Génération aléatoire d'une permutation de Genocchi

Ramelina L.A[1], Randimbindrainibe F.[2]

Laboratoire de recherche : Sciences Cognitives et Applications

(LR – SCA)

Ecole Doctorale en Science et Technique de l'Ingénierie et de l'Information

(ED – STII)

Ecole Supérieure Polytechnique Antananarivo

BP 1500, Ankatso – Antananarivo 101 – Madagascar

[1]monjyram@gmail.com, [2]falimanana@mail.ru

Résumé

Une méthode récursive pour générer des permutations de Genocchi reste à proposer. D'une part, ce travail vise à fournir une génération aléatoire d'une permutation Genocchi et d'autre part, à faire appel à un logiciel libre pour permettre l'utilisation de cette permutation dans des aspects appliqués tels que la gestion des files d'attente ou toute autre perspective connexe

Mots clés : Nombres de Genocchi, Permutations, Permutations de Genocchi.

Abstract

A recursive method to generate Genocchi permutations is yet to be proposed. On the one hand, this work aims to provide a random generation of a Genocchi permutation and on the other hand, to call upon a free software to allow the use of this permutation in applied aspects such as the queue management or any other related perspectives.

Keywords : Genocchi numbers, Permutations, Genocchi permutations.

1. Introduction

Les nombres de Genocchi sont définis comme des séries de nombres utiles pour certaines énumérations avancées telles que les permutations de Dumont; dans lequel chaque nombre pair est suivi d'un nombre plus petit; puis chaque entrée impaire est suivie d'un nombre plus grand (une ascension) ou termine la chaîne [1]. Les recherches sur les permutations de Genocchi ont manqué d'offrir une méthode récurrente pour les générer. Cependant, Dumont [1] a démontré qu'ils peuvent être dénombrés à partir des nombres de Genocchi. Le but de ce travail est de proposer une génération aléatoire de ces permutations en utilisant la finesse du langage Python (logiciel libre) afin d'appliquer cette problématique par exemple dans la gestion des files d'attente.

MADA-ETI

1.1 Nombres de Genochi

Rappelons que les ***nombres de Genocchi*** G_{2n} sont définis [1] par la relation :

$$\frac{2t}{e^t+1} = t + \sum_{n\geq 1}(-1)^n G_{2n}\frac{t^{2n}}{(2n)!}.$$

Notons ont des relations avec les *nombres de Bernoulli* B_{2n} et *les nombres tangents* T_{2n-1} définis respectivement par :

$$\frac{t}{e^t-1} = 1 - \frac{1}{2}t + \sum_{n\geq 1}(-1)^{n+1}B_{2n}\frac{t^{2n}}{(2n)!}$$

et

$$\tan t = \sum_{n\geq 1} T_{2n-1}\frac{t^{2n-1}}{(2n-1)!}.$$

On a :

$$G_{2n} = 2(2^{2n}-1)B_{2n}$$

et

$$2^{2n-2}G_{2n} = nT_{2n-1}.$$

Le tableau suivant nous montre les premières valeurs de ces trois suites de nombres :

n	1	2	3	4	5	6	7
G_{2n}	1	1	3	17	155	2073	38227
B_{2n}	$\frac{1}{6}$	$\frac{1}{30}$	$\frac{1}{42}$	$\frac{1}{30}$	$\frac{5}{66}$	$\frac{691}{2730}$	$\frac{7}{6}$
T_{2n-1}	1	2	16	272	7936	353792	22358256

1.2 Méthode de calcul des nombres de Gennochi

Dumont et Viennot [1] ont trouvé une méthode simple pour calculer les nombres de Genocchi

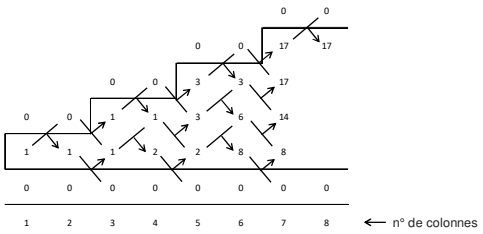

Figure 1 : *Méthode de calcul des nombres de Genocchi*

L'extérieur de ce tableau en forme d'escalier est rempli par des zéros. On place 1 à l'intérieur à l'extrême gauche. Un nombre dans une colonne d'indice pair est obtenu en additionnant le nombre qui se trouve à sa gauche et le nombre qui se trouve au-dessus. De même, un nombre dans une colonne d'indice impair est obtenu en additionnant le nombre qui se trouve à sa gauche et le nombre qui se trouve en-dessous. Les nombres de Genocchi sont les nombres qui se trouvent sur la partie supérieure de l'escalier. Notons que Dumont a donné les premières interprétations combinatoires des nombres de Genocchi en termes de permutations de Genocchi.

2. Des algorithmes de permutations

Une permutation est le réarrangement de n éléments distincts deux à deux. Appelons $\mathcal{S}_n$ l'ensemble des permutations de $\{1,2,\cdots,n\}$, on a $\operatorname{card}\mathcal{S}_n = n!$.

Pour toute permutation σ de $\mathcal{S}_n$ définie par

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{pmatrix},$$

nous allons adopter pour la suite son écriture linéaire :

$$\sigma = \sigma(1)\sigma(2)\cdots\sigma(n).$$

Pour générer une permutation, plusieurs algorithmes peuvent être utilisés. Nous allons en citer trois algorithmes principaux : algorithme d'insertion, algorithme de Heap et algorithme de Steinhaus-Johnson-Trotter ou SJT

2.1 Algorithme par insertion

Après avoir trouvé toutes les permutations de longueur $n-1$, on insère n dans les espaces possibles de chaque permutation de longueur $n-1$ pour avoir toute les permutations de longueur n [2]

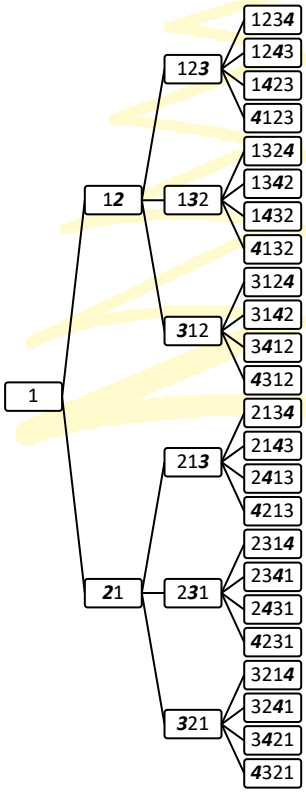

Figure 2 : *Méthode par insertion*

2.2 Algorithme de Heap

Il s'agit de trouver toutes les permutations de longueur n à partir de la donnée d'une permutation de longueur n. Etant donnée une permutation de longueur n, on fixe son premier élément, et on permute le reste. Pour permuter le reste, on fixe son premier élément et on permute le reste. On réitère le même procédé jusqu'à ce que le reste soit de longueur 2 que l'on permute facilement. [2]

2.3 Algorithme de Steinhaus-Johnson-Trotter (SJT)

Les algorithmes classiques délivrent les permutations à partir d'un ordre lexicographique (alphabétique ou croissant pour les nombres). Ces algorithmes fonctionnent par propagation.

L'algorithme de Steinhaus-Johnson-Trotter (comme l'algorithme de Heap) délivre une permutation et dans un ordre particulier [2]. Cet algorithme fonctionne par cheminement direct parmi toutes les permutations.

On commence par écrire tous les éléments dans l'ordre croissants : 1, 2, 3 par exemple. On donne une direction identique à chaque élément au départ : $\overleftarrow{1}$, $\overleftarrow{2}$, $\overleftarrow{3}$. Un nombre est mobile si le nombre pointé par sa flèche lui est plus petit : ici 3 et 2 sont mobiles. On peut alors effectuer les opérations suivantes :

1) On choisit le plus grand élément mobile et on permute avec le voisin pointé par sa flèche.	$\overleftarrow{1}$,	$\overleftarrow{3}$,	$\overleftarrow{2}$
2) On vérifie s'il existe d'autres éléments plus grands mobiles. Si oui, changer leur direction	**ici, non**		
3) Sinon, on poursuit avec le plus grand en cours de traitement (3).	$\overrightarrow{3}$,	$\overleftarrow{1}$,	$\overleftarrow{2}$
Lorsque cet élément ne peut plus bouger, alors, on reprend l'étape 1).	$\overrightarrow{3}$,	$\overleftarrow{2}$,	$\overleftarrow{1}$
Puisque 3 est plus grand que 2, alors on inverse le sens de 3.	$\overrightarrow{3}$,	$\overrightarrow{2}$,	$\overleftarrow{1}$
Ce nombre 3 redevient le plus grand mobile.	$\overrightarrow{2}$,	$\overrightarrow{3}$,	$\overleftarrow{1}$
Nouvelle possibilité de permutation.	$\overrightarrow{2}$,	$\overleftarrow{1}$,	$\overleftarrow{3}$
Aucun élément mobile.	**Fin**		

Figure 3: *Algorithme SJT*

En résumé, ce premier aperçu peut être décrit comme suit :

Mettre les éléments dans l'ordre croissant.

Tant qu'il existe un élément mobile:

Trouver l'élément mobile M le plus grand;

Permuter ce nombre M et son voisin pointé; et

Changer la direction de tous les nombres plus grands que M.

3. Permutation de Genocchi

Une permutation $\sigma \in \mathcal{S}_{2n}$ est appelée ***permutation de Genocchi*** [3] si $\sigma(2i-1) > 2i-1$, $\sigma(2i) \leq 2i$ pour tout $i \in \{1,2,\cdots,n\}$. L'ensemble des permutations de Genocchi de $\{1,2,\cdots,2n\}$ sera notée $\mathcal{G}_{2n}$.

Considérons par exemple $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 2 & 7 & 1 & 6 & 5 & 8 & 3 \end{pmatrix}$. On voit bien que $\sigma \in \mathcal{G}_8$ et l'on peut le présenter graphiquement de la manière suivante :

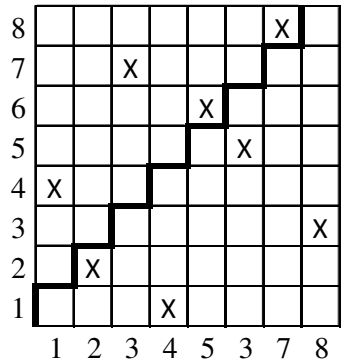

Figure 4 : *Une permutation de* $\mathcal{G}_8$

Remarquons que, pour toute permutation $\sigma \in \mathcal{G}_{2n}$, on a toujours $\sigma(2n-1) = 2n$ et Dumont à démontrer que $\text{card}(\mathcal{G}_{2n}) = G_{2n+2}$.

3.1 Construction d'une permutation de Genocchi

Proposons une méthode pour construire une permutation de Genocchi. Pour la permutation de Genocchi σ de taille $2n$, il suffit de trouver les $2n-2$ premiers termes étant donné que $\sigma(2n-1)$ est toujours égal à $2n$ et, $\sigma(2n)$ serait le nombre qu'on n'a pas utiliser dans la construction.

Considérons dans cette construction deux matrices lignes :

- La matrice $C = [1,2,\ldots,2n-2]$ dans laquelle on va choisir aléatoirement les $2n-2$ premiers termes de σ ;
- La matrice B qui est vide au début et qui va contenir au fur et à mesure les éléments de σ.

Considérons par exemple la construction d'une permutation de Genocchi de taille 8 avec son écriture $\sigma(1)\sigma(2)\sigma(3)\sigma(4)\sigma(5)\sigma(6)\sigma(7)\sigma(8)$.

On a nécessairement $\sigma(7) = 8$ et $\sigma(8)$ représente le dernier nombre qu'on n'a pas choisi. Sachant que $\sigma(2i-1) > 2i-1$ et $\sigma(2i) \leq 2i$ pour i allant de 1 à n, on commence par construire les éléments

d'indice impair du plus grand au plus petit, puis, les éléments d'indice pair du plus petit au plus grand :

1) Le choix de $\sigma(5)$ se fait dans $C = [1,2,3,4,5,6,7]$ avec la condition $\sigma(5) > 5$. Supposons que 6 soit choisi, la matrice B devient $B = [6]$ et 6 n'apparait plus dans C. On a alors $C = [1,2,3,4,5,7]$;
2) La condition $\sigma(3) > 3$ doit être vérifiée. Si $\sigma(3) = 7$, on insère 7 au début et B devient $B = [7,6]$, et, $C = [1,2,3,4,5]$.
3) Si $\sigma(1) = 2$, $B = [2,7,6]$ et $C = [1,3,4,5]$;
4) Venons maintenant aux éléments d'indice pair. La condition $\sigma(2) \leq 2$ nous force à choisir $\sigma(2) = 1$. On insère $\sigma(2)$ à la $2^{ème}$ place dans B. On a $B = [2,1,7,6]$ et $C = [3,4,5]$.
5) Si $\sigma(4) = 4$, on insère $\sigma(4)$ à la $4^{ème}$ place dans B. On a $B = [2,1,7,4,6]$ et $C = [3,5]$.
6) Si $\sigma(6) = 3$, $B = [2,1,7,4,6,3]$ et $C = [5]$.
7) Finalement, il suffit d'insérer $\sigma(7) = 8$ et $\sigma(8) = 5$. L'écriture linéaire de σ est 21746385.

3.2 Génération aléatoire d'une permutation de Genocchi utilisant Python

Une génération aléatoire d'une permutation Genocchi est possible en utilisant un logiciel libre comme Python. La flexibilité du réglage offre plus de liberté pour programmer les algorithmes de permutation. Le principe détaillé dans la section 3.1 est conservé, mais certaines syntaxes sont nécessaires pour faire fonctionner le logiciel et elles sont listées ci-dessous par leur signification. Proposons d'abord le programme suivant :

```
n=int(input('n = ?  '))
print(' ')
c=list(range(1,2*n))
random.shuffle(c)
b=[]
i=2*n-3
while i>=0:
    t=c.copy()
    d=t[0]
    t.remove(d)
    while d<=i:
        random.shuffle(t)
        d=t[0]
        t.remove(d)
    b.insert(0,d)
    c.remove(d)
    del d
    i-=2
x=2
while x<2*n:
    t=c.copy()
    d=t[0]
    t.remove(d)
    while d>x:
        random.shuffle(t)
        d=t[0]
        t.remove(d)
    b.insert(x-1,d)
    c.remove(d)
    del d
    x+=2
b.append(2*n)
b.append(c[0])
print(b)
```

Détaillons quelques syntaxes du programme proposé :

- c=list(range(1,2*n)) : on initialise la matrice $C = [1,2,\ldots,2n-1]$. Dans cette instruction, le dernier élément de C est $2n-1$;
- random.shuffle(c) : cette instruction permet de mélanger aléatoirement les éléments de C. Dans la suite du programme, on choisit toujours le premier élément de C après que celui-ci soit mélangé aléatoirement ;
- d=t[0] : on importe la valeur t[0] dans la variable d. La matrice t est temporairement une copie de la dernière matrice C. Cette copie est effectuée à partir de la commande t=c.copy(c) ;
- b.insert(0,d) : cela permet d'insérer au début de la matrice B indicé par 0 l'élément de valeur d. Ainsi une instruction de type b.insert(4,d) par exemple insère la valeur d au rang $4-1=3$. La taille de la matrice B augmente, ce qui î pas possible dans certains langages de programmations ;
- c.remove(d) : on supprime l'élément de valeur égale à la valeur d. La taille de la matrice C diminue de taille après cette instruction ;
- b.append(c[0]) : cette instruction permet d'ajouter au dernier rang de la matrice B l'élément c[0].

Quelques résultats

Le lancement du programme donne le résultat suivant pour $n = 4$:

Donner n = ? 4
Une permutation de taille 8 est :
[2, 1, 6, 4, 7, 3, 8, 5]

Comme dans l'exemple précédent, donnons 10 permutations de Genocchi de taille $2 \times 4 = 8$ choisis au hasard par le programme :

[2, 1, 4, 3, 6, 5, 8, 7]
[2, 1, 4, 3, 7, 6, 8, 5]
[3, 2, 5, 4, 6, 1, 8, 7]
[6, 1, 4, 2, 7, 5, 8, 3]
[3, 1, 5, 4, 6, 2, 8, 7]
[2, 1, 5, 3, 7, 6, 8, 4]
[2, 1, 5, 4, 7, 6, 8, 3]
[3, 1, 6, 2, 7, 5, 8, 4]
[2, 1, 4, 3, 6, 5, 8, 7]
[3, 2, 7, 1, 6, 5, 8, 4]

Nous pouvons remarquer que dans la donnée des 10 permutations, on ne trouve pas de répétitions : ceci peut montrer l'efficacité du générateur de nombres pseudo-aléatoires de Python. Rappelons que pour $n = 4$ le nombre de permutations de Genocchi de taille 8 est 155.

Donnons maintenant un résultat en variant l'entier n de 5 à 10 :

[6, 1, 4, 2, 9, 5, 8, 7, 10, 3]

[3, 1, 5, 2, 6, 4, 11, 8, 10, 9, 12, 7]

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

[2, 1, 5, 3, 12, 4, 8, 7, 13, 9, 14, 10, 15, 6, 16, 11]

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

Dans ces exemples tous les éléments sont distinct 2 à 2, et les conditions sont toujours vérifiées. Pour un entier assez grand, le programme ne présente pas de difficultés particulières sauf que le temps d'exécution est un peu long.

Le module "time" avec l'instruction "time.time()" permet de donner la durée d'exécution d'un série d'instructions exprimée en secondes. Si le début de l'exécution est défini par t1= time.time(), et la fin par t2= time.time(), la durée est donné par t2-t1.

Proposons un tableau qui montre la durée en seconde d'exécution de génération d'une permutation de Genocchi en fonction de sa taille.

Tableau 1 : *Durées d'exécution de génération d'une permutation*

taille / processeur	100	200	300	400	500	600	700	800	900	1000
pentium dual core 32 bits	0,036	0,14	0,312	0,577	0,952	1,607	1,685	2,621	3,806	4,368
intel core i5 64 bits	0,016	0,109	0,266	0,285	0,447	0,688	1,095	1,409	1,866	2,134

Nous pouvons donner un graphique exprimant la durée d'exécution en fonction de la taille de la permutation de Genocchi cherchée :

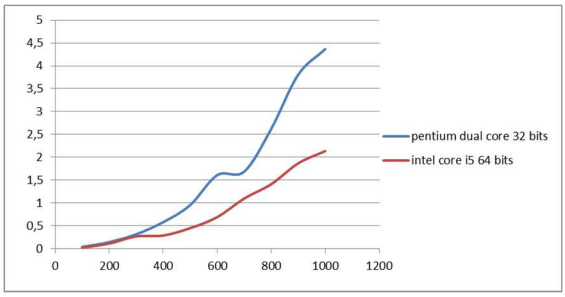

Figure 5: *Durées d'exécution de génération d'une permutation*

Interprétations :

La courbe en ligne continue représente le résultat obtenu avec un processeur "intel core 15 64 bits" qui est évidemment plus rapide en exécution qu'un processeur "pentium dual core 32 bits". Dans les deux cas, les courbes sont croissantes en générale, mais représentent des irrégularités. Ces dernières sont dues à l'aléa de la génération des nombres entiers suivant des conditions précises : le nombre de boucles effectuées est aussi aléatoire. On peut alors proposer des durées moyennes pour chaque taille de permutation pour avoir beaucoup plus de précision. Au lieu de donner une valeur pour une taille, il serait préférable de donner la moyenne des 100 durées pour une taille. On a le tableau suivant :

Tableau 1 : *Durées moyennes d'exécution de génération d'une permutation*

taille / processeur	100	200	300	400	500	600	700	800	900	1000
pentium dual core 32 bits	0,021	0,112	0,279	0,556	0,916	1,404	2,001	2,752	3,573	4,601
intel core i5 64 bits	0,01	0,051	0,132	0,265	0,442	0,671	0,965	1,387	1,75	2,188

Graphiquement, on a :

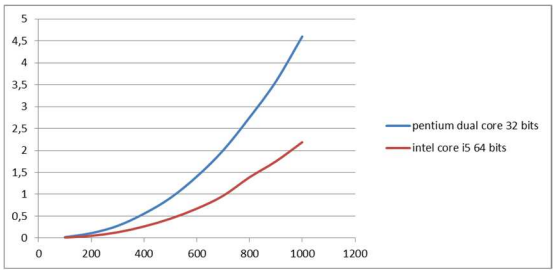

Figure 6: *Durées moyenne d'exécution de génération d'une permutation*

Pour chacune des deux courbes, l'évolution est plus régulière. A titre d'information, signalons que la durée moyenne pour générer une permutation de Genocchi de longueur 2000 est de 10,515 secondes. Il a fallu donc 1051,5 secondes soit 17 mn 31,5 secondes pour l'ordinateur "intel core i5 64 bits" pour le calculer.

4. Conclusion

En résumé, ce travail suggère une génération aléatoire des permutations de Genocchi. Si des recherches antérieures ont calculé le nombre de permutations à partir des nombres de Genocchi, elles n'ont pas réussi à fournir au hasard leur combinaison. Ainsi, cette étude a mis en évidence une génération aléatoire d'une permutation de Genocchi. Ces travaux ont favorisé l'utilisation de logiciels libres pour générer des permutations. Le choix du logiciel Python reste dans sa souplesse de programmation et plus d'automatisme dans la randomisation. Cependant, l'exécution du temps peut être difficile en fonction du type de processeur de l'ordinateur. Le présent travail a fourni - une génération aléatoire d'une permutation Genocchi - a favorisé son utilisation dans des aspects appliqués tels que la gestion des files d'attente.

BIBLIOGRAPHIE

[1]. D. Dumont and G. Viennot, "*A combinatorial interpretation of the Seidel generation of Genocchi numbers*", Ann. Discrete Math. 6 (1980) 77-87.
[2]. E. J. Hartung, H. P. Hoang, T. Mütz and A. Williams, "*Combinatorial generation via permutation langage. I. Fundamentals*", Efficient Generation of Combinatorial Objects using Generalized Gray Codes, (2019).
[3]. A. Randrianarivony and J. Zeng, "*Some equidistributed statistics on Genocchi permutations*", Electronic journal of combinatorics, 3 (2) (1996).

MADA-ETI