]: import random
import math
import matplotlib.pyplot as plt
```

1.1 1. Introduction

1.1.1 1.1 Permutations et Python

Dans tout ce notebook, n désigne un entier naturel. On note $\mathfrak{S}_n = \mathfrak{S}([0, n-1])$ l'ensemble des permutations de l'ensemble $[0, n-1]$.

On représente en Python une permutation $\sigma \in \mathfrak{S}_n$ par la liste $[\sigma(0), \dots, \sigma(n-1)]$.

La fonction `print_perm` permet d'afficher de façon agréable une permutation s .

```
[2]: def print_perm(s):
    n = len(s)
    ln = n * '+---' + '+'
    print(ln)
    for k in range(n):
        print('|%2d ' % k, end='')
    print('|')
    print(ln)
    for k in range(n):
        print('|%2d ' % s[k], end='')
    print('|')
    print(ln)
```

```
[3]: print_perm([7, 1, 2, 4, 3, 6, 5, 8, 10, 9, 0])
```

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

```

+---+---+---+---+---+---+---+---+---+---+
| 7 | 1 | 2 | 4 | 3 | 6 | 5 | 8 |10 | 9 | 0 |
+---+---+---+---+---+---+---+---+---+---+

```

1.1.2 1.2 Permutations aléatoires

Il sera bien utile d'avoir pour nos tests une fonction qui renvoie une permutation « aléatoire ». La fonction `random_perm` prend en paramètre un entier n et renvoie une permutation de $\mathfrak{S}_n$. L'algorithme utilisé est *l'algorithme de Fisher-Yates*.

Munissons $\mathfrak{S}_n$ de la probabilité uniforme $\mathbb{P}$. La fonction `random_perm` peut être vue comme une variable aléatoire X à valeurs dans $\mathfrak{S}_n$. On peut montrer (nous ne le ferons pas ici) que pour toute permutation $\sigma \in \mathfrak{S}_n$, on a $\mathbb{P}(X = \sigma) = \frac{1}{n!}$. Comme le cardinal de $\mathfrak{S}_n$ est $|\mathfrak{S}_n| = \frac{1}{n!}$, la variable aléatoire X suit une loi uniforme.

```

[4]: def random_perm(n):
      s = list(range(n))
      for i in range(n - 1, -1, -1):
          j = random.randint(i, n - 1)
          s[i], s[j] = s[j], s[i]
      return s

```

```

[5]: s = random_perm(15)
      print_perm(s)

```

```

+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+
| 2 | 7 | 0 | 8 |11 | 9 | 1 | 5 |13 | 6 |10 | 4 | 3 |14 |12 |
+---+---+---+---+---+---+---+---+---+---+

```

1.2 2. Opérations sur les permutations

$(\mathfrak{S}_n, \circ)$ est un groupe pour la composition des applications.

1.2.1 2.1 Élément neutre

La fonction `id` renvoie la permutation identité, neutre de $\mathfrak{S}_n$.

```

[6]: def id(n):
      return list(range(n))

```

```

[7]: print_perm(id(10))

```

```

+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+

```

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
```

1.2.2 2.2 Produit

La fonction `produit` renvoie la composée des deux permutations s_1 et s_2 , supposées de même taille.

```
[8]: def produit(s1, s2):
      n = len(s1)
      return [s1[s2[k]] for k in range(n)]
```

```
[9]: print_perm(produit([1, 4, 0, 3, 2], [4, 0, 1, 3, 2]))
```

```
+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+
| 2 | 1 | 4 | 3 | 0 |
+---+---+---+---+---+
```

```
[10]: s1 = random_perm(10)
      s2 = random_perm(10)
      print_perm(s1)
      print_perm(s2)
      print_perm(produit(s1, s2))
      print_perm(produit(s2, s1))
```

```
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
| 1 | 4 | 3 | 7 | 5 | 0 | 6 | 2 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
| 5 | 9 | 4 | 6 | 1 | 7 | 2 | 8 | 0 | 3 |
+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
| 0 | 9 | 5 | 6 | 4 | 2 | 3 | 8 | 1 | 7 |
+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+---+---+---+---+---+---+---+---+---+---+
| 9 | 1 | 6 | 8 | 7 | 5 | 2 | 4 | 0 | 3 |
+---+---+---+---+---+---+---+---+---+---+
```

1.2.3 2.3 Inverse

La fonction `inverse` renvoie l'inverse de la permutation s .

```
[11]: def inverse(s):  
      n = len(s)  
      s1 = n * [None]  
      for k in range(n):  
          s1[s[k]] = k  
      return s1
```

```
[12]: print_perm(inverse([5,3,1,2,0,4]))
```

```
+---+---+---+---+---+---+  
| 0 | 1 | 2 | 3 | 4 | 5 |  
+---+---+---+---+---+---+  
| 4 | 2 | 3 | 1 | 5 | 0 |  
+---+---+---+---+---+---+
```

Le produit d'une permutation et de son inverse est égal à id . Faisons un petit test.

```
[13]: s1 = random_perm(15)  
      s2 = inverse(s1)  
      print_perm(s1)  
      print_perm(s2)  
      print(produit(s1, s2) == id(15), produit(s2, s1) == id(15))
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| 6 | 8 | 3 | 1 |12 | 9 |14 | 4 | 5 | 2 | 0 |11 |13 | 7 |10 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
|10 | 3 | 9 | 2 | 7 | 8 | 0 |13 | 1 | 5 |14 |11 | 4 |12 | 6 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
True True
```

1.2.4 2.4 Puissances

Comme dans tous les groupes, on dispose dans $\mathfrak{S}_n$ de la notion de puissance.

La fonction `puissance` prend en paramètre une permutation s et un entier relatif n . Elle renvoie s^n . Elle utilise l'algorithme d'exponentiation rapide.

```
[14]: def puissance(s, n):  
      if n < 0:
```

```

    s = inverse(s)
    n = -n
    z = list(range(len(s)))
    while n > 0:
        if n % 2 == 1:
            z = produit(z, s)
        s = produit(s, s)
        n = n // 2
    return z

```

```

[15]: s = random_perm(15)
      print_perm(puissance(s, 123456789101112))

```

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

```

[16]: s = random_perm(15)
      print_perm(puissance(s, -123456789101112))

```

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

```

[17]: s = random_perm(15)
      print_perm(produit(puissance(s, 123456789101112), puissance(s,
↪-123456789101112)))

```

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

1.3 3. Décomposition en produit de cycles de supports disjoints

Toute permutation se décompose en produit de cycles de supports disjoints. La décomposition est unique à l'ordre près des facteurs.

On représente en Python un cycle γ de longueur $k \geq 2$ par une liste $[a_0 \dots a_{k-1}]$, où $\gamma(a_0) = a_1$, $\gamma(a_1) = a_2$, $\dots$, $\gamma(a_{k-1}) = a_0$.

1.3.1 3.1 Orbites

La fonction `orbite` prend en paramètres une permutation s et un entier x . Elle renvoie sous forme de liste l'orbite de x suivant s , parcourue dans l'ordre. Le troisième paramètre, e , est optionnel. Si e est différent de `None`, c'est un ensemble auquel on retire au fur et à mesure les éléments de l'orbite de x .

```
[18]: def orbite(s, x, e=None):  
    c = [x]  
    y = s[x]  
    while y != x:  
        c.append(y)  
        if e != None: e.remove(y)  
        y = s[y]  
    return c
```

```
[53]: s = random_perm(15)  
c = orbite(s, 0)  
print_perm(s)  
print(c)
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| 9 | 4 |14 |12 | 0 |13 | 7 | 8 | 6 |10 |11 | 5 | 1 | 3 | 2 |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
[0, 9, 10, 11, 5, 13, 3, 12, 1, 4]
```

1.3.2 3.2 Décomposition en produit de cycles

La fonction `vers_cycles` prend en paramètre une permutation σ et renvoie une liste de cycles dont le produit est σ .

```
[20]: def vers_cycles(s):  
    n = len(s)  
    cs = []  
    e = set(range(n))  
    while len(e) != 0:  
        x = e.pop()  
        c = orbite(s, x, e)  
        if len(c) != 1: cs.append(c)  
    return cs
```

```
[52]: s = random_perm(15)  
cs = vers_cycles(s)  
print_perm(s)  
print(cs)
```

```

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|11 | 3 | 8 |12 | 2 |13 | 0 | 7 |14 | 1 | 9 | 5 | 6 |10 | 4 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
[[0, 11, 5, 13, 10, 9, 1, 3, 12, 6], [2, 8, 14, 4]]

```

1.3.3 3.3 Recomposition

La fonction `vers_permut` effectue l'opération inverse de `vers_cycles`. Elle prend en paramètre une liste de cycles (supposés de supports disjoints) et renvoie leur produit.

```
[22]: def vers_permut(cs, n):
      s = list(range(n))
      for c in cs:
          lc = len(c)
          for i in range(lc - 1):
              s[c[i]] = c[i + 1]
          s[c[lc - 1]] = c[0]
      return s

```

Vérifions que `vers_permut` est bien inverse à gauche de `vers_cycles`.

```
[23]: s1 = random_perm(1000)
      cs = vers_cycles(s1)
      s2 = vers_permut(cs, 1000)
      print(s1 == s2)

```

True

1.3.4 3.4 Nombre d'orbites

La fonction `nombre_orbites` prend en paramètre une permutation s . Elle renvoie un triplet (n_1, n_2, n_3) où

- n_1 est le nombre d'orbites de s non réduites à un point
- n_2 est le nombre de points fixes de s
- $n_3 = n_1 + n_2$ est le nombre total d'orbites.

```
[24]: def nombre_orbites(s):
      cs = vers_cycles(s)
      n1 = len(cs)
      n2 = len(s)
      for c in cs: n2 = n2 - len(c)
      return (n1, n2, n1 + n2)

```

```
[25]: s = random_perm(15)
print_perm(s)
print(vers_cycles(s))
print(nombre_orbites(s))
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 6 | 3 |14 |13 | 4 | 8 | 5 | 2 | 9 |12 |11 | 1 |10 | 0 | 7 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+
[[0, 6, 5, 8, 9, 12, 10, 11, 1, 3, 13], [2, 14, 7]]
(2, 1, 3)
```

1.4 4. Quelques statistiques

Nous allons dans cette section nous livrer à quelques statistiques concernant les orbites et les points fixes d'une permutation. Commençons par quelque chose de simple.

1.4.1 4.1 Points fixes

Combien une permutation possède-t-elle de points fixes ?

```
[26]: def nombre_points_fixes(s):
return nombre_orbites(s)[1]
```

La fonction `stats_points_fixes` renvoie une estimation du nombre « moyen » de points fixes d'une permutation de $\mathfrak{S}_n$.

```
[27]: def stats_points_fixes(n, N=1000):
a = 0.
for k in range(N):
s = random_perm(n)
a = a + nombre_points_fixes(s)
return a / N
```

```
[28]: stats_points_fixes(100)
```

```
[28]: 1.009
```

Proposition vague. Le nombre moyen de points fixes d'une permutation de $\mathfrak{S}_n$ est égal à 1.

Rendons tout d'abord la proposition moins vague. Munissons $\mathfrak{S}_n$ de la probabilité uniforme $\mathbb{P}$. Pour $i \in [0, n-1]$, soit $X_i : \mathfrak{S}_n \rightarrow \mathbb{R}$ définie par $X_i(\sigma) = 1$ si $\sigma(i) = i$ et $X_i(\sigma) = 0$ sinon. Posons enfin

$$X = \sum_{i=0}^{n-1} X_i$$

Pour toute permutation $\sigma \in \mathfrak{S}_n$, $X(\sigma)$ est le nombre de points fixes de σ .

Proposition. L'espérance de la variable aléatoire X est $E(X) = 1$.

Démonstration. Pour tout $i \in [0, n-1]$, X_i suit une loi de Bernoulli. Le paramètre de cette loi est

$$p_i = \mathbb{P}(X_i = 1)$$

Considérons l'événement $A_i = \{X_i = 1\}$. On a

$$A_i = \{\sigma \in \mathfrak{S}_n : \sigma(i) = i\}$$

On vérifie facilement que l'ensemble A_i est en bijection avec $\mathfrak{S}_{n-1}$. De là,

$$p_i = \mathbb{P}(A_i) = \frac{|A_i|}{n!} = \frac{(n-1)!}{n!} = \frac{1}{n}$$

Ainsi, $E(X_i) = \frac{1}{n}$. Il ne reste plus qu'à utiliser la linéarité de l'espérance.

$$E(X) = \sum_{i=0}^{n-1} E(X_i) = 1$$

1.4.2 4.2 Nombre d'orbites

Généralisons un peu. Soit $k \in [1, n]$. Combien une permutation $\sigma \in \mathfrak{S}_n$ possède-t-elle d'orbites de cardinal k ?

Notons $C_k(\sigma)$ le nombre d'orbites selon σ qui sont de cardinal k .

Proposition. On a

$$\sum_{\sigma \in \mathfrak{S}_n} k C_k(\sigma) = n!$$

Démonstration. Considérons l'ensemble

$$A = \{(a, \sigma) : \sigma \in \mathfrak{S}_n, a \text{ appartient à une orbite de cardinal } k \text{ suivant } \sigma\}$$

Soit $\sigma \in \mathfrak{S}_n$. La permutation σ a $C_k(\sigma)$ orbites de cardinal k , et dans chacune de ces orbites il y a k points. Ainsi,

$$|A| = \sum_{\sigma \in \mathfrak{S}_n} k C_k(\sigma)$$

Calculons le cardinal de A d'une autre façon. Soit $a \in [0, n-1]$. Pour créer une permutation pour laquelle a est dans une orbite de cardinal k :

- On choisit un ensemble $B \subseteq [0, n-1]$ de cardinal $k-1$ (l'orbite sera $B \cup \{a\}$) : $\binom{n-1}{k-1}$ possibilités.
- On choisit un cycle de longueur k dont le support est $B \cup \{a\}$: $(k-1)!$ possibilités.
- On choisit une permutation des $n-k$ entiers restants : $(n-k)!$ possibilités.

Le nombre de permutations pour lesquelles a est dans une orbite de cardinal k est donc

$$\binom{n-1}{k-1} (k-1)! (n-k)! = (n-1)!$$

De là,

$$|A| = \sum_{a=0}^{n-1} (n-1)! = n(n-1)! = n!$$

Proposition. Pour tout $k \in [1, n]$, $E(C_k) = \frac{1}{k}$.

Démonstration. On a

$$E(C_k) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} C_k(\sigma) = \frac{1}{k}$$

Pour $k=1$, nous retrouvons que l'espérance du nombre de points fixes d'une permutation est égal à 1.

Maintenant, quel est le nombre d'orbites d'une permutation $\sigma \in \mathfrak{S}_n$? Notons ce nombre $\Omega(\sigma)$. On a évidemment

$$\Omega(\sigma) = \sum_{k=1}^n C_k(\sigma)$$

De là, par la linéarité de l'espérance,

$$E(\Omega) = \sum_{k=1}^n E(C_k) = \sum_{k=1}^n \frac{1}{k} = H_n$$

où H_n est le n ième nombre harmonique.

```
[29]: def H(n):
      s = 0
      for k in range(1, n + 1):
          s += 1 / k
      return s
```

```
[30]: for n in range(1, 11):
      print(n, H(n))
```

```

1 1.0
2 1.5
3 1.8333333333333333
4 2.0833333333333333
5 2.2833333333333333
6 2.4499999999999997
7 2.5928571428571425
8 2.7178571428571425
9 2.8289682539682537
10 2.9289682539682538

```

La fonction `stats_orbites` renvoie une liste S de longueur $n + 1$. Pour $0 \leq k \leq n$, $S[k]$ est une estimation de $E(C_k)$. Bien entendu, $S[0] = 0$.

```

[31]: def stats_orbites(n, N=1000):
        S = (n + 1) * [0]
        for k in range(N):
            s = random_perm(n)
            cs = vers_cycles(s)
            for c in cs:
                l = len(c)
                S[l] = S[l] + 1
            S[1] += nombre_points_fixes(s)
        for k in range(n + 1):
            S[k] = S[k] / N
        return S

```

```

[32]: print(stats_orbites(20))

```

```

[0.0, 1.019, 0.485, 0.336, 0.252, 0.177, 0.157, 0.155, 0.152, 0.113, 0.096,
0.08, 0.083, 0.081, 0.073, 0.063, 0.065, 0.063, 0.054, 0.049, 0.049]

```

```

[33]: plt.rcParams['figure.figsize'] = (8, 3)

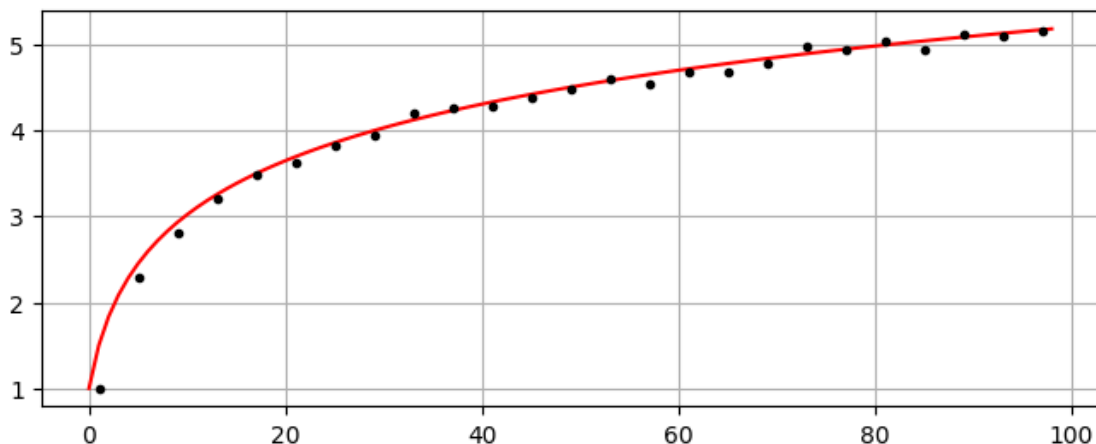
```

Ci-dessous, en noir le nombre « moyen » d'orbites en fonction de n . En rouge, la « courbe » de la suite $n \mapsto H_n$.

```

[34]: ks = range(1, 100, 4)
        s = [sum(stats_orbites(k, 1000)) for k in ks]
        lg = [H(k) for k in range(1, 100)]
        plt.plot(lg, 'r')
        plt.plot(ks, s, 'ok', ms=3)
        plt.grid()

```



1.5 5. Signature

La signature de la permutation $\sigma \in \mathfrak{S}_n$ est $\varepsilon(\sigma) = (-1)^{n-m}$ où m est le nombre d'orbites selon σ . La fonction $\varepsilon : \mathfrak{S}_n \rightarrow \{-1, 1\}$ est un morphisme de groupes, et elle est facile à calculer pour un cycle.

```
[35]: def signature_cycle(c):
      return (-1) ** (len(c) + 1)
```

La signature d'une permutation quelconque s'en déduit.

```
[36]: def signature(s):
      cs = vers_cycles(s)
      p = 1
      for c in cs:
          p = p * signature_cycle(c)
      return p
```

```
[37]: s = random_perm(20)
      print(s)
      print(signature(s))
```

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

La fonction `stats_signature` renvoie une estimation de l'espérance de la signature. Si $n \geq 2$, la valeur exacte de cette espérance est 0. En effet

$$E(\varepsilon) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \varepsilon(\sigma) = \frac{1}{n!} (|\mathfrak{A}_n| - |\mathfrak{S}_n \setminus \mathfrak{A}_n|)$$

où $\mathfrak{A}_n$ est l'ensemble des permutations paires. Or,

$$|\mathfrak{A}_n| = |\mathfrak{S}_n|/2 = \frac{1}{2}n!$$

On a donc $E(\varepsilon) = 0$.

```
[38]: def stats_signature(n):  
      N = 1000  
      av = 0.  
      for k in range(N):  
          s = random_perm(n)  
          av = av + signature(s)  
      return av / N
```

```
[39]: stats_signature(20)
```

```
[39]: -0.04
```

1.6 6. Ordre d'une permutation

L'ordre d'une permutation est le ppcm des ordres des cycles qui la composent. Or, l'ordre d'un cycle est sa longueur.

On commence par définir pgcd et ppcm.

```
[40]: def pgcd(a, b):  
      while b != 0: a, b = b, a % b  
      return a
```

```
[41]: pgcd(18, 28)
```

```
[41]: 2
```

```
[42]: def ppcm(a, b):  
      if a == 0 and b == 0: return 0  
      else:  
          d = pgcd(a, b)  
          return (a // d) * b
```

```
[43]: ppcm(4, 6)
```

```
[43]: 12
```

Maintenant il est facile de calculer l'ordre d'une permutation.

```
[44]: def ordre(s):  
      cs = vers_cycles(s)
```

```

p = 1
for c in cs:
    p = ppcm(p, len(c))
return p

```

Soyons sans peur, testons sur une permutation s de taille 100000. On calcule l'ordre ω de s , puis on vérifie que $s^\omega = id$. Le tout prend moins de 5 secondes sur ma machine (qui n'est pas très rapide). Cela dépend un peu évidemment de la permutation tirée au sort.

Pour $n = 10^6$, le calcul de l'ordre se fait sur ma machine en 4 ou 5 secondes, la vérification en une trentaine de secondes.

```

[45]: n = 100000
s = random_perm(n)
o = ordre(s)
print(o)
print(puissance(s, o) == id(n))

```

1509852252538453309200

True

Quel est l'ordre moyen μ_n d'une permutation de taille n ? Nous admettrons ici que

$$\ln \mu_n \sim c \sqrt{\frac{n}{\ln n}}$$

où

$$c = 2 \sqrt{2 \int_0^\infty \frac{\ln(1+t)}{e^t - 1} dt} \simeq 2.99047$$

L'intégrale apparaissant dans la constante est la **constante de Goh et Schmutz**.

Juste histoire de vérifier cette valeur, voici une fonction `trapezes`. Elle renvoie une approximation de $\int_a^b f(x) dx$ par la méthode des trapèzes.

```

[46]: def trapezes(f, a, b, n):
    d = (b - a) / n
    s = 0
    for k in range(n):
        s += f(a + k * d)
    s += (f(b) - f(a)) / 2
    return d * s

```

Voici la constante de Goh et Schmutz.

```

[47]: goh_schmutz = trapezes(lambda x: math.log(1 + x) / (math.exp(x) - 1), 0.00001,
    ↪ 100, 10000)
print(goh_schmutz)

```

1.1178624844223848

Et voici une estimation de la constante c ci-dessus.

```
[48]: print(2 * math.sqrt(2 * goh_schmutz))
```

2.99046816993244

Ne soyons pas trop exigeants dans nos tests. Je cite l'article dans lequel j'ai trouvé ce résultat :

“It turns out that a small set of exceptional permutations contributes significantly to the value of μ_n ”.

L'article en question est *William M. Y. Goh, Eric Schmutz, The expected Order of a Random Permutation, Bull. London Math. Soc. 23 (1991) 34-42*.

Cauchemar du statisticien, un petit nombre d'individus domine le groupe. Un calcul sur quelques (milliers de) permutations donnera donc un résultat a priori pas bon.

```
[49]: def stats_ordre(n, N=1000):
      av = 0.
      for k in range(N):
          s = random_perm(n)
          av = av + ordre(s)
      return av / N
```

```
[50]: mu = stats_ordre(1000)
      print(mu)
```

13042078982294.71

```
[51]: n = 1000
      c = 2.99047
      print(math.log(mu))
      print(c * math.sqrt(n / math.log(n)))
```

30.199202090876827

35.980813506042125

On s'en contentera :-).

Ce n'est évidemment pas la fin de l'histoire, d'autres quantités associées aux permutations se prêtent aussi à des explorations probabilistes très intéressantes (et souvent non triviales).

```
[ ]:
```