3
Puissances et polynômes

1. On peut calculer n'importe quelle puissance d'un réel strictement positif à l'aide de l'exponentielle et du logarithme :

$$\forall\, x > 0,\ \forall\, \alpha \in \mathbb{R}, \quad x^\alpha = \exp\big(\alpha \cdot \ell\mathrm{n}\, x\big).$$

En pratique, cela demande de savoir calculer n'importe quelle valeur de exp et de ℓn avec une bonne précision et une certaine efficacité.

2. Pour un exposant entier $\alpha \in \mathbb{N}$, il est alors peut-être plus efficace de revenir à la définition :

$$x^0 = x, \qquad \forall\, n \geqslant 1, \quad x^n = x^{n-1} \cdot x.$$

En utilisant cette relation de récurrence, il faut effectuer n multiplications pour calculer x^n. Comme effectuer *une* multiplication est une opération beaucoup plus simple (et donc beaucoup plus rapide) que de calculer une valeur de exp ou de ℓn, cette méthode présente un réel intérêt... sauf si l'exposant n est vraiment grand.

Q 1. Mises en œuvre de l'algorithme naïf
Proposer une version itérative et une version récursive d'une fonction `puissance(x, n)` *qui retourne* x^n *calculée avec la méthode exposée ci-dessus.*

I
Exponentiation rapide

3. L'algorithme d'exponentiation rapide repose lui aussi sur une relation de récurrence :
– Si l'exposant est pair : $q_k = 2q_{k+1}$, alors

$$x^{q_k} = (x^2)^{q_{k+1}}.$$

– Si l'exposant est impair : $q_k = 2q_{k+1} + 1$, alors

$$x^{q_k} = x \cdot (x^2)^{q_{k+1}}.$$

4. Comme d'habitude, il est facile d'en déduire un algorithme récursif.

```
def puissance2(x, q):
  if (q==0):          # Si q_k est nul
    return 1
  else:
    if (q%2==0):      # Si q_k est pair et non nul
      return puissance2(x*x, q//2)
    else:             # Si q_k est impair
      return x*puissance2(x*x, q//2)
```

Q 2.a *On suppose qu'il existe un entier $p \geqslant 1$ tel que $2^p \leqslant n < 2^{p+1}$. Quel encadrement de q peut-on en déduire ?*

Q 2.b *Démontrer que l'algorithme termine. Exprimer en fonction de p le nombre d'appels récursifs effectués.*

Q 2.c *Estimer le nombre de multiplications effectuées pour calculer x^n. Pour quelles valeurs de n l'algorithme rapide est-il 10 fois plus rapide que l'algorithme naïf ?*

5. Décomposons l'exposant $2^p \leqslant n < 2^{p+1}$ en base 2 :

$$n = \sum_{k=0}^{p} r_k 2^k$$

avec $r_k \in \{0,1\}$ pour tout $0 \leqslant k < p$ et $r_p = 1$. On en déduit que

$$x^n = \prod_{k=0}^{p} (x^{2^k})^{r_k} = \prod_{\substack{0 \leqslant k \leqslant p \\ r_k = 1}} x^{2^k}$$

en remarquant que les différents facteurs peuvent être calculés de proche en proche :

$$\forall\, k \in \mathbb{N}, \quad x^{2^{k+1}} = x^{2 \cdot 2^k} = (x^{2^k})^2.$$

Q 3. *Proposer une version itérative de l'algorithme d'exponentiation rapide.*

II

Applications

II.1 Suites récurrentes linéaires

6. Une suite récurrente linéaire d'ordre $N \geqslant 2$ peut se traduire en une suite géométrique dont la raison est une matrice compagnon $A \in \mathfrak{M}_N(\mathbb{R})$.

6.1 Par exemple, la suite de Fibonacci définie par la donnée de $F_0 = F_1 = 1$ et la relation de récurrence

$$\forall\, n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

peut-elle s'écrire

$$\forall\, n \geqslant 2, \quad \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix}.$$

On peut alors calculer le n-ième terme de cette suite récurrente *sans avoir calculer les termes intermédiaires !*

$$\forall\, n \geqslant 1, \quad F_n = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

6.2 La multiplication matricielle étant une opération sensiblement plus coûteuse que la multiplication des nombres (et d'autant plus coûteuse que la taille N est grande), il est intéressant de calculer la puissance de cette matrice par l'algorithme d'exponentiation rapide.

Q 4.a *Calculer un équivalent de F_n.*

Q 4.b *On peut calculer F_n par un algorithme naïf (qui reproduit la relation de récurrence).*

```
def Fib1(n):
  F = 1, 1
  for k in range(n-1):
    F = F[1], F[0]+F[1]
  return F[1]
```

Prouver cet algorithme et estimer sa complexité.

Q 4.c *On peut aussi calculer F_n avec l'algorithme d'exponentiation rapide.*

```
import numpy as np

def Fib2(n):
  L = np.matrix([[0,1]])
  A = np.matrix([[0,1], [1,1]])
  B = puissance2(A, n-1)
  C = np.matrix([[1],[1]])
  return (L*B*C)[0,0]
```

Estimer la complexité de ce nouvel algorithme.

Q 4.d *Calculer F_{46} avec* `Fib1` *et* `Fib2`*. Commenter.*

II.2 Systèmes différentiels

7. Un système différentiel linéaire à coefficients constants est de la forme

$$\forall\, t \geqslant 0, \quad X_t' = AX_t.$$

7.1 Le schéma d'Euler consiste alors à choisir un pas de temps δt et à transformer le système différentiel précédent en

$$X(t+\delta t) - X(t) \approx \delta t \cdot AX(t)$$

puis en

$$\forall\, k \in \mathbb{N}, \quad X((k+1)\delta t) \approx (I_N + \delta t \cdot A)X(k\delta t).$$

On en déduit une valeur approchée de $X(k\delta t)$ en fonction de la condition initiale $X(0)$:

$$\forall\, k \in \mathbb{N}, \quad X(k\delta t) \approx (I_N + \delta t \cdot A)^k X(0).$$

7.2 Cela suggère que, pour un instant $T > 0$ fixé,

$$X(T) \approx (I_N + \delta t \cdot A)^n X(0)$$

où $T = n\delta t$. Si on choisit le pas de temps δt assez petit pour que le résultat ainsi obtenu soit raisonnablement précis, il faut que l'entier n soit grand : l'algorithme d'exponentiation rapide prend tout son sens.

7.3 Cependant, si on s'intéresse plus à la *trajectoire* $\big(X(t)\big)_{0\leqslant t\leqslant T}$ de la solution qu'à sa valeur finale à l'instant T, il vaut mieux calculer de proche en proche des valeurs approchées de $X(i\delta t)$ pour $0 \leqslant i \leqslant n$ à partir de la relation de récurrence

$$\forall\, 0 \leqslant i < n, \quad X_{i+1} = (I_N + \delta t \cdot A)X_i$$

que de calculer directement chacun des $X(k\delta t)$: on effectuera ainsi beaucoup moins de calculs (puisqu'on se contentera ainsi d'*un seul* produit matriciel par vecteur calculé).

III

Évaluation d'un polynôme

8. Un polynôme étant donné sous forme développée :

$$P = a_0 + a_1 X + \cdots + a_d X^d$$

on cherche à évaluer ce polynôme, c'est-à-dire à substituer diverses valeurs x à l'indéterminée X (qu'il s'agisse de valeurs réelles, complexes ou même matricielles).

8.1 La méthode élémentaire consiste à calculer les puissances de x de proche en proche.

```
def evaluer(P, x):
  val, puiss = P[0], 1.0  # v_0 = a_0 , p_0 = 1
  for a in P[1:]:         # Pour 1 ⩽ k ⩽ d = deg P
    puiss *= x            # p_k = x p_{k-1}
    val += a*puiss        # v_k = v_{k-1} + a_k p_k
  return val              # v_d
```

8.2 Pour vérifier la correction de cet algorithme, on démontre par récurrence que

$$\forall\, 0 \leqslant k \leqslant d, \quad p_k = x^k \quad \text{et} \quad v_k = \sum_{i=0}^{k} a_i x^i.$$

La valeur retournée est alors $v_d = P(x)$.

8.3 Comme on le voit, si P est un polynôme de degré d, cette méthode effectue $2d$ multiplications et d additions.

Plus précisément, si x est une *matrice*, cette méthode effectue d multiplications matricielles (très coûteuses) et d multiplications scalaires (beaucoup moins coûteuses).

Il s'agit donc d'un algorithme linéaire en temps et constant en espace.

8.4 Utiliser l'algorithme d'exponentiation rapide pour évaluer un polynôme n'apporte donc un gain réel que dans une situation très particulière : lorsque le polynôme compte *peu* de termes non nuls et que certains de ces termes sont de degré très élevé, comme par exemple

$$1 - X + X^{1\,024}.$$

Pour un polynôme comme

$$1 + X + X^2 + X^3 + \cdots + X^{999} + X^{1\,000},$$

le gain de temps réalisé en calculant rapidement chaque puissance est largement compensé par le fait de calculer *isolément* toutes ces puissances : évaluer ce polynôme avec l'algorithme d'exponentiation rapide est une perte de temps.

9. Schéma de Horner

Pour évaluer un polynôme, le **schéma de Horner** n'est pas beaucoup plus efficace que la méthode élémentaire présentée ci-dessus [**8.1**]. En revanche, il donne une information supplémentaire : on obtient en même temps le quotient de la division euclidienne de P par x.

9.1 On a intérêt à représenter ici le polynôme P sous la forme

$$a_0X^d + a_1X^{d-1} + \cdots + a_{d-1}X + a_d.$$

L'algorithme consiste à poser $b_0 = a_0$ puis à suivre la relation de récurrence suivante :

$$\forall\ 1 \leqslant k \leqslant d, \qquad b_k = xb_{k-1} + a_k.$$

9.2 On peut démontrer que

$$b_d = P(x)$$

et que le polynôme

$$Q = b_0X^{d-1} + b_1X^{d-2} + \cdots + b_{d-2}X + b_{d-1}$$

est le quotient de la division euclidienne de P par $(X - x)$.

Q 5. *Avec les notations précédentes, un polynôme P est représenté par la liste $(a_0, a_1, \ldots, a_d)$.*

Q 5.a *Comparer la complexité du schéma de Horner à celle de l'algorithme naïf* **[8.1]**. *Quel est l'intérêt du schéma de Horner ?*

Q 5.b *Démontrer les propriétés* **[9.2]**.

Q 5.c *Programmer l'algorithme de Horner : l'appel* `Horner(P, x)` *doit retourner le couple $(Q, P(x))$.*

Q 5.d *Utiliser la fonction* `Horner` *pour évaluer un polynôme en un réel x.*

10. Algorithme de Ruffini

Soit P_k, un polynôme de degré d.

10.1 Le schéma de Horner permet d'expliciter $P_k(x)$ et le polynôme P_{k+1} tel que

$$P_k = P_k(x) + (X - x)P_{k+1}.$$

On a donc

$$P_k(X + x) = P_k(x) + XP_{k+1}(X + x) \qquad \text{et} \qquad \deg P_{k+1} = (\deg P_k) - 1.$$

10.2 L'algorithme de Ruffini consiste à poser $P_0 = P$ et à appliquer $(d + 1)$ fois le schéma de Horner pour calculer la famille

$$\big(P_0(x), P_1(x), \ldots, P_d(x)\big).$$

10.3 On peut déduire de **[10.1]** que $\deg P_n = (\deg P) - n$ et que

$$P(X + x) = \sum_{k=0}^{n-1} P_k(x)X^k + X^nP_n(X + x)$$

pour tout entier $1 \leqslant n \leqslant d$. En particulier, pour $n = d$,

$$P(X + x) = \sum_{k=0}^{d} P_k(x)X^k$$

puisque P_d est un polynôme constant.

10.4 D'après la formule de Taylor pour les polynômes,

$$\forall\ 0 \leqslant k \leqslant d, \qquad P^{(k)}(x) = k!\, P_k(x).$$

Q 6. *Programmer l'algorithme de Ruffini pour déduire le polynôme $P_x = P(X + x)$ du polynôme P et du scalaire x. Les polynômes*

$$P = \sum_{k=0}^{d} a_kX^{d-k} \quad \textit{et} \quad P_x = \sum_{k=0}^{d} c_kX^{d-k}$$

seront représentés de la même façon par les listes $(a_0, \ldots, a_d)$ et $(c_0, \ldots, c_d)$.

11. Localisation d'une racine

Soit une fonction polynomiale $f : \mathbb{R} \to \mathbb{R}$. On suppose que $f(0) < 0$ et que f tend vers $+\infty$ au voisinage de $+\infty$: d'après le théorème des valeurs intermédiaires, cette fonction admet (au moins) une racine positive.

Plus précisément, l'ensemble

$$\{n \in \mathbb{N} : f(n) > 0\}$$

est une partie non vide de $\mathbb{N}$. Il admet donc un plus petit élément n_0 et la fonction f possède (au moins) une racine entre $n_0 - 1$ et n_0.

Q 7. *La fonction polynomiale f est représentée par la liste* `P` *de ses coefficients en suivant la convention choisie au* **[9.1]**.

Q 7.a *Écrire une fonction* `racinePositive(P)` *qui retourne* `True` *si, et seulement si, les hypothèses du* **[11]** *sont vérifiées.*

Q 7.b *Écrire une fonction* `balayage(P)` *qui retourne un entier n tel que la fonction polynomiale associée à* `P` *admette une racine dans l'intervalle $[n, n+1]$.*
On pourra utiliser le code élaboré au **[Q5.d]**.

12. Algorithme de Horner-Ruffini

Soit f_0, une fonction polynomiale à coefficients *entiers*. On suppose que d_0 est un *entier* tel que $f_0(d_0)f_0(d_0+1) < 0$: la fonction f_0 admet une racine entre d_0 et $d_0 + 1$.

12.1 Si le degré de f_0 est égal à d, la fonction définie par

$$f_1(x) = 10^d f_0\Big(d_0 + \frac{x}{10}\Big)$$

est une fonction polynomiale de degré d à coefficients *entiers*.

12.2 La fonction f_1 change de signe entre $x = 0$ et $x = 10$, donc il existe un entier $0 \leqslant d_1 < 10$ tel que $f_1(d_1)f_1(d_1+1) < 0$: la fonction f_1 vérifie les mêmes hypothèses que la fonction f_0 et la fonction f_0 possède une racine comprise entre

$$d_0 + \frac{d_1}{10} \quad \text{et} \quad d_0 + \frac{d_1+1}{10}.$$

12.3 On peut donc itérer le procédé pour obtenir une famille d'entiers $(d_0, d_1, \ldots, d_n)$ et une famille de fonctions polynomiales $(f_0, f_1, \ldots, f_n)$ telles que, pour tout $0 \leqslant k \leqslant n$, la fonction f_k admette une racine entre d_k et d_{k+1} et donc telles que f_0 possède une racine comprise entre

$$d_0 + \sum_{k=1}^{n} \frac{d_k}{10^k} \qquad \text{et} \qquad d_0 + \sum_{k=1}^{n} \frac{d_k}{10^k} + \frac{1}{10^n}.$$

12.4 On dispose ainsi d'une méthode pour calculer de proche en proche autant de décimales qu'on souhaite d'une racine de f_0 en n'effectuant que des opérations élémentaires sur des *entiers* (additions, soustractions et multiplications).

Q 8. *Écrire une fonction* `HR(P, d0)` *qui, à la liste* `P` *des coefficients de f_0 et au scalaire* `d0`*, associe la fonction polynomiale f_1 définie au* **[12.1]**.

Q 9. Application

Écrire une fonction `rCub(a, n)` *qui retourne la partie entière et les n premières décimales de $\sqrt[3]{a}$ sous forme de liste.*

Réponses aux questions

R 1. Mises en œuvre de l'algorithme naïf

Puisqu'on dispose d'une relation de récurrence, la version récursive est la plus simple à rédiger.

```
def puissance(x, n):
  if (n==0):
    return 1
  else:
    return puissance(x, n-1)*x
```

❧ La version itérative repose sur une boucle `for` puisqu'on connaît le nombre d'itérations à effectuer : on part de $u_0 = 1$ et on effectue $u_{k+1} = x \cdot u_k$ à partir de $k = 0$ (initialisation) et jusqu'à $k + 1 = n$ (terminaison), c'est-à-dire pour $1 \leqslant k < n$.

```
def puissance(x, n):
  u = 1                # u_0
  for k in range(n):   # ∀ 0 ⩽ k < n
    u *= x             # u_{k+1} = x · u_k
  return u             # u_n
```

I Exponentiation rapide

R 2.a Comme q est le quotient de la division euclidienne de n par 2, on a $2^{p-1} \leqslant q < 2^p$.

R 2.b En notant q_k, le quotient calculé lors du k-ième appel récursif, on part de $q_0 = n$ et $q_1 = q$ et on déduit de l'encadrement précédent que

$$2^{p-k} \leqslant q_k < 2^{p+1-k}.$$

En particulier, $2^0 \leqslant q_p < 2^{(p+1)-p} = 2$, c'est-à-dire $q_p = 1$. En divisant q_p par 2, on obtient $q_p = 2 \times 0 + 1$, donc $q_{p+1} = 0$. L'algorithme s'achève alors puisque l'argument `q` prend la valeur $q_{p+1} = 0$.

En prenant $n = q_0$ pour valeur de `q` au premier appel et en prenant $0 = q_{p+1}$ pour valeur de `q` au dernier appel, on effectue en tout $(p + 2)$ appels à la fonction `puissance2`. Le dernier appel, pour lequel `q` est nul, se borne à signaler la fin du calcul.

R 2.c On effectue donc $(p + 1)$ appels avec une ou deux multiplications par appel, donc on effectue entre $(p + 1)$ et $(2p + 2)$ multiplications. Comme $p \leqslant \lg n < p + 1$, on calcule ainsi x^n en effectuant environ $\lg n$ multiplications (au lieu de n multiplications avec l'algorithme naïf).

❧ On cherche les entiers n tels que $10 \times 2 \lg n \leqslant n$. Par tâtonnements, on vérifie que cette inégalité est vérifiée à partir de $n = 150$ environ...

R 3. La première méthode consiste à calculer d'abord la représentation binaire

$$b = (b_0, b_1, \dots, b_p) \in \{0,1\}^p$$

de l'exposant n.

```
def base2(n):
  q, B = n, []          # q_0, B_{-1}
  while (q>0):          # ∀ 0 ⩽ k ⩽ p
    B.append(q%2)       # B_k = B_{k-1} + [r_k] = [r_0, r_1, ..., r_k]
    q = q//2            # q_{k+1}
  return B              # B_p = [r_0, r_1, ..., r_p]
```

Pour vérifier que cet algorithme est correct, on pose $q_0 = n$ et, *tant que* $q_k \geqslant 1$, on note q_{k+1} et r_k, le quotient et le reste de la division euclidienne de q_k par 2 :

$$q_k = 2q_{k+1} + r_k.$$

On vérifie par récurrence que

$$\forall\ 0 \leqslant k \leqslant p, \quad n = q_0 = 2^{k+1} q_{k+1} + \sum_{i=0}^{k} r_i 2^i.$$

En particulier, pour $k = p$, on a $q_{p+1} = 0$ donc

$$n = 2^{p+1} \times 0 + \sum_{i=0}^{p} r_i 2^i = \sum_{i=0}^{p} r_i 2^i.$$

La fonction `base2` retourne la famille des restes $(r_0, r_1, \dots, r_p)$, qui sont bien les chiffres l'écriture de n en base 2 : $r_k = b_k$ pour tout $0 \leqslant r \leqslant p$.

❧ On utilise ensuite la décomposition binaire de n pour calculer x^n de proche en proche.

```
def puissance2(x, n):
  L, v, y = base2(n), 1, x      # v_{-1}, y_0
  for r in L:                   # r_k pour 0 ⩽ k ⩽ p
    if r==1:                    # q_k est impair si, et seulement si, r_k = 1
      v *= y                    # v_k = v_{k-1} y_k
    y *= y                      # y_{k+1} = y_k^2
  return v
```

On part de $v_{-1} = 1$ et $y_0 = x$. On parcourt la liste des chiffres r_k qui représente n en base 2 : pour tout $0 \leqslant k \leqslant p$, on a

$$v_k = \begin{cases} v_{k-1} y_k & \text{si } r_k = 1, \\ v_{k-1} & \text{si } r_k = 0 \end{cases} \quad \text{et} \quad y_{k+1} = y_k^2.$$

On calcule ainsi de proche en proche $y_k = x^{2^k}$ et

$$\forall\ 0 \leqslant k \leqslant p, \quad v_k = \prod_{\substack{0 \leqslant i \leqslant k \\ r_i = 1}} x^{2^i}.$$

En particulier, la valeur retournée v_p est égale à x^n.

❧ On peut raccourcir le code précédent en calculant les v_k à mesure qu'on obtient les r_k (et sans avoir à conserver en mémoire la totalité des r_k).

```
def puissance2(x, n):
  v, y, q = 1, x, n     # v_{-1}, y_0, q_0
  while (q>0):          # q_k > 0 équivaut à 0 ⩽ k ⩽ p
    if (q%2==1):        # Si r_k = 1
      v *= y            # alors v_k = v_{k-1} y_k.
    y, q = y**2, q//2   # y_{k+1} = y_k^2
  return v              # v_p = x^n
```

II Applications

R 4.a La suite de Fibonacci est une suite récurrente linéaire d'ordre deux, dont l'équation caractéristique $X^2 - X - 1 = 0$ admet pour racines

$$\alpha = \frac{1+\sqrt{5}}{2} \approx 1,618 \quad \text{et} \quad \beta = \frac{1-\sqrt{5}}{2} \approx -0,618.$$

Il existe deux constantes A et B telles que

$$\forall\, n \in \mathbb{N}, \qquad F_n = A\alpha^n + B\beta^n.$$

Connaissant F_0 et F_1, on en déduit que $A + B = 1$ et que $\alpha A + \beta B = 1$. D'après les formules de Cramer,

$$A = \frac{\beta - 1}{\beta - \alpha} = \frac{5+\sqrt{5}}{10} \approx 0,724$$

et comme $\alpha > 1$ et $|\beta| < 1$,

$$F_n \sim \frac{5+\sqrt{5}}{10} \cdot \alpha^n$$

lorsque n tend vers $+\infty$.

R 4.b Initialement, la variable `F` est égale au couple (F_0, F_1). Chaque itération fait passer de (F_k, F_{k+1}) à (F_{k+1}, F_{k+2}) et l'indice `k` varie de 0 (inclus) à $(n-1)$ (exclu), donc la valeur finale de la variable `F` est le couple

$$(F_{(n-2)+1}, F_{(n-2)+2}) = (F_{n-1}, F_n).$$

La valeur retournée `F[1]` est donc bien égale à F_n.

❧ Cet algorithme effectue n additions sur des entiers de plus en plus grands : plus l'indice `k` augmente, plus ces additions sont coûteuses. Le nombre d'opérations est bien de l'ordre de n mais la complexité en temps ne l'est pas !

R 4.c On effectue environ $\lg n$ multiplications matricielles. La taille des matrices est petite, donc ce ne sont pas des opérations très coûteuses.

REMARQUE.— L'indexation est prioritaire sur la multiplication matricielle : si on remplace

`(L*B*C)[0,0]` par `L*B*C[0,0]`,

Python constate d'abord que `C[0,0]` est une matrice de $\mathfrak{M}_1(\mathbb{R})$ dont l'unique coefficient est égal à 1 et en déduit que la multiplication par `B` n'est pas une multiplication matricielle, mais une multiplication de tableaux numpy. Pour ces raisons, l'expression `L*B*C[0,0]` donne le même résultat que l'opération `L*B`.

R 4.d On obtient 2 971 215 073 avec `Fib1` et −1 323 752 223 avec `Fib2`. Le résultat fourni par `Fib2` est négatif et donc manifestement faux.
Par défaut, les matrices de numpy sont constituées de coefficients de type `'int32'`, c'est-à-dire des entiers codés sur 4 octets (soit 32 bits). On doit donc travailler sur des entiers compris entre -2^{31} (au sens large) et 2^{31} (au sens strict), l'ensemble des entiers étant muni d'une structure de groupe additif cyclique analogue à celle de $\mathbb{Z}/N\mathbb{Z}$. Comme la suite $(F_n)_{n\in\mathbb{N}}$ tend vers $+\infty$, il est inévitable qu'un dépassement de capacité se produise à partir d'un certain rang (sans le moindre message d'alerte...).

❧ On pourrait repousser l'échéance en travaillant avec des entiers codés sur 8 octets. En définissant la matrice A par

```
A = np.matrix([[0,1], [1,1]], dtype='int64')
```

on travaille avec des entiers compris entre -2^{63} et 2^{63}, ce qui permet de calculer F_{91} sans erreur. Mais F_{92} est à nouveau négatif !

❧ Si on est prêt à faire des concessions, on peut travailler avec des matrices à coefficients flottants. Dans ce cas, on n'aura que des valeurs approchées des F_n (avec une dizaine de chiffres exacts). En définissant la matrice A par

```
A = np.matrix([[0,1], [1,1]], dtype='float64')
```

on peut calculer les nombres de Fibonacci jusqu'aux alentours de $F_{1\,470}$ (qui est de l'ordre de 10^{307}). Si on va plus loin, on provoque une erreur ou on obtient `nan` pour résultat (pour *not a number*).

❧ Si on n'est prêt à aucune concession, il ne reste que deux pistes :
– abandonner numpy, c'est-à-dire remplacer les matrices par des listes de listes pour pouvoir travailler avec des entiers de taille quelconque — mais il faut alors redéfinir à la main le produit matriciel ;
– utiliser `Fib1` !
L'algorithme naïf mis en œuvre par `Fib1` n'effectue que des additions (environ n). L'algorithme de `Fib2` n'effectue que $\lg n$ multiplications, mais il s'agit de multiplications matricielles et on envisage ici que les coefficients des matrices soient des entiers arbitrairement grands : on peut parier que le coût de ces multiplications matricielles excède de loin le coût des additions faites par `Fib1`. (C'est pour que le coût des additions et des multiplications soit fixe que numpy impose de travailler avec des entiers représentés sur un nombre fixe d'octets.)

❧ En conclusion,
– ou bien on veut les F_n exacts et `Fib1` est un algorithme simple et relativement efficace ;
– ou bien on peut se contenter d'une valeur approchée et il suffit alors de calculer l'équivalent trouvé en **[R4.a]**.

III Évaluation d'un polynôme

R 5.a Les deux algorithmes ont une complexité linéaire en fonction du degré du polynôme.
Le schéma de Horner effectue le même nombre d'additions et deux fois moins de multiplications. Dans le cas où x est une matrice carrée, les deux algorithmes effectuent le même nombre de produits matriciels. Comme ces opérations sont de loin les plus coûteuses à effectuer, l'économie réalisée par le schéma de Horner est négligeable.
Le véritable avantage du schéma de Horner est qu'il donne le quotient de la division euclidienne sans autre calcul.

R 5.b D'après la définition des b_k,

$$\begin{aligned}(X-x)Q+b_d &= \sum_{k=0}^{d-1} b_k X^{d-k} - \sum_{k=1}^{d} x b_{k-1} X^{d-k} + b_d \\ &= b_0 X^d + \sum_{k=1}^{d-1} (b_k - x b_{k-1}) X^{d-k} + (b_d - x b_{d-1}) \\ &= P.\end{aligned}$$

R 5.c

```
def Horner(P, x):
    b = P[0]                # coeff dominant
    L = [b]
    for a in P[1:]:         # de a = a_1 jusqu'à a = a_d
        b = b*x+a           # b_k = x b_{k-1} + a_k
        L.append(b)
    return L[:-1], L[-1]    # liste des coefficients du quotient, reste
```

R 5.d Le plus dur est fait !

```
def evaluer(P, x):
    return Horner(P, x)[1]
```

R 6. L'algorithme de Ruffini calcule les coefficients du polynôme $P_x = P(X+x)$ en commençant par le coefficient constant c_d et en finissant par le coefficient dominant c_0. Pour que le polynôme P_x soit correctement représenté, il faut donc ajouter chaque coefficient *en tête de liste* (et non pas en queue de liste, comme le ferait la méthode `append`).

```
def Ruffini(P, x):
    p, Px = P, []           # P_0 = P
    while (p!=[]):
        p, r = Horner(p, x) # (P_{k+1}, P_k(x))
        Px = [r]+Px         # c_{d-k} = P_k(x)
    return Px
```

R 7.a La valeur de $f(0)$ est donnée par le terme constant et la limite en $+\infty$ se déduit du coefficient dominant.

```
def racinePositive(P):
    c_constant = P[-1]
    c_dominant = P[0]
    return ((c_constant<0) and (c_dominant>0))
```

R 7.b Par hypothèse, on sait que $f(0) < 0$. On calcule $f(n)$ jusqu'à ce qu'on trouve une valeur positive : le dernier entier calculé est alors n_0 et on retourne $(n_0 - 1)$.

```
def balayer(P):
    n = 1
    while (evaluer(P, n)<0):
        n += 1
    return n-1
```

La fonction `evaluer(P, x)`, qui calcule la valeur de $f(x)$, a été définie au **[R5.d]**.

R 8. L'algorithme de Ruffini nous donne les coefficients $(c_0, \dots, c_d)$ tels que

$$\forall\, t \in \mathbb{R}, \qquad f_0(d_0 + y) = \sum_{k=0}^{d} c_k y^{d-k}.$$

Par suite,

$$f_1(x) = 10^d f_0\left(d_0 + \frac{x}{10}\right) = \sum_{k=0}^{d} c_k \cdot \frac{x^{d-k}}{10^{d-k}} \cdot 10^d = \sum_{k=0}^{d} 10^k c_k \cdot x^{d-k}.$$

Il suffit donc de retourner la liste $(c_0, 10c_1, \dots, 10^k c_k, \dots, 10^d c_d)$.

```
def HR(P, d0):
    d = len(P)
    Q = Ruffini(P, d0)
    return [10**k*Q[k] for k in range(d)]
```

R 9. Application

La fonction polynomiale définie par $f(x) = x^3 - a$ vérifie les hypothèses du **[11]**.
Une première application de la fonction `balayer` permet de trouver la partie entière d_0 de $\sqrt[3]{a}$. On calcule ensuite les n décimales suivantes pour obtenir une valeur approchée de $\sqrt[3]{a}$ à 10^{-n} près par défaut.

```
def rCub(a, n):
    P = [1, 0, 0, -a]   # X^3 - a = 1·X^3 + 0·X^2 + 0·X - a
    d = balayer(P)      # Partie entière d_0
    R = [d]
    for k in range(n):  # 0 ⩽ k < n
        P = HR(P, d)    # On déduit f_{k+1} de f_k et de d_k
        d = balayer(P)  # puis la décimale d_{k+1} de f_{k+1}
        R.append(d)
    return R
```