From the BigInt

Vincent Lesbros
17 novembre 2020 / 21 novembre 2020 / 28 novembre 2020

Implémentation de rationnels en JavaScript à partir des BigInts

On se propose d'implémenter les fractions ou plutôt les nombres rationnels en utilisant le support des grands entiers ou '***BigInt***' de JavaScript.

Les *fractions* sont un couple de deux entiers, numérateur / dénominateur, avec le dénominateur différent de zéro. Plusieurs fractions représentent le même nombre rationnel, ou plus exactement sont dans la même classe d'équivalence sur la relation $(a, b)\ R\ (c, d) <=> ad = bc$.

3/6 et 1/2 par exemple représentent le même rationnel.

Ce que nous appelons ***fraction*** dans la suite est une représentation de rationnel sous la forme simplifiée, c'est-à-dire sans facteur commun entre le numérateur et le dénominateur et où de plus on impose que le dénominateur soit positif en plaçant le signe négatif éventuel au numérateur.

Pour un rationnel x/y représenté par le couple (x, y), on a toujours $y > 0$, avec x portant le signe.

Par exemple 3/-9 s'écrit -1/3.

Le signe de la ***fraction*** peut donc se mesurer en observant uniquement le signe de son numérateur.

Tout objet ***fraction*** est sous la forme simplifiée où le PGCD du numérateur et du dénominateur est 1.

Les BigInts

(cf. https://tc39.es/ecma262/#sec-ecmascript-language-types-bigint-type)

En résumé, les BigInts sont des valeurs primitives du langage représentant les nombres entiers, positifs et négatifs de longueur quelconque.

Création

Pour les créer, on suffixe l'écriture de l'entier par la lettre 'n'.
Par exemple :

```
0B10100n // l'entier 10100 en Binaire (0B au début) avec le suffixe 'n'
=> 20n      // donne le BigInt 20n (affiché en base dix).
```

Utilisez 0x pour l'hexadécimal, 0o pour l'octal et 0b pour le binaire. Par défaut de préfixe, c'est la base 10 qui est utilisée.
On peut également utiliser la fonction BigInt(x) pour convertir x en BigInt si x est un nombre ou une chaîne de chiffres.

```
// Attention :
BigInt(2434343434343434343434343434344545)
=>      2434343434343434490674800086199828 48n
```

BigInt("2434343434343434343434343434344545")
=> 2434343434343434343434343434344545n
// Dans le premier cas, le nombre flottant a été lu et converti avec une perte.
// Dans le second, à partir de la chaîne, la transcription est exacte.
On peut, comme pour les nombres, les encapsuler dans un objet avec Object(x).

Arithmétique

On ne connais pas la base de numération dans laquelle ils sont représentés, ce n'est pas important et peut changer selon les implémentations du langage. Ce qui importe, c'est qu'on dispose des opérations arithmétiques suivantes :
+, -, /, %, *, **
Addition, soustraction, division entière, reste de la division entière, multiplication et exponentiation. Pour l'exponentiation, l'exposant doit être un nombre positif.
Pour toutes les opérations, les opérandes doivent être des BigInts : on ne mélange pas les Numbers (nombre flottants 64 bits, https://tc39.es/ecma262/#sec-ecmascript-language-types-number-type) avec les BigInts. Dans le futur, un type 'value' est prévu pour cela.
-x : opposé de x
~x : ne change que le dernier bit
++x, x++, incrémenter la variable (pas la valeur !)
--x, x--, décrémenter la variable (idem)
<<,>> décalages en base 2, (>>> considère un décalage non signé et n'est pas applicable aux BigInts)

Comparateurs, relation d'ordre

<,>, <=, >=
À la différence des opérations arithmétiques, les nombres flottants, et même les chaînes sont comparables avec les BigInts.
On peut savoir par exemple, si un *BigInt* est représentable sans erreur par un *Number* :
Number.MAX_SAFE_INTEGER
=> 9007199254740991
Number.MIN_SAFE_INTEGER
=> -9007199254740991
x >= Number.MIN_SAFE_INTEGER && x <= Number.MAX_SAFE_INTEGER
=> true / false
// NOTE les comparateurs peuvent ramener true, false ou undefined !
// undefined dans le cas de comparaison avec NaN (Not a Number).

Comparateurs égalité et identité

== et != effectuent les conversions
=== et !== ne les font pas.

9007199254740991 == 9007199254740991n
=> true
9007199254740991 === 9007199254740991n
=> false

// Attention aux encapsulations :
x = Object(-1000n)
=> BigInt {-1000n}
x === -1000n
=> false

x.valueOf() === -1000n
=> true

Définition de la classe Fract

Fract étant le nom que l'on a donné au fractions dans ce système.

Page d'expérimentation

https://www.cyclonium.com/atelier/nombres/fromTheBigInt.html

Pour voir le code et expérimenter les formules, utilisez F12 sous Chrome ou Firefox.

Définition

Voici le début du code :

```
//// Fractions sur les big numbs
class Fract {
	constructor(numérateur = 0n, dénominateur = 1n) {
		this._numérateur = BigInt(numérateur);
		this._dénominateur = BigInt(dénominateur);
		this.simplifier();
	}

	static get Zéro() {
		if (Fract._Zéro === undefined) {
			Fract._Zéro = new Fract();
		}
		return Fract._Zéro;
	}
	static get Unité() {
		if (Fract._Unité === undefined) {
			Fract._Unité = new Fract(1n);
		}
		return Fract._Unité;
	}

	simplifier() {
		if (this._dénominateur < 0n) {
			this._dénominateur = -this._dénominateur;
			this._numérateur = -this._numérateur;
			// le signe négatif toujours au numérateur
		}
		let s = this.signe;
		let p = bigPGCD(s * this._numérateur, this._dénominateur);
		this._numérateur = this._numérateur / p;
		this._dénominateur = this._dénominateur / p;
		if (this._numérateur == 0n) {
			this._dénominateur = 1n;
		}
	}
```

...

Création

La création de la fraction se fait en donnant le numérateur et le dénominateur.
Le constructeur simplifie la fraction : en réduisant au plus simple et en imposant le signe (positif ou négatif au numérateur).
Pour réduire la fraction, on utilise la fonction `bigPGCD` :

```
/**
 * PGCD de deux grands entiers non négatifs, Algorithme Knuth Vol. 2 p 237
 * @param {bigint} u : entier non négatif
 * @param {bigint} v : entier non négatif
 * @returns {bigint} : PGCD
 */
function bigPGCD(u, v) {
	if (u < 0n || v < 0n) throw "bigPGCD sur les entiers positifs";
	while (v !== 0n) {
		let r = u % v;
		u = v;
		v = r;
	}
	return u;
}
```

La fonction comporte un *while*, et le temps de calcul peut être long. Mais si on crée une fraction, on obtiendra la forme simplifiée : pas de facteur commun entre le numérateur et le dénominateur.

Les deux méthodes statiques (méthodes de classe), `Zéro et Unité`, permettent d'éviter de recréer plusieurs fois les fractions 0/1 et 1/1.

Création depuis une chaîne

```
/// Ajouter aux Strings une méthode de lecture des fractions en big numbs
Object.defineProperty(String.prototype, "Fract",
{
	get: function () {
		let s = this.split('/');
		let num = BigInt(s[0]);
		let den = BigInt(s[1]);
		return (new Fract(num, den));
	},
	enumerable: false,
	configurable: true
	});
```

L'idée est de saisir une fraction depuis son écriture en base dix classique.
Par exemple : 1/3 s'écrira :
"1/3".Fract
=> Fract {_numérateur: 1n, _dénominateur: 3n}
La chaîne est séparée en deux par le '/', et les deux parties sont transformées en BigInt*s*.

Opérateurs sur les fractions

Malheureusement, JavaScript ne permet pas de redéfinir les opérateurs. Le langage permet de définir le comportement d'un objet en tant que « primitive », mais pas de définir l'action des opérateurs.

Heureusement, les propriétés peuvent être des chaînes de caractères quelconques (ou des '*Symbols*').

Et on peut définir les méthodes '+', '-', '/', ... en utilisant les chaînes de caractères "+", "-", "/", ... comme nom de propriété directement dans la classe.

D'où la suite du code (dans la classe) :

```
"+"(f) {
	return new Fract(
		this._numérateur * f._dénominateur + f._numérateur * this._dénominateur,
```

```
                this._dénominateur * f._dénominateur);
}

"*"(f) {
        return new Fract(this._numérateur * f._numérateur,
                this._dénominateur * f._dénominateur);
}

"/"(f) {
        return new Fract(this._numérateur * f._dénominateur,
                this._dénominateur * f._numérateur);
}

"-"(f) {
        return new Fract(
                this._numérateur * f._dénominateur - f._numérateur * this._dénominateur,
                this._dénominateur * f._dénominateur);
}

"%"(f) {
        let fdiv = this["/"](f);
        return fdiv.frac()["*"](f);
}
```

Donc, on a défini les quatre opérations, mais cela impose une petite gymnastique syntaxique pour les utiliser :
Si A et B sont deux fractions, au lieu d'écrire simplement A + B pour faire la somme,
on écrira A["+"](B)
Et de même pour tous nos 'opérateurs'.

L'opérateur %

Pour l'écrire, on utilise la division des fractions et la décomposition de la fraction en sa partie entière et 'partie fractionnaire'.
La partie entière est le résultat de la division entière du numérateur par le dénominateur.
La partie fractionnaire est se qui reste de la fraction quand on lui retranche sa partie entière, c'est-à-dire une fraction comprise entre 0 et 1.

Modulo

En pus de l'opérateur %, on écrit une méthode modulo, ramenant une valeur toujours positive.

```
modulo(f) {
        let c = this["%"](f);
        if (c._numérateur < 0n) {
                if (f._numérateur > 0n)
                        return c["+"](f);
                else
                        return c["-"](f);
        }
        return c;
}
```

Surcharge de toString

Pour récupérer une chaîne de caractères représentant la fraction, une première forme est une surcharge de la méthode ***toString***.

```
x= new Fract(100, 13)
Fract {_numérateur: 100n, _dénominateur: 13n}
x.toString(2) // en base 2
"1100100/1101"
x.toString()
"100/13"
```

Extensions

Si on souhaite développer l'écriture de la fraction dans une base donnée, par exemple l'extension décimale, une première méthode calcule :
1/ la chaîne représentant la partie entière
2/ la suite de chiffres de la partie fractionnaire dans la base donnée.
Note : la méthode utilise la méthode *toString* des BigInt*s* donc la base doit être entre 2 et 36.

```
// ramène la chaîne de caractère représentant l'extension décimale ou dans une autre base
// utilise le toString des BigInt donc la base est limité
// entre 2 et 36 (10 chiffres plus 26 lettres)
                    extension(base = 10, précision = 100) {
                           let s = this.signe;
                           let p = this.abs();
                           let b = new Fract(base); // en base b

                           if (s == 0n) return "0";

                           let e = p.ent(); // partie entière : x/1
                           let pe = e._numérateur.toString(base); // base de 2 à 36.
                           if (s == -1n) pe = "-" + pe; // négatif

                           let f = p.frac(); // partie fractionnaire
                           if (f._numérateur == 0n) return pe; // un entier

                                  // partie fractionnaire non nulle
                           pe += "."; // point décimal

                           for (let k = 0; k < précision; ++k) {
                                  f = f["*"](b); // multiplie par la base
                                  e = f.ent();
                                  pe += e._numérateur.toString(base); // un chiffre
                                  f = f.frac(); // le reste
                                  if (f._numérateur == 0n) return pe; // reste 0
                           }
                           return pe;
                    }
```

On ne détecte pas les cycles, et on s'arrête dès que le reste est nul ou dès que la précision demandée est atteinte.

Méthode intuitive de détection des cycles

La mal-nommée méthode '*division*' ramène un objet décrivant la fraction avec son signe, sa mantisse entière, et les chiffres après le point décimal. Si lors de cette division et avant d'atteindre la précision demandé, on retrouve un reste déjà rencontré, on s'arrête et on note le point de cycle des chiffres.

```
                    division(base = 10, précision = 15000000)
                    {
                           let s = this.signe;
                           let p = this.abs();
                           let b = BigInt(base);
                           if (s == 0n) return ["0"];
                           let e = p._numérateur / p._dénominateur; // division entière
                           let mantisseEntière = e.toString(base).split("");
                           let reste = p._numérateur % p._dénominateur;
                           let décimales = [];
                           let rang = 0;
                           let mémoire = new Map();
                           mémoire.set(reste, rang);
                           while (rang < précision) {
```

```
                        reste *= b;
                        let div = reste / p._dénominateur;
                        décimales.push(div.toString(base));
                        ++rang;
                        reste = reste % p._dénominateur;
                        if (mémoire.has(reste)) {
                                return {
                                        signe: s,
                                        mantisseEntière: mantisseEntière,
                                        décimales: décimales,
                                        cycle: mémoire.get(reste),
                                        rang: rang
                                };
                        }
                        mémoire.set(reste, rang);
                }
                return {
                        signe: s,
                        mantisseEntière: mantisseEntière,
                        décimales: décimales,
                        précisionAtteinte: rang
                };
        }


// Exemple :
x = new Fract(-103, 12)
=> Fract {_numérateur: -103n, _dénominateur: 12n}
x.division()
=> {signe: -1n, mantisseEntière: Array(1), décimales: Array(3), cycle: 2, rang: 3}
        cycle: 2
        décimales: (3) ["5", "8", "3"]
        mantisseEntière: ["8"]
        rang: 3
        signe: -1n

x.extension()
=> "-8.583333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333"
// le rang 3 indique que c'est au rang 3 que le cycle à été détecté. le cycle 2 indique que le cycle
// commence à l'indice 2 dans les décimales, ici la case contenant "3".
// Le signe est 0n, 1n ou -1n selon que la fraction est nulle, positive ou négative.
```

Attention

"78078077/780780783".Fract.division(10, 1000000000)
{signe: 1n, mantisseEntière: Array(1), décimales: Array(14453318), cycle: 0, rang: 14453318}

À partir de nombres relativement *petits* on arrive vite à des cycles de « grande longueur ». Comme on doit mémoriser les restes successifs des divisions pour détecter le cycle, la mémoire peut vite être débordée (pour l'instant sans moyen de trapper la faute de mémoire).
Dans l'exemple ci-dessus, le cycle fait 14 453 318 chiffres.

Comparaisons de fractions

Si on se base simplement sur le signe de la différence, le code est le suivant :

```
// comparaisons
                "=="(f) {
                        return this._numérateur == f._numérateur &&
                                this._dénominateur == f._dénominateur;
                }

                "<"(f) {
```

```
        return this["-"](f).signe == -1n;
    }
    ">"(f) {
        return  this["-"](f).signe == 1n;
    }
    "<="(f) {
        let s = this["-"](f).signe;
        return s == -1n || s == 0n;
    }
    ">="(f) {
        let s = this["-"](f).signe;
        return s == 1n || s == 0n;
    }
```

Si on veut économiser du temps en évitant une construction de fraction avec calcul de PGCD, on peut se baser sur la comparaison des produits *ad* et *cb*:

```
// comparaisons
    "<"(f) {
        return this._numérateur * f._dénominateur
            < f._numérateur * this._dénominateur;
    }
    ">"(f) {
        return this._numérateur * f._dénominateur
            > f._numérateur * this._dénominateur;
    }
    "<="(f) {
        return this._numérateur * f._dénominateur
            <= f._numérateur * this._dénominateur;
    }
    ">="(f) {
        return this._numérateur * f._dénominateur
            >= f._numérateur * this._dénominateur;
    }
```

Décompositions

On se propose de décomposer une fraction positive ou nulle en une somme de fractions de la forme suivante : $n_0/1 + 1/n_1 + 1/n_2 + ... + 1/n_k$ avec les n_i entiers.

On suppose que cette série est finie quelque soit la fraction initiale.

Dans la littérature il s'agit de *fractions égyptiennes*.

Procédé

1°) On décompose la fraction en deux parties : la partie entière de la forme *n/1* et la partie fractionnaire de la forme *f = x/y.*

On ajoute n/1 à la liste.

2°) Tant que *f* n'est pas nulle, on itère sur les point suivants :

a) l'inverse de *f* est *y/x*. On calcule *pe* la partie entière de *y/x* et *pf* la partie fractionnaire de *y/x*.

b) à partir de *pe* et *pf* on calcule un diviseur *d* :

si *pf* est nulle, *d = pe* = le numérateur de *pe* puisque *pe* est entier.

si *pf* n'est pas nulle *d = pe + 1*.

c) *On ajoute 1/d à la liste.*

On retranche 1/d de f :

f = f - 1/d

3°) f est nulle, on arrête en ramenant la liste.

Exemples :

10238/89

= 115/1 + 1/30 + 1/2670

78078077/780780783

= 0/1 + 1/11 + 1/111 + 1/12211 + 1/198331080 + 1/53791198247909429 + 1/5194978686149597897279601551416052 + 1/3118933362243470239643578265524237972782568097856998197289973308955 95 + 1/1405319858514711711319614153798514240350539707607796447391548914717204339991016315292874262907979137760831380239798702586784652244360787 78 + 1/1415362131727330028712949760299171578634526334543643455621544375694555604898332262056455652103471355325262168028363923027880258468934013872310172937643826658419597568239256051564706478876373600061965813704528694249543855362964489608462171310349616983349851444473483620213964 24

Variante originale

On se propose de décomposer les fractions en une somme de fractions dont les dénominateurs sont des nombres premiers. Pour simplifier on ne considère toujours que les fractions positives.

1°) on sépare la partie entière du reste fractionnaire. La fraction est alors comprise entre [0 et 1[.

La première fraction de la somme à la forme x / 1.

2°) On décompose la partie fractionnaire x/y restante en choisissant p, le plus petit nombre premier tel que 1/p soit inférieur à x/y. Et on réitère avec f = x/y - 1/p.

La série est-elle finie ?

Les fractions se retrouvent séparées en deux classes :

- Celles dont la décomposition est finie

par exemple 1/3, trois est premier et la décomposition est terminée.

pour 9/14, on trouve 1/2 + 1/7.

- Celles dont la décomposition est infinie.

Par exemple, 1/4 = 0/1 + 1/5 + 1/23 + 1/157 + 1/6569 + ...

et il reste encore 39/474413180 à décomposer...

Ou encore 2/3 = 0/1 + 1/2 + 1/7 + 1/43 + 1/1811 + 1/654149 + ...

et il reste 79/2139502893234 à décomposer.

Les dénominateurs des derniers restes calculés ne sont pas premiers, et comme l'algorithme effectue une soustraction de fraction, impliquant un produit des dénominateurs, il ne seront jamais premier et la série est infinie.

Variante « périgourdine »

On peut raccourcir la série si on admet des numérateurs > 1, tout en imposant aux dénominateurs d'être premiers. De plus on fixe une limite à la décomposition en indiquant le plus grand nombre premier utilisable pour ne pas faire exploser la mémoire ou le temps de calcul.

Une fois le nombre premier maximal atteint, le reste est alors écrit sous la forme d'un produit de fractions dont le dénominateur est premier en décomposant ce dénominateur en facteurs premiers.

La forme « Périgourdine » est donc :

$$n_0/1 + n_1/p_1 + n_2/p_2 + ... + n_k/p_k + m_0/1 * m_1/ q_1 * m_2/ q_2 * ... * m_j/p_j$$

Les n et m sont des entiers. Les p et q sont des premiers.

Les p_i sont tels que $p_i < p_j$ si $i < j$.

Les q_i sont tels que $q_i <= q_j$ si $i < j$.

Exemple :

7/4

forme égyptienne : 7/4 = 1/1 + 1/2 + 1/4

forme périgourdine :

7/4 = 1/1 + 1/2 + 1/5 + 1/23 + 1/157 + 1/6569 + 39/1 * 1/2 * 1/2 * 1/5 * 1/23 * 1/157 * 1/6569

ou encore si on amène la limite de recherche des nombres premier en dessous de 6569 :

7/4 = 1/1 + 1/2 + 1/5 + 1/23 + 1/157 + 11/1 * 1/2 * 1/2 * 1/5 * 1/23 * 1/157

À noter que les dénominateurs q de la partie produit sont les mêmes que les dénominateurs p de la partie somme.

Arbre binaire de fractions

On part de deux « méta-racines » 0/1 à gauche et 1/0 à droite. En admettant le 0 au dénominateur pour une fois, ces deux objets sont en dehors de l'arbre, ou du moins au dessus de la racine et ne représentent pas vraiment des rationnels mais des générateurs.

On définit une loi de composition « ° » de deux fractions a/b et c/d donnant a+c/b+d, une sorte d'addition membre à membre en additionnant respectivement les numérateurs et dénominateurs.

La racine est l'unité : 1/1 obtenue par 0/1 ° 1/0.

Chaque fraction dans l'arbre, à partir de la racine aura un fils gauche et un fils droit.

Mais aussi, chaque fraction dans l'arbre, à partir de la racine aura un parent gauche et un parent droit. Récursivement, à part la racine qui a pour parent gauche 0/1 et parent droit 1/0, tout nœud, ou toute fraction possède un parent direct, il est gauche si le nœud en question est son fils droit et droit

si nœud est son fils gauche. L'autre parent sera le premier parent direct du côté opposé rencontré dans l'héritage.

Pour clarifier voici ci-dessous le début de l'arbre (avec les méta-racines).

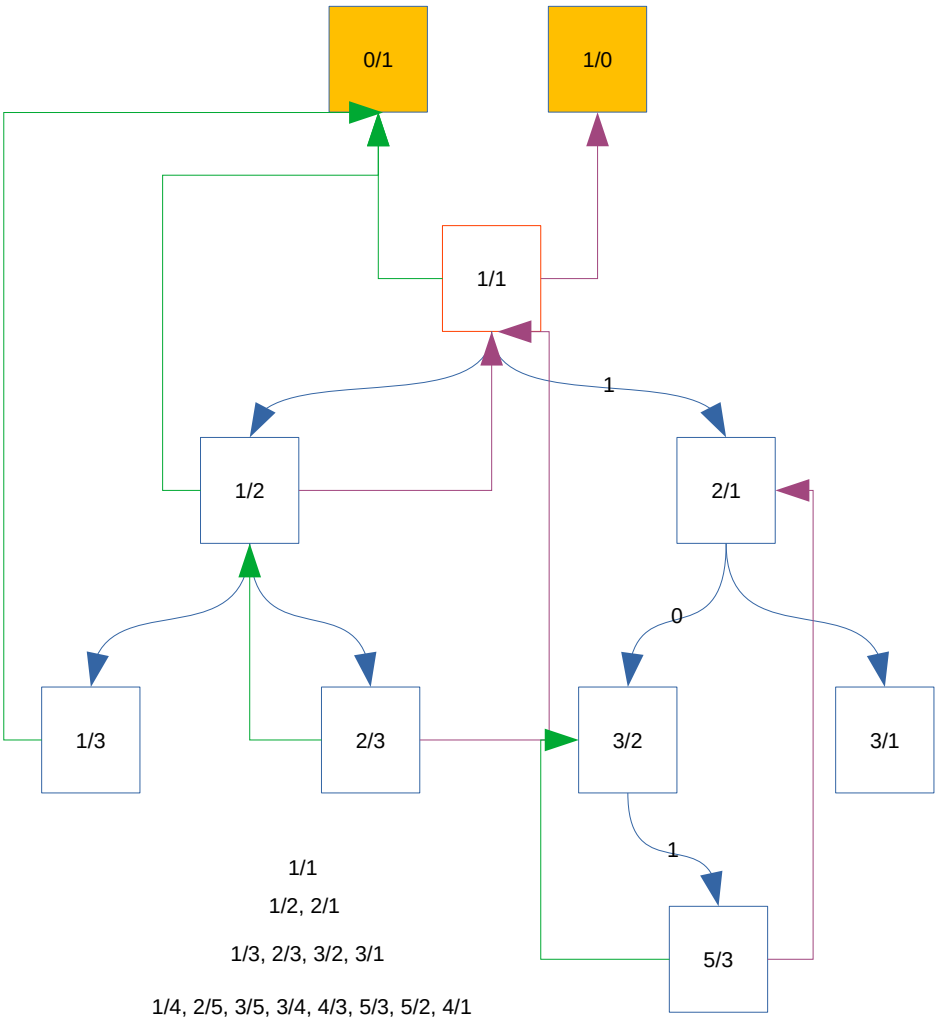

Figure ci-dessus : Avec un fond orange, les « méta-racines » 0/1 et 1/0. Avec le cadre rouge la racine unité.
Les flèches courbes pointent vers les fils.
Les flèches vertes pointent vers les parents gauches, et les violettes vers les parents droits (elles ne sont pas toutes figurées).
Par exemple, 5/3 a pour parent gauche 3/2 et pour parent droit 2/1 : c'est le premier parent droit que l'on rencontre en remontant. 3/2 a aussi 2/1 pour parent droit, puisqu'il est le fils gauche de 2/1.
On peut vérifier que $3/2 \circ 2/1 = 5/3$.

Cet arbre est connu sous le nom d'arbre de Stern-Brocot (voir https://fr.wikipedia.org/wiki/Arbre_de_Stern-Brocot).

Remarques

- Le bord droit de l'arbre représente les nombres entiers naturels positifs.
- Le bord gauche leurs inverses.

- À un niveau donné, les fractions sont classées de gauche à droite dans l'ordre croissant.
- La loi ° appliquée à tout nœud entre ses parents gauche et droite conduit directement à une fraction sous la forme simplifiée.
- Si le niveau 0 est celui de la racine, le niveau *n* contient *2 ** n* fractions (ce qui est le cas pour tout arbre binaire).
- Un miroir vertical passant par la racine donne une correspondance entre une fraction et son inverse.

On numérote 1 la racine 1/1. On numérote *2i* le fils gauche du nœud *i* et *2i + 1* son fils droit.

Tous les rationnels positifs sont représentés dans l'arbre, et on a ainsi une bijection des entiers vers les rationnels positifs.

Par exemple, 5/3 => 13. Treize en binaire est 1**101**.

Pour arriver à 5/3 on suit le chemin 101 à partir de la racine 1 (voir la figure ci-dessus les 0 et 1 sur les flèches bleues).

Code binaire d'une fraction

Le chemin dans l'arbre des fractions codé avec un 0 pour la gauche et un 1 pour la droite, en initiant à 1 pour la racine donne un nombre écrit en binaire quand on ajoute un chiffre à droite à chaque descente de niveau.
Comme on l'a vu 5/3 => 1**101**
L'inverse de 5/3, 3/5 s'écrira donc 1 suivit de la négation de 101, 010, c'est-à-dire finalement 1010.

Retrouver la fraction à partir du code

Pour analyser un nombre binaire on trouve facilement le bit de poids faible en faisant un 'et' logique avec 1. Or, c'est la dernière étape du chemin que l'on trouve.
Il nous faut donc définir une « inversion » ou un « *reverse* » des bits, conservant le 1 initial et inversant la suite des autre bits.
1abcd...xyz donnant *1zyx...dcba*
Une fois cette inversion faite, on peut descendre dans l'arbre pour recalculer la fraction.
NB : Le 1 initial permet de de pas perdre le nombre de 0 : 100000 inversé par notre méthode donne bien 100000. Alors que par exemple si on renverse simplement l'ordre des bits de 1110, on obtient 0111, le 0 initial est perdu et 111 ne signifie plus la même chose.

Quel intérêt

Quel intérêt de représenter par exemple l'entier '14 millions' par un nombre binaire de 14 millions de chiffres ?

C'est en effet une très mauvaise représentation des entiers, et de même de leurs inverses.

Mais Par exemple, si on calcule $-(1 - 5^{1/2}) / 2$,

en JavaScript :

```
-(1 - 5 ** (1/2)) / 2
```

on obtient : `0.6180339887498949`

en nombre flottant sur 64 bits.

Si maintenant on converti le nombre binaire 0101010101010101010101010101010101010101 en fraction avec la méthode ci-dessus :

on obtient la fraction 63245986/102334155 et un développement décimal

```
0.6180339887498948909091...
```

plus précis.

À condition de savoir que :

en répétant le pattern (01) dans l’écriture binaire on tend vers cet irrationnel.