université de BORDEAUX

LICENCE 3 : Mathématiques et Informatique

Projet tutoré : Un algorithme pour le théorème d'Erdős-Ginzburg-Ziv

Présenté par :
Sadick Youssouf et Jean Mayol

Tuteur enseignant :
Eric Balandraud

Date :
12 mai 2024

12 mai 2024

Table des matières

1 Introduction

FIGURE 1 – Paul Erdős en 1992.

FIGURE 2 – Abraham Ginzburg.

FIGURE 3 – Ziv Abraham.

Pour ce projet tutoré, nous avons travaillé sur des théorèmes mathématiques de combinaison mathématique et nous avons implémenté des algorithmes donnant les résultats de ces théorèmes. Tout du long, nous avons travaillé dans un groupe abélien fini $\mathbb{Z}/n\mathbb{Z}$ d'ordre n, mais les théorèmes s'étendent à des groupes abéliens quelconques et pour les groupes non abéliens ce n'est que conjecturale.

Le théorème d'Erdős-Ginzburg-Ziv, également connu sous le nom de théorème de somme zéro, est un résultat fondamental en théorie des nombres qui concerne les ensembles de nombres dans les groupes abéliens finis. Ce théorème énonce une propriété importante sur les sous-suites de sommes nulles dans de tels groupes. Dans un groupe abélien fini G, le problème de la somme nulle est de déterminer, pour tout entier $n > 0$, le plus petit entier k tel que toute suite de k éléments de G contienne une sous-suite de n termes de somme 0. Ce théorème a une grande importance en théorie des nombres et en combinatoire, car il fournit des informations cruciales sur les propriétés des ensembles de nombres dans les groupes abéliens finis. Il permet notamment de déterminer la structure des sous-ensembles de sommes nulles dans de tels groupes, ce qui trouve des applications dans de nombreux domaines des mathématiques et de l'informatique. Le théorème a été démontré pour la première fois en 1961 par Paul Erdős (un mathématicien hongrois prolifique et influent du XXe siècle, célèbre pour ses contributions dans de nombreux domaines des mathématiques, y compris la théorie des nombres, la combinatoire, et la théorie des graphes.), Abraham Ginzburg (mathématicien et informaticien Biélorusse, qui a travaillé sur des problèmes liés à la théorie des groupes, théorie de graphes et les automates) et Abraham Ziv (un informaticien israélien). Leur démonstration a établi que dans le groupe additif de l'anneau $\mathbb{Z}/n\mathbb{Z}$, le plus petit entier k tel que toute suite de k éléments contienne une sous-suite de n termes de somme 0 est égal à $2n-1$. [4]

Notre objectif était d'implémenter un algorithme recent dû à Del Lungo Marini, Mori [1]. Pour ce faire, cet algorithme s'appuie sur un théorème de Marshall Hall, Jr, Salzborn et Szekeres qui dit que dans un groupe abélien fini d'ordre n G, pour $a(1), \ldots, a(n-1)$ une suite d'éléments de G, non nécessairement distincts, il existe deux $(n-1)$-uplets ordonnés d'éléments distincts de G, $(p(1), \ldots, p(n-1))$ et $(q(1), \ldots, q(n-1))$, tels que $p(i) = a(i) + q(i)$, pour $i = 1, \ldots, n-1$. En 1952, Hall a prouvé ce théorème et en 1979, Salzborn et Szekeres ont redemontré ce théorème sans savoir que Hall l'a déjà fait, mais leurs preuves étaient similaires [2], [3].

L'algorithme de Del Lungo, Marini, Mori répose sur le théorème de Hall, Salzborn et Szekeres, cet algorithme donne une preuve du théorème d'Erdős-Ginzburg-Ziv qui s'énonce comme suit : dans un groupe abélien fini G d'ordre n, pour $x(1), \ldots, x(2n-1)$ des éléments de ce groupe, pas nécessairement distincts, il existe un sous-ensemble $I \subset \{1, 2, \ldots, 2n-1\}$ tel que $|I| = n$ et $\sum_{i \in I} x(i) = 0$.

Cet algorithme peut se comprendre comme une suite d'échange entre deux parties.

2 Théorème de Hall, Salzborn et Szekeres

2.1 Théorème et preuve

Théorème 1 (Hall, Salzborn et Szekeres)**.** *Soit G un groupe abélien fini d'ordre n, soit $a(1), \ldots, a(n-1)$ des éléments de ce groupe, pas nécessairement distincts. Alors, il existe deux ensembles ordonnés de $(n-1)$ éléments distincts de G, $(p(1), \ldots, p(n-1))$ et $(q(1), \ldots, q(n-1))$, où les $p(i)$ sont tous 2 à 2 distincts et les $q(i)$ aussi, tels que $p(i) = a(i) + q(i)$ pour $i = 1, \ldots, n-1$.*

Démonstration. On complète la liste avec $a(n) = -\sum_{i=1}^{n-1} a(i)$

Notons la liste $La = [(a(1), \ldots, a(n))]$, on construit les listes Lp et Lq, par induction.

Étape initiale : Nous choisissons $p(1) = a(1)$ et $q(1) = 0$, $Lp = [p(1)]$ et $Lq = [q(1)]$.

Étape intermédiaire : Supposons qu'on a construit Lp et Lq de taille $k \leq n-2$. Nous choisissons un $\tilde{q} \notin Lq$, puis nous calculons $a(k+1) + \tilde{q} = \tilde{p}$.

- Si $\tilde{p} \notin Lp$, nous ajoutons $\tilde{p}$ à Lp et $\tilde{q}$ à Lq. Ainsi nous aurons $Lp = [p(1), \ldots, p(k), \tilde{p}]$ et $Lq = [q(1), \ldots, q(k), \tilde{q}]$.
- Si $\tilde{p} \in Lp$, alors nous avons que $a(k+1) + \tilde{q} = p(i_1)$ et nous calculons $a(i_1) + \dot{q} = p(i_2)$, avec $\dot{q} \notin Lq$ et $\dot{q} \neq \tilde{q}$.
  - Si $p(i_2) \notin Lp$, nous l'ajoutons à la liste.
  - Si $p(i_2) \in Lp$, continuons ce processus : c'est-à-dire

$$
\begin{aligned}
a(k+1) + \tilde{q} &= p(i_1) \\
a(i_1) + \dot{q} &= p(i_2) \\
a(i_2) + q(i_1) &= p(i_3) \\
a(i_3) + q(i_2) &= p(i_4) \\
&\vdots \\
a(i_f) + q(i_f - 1) &= \tilde{p}
\end{aligned}
$$

Montrons que les indices $i_1, i_2 ... i_f$ sont finis et distincts : On a que pour un certain indice $r > 0$:

$$
\begin{cases}
a(r+1) + \tilde{q} + \dot{q} = p(i_1) + \dot{q} \\
\qquad\qquad\quad = p(i_2) + q(i_1) \\
\qquad\qquad\quad = p(i_3) + q(i_2) \\
\qquad\qquad\quad \vdots \\
\qquad\qquad\quad = p(i_f) + q(i_f - 1) \\
\qquad\qquad\quad = \tilde{p} + q(i_f)
\end{cases}
$$

Si cela était infini on aurait par exemple $p(i_3) + q(i_2)$ serait égale à $p(i_m) + q(i_m - 1)$.

pour un certain idice m et donc on aura que $p(i_3) = p(i_m)$ et cela en train une **contradiction** car on a supposé que tous les $p(i)$ étaient tous 2 à 2 distincts.

Étape finale : Ainsi nous avons pu former les $n-1$ éléments $[(p(1), \ldots, p(n-1))]$ de la liste Lp le dernier élément de la liste n'est rien d'autre $\sigma - (\sum_{i=1}^{n-1} p(i))$, où σ est la somme de tous les éléments de G. De la même méthode nous avons les $n-1$ éléments $[(q(1), \ldots, q(n-1))]$ de la liste Lq le dernier élément est $\sigma - (\sum_{i=1}^{n-1} q(i))$

□

Dans notre implémentation nous nous sommes limités au cas où $G = \mathbb{Z}/n\mathbb{Z}$.

2.2 Exemple d'illustration de l'algorithme de Hall :

Voici un **exemple** d'application de l'algorithme de Hall avec $n = 6$:

a(i)=[4,3,2,2,0,1]	$\tilde{q}$	$\dot{q}$	$a(i) + \tilde{q}$	$p(i)$	$q(i)$
$a(1) = 4$				$p(1) = a(1) => [4]$	$q(1) = 0 => [0]$
a(2)=3	1		$a(2) + \tilde{q} = 4 = p(1)$		
		2	$a(1) + \dot{q} = 0$	$p(i) = [0, 4]$	$q(i) = [2, 1]$
a(3)=2	0		$a(3) + \tilde{q} = 2$	$p(i) = [0, 4, 2]$	$q(i) = [2, 1, 0]$
a(4)=2	3		$a(4) + \tilde{q} = 5$	$p(i) = [0, 4, 2, 5]$	$q(i) = [2, 1, 0, 3]$
a(5)=0	4		$a(5) + \tilde{q} = 0 + 4 = 4 = p(2)$		
		5	$a(2) + \dot{q} = 3 + 5 = 8 = 2 = p(3)$		
			$a(3) + q(2) = 3 + 0 = 3$	$p(i) = [0, 2, 3, 5, 4]$	$q(i) = [2, 5, 1, 3, 4]$
a(6)=1				$p(i) = [0, 2, 3, 5, 4]+[1]$	q(i) = [2,5,1,3,4]+[0]

2.3 Statistiques de l'algorithme de HALL

Voici quelques temps d'exécution moyens de l'algorithme de Hall.

n	5	10	20	50	100	200	300	400	500
Nb d'exécutions	10000	10000	10000	10000	10000	10000	1000	1000	1000
Temps d'exécution en secondes	0.000037	0.000046	0.00065	0.0019	0.0079	0.027	0.07	0.13	0.2

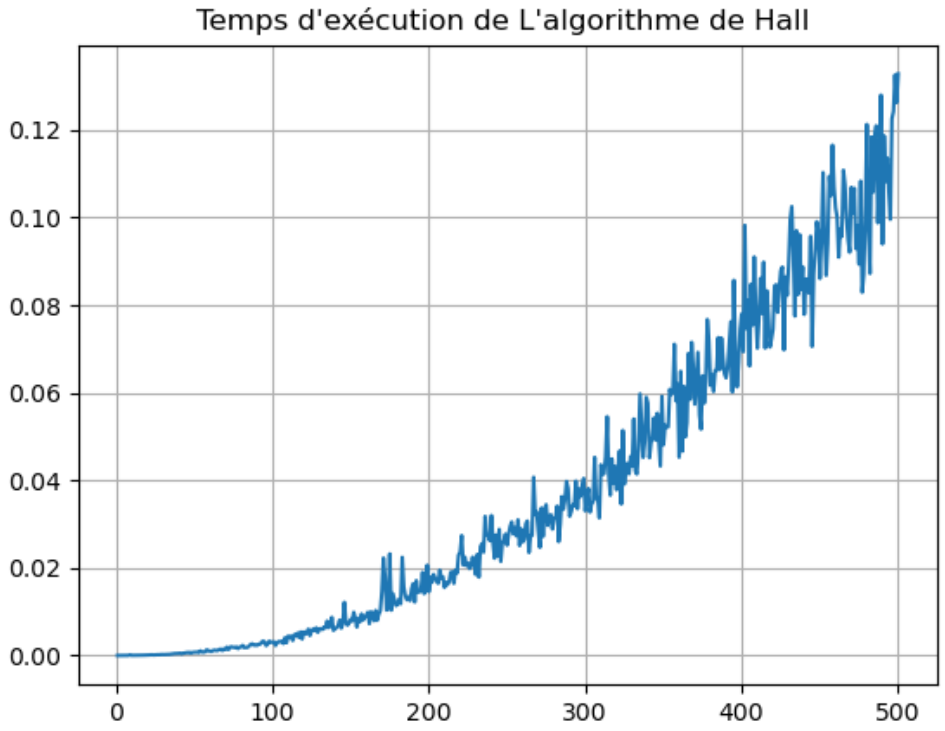

FIGURE 4 – Temps d'exécution pour n=5,...,500

3 Algorithme de Del Lungo, Marini, Mori

3.1 Présentation du théorème d'Erdős-Ginzburg-Ziv

Abordons maintenant le théorème de Erdős-Ginzburg-Ziv.

Théorème 2 (Erdős-Ginzburg-Ziv)**.** *Soit G un groupe abélien fini d'ordre n, et $x(1), \ldots, x(2n-1)$ des éléments de ce groupe, pas nécessairement distincts, il existe un sous-ensemble d'indices $I \subset \{1, 2, \ldots, 2n-1\}$ tel que $|I| = n$ et $\sum_{i \in I} x(i) = 0$.*

Del Lungo, Marini et Mori ce sont intéressés à ce théorème et en ont donné une nouvelle preuve basée sur le théorème de Hall. Cette nouvelle preuve est algorithmique. Selon eux, il y aurait une douzaine de preuves pour le théorème de Erdős-Ginzburg-Ziv, mais aucune ne produit d'algorithme efficace pour trouver la suite de somme à 0. Leur preuve permet de trouver un algorithme qui, sur une suite de $2n-1$ éléments dans un groupe abélien fini, trouve une suite de somme à 0 avec n éléments, le tout en $O(n^2)$, temps quadratique. Cet algorithme travaille directement dans le groupe G mais pas dans la structure du groupe G.

Cependant dans notre projet on se contente de travailler dans le groupe abélien additif $G = \mathbb{Z}/n\mathbb{Z}$.

3.2 Présentation de l'algorithme

L'algorithme ressemble à un jeu de ping-pong. Sur notre liste de longueur $2n-1$, on va construire 2 sous-listes de longueur $n-1$ et on va leur passer une « balle » à tour de rôle. Cette balle représente le dernier élément potentiel de la liste. Lorsqu'une balle rentre dans une liste, on envoie une autre balle à l'autre liste et ce jusqu'à ce que l'algorithme s'arrête. Cela se termine lorsqu'une sous-liste vérifie le théorème, c'est-à-dire lorsque la somme des éléments est nulle.

Soit une suite $x(1), \ldots, x(2n-1)$. On sépare cette suite en 2 sous-suites : $A = [x(1), \ldots, x(n-1)]$ et $B = [x(n), \ldots, x(2n-2)]$ et $c(0) = x(2n-1)$ le dernier élément. On ajoute un élément en dernière position de A ($a(0) = n$) et de B ($b(0) = n$) tel que la somme des éléments de chaque liste soit nulle et on applique l'algorithme de Hall afin d'avoir 2 listes de permutations telles que pour A :

$$p(i) = x(i) + q(i) \quad \text{avec} \quad 0 < i < n$$

Tous les éléments de $p(i)$ et $q(i)$ sont distincts.

Et pour B :

$$p'(i) = x(i+n) + q'(i) \quad \text{avec} \quad 0 < i < n$$

Tous les éléments de $p(i)$ et $q(i)$ sont distincts.

On a alors :

$$A = [x(1), \ldots, x(n-1)] \quad B = [x(n), \ldots, x(2n-2)]$$
$$p_A = [p(1), \ldots, p(n-1)] \quad p_B = [p'(1), \ldots, p'(n)]$$
$$q_A = [q(1), \ldots, q(n-1)] \quad q_B = [q'(1), \ldots, q'(n)]$$

On commence alors le « jeu ». Lorsqu'on échange une balle de A à B ou de B à A alors on appelle ça une transition. On définit $c(i)$ lorsque la balle est envoyée sur A et $d(i)$ lorsque la balle est envoyée sur B. On place l'élément $c(i)$ en position $a(i)$ (pour commencer en position n, c'est-à-dire à la place de $x(n)$). Ensuite, on vérifie la propriété. Si c'est juste, on s'arrête et la partie est finie. Sinon, on calcule le nouveau $p(i)$ correspondant par la relation $p(i) = x(i) + q(i)$. Vu que la propriété est fausse mais que tous les autres $p(i)$ sont distincts entre eux (on a construit p_A de telle sorte que les p_i soient distincts), il y a un autre élément égal à $p(i)$ dans p_A : $p(j)$. On regarde alors cet élément et on enregistre sa position ($a(i+1) = \text{pos}(p(j))$). On regarde l'élément de A à la nouvelle position $a(i+1)$ et on envoie cet élément à B $d(i)$. On répète la même procédure à B et on continue jusqu'à ce que la propriété de Erdős-Ginzburg-Ziv soit vérifiée.

3.3 Exemple d'illustration de l'algorithme

Prenons $n = 6$. On travaille donc dans $\mathbb{Z}/6\mathbb{Z}$. Soit la suite suivante : $[2, 5, 3, 4, 4, 2, 3, 3, 1, 0, 1]$ de longueur $6 \times 2 - 1 = 11$. On la sépare en 2 suites : $A = [2, 5, 3, 4, 4]$ et $B = [2, 3, 3, 1, 0]$ et $c(0) = 1$. On complète les 2 suites et on applique Hall afin de trouver 2 suites de permutations d'éléments distincts. $a(0) = b(0) = 5$ (car c'est la position où on ajoute l'élément en plus à A et B).

$$A = [2, 5, 3, 4, 4, 0] \qquad B = [2, 3, 3, 1, 0, 3]$$
$$p_A = [1, 0, 5, 4, 2, 3] \qquad p_B = [0, 2, 5, 4, 1, 3]$$
$$q_A = [5, 1, 2, 0, 4, 3] \qquad q_B = [4, 5, 2, 3, 1, 0]$$

Ensuite, on exécute le jeu de la balle jusqu'à trouver une solution :

$a(0) = 5$			$b(0) = 5$		
A	$[2, 5, 3, ④, 4, 1]$	$\leftarrow c(0) = 1$	B	$[2, 3, 3, ①, 0, ④]$	$d(0) = ④ \leftarrow$
p_A	$[1, 0, 5, \boxed{4}, 2, 4]$	$d(0) = ④ \rightarrow$	p_B	$[0, 2, 5, \boxed{4}, 1, 4]$	$c(1) = ① \rightarrow$
q_A	$[5, 1, 2, 0, 4, 3]$		q_B	$[4, 5, 2, 3, 1, 0]$	

$a(1) = 3$			$b(1) = 3$		
A	$[②, 5, 3, ①, 4, 1]$	$\leftarrow c(1) = ①$	B	$[2, 3, ③, ②, 0, 4]$	$d(1) = ② \leftarrow$
p_A	$[\boxed{1}, 0, 5, 1, 2, 4]$	$d(1) = ② \rightarrow$	p_B	$[0, 2, \boxed{5}, 5, 1, 4]$	$\rightarrow c(2) = ③$
q_A	$[5, 1, 2, 0, 4, 3]$		q_B	$[4, 5, 2, 3, 1, 0]$	

$a(2) = 0$			$b(2) = 2$		
A	$[③, 5, 3, 1, ④, 1]$	$\leftarrow c(2) = ③$	B	$[②, 3, ④, 2, 0, 4]$	$d(1) = ④ \leftarrow$
p_A	$[2, 0, 5, 1, \boxed{2}, 4]$	$d(2) = ④ \rightarrow$	p_B	$[0, 2, 0, 5, 1, 4]$	$\rightarrow c(2) = ②$
q_A	$[5, 1, 2, 0, 4, 3]$		q_B	$[4, 5, 2, 3, 1, 0]$	

$a(3) = 4$			$b(3) = 0$		
A	$[3, ⑤, 3, 1, ②, 1]$	$\leftarrow c(3) = ②$	B	$[⑤, 3, 4, 2, 0, 4]$	
p_A	$[2, \boxed{0}, 5, 1, 0, 4]$	$d(3) = ⑤ \rightarrow$	p_B	$[3, 2, 0, 5, 1, 4]$	
q_A	$[5, 1, 2, 0, 4, 3]$		q_B	$[4, 5, 2, 3, 1, 0]$	

3.4 Justification de l'algorithme

On voudrait savoir pourquoi l'algorithme se termine et renvoie bien une liste tel que la propriété soit vérifié. Pour cela, on suppose 2 cas particulier :

① : $\exists i, j$ avec $j > i$ tel que $a(0), \ldots, a(i), \ldots, a(j)$ tous distincts et $b(0), \ldots, b(j)$ tous distincts, mais $a(j+1) = a(i)$.

② : $\exists i, j$ avec $j > i$ tel que $a(0), \ldots, a(j+1)$ tous distincts et $b(0), \ldots, b(i), \ldots, b(j)$ tous distincts, mais $b(j+1) = b(i)$.

D'après certaines instances et calculs, on trouve que $b(i) = b(j)$ et que $a(i+1) = a(j+1)$. Or d'après les hypothèses, les $b(0), \ldots, b(i), \ldots, b(j)$ doivent être tous distincts, ce qui est **contradictoire**. Il en va de même pour les $a(0), \ldots, a(i+1), \ldots, a(j+1)$.

Donc il ne peut pas y avoir des répétitions d'échanges, or on travaille dans un groupe abélien fini donc l'algorithme se termine bien pour $2n - 1$ éléments.

3.5 Statistiques de l'algorithme

On calcule le temps d'exécution en moyenne de l'algorithme de Del Lungo, Marini, Mori pour n allant de 5 à 500.

n	5	10	20	50	100	200	300	400	500
Nb d'exécutions	10000	10000	10000	10000	10000	1000	1000	1000	1000
Nb_balle	$4-5$	$7-8$	14	34	68	136	200	268	334
Temps d'exécutions en secondes	0.00006	0.00008	0.0001	0.005	0.0014	0.058	0.14	0.27	0.52

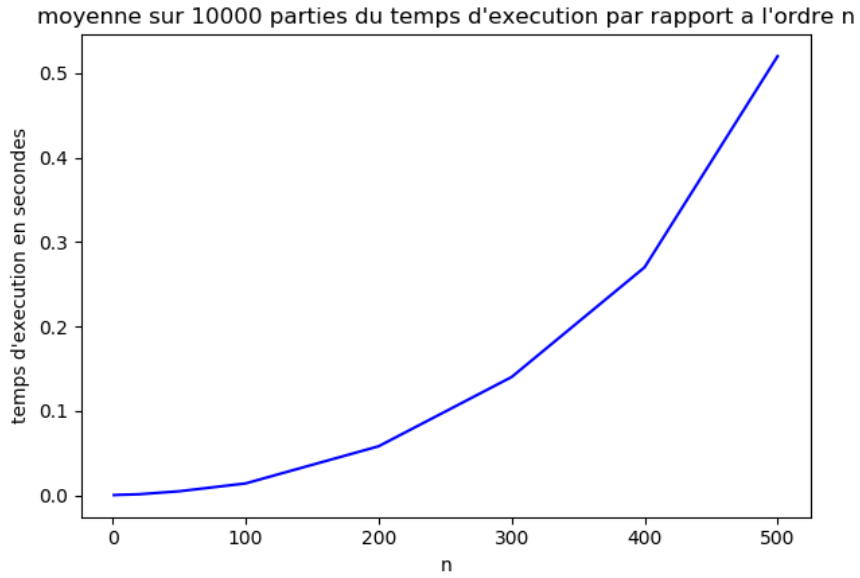

FIGURE 5 – Temps d'exécution pour n=5,...,500

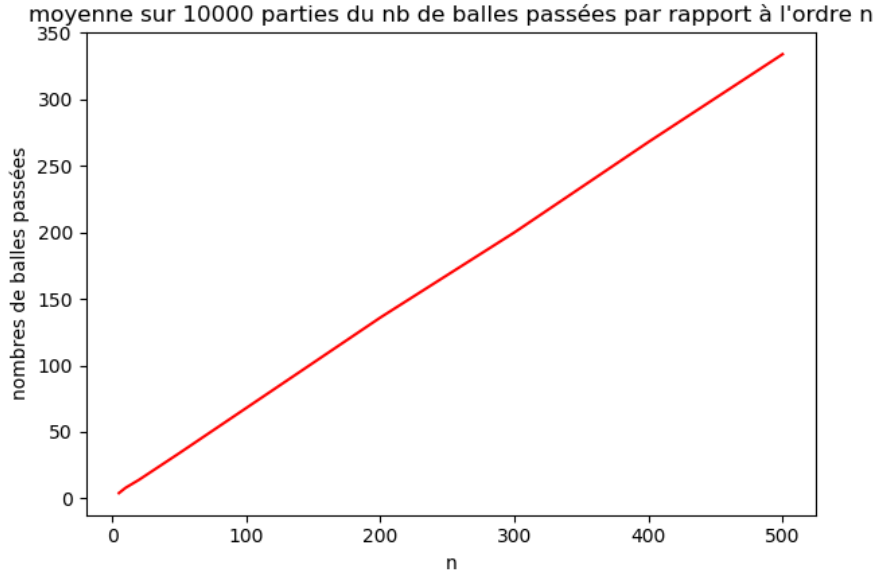

FIGURE 6 – Nombre de balles pour n=5,...,500

On remarque que c'est linéaire de coéfficient $\frac{2}{3}$, on va trouver en moyenne une liste qui satisfait le théorème proportionnellement au nombre d'échange entre les deux listes.

Nous avons voulu connaitre les nombres de listes qui satisfassent le théorème pour certains n. Pour ce faire, nous avons effectué 10000 mélanges d'une même liste et exécuter l'algorithme.

Nous l'avons fait sur 15 listes différentes pour avoir une moyenne probable.

n	5	6	7	10	12	15
nb_list	4	10	16	182	1012	5089

Le tableau représente la probabilité que la liste en sortie soit trouvée par la liste A ou par la liste B. On a fait la moyenne des probabilités sur 10000 parties.

n	2	3	5	10	20	50	60
Proba A	0,74	0,68	0,61	0,55	0,54	0,53	0,5
Proba B	0,26	0,32	0,39	0,45	0,46	0,47	0,5

4 Annexes

```
# Le code ci-dessous genere une liste de n elements aleatoire
# dont la somme est congru a zero modulo n.
def generate_sequence_a_i(n):
    liste_a_i = []
    for _ in range(n-1):
        random_num = random.randint(0, n-1)
        liste_a_i.append(random_num)

    total = sum(liste_a_i)

    remainder = total % n

    missing_num = (n - remainder) % n
    liste_a_i.append(missing_num)

    return liste_a_i
```

Explication de l'algorithme de Hall, Salzborn et Szekeres

1. Entreé : une liste d'entiers telle que la somme des entiers est congru à 0 modulo la longueur de la liste.
2. Initialisation :
   (a) Créer deux listes vides b et c
   (b) Créer une liste triée d de 0 à la longueur de la liste moins un (len(liste)-1)
   (c) Initialiser l'indice indice avec le minimum de la liste triée
   (d) Initialiser un autre indice i avec la même valeur que indice
3. Pour chaque élément i dans la liste :
   (a) Si i n'appartient pas à c :
      i. Ajouter i à c
      ii. Calculer b tel que $b[i] = a[i] + c[i]$
      iii. Supprimer la valeur minimale de d
      iv. Initialiser l'indice j à 0
      v. Mettre à jour indice avec la position de l'élément i dans c
      vi. Tant qu'il existe deux éléments identiques dans b :
         A. Stocker la valeur de c à l'endroit problématique dans une variable locale
         B. Intervertir cette valeur avec la première valeur de d
         C. Recalculer b pour que le résultat soit correct
4. **Retourner** les listes b et c

```
#Le code ci-dessous applique le theoreme de Hall/Salzborn-Szekeres
def Hall(a):
    Lp = []
    Lq = []
    liste_indice = [] # listes de 0 a n-1 trier dans l'ordre
    for k in range(len(a)):
        liste_indice.append(k)
    indice = min(liste_indice)  #minimum de d
```

```
    i = indice   #i choisi pour Lq

    while (i < len(a)):
        if (i not in Lq): #pour se permettre de reprendre les i deja utilises si ne
            Lq.append(i)
            calcul = (a[len(a)-len(liste_indice)] + i)%len(a)
            Lp.append(calcul)
            liste_indice.pop(0)  #on supprime la valeur minimal dans d
            j = 0 #indice pour comparer les elements de Lp
            indice = len(a) - len(liste_indice) - 1  #position de l'element i dans

            while (j < len(Lq)):
                if (Lp[j] == Lp[indice]): #si 2 elements de Lp sont les memes
                    if (indice != j): #cas ou c'est le meme element
                        permu = Lq[j] #on stoke la valeur qu'on va supprimer
                        Lq[j] = liste_indice[0] #on remplace la valeur qui pose pro
                        liste_indice[0] = permu #on remet la valeur supprimer dans
                        calcul2 = (a[j] + Lq[j])%len(a)
                        Lp[j] = calcul2 #on recalcul le Lp correspondant
                        indice = j #on remplace l'indice pour verifier si la nouve
                        j = 0
                    else:
                        j += 1
                else:
                    j+=1

        if (liste_indice == []):
            return Lp, Lq
        else:
            i = min(liste_indice)

    return Lp, Lq
```

Explication de l'algorithme de Del Lungo, Marini, Mori

Fonction théorème :

— Prend en paramètre une liste de longueur n et vérifie si la somme congrue modulo n vaut 0.

Algorithme principal :

1. **Entrée :** une liste d'entiers de longueur $(2n-1)$.
2. **Initialisation :**
   (a) On crée deux listes de longueur $(n-1)$ en attribuant à la première liste les $n-1$ premiers éléments de la liste en entrée et à la deuxième liste les $n-1$ éléments suivants.
   (b) On crée une balle qui prend le dernier élément de la liste en entrée.
   (c) On complète A et B.
   (d) On ajoute à A et à B le dernier élément tel que la somme des éléments modulo n soit nulle. On applique ensuite Hall pour trouver les suites de permutations de A et de B.
3. **Jeu de ping-pong :**
   (a) On insère la balle sur la liste A.
   (b) On vérifie si le théorème est vrai. Si c'est le cas, on s'arrête et on renvoie la liste.
   (c) On calcule $p(i) = a(i) + q(i)$, où i correspond à la position actuelle de A.

(d) On copie la valeur de la position de A pour qu'on puisse changer la valeur de la position et de la balle sans qu'on puisse comparer l'élément $p(i)$ avec lui-même.

(e) On parcourt la liste afin de trouver le doublon (la même valeur) de $p(i)$. Lorsqu'on l'a trouvé, on change la position de A par la position du doublon et on prend comme nouvelle balle la valeur de la liste A en nouvelle position A.

(f) On fait exactement la même chose pour B.

(g) On boucle ainsi l'algorithme jusqu'à trouver une solution. On boucle en $(2n-1)$ car c'est le nombre maximum d'échanges possibles entre A et B.

```
#Ici on applique l'algorithme de del Lungo, Marini et Mori

#C'est une fonction test pour savoir si la somme d'une liste est bien nulle
def theoreme(liste):
    if (sum(liste) % len(liste) == 0):
        return True

    return False

#Algo de ping-pong
def algo_DMM(liste):
    if (len(liste) % 2 == 0):
        return False
    lg_listes = int((len(liste)-1)/2)
    liste_A = [0] * lg_listes
    liste_B = [0] * lg_listes
    balle = liste[len(liste) - 1]
    for i in range(lg_listes):
        liste_A[i] = liste[i]
        liste_B[i] = liste[i + lg_listes]
    pos_A = lg_listes
    pos_B = lg_listes

    print("A = ", liste_A)
    print("B = ", liste_B)
    print("balle =", balle)
    print("position_A =", pos_A)
    print("position_B =", pos_B)

    liste_A.append((-sum(liste_A)) % (lg_listes + 1))
    liste_B.append((-sum(liste_B)) % (lg_listes + 1))

    print(liste_A)
    print(liste_B)
    p_A, q_A = Hall(liste_A)
    p_B, q_B = Hall(liste_B)

    print("les listes de permutations de A sont", p_A , q_A)
    print("les listes de permutations de B sont", p_B , q_B)

    nb_balle = 0

    for i in range(len(liste)):
        liste_A[pos_A] = balle
```

```
        nb_balle += 1

        if (theoreme(liste_A) == True):
            return 1

        p_A[pos_A] = (q_A[pos_A] + liste_A[pos_A]) % (lg_listes + 1)

        pos_init_A = pos_A
        for j in range(len(p_A)):
            if (p_A[j] == p_A[pos_A] and pos_init_A != j):
                pos_A = j
                balle = liste_A[pos_A]

        nb_balle += 1
        liste_B[pos_B] = balle

        if (theoreme(liste_B) == True):
            return 0

        p_B[pos_B] = (q_B[pos_B] + liste_B[pos_B]) % (lg_listes + 1)

        pos_init_B = pos_B
        for k in range(len(p_B)):
            if (p_B[k] == p_B[pos_B] and pos_init_B != k):
                pos_B = k
                balle = liste_B[pos_B]

    return False
```

5 Références

Références

[1] A. del Lungo, C. Marini, and E. Mori. A polynomial-time algorithm for finding zero-sums. Discrete Mathematics, 309 :2658–2662, 2009.

[2] M. Hall. A combinatorial problem on abelian groups. Proceedings of the American Mathematical Society, 3(4) :584–587, 1952.

[3] F. J. M. Salzborn and G. Szekeres. Ars combinatoria, a problem in combinatorial groups theory. Scheduling bus systems with interchanges, 7 :3–5, 1979.

[4] Wikipédia. Problème de la somme nulle. 11 janvier 2021.