UNIVERSITE DE ROUEN

Projet de Combinatoire

Master d'informatique

— Combinatoire et Calcul Symbolique —

Génération d'objets combinatoires décrit par une grammaire

Le but de ce projet est de compter et d'engendrer l'ensemble des objets combinatoires décrits par une grammaire. Il est ainsi possible d'engendrer une grande variété d'objets comme des arbres ou des mots.

Le projet sera implanté en `Sage`. On pourra travailler seul ou en binôme. La date de remise sera précisée ultérieurement. On rédigera également un rapport présentant les fonctionnalités. Les algorithmes devront être expliqués. Toutes les fonctions de ce projet devront être commentées.

1 Introduction : quelques exemples

1.1 Les arbres binaires complets

Un **arbre binaire complet** est soit une feuille, soit un noeud sur lequel on a greffé deux arbres binaires complets. Une manière simple et rapide d'implanter la structure de donnée en `Sage` est de définir une variable formelle `Leaf` pour les feuilles et une fonction formelle `Node` d'arité 2 (demande la version 4.3 de sage) :

```
sage: var("Leaf")               # variable formelle
sage: function("Node", nargs=2) # fonction formelle d'arité 2
sage: tr = Node(Leaf, Node(Leaf, Leaf))
```

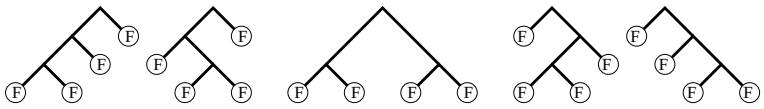
On peut ainsi décrire l'ensemble des arbres binaires par les définitions récursives suivantes :

- l'ensemble `"Trees"` des arbres est la réunion disjointe (que l'on notera en `Sage` par le constructeur `UnionRule`) de deux ensembles : l'ensemble `"Nodes"` des noeuds et l'ensemble `"Leaf"` des feuilles ;
- l'ensemble `"Nodes"` des noeuds est obtenu à partir de l'ensemble des paires d'arbres, c'est-à-dire le produit cartésien (noté par le constructeur `ProdRule`) de l'ensemble des arbres avec lui même ;
- il n'y a qu'un seul arbre possible constitué d'une feuille. C'est le singleton (constructeur `SingletonRule`) `"Leaf"`.

Une telle définition est appelée **grammaire**. On écrira en `Sage` la grammaire des arbres de la manière suivante :

```
treeGram = {"Tree" : UnionRule("Node", "Leaf"),
            "Node" : ProductRule("Tree", "Tree",
                                 lambda (a, b) : Node(a, b)),
            "Leaf" : SingletonRule(Leaf)}
init_grammar(treeGram)
```

Le but de ce projet est d'implanter un algorithme permettant de compter, d'engendrer automatiquement la liste ainsi que de tirer au hasard un des objets décrit par une grammaire de ce type : par exemple il y a 5 arbres binaires complets à quatre feuilles :



Ce que l'on peut obtenir par le programme

```
sage: treeGram['Tree'].count(4)
5
```

La liste des objets décrits par la grammaire peut ensuite être obtenue comme suit :

```
sage: for t in treeGram['Tree'].list(4): print t
Node(Leaf, Node(Leaf, Node(Leaf, Leaf)))
Node(Leaf, Node(Node(Leaf, Leaf), Leaf))
Node(Node(Leaf, Leaf), Node(Leaf, Leaf))
Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)
Node(Node(Node(Leaf, Leaf), Leaf), Leaf)
```

1.2 Les mots de Fibonacci

On appelle mot de Fibonacci tout mot sur l'alphabet `A` et `B` qui ne contient pas deux `B` à la suite. Un tel mot w est décrit par la grammaire suivante :

- soit w est vide ;
- soit w est de la forme `A`u où u est un mot de Fibonacci ;
- soit w est le mot `B` ;
- soit w est de la forme `BA`u où u est un mot de Fibonacci ;

Ceci ce traduit en `Sage` par la grammaire :

```
fiboGram = {"Fib"    : UnionRule("Vide", "Cas1"),
            "Cas1"   : UnionRule("CasAu", "Cas2"),
            "Cas2"   : UnionRule("AtomB", "CasBAu"),
            "Vide"   : EpsilonRule(""),
            "CasAu"  : ProductRule("AtomA", "Fib", "".join),
            "AtomA"  : SingletonRule("A"),
            "AtomB"  : SingletonRule("B"),
            "CasBAu" : ProductRule("AtomB", "CasAu", "".join)}
init_grammar(fiboGram)
```

Note : La commande `"".join` utilise la méthode `join` de la classe `string` qui concatène une liste ou un tuple de chaînes de caractères passé en argument :

```
sage: "".join(["ab","toto"])
'abtoto'
```

Voici la liste des mots de Fibonacci de longueur 3 : `AAA`, `AAB`, `ABA`, `BAA`, `BAB`. Ce qui se calcule en `Sage` par

```
sage: fiboGram['Fib'].count(3)
5
sage: fiboGram['Fib'].list(3)
['AAA', 'AAB', 'ABA', 'BAA', 'BAB']
```

On peut de la même manière obtenir les 21 mots de Fibonacci de longueur 6 :

```
sage: fiboGram['Fib'].list(6)
['AAAAAA', 'AAAAAB', 'AAAABA', 'AAABAA', 'AAABAB', 'AABAAA', 'AABAAB',
 'AABABA', 'ABAAAA', 'ABAAAB', 'ABAABA', 'ABABAA', 'ABABAB', 'BAAAAA',
 'BAAAAB', 'BAAABA', 'BAABAA', 'BAABAB', 'BABAAA', 'BABAAB', 'BABABA']
```

2 Définitions formelles

Une grammaire décrit récursivement un ensemble d'objets. Elle est constituée d'un ensemble de règles ayant chacune un nom (chaîne de caractères). Le nom d'une règle est appelé **symbole non-terminal** ou plus simplement non-terminal de la grammaire.

Une **règle de grammaire** R décrit un ensemble qui est

- soit un singleton dont le seul élément est un objet **atomique** et qui sera de poids 1 (par exemple la feuille d'un arbre).
- soit un ensemble dont le seul élément est un objet **vide** qui sera de poids 0 (par exemple la chaîne vide).
- soit l'**union de deux ensembles** décrit par deux non-terminaux N_1 et N_2 ;
- soit en bijection avec le **produit cartésien de deux ensembles** décrit par deux non-terminaux N_1 et N_2 ; L'ensemble est alors construit à partir des paires d'éléments $(e_1, e_2) \in N_1 \times N_2$. Dans ce cas, il faut de plus donner à `Sage` la bijection qui construit l'objet correspondant à la paire (e_1, e_2) (concaténation pour les chaînes de caractères où constructeur `Node` pour les arbres).

La **taille** ou **poids** d'un objet est le nombre d'atomes qu'il contient. Le poids d'un élément correspondant à une paire (e_1, e_2) est donc la somme des poids de e_1 et de e_2.

À chaque non-terminal on associe la taille du plus petit objet qui en dérive. Cette taille est appelé **valuation** du non-terminal.

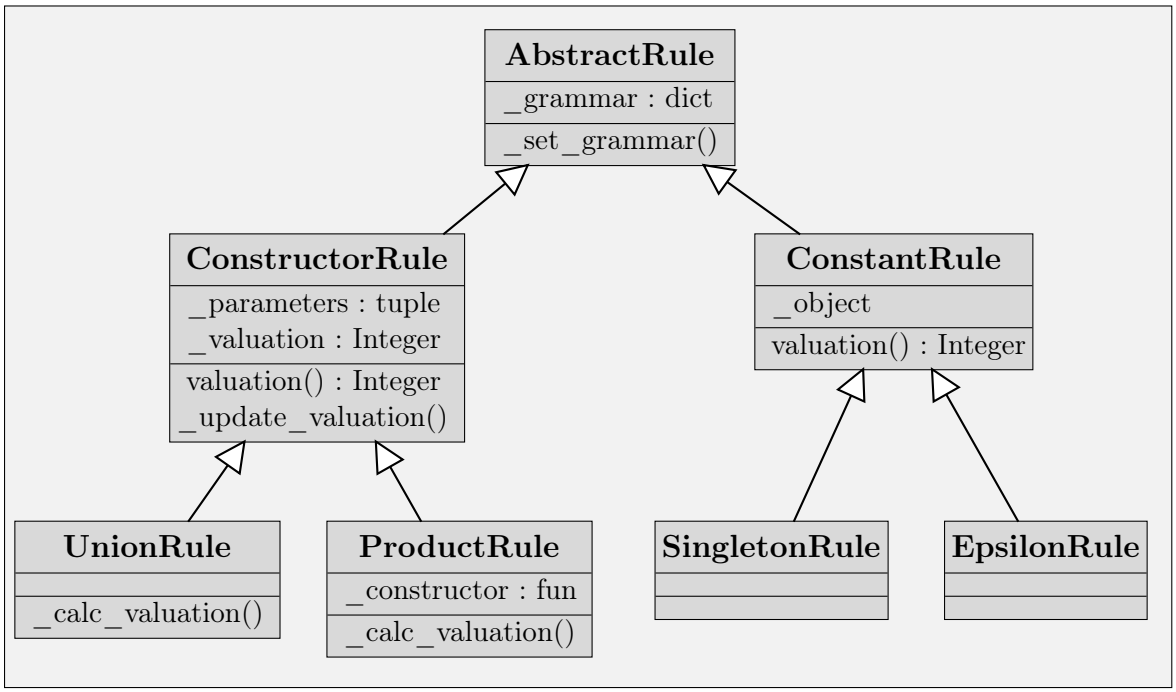
2.1 Structures de données

On modélise chacun de ces ensembles décrits récursivement par un objet (au sens de la programmation orientées objets) `Sage`. Dans l'exemple des arbres binaires, l'objet `treeGram["Tree"]` modélise l'ensemble de tous les arbres binaires. Cet ensemble est construit comme l'union de deux sous ensembles modélisés par les objets `treeGram["Node"]` et `treeGram["Leaf"]`. La classe de l'objet `treeGram["Tree"]` est ainsi `UnionRule`, il est construit grâce à un appel au constructeur par `UnionRule("Node", "Leaf")`.

Une grammaire sera stockée sous la forme d'un dictionnaire qui associe un objet à une chaîne de caractère. Dans le but de ne pas recopier plusieurs fois du code, on utilisera avantageusement la programmation objet et l'héritage. Ainsi chaque règle de grammaire sera un objet de la classe abstraite `AbstractRule`. On distingue deux types de règles de grammaires :

- les règles de constructions qui sont les objets des classes `UnionRule` et `ProductRule` qui dérivent de la classe abstraite `ConstructorRule` ;
- les règles de constantes qui sont les objets des classes `SingletonRule` et `EpsilonRule` qui dérivent de la classe abstraite `ConstantRule` ;

Voici un schéma de la hiérarchie de classe :



Voici la liste des constructeurs (méthodes `__init__`) des différentes classes avec les paramètres et leurs types :

- `SingletonRule.__init__(self, obj)` où `obj` est un `object` python quelconque ;
- `EpsilonRule.__init__(self, obj)` où `obj` est un `object` python quelconque.
- `UnionRule.__init__(self, fst, snd)` où `fst` et `snd` sont deux non terminaux (de type `Sage string`) ;
- `ProductRule.__init__(self, fst, snd, cons)` où `fst` et `snd` sont deux non terminaux et `cons` est une fonction qui prend un couple d'`object` et qui retourne un `object`.

En plus des méthodes listées dans ce diagramme, chaque classe devra implanter (ou bien hériter) les méthodes d'énumération suivantes : `count, count_naive, list, unrank, random`,...

Le principal problème d'implantation provient du fait qu'une grammaire est un **ensemble de définitions mutuellement récursives**. Il y a donc un travail à faire pour « casser des boucles infinies » et s'assurer que la récursion est bien fondée. Voici quelques éléments permettant de résoudre ce problème :

- Pour les règles constructeur par exemple `UnionRule("Node", "Leaf")` les sous règles (ici : `"Node"` et `"Leaf"`) ne sont connues que par les chaînes de caractères qui représentent les symboles non terminaux. Pour pouvoir associer ces chaînes aux objets associés, il faut que l'objet `UnionRule("Node", "Leaf")` ait accès à la grammaire.
- Au moment de la construction d'une règle, la grammaire qui va contenir la règle n'existe pas encore ; il faut donc attendre que la grammaire soit complètement créée pour appeler la méthode `_set_grammar` sur chaque règle. C'est le rôle de la fonction `init_grammar`.
- La fonction `init_grammar` se charge également de calculer les valuations selon l'algorithme décrit plus bas.

2.2 Pour s'entraîner

1. Donner la grammaire de tous les mots sur l'alphabet `A`, `B`.
2. Donner la grammaire des mots de Dyck, c'est-à-dire les mots sur l'alphabet `{(,)}` et qui sont correctement parenthésés.
3. Donner la grammaire des palindromes sur l'alphabet `A`, `B`.
4. Donner la grammaire des mots sur l'alphabet `A`, `B` qui contiennent autant de lettres `A` que de lettres `B`.
5. Écrire une fonction qui vérifie qu'une grammaire est correcte, c'est-à-dire que chaque non-terminal apparaissant dans une règle est bien défini par la grammaire.

3 Algorithmes

On demande d'implanter les algorithmes de calculs suivants (dans ce qui suit `rule` est une règle quelconque) :

- Calcul de la valuation d'une grammaire (ce qui permet de vérifier la grammaire) ;
- méthode `rule.count_naive(self, n)` qui calcule le nombre d'objets d'un poids donné n par la méthode naïve ;
- méthode `rule.count(self, n)` qui calcule le nombre d'objets de poids n par la méthode des séries génératrices ;
- méthode `rule.list(self, n)` qui calcule la liste des objets de poids n ;
- méthode `rule.unrank(self, n)` qui calcule le i-ème élément de la liste des objets de poids n, sans calculer la liste ; Note importante : en `Sage` la position dans une liste commence à 0 ;
- méthode `rule.random(self, n)` qui choisit équitablement au hasard un objet de poids n (on utilisera `rule.unrank(self, n, i)` où i sera choisi aléatoirement).

Le calcul de la valuation est nécessaire aux étapes suivantes qui sont indépendantes. Je les ai néanmoins classé par ordre croissant de difficulté.

3.1 Calcul de la valuation

La **valuation** du non-terminal `nt` est la taille du plus petit objet qui en dérive. La valuation d'une grammaire est l'ensemble des valuations des terminaux. Elle vérifie les quatres règles suivantes :

- la valuation d'un `Singleton` est 1 ;
- la valuation d'un `Epsilon` est 0 ;
- la valuation de l'union `Union` des non-terminaux N_1 et N_2 est le minimum des valuations de N_1 et de N_2 ;
- la valuation du produit `Prod` des non-terminaux N_1 et N_2 est la somme des valuations de N_1 et de N_2 ;

Pour la calculer, on utilise l'algorithme de point fixe suivant. On part de la valuation V_0 (incorrecte) qui associe à chaque non-terminal la valeur ∞. En appliquant une fois non récursivement (pour éviter les boucles infinies) les règles précédentes à partir de V_0, on calcule une nouvelle valuation V_1. On calcule ensuite de même une valuation V_2 a partir de V_1. On recommence tant que la valuation V_n est différente de V_{n-1}. La valuation cherchée $V := V_n$ est obtenue quand $V_n = V_{n-1}$.

Note : Si $V(N) := V_n(N) = \infty$ pour un certain non terminal N, alors aucun objet ne dérive de ce non-terminal. On considère alors que la grammaire est incorrecte.

Par exemple, sur les arbres, le calcul se fait en 4 étapes :

n	`Tree`	`Leaf`	`Node`
règle :	$\min(V_{n-1}(\texttt{Leaf}), V_{n-1}(\texttt{Node}))$	1	$V_{n-1}(\texttt{Tree}) + V_{n-1}(\texttt{Tree})$
0	∞	∞	∞
1	∞	1	∞
2	1	1	∞
3	1	1	2
4	1	1	2
Final	1	1	2

3.1.1 À faire :

6. Écrire un fonction `init_grammar` qui prend en paramètre une grammaire, qui appelle sur chaque règle de la grammaire la méthode `set_grammar` et qui implante l'algorithme de calcul de la valuation.

3.2 Comptage naïf du nombre d'objets

Le comptage du nombre d'objets de poids i se fait en appliquant récursivement les règles suivantes : Soit N un non-terminal. On note $C_N(i)$ le nombre d'objet de poids i.

- si N est un `Singleton` alors $C_N(1) = 1$ et $C_N(i) = 0$ si i est différent de 1 ;
- si N est un `Epsilon` alors alors $C_N(0) = 1$ et $C_N(i) = 0$ si i est différent de 0 ;

- si N est l'union `Union` des non-terminaux N_1 et N_2 alors

$$C_N(i) = C_{N_1}(i) + C_{N_2}(i)\,;$$

- si N est le produit `Prod` des non-terminaux N_1 et N_2

$$C_N(i) = \sum_{k+l=i} C_{N_1}(k) + C_{N_2}(l)\,;$$

Pour aller plus vite et éviter des boucles infinies, on ne considérera dans la somme précédente que les cas où $k \geq V(N_1)$ et $l \geq V(N_2)$, où $V(N)$ désigne la valuation du non-terminal N. En effet, par définition $C_N(i) = 0$ si $V(N) > i$.

3.2.1 À faire :

7. Implanter pour chaque règle de grammaire une méthode `count_naive` qui compte le nombre d'objets d'une grammaire dérivant d'un non-terminal et d'un poids donné.

3.3 Calcul de la liste des objets

On applique récursivement les définitions des constructeurs `Singleton`, `Epsilon`, `Union` et `Prod` pour construire la liste des objets de taille i. En particulier, si N est le produit `Prod` des non-terminaux N_1 et N_2, la liste des objets dérivant de N et de poids i est la concaténation des listes de tous les produits cartésiens d'éléments dérivant de N_1 et de taille k et d'éléments dérivant de N_2 et de taille l, pour tous les couples k, l tels que $k + l = i$, $k \geq V(N_1)$ et $l \geq V(N_2)$ (comme précédemment $V(M)$ désigne la valuation du non-terminal M).

Par exemple, pour obtenir les arbres de taille 3, on procède de la manière suivante

Calcul de `Tree = Union (Leaf, Node)` avec $i = 3$.

- Application de `Leaf = Singleton Leaf` avec $i = 3$, on retourne la liste vide `[]` ;
- Application de `Node = Prod(Tree, Tree)` avec $i = 3$. La valuation de `Tree` est 1. Il y a donc deux possibilités $3 = 1 + 2$ ou $3 = 2 + 1$.
  1. – Application de `Tree = Union (Leaf, Node)` avec $i = 2$.
     - `Leaf` est vide avec $i = 2$ on retourne la liste vide.
     - Application de `Node = Prod(Tree, Tree)` avec $i = 2$. La valuation de `Tree` est 1. Une seule décomposition est possible $1 + 1 \ldots$ On appelle donc deux fois `Tree` avec $i = 1 \ldots$ (Je n'écrit pas les appels récursifs)$\ldots$ qui retourne la liste `[Leaf]`. On retourne donc la liste

     `[Node(Leaf, Leaf)]`

  – Application de `Tree = Union (Leaf, Node)` avec $i = 1$. On retourne la liste

  `[Leaf]`

  Le produit cartésien des deux listes précédentes est la liste formée du seul élément

  `[Node(Node(Leaf, Leaf), Leaf)]`

2. – Application de `Tree = Union (Leaf, Node)` avec $i = 1$. On retourne donc la liste

```
[Leaf]
```

– Application de `Tree = Union (Leaf, Node)` avec $i = 2$. On retourne donc la liste

```
[Node(Leaf, Leaf)]
```

Le produit cartésien des deux listes précédentes est la liste formée du seul élément

```
[Node(Leaf, Node(Leaf, Leaf))]
```

On retourne donc la liste des deux arbres :

```
[Node(Leaf, Node(Leaf, Leaf)), Node(Node(Leaf, Leaf), Leaf)]
```

Pour $i = 6$, il faut essayer les décompositions $6 = 5 + 1$, $6 = 4 + 2$, $6 = 3 + 3$, $6 = 2 + 4$ et $6 = 1 + 5$. Étudions le cas $3 + 3$. Par appel récursif on trouve deux arbres de poids 3 :

```
[Node(Node(Leaf, Leaf), Leaf); Node(Leaf, Node(Leaf, Leaf))]
```

Le produit cartésien est donc formé de 4 éléments qui correspondent aux 4 arbres suivants :

```
Node(Node(Leaf, Node(Leaf, Leaf)), Node(Node(Leaf, Leaf), Leaf));
Node(Node(Node(Leaf, Leaf), Leaf), Node(Node(Leaf, Leaf), Leaf));
Node(Node(Leaf, Node(Leaf, Leaf)), Node(Leaf, Node(Leaf, Leaf)));
Node(Node(Node(Leaf, Leaf), Leaf), Node(Leaf, Node(Leaf, Leaf)));
```

3.4 Calcul à l'aide des séries génératrices

En sage, chaque variable formelle doit être déclarée à l'aide de la commande `var`. On peut ensuite manipuler des expressions où la variable apparaît. Voici par exemple la résolution à la main du cas des arbres binaires :

```
sage: var("tr")
sage: sys = [tr == x + tr*tr]
sage: sol = solve(sys, tr, solution_dict=True)
sage: sol
[{tr: -1/2*sqrt(-4*x + 1) + 1/2}, {tr: 1/2*sqrt(-4*x + 1) + 1/2}]
```

On a alors deux solutions :

```
sage: s0 = sol[0][tr]
sage: s1 = sol[1][tr]
sage: taylor(s0, x, 0, 5)
14*x^5 + 5*x^4 + 2*x^3 + x^2 + x
sage: taylor(s1, x, 0, 5)
-14*x^5 - 5*x^4 - 2*x^3 - x^2 - x + 1
```

Pour trouver quelle est la bonne on peut remplacer x par zéro

```
sage: s0.subs(x=0), s1.subs(x=0)
(0, 1)
```

Dans le cas où la série est une fraction rationnelle, on sait que le dénominateur encode la récurrence. Ceci permet un calcul beaucoup plus rapide. Pour savoir si une expression est une fraction rationnelle, il faut la factoriser, extraire le numérateur et le dénominateur et vérifier que ce sont bien des polynômes. Dans le cas des arbres binaires, cette méthode ne marche pas :

```
sage: s0
-1/2*sqrt(-4*x + 1) + 1/2
sage: s0.factor().numerator().is_polynomial(x)
False
```

Voici un exemple d'une fraction rationnelle :

```
sage: ex = x*2/(2*x^2+x+1) + 2*x+1
sage: ex = ex.factor(); ex
(4*x^3 + 4*x^2 + 5*x + 1)/(2*x^2 + x + 1)
sage: ex.numerator().is_polynomial(x)
True
sage: ex.denominator().is_polynomial(x)
True
```

On peut alors extraire les coefficients :

```
sage: ex.denominator().coefficients(x)
[[1, 0], [1, 1], [2, 2]]
```

3.4.1 À faire :

8. Écrire une fonction qui transforme une grammaire en un système d'équations sur des séries génératrices.

Puisque la grammaire est supposée rationnelle, le système ainsi obtenu est linéaire. La fonction `solve` permet de résoudre un tel système, elle fonctionne essentiellement par pivot de Gauss. La solution est alors une fraction rationnelle.

9. À l'aide de la fonction précédente, ajouter une méthode `generating_series` aux différentes classes représentant les règles de grammaire.

La fonction `taylor` permet alors de calculer les premiers coefficients. Pour calculer les suivants, on utilise le fait que le dénominateur encode une récurrence vérifiée par les coefficients de la série. Ainsi, si

$$f(X) = \sum_{i \geq 0} f_i X^i = \frac{N(X)}{D(X)} = \frac{N(X)}{c_0 + c_1 X + c_2 X^2 + \cdots + C_d X^d} \tag{1}$$

et si n est supérieur au degré du numérateur, alors le coefficient de X^n de $f(X)D(X)$ est nul. On peut sans perte de généralité supposer $c_0 = 1$. On obtient donc les équations pour tout $n > \deg(N(X))$,

$$\sum_{i=0}^{d} c_i f_{n-i} = 0 \qquad \text{soit} \qquad f_n = -\sum_{i=1}^{d} c_i f_{n-i} \tag{2}$$

10. Écrire une méthode qui calcule les coefficients de la série en utilisant la récurrence.

Il est possible d'abaisser la complexité du calcul de f_n utilisant une matrice. En effet, la récurrence s'écrit

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ \vdots \\ f_{n-d+1} \end{pmatrix} = \begin{pmatrix} -c_1 & -c_2 & \dots & -c_{d-1} & -c_d \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \\ \vdots \\ f_{n-d} \end{pmatrix} \tag{3}$$

Pour calculer coefficient f_n il suffit donc d'élever la matrice à la puissance n par exponentiation binaire.

11. Écrire une fonction qui calcule le coefficient de X^n d'une fraction rationnelle par la méthode matricielle.

3.5 Calcul du i-ème élément : la méthode `unrank`

Pour calculer l'élément de taille n et de rang i, on fait appel à la méthode `unrank`. Voici par exemple l'arbre de taille 6 et de rang 12.

```
sage: treeGram['Tree'].unrank(6, 12)
Node(Leaf, Node(Node(Node(Leaf, Node(Leaf, Leaf)), Leaf), Leaf))
```

Attention ! La numérotation commence à zéro.

On procédera récursivement comme suit :

- si l'on demande un objet dont le rang est supérieur au nombre d'objet on lève une exception `ValueError`.
- dans le cas `EpsilonRule` ou `SingletonRule`, on retourne l'objet.
- dans le cas d'une union : `"U" : UnionRule("A", "B")`, on suppose connu les nombres d'objets : $C_U(n) = C_A(n) + C_B(n)$. Alors l'objet de U de rang i est l'objet de A de rang i si $i < C_A(n)$ et l'objet de B de rang $i - C_A(n)$ sinon.
- dans le cas d'un produit : `"U" : ProductRule("A", "B")`, on suppose connu les nombres d'objets :

$$C_U(n) = \sum_{i=0}^{n} C_A(i)\, C_B(n-i)\,.$$

En s'inspirant de l'union, on calcule la valeur de i :

$$U(n) = \bigsqcup_{i=0}^{n} A(i) \times B(n-i)\,.$$

Il reste finalement à trouver l'élément de rang j d'un produit cartésien d'ensemble $A \times B$ où $A = \{a_1, \ldots, a_k\}$ est de cardinalité k et $B = \{b_1, \ldots, b_l\}$ est de cardinalité l. Si l'on choisi comme ordre d'énumération

$$A \times B = \{(a_1, b_1), (a_1, b_2), \ldots, (a_1, b_l), (a_2, b_1), \ldots, (a_2, b_l), \ \ldots \ , (a_k, b_1), \ldots, (a_k, b_l)\}$$

alors l'élément j est (a_r, b_q) où q et r sont respectivement le quotient et le reste de la division euclidienne de j par k que l'on peut calculer en sage par la méthode `quo_rem` des entiers.
Par exemple, pour les arbre de taille 7 :

i	0	1	2	3	4	5	6	7
$C_A(i)\ C_B(n-i)$	0	42	14	10	10	14	42	0

Ainsi, si l'on veut l'arbre de rang 73, comme $73 = 0+42+14+10+7$. On prendra $i = 4$ et l'on retournera l'arbre de rang $j = 7$ du produit cartésien $\texttt{Tree}(4) \times \texttt{Tree}(7-4)$... On a maintenant $j = 7$, $k = 5$, $l = 2$. On retournera donc l'arbre `Node(u,v)` où u est l'arbre de taille 4 et de rang 2 et v l'arbre de taille 3 de rang 1.

4 Pour aller plus loin, on peut...

12. Lors des appels récursifs, on calcule plusieurs fois la même chose. Une amélioration consiste à stocker les résultats des différents appels récursifs dans un tableau pour ne pas refaire le calcul. Pour ceci on peut utiliser les décorateurs `cached_function` et `cached_method` de `Sage`. C'est particulièrement utile pour les méthodes de comptage.
13. Pour avoir un programme plus facile à utiliser, on aimerait bien pouvoir donner des grammaires sous le format

```
{"Tree" :  Union (Singleton Leaf,
                  Prod(NonTerm "Tree", NonTerm "Tree", "".join)}
```

    Une telle grammaire est dite condensée. Une amélioration consiste à écrire une fonction qui développe automatiquement en grammaire simple une grammaire condensée.
14. Ajouter le constructeur `Sequence("NonTerm", casvide, cons)` aux constructeurs autorisés. Ceci peut se faire soit dans les grammaires simples, soit dans les grammaires condensées. En effet, le constructeur `Sequence` peut s'écrire avec `Epsilon` et `Prod` :

```
"Sequence" = Union(Epsilon casvide, Prod("Sequence", "NonTerm", cons))
```

15. Écrire des grammaires pour engendrer des documents `XML` ou `HTML` complexe.

Bon Travail !