Arbres binaires aléatoires

Marc Lorenzi - 24 avril 2018

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import random
from math import sqrt, pi
```

Il existe un certain nombre de méthodes pour créer des arbres binaires aléatoires avec une probabilité uniforme. Un certain nombre d'entre-elles reposent sur une numérotation des arbres binaires à n noeuds et ont pour inconvénient de devoir manipuler des entiers exponentiels en n. En effet, le nombre d'arbres binaires à n noeuds est le $n$ième nombre de Catalan $C_n = \frac{1}{n+1}\binom{2n}{n}$. La méthode présentée ici est au contraire linéaire en n.

1. Parenthésages bien formés

Cette présentation s'inspire de l'article de Atkinson et Sak, "Generating binary trees at random" (Information Processing Letters 41, North-Holland 1992). Les résultats donnés ici sont démontrés dans l'article en question.

On pose $\mathcal{A} = \{+, -\}$. On note $\mathcal{L} = \mathcal{A}^*$ le langage des mots sur l'alphabet $\mathcal{A}$. Les symboles $+$ et $-$ représentent respectivement la parenthèse ouvrante et la parenthèse fermante, et un mot de $\mathcal{L}$ est donc une suite de parenthèses. Le choix de $+$ et de $-$ m'a semblé un bon choix concernant la lisibilité.

Définition : Le **poids** de la lettre $+$ vaut 1, celui de la lettre $-$ vaut -1. Le poids $\mu(w)$ d'un mot w est la somme des poids de ses lettres.

La fonction poids $\mu : \mathcal{L} \to \mathbb{Z}$ est un morphisme de monoïdes de $\mathcal{L}$ muni du produit des mots vers $\mathbb{Z}$ muni de l'addition.

Notation : On note $|w|$ la longueur du mot w. Pour toute lettre $a \in \mathcal{A}$, on note $|w|_a$ le nombre d'occurences de la lettre a dans le mot w.

Définition : Un mot w est **équilibré** lorsque $|w|_+ = |w|_-$. Bien entendu un mot est équilibré ssi son poids est nul. Et tout aussi bien entendu, la longueur d'un mot équilibré est paire.

Les fonctions $w \mapsto |w|$ et $w \mapsto |w|_a$ sont des morphismes de monoïdes de $\mathcal{L}$ muni du produit des mots vers $\mathbb{N}$ muni de l'addition.

Notation : $\mathcal{E} \subset \mathcal{L}$ est l'ensemble des mots équilibrés. On note également, pour tout entier n, $\mathcal{E}_n = \{w \in \mathcal{E}, |w| = 2n\}$.

1.1 Parenthésages aléatoires

La fonction ci-dessous choisit k objets dans une liste s ($k \leq |s|$). Pour cela, elle effectue une permutation aléatoire de la liste et renvoie les k premiers éléments. L'algorithme utilisé est une version modifiée de l'algorithme de Fisher-Yates effectuant le mélange d'une liste.

In [2]:
```
def choose(k, s):
    n = len(s)
    for i in range(k):
        j = random.randint(i, n - 1)
        s[j], s[i] = s[i], s[j]
    return s[:k]
```

In [3]:
```
choose(5, list(range(100)))
```

Out[3]:
```
[89, 90, 97, 45, 37]
```

La fonction `random_balanced_word` renvoie un mot aléatoire équilibré de $\mathcal{E}_n$ avec une probabilité uniforme.

In [4]:
```
def random_balanced_word(n):
    s = choose(n, list(range(2 * n)))
    p = (2 * n) * ['+']
    for k in range(n):
        p[s[k]] = '-'
    w = ''
    for k in range(2 * n):
        w = w + p[k]
    return w
```

In [5]:
```
random_balanced_word(20)
```

Out[5]:
```
'+---+++--++--+--+-++++++-+----+---+--+++'
```

Proposition : la fonction `random_balanced_word` renvoie un mot équilibré de longueur $2n$ avec une probabilité uniforme.

1.2 Défaut d'un mot équilibré

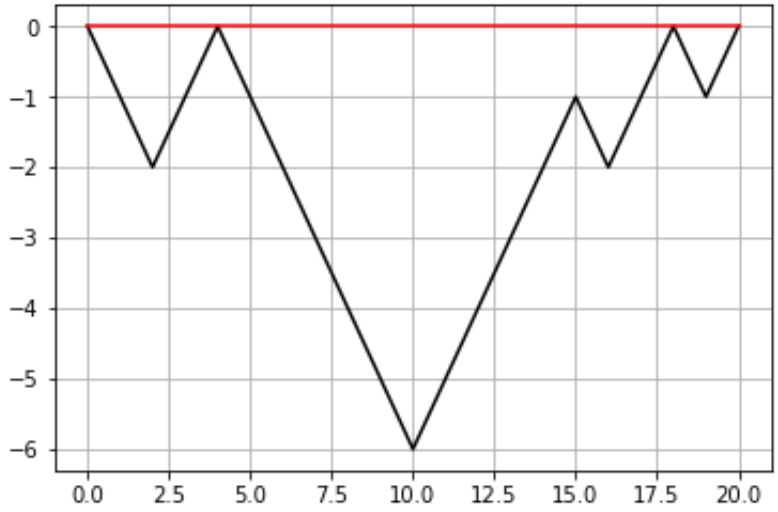
Le poids d'un mot w est la somme $\mu(w)$ des poids de ses lettres, où l'on a posé $\mu(+) = 1$ et $\mu(-) = -1$. La fonction `poids_partiels` prend un mot en paramètre et renvoie la liste des poids de ses préfixes.

In [6]:
```
def poids_partiels(w):
    s = [0]
    mu = 0
    for c in w:
        if c == '+': mu += 1
        else: mu -= 1
        s.append(mu)
    return s
```

La fonction `plot_paren` affiche la représentation graphique de la liste renvoyée par `poids_partiels` .

In [7]:
```
def plot_paren(s):
    n = len(s)
    ws = poids_partiels(s)
    plt.plot(ws, 'k')
    plt.plot((n + 1) * [0], 'r')
    plt.grid()
    plt.show()
```

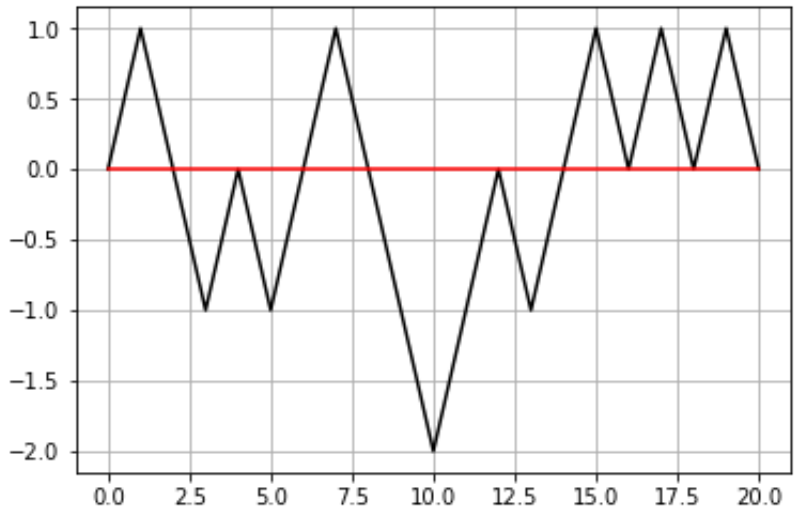
In [8]:
```
plot_paren(random_balanced_word(10))
```



Pour un mot équilibré w, le graphique obtenu est une courbe en zigzag commençant et finissant sur l'axe des abscisses. Le nombre de segments au-dessous de l'axe des x est un nombre paire, $2i$. L'entier i est appelé le **défaut** de w.

In [9]:
```
def defaut(w):
    mu = poids_partiels(w)
    c = 0
    for i in range(len(mu)  - 1):
        if mu[i] <= 0 and mu[i + 1] <= 0: c += 1
    return c // 2
```

In [10]:
```
s = random_balanced_word(10)
plot_paren(s)
print(defaut(s))
```



5

Pour $i = 0, \dots, n$ notons $\mathcal{E}_{ni}$ l'ensemble des mots équilibrés de défaut i. Les $\mathcal{E}_{ni}$ forment une partition de E_n. Il s'avèrent que ces ensembles sont tous de même cardinal.

1.4 Mots bien formés

Définition : Un mot w est dit **bien formé** lorsqu'il est équilibré et de défaut nul, c'est à dire lorsqu'il appartient à $\mathcal{E}_{n0}$.

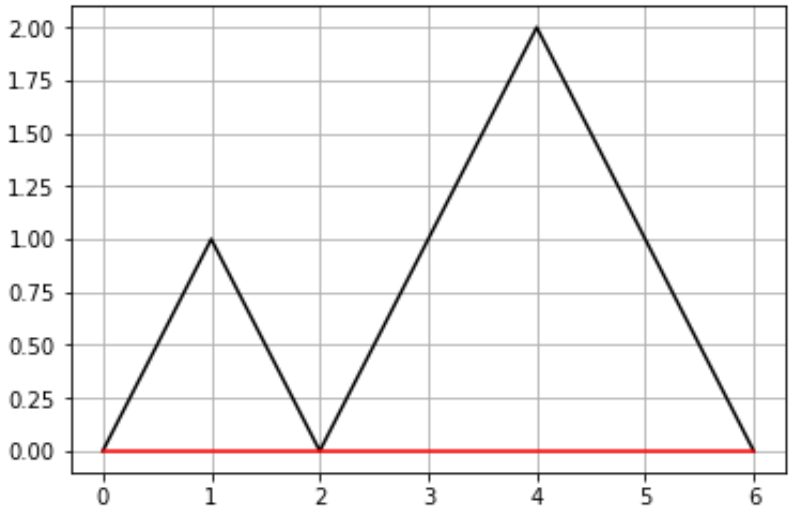
Un mot u est bien formé lorsqu'il correspond à un parenthésage licite.

Notation : On note $\mathcal{F} \subset \mathcal{E}$ l'ensemble des mots bien formés et $\mathcal{F}_n = \mathcal{E}_{n0}$ l'ensemble des mots bien formés de longueur $2n$.

In [11]:
```
weight = {'+': 1, '-': -1}
```

In [12]:
```
def well_formed(w):
    c = 0
    for k in range(len(w)):
        c = c + weight[w[k]]
        if c < 0: return False
    return c == 0
```

In [13]:
```
w = '+-++--'
plot_paren(w)
well_formed(w)
```



Out[13]: True

In [14]:
```
w = '++---+'
plot_paren(w)
well_formed(w)
```

Out[14]: False

1.5 Découpage d'un mot

Notation : pour tout mot w, w^* désigne le mot w dans lequel on a échangé les $+$ et les $-$.

In [15]:
```
oppose = {'+':'-', '-':'+'}
```

In [16]:
```
def star(w):
    w1 = ''
    for c in w:
        w1 = w1 + oppose[c]
    return w1
```

In [17]:
```
star('+--+--')
```

Out[17]: '-++-++'

L'application $\mapsto w^*$ est un endomorphisme du monoïde $\mathcal{L}$.

Définition : Soit $w \in \mathcal{E}$ un mot équilibré. On dit que w est **réductible** lorsque $w = uv$ pù u et v sont équilibrés et non vides. Sinon on dit que w est **irréductible**.

Clairement, un mot équilibré et non vide est irréductible si et seulement si aucun des ses préfixes propres n'est de poids nul. Ce qui équivaut encore à dire que tous ses préfixes propres sont de poids strictement positif ou bien tous ses préfixes propres sont de poids strictement négatif.

Proposition : Si un mot équilibré w est irréductible, alors l'un des deux mots w et w^* est bien formé. Plus précisément, $w = +u-$ où u est bien formé, ou bien $w = -u+$ où u^* est bien formé.

Proposition : Tout mot w équilibré s'écrit de façon unique comme un produit $w = w_1 w_2 \dots w_n$ de mots irréductibles. Le mot w est bien formé si et seulement si les w_i le sont aussi.

La fonction `cut` prend en paramètre un mot w supposé équilibré. Elle renvoie un couple (u, v) tel que $w = uv$ et u est irréductible.

Des appels successifs à `cut` permettent de décomposer w en produit de mots irréductibles.

In [18]:
```
def cut(w):
    k = 1
    p = weight[w[0]]
    while k < len(w) and p != 0:
        p = p + weight[w[k]]
        k = k + 1
    return (w[:k], w[k:])
```

In [19]:
```
w = random_balanced_word(10)
print(w)
print(cut(w))
```

```
--++-+++-+--+---+-++
('--++', '-+++-+--+---+-++')
```

La foncction `decomposer` prend en paramètre un mot `w` supposé équilibré. Elle renvoie la décomposition du mot w en produit de mots irréductibles.

In [20]:
```
def decomposer(w):
    if len(w) == 0: return []
    else:
        u, v = cut(w)
        us = decomposer(v)
        return [u] + us
```

In [21]:
```
w = random_balanced_word(30)
print(w)
print(decomposer(w))
```

```
+++-+-+-++---+-+--+--++---+-+++-++--+++-+-+---+---++-++-++--
['+++-+-+-++---+-+--', '+-', '-+', '+-', '--+-++', '+-', '++--', '++
+-+-+---', '+-', '--++', '-+', '+-', '++--']
```

1.6 Création de mots bien formés

La fonction $\varphi : \mathcal{E} \to \mathcal{F}$ est définie par induction structurelle sur les mots.

- $\varphi(\varepsilon) = \varepsilon$, le mot vide.
- Pour tout $w \in \mathcal{E} \setminus \{\varepsilon\}$, on écrit $w = uv$ où u est irréductible.
  - Si $u \in \mathcal{F}$, $\varphi(w) = u\varphi(v)$.
  - Sinon, $u = -t+$ où $t^* \in \mathcal{F}$, et $\varphi(w) = +\varphi(v) - t^*$.

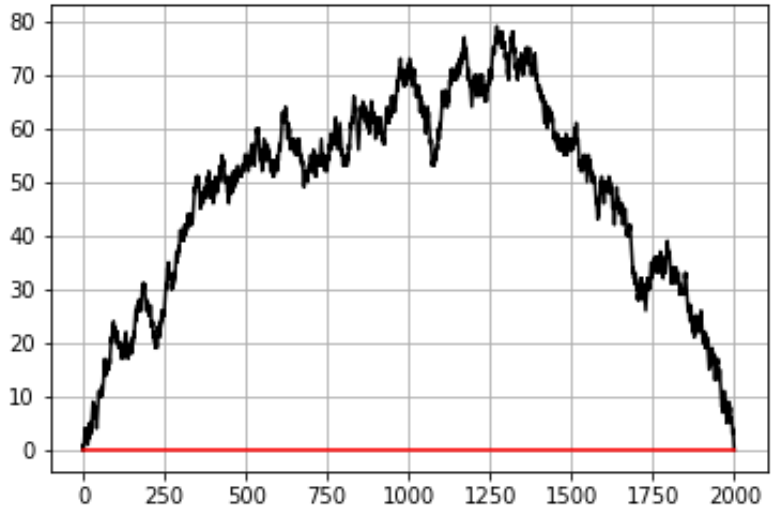
In [22]:
```
def phi(w):
    if len(w) == 0: return ''
    else:
        u, v = cut(w)
        v1 = phi(v)
        if well_formed(u): return u + v1
        else:
            return '+' + v1 + '-' + star(u[1:-1])
```

In [23]:
```
def random_well_formed_word(n):
    return phi(random_balanced_word(n))
```

In [24]:
```
w = random_well_formed_word(500)
print(w)
print(well_formed(w))
```

```
+++-++---+++++-++--+-+--+++---+-++++--+-++++-++++---+--+++-++++----+
-+---+++------+++--+-+++-++---+++--+-+-++--+++-+++----+++++------+++
-++++++-++-+--+-+++-+++--++-+++-++++--+-+++++--++--+-++-+-++--+--+--
+++-+-+++--+-+++--------+-+---+--+++-++-++-++++--++---+++-++-+--+-++
--+-+++-++--+++++-++-+-+-++++----++-+--++---++--+++---+---+--++++---
++---+++++--+--+--+-----+++---++-+--+-+---+-++---++++-+-+-++-++-+-+-
+-+----+------+-+-+----+-+-+--++++----+-+++-+-+---+-+-+---+---++++-
-+----+-+-++++-+-+--++-+-+++-++++---+++-+-+-+-++++---+++--+---+-++--
+-++-+-+-++----------++-+++---+---++-+-+++--+-+-+-++---+--+--++-++--
++++++++-+++--++---+++--+++-+--------+-+++-+-++----++-+-+-+--++----
---++++---+++-+---+---+++---+---++-+-------+-+-+--++---++++-+-++++
+---+++-+-++++-+--++++++--+-++-+-+-----++---+--+++++++++--+---+---+-
+-----+-++---+-++-+--+--++-+-++++--++-+-+-+--+--++---+-+---+-+--++-+
++--+--++-+----+-+--++---+----+++-+----++---++-++-+--+-+++-++-++----
--++--+++-++-++-+++-+---++-+--+-++-+--++--+-----
True
```

In [25]:
```
w = random_well_formed_word(1000)
plot_paren(w)
```



Proposition : $\varphi : \mathcal{E}_{ni} \to \mathcal{F}_n$ est une bijection.

Proposition : $\varphi : \mathcal{E}_n \to \mathcal{F}_n$ est surjective. Mieux, tout mot de $\mathcal{F}_n$ a exactement $n+1$ antécédents par φ, un dans cacun des $\mathcal{E}_{ni}, i = 0, \dots, n$.

Proposition : Soit $X : \Omega \to \mathcal{E}_n$ une variable aléatoire suivant une loi uniforme. Alors $\phi(X) : \Omega \to \mathcal{F}_n$ suit une loi uniforme.

1.7 Bilan

Nous disposons d'un algorithme renvoyant un mot bien formé aléatoire de longueur $2n$ avec une loi uniforme. Nous allons maintenant établir une bijection entre les mots bien formés de $\mathcal{F}_n$ et les arbres binaires à n noeuds internes. La composition de notre algorithme avec la bijection fournira ainsi un algorithme renvoyant un arbre binaire aléatoire à n noeuds avec une loi uniforme.

2. Arbres binaires aléatoires

On s'intéresse uniquement à la forme des arbres binaires, et pas à la valeur de leurs noeuds. Un arbre est soit vide (nous noterons e l'arbre vide), soit un couple (t_1, t_2) où t_1 et t_2 sont eux-mêmes des arbres.

On note $\mathcal{T}$ l'ensemble des arbres binaires et, pour tout entier n, $\mathcal{T}_n$ l'ensemble des arbres binaires à n noeuds (internes).

Nous représenterons l'arbre vide en Python par l'entier 0 (choix arbitraire, encore une fois dicté par des considérations de lisibilité).

In [26]:
```
empty_tree = 0
def node(t1, t2): return (t1, t2)

def left(t): return t[0]
def right(t): return t[1]
```

2.1 Conversions entre mots bien formés et arbres binaires

Soit $\psi : \mathcal{F} \to \mathcal{T}$ la fonction définie comme suit. Soit $w \in \mathcal{F}$ un mot bien formé. Tout d'abord, $\psi(\varepsilon) = e$, l'arbre vide. Pour tout mot $w \in \mathcal{F} \setminus \{\varepsilon\}$, $w = uv$ où $u = +t-$ est irréductible, et t est bien formé. Soient $t_1 = \psi(t)$ et $t_2 = \psi(v)$. On pose alors $\psi(w) = (t_1, t_2)$.

On vérifie aisément que pour tout entier n, $\psi : \mathcal{F}_n \to \mathcal{T}_n$.

In [27]:
```
def well_formed_word_to_binary_tree(w):
    if len(w) == 0: return empty_tree
    else:
        u, v = cut(w)
        t1 = well_formed_word_to_binary_tree(u[1:-1])
        t2 = well_formed_word_to_binary_tree(v)
        return node(t1, t2)
```

In [28]:
```
well_formed_word_to_binary_tree('++--+-+-')
```

Out[28]:
```
((0, 0), (0, (0, 0)))
```

La réciproque $\chi : \mathcal{T} \to \mathcal{F}$ de la fonction ψ est la suivante. $\chi(e) = \varepsilon$. Et pour tout $t \in \mathcal{T}$, $t = (t_1, t_2)$ où t_1 et t_2 sont des arbres binaires. On pose alors $\chi(t) = +\chi(t_1) - \chi(t_2)$.

In [29]:
```
def binary_tree_to_well_formed_word(t):
    if t == empty_tree: return ''
    else:
        w1 = binary_tree_to_well_formed_word(left(t))
        w2 = binary_tree_to_well_formed_word(right(t))
        return '+' + w1 + '-' + w2
```

In [30]:
```
w = random_well_formed_word(40)
w1 = binary_tree_to_well_formed_word(well_formed_word_to_binary_tree(w
print(w)
print(w1)
print(w1 == w)
```

```
++++--+-++-++--+-+--++++-++---+-++--++--+-+++------+-+++-+++-++-+-+-
-----+-+-+--
++++--+-++-++--+-+--++++-++---+-++--++--+-+++------+-+++-+++-++-+-+-
-----+-+-+--
True
```

2.2 Arbre binaire aléatoire

Il est maintenant évident de créer un arbre binaire aléatoire.

In [31]:
```
def random_binary_tree(n):
    w = random_well_formed_word(n)
    return well_formed_word_to_binary_tree(w)
```

In [32]:
```
print(random_binary_tree(100))
```

```
((0, ((0, (((0, ((0, 0), ((0, 0), ((((((0, (((0, (0, (0, 0))), (0, 0
)), 0)), 0), 0), 0), 0), (0, 0))))), (((0, 0), (0, (0, 0))), ((0, ((
0, ((((0, 0), ((((((0, ((0, (0, (0, 0))), (0, (((((0, 0), 0), 0), 0)
, ((0, ((0, 0), (0, ((0, (0, 0)), 0)))), 0)))))), 0), 0), 0), 0), (((
0, 0), 0), 0))), 0), 0)), 0)), 0))), ((0, (0, 0)), ((0, 0), ((0, 0),
0))))), 0)), (((0, (((((((0, 0), ((0, 0), 0)), 0), 0), 0), 0), ((0,
0), 0))), 0), (0, (((0, (0, 0)), (((0, (0, (0, ((0, (0, 0)), 0)))),
(0, 0)), (0, 0))), 0))))
```

2.3 Hauteur moyenne

La fonction ci-dessous renvoie la hauteur de l'arbre t.

In [33]:
```
def hauteur(t):
    if t == empty_tree: return 0
    else:
        h1 = hauteur(left(t))
        h2 = hauteur(right(t))
        return 1 + max([h1, h2])
```

In [34]:
```
hauteur(random_binary_tree(100))
```

Out[34]: 28

La théorie prévoit que la hauteur moyenne d'un arbre binaire à n noeuds est équivalente à $2\sqrt{\pi n}$ lorsque n tend vers l'infini. L'écart-type de cette même hauteur étant équivalent à $2\sqrt{\pi(\frac{\pi}{3}-1)n}$ (**source peu sûre, à vérifier !**), donc du même ordre de grandeur que la moyenne. Les tests numériques risquent donc de renvoyer des résulats pas très convaincants.

In [35]:
```
def average_height(n):
    return 2 * sqrt(pi * n)
```

In [36]:
```
average_height(100)
```

Out[36]:
```
35.44907701811032
```

La fonction `stats_hauteur` prend deux entiers n et m en paramètres. Elle tire au sort m arbres aléatoires à n noeuds et renvoie la moyenne de leurs hauteurs.

In [37]:
```
def stats_hauteur(n, m):
    s = 0
    for k in range(m):
        s = s + hauteur(random_binary_tree(n))
    return s / m
```

In [38]:
```
stats_hauteur(100, 1000)
```

Out[38]:
```
29.949
```

L'ordre de grandeur est bon, sans plus. Il est possible (à vérifier) qu'un petit nombre d'arbres participent de façon signifactive à l'augmentation de l'espérance de la hauteur.

2.4 Nombre moyen de feuilles

Le nombre moyen de feuilles d'un arbre binaire à n noeuds est $\frac{n(n+1)}{2(2n-1)} simfracn4$. Je n'ai pas d'informations sur l'écart-type ...

In [39]:
```
def nombre_moyen_feuilles(n):
    return n * (n + 1) / (2 * (2 * n - 1))
```

In [40]:
```
nombre_moyen_feuilles(100)
```

Out[40]:
```
25.376884422110553
```

In [41]:
```
def nombre_feuilles(t):
    if t == empty_tree: return 0
    else:
        t1, t2 = left(t), right(t)
        if t1 == empty_tree and t2 == empty_tree: return 1
        else:
            f1 = nombre_feuilles(t1)
            f2 = nombre_feuilles(t2)
            return f1 + f2
```

In [42]:
```
nombre_feuilles(random_binary_tree(100))
```

Out[42]:
```
21
```

In [43]:
```
def stats_feuilles(n, m):
    s = 0
    for k in range(m):
        s = s + nombre_feuilles(random_binary_tree(n))
    return s / m
```

In [44]:
```
stats_feuilles(100, 1000)
```

Out[44]:
```
25.371
```

Pas mal.

3. Dessiner un arbre binaire

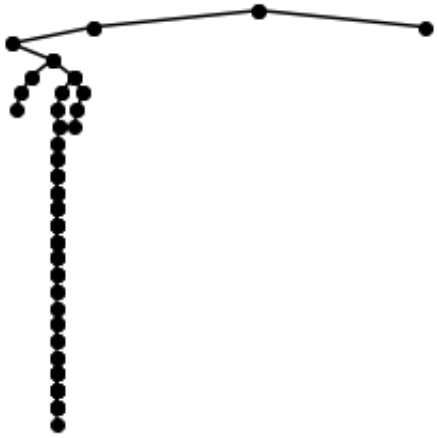
À quoi **ressemble** un arbre binaire aléatoire "moyen" ? La réponse est un peu déroutante ...

La fonction `draw_tree` ci-dessous dessine un arbre binaire. Elle est lente, la raison en est le choix de `matplotlib` pour le tracé. Mieux vaut ne pas tenter de dessiner des arbres ayant plus de quelques centaines de noeuds.

In [45]:
```
def draw_tree_aux(t, rect, dy):
    if t == empty_tree: return
    x1, x2, y1, y2 = rect
    xm = (x1 + x2) // 2
    t1, t2 = left(t), right(t)
    draw_tree_aux(t1, (x1, xm, y1, y2 - dy), dy)
    draw_tree_aux(t2, (xm, x2, y1, y2 - dy), dy)
    if not t1 == empty_tree:
        a, b = ((xm, (x1 + xm) // 2), (y2, y2 - dy))
        plt.plot(a, b, 'k', marker='o')
    if not t2 == empty_tree:
        c, d = ((xm, (x2 + xm) // 2), (y2, y2 - dy))
        plt.plot(c, d, 'k', marker='o')
```

In [46]:
```
def draw_tree(t):
    d = 512
    pad = 20
    dy = (d - 2 * pad) / (hauteur(t))
    draw_tree_aux(t, (pad, d - pad, pad, d - pad), dy)
    plt.axis([0, d, 0, d])
    plt.axis('off')
    plt.show()
```

In [47]:
```
N = 50
t = random_binary_tree(N)
draw_tree(t)
```



Les arbres dessinés apparaissent TRÈS déséquilibrés. Soit c'est normal, soit je dois refaire tous mes calculs, ce qui n'est pas concevable. Tentons juste une petite estimation. Soit t un arbre binaire aléatoire à n noeuds. Quelle est la probabilité que t ait un seul fils ? Notons $\mathcal{T}_{l,n}$ et $\mathcal{T}_{r,n}$ les ensembles formés des arbres ayant respectivement seulement un fils gauche et seulement un fils droit.

Notons $C_n = |\mathcal{T}_n|$ le cardinal de $\mathcal{T}_n$. C_n est le n-ième **nombre de Catalan**, il vaut

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

De là $|\mathcal{T}_{ln}| = C_{n-1}$: un arbre à n noeuds ayant seulement un fils gauche est formé d'une racine et d'un unique fils ayant $n-1$ noeuds. La probabilité qu'un arbre soit dans $|\mathcal{T}_{ln}|$ est donc (probabilité uniforme !) $\frac{C_{n-1}}{C_n} = \frac{n+1}{n}\frac{\binom{2n-2}{n-1}}{\binom{2n}{n}} = \frac{n+1}{2(2n-1)}$.

Il en est évidemment de même pour $\mathcal{T}_{rn}$. Or, pour $n \geq 2$, les ensembles $\mathcal{T}_{ln}$ et $\mathcal{T}_{rn}$ sont disjoints (un arbre ne peut pas avoir ses deux fils vides s'il a au moins deux noeuds). Donc, $P(\mathcal{T}_{rn} \cup \mathcal{T}_{rn}) = P(\mathcal{T}_{ln}) + P(\mathcal{T}_{rn}) = \frac{n+1}{2n-1} \sim \frac{1}{2}$.

La probabilité qu'un arbre ait un seul fils est asymptotiquement $\frac{1}{2}$. Il y a une chance sur deux que l'arbre ait l'air penché :-).

In [ ]: