Rapport TIPE

Lucas Malaizier MPI

Les semigroupes numériques

Introduction

Commençons par poser une énigme d'une aberrante simplicité :

> Imaginons un immeuble infini muni d'un ascenseur (c'est plus pratique, pour monter à l'étage 1586). Or, cet ascenseur ne comporte qu'un nombre fini de boutons, puisqu'il n'est lui-même pas infini.
>
> Supposons qu'il ne comporte que 3 boutons : $+3$, $+5$ et $+7$ permettant de monter respectivement de 3, 5 et 7 étages (ce qui reste peu pratique pour monter à l'étage 1586, certes).
>
> La question est donc : Quels étages ne peut-on pas atteindre avec l'ascenseur ?

Les étages 1, 2 et 4 sont inaccessibles.

> Supposons à présent que l'ascenseur possède $n \in \mathbb{N}^*$ boutons (toujours un nombre fini) : $+n_1$, $+n_2$, ..., $+n_n$. La question reste la même : Quels étages ne peut-on pas atteindre avec l'ascenseur ?

Cette fois, la question est beaucoup plus complexe, et il faut se pencher sur la notion de "semigroupes numériques" pour y trouver la réponse.

Première partie : Approche mathématique

I) Définition et exemples

> (1) ***Définition :*** Un **semigroupe numérique** est un sous-ensemble S de $\mathbb{N}$ tel que :
>
> - $0 \in S$.
> - S est stable par somme i.e. $\forall (x, y) \in S^2, x + y \in S$.
> - $\mathbb{N} \setminus S$ est fini.

Dans toute la suite, S désigne un semigroupe numérique et on notera $\overline{S} = \mathbb{N} \setminus S$.

Exemples :

- $\mathbb{N}$ est un semigroupe numérique car $0 \in \mathbb{N}$, $\mathbb{N}$ est stable par somme, et $\mathbb{N} \setminus \mathbb{N} = \emptyset$.
- $\mathbb{N} \setminus \{1\}$ est un semigroupe numérique.
- L'ensemble I des entiers impairs n'en est pas un car $0 \notin I$.
- L'ensemble $\mathbb{N} \setminus \{1, 5\}$ n'est pas un semigroupe numérique car $(2, 3) \in S^2$ mais $2 + 3 = 5 \notin S$.
- L'ensemble P des entiers pairs n'en est pas un non plus car $\mathbb{N} \setminus P = I$ ensemble infini.

> (2) ***Lemme :*** L'ensemble des semigroupes numériques est infini.

Démonstration : On peut aisément montrer que $\forall n \in \mathbb{N}^*, \mathbb{N} \setminus [\![1, n]\!]$ est un semigroupe numérique. Ainsi, l'ensemble $\{\mathbb{N} \setminus [\![1, n]\!]; n \in \mathbb{N}^*\}$ est infini et inclus dans l'ensemble des semigroupes numériques. Donc l'ensemble des semigroupes numériques est infini.

II) Vocabulaire

(3) ***Définitions :***

- On appelle **trou** de S un élément de $\overline{S}$.
- On définit :
  - Le **genre** de S, noté $g(S)$, par $g(S) = \text{card}(\overline{S})$.
  - La **multiplicité** de S, notée $m(S)$, par $m(S) = \min(S \setminus \{0\})$.
  - Le **nombre de Frobenius** de S, noté $f(S)$, par $f(S) = \max(\overline{S})$. Pour $S = \mathbb{N}$, on prendra par convention $f(S) = -1$.
  - Le **conducteur** de S, noté $c(S)$, par $c(S) = f(S) + 1$.
- Soit $X \subset S$. On dit que X est un **ensemble générateur** de S si tout élément de S peut s'écrire sous la forme d'une somme d'éléments de X i.e. $\forall y \in S, \exists ((a_i, x_i))_{i \in [\![0,n]\!]} \in (\mathbb{N} \times X)^n / y = \sum_{i=0}^{n} a_i x_i$.
- Soit $x \in S \setminus \{0\}$. On dit que x est **irréductible** s'il ne peut pas être exprimé comme la somme de deux éléments non nuls de S i.e. $\nexists (a, b) \in (S \setminus \{0\})^2 / x = a + b$. On note $Irr(S)$ l'ensemble des irréductibles de S.
- On définit la **dimension d'incorporation** de S, notée $e(S)$, par $e(S) = \text{card}(Irr(S))$

Soit $x \in S$. On note $S^x = S \setminus \{x\}$.

III) Propriétés

(4) ***Lemme :*** L'ensemble $Irr(S)$ est le plus petit ensemble générateur de S.

Démonstration : Admise

(5) ***Proposition :***

- $m(S) \leqslant g(S) + 1$
- $c(S) \leqslant 2g(S)$
- $x \in Irr(S) \Rightarrow x \leqslant c(S) + m(S) - 1 \leqslant 3g(S)$

Démonstration :

- On a deux cas :
  - Si $\overline{S} = [\![1, g(S)]\!]$, alors $m(S) = g(S) + 1$.
  - Si $\overline{S} \subset [\![1, n]\!]$ avec $n > g(S)$.

    Alors il existe $x_0 \in [\![1, n]\!]$ tel que $x_0 \in S$.

    Alors l'ensemble $S \cap [\![1, n]\!]$ est un ensemble d'entiers positifs non nuls. Alors il possède un minimum.

    Ce minimum est $m(S)$ par définition. Ainsi, $m(S) \leqslant g(S) + 1$.
- Soit $x \in [\![0, f(S)]\!]$. Alors $f(S) - x \in [\![0, f(S)]\!]$.

$x + (f(S) - x) = f(S) \notin S$ et S stable par somme donc $x \notin S$ ou $f(S) - x \notin S$.

Ainsi, au moins la moitié des éléments de $[\![0, f(S)]\!]$ ne sont pas dans S et $g(S)$ éléments de $\mathbb{N}$ ne sont pas dans S.

Donc $g(S) \geqslant \dfrac{f(S)+1}{2}$. Ainsi, $f(S) + 1 \leqslant 2g(S)$. Alors $c(S) \leqslant 2g(S)$

- L'inégalité de gauche est admise. Celle de droite découle des deux précédentes.

> (6) ***Proposition :*** $\forall g \in \mathbb{N}$, il existe un nombre fini de semigroupes numériques de genre g. On note n_g ce nombre.

Démonstration : Si $g = 0$, on n'a qu'un seul semigroupe numérique à 0 trous : $\mathbb{N}$ lui-même. Supposons $g \neq 0$.

D'après la proposition précédente, on a $c(S) \leqslant 2g$ donc $f(S) \leqslant 2g - 1$.

On a S entièrement caractérisé par $\overline{S}$ et $\overline{S} \subset [\![1, f(S)]\!] \subset [\![1, 2g-1]\!]$.

Comme $[\![1, 2g-1]\!]$ est fini, alors il existe un nombre fini d'ensembles $\overline{S} \subset [\![1, 2g-1]\!]$, donc il existe un nombre fini de semigroupes numériques de genre g.

> (7) ***Proposition :***
>
> $$S^x \text{ est un semigroupe numérique.} \Leftrightarrow x \in Irr(S)$$

Démonstration : Admise

IV) Degré de décomposition

> (8) ***Définition :*** $\forall x \in \mathbb{N}$, on pose :
>
> $$D_S(x) = \{y \in S \mid x - y \in S \text{ et } 2y \leqslant x\}$$
>
> On définit le **degré de décomposition de x**, noté $d_S(x)$ par $d_S(x) = \operatorname{card}(D_S(x))$.

Explication :

Soit $y \in D_S(x)$. Posons $z = x - y \in S$.

On a $x = y + z$ et $2y \leqslant x$ donc $y \leqslant z$.

Soit $D'_S(x) = \{(y,z) \in S^2 \mid x = y + z \text{ et } y \leqslant z\}$.

Alors $D_S(x) = \{y \mid (y,z) \in D'_S(x)\}$

Donc $d_S(x)$ donne le nombre de manières de décomposer x sous la forme d'une somme de deux éléments de S. L'appellation "degré de décomposition" est donc justifié.

> (9) ***Lemme :*** Soit $x \in \mathbb{N}$. Alors on a :
>
> $$d_S(x) \leqslant 1 + \left\lfloor \frac{x}{2} \right\rfloor$$
>
> Avec égalité si $S = \mathbb{N}$.

Démonstration :

Comme $D_S(x) \subset [\![0, \lfloor \frac{x}{2} \rfloor]\!]$, on a $d_S(x) \leqslant 1 + \left\lfloor \dfrac{x}{2} \right\rfloor$.

Si $S = \mathbb{N}$, $D_S(x) = [\![0, \lfloor \frac{x}{2} \rfloor]\!]$ et $d_S(x) = 1 + \left\lfloor \dfrac{x}{2} \right\rfloor$.

> (10) ***Proposition :*** Soit $x \in N^*$.
>
> - $x \in S \Leftrightarrow d_S(s) > 0$
> - $x \in Irr(S) \Leftrightarrow d_S(x) = 1$

Démonstration : Conséquence directe de la définition de $d_S(x)$.

> (11) ***Proposition :*** Soient $x \in S$ et $y \in N^*$.
>
> $$d_{S^x}(y) = \begin{cases} d_S(y) - 1 & \text{si } y \geqslant x \text{ et } d_S(y-x) > 0 \\ d_S(y) & \text{sinon} \end{cases}$$

Démonstration : Conséquence directe de $D_{S^x}(y) = D_S(y) \setminus \{y - x, x\}$.

> (12) ***Proposition :*** Soit $G \in \mathbb{N}^*$ et S un semigroupe numérique de genre $0 < g(S) \leqslant G$.
>
> Alors S est entièrement caractérisé par le vecteur $\delta_S = (d_S(i))_{i \in [\![0,3G]\!]}$.
>
> Plus précisément, on peut obtenir $c(S)$, $g(S)$, $m(S)$ et $Irr(S)$ grâce à δ_S.

Démonstration :

- $c(S)$: On a montré que $c(S) \leqslant 2g(S) \leqslant 2G$. Ainsi, on peut trouver :

$$c(S) = 1 + \max\{i \in [\![0, 3G]\!] \mid d_S(i) = 0\}$$

- $g(S)$: Comme tout élément de $\overline{S}$ est inférieur à $c(S)$, ils sont tous inférieurs à $3G$. On a alors :

$$g(S) = \text{card}\{i \in [\![0, 3G]\!] \mid d_S(i) = 0\}$$

- $m(S)$: On a montré que $m(S) \leqslant g(S) + 1$ donc $m(S) \leqslant 3G$. Ainsi :

$$m(S) = \min\{i \in [\![1, 3G]\!] \mid d_S(i) > 0\}$$

- $Irr(S)$: On a montré que tout élément de $Irr(S)$ est inférieur à $3g(S) \leqslant 3G$. Ainsi, on a :

$$Irr(S) = \{i \in [\![1, 3G]\!] \mid d_S(i) = 1\}$$

Deuxième partie : Un premier programme pour calculer n_g

I) Définition de types

Tout d'abord, on définit le type `sgn`, pour semigroupe numérique, donnant son genre, son conducteur, sa multiplicité et l'ensemble de ses irréductibles sous la forme d'un tableau d'entiers.

```
type sgn = (*Semigroupe numérique*)
  { g : int (*Genre*)
  ; m : int (*Multiplicité*)
  ; c : int (*Conducteur*)
  ; irr : int array (*Ensemble des irréductibles*)
```

```
  }
;;
```

On définit ensuite un type `vect_sgn` pour la caractérisation par un vecteur, que l'on implémentera par un tableau d'entiers, pour avoir facilement accès à ses éléments par leurs indices.

```
type vect_sgn = int array ;; (*Caractérisation par un vecteur*)
```

II) Fonctions pour passer d'un type à l'autre

1) `vect_sgn` vers `sgn`

On souhaite écrire une fonction `vect_to_sgn` prenant en argument un vecteur et renvoyant le semigroupe numérique correspondant :

```
vect_to_sgn : vect_sgn -> sgn
```

Pour ce faire, on écrit 4 `genre`, `multiplicite`, `conducteur` et `irreductibles`, chacune prenant un vecteur en argument et renvoyant respectivement le genre, la multiplicité, le conducteur et l'ensemble des irréductibles du semigroupe numérique représenté par le vecteur.

```
genre : vect_sgn -> int
multiplicite : vecct_sgn -> int
conducteur : vect_sgn -> int
irreductible : vect_sgn -> int array
```

On utilise les éléments de la démonstration de la proposition (12) pour écrire les fonctions voulues :

```
let genre vect =
  let n = Array.length vect in
  let compteur = ref 0 in
  for i=0 to n-1 do
    if vect.(i) = 0 then compteur := !compteur + 1
  done;
  !compteur
;;

let multiplicite vect =
  let n = Array.length vect in
  let rec aux i =
    if i >= n then failwith "Ce cas ne devrait pas arriver." else
    if vect.(i) > 0 then i else aux (i+1)
  in aux 1
;;
```

```
let conducteur vect =
  let n = Array.length vect in
  let rec aux i =
    if i <= 1 then failwith "Ce cas ne devrait pas arriver." else
    if vect.(i-1) = 0 then i else aux (i-1)
  in aux n
;;

let irreductibles vect =
  let n = Array.length vect in
  let compteur = ref 0 in
  for i=1 to n-1 do
    if vect.(i) = 1 then compteur := 1 + !compteur
    done;
  let tab = Array.make !compteur 0 in
  for i=n-1 downto 1 do
    if vect.(i) = 1 then (compteur := !compteur - 1 ; tab.(!compteur) <- i)
  done;
  tab
;;
```

Explications :

- La fonction `genre` :

  La fonction `genre` compte le nombre de trous de S. Pour cela, elle parcourt le vecteur, en incrémentant un compteur lorsqu'un élément n'appartenant pas à S est rencontré, c'est-à-dire lorsqu'on rencontre un élément valant 0 dans le vecteur (propositions (10) et (12)).

  Le compteur est implémenté par une référence d'entier pour assurer le caractère global à la fonction et son caractère mutable.

  On rappelle que pour un vecteur `vect` donné en entrée, l'élément `vect.(i)` correspond à $d_S(i)$. Ainsi, si `vect.(i)` est nul, alors l'élément i n'est pas dans S.

  Sa complexité est en $O(n)$.

- La fonction `multiplicite` :

  La fonction `multiplicite` cherche le plus petit élément non nul de `vect`. Pour cela, on crée une fonction auxiliaire `aux`, récursive terminale, qui itère une variable `i`, parcourant le vecteur, et renvoyant la plus petite valeur de `i` telle que `vect.(i)` soit positif, c'est-à-dire le premier i tel que $i \in S$, soit $d_S(i) > 0$ (proposition (10)).

  La fonction `aux` termine grâce au cas `i >= n`, mais terminerait de toute façon car si la fonction est appelée avec des arguments légaux, elle devrait renvoyer une valeur avant d'arriver à ce cas.

  Sa complexité est en $O(n)$.

- La fonction `conducteur` :

La fonction `conducteur` fonctionne de manière similaire.

Elle recherche le plus grand `i` tel que `vect.(i-1)` est nul. Pour cela, une fonction auxilaire récursive terminale itère en décroissant sur `i` et renvoie le plus grand i tel que $i-1 \notin S$ donc $d_S(i-1)=0$ (proposition (10)).

La fonction `aux` termine grâce au cas `i <= 0`, mais terminerait de toute façon car si la fonction est appelée avec des arguments légaux, elle devrait renvoyer une valeur avant d'arriver à ce cas.

Sa complexité est en $O(n)$.

- La fonction `irreductibles` :

  La fonction `irreductibles` construit l'ensemble des irréductibles de S, sous la forme d'un tableau d'entiers. Pour ce faire, il compte tout d'abord le nombre d'irréductibles de S afin de créer un tableau de la bonne taille. Il crée alors le tableau en question, et reparcourt le vecteur en ajoutant les éléments irréductibles dans le tableau.

  Pour compter le nombre d'irréductibles, on incrémente une référence `compteur` dès qu'un irréductible est rencontré, donc dès qu'un élément égal à 1 est rencontré dans `vect` car si $d_S(i)=1$ si et seulement si c'est un irréductible (proposition (10)).

  On peut alors créer le tableau des irréductibles de la bonne taille (la valeur de `compteur`). Pour insérer les éléments dans le tableau, on parcourt `vect` en décroissant. Dès que l'on rencontre un irréductible, on décrémente le compteur, avant d'insérer l'élément considéré à l'indice de la nouvelle valeur du compteur. Le tableau ainsi rempli est donc trié dans l'ordre croissant, puisqu'on ajoute les éléments en ordre décroissant aux indices décroissants.

  Sa complexité est en $O(n)$.

Ayant défini ainsi ces 4 fonctions, nous pouvons construire la fonction voulue :

```
let vect_to_sgn delta =
  { g = genre delta
  ; m = multiplicite delta
  ; c = conducteur delta
  ; irr = irreductibles delta
  }
;;
```

La fonction `vect_to_sgn` renvoie simplement un objet de type `sgn`, associant à chaque caractéristique du semigroupe numérique l'appel à la fonction qui lui est associée.

Sa complexité est ainsi en $O(n)$, ne faisant appel qu'à des fonctions en $O(n)$, avec n la taille du vecteur `delta`.

2) `sgn` vers `vect_sgn`

On souhaite écrire une fonction `sgn_to_vect` prenant en argument un semigroupe numérique et renvoyant le vecteur correspondant :

```
sgn_to_vect : sgn -> vect_sgn
```

Pour cela, écrivons une fonction permettant de calculer le degré de décomposition d'un élément x dans S :

```
degre_decomposition : sgn -> int -> int
```

Il nous faut pour cela une fonction déterminant si un entier x appartient ou non à S :

```
appartient : sgn -> int -> bool
```

On obtient alors la fonction suivante :

```
exception Trouve

let rec appartient s x =
  if Array.mem x s.irr || x = 0 then true else
    try
      for i=0 to Array.length s.irr - 1 do
        if x > s.irr.(i) && appartient s (x-s.irr.(i)) then raise Trouve
      done;
      false
    with
    | Trouve -> true
;;
```

Cette fonction récursive teste si un élément x appartient ou non à un semigroupe numérique S :

- Si x est dans l'ensemble des irréductibles de S ou que $x = 0$, alors x est dans S.
- Sinon, on parcours le tableau des irréductibles de S, et pour tout irréductible $a > x$ de S, on teste si $x - a$ est dans S. Si $x - a \in S$, alors $x \in S$.
    - Si on a parcouru tout le tableau des irréductibles sans succès, alors $x \notin S$ et on envoie `false`.
    - Sinon, on a trouvé une décomposition qui convient, alors on arrête le programme dès qu'on a trouvé cette décomposition grâce à une exception, et alors on renvoie `true`

Cette fonction termine car le est `x > s.irr.(i)` permet de n'appeler récusrivement la fonction que sur des arguments `x` positifs, de plus les cas de base `Array.mem x s.irr` et `x = 0` garantissent la terminaison de la fonction, en ajoutant le fait que si `x` est assez petit, on parcourera la boucle sans succès et la fonction terminera en renvoyant `false`.

Sa complexité est en $O(e(S) \times x)$.

On peut, grâce à la fonction `appartient` écrire la fonction suivante :

```
let degre_decomposition s x =
  if not (appartient s x) then 0 else
```

```
  if Array.mem x s.irr then 1 else
  let compteur = ref 1 in
  for y=1 to x/2 do
    if appartient s y && appartient s (x - y) then compteur := !compteur + 1
  done;
  !compteur
;;
```

La fonction `degre_decomposition` renvoie le degré de décomposition d'un élément x dans S, passés en argument :

- Si $x \notin S$, alors $d_S(x) = 0$.
- Si $x \in Irr(S)$, alors $d_S(x) = 1$.
- Sinon, la fonction incrémente un compteur (un référence d'entier), chaque fois qu'elle trouve une décomposition possible de x dans S, c'est-à-dire un nombre $y \leqslant \left\lfloor \frac{x}{2} \right\rfloor$ tel que $(y, x-y) \in S^2$.

Sa complexité est en $O(e(S) \times x^2)$.

Avec cette fonction, on peut aisément écrire la fonction voulue :

```
let sgn_to_vect s =
  let delta = Array.make (3*gg + 1) (-1) in
  delta.(0) <- 1 ;
  for i=1 to 3*gg do
    delta.(i) <- degre_decomposition s i
  done;
  delta
;;
```

La fonction prend en argument un objet `s` de type `sgn` et renvoie le vecteur `delta`, de type `vect_sgn` correspondant à `s`.

Pour cela, il crée le tableau `delta` de taille `3*gg`, avec `gg` correspondant à G (les majuscules étant proscrites pour les noms de variables en OCaml) (proposition (12)) et le remplit, appelant la fonction `degre_decomposition` définie précédemment.

Sa complexité est en $O(e(S) \times G^3)$.

II) Arbre des semigroupes numériques

Pour compter le nombre de semigroupes numériques, nous allons construire l'arbre des semigroupes numériques définit comme suit.

On place l'ensemble $\mathbb{N}$ à la racine, semigroupe numérique de genre 0. On fixe un genre $g \in \mathbb{N}$ et on construit l'arbre :

- Si un semigroupe donné est de genre g, alors c'est une feuille.
- Si un semigroupe donné est de genre strictement inférieur à g, alors c'est un nœud interne, et on définit ses fils comme tous les semigroupes numériques S' tels que $\text{card}(S \setminus S') = 1$ donc

$$g(S') = 1 + g(S).$$

On définit alors une première fonction prenant en argument un semigroupe numérique S et un élément x et qui renvoie le semigroupe numérique $S' = S^x$.

```
enlever : sgn -> int -> sgn
```

On écrit alors cette fonction :

```
let enlever s x =
  let vect_s' = sgn_to_vect s in
  for y=(Array.length vect_s' - 1) downto x do
    if vect_s'.(y - x) > 0 then vect_s'.(y) <- vect_s'.(y) - 1
  done;
  vect_to_sgn vect_s'
;;
```

On utilise la proposition (11) pour construire le vecteur associé à S^x.

Pour cela, on modifie une copie du vecteur correspondant à S afin de créer celui correspondant à S'. On parcourt tous les indices du tableau supérieurs à x, et si $y - x \in S$, on décrémente la case du tableau correspondante.

Afin de pouvoir tester si $y - x \in S$ sans utiliser `appartient` ou créer un autre vecteur, on parcourt les indices en ordre décroissant. Ainsi, la case considérée n'a pas encore été visitée donc modifiée.

Pour finir, on appelle la fonction `vect_to_sgn` pour transformer le vecteur en son semigroupe numérique associé.

Sa complexité est en $O(G)$.

On peut alors écrire une fonction prenant en argument un semigroupe numérique et renvoyant les fils de ce semigroupe numérique.

```
fils : sgn -> sgn list
```

On a alors cette fonction :

```
let fils s =
  let liste = ref [] in
  for i=0 to Array.length s.irr - 1 do
    if s.irr.(i) >= s.c then liste := (enlever s s.irr.(i)) :: !liste
  done;
  !liste
;;
```

On sait d'après la proposition (7) que S^x est un semigroupe numérique si et seulement si $x \in Irr(S)$. Ainsi, la fonction parcourt l'ensemble des irréductibles de S, et ajoute à la liste S^x pour tout $x \in Irr(S)$.

On ne considère que les éléments irréductibles supérieurs au conducteur car si on ne le faisait pas, on retrouverait plusieurs fois le même semigroupe numérique à des endroits différents de l'arbre. Or, on souhaite qu'un semigroupe numérique donné ne figure qu'une unique fois dans l'arbre.

Sa complexité est en $O(e(S) \times G)$.

Ainsi, on peut aisément construire l'arbre des semigroupes numériques.

Définissons tout d'abord un type `arbre` :

```
type arbre =
| F of sgn
| N of sgn * arbre list
;;
```

Un arbre est donc soit une feuille `F` ayant pour étiquette un semigroupe numérique, soit un nœud interne `N` ayant également pour étiquette un semigroupe numérique, associé à la liste de ses fils.

Avec ce type, on cherche à écrire une fonction construisant l'arbre des semigroupes numériques :

```
construire_arbre : int -> arbre
```

On peut écrire cette fonction comme suit :

```
let construire_arbre n =
  let s0 = {g = 0 ; m = 1 ; c = 0 ; irr = [|1|]} in
  let rec aux n s =
    if n = 0 then F s else
      N (s, List.map (aux (n-1)) (fils s))
  in aux n s0
;;
```

On commence tout d'abord par construire `s0`, la racine de l'arbre, le semigroupe numérique $\mathbb{N}$.

On crée ensuite une fonction auxiliaire `aux` prenant en argument un entier `n` et un semigroupe numérique `s` et renvoyant l'arbre de racine `s` et de hauteur `n`. Ainsi, si `n = 0`, on crée la feuille d'étiquette `s`. Sinon, on crée un nœud interne d'étiquette `s` et appelant récursivement la fonction `aux` avec l'argument `n-1` sur chacun des fils de `s`.

La fonction `aux` termine car `n` est un entier positif strictement décroissant à chaque appel récursif.

Sa complexité est en $O(n^3 \times G)$.

III) Décompte du nombre de semigroupe numérique de genre g

On souhaite écrire deux fonctions : l'une comptant le nombre de semigroupes de genre g fixé, l'autre comptant le nombre de semigroupes de genre inférieur ou égal à g fixé.

```
compter_sgn_n : int -> int
compter_sgn_to_n : int -> int array
```

On utilise pour ces deux fonctions un parcours en profondeur de l'arbre des semigroupes numériques.

```
let compter_sgn_n n =
  let arbre = construire_arbre n in
  let compteur = ref 0 in
  let rec aux a =
    match a with
    | F s -> if s.g = n then compteur := !compteur + 1
    | N (_, l) -> List.iter aux l
  in aux arbre ; !compteur
;;

let compter_sgn_to_n n =
  let arbre = construire_arbre n in
  let ng = Array.make (n+1) 0 in
  let rec aux a =
    match a with
    | F s -> ng.(s.g) <- ng.(s.g) + 1
    | N (s, l) -> ng.(s.g) <- ng.(s.g) + 1 ; List.iter aux l
  in aux arbre ; ng
;;
```

Dans la première fonction, une référence d'entier est incrémentée chaque fois qu'une feuille ayant pour étiquette un semigroupe numérique de genre g.

Dans la seconde fonction, pour chaque semigroupe numérique rencontré, on incrémente la case d'indice $g(S)$ d'un tableau comptant n_g pour tout genre inférieur ou égal à `n`.

IV) Analyse du temps d'exécution du programme

On souhaite analyser le temps d'exécution du programme, et plus précisément tracer la courbe du temps d'exécution du programme comptant le nombre de semigroupes numériques de genre inférieur ou égal à g fixé en fonction de ce même g fixé.

Pour cela, nous utiliserons le module `unix.cma`, disponible sur OCaml. Nous utiliserons une fonction en particulier :

```
Unix.gettimeofday : unit -> float
```

Cette fonction renvoie le temps en secondes écoulé depuis le 1er janvier 1970, minuit pile, avec une précision de 10^{-8} secondes. On l'utilisera pour calculer le temps d'exécution du programme.

On écrit alors une fonction `temps` qui calcule, pour tout $g \leqslant n$ le temps que met le programme pour calculer n_g.

```
temps : int -> float array
```

On a alors la fonction suivante :

```
let temps n =
  let tab_temps = Array.make (n+1) (-1.) in
  for i=0 to n do
    let time_init = Unix.gettimeofday () in
    let _ = compter_sgn_to_n i in
    tab_temps.(i) <- (Unix.gettimeofday () -. time_init)
  done;
  tab_temps
;;
```

La fonction remplit un tableau de flottants. La case d'indice `i` contiendra en fin d'exécution le temps qu'aura mis le programme pour calculer tous les n_g pour $g \leqslant i$.

On écrit une fonction `exporter` qui exporte les données dans un fichier de sortie, ce qui nous permettra de tracer la courbe du temps d'exécution en fonction de n.

```
exporter : string -> float array -> unit
```

La fonction prendra en argument une chaîne de caractères, le nom du fichier de sortie, et un tableau de flottants correspondant aux temps d'exécution du programme. On a alors la fonction suivante :

```
let exporter fichier tableau =
  let f_out = open_out fichier in
  output_string f_out ((string_of_int n) ^ "\n") ;
  for i=0 to Array.length tableau - 1 do
    output_string f_out ((string_of_float (tableau.(i))) ^ "\n")
  done;
  close_out f_out
;;
```

Sur le fichier de sortie, la première ligne contiendra un entier `n`. Sur les `n` lignes suivantes, il y aura un unique flottant : le temps d'exécution de l'appel `compter_sgn_to_n n`.

Ainsi, on peut récupérer les données pou tracer le graphe du temps d'exécution en fonction de g. Pour cela, on utilise le module `matplotlib.pyplot` du langage Python pour tracer les graphiques voulus. On a alors le programme suivant :

```python
import matplotlib.pyplot as plt

def lire(f):
    return f.readline().strip()

with open("donnees2.txt", "r") as f:
    n = int(lire(f))
    t = [0]*(n+1)
    for i in range(n+1):
        t[i] = float(lire(f))
    plt.plot(list(range(n+1)), t)
    plt.title("Temps de calcul en sec en fonction de g pour g <= 20")
    plt.savefig("graphe2.png")
    plt.show()
```

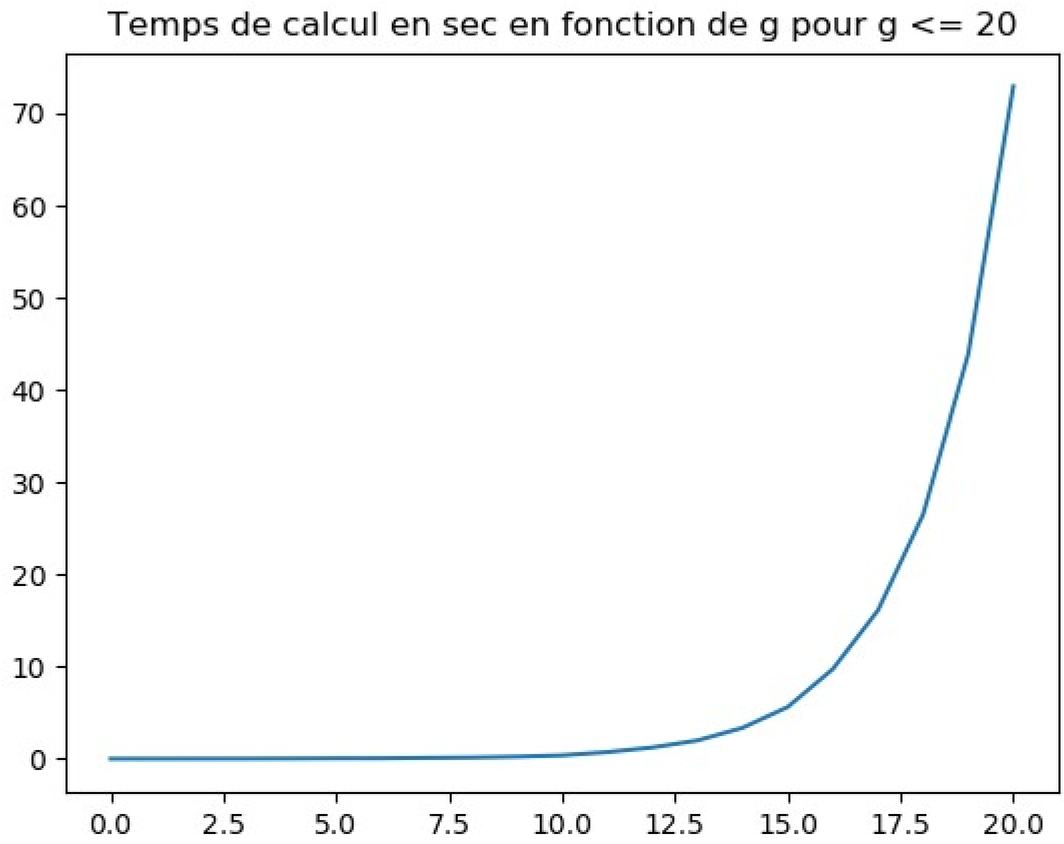
V) Performances

En lançant le programme pour $n = 20$, on obtient les résultats suivants.

g	Temps d'exécution de `compter_sng_to_n g` en secondes
0	≈ 0
1	$0,004$
2	$0,011$
3	$0,013$
4	$0,027$
5	$0,048$
6	$0,054$
7	$0,105$
8	$0,140$
9	$0,233$
10	$0,376$
11	$0,727$
12	$1,232$
13	$1,996$
14	$3,368$
15	$5,629$
16	$9,750$

g	Temps d'exécution de `compter_sng_to_n g` en secondes
17	$16,124$
18	$26,523$
19	$43,887$
20	$72,948$

Avec le programme python décrit ci-dessus, on obtient le graphique suivant :



VI) Conclusion

On remarque que la courbe semble se comporter de manière exponentielle, atteignant déjà plus d'une minute de temps d'exécution pour $g = 20$.

Ce premier algorithme semble donc ne pas être très efficace. On se confronte très rapidement à un problème de temps d'exécution lorsqu'on prend des plus grandes valeurs.

Nous allons dans la suite chercher d'autres implémentations des semigroupes numériques visant à accélérer le décompte des semigroupes numériques de genre g.

Annexes

Fonctions d'affichage

Pour s'aider dans notre travail, nous avons défini plusieurs fonctions d'affichage. Les voici, bièvement décrites :

```
open Printf (*On ouvre le module d'affichage*)

(*Fonction permettant d'afficher un vecteur*)
let print_vect vect =
  let n = Array.length vect in
  printf "(" ;
  for i=0 to n-2 do
    printf "%d, " vect.(i)
  done;
  printf "%d)\n" vect.(n-1)
;;

(*Fonction permettant d'afficher un tableau d'entiers*)
let print_int_array tab =
  let n = Array.length tab in
  printf "[" ;
  for i=0 to n-2 do
    printf "%d, " tab.(i)
  done;
  printf "%d]\n" tab.(n-1)
;;

(*Fonction permettant d'afficher un objet de type sgn*)
let print_sgn s =
  printf "Genre : %d ; Multiplicité : %d ; Conducteur : %d ; Irréductibles : " s.g s.m
  print_int_array s.irr
;;

(*Fonction permettant d'afficher une liste de sgn*)
let rec print_sgn_list l =
  match l with
  | [] -> ()
  | x :: y -> print_sgn x ; print_sgn_list y
;;

(*Fonction permettant d'afficher un arbre*)
(*Fonction très peu qualitative*)
let rec print_arbre a =
  match a with
  | F s -> printf "F : " ; print_sgn s
  | N (s, l) -> printf "N : " ; print_sgn s ; printf "L : (" ; List.iter (fun b -> pri
;;

(*Fonction affichant "g   n_g" pour tout g inférieur à n, passé en argument*)
let solution_finale n =
```

```
  let tab = compter_sgn_to_n n in
  for i=0 to n do
    if i < 10 then printf "%d    %d\n" i tab.(i) else printf "%d   %d\n" i tab.(i)
  done;
;;

(*Fonction affichant "n = g :  t_g" pour tout g inférieur à n, passé en argument, t_g
let print_temps n =
  let tab_temps = temps n in
  for i=0 to n do
    if i < 10 then printf "n = %d :  %f\n" i tab_temps.(i) else printf "n = %d : %f\n'
  done;
;;
```