Algorithmes et mathématiques

Vidéo ■ partie 1. Premiers pas avec Python
Vidéo ■ partie 2. Ecriture des entiers
Vidéo ■ partie 3. Calculs de sinus, cosinus, tangente
Vidéo ■ partie 4. Les réels
Vidéo ■ partie 5. Arithmétique - Algorithmes récursifs
Vidéo ■ partie 6. Polynômes - Complexité d'un algorithme

1. Premiers pas avec Python

Dans cette partie on vérifie d'abord que Python fonctionne, puis on introduira les boucles (`for` et `while`), le test `if ... else ...` et les fonctions.

1.1. Hello world !

Pour commencer testons si tout fonctionne !

Travaux pratiques 1.

1. Définir deux variables prenant les valeurs 3 et 6.
2. Calculer leur somme et leur produit.

Voici à quoi cela ressemble :

Code 1 (*hello-world.py*).

```
>>> a=3
>>> b=6
>>> somme = a+b
>>> print(somme)
9
>>> # Les résultats
>>> print("La somme est", somme)
La somme est 9
>>> produit = a*b
>>> print("Le produit est", produit)
Le produit est 18
```

On retient les choses suivantes :

- On affecte une valeur à une variable par le signe égal =.
- On affiche un message avec la fonction `print()`.
- Lorsque qu'une ligne contient un dièse #, tout ce qui suit est ignoré. Cela permet d'insérer des commentaires, ce qui est essentiel pour relire le code.

Dans la suite on omettra les symboles >>>. Voir plus de détails sur le fonctionnement en fin de section.

1.2. Somme des cubes

Travaux pratiques 2.

1. Pour un entier n fixé, programmer le calcul de la somme $S_n = 1^3 + 2^3 + 3^3 + \cdots + n^3$.
2. Définir une fonction qui pour une valeur n renvoie la somme $\Sigma_n = 1 + 2 + 3 + \cdots + n$.
3. Définir une fonction qui pour une valeur n renvoie S_n.
4. Vérifier, pour les premiers entiers, que $S_n = (\Sigma_n)^2$.

1.

Code 2 (*somme-cubes.py (1)*).
```
n = 10
somme = 0
for i in range(1,n+1):
    somme = somme + i*i*i
print(somme)
```

Voici ce que l'on fait pour calculer S_n avec $n = 10$.
- On affecte d'abord la valeur 0 à la variable `somme`, cela correspond à l'initialisation $S_0 = 0$.
- Nous avons défini une ***boucle*** avec l'instruction `for` qui fait varier i entre 1 et n.
- Nous calculons successivement S_1, S_2,... en utilisant la formule de récurrence $S_i = S_{i-1} + i^3$. Comme nous n'avons pas besoin de conserver toutes les valeurs des S_i alors on garde le même nom pour toutes les sommes, à chaque étape on affecte à `somme` l'ancienne valeur de la somme plus i^3 : `somme = somme + i*i*i`.
- `range(1,n+1)` est l'ensemble des entiers $\{1, 2, \ldots, n\}$. C'est bien les entiers **strictement inférieurs à** $n+1$. La raison est que `range(n)` désigne $\{0, 1, 2, \ldots, n-1\}$ qui contient n éléments.

2. Nous savons que $\Sigma_n = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$ donc nous n'avons pas besoin de faire une boucle :

Code 3 (*somme-cubes.py (2)*).
```
def somme_entiers(n):
    return n*(n+1)/2
```

Une ***fonction*** en informatique est similaire à une fonction mathématique, c'est un objet qui prend en entrée des variables (dites variables formelles ou variables muettes, ici n) et retourne une valeur (un entier, une liste, une chaîne de caractères,... ici $\frac{n(n+1)}{2}$).

3. Voici la fonction qui retourne la somme des cubes :

Code 4 (*somme-cubes.py (3)*).
```
def somme_cubes(n):
    somme = 0
    for i in range(1,n+1):
        somme = somme + i**3
    return somme
```

4. Et enfin on vérifie que pour les premiers entiers $S_n = \left(\frac{n(n+1)}{2}\right)^2$, par exemple pour $n = 12$:

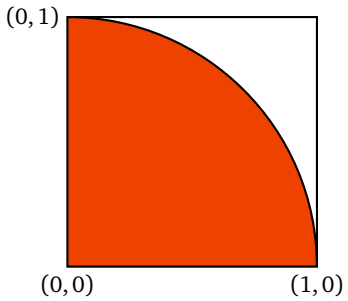
Code 5 (*somme-cubes.py (4)*).
```
n = 12
if somme_cubes(n) == (somme_entiers(n)**2):
    print("Pour␣n=", n, "l'assertion␣est␣vraie.")
else:
    print("L'assertion␣est␣fausse␣!")
```

On retient :

- Les puissances se calculent aussi avec `**` : 5^2 s'écrit `5*5` ou `5**2`, 5^3 s'écrit `5*5*5` ou `5**3`,...
- Une fonction se définit par `def ma_fonction(variable):` et se termine par `return resultat`.
- `if condition: ... else: ...` exécute le premier bloc d'instructions si la condition est vraie ; si la condition est fausse cela exécute l'autre bloc.
- Exemple de conditions
  — `a < b` : $a < b$,
  — `a <= b` : $a \leqslant b$,
  — `a == b` : $a = b$,
  — `a != b` : $a \neq b$.
- Attention ! Il est important de comprendre que `a==b` vaut soit vraie ou faux (on compare a et b) alors qu'avec `a=b` on affecte dans a la valeur de b.
- Enfin en `Python` (contrairement aux autres langages) c'est l'indentation (les espaces en début de chaque ligne) qui détermine les blocs d'instructions.

1.3. Calcul de π au hasard

Nous allons voir qu'il est possible de calculer les premières décimales de π par la méthode de Monte-Carlo, c'est à dire avec l'aide du hasard. On considère le carré de coté 1, le cercle de rayon 1 centré à l'origine, d'équation $x^2 + y^2 = 1$, et la portion de disque dans le carré (voir la figure).



Travaux pratiques 3.

1. Calculer l'aire du carré et de la portion de disque.
2. Pour un point (x, y) tiré au hasard dans le carré, quelle est la probabilité que le point soit en fait dans la portion de disque ?
3. Tirer un grand nombre de points au hasard, compter ceux qui sont dans la portion de disque.
4. En déduire les premières décimales de π.

Voici le code :

Code 6 (*pi-hasard.py*).

```
import random              # Module qui génère des nombres aléatoires

Tir = 0                    # Numéro du tir
NbTirDansLeDisque = 0      # Nombre de tirs dans le disque

while (Tir < 1000):
    Tir = Tir + 1
    # On tire au hasard un point (x,y) dans [0,1] x [0,1]
    x = random.random()
    y = random.random()
    if (x*x+y*y <= 1):     # On est dans le disque
        NbTirDansLeDisque = NbTirDansLeDisque + 1
```

```
MonPi = 4*NbTirDansLeDisque / Tir
print("Valeur␣expérimentale␣de␣Pi␣:␣%0.3f" %MonPi)
```

Commentaires :

- Un petit calcul prouve que l'aire de la portion de disque est $\frac{\pi}{4}$, l'aire du carré est 1. Donc la probabilité de tomber dans le disque est $\frac{\pi}{4}$.
- Pour tirer un nombre au hasard on utilise une fonction `random()` qui renvoie un nombre réel de l'intervalle $[0,1[$. Bien sûr à chaque appel de la fonction `random()` le nombre obtenu est différent !
- Cette fonction n'est pas connue par défaut de `Python`, il faut lui indiquer le nom du ***module*** où elle se trouve. En début de fichier on ajoute `import random` pour le module qui gère les tirages au hasard. Et pour indiquer qu'une fonction vient d'un module il faut l'appeler par `module.fonction()` donc ici `random.random()` (module et fonction portent ici le même nom !).
- La boucle est `while condition: ...` Tant que la condition est vérifiée les instructions de la boucle sont exécutées. Ici `Tir` est le compteur que l'on a initialisé à 0. Ensuite on commence à exécuter la boucle. Bien sûr la première chose que l'on fait dans la boucle est d'incrémenter le compteur `Tir`. On continue jusqu'à ce que l'on atteigne 999. Pour `Tir`$=1000$ la condition n'est plus vraie et le bloc d'instructions du `while` n'est pas exécuté. On passe aux instructions suivantes pour afficher le résultat.
- À chaque tir on teste si on est dans la portion de disque ou pas à l'aide de l'inégalité $x^2+y^2 \leqslant 1$.
- Cette méthode n'est pas très efficace, il faut beaucoup de tirs pour obtenir le deux premières décimales de π.

1.4. Un peu plus sur Python

- Le plus surprenant avec `Python` c'est que c'est ***l'indentation*** qui détermine le début et la fin d'un bloc d'instructions. Cela oblige à présenter très soigneusement le code.
- Contrairement à d'autres langages on n'a pas besoin de déclarer le type de variable. Par exemple lorsque l'on initialise une variable par `x=0`, on n'a pas besoin de préciser si x est un entier ou un réel.
- Nous travaillerons avec la version 3 (ou plus) de `Python`, que l'on appelle par `python` ou `python3`. Pour savoir si vous avez la bonne version tester la commande `4/3`. Si la réponse est `1.3333...` alors tout est ok. Par contre avec les versions 1 et 2 de `Python` la réponse est `1` (car il considérait que c'est quotient de la division euclidienne de deux entiers).
- La première façon de lancer `Python` est en ligne de commande, on obtient alors l'invite `>>>` et on tape les commandes.
- Mais le plus pratique est de sauvegarder ses commandes dans un fichier et de faire un appel par `python monfichier.py`
- Vous trouverez sans problème de l'aide et des tutoriels sur internet !

Mini-exercices. 1. Soit le produit $P_n = (1-\frac{1}{2})\times(1-\frac{1}{3})\times(1-\frac{1}{4})\times\cdots\times(1-\frac{1}{n})$. Calculer une valeur approchée de P_n pour les premiers entiers n.

2. Que vaut la somme des entiers i qui apparaissent dans l'instruction `for i in range(1,10)`. Idem pour `for i in range(11)`. Idem pour `for i in range(1,10,2)`. Idem pour `for i in range(0,10,2)`. Idem pour `for i in range(10,0,-1)`.
3. On considère le cube $[0,1]\times[0,1]\times[0,1]$ et la portion de boule de rayon 1 centrée à l'origine incluse dans ce cube. Faire les calculs de probabilité pour un point tiré au hasard dans le cube d'être en fait dans la portion de boule. Faire une fonction pour le vérifier expérimentalement.
4. On lance deux dés. Expérimenter quelle est la probabilité que la somme soit 7, puis 6, puis 3 ? Quelle est la probabilité que l'un des deux dés soit un 6 ? d'avoir un double ? La fonction `randint(a, b)` du module `random` retourne un entier k au hasard, vérifiant $a \leqslant k \leqslant b$.
5. On lance un dé jusqu'à ce que l'on obtienne un 6. En moyenne au bout de combien de lancer s'arrête-t-on ?

2. Écriture des entiers

Nous allons faire un peu d'arithmétique : le quotient de la division euclidienne `//`, le reste `%` (modulo) et nous verrons l'écriture des entiers en base 10 et en base 2. Nous utiliserons aussi la notion de listes et le module `math`.

2.1. Division euclidienne et reste, calcul avec les modulo

La division euclidienne de a par b, avec $a \in \mathbb{Z}$ et $b \in \mathbb{Z}^*$ s'écrit :

$$a = bq + r \quad \text{et} \quad 0 \leqslant r < b$$

où $q \in \mathbb{Z}$ est le ***quotient*** et $r \in \mathbb{N}$ est le ***reste***.
En `Python` le quotient se calcule par : `a // b`. Le reste se calcule par `a % b`. Exemple : `14 // 3` retourne 4 alors que `14 % 3` (lire 14 modulo 3) retourne 2. On a bien $14 = 3 \times 4 + 2$.
Les calculs avec les modulos sont très pratiques. Par exemple si l'on souhaite tester si un entier est pair, ou impair cela revient à un test modulo 2. Le code est `if (n%2 == 0): ... else: ...`. Si on besoin de calculer $\cos(n\frac{\pi}{2})$ alors il faut discuter suivant les valeurs de `n%4`.

Appliquons ceci au problème suivant :

Travaux pratiques 4.
Combien y-a-t-il d'occurrences du chiffre 1 dans les nombres de 1 à 999 ? Par exemple le chiffre 1 apparaît une fois dans 51 mais deux fois dans 131.

Code 7 (*nb-un.py*).
```
NbDeUn = 0
for N in range(1,999+1):
    ChiffreUnite = N % 10
    ChiffreDizaine = (N // 10) % 10
    ChiffreCentaine = (N // 100) % 10
    if (ChiffreUnite == 1):
        NbDeUn = NbDeUn + 1
    if (ChiffreDizaine == 1):
        NbDeUn = NbDeUn + 1
    if (ChiffreCentaine == 1):
        NbDeUn = NbDeUn + 1
print("Nombre␣d'occurences␣du␣chiffre␣'1'␣:", NbDeUn)
```

Commentaires :
- Comment obtient-on le chiffre des unités d'un entier N ? C'est le reste modulo 10, d'où l'instruction `ChiffreUnite = N % 10`.
- Comment obtient-on le chiffre des dizaines ? C'est plus délicat, on commence par effectuer la division euclidienne de N par 10 (cela revient à supprimer le chiffre des unités, par exemple si $N = 251$ alors `N // 10` retourne 25). Il ne reste plus qu'à calculer le reste modulo 10, (par exemple `(N // 10) % 10` retourne le chiffre des dizaines 5.
- Pour le chiffre des centaines on divise d'abord par 100.

2.2. Écriture des nombres en base 10

L'écriture décimale d'un nombre, c'est associer à un entier N la suite de ses chiffres $[a_0, a_1, \ldots, a_n]$ de sorte que a_i soit le i-ème chiffre de N. C'est-à-dire

$$N = a_n 10^n + a_{n-1} 10^{n-1} + \cdots + a_2 10^2 + a_1 10 + a_0 \quad \text{et } a_i \in \{0, 1, \ldots, 9\}$$

a_0 est le chiffre des unités, a_1 celui des dizaines, a_2 celui des centaines,...

Travaux pratiques 5.

1. Écrire une fonction qui à partir d'une liste $[a_0, a_1, \ldots, a_n]$ calcule l'entier N correspondant.
2. Pour un entier N fixé, combien a-t-il de chiffres ? On pourra s'aider d'une inégalité du type $10^n \leqslant N < 10^{n+1}$.

3. Écrire une fonction qui à partir de N calcule son écriture décimale $[a_0, a_1, \ldots, a_n]$.

Voici le premier algorithme :

Code 8 (*decimale.py (1)*).
```
def chiffres_vers_entier(tab):
    N = 0
    for i in range(len(tab)):
        N = N + tab[i] * (10 ** i)
    return N
```

La formule mathématique est simplement $N = a_n 10^n + a_{n-1}10^{n-1} + \cdots + a_2 10^2 + a_1 10 + a_0$. Par exemple `chiffres_vers_entier([4,3,2,1])` renvoie l'entier 1234.

Expliquons les bases sur les ***listes*** (qui s'appelle aussi des ***tableaux***)

- En Python une liste est présentée entre des crochets. Par exemple pour `tab = [4,3,2,1]` alors on accède aux valeurs par `tab[i]` : `tab[0]` vaut 4, `tab[1]` vaut 3, `tab[2]` vaut 2, `tab[3]` vaut 1.
- Pour parcourir les éléments d'un tableau le code est simplement `for x in tab`, x vaut alors successivement 4, 3, 2, 1.
- La longueur du tableau s'obtient par `len(tab)`. Pour notre exemple `len([4,3,2,1])` vaut 4. Pour parcourir toutes les valeurs d'un tableau on peut donc aussi écrire `for i in range(len(tab))`, puis utiliser `tab[i]`, ici i variant ici de 0 à 3.
- La liste vide est seulement notée avec deux crochets : `[]`. Elle est utile pour initialiser une liste.
- Pour ajouter un élément à une liste `tab` existante on utilise la fonction `append`. Par exemple définissons la liste vide `tab=[]`, pour ajouter une valeur à la fin de la liste on saisit : `tab.append(4)`. Maintenant notre liste est [4], elle contient un seul élément. Si on continue avec `tab.append(3)`. Alors maintenant notre liste a deux éléments : [4, 3].

Voici l'écriture d'un entier en base 10 :

Code 9 (*decimale.py (2)*).
```
def entier_vers_chiffres(N):
    tab = []
    n = floor(log(N,10))   # le nombre de chiffres est n+1
    for i in range(0,n+1):
        tab.append((N // 10 ** i) % 10)
    return tab
```

Par exemple `entier_vers_chiffres(1234)` renvoie le tableau $[4, 3, 2, 1]$. Nous avons expliqué tout ce dont nous avions besoin sur les listes au-dessus, expliquons les mathématiques.

- Décomposons $\mathbb{N}^*$ sous la forme $[1, 10[\cup [10, 100[\cup [100, 1000[\cup [1\,000, 10\,000[\cup \cdots$ Chaque intervalle est du type $[10^n, 10^{n+1}[$. Pour $N \in \mathbb{N}^*$ il existe donc $n \in \mathbb{N}$ tel que $10^n \leqslant N < 10^{n+1}$. Ce qui indique que le nombre de chiffres de N est $n+1$.
  Par exemple si $N = 1234$ alors $1\,000 = 10^3 \leqslant N < 10^4 = 10\,000$, ainsi $n = 3$ et le nombre de chiffres est 4.
- Comment calculer n à partir de N ? Nous allons utiliser le logarithme décimal $\log_{10}$ qui vérifie $\log_{10}(10) = 1$ et $\log_{10}(10^i) = i$. Le logarithme est une fonction croissante, donc l'inégalité $10^n \leqslant N < 10^{n+1}$ devient $\log_{10}(10^n) \leqslant \log_{10}(N) < \log_{10}(10^{n+1})$. Et donc $n \leqslant \log_{10}(N) < n+1$. Ce qui indique donc que $n = E(\log_{10}(N))$ où $E(x)$ désigne la partie entière d'un réel x.

2.3. Module `math`

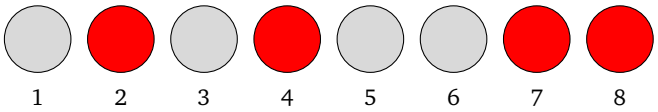
Quelques commentaires informatiques sur un module important pour nous. Les fonctions mathématiques ne sont pas définies par défaut dans Python (à part $|x|$ et x^n), il faut faire appel à une librairie spéciale : le module `math` contient les fonctions mathématiques principales.

`abs(x)`	$\lvert x\rvert$
`x ** n`	x^n
`sqrt(x)`	$\sqrt{x}$
`exp(x)`	$\exp x$
`log(x)`	$\ln x$ logarithme népérien
`log(x,10)`	$\log x$ logarithme décimal
`cos(x), sin(x), tan(x)`	$\cos x$, $\sin x$, $\tan x$ en radians
`acos(x), asin(x), atan(x)`	$\arccos x$, $\arcsin x$, $\arctan x$ en radians
`floor(x)`	partie entière $E(x)$:plus grand entier $n \leqslant x$ (*floor* = plancher)
`ceil(x)`	plus petit entier $n \geqslant x$ (*ceil* = plafond)

- Comme on aura souvent besoin de ce module on l'appelle par le code `from math import *`. Cela signifie que l'on importe toutes les fonctions de ce module et qu'en plus on n'a pas besoin de préciser que la fonction vient du module `math`. On peut écrire `cos(3.14)` au lieu `math.cos(3.14)`.
- Dans l'algorithme précédent nous avions utilisé le logarithme décimal `log(x,10)`, ainsi que la partie entière `floor(x)`.

2.4. Écriture des nombres en base 2

On dispose d'une rampe de lumière, chacune des 8 lampes pouvant être allumée (rouge) ou éteinte (gris).



On numérote les lampes de 0 à 7. On souhaite contrôler cette rampe : afficher toutes les combinaisons possibles, faire défiler une combinaison de la gauche à droite (la "chenille"), inverser l'état de toutes les lampes,... Voyons comment l'écriture binaire des nombres peut nous aider. L'*écriture binaire* d'un nombre c'est son écriture en base 2.
Comment calculer un nombre qui est écrit en binaire ? Le chiffre des "dizaines" correspond à 2 (au lieu de 10), le chiffre des "centaines" à $4 = 2^2$ (au lieu de $100 = 10^2$), le chiffres des "milliers" à $8 = 2^3$ (au lieu de $1000 = 10^3$),...
Pour le chiffre des unités cela correspond à $2^0 = 1$ (de même que $10^0 = 1$).
Par exemple 10011_b vaut le nombre 19. Car

$$10011_b = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19.$$

De façon générale tout entier $N \in \mathbb{N}$ s'écrit de manière unique sous la forme

$$N = a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_2 2^2 + a_1 2 + a_0 \quad \text{et} \quad a_i \in \{0, 1\}$$

On note alors $N = a_n a_{n-1} \ldots a_1 a_0\ _b$ (avec un indice b pour indiquer que c'est son écriture binaire).

Travaux pratiques 6.

1. Écrire une fonction qui à partir d'une liste $[a_0, a_1, \ldots, a_n]$ calcule l'entier N correspondant à l'écriture binaire $a_n a_{n-1} \ldots a_1 a_0\ _b$.
2. Écrire une fonction qui à partir de N calcule son écriture binaire sous la forme $[a_0, a_1, \ldots, a_n]$.

La seule différence avec la base 10 c'est que l'on calcule avec des puissances de 2.

Code 10 (*binaire.py (1)*).

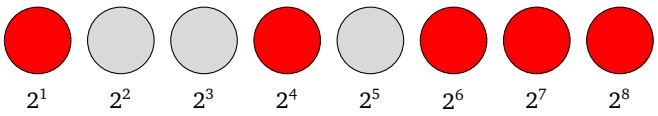
```
def binaire_vers_entier(tab):
    N = 0
    for i in range(len(tab)):
        N = N + tab[i] * (2 ** i)
```

```
    return N
```

Idem pour le sens inverse où l'on a besoin du logarithme en base 2, qui vérifie $\log_2(2) = 1$ et $\log_2(2^i) = i$.

Code 11 (*binaire.py (2)*).
```
def entier_vers_binaire(N):
    tab = []
    n = floor(log(N,2))   # le nombre de chiffres est n+1
    for i in range(0,n+1):
        tab.append((N // 2 ** i) % 2)
    return tab
```

Maintenant appliquons ceci à notre problème de lampes. Si une lampe est allumée on lui attribut 1, et si elle est éteinte 0. Pour une rampe de 8 lampes on code $[a_0, a_1, \ldots, a_7]$ l'état des lampes.
Par exemple la configuration suivante :



est codé $[1,0,0,1,0,1,1,1]$ ce qui correspond au nombre binaire $11101001_b = 233$.

Travaux pratiques 7.

1. Faire une boucle qui affiche toutes les combinaisons possibles (pour une taille de rampe donnée).
2. Quelle opération mathématique élémentaire transforme un nombre binaire $a_n \ldots a_1 a_0{}_b$ en $a_n \ldots a_1 a_0 0{}_b$ (décalage vers la gauche et ajout d'un 0 à la fin) ?
3. Soit $N' = a_n a_{n-1} \ldots a_1 a_0 0{}_b$ (une écriture avec $n+2$ chiffres). Quelle est l'écriture binaire de N' (mod 2^{n+1}) ? (C'est une écriture avec $n+1$ chiffres.)
4. En déduire un algorithme qui pour une configuration donnée de la rampe, fait permuter cycliquement (vers la droite) cette configuration. Par exemple $[1,0,1,0,1,1,1,0]$ devient $[0,1,0,1,0,1,1,1]$.
5. Quelle opération mathématique élémentaire permet de passer d'une configuration à son opposée (une lampe éteinte s'allume, et réciproquement). Par exemple si la configuration était $[1,0,1,0,1,1,1,0]$ alors on veut $[0,1,0,1,0,0,0,1]$. (Indication : sur cet exemple calculer les deux nombres correspondants et trouver la relation qui les lie.)

1. Il s'agit d'abord d'afficher les configurations. Par exemple si l'on a 4 lampes alors les configurations sont $[0,0,0,0]$, $[1,0,0,0]$, $[0,1,0,0]$, $[1,1,0,0]$,..., $[1,1,1,1]$. Pour chaque lampe nous avons deux choix (allumé ou éteint), il y a $n+1$ lampes donc un total de 2^{n+1} configurations. Si l'on considère ces configurations comme des nombres écrits en binaire alors l'énumération ci-dessus correspond à compter $0, 1, 2, 3, \ldots, 2^{n+1} - 1$.
   D'où l'algorithme :

   Code 12 (*binaire.py (3)*).
   ```
   def configurations(n):
       for N in range(2**(n+1)):
           print(entier_vers_binaire_bis(N,n))
   ```

   Où `entier_vers_binaire_bis(N,n)` est similaire à `entier_vers_binaire(N)`, mais en affichant aussi les zéros non significatifs, par exemple 7 en binaire s'écrit 111_b, mais codé sur 8 chiffres on ajoute devant des 0 non significatifs : 00000111_b.
2. En écriture décimale, multiplier par 10 revient à décaler le nombre initial et rajouter un zéro. Par exemple $10 \times 19 = 190$. C'est la même chose en binaire ! Multiplier un nombre par 2 revient sur l'écriture à un décalage vers la gauche et ajout d'un zéro sur le chiffre des unités. Exemple : $19 = 10011_b$ et $2 \times 19 = 38$ donc $2 \times 10011_b = 100110_b$.

3. Partant de $N = a_n a_{n-1} \dots a_1 a_0\ _b$. Notons $N' = 2N$, son écriture est $N' = a_n a_{n-1} \dots a_1 a_0 0\ _b$. Alors $N' \pmod{2^{n+1}}$ s'écrit exactement $a_{n-1} a_{n-2} \dots a_1 a_0 0\ _b$ et on ajoute a_n qui est le quotient de N' par 2^{n+1}.
   Preuve : $N' = a_n \cdot 2^{n+1} + a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2$. Donc $N' \pmod{2^{n+1}} = a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2$. Donc $N' \pmod{2^{n+1}} + a_n = a_{n-1} \cdot 2^n + \dots + a_0 \cdot 2 + a_n$.
4. Ainsi l'écriture en binaire de $N' \pmod{2^{n+1}} + a_n$ s'obtient comme permutation circulaire de celle de N. D'où l'algorithme :

Code 13 (*binaire.py (4)*).
```
def decalage(tab):
    N = binaire_vers_entier(tab)
    n = len(tab)-1   # le nombre de chiffres est n+1
    NN = 2*N % 2**(n+1) + 2*N // 2**(n+1)
    return entier_vers_binaire_bis(NN,n)
```

5. On remarque que si l'on a deux configurations opposées alors leur somme vaut $2^{n+1} - 1$: par exemple avec $[1,0,0,1,0,1,1,1]$ et $[0,1,1,0,1,0,0,0]$, les deux nombres associés sont $N = 11101001_b$ et $N' = 00010110_b$ (il s'agit juste de les réécrire de droite à gauche). La somme est $N + N' = 11101001_b + 00010110_b = 11111111_b = 2^8 - 1$. L'addition en écriture binaire se fait de la même façon qu'en écriture décimale et ici il n'y a pas de retenue. Si M est un nombre avec $n+1$ fois le chiffres 1 alors $M + 1 = 2^{n+1}$. Exemple si $M = 11111_b$ alors $M + 1 = 100000_b = 2^5$; ainsi $M = 2^5 - 1$. Donc l'opposé de N est $N' = 2^{n+1} - 1 - N$ (remarquez que dans $\mathbb{Z}/(2^{n+1} - 1)\mathbb{Z}$ alors $N' \equiv -N$).
   Cela conduit à :

Code 14 (*binaire.py (5)*).
```
def inversion(tab):
    N = binaire_vers_entier(tab)
    n = len(tab)-1   # le nombre de chiffres est n+1
    NN = 2**(n+1)-1 - N
    return entier_vers_binaire_bis(NN,n)
```

Mini-exercices. 1. Pour un entier n fixé, combien y-a-t-il d'occurrences du chiffre 1 dans l'écriture des nombres de 1 à n ?

2. Écrire une fonction qui calcule l'écriture décimale d'un entier, sans recourir au log (une boucle `while` est la bienvenue).
3. Écrire un algorithme qui permute cycliquement une configuration de rampe vers la droite.
4. On dispose de $n+1$ lampes, chaque lampe peut s'éclairer de trois couleurs : vert, orange, rouge (dans cet ordre). Trouver toutes les combinaisons possibles. Comment passer toutes les lampes à la couleur suivante ?
5. Générer toutes les matrices 4×4 n'ayant que des 0 et des 1 comme coefficients. On codera une matrice sous la forme de lignes $[[1,1,0,1],[0,0,1,0],[1,1,1,1],[0,1,0,1]]$.
6. On part du point $(0,0) \in \mathbb{Z}^2$. A chaque pas on choisit au hasard un direction Nord, Sud, Est, Ouest. Si on va au Nord alors on ajoute $(0,1)$ à sa position (pour Sud on ajoute $(0,-1)$; pour Est $(1,0)$; pour Ouest $(-1,0)$). Pour un chemin d'une longueur fixée de n pas, coder tous les chemins possibles. Caractériser les chemins qui repassent par l'origine. Calculer la probabilité p_n de repasser par l'origine. Que se passe-t-il lorsque $n \to +\infty$?
7. Écrire une fonction, qui pour un entier N, affiche son écriture en chiffres romains : $M = 1000$, $D = 500$, $C = 100$, $X = 10$, $V = 5$, $I = 1$. Il ne peut y avoir plus de trois symboles identiques à suivre.

3. Calculs de sinus, cosinus, tangente

Le but de cette section est le calcul des sinus, cosinus, et tangente d'un angle par nous même, avec une précision de 8 chiffres après la virgule.

3.1. Calcul de $\mathrm{Arctan}\, x$

Nous aurons besoin de calculer une fois pour toute $\mathrm{Arctan}(10^{-i})$, pour $i = 0, \dots, 8$, c'est-à-dire que l'on cherche les angles $\theta_i \in]-\frac{\pi}{2}, \frac{\pi}{2}[$ tels que $\tan\theta_i = 10^{-i}$. Nous allons utiliser la formule :

$$\mathrm{Arctan}\, x = \sum_{k=0}^{+\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Travaux pratiques 8.

1. Calculer $\mathrm{Arctan}\, 1$.
2. Calculer $\theta_i = \mathrm{Arctan}\, 10^{-i}$ (avec 8 chiffres après la virgule) pour $i = 1, \dots, 8$.
3. Pour quelles valeurs de i, l'approximation $\mathrm{Arctan}\, x \simeq x$ était-elle suffisante ?

Code 15 (*tangente.py (1)*).

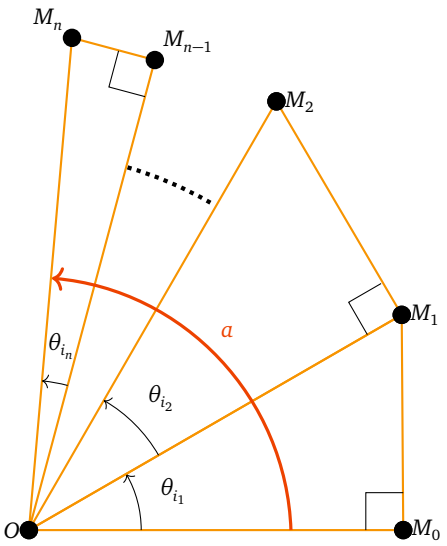
```
def mon_arctan(x,n):
    somme = 0
    for k in range(0,n+1):
        if (k%2 == 0):  # si k est pair signe +
            somme = somme + 1/(2*k+1) * (x ** (2*k+1))
        else:           # si k est impair signe -
            somme = somme - 1/(2*k+1) * (x ** (2*k+1))
    return somme
```

- La série qui permet de calculer $\mathrm{Arctan}\, x$ est une somme infinie, mais si x est petit alors chacun des termes $(-1)^k \frac{x^{2k+1}}{2k+1}$ est très très petit dès que k devient grand. Par exemple si $0 \leqslant x \leqslant \frac{1}{10}$ alors $x^{2k+1} \leqslant \frac{1}{10^{2k+1}}$ et donc pour $k \geqslant 4$ nous aurons $\left|(-1)^k \frac{x^{2k+1}}{2k+1}\right| < 10^{-9}$. Chacun des termes suivants ne contribue pas aux 8 premiers chiffres après la virgule. Attention : il se pourrait cependant que la somme de beaucoup de termes finissent par y contribuer, mais ce n'est pas le cas ici (c'est un bon exercice de le prouver).
- Dans la pratique on calcule la somme à un certain ordre $2k+1$ jusqu'à ce que les 8 chiffres après la virgules ne bougent plus. Et en fait on s'aperçoit que l'on a seulement besoin d'utiliser $\mathrm{Arctan}\, x \simeq x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7}$.
- Pour $i \geqslant 4$, $\mathrm{Arctan}\, x \simeq x$ donne déjà 8 chiffres exacts après la virgule !

On remplit les valeurs des angles θ_i obtenus dans une liste nommée `theta`.

3.2. Calcul de $\tan x$

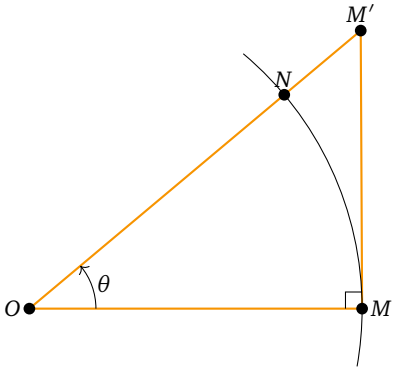
Le principe est le suivant : on connaît un certain nombre d'angles avec leur tangente : les angles θ_i (calculés ci-dessus) avec par définition $\tan\theta_i = 10^{-i}$. Fixons un angle $a \in [0, \frac{\pi}{2}]$. Partant du point $M_0 = (1, 0)$, nous allons construire des points $M_1, M_2, \dots, M_n$ jusqu'à ce que M_n soit (à peu près) sur la demi-droite correspondant à l'angle a. Si M_n a pour coordonnées (x_n, y_n) alors $\tan a = \frac{y_n}{x_n}$. L'angle pour passer d'un point M_k à M_{k+1} est l'un des angles θ_i.

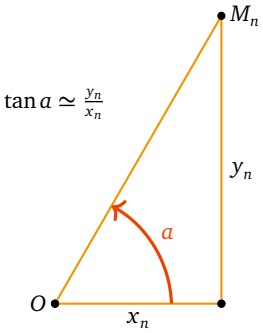


Rappelons que si l'on a un point $M(x,y)$ alors la rotation centrée à l'origine et d'angle θ envoie $M(x,y)$ sur le point $N(x',y')$ avec

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{c'est-à-dire} \quad \begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

Pour un point M, on note M' le point de la demi-droite $[ON)$ tel que les droites (OM) et (MM') soient perpendiculaires en M.





Travaux pratiques 9.

1. (a) Calculer la longueur OM'.
   (b) En déduire les coordonnées de M'.
   (c) Exprimez-les uniquement en fonction de x, y et $\tan\theta$.
2. Faire une boucle qui décompose l'angle a en somme d'angles θ_i (à une précision de 10^{-8} ; avec un minimum d'angles, les angles pouvant se répéter).
3. Partant de $M_0 = (1,0)$ calculer les coordonnées des différents M_k, jusqu'au point $M_n(x_n, y_n)$ correspondant à l'approximation de l'angle a. Renvoyer la valeur $\frac{y_n}{x_n}$ comme approximation de $\tan a$.

Voici les préliminaires mathématiques :

- Dans le triangle rectangle OMM' on a $\cos\theta = \frac{OM}{OM'}$ donc $OM' = \frac{OM}{\cos\theta}$.
- D'autre part comme la rotation d'angle θ conserve les distances alors $OM = ON$. Si les coordonnées de M' sont (x'', y'') alors $x'' = \frac{1}{\cos\theta}x'$ et $y'' = \frac{1}{\cos\theta}y'$.
- Ainsi

$$\begin{cases} x'' = \frac{1}{\cos\theta}x' = \frac{1}{\cos\theta}\big(x\cos\theta - y\sin\theta\big) = x - y\tan\theta \\ y'' = \frac{1}{\cos\theta}y' = \frac{1}{\cos\theta}\big(x\sin\theta + y\cos\theta\big) = x\tan\theta + y \end{cases}$$

  Autrement dit :

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Voici une boucle simple pour décomposer l'angle θ : on commence par retirer le plus grand angle θ_0 autant de fois que l'on peut, lorsque ce n'est plus possible on passe à l'angle θ_1,...

Code 16 (*tangente.py (2)*).
```
i = 0
while (a > precision):      # boucle tant que la precision pas atteinte
    while (a < theta[i]):   # choix du bon angle theta_i à soustraire
        i = i+1
    a = a - theta[i]        # on retire l'angle theta_i et on recommence
```

Ici `precision` est la précision souhaité (pour nous 10^{-9}). Et le tableau `theta` contient les valeurs des angles θ_i.
Posons $x_0 = 1$, $y_0 = 0$ et $M_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$. Alors on définit par récurrence $M_{k+1} = P(\theta_i) \cdot M_k$ où $P(\theta) = \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix}$.
Les θ_i sont ceux apparaissant dans la décomposition de l'angle en somme de θ_i, donc on connaît $\tan\theta_i = 10^{-i}$. Ainsi si l'on passe d'un point M_k à M_{k+1} par un angle θ_i on a simplement :

$$\begin{cases} x_{k+1} = x_k - y_k \cdot 10^{-i} \\ y_{k+1} = x_k \cdot 10^{-i} + y_k \end{cases}$$

La valeur $\frac{y_n}{x_n}$ est la tangente de la somme des angles θ_i, donc une approximation de $\tan a$.
Le code est maintenant le suivant.

Code 17 (*tangente.py (3)*).
```
def ma_tan(a):
    precision = 10**(-9)
    i = 0 ; x = 1 ; y = 0
    while (a > precision):
        while (a < theta[i]):
            i = i+1
        newa = a - theta[i]        # on retire l'angle theta_i
        newx = x - (10**(-i))*y    # on calcule le nouveau point
        newy = (10**(-i))*x + y
        x = newx
        y = newy
        a = newa
    return y/x                     # on renvoie la tangente
```

Commentaires pour conclure :

- En théorie il ne faut pas confondre «précision» et «nombre de chiffres exacts après la virgule». Par exemple 0.999 est une valeur approchée de 1 à 10^{-3} près, mais aucun chiffre après la virgule n'est exact. Dans la pratique c'est la précision qui importe plus que le nombre de chiffres exacts.
- Notez à quel point les opérations du calcul de $\tan x$ sont simples : il n'y a quasiment que des additions à effectuer. Par exemple l'opération $x_{k+1} = x_k - y_k \cdot 10^{-i}$ peut être fait à la main : multiplier par 10^{-i} c'est juste décaler la virgule à droite de i chiffres, puis on additionne. C'est cet algorithme «CORDIC» qui est implémenté dans les calculatrices, car il nécessite très peu de ressources. Bien sûr, si les nombres sont codés en binaire on remplace les 10^{-i} par 2^{-i} pour n'avoir qu'à faire des décalages à droite.

3.3. Calcul de $\sin x$ et $\cos x$

Travaux pratiques 10.
Pour $0 \leqslant x \leqslant \frac{\pi}{2}$, calculer $\sin x$ et $\cos x$ en fonction de $\tan x$. En déduire comment calculer les sinus et cosinus de x.

Solution : On sait $\cos^2 + \sin^2 x = 1$, donc en divisant par $\cos^2 x$ on trouve $1 + \tan^2 x = \frac{1}{\cos^2 x}$. On en déduit que pour $0 \leqslant x \leqslant \frac{\pi}{2}$ $\cos x = \frac{1}{\sqrt{1+\tan^2 x}}$. On trouve de même $\sin x = \frac{\tan x}{\sqrt{1+\tan^2 x}}$.

Donc une fois que l'on a calculé $\tan x$ on en déduit $\sin x$ et $\cos x$ par un calcul de racine carrée. Attention c'est valide car x est compris entre 0 et $\frac{\pi}{2}$. Pour un x quelconque il faut se ramener par les formules trigonométriques à l'intervalle $[0, \frac{\pi}{2}]$.

Mini-exercices. 1. On dispose de billets de 1, 5, 20 et 100 euros. Trouvez la façon de payer une somme de n euros avec le minimum de billets.

2. Faire un programme qui pour **n'importe quel** $x \in \mathbb{R}$, calcule $\sin x$, $\cos x$, $\tan x$.
3. Pour $t = \tan \frac{x}{2}$ montrer que $\tan x = \frac{2t}{1-t^2}$. En déduire une fonction qui calcule $\tan x$. (Utiliser que pour x assez petit $\tan x \simeq x$).
4. Modifier l'algorithme de la tangente pour qu'il calcule aussi directement le sinus et le cosinus.

4. Les réels

Dans cette partie nous allons voir différentes façons de calculer la constante γ d'Euler. C'est un nombre assez mystérieux car personne ne sait si γ est un nombre rationnel ou irrationnel. Notre objectif est d'avoir le plus de décimales possibles après la virgule en un minimum d'étapes. Nous verrons ensuite comment les ordinateurs stockent les réels et les problèmes que cela engendre.

4.1. Constante γ d'Euler

Considérons la ***suite harmonique*** :

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

et définissons

$$u_n = H_n - \ln n.$$

Cette suite (u_n) admet une limite lorsque $n \to +\infty$: c'est la ***constante γ d'Euler***.

Travaux pratiques 11.

1. Calculer les premières décimales de γ. Sachant que $u_n - \gamma \sim \frac{1}{2n}$, combien de décimales exactes peut-on espérer avoir obtenues ?
2. On considère $v_n = H_n - \ln\left(n + \frac{1}{2} + \frac{1}{24n}\right)$. Sachant $v_n - \gamma \sim -\frac{1}{48n^3}$, calculer davantage de décimales.

Code 18 (*euler.py (1)*).

```
def euler1(n):
    somme = 0
    for i in range(n,0,-1):
        somme = somme + 1/i
    return somme - log(n)
```

Code 19 (*euler.py (2)*).

```
def euler2(n):
    somme = 0
    for i in range(n,0,-1):
        somme = somme + 1/i
    return somme - log(n+1/2+1/(24*n))
```

Vous remarquez que la somme est calculée à partir de la fin. Nous expliquerons pourquoi en fin de section.

4.2. 1000 décimales de la constante d'Euler

Il y a deux techniques pour obtenir plus de décimales : (i) pousser plus loin les itérations, mais pour avoir 1000 décimales de γ les méthodes précédentes sont insuffisantes ; (ii) trouver une méthode encore plus efficace. C'est ce que nous allons voir avec la méthode de Bessel modifiée.
Soit

$$w_n = \frac{A_n}{B_n} - \ln n \quad \text{avec} \quad A_n = \sum_{k=1}^{E(\alpha n)} \left(\frac{n^k}{k!}\right)^2 H_k \quad \text{et} \quad B_n = \sum_{k=0}^{E(\alpha n)} \left(\frac{n^k}{k!}\right)^2$$

où $\alpha = 3.59112147\ldots$ est la solution de $\alpha(\ln\alpha - 1) = 1$ et $E(x)$ désigne la partie entière. Alors

$$|w_n - \gamma| \leqslant \frac{C}{e^{4n}}$$

où C est une constante (non connue).

Travaux pratiques 12.

1. Programmer cette méthode.
2. Combien d'itérations faut-il pour obtenir 1000 décimales ?
3. Utiliser le module `decimal` pour les calculer.

Voici le code :

Code 20 (*euler.py (3)*).

```
def euler3(n):
    alpha = 3.59112147
    N = floor(alpha*n)      # Borne des sommes
    A = 0 ; B = 0
    H = 0
    for k in range(1,N+1):
        c = ( (n**k)/factorial(k) ) ** 2   # Coefficient commun
        H = H + 1/k                        # Somme harmonique
        A = A + c*H
        B = B + c
    return A/B - log(n)
```

Pour obtenir N décimales il faut résoudre l'inéquation $\frac{C}{e^{4n}} \leqslant \frac{1}{10^N}$. On «passe au log» pour obtenir $n \geqslant \frac{N\ln(10)+\ln(C)}{4}$. On ne connaît pas C mais ce n'est pas si important. Moralement pour une itération de plus on obtient (à peu près) une décimale de plus (c'est-à-dire un facteur 10 sur la précision !). Pour $n \geqslant 800$ on obtient 1000 décimales exactes de la constante d'Euler :

0,
5772156649015328606065120900824024310421593359399235988057672348848677267776646709369470632917467495
1463144724980708248096050401448654283622417399764492353625350033374293733773767394279259525824709491
6008735203948165670853233151776611528621199501507984793745085705740029921354786146694029604325421519
0587755352673313992540129674205137541395491116851028079842348775872050384310939973613725530608893312
6760017247953783675927135157722610273492913940798430103417771778088154957066107501016191663340152278
9358679654972520362128792265559536696281763887927268013243101047650596370394739495763890657296792960
1009015125195950922243501409349871228247949747195646976318506676129063811051824197444867836380861749
4551698927923018773910729457815543160050021828440960537724342032854783670151773943987003023703395183
2869000155819398804270741154222781971652301107356583396734871765049194181230004065469314299929777956
9303100503086303418569803231083691640025892970890985486825777364288253954925873629596133298574739302

4.3. Un peu de réalité

En mathématique un réel est un élément de $\mathbb{R}$ et son écriture décimale est souvent infinie après la virgule : par exemple $\pi = 3,14159265\ldots$ Mais bien sûr un ordinateur ne peut pas coder une infinité d'informations. Ce qui se rapproche d'un réel est un ***nombre flottant*** dont l'écriture est :

$$\underbrace{\pm 1,234567890123456789}_{\text{mantisse}}\ e\ \underbrace{\pm 123}_{\text{exposant}}$$

pour $\pm 1,234\ldots \times 10^{\pm 123}$. La ***mantisse*** est un nombre décimal (positif ou négatif) appartenant à $[1,10[$ et l'exposant est un entier (lui aussi positif ou négatif). En `Python` la mantisse à une précision de 16 chiffres après la virgule. Cette réalité informatique fait que des erreurs de calculs peuvent apparaître même avec des opérations simples. Pour voir un exemple de problème faites ceci :

Travaux pratiques 13.
Poser $x = 10^{-16}$, $y = x + 1$, $z = y - 1$. Que vaut z pour `Python` ?

Comme `Python` est très précis nous allons faire une routine qui permet de limiter drastiquement le nombre de chiffres et mettre en évidence les erreurs de calculs.

Travaux pratiques 14.

1. Calculer l'exposant d'un nombre réel. Calculer la mantisse.
2. Faire une fonction qui ne conserve que 6 chiffres d'un nombre (6 chiffres en tout : avant + après la virgule, exemple $123,456789$ devient $123,456$).

Voici le code :

Code 21 (*reels.py (1)*).
```
precision = 6                 # Nombre de décimales conservées
def tronquer(x):
    n = floor(log(x,10))                      # Exposant
    m = floor( x * 10 ** (precision-1 - n))   # Mantisse
    return m * 10 ** (-precision+1+n)         # Nombre tronqué
```

Comme on l'a déjà vu auparavant l'exposant se récupère à l'aide du logarithme en base 10. Et pour tronquer un nombre avec 6 chiffres, on commence par le décaler vers la gauche pour obtenir 6 chiffres avant la virgule ($123,456789$ devient $123456,789$) il ne reste plus qu'à prendre la partie entière (123456) et le redécaler vers la droite (pour obtenir $123,456$).

Absorption

Travaux pratiques 15.

1. Calculer `tronquer(1234.56 + 0.007)`.
2. Expliquer.

Chacun des nombres $1234,56$ et $0,007$ est bien un nombre s'écrivant avec moins de 6 décimales mais leur somme $1234,567$ a besoin d'une décimale de plus, l'ordinateur ne retient pas la 7-ème décimale et ainsi le résultat obtenu est $1234,56$. Le $0,007$ n'apparaît pas dans le résultat : il a été victime d'une ***absorption***.

Élimination

Travaux pratiques 16.

1. Soient $x = 1234,8777$, $y = 1212,2222$. Calculer $x - y$ à la main. Comment se calcule la différence $x - y$ avec notre précision de 6 chiffres ?
2. Expliquer la différence.

Comme $x - y = 22,6555$ qui n'a que 6 chiffres alors on peut penser que l'ordinateur va obtenir ce résultat. Il n'en est rien, l'ordinateur ne stocke pas x mais `tronquer(x)`, idem pour y. Donc l'ordinateur effectue en fait le calcul suivant : `tronquer(tronquer(x)-tronquer(y))`, il calcule donc $1234,87 - 1212,22 = 22,65$. Quel est le problème ? C'est qu'ensuite l'utilisateur considère –à tort– que le résultat est calculé avec une précision de 6 chiffres. Donc on peut penser que le résultat est $22,6500$ mais les 2 derniers chiffres sont une pure invention.
C'est un phénomène d'***élimination***. Lorsque l'on calcule la différence de deux nombres proches, le résultat a en fait une précision moindre. Cela peut être encore plus dramatique avec l'exemple $\delta = 1234,569 - 1234,55$ la différence est $0,01900$ alors que l'ordinateur retournera $0,01000$. Il y a presque un facteur deux, et on aura des problèmes si l'on a besoin de diviser par δ.

Signalons au passage une erreur d'interprétation fréquente : ne pas confondre la ***précision*** d'affichage (exemple : on calcule avec 10 chiffres après la virgule) avec l'***exactitude*** du résultat (combien de décimales sont vraiment exactes ?).

Conversion binaire – décimale

Enfin le problème le plus troublant est que les nombres flottants sont stockés en écriture binaire et pas en écriture décimale.

Travaux pratiques 17.
Effectuer les commandes suivantes et constater !

1. `sum = 0` puis `for i in range(10): sum = sum + 0.1`. Que vaut `sum` ?
2. `0.1 + 0.1 == 0.2` et `0.1 + 0.1 + 0.1 == 0.3`
3. `x = 0.2 ; print("0.2␣en␣Python␣=␣%.25f" %x)`

La raison est simple mais néanmoins troublante. L'ordinateur ne stocke pas $0,1$, ni $0,2$ en mémoire mais le nombre en écriture binaire qui s'en rapproche le plus.
En écriture décimale, il est impossible de coder $1/3 = 0,3333\ldots$ avec un nombre fini de chiffres après la virgule. Il en va de même ici : l'ordinateur ne peut pas stocker exactement $0,2$. Il stocke un nombre en écriture binaire qui s'en rapproche le plus ; lorsqu'on lui demande d'afficher le nombre stocké, il retourne l'écriture décimale qui se rapproche le plus du nombre stocké, mais ce n'est plus $0,2$, mais un nombre très très proche :

$$0.20000000000000000111022302\ldots$$

4.4. Somme des inverses des carrés

Voyons une situation concrète où ces problèmes apparaissent.

Travaux pratiques 18.

1. Faire une fonction qui calcule la somme $S_n = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$.
2. Faire une fonction qui calcule cette somme mais en utilisant seulement une écriture décimale à 6 chiffres (à l'aide de la fonction `tronquer()` vue au-dessus).
3. Reprendre cette dernière fonction, mais en commençant la somme par les plus petits termes.
4. Comparez le deux dernières méthodes, justifier et conclure.

La première fonction ne pose aucun problème et utilise toute la précision de `Python`.
Dans la seconde on doit, à chaque calcul, limiter notre précision à 6 chiffres (ici 1 avant la virgule et 5 après).

Code 22 (*reels.py (2)*).

```
def somme_inverse_carres_tronq(n):
  somme = 0
```

```
    for i in range(1,n+1):
        somme = tronquer(somme + tronquer(1/(i*i)))
    return somme
```

Il est préférable de commencer la somme par la fin :

Code 23 (*reels.py (3)*).
```
def somme_inverse_carres_tronq_inv(n):
    somme = 0
    for i in range(n,0,-1):
        somme = tronquer(somme + tronquer(1/(i*i)))
    return somme
```

Par exemple pour $n = 100\,000$ l'algorithme `somme_inverse_carres_tronq()` (avec écriture tronquée, sommé dans l'ordre) retourne $1,64038$ alors que l'algorithme `somme_inverse_carres_tronq_inv()` (avec la somme dans l'ordre inverse) on obtient $1,64490$. Avec une précision maximale et n très grand on doit obtenir $1,64493\ldots$ (en fait c'est $\frac{\pi^2}{6}$).
Notez que faire grandir n pour l'algorithme `somme_inverse_carres_tronq()` n'y changera rien, il bloque à 2 décimales exactes après la virgule : $1,64038$! La raison est un phénomène d'absorption : on rajoute des termes très petits devant une somme qui vaut plus de 1. Alors que si l'on part des termes petits, on ajoute des termes petits à une somme petite, on garde donc un maximum de décimales valides avant de terminer par les plus hautes valeurs.

Mini-exercices. 1. Écrire une fonction qui approxime la constante α qui vérifie $\alpha(\ln\alpha - 1) = 1$. Pour cela poser $f(x) = x(\ln x - 1) - 1$ et appliquer la méthode de Newton : fixer u_0 (par exemple ici $u_0 = 4$) et $u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$.

2. Pour chacune des trois méthodes, calculer le nombre approximatif d'itérations nécessaires pour obtenir 100 décimales de la constante γ d'Euler.

3. Notons $C_n = \frac{1}{4n}\sum_{k=0}^{2n} \frac{[(2k)!]^3}{(k!)^4(16n)^{2k}}$. La formule de Brent-McMillan affirme $\gamma = \frac{A_n}{B_n} - \frac{C_n}{B_n^2} - \ln n + O(\frac{1}{e^{8n}})$ où cette fois les sommations pour A_n et B_n vont jusqu'à $E(\beta n)$ avec $\beta = 4,970625759\ldots$ la solution de $\beta(\ln\beta - 1) = 3$. La notation $O(\frac{1}{e^{8n}})$ indique que l'erreur est $\leqslant \frac{C}{e^{8n}}$ pour une certaine constante C. Mettre en œuvre cette formule. En 1999 cette formule a permis de calculer 100 millions de décimales. Combien a-t-il fallu d'itérations ?

4. Faire une fonction qui renvoie le terme u_n de la suite définie par $u_0 = \frac{1}{3}$ et $u_{n+1} = 4u_n - 1$. Que vaut u_{100} ? Faire l'étude mathématique et commenter.

5. Arithmétique – Algorithmes récursifs

Nous allons présenter quelques algorithmes élémentaires en lien avec l'arithmétique. Nous en profitons pour présenter une façon complètement différente d'écrire des algorithmes : les fonctions récursives.

5.1. Algorithmes récursifs

Voici un algorithme très classique :

Code 24 (*recursif.py (1)*).
```
def factorielle_classique(n):
    produit = 1
    for i in range(1,n+1):
        produit = i * produit
    return produit
```

Voyons comment fonctionne cette boucle. On initialise la variable `produit` à 1, on fait varier un indice i de 1 à n. À chaque étape on multiplie `produit` par i et on affecte le résultat dans `produit`. Par exemple si $n = 5$ alors la

variable `produit` s'initialise à 1, puis lorsque i varie la variable produit devient $1 \times 1 = 1$, $2 \times 1 = 2$, $3 \times 2 = 6$, $4 \times 6 = 24$, $5 \times 24 = 120$. Vous avez bien sûr reconnus le calcul de 5!

Étudions un autre algorithme.

Code 25 (*recursif.py (2)*).
```
def factorielle(n):
    if (n==1):
        return 1
    else:
        return n * factorielle(n-1)
```

Que fait cet algorithme ? Voyons cela pour $n = 5$. Pour $n = 5$ la condition du «si» (if) n'est pas vérifiée donc on passe directement au «sinon» (else). Donc `factorielle(5)` renvoie comme résultat : `5 * factorielle(4)`. On a plus ou moins progressé : le calcul n'est pas fini car on ne connaît pas encore `factorielle(4)` mais on s'est ramené à un calcul au rang précédent, et on itère :
`factorielle(5) = 5 * factorielle(4) = 5 * 4 * factorielle(3) = 5 * 4 * 3 * factorielle(2)` et enfin `factorielle(5) = 5 * 4 * 3 * 2 * factorielle(1)`. Pour `factorielle(1)` la condition du `if (n==1)` est vérifiée et alors `factorielle(1)=1`. Le bilan est donc que `factorielle(5) = 5 * 4 * 3 * 2 * 1` c'est bien 5!
Une fonction qui lorsque elle s'exécute s'appelle elle-même est une ***fonction récursive***. Il y a une analogie très forte avec la récurrence. Par exemple on peut définir la suite des factorielles ainsi :

$$u_1 = 1 \quad \text{et} \quad u_n = n \times u_{n-1} \text{ si } n \geqslant 2.$$

Nous avons ici $u_n = n!$ pour tout $n \geqslant 1$.
Comme pour la récurrence une fonction récursive comporte une étape d'***initialisation*** (ici `if (n==1): return 1` correspondant à $u_1 = 1$) et une étape d'***hérédité*** (ici `return n * factorielle(n-1)` correspondant à $u_n = n \times u_{n-1}$).

On peut même faire deux appels à la fonction :

Code 26 (*recursif.py (3)*).
```
def fibonacci(n):
    if (n==0) or (n==1):
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Faites-le calcul de `fibonacci(5)`. Voici la version mathématique des nombres de Fibonacci.

$$F_0 = 1,\ F_1 = 1 \quad \text{et} \quad F_n = F_{n-1} + F_{n-2} \quad \text{si } n \geqslant 2.$$

On obtient un nombre en additionnant les deux nombres des rangs précédents :

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \dots$$

5.2. L'algorithme d'Euclide

L'algorithme d'Euclide est basé sur le principe suivant

$$\text{si } b|a \text{ alors } \operatorname{pgcd}(a,b) = b \qquad \text{sinon } \operatorname{pgcd}(a,b) = \operatorname{pgcd}(b, a \bmod b)$$

Travaux pratiques 19.

1. Créer une fonction récursive `pgcd(a,b)` qui calcule le pgcd.
2. On note p_n la probabilité que deux entiers a, b tirés au hasard dans $1, 2, \dots, n$ soient premiers entre eux. Faire une fonction qui approxime p_n. Lorsque n devient grand, comparer p_n et $\frac{6}{\pi^2}$.

Voici le code pour l'algorithme d'Euclide récursif. Notez à quel point le code est succinct et épuré !

Code 27 (*arith.py (1)*).
```
def pgcd(a,b):
    if a%b == 0:
        return b
    else:
        return pgcd(b, a%b)
```

Deux entiers a, b sont premiers entre eux ssi $\text{pgcd}(a, b) = 1$, donc voici l'algorithme :

Code 28 (*arith.py (2)*).
```
def nb_premiers_entre_eux(n,nbtirages):
    i = 1
    nbpremiers = 0
    while i <= nbtirages:
        i = i+1
        a = random.randint(1,n)
        b = random.randint(1,n)
        if pgcd(a,b)==1:
            nbpremiers = nbpremiers + 1
    return nbpremiers
```

On tire au hasard deux entiers a et b entre 1 et n et on effectue cette opération `nbtirages` fois. Par exemple entre 1 et 1000 si l'on effectue 10 000 tirage on trouve une probabilité mesurée par `nbpremiers/nbtirages` de $0,60\ldots$ (les décimales d'après dépendent des tirages).
Lorsque n tend vers $+\infty$ alors $p_n \to \frac{6}{\pi^2} = 0.607927\ldots$ et on dit souvent que : «la probabilité que deux entiers tirés au hasard soient premiers entre eux est $\frac{6}{\pi^2}$.»

Commentaires sur les algorithmes récursifs :
- Les algorithmes récursifs ont souvent un code très court, et proche de la formulation mathématique lorsque l'on a une relation de récurrence.
- Selon le langage ou la fonction programmée il peut y avoir des problèmes de mémoire (si par exemple pour calculer 5! l'ordinateur a besoin de stocker 4! pour lequel il a besoin de stocker 3!...).
- Il est important de bien réfléchir à la condition initiale (qui est en fait celle qui termine l'algorithme) et à la récurrence sous peine d'avoir une fonction qui boucle indéfiniment !
- Il n'existe pas des algorithmes récursifs pour tout (voir par exemple les nombres premiers) mais ils apparaissent beaucoup dans les algorithmes de tris. Autre exemple : la dichotomie se programme très bien par une fonction récursive.

5.3. Nombres premiers

Les nombres premiers offrent peu de place aux algorithmes récursifs car il n'y a pas de lien de récurrence entre les nombres premiers.

Travaux pratiques 20.

1. Écrire une fonction qui détecte si un nombre n est premier ou pas en testant s'il existe des entiers k qui divise n. (On se limitera aux entiers $2 \leqslant k \leqslant \sqrt{n}$, pourquoi ?).
2. Faire un algorithme pour le crible d'Eratosthène : écrire tous les entiers de 2 à n, conserver 2 (qui est premier) et barrer tous les multiples suivants de 2. Le premier nombre non barré (c'est 3) est premier. Barrer tous les multiples suivants de 3,...

3. Dessiner la spirale d'Ulam : on place les nombres entiers en spirale, et on colorie en rouge les nombres premiers.

$$\begin{array}{cccc} & & \cdots & \cdots \\ 5 & 4 & 3 & \vdots \\ 6 & 1 & 2 & 11 \\ 7 & 8 & 9 & 10 \end{array}$$

1. Si n n'est pas premier alors $n = a \times b$ avec $a, b \geqslant 2$. Il est clair que soit $a \leqslant \sqrt{n}$ ou bien $b \leqslant \sqrt{n}$ (sinon $n = a \times b > n$). Donc il suffit de tester les diviseurs $2 \leqslant k \leqslant \sqrt{n}$. D'où l'algorithme :

Code 29 (*arith.py (3)*).
```
def est_premier(n):
    if (n<=1): return False
    k = 2
    while k*k <= n:
        if n%k==0:
            return False
        else:
            k = k +1
    return True
```

Notez qu'il vaut mieux écrire la condition `k*k <= n` plutôt que `k <= sqrt(n)` : il est beaucoup plus rapide de calculer le carré d'un entier plutôt qu'extraire une racine carrée.

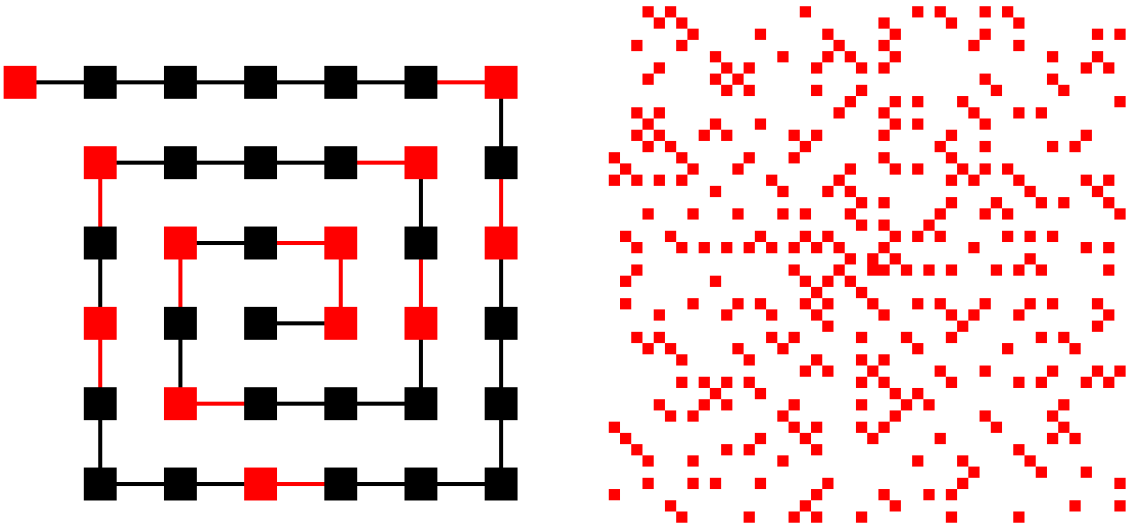
Nous avons utilisé un nouveau type de variable : un ***booléen*** est une variable qui ne peut prendre que deux états Vrai ou Faux (ici *True* or *False*, souvent codé 1 et 0). Ainsi `est_premier(13)` renvoie `True`, alors que `est_premier(14)` renvoie `False`.

2. Pour le crible d'Eratosthène le plus dur est de trouver le bon codage de l'information.

Code 30 (*arith.py (4)*).
```
def eratosthene(n):
    liste_entiers = list(range(n+1))  # tous les entiers
    liste_entiers[1] = 0              # 1 n'est pas premier
    k = 2                             # on commence par les multiples de 2
    while k*k <= n:
        if liste_entiers[k] != 0: # si le nombre k n'est pas barré
            i = k                 # les i sont les multiples de k
            while i <= n-k:
                i = i+k
                liste_entiers[i] = 0  # multiples de k : pas premiers
        k = k +1
    liste_premiers = [k for k in liste_entiers if k !=0]  # efface les 0
    return liste_premiers
```

Ici on commence par faire un tableau contenant les entiers `[0,1,2,3,4,5,6,7,8,9,10,11,12,13,...]`. Pour signifier qu'un nombre n'est pas premier ou remplace l'entier par 0. Comme 1 n'est pas un nombre premier : on le remplace par 0. Puis on fait une boucle, on part de 2 et on remplace tous les autres multiples de 2 par 0 : la liste est maintenant : `[0,0,2,3,0,5,0,7,0,9,0,11,0,13,...]`. Le premiers nombre après 2 est 3 c'est donc un nombre premier. (car s'il n'a aucun diviseur autre que 1 et lui-même car sinon il aurait été rayé). On garde 3 et remplace tous les autres multiples de 3 par 0. La liste est maintenant : `[0,0,2,3,0,5,0,7,0,0,0,11,0,13,...]`. On itère ainsi, à la fin on efface les zéros pour obtenir : `[2,3,5,7,11,13,...]`.

3. Pour la spirale d'Ulam la seule difficulté est de placer les entiers sur une spirale, voici le résultat.



À gauche le début de la spirale (de $n = 1$ à 37) en rouge les nombres premiers (en noir les nombres non premiers) ; à droite le motif obtenu jusqu'à de grandes valeurs (en blanc les nombres non premiers).

Mini-exercices. 1. Écrire une version itérative et une version récursive pour les fonctions suivantes : (a) la somme des carrés des entiers de 1 à n ; (b) 2^n (sans utiliser d'exposant) ; (c) la partie entière d'un réel $x \geqslant 0$; (d) le quotient de la division euclidienne de a par b (avec $a \in \mathbb{N}$, $b \in \mathbb{N}^*$) ; (e) le reste de cette division euclidienne (sans utiliser les commandes `%` ni `//`).

2. Écrire une version itérative de la suite de Fibonacci.
3. Écrire une version itérative de l'algorithme d'Euclide. Faire une version qui calcule les coefficients de Bézout.
4. Écrire une fonction itérative, puis récursive, qui pour un entier n renvoie la liste de ses diviseurs. Dessiner une spirale d'Ulam, dont l'intensité de la couleur dépend du nombre de diviseurs.
5. Une suite de Syracuse est définie ainsi : partant d'un entier s'il est pair on le divise par deux, s'il est impair on le multiplie par 3 et on ajoute 1. On itère ce processus. Quelle conjecture peut-on faire sur cette suite ?
6. Dessiner le triangle de Pascal $\begin{matrix} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ & \cdots & & & & \cdots & \\ 1 & & & & & & 1 \end{matrix}$ Ensuite effacer tous les coefficients pairs (ou mieux : remplacer les coefficients pairs par un carré blanc et les coefficients impairs par un carré rouge). Quelle figure reconnaissez-vous ?

6. Polynômes – Complexité d'un algorithme

Nous allons étudier la complexité des algorithmes à travers l'exemple des polynômes.

6.1. Qu'est-ce qu'un algorithme ?

Qu'est ce qu'un algorithme ? Un algorithme est une succession d'instructions qui renvoie un résultat. Pour être vraiment un algorithme on doit justifier que le résultat retourné est ***exact*** (le programme fait bien ce que l'on souhaite) et ceci en un ***nombre fini d'étapes*** (cela renvoie le résultat en temps fini).
Maintenant certains algorithmes peuvent être plus rapides que d'autres. C'est souvent le temps de calcul qui est le principal critère, mais cela dépend du langage et de la machine utilisée. Il existe une manière plus mathématique de faire : la ***complexité*** d'un algorithme c'est le nombre d'opérations élémentaires à effectuer.
Ces opérations peuvent être le nombre d'opérations au niveau du processeur, mais pour nous ce sera le nombre d'additions $+$, le nombre de multiplications $\times$ à effectuer. Pour certains algorithmes la vitesse d'exécution n'est pas le seul paramètre mais aussi la taille de la mémoire occupée.

6.2. Polynômes

Travaux pratiques 21.
On code un polynôme $a_0 + a_1X + \cdots + a_nX^n$ sous la forme d'une liste $[a_0, a_1, \ldots, a_n]$.

1. Écrire une fonction correspondant à la somme de deux polynômes. Calculer la complexité de cet algorithme (en terme du nombre d'additions sur les coefficients, en fonctions du degré des polynômes).
2. Écrire une fonction correspondant au produit de deux polynômes. Calculer la complexité de cet algorithme (en terme du nombre d'additions et de multiplications sur les coefficients).
3. Écrire une fonction correspondant au quotient et au reste de la division euclidienne de A par B où B est un polynôme unitaire (son coefficient de plus haut degré est 1). Majorer la complexité de cet algorithme (en terme du nombre d'additions et de multiplications sur les coefficients).

1. La seule difficulté est de gérer les indices, en particulier on ne peut appeler un élément d'une liste en dehors des indices où elle est définie. Une bonne idée consiste à commencer par définir une fonction `degre(poly)`, qui renvoie le degré du polynôme (attention au 0 non significatifs).
   Voici le code dans le cas simple où $\deg A = \deg B$:

   Code 31 (*polynome.py (1)*).
   ```
   def somme(A,B):            # si deg(A)=deg(B)
       C = []
       for i in range(0,degre(A)+1):
           s = A[i]+B[i]
           C.append(s)
   ```

   Calculons sa complexité, on suppose $\deg A \leqslant n$ et $\deg B \leqslant n$: il faut faire l'addition des coefficients $a_i + b_i$, pour i variant de 0 à n : donc la complexité est de $n+1$ additions (dans $\mathbb{Z}$ ou $\mathbb{R}$).
2. Pour le produit il faut se rappeler que si $A(X) = \sum_{i=0}^{m} a_iX^i$, $B(X) = \sum_{j=0}^{n} b_jX^j$ et $C = A \times B = \sum_{k=0}^{m+n} c_kX^k$ alors le k-ème coefficient de C est $c_k = \sum_{i+j=k} a_i \times b_j$. Dans la pratique on fait attention de ne pas accéder à des coefficients qui n'ont pas été définis.

   Code 32 (*polynome.py (2)*).
   ```
   def produit(A,B):
       C = []
       for k in range(degre(A)+degre(B)+1):
           s = 0
           for i in range(k+1):
               if (i <= degre(A)) and (k-i <= degre(B)):
                   s = s + A[i]*B[k-i]
           C.append(s)
       return C
   ```

   Pour la complexité on commence par compter le nombre de multiplications (dans $\mathbb{Z}$ ou $\mathbb{R}$). Notons $m = \deg A$ et $n = \deg B$. Alors il faut multiplier les $m+1$ coefficients de A par les $n+1$ coefficients de B : il y a donc $(m+1)(n+1)$ multiplications.
   Comptons maintenant les additions : les coefficients de $A \times B$ sont : $c_0 = a_0b_0$, $c_1 = a_0b_1 + a_1b_0$, $c_2 = a_2b_0 + a_1b_1 + a_2b_0$,...
   Nous utilisons l'astuce suivante : nous savons que le produit $A \times B$ est de degré $m+n$ donc a (au plus) $m+n+1$ coefficients. Partant de $(m+1)(n+1)$ produits, chaque addition regroupe deux termes, et nous devons arriver à $m+n+1$ coefficients. Il y a donc $(m+1)(n+1) - (m+n+1) = mn$ additions.
3. Pour la division euclidienne, le principe est de poser une division de polynôme. Par exemple pour $A = 2X^4 - X^3 - 2X^2 + 3X - 1$ et $B = X^2 - X + 1$.

$$
\begin{array}{rl|l}
 & 2X^4-X^3-2X^2+3X-1 & X^2-X+1 \\
- & 2X^4-2X^3+2X^2 & \hline \\
\cline{2-2}
 & X^3-4X^2+3X-1 & 2X^2+X-3 \\
- & X^3-X^2+X & \\
\cline{2-2}
 & -3X^2+2X-1 & \\
- & -3X^2+3X-3 & \\
\cline{2-2}
 & -X+2 &
\end{array}
$$

Alors on cherche quel monôme P_1 fait diminuer le degré de $A-P_1B$, c'est $2X^2$ (le coefficient 2 est le coefficient dominant de A). On pose ensuite $R_1 = A-P_1B = X^3-4X^2+3X-1$, $Q_1 = 2X^2$, on recommence avec R_1 divisé par B, $R_2 = R_1 - P_2B$ avec $P_2 = X$, $Q_2 = Q_1 + P_2$,... On arrête lorsque $\deg R_i < \deg B$.

Code 33 (*polynome.py (3)*).

```
def division(A,B):
    Q = [0]      # Quotient
    R = A        # Reste
    while (degre(R) >= degre(B)):
        P = monome(R[degre(R)],degre(R)-degre(B))
        R = somme(R,produit(-P,B))
        Q = somme(Q,P)
    return Q,R
```

C'est une version un peu simplifiée du code : où $P = r_n X^{\deg R - \deg B}$ et où il faut remplacer $-P$ par $[-a_0, -a_1, \ldots]$. Si $A, B \in \mathbb{Z}[X]$ alors le fait que B soit unitaire implique que Q et R sont aussi à coefficients entiers.

Quelle est la complexité de la division euclidienne ? À chaque étape on effectue une multiplication de polynômes $(P_i \times B)$ puis une addition de polynôme $(R_i - P_iB)$; à chaque étape le degré de R_i diminue (au moins) de 1. Donc il y a au plus $\deg A - \deg B + 1$ étapes.

Mais dans la pratique c'est plus simple que cela. La multiplication $P_i \times B$ est très simple : car P_i est un monôme $P_i = p_i X^i$. Multiplier par X^i c'est juste un décalage d'indice (comme multiplier par 10^i en écriture décimale) c'est donc une opération négligeable. Il reste donc à multiplier les coefficients de B par p_i : il y a donc $\deg B + 1$ multiplications de coefficients. La soustraction aussi est assez simple on retire à R_i un multiple de B, donc on a au plus $\deg B + 1$ coefficients à soustraire : il y a à chaque étape $\deg B + 1$ additions de coefficients.

Bilan : si $m = \deg A$ et $n = \deg B$ alors la division euclidienne s'effectue en au plus $(m-n+1)(m+1)$ multiplications et le même nombre d'additions (dans $\mathbb{Z}$ ou $\mathbb{R}$).

6.3. Algorithme de Karatsuba

Pour diminuer la complexité de la multiplication de polynômes, on va utiliser un paradigme très classique de programmation : « diviser pour régner ». Pour cela, on va décomposer les polynômes à multiplier P et Q de degrés strictement inférieurs à $2n$ en

$$P = P_1 + P_2 \cdot X^n \quad \text{et} \quad Q = Q_1 + Q_2 \cdot X^n$$

avec les degrés de P_1, P_2, Q_1 et Q_2 strictement inférieurs à n.

Travaux pratiques 22.

1. Écrire une formule qui réduit la multiplication des polynômes P et Q de degrés strictement inférieurs à $2n$ en multiplications de polynômes de degrés strictement inférieurs à n.

2. Programmer un algorithme récursif de multiplication qui utilise la formule précédente. Quelle est sa complexité ?
3. On peut raffiner cette méthode avec la remarque suivante de Karatsuba : le terme intermédiaire de $P \cdot Q$ s'écrit
$$P_1 \cdot Q_2 + P_2 \cdot Q_1 = (P_1 + P_2) \cdot (Q_1 + Q_2) - P_1Q_1 - P_2Q_2$$
Comme on a déjà calculé P_1Q_1 et P_2Q_2, on échange deux multiplications et une addition (à gauche) contre une multiplication et quatre additions (à droite). Écrire une fonction qui réalise la multiplication de polynômes à la Karatsuba.
4. Trouver la formule de récurrence qui définit la complexité de la multiplication de Karatsuba. Quelle est sa solution ?

1. Il suffit de développer le produit $(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2)$:
$$(P_1 + X^n P_2) \cdot (Q_1 + X^n Q_2) = P_1Q_1 + X^n \cdot (P_1Q_2 + P_2Q_1) + X^{2n} \cdot P_2Q_2$$
On se ramène ainsi aux quatre multiplications P_1Q_1 , P_1Q_2 , P_2Q_1 et P_2Q_2 entre polynômes de degrés strictement inférieurs à n, plus deux multiplications par X^n et X^{2n} qui ne sont que des ajouts de zéros en tête de liste.
2. On sépare les deux étapes de l'algorithme : d'abord la découpe des polynômes (dans laquelle il ne faut pas oublier de donner n en argument car ce n'est pas forcément le milieu du polynôme, n doit être le même pour P et Q). Le découpage `P1,P2 = decoupe(P,n)` correspond à l'écriture $P = P_1 + X^n P_2$.

Code 34 (*polynome.py (4)*).
```
def decoupe(P,n):
    if (degre(P)<n): return P, [0]
    else: return P[0:n], P[n:]
```

On a aussi besoin d'une fonction `produit_monome(P,n)` qui renvoie le polynôme $X^n \cdot P$ par un décalage. Voici la multiplication proprement dite avec les appels récursifs et leur combinaison.

Code 35 (*polynome.py (5)*).
```
def produit_assez_rapide(P,Q):
    p = degre(P) ; q = degre(Q)
    if (p == 0): return [P[0]*k for k in Q]  # Condition initiale: P=cst
    if (q == 0): return [Q[0]*k for k in P]  # Condition initiale: Q=cst
    n = (max(p,q)+1)//2                      # demi-degré
    P1,P2 = decoupe(P,n)                     # decoupages
    Q1,Q2 = decoupe(Q,n)
    P1Q1 = produit_assez_rapide(P1,Q1)       # produits en petits degrés
    P2Q2 = produit_assez_rapide(P2,Q2)
    P1Q2 = produit_assez_rapide(P1,Q2)
    P2Q1 = produit_assez_rapide(P2,Q1)
    R1 = produit_monome(somme(P1Q2,P2Q1),n)  # décalages
    R2 = produit_monome(P2Q2,2*n)
    return somme(P1Q1,somme(R1,R2))          # sommes
```

La relation de récurrence qui exprime la complexité de cet algorithme est $C(n) = 4C(n/2) + O(n)$ et elle se résout en $C(n) = O(n^2)$. Voir la question suivante pour une méthode de résolution.

3.

Code 36 (*polynome.py (6)*).
```
def produit_rapide(P,Q):
    p = degre(P) ; q = degre(Q)
    if (p == 0): return [P[0]*k for k in Q]  # Condition initiale: P=cst
    if (q == 0): return [Q[0]*k for k in P]  # Condition initiale: Q=cst
    n = (max(p,q)+1)//2                      # demi-degré
    P1,P2 = decoupe(P,n)                     # decoupages
```

```
    Q1,Q2 = decoupe(Q,n)
    P1Q1 = produit_rapide(P1,Q1)              # produits en petits degrés
    P2Q2 = produit_rapide(P2,Q2)
    PQ = produit_rapide(somme(P1,P2),somme(Q1,Q2))
    R1 = somme(PQ,somme([-k for k in P1Q1],[-k for k in P2Q2]))
    R1 = produit_monome(R1,n)                 # décalages
    R2 = produit_monome(P2Q2,2*n)
    return somme(P1Q1,somme(R1,R2))           # sommes
```

4. Notons $C(n)$ la complexité de la multiplication entre deux polynômes de degrés strictement inférieurs à n. En plus des trois appels récursifs, il y a des opérations linéaires : deux calculs de degrés, deux découpes en $n/2$ puis des additions : deux de taille $n/2$, une de taille n, une de taille $3n/2$ et une de taille $2n$. On obtient donc la relation de récurrence suivante :
$$C(n) = 3 \cdot C(n/2) + \gamma n$$
où $\gamma = \frac{15}{2}$. Une méthode de résolution est de poser $\alpha_\ell = \frac{C(2^\ell)}{3^\ell}$ qui vérifie $\alpha_\ell = \alpha_{\ell-1} + \gamma\left(\frac{2}{3}\right)^\ell$. D'où on tire, puisque $\alpha_0 = C(1) = 1$,
$$\alpha_\ell = \gamma \sum_{k=1}^{\ell} \left(\frac{2}{3}\right)^k + \alpha_0 = 3\gamma\left(1 - \left(\frac{2}{3}\right)^{\ell+1}\right) + 1 - \gamma$$
puis pour $n = 2^\ell$:
$$C(n) = C(2^\ell) = 3^\ell \alpha_\ell = \gamma(3^{\ell+1} - 2^{\ell+1}) + (1-\gamma)3^\ell = O(3^\ell) = O(2^{\ell \frac{\ln 3}{\ln 2}}) = O(n^{\frac{\ln 3}{\ln 2}})$$
La complexité de la multiplication de Karatsuba est donc $O(n^{\frac{\ln 3}{\ln 2}}) \simeq O(n^{1.585})$.

6.4. Optimiser ses algorithmes

Voici quelques petites astuces pour accélérer l'écriture ou la vitesse des algorithmes :
- `k ** 3` au lieu de `k * k * k` (cela économise de la mémoire, une seule variable au lieu de 3) ;
- `k ** 2 <= n` au lieu de `k <= sqrt(n)` (les calculs avec les entiers sont beaucoup plus rapides qu'avec les réels) ;
- `x += 1` au lieu de `x = x +1` (gain de mémoire) ;
- `a,b = a+b, a-b` au lieu de `newa = a+b ; newb = a-b ; a = newa ; b = newb` (gain de mémoire, code plus court).

Cependant il ne faut pas que cela nuise à la lisibilité du code : il est important que quelqu'un puisse relire et modifier votre code. Le plus souvent c'est vous même qui modifierez les algorithmes qui vous avez écrits et vous serez ravi d'y trouver des commentaires clairs et précis !

Mini-exercices. 1. Faire une fonction qui renvoie le pgcd de deux polynômes.

2. Comparer les complexités des deux méthodes suivantes pour évaluer un polynôme P en une valeur $x_0 \in \mathbb{R}$: $P(x_0) = a_0 + a_1 x_0 + \cdots + a_{n-1} x_0^{n-1} + a_n x_0^n$ et $P(x_0) = a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)\big)\Big)$ (méthode de Horner).
3. Comment trouver le maximum d'une liste ? Montrer que votre méthode est de complexité minimale (en terme du nombre de comparaisons).
4. Soit $f : [a,b] \to \mathbb{R}$ une fonction continue vérifiant $f(a) \cdot f(b) \leqslant 0$. Combien d'itérations de la méthode de dichotomie sont nécessaires pour obtenir une racine de $f(x) = 0$ avec une précision inférieure à ϵ ?
5. Programmer plusieurs façons de calculer les coefficients du binôme de Newton $\binom{n}{k}$ et les comparer.
6. Trouver une méthode de calcul de 2^n qui utilise peu de multiplications. On commencera par écrire n en base 2.

Auteurs du chapitre Rédaction : Arnaud Bodin
Relecture : Jean-François Barraud
Remerciements à Lionel Rieg pour son tp sur l'algorithme de Karatsuba