# 1
# Algorithmes fondamentaux

**1.** Un petit nombre d'algorithmes servent de *modèle* à la plupart des algorithmes et pour cette raison, il doivent être lus, compris, appris et sus aussi bien dans leur version impérative que dans leur version récursive.

**2.** On pourra mettre en pratique les différents algorithmes sur des listes aléatoires comme celles qui suivent.

```
import random as rd

# Liste de 20 entiers compris entre 0 et 9
L = [rd.randint(0,9) for i in range(20)]
# Liste de 20 entiers triée par ordre croissant
L = [0]
for i in range(20):
  L.append(L[-1]+rd.randint(0,5))
# Liste de 20 réels compris entre -1 et 1
L = [2*rd.random()-1 for i in range(20)]
# Liste de 20 réels triée par ordre croissant
L = [-1]
for i in range(20):
  L.append(L[-1]+0.2*rd.random())
```

**3. Échantillonnage d'une fonction**

Soit $f$, une fonction définie sur un ensemble $\Omega$. Étant donnée une liste $X$ de valeurs appartenant à $\Omega$, on peut calculer sous forme de liste un échantillon de valeurs de $f$.

```
Y = [f(x) for x in X]
```

**4. Premiers termes d'une suite récurrente**

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée de $u_0 \in \Omega$ et la relation de récurrence

$$\forall\, i \geqslant 1, \quad u_i = f(u_{i-1})$$

où $f$ est une application de $\Omega$ dans $\Omega$.
On peut rassembler sous forme de liste les $n$ premières valeurs de la suite.

```
U = [u]       # Valeur initiale u_0
for i in range(1,n):
  u = f(u)    # Itération u_i = f(u_{i-1})
  U.append(u)
```

# I

# Listes de nombres

**5.** On considère une liste de nombres $L = (L_k)_{0 \leqslant k < n}$, entiers ou réels.

## I.1 Calculs de sommes

**6.** L'instruction `sum(L)` retourne la somme des éléments de $L$.

**7.** On calcule la somme de la liste $L$ en calculant les sommes partielles successives.

– La première somme partielle est nulle.

$$S_{-1} = 0$$

– Les sommes partielles suivantes sont calculées en suivant la relation suivante :

$$\forall\ 0 \leqslant i < n, \quad \sum_{k=0}^{i} L_k = \sum_{k=0}^{i-1} + L_i.$$

La dernière somme partielle calculée est alors égale à la somme de la liste $L$.

$$S_{n-1} = \sum_{k=0}^{n-1} L_k.$$

```
def somme(L):
  s = 0
  for x in L:
    s += x
  return s
```

**8.** L'algorithme récursif distingue deux cas.

– Si la liste est vide, alors sa somme est nulle.

– Sinon, la somme de la liste $L$ est la somme du premier terme $L_0$ et de la somme de la sous-liste

$$(L_k)_{1 \leqslant k < n},$$

qui est éventuellement vide.

```
def somme(L):
  if (L==[]):
    return 0
  else:
    return L[0]+somme(L[1:])
```

La profondeur de récursion étant limitée, cette fonction ne peut pas calculer la somme d'une liste qui contient plus de 1 000 nombres.

**Q 1. Variantes**

*Comment calculer les quantités suivantes ?*

**Q 1.a** *La **moyenne** de L est définie par*

$$\langle L \rangle = \frac{1}{n} \sum_{k=0}^{n-1} L_k.$$

**Q 1.b** *La **variance** de $L$ est définie par*

$$\mathbf{V}(L) = \frac{1}{n}\sum_{k=0}^{n-1} L_k^2 - \langle L\rangle^2.$$

**Q 1.c** *Étant donnée une liste de réels positifs $C = (C_k)_{0\leqslant k<n}$, la **moyenne pondérée** de $L$ par $C$ est définie par*

$$\langle L\rangle_C = \frac{\sum_{k=0}^{n-1} C_k L_k}{\sum_{k=0}^{n-1} C_k}.$$

## I.2 Calcul du maximum

**9.** On suppose que la liste $L$ n'est pas vide (pour qu'elle ait un maximum).
Les instructions `max(L)` et `min(L)` retournent respectivement le plus grand et le plus petit élément de la liste $L$.

**10.** On calcule le maximum de la liste $L$ de proche en proche en calculant le maximum de sous-listes de $L$ de plus en plus grandes.
– Le premier maximum est égal au premier terme de la liste.

$$M_0 = L_0$$

– Les maxima suivants sont calculés en suivant la relation suivante :

$$\forall\, 1 \leqslant i < n, \quad M_i = \max\{L_i, M_{i-1}\}$$

de telle sorte que

$$\forall\, 0 \leqslant i < n, \quad M_i = \max\{L_k,\, 0 \leqslant k < i\}.$$

```
def maximum(L):
  M = L[0]
  for x in L[1:]:
    if (x>M):
      M = x
  return M
```

**11.** L'algorithme récursif suppose connue une fonction `maxi(x, y)` qui retourne le plus de ses deux arguments.

```
def maxi(x, y):
  if (x<y):
    return y
  else:
    return x
```

On distingue alors deux cas :
– Si la liste $L$ ne contient qu'un seul élément, cet élément est le plus grand de la liste.

$$\max(L_0) = L_0$$

– Sinon, le plus grand élément de $L$ est le plus grand nombre entre $L_0$ et le plus grand élément du reste de la liste.

$$\max L = \max\left\{L_0, \max(L_k)_{1\leqslant k<n}\right\}$$

```
def max_liste(L):
  if (len(L)==1):
    return L[0]
  else:
    return maxi(L[0], max_liste(L[1:]))
```

La profondeur de récursion étant limitée, cette fonction ne peut pas calculer le maximum d'une liste qui contient plus de 1 000 nombres.

#### Positions du maximum

**12.** En calculant le maximum d'une liste, on peut aussi noter la (ou les) position(s) où le maximum est atteint.
Pour chaque élément de la liste, on vérifie
– s'il s'agit d'un nouveau maximum, plus grand que le maximum connu ;
– s'il s'agit d'une valeur égale au maximum connu
– ou s'il s'agit d'une valeur inférieure au maximum connu.

```
def max_pos(L):
  M, positions = L[0], [0]
  for i in range(1, len(L)):
    if (L[i]>M):
      M, positions = L[i], [i]
    elif (L[i]==M):
      positions.append(i)
  return M, positions
```

**Q 2. Variantes**
*Simplifier l'algorithme précédent pour ne conserver que le plus petit indice (resp. le plus grand indice) où est atteint le maximum de la liste $L$.*

### I.3 Présence d'un élément

**13.** L'objet $x$ appartient à la liste $L$ si, et seulement si, le booléen `(x in L)` prend la valeur `True`.

**14.** On parcourt la liste $L$.
Le booléen `present`, initialement à `False`, reste égal à `False` tant qu'on n'a pas trouvé $x$, passe à `True` lorsqu'on rencontre $x$ et reste ensuite à `True` jusqu'au terme du parcours de la liste $L$.

```
def presence(x, L):
  present = False
  for y in L:
    present = present or (x==y)
  return present
```

**15.** Faut-il parcourir en entier la liste $L$ ?
– C'est bien sûr nécessaire si la liste ne contient pas $x$.
– Mais si elle contient $x$, on peut arrêter les vérifications dès qu'on a rencontré $x$ !
On va donc avancer dans le parcours de la liste $L$ tant qu'on n'a pas rencontré $x$ *et* qu'on n'a pas atteint le terme de la liste.

```
def presence(x, L):
  absent = True
  i = 0
  while (absent and (i<len(L))):
    absent = (L[i]!=x)
    i += 1
  return not(absent)
```

**16.** En style récursif, on distingue trois cas :
– Si la liste est vide, elle ne peut pas contenir $x$ !
– Si la liste n'est pas vide, alors
  – ou bien $x$ est en tête de la liste,
  – ou bien on cherche si $x$ appartient au reste de la liste.

```
def presence(x, L):
  if (L==[]):
    return False
  elif (x==L[0]):
    return True
  else:
    return presence(x, L[1:])
```

La profondeur de récursion étant limitée, cette fonction ne peut pas repérer un élément qui ne figure pas parmi les 1 000 premiers éléments de la liste.

### Position d'un élément

**17.** Si $x$ est un élément de la liste $L$, alors l'instruction `L.index(x)` retourne le plus petit indice $0 \leqslant i < n$ tel que $L_i = x$.
Mais si $x$ n'est pas un élément de la liste $L$, alors la méthode `index` retourne une erreur de type `ValueError`.

**18.** On reprend l'algorithme de recherche **[14]** pour retourner la liste de *tous* les indices $0 \leqslant i < n$ tels que $L_i = x$.
Il faut donc parcourir la liste $L$ en entier en connaissant l'indice de chaque élément rencontré et, à chaque fois qu'on rencontre $x$, enregistrer la valeur de cet indice.

```
def positions(x, L):
  liste_positions = []
  for i in range(len(L)):
    if (L[i]==x):
      liste_positions.append(i)
  return liste_positions
```

**Q 3.** *Modifier l'algorithme* **[15]** *pour que* `position(x, L)` *imite* `L.index(x)` *en ne retournant que la plus petite valeur de $i$ telle que $L_i = x$. Si $x$ n'appartient pas à $L$, la fonction retournera* `None`.

**19.** On cherche encore à imiter la fonction `index` mais en style récursif.
L'indice de chaque élément doit maintenant faire partie des arguments de la fonction.
On distingue trois cas :
– Si la liste est vide, elle ne peut pas contenir $x$ et on retourne `None`.
– Sinon :
  – ou bien $x$ est en tête de liste et on retourne la valeur de $i$,
  – ou bien on cherche la position de $x$ dans le reste de la liste en incrémentant la valeur de $i$.

```
def position(x, L, i=0):
  if (len(L)==0):
    return None
  elif (x==L[0]):
    return i
  else:
    return position(x, L[1:], i+1)
```

**Q 4.** *Proposer une version récursive de l'algorithme* **[18]**. *L'un des arguments sera une liste à laquelle on adjoindra les indices des différentes positions de l'élément $x$ dans $L$.*

# II

# Arithmétique

## II.1 Algorithme d'Euclide

**20.** L'algorithme d'Euclide permet de calculer le pgcd de deux entiers $a$ et $b$ par divisions euclidiennes successives en remarquant que si la division euclidienne de $a$ par $b$ s'écrit

$$a = qb + r$$

alors le pgcd de $a$ et $b$ est aussi le pgcd de $b$ et $r$.
Comme $0 \leqslant r < |b|$ par définition de la division euclidienne, la suite des restes calculés finit par prendre la valeur 0, ce qui signe l'arrêt de la procédure.
**20.1** On commence par se débarrasser des signes éventuels pour obtenir un pgcd dans $\mathbb{N}$.
– Si $b$ est nul, on ne peut pas effectuer la division euclidienne de $a$ par $b$, mais dans ce cas le pgcd de $a$ et $b$ est égal à $a$.
– Sinon, on effectue des divisions euclidiennes successives tant que le reste n'est pas nul.

```
def pgcd(a, b):
  a, b = abs(a), abs(b)
  if (b==0):
    return a
  else:
    while (b!=0):
      a, b = b, a%b
  return a
```

**20.2** L'écriture récursive est plus concise car cet algorithme est naturellement récursif.

```
def pgcd(a, b):
  if (b==0):
    return a
  else:
    return pgcd(b, a%b)
```

**21. Cas d'une liste d'entiers**

Pour calculer le pgcd d'une liste $L$ d'entiers, on procède de proche en proche.

**21.1** En style impératif, on calcule le pgcd $d_1$ des deux premiers éléments de la liste et on continue ensuite en calculant

$$\forall\, 2 \leqslant i < n, \quad d_i = \text{pgcd}\,(a_i, d_{i-1}).$$

On vérifie par récurrence que :

$$\forall\, 1 \leqslant i < n, \quad d_i = \text{pgcd}\,(a_0, a_1, \ldots, a_i)$$

et la dernière valeur calculée, $d_{n-1}$, est donc le pgcd de la liste $L$.

```
def pgcd_liste(L):
    d = pgcd(L[0], L[1])
    for i in range(2, len(L)):
        d = pgcd(L[i], d)
    return d
```

On peut limiter le nombre de calculs effectués en interrompant la boucle `for` dès que la variable `d` prend la valeur 1.

```
def pgcd_liste(L):
    d = pgcd(L[0], L[1])
    for i in range(2, len(L)):
        d = pgcd(L[i], d)
        if (d==1):
          return 1
    return d
```

**21.2** L'écriture récursive repose sur la même relation de récurrence mais n'effectue pas les calculs dans le même ordre !

```
def pgcd_liste(L):
    if (len(L)==2):
        return pgcd(L[0], L[1])
    else:
        return pgcd(L[0], pgcd_liste(L[1:]))
```

La profondeur de récursion étant limitée, cette fonction ne peut pas calculer le pgcd d'une liste qui contient plus de 1 000 nombres.

**Q 5.**
*On applique la fonction* `pgcd_liste` *à une liste $L$ de longueur $n = 6$. Quels sont, dans l'ordre chronologique, les calculs effectués ?*

## II.2 Représentation binaire

### 22. Décomposition d'un entier en base 2

Chaque entier $n$ peut être décomposé de manière unique sous la forme

$$n = \sum_{k=0}^{d} \varepsilon_k 2^k$$

où $\varepsilon_k \in \{0, 1\}$ pour tout $k \in \mathbb{N}$ et $\varepsilon_d = 1$.
Les fonctions suivantes associent la liste

$$(\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_d)$$

où le *bit de poids fort* $\varepsilon_d$ (ou **Most Significant Bit**) apparaît en dernier, à l'entier $n$.

**22.1** En reprenant les notations précédentes, on remarque que la division euclidienne de $n$ par 2 s'écrit

$$n = 2 \times \left( \sum_{k=1}^{d} \varepsilon_k 2^k \right) + \varepsilon_0.$$

On va donc enregistrer le reste de cette division et continuer le calcul en remplaçant $n$ par le quotient de cette division. Chaque nouvelle valeur de $\varepsilon_k$ calculée est ajoutée en fin de liste.

```
def binaire(n):
  L = []
  while (n>0):
    L.append(n%2)
    n = n//2
  return L
```

**22.2** En style récursif, on distingue deux cas.
- Si $n = 0$ ou $n = 1$, on peut identifier $n$ à sa représentation en base 2.
- Sinon on constitue une liste en commençant par le reste $\varepsilon_0$ et en continuant avec la représentation binaire du quotient.

```
def binaire(n):
  if (n<2):
    return [n]
  else:
    return [n%2]+binaire(n//2)
```

**Q 6.** *Comment associer à l'entier n la liste*

$$(\varepsilon_d, \varepsilon_{d-1}, \ldots, \varepsilon_1, \varepsilon_0)$$

*où, cette fois, le bit de poids fort $\varepsilon_d$ apparaît en premier ?*

## 23. Exponentiation rapide

Rien n'a l'air plus simple que le calcul des puissances d'un nombre.

```
def puissance(x, n):
  p = 1
  for i in range(n):
    p *= x
  return x
```

**23.1** Pourtant, on peut concevoir un algorithme qui calcule $x^n$ en effectuant bien moins de $(n-1)$ multiplications, ce qui est particulièrement intéressant lorsque l'entier $n$ est très grand ou lorsque chaque multiplication est une opération coûteuse, comme c'est le cas pour la multiplication de matrices de grande taille.

**23.2** La décomposition binaire

$$n = \sum_{k=0}^{d} \varepsilon_k 2^k$$

donne

$$x^n = x^{\varepsilon_0 + 2\varepsilon_1 + \cdots + 2^d \varepsilon_d} = \prod_{k=0}^{d} (x^{(2^k)})^{\varepsilon_k} = \prod_{\substack{0 \leqslant k \leqslant d \\ \varepsilon_k = 1}} x^{(2^k)}$$

où le nombre de produits est inférieur à $d = \lg n$.
Pour calculer $x^n$, il suffit donc de connaître les facteurs

$$f_0 = x, \quad f_1 = x^2, \quad f_2 = x^4, \quad f_3 = x^8, \quad f_4 = x^{16} \ldots$$

et ces valeurs se calculent facilement par récurrence puisque

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

Il suffit donc de $d$ multiplications pour obtenir les $f_k$ pour $k \leqslant d$ et le nombre total de multiplications pour obtenir $x^n$ est donc inférieur à $2 \lg n$.

**23.3** En version impérative, on reprend l'algorithme **[22.1]** de décomposition en base 2.
À chaque étape du calcul, on multiplie le produit partiel par $f_k$ (seulement pour $\varepsilon_k = 1$) avant de calculer $f_{k+1}$.

```
def puissance(x, n):
  p, f = 1, x     # p_{k-1} = 1, f_0 = x
  while (n>0):
    if (n%2):     # Ici, ε_k = 1
      p *= f      # donc p_k = p_{k-1} f_k.
    f *= f        # f_{k+1} = (f_k)^2
    n = n//2
  return p
```

**23.4** En version récursive, on distingue trois cas.
– Si l'exposant $n$ est égal à 1, alors $x^n = x$.
– Sinon, on discute selon la parité de $n$ :
  – si l'exposant est pair : $n = 2q$, alors $x^n = (x^q)^2$ ;
  – si l'exposant est impair : $n = 2q+1$, alors $x^n = x \times (x^q)^2$.

```
def puissance(x, n):
  if (n==1):
    return x
  else:
    y = puissance(x, n//2)
    if (n%2==1):
      return x*y*y
    else:
      return y*y
```

Il faut veiller à n'effectuer qu'*un seul* appel récursif par itération pour que l'algorithme conserve toute son efficacité, raison pour laquelle on a introduit la variable `y`.

# III

# Valeurs d'un polynôme

**24.** On peut calculer une expression polynomiale comme

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$$

par récurrence à partir de la liste $L = (a_0, a_1, \ldots, a_d)$ des coefficients du polynôme avec

$$s_{-1} = 0 \quad \text{et} \quad \forall\, 0 \leqslant i \leqslant d, \quad s_i = s_{i-1} + a_i x^i$$

en remarquant que les puissances de $x$ sont les termes d'une suite géométrique :

$$p_0 = 1 \quad \text{et} \quad \forall\, 0 \leqslant i < d, \quad p_{i+1} = x p_i.$$

```
def evaluation(L, x):
  s, p = 0, 1   # s_{-1}, p_0
  for a in L:
    s += a*p    # s_i = s_{i-1} + a_i p_i
    p *= x      # p_{i+1} = x p_i
  return s
```

L'évaluation d'un polynôme de degré $d$ demande alors $d$ additions et $2d$ multiplications.

**25. Schéma de Horner**
Le schéma de Horner est légèrement plus efficace : il permet d'évaluer un polynôme de degré $d$ en n'effectuant que $d$ additions et $d$ multiplications.
Si le polynôme $P = a_0 + a_1 X + \cdots + a_d X^d$ n'est pas nul, il est comme plus haut représenté par la liste de ses coefficients : $L = (a_0, a_1, \ldots, a_d)$. Le polynôme nul est représenté par la liste vide.

**25.1** Pour le calcul récursif, on distingue deux cas.
– Si $P = 0$, alors la valeur calculée $P(x)$ est égale à 0.
– Sinon on remarque que

$$P(x) = \sum_{k=0}^{d} a_k x^k = a_0 + x \times \left( \sum_{k=0}^{d-1} a_{k+1} x^k \right).$$

```
def Horner(L, x):
  if (L==[]):
    return 0
  else:
    return L[0]+x*Horner(L[1:], x)
```

**25.2** Le style impératif est moins simple à concevoir, il faut le déduire de la version récursive. Avec la condition initiale $p_0 = 0$ et la relation de récurrence

$$\forall\, 0 \leqslant i \leqslant d, \quad p_i = a_i + x p_{i-1},$$

on obtient finalement $P(x) = p_d$.

```
def Horner(L, x):
  p = 0
  for a in reversed(L):
    p = x*p+a
  return p
```

Si le langage Python nous permet facilement de parcourir à l'envers la liste $L$, rien ne nous autorise à transformer cette liste. C'est pourquoi on utilise `reversed(L)` et non pas `L.reverse()`.

## 26. Cas des polynômes creux

Lorsque le degré $d$ du polynôme est élevé et que la plupart de ses coefficients de degré inférieur à $d$ sont nuls, la méthode **[24]** aussi bien que la méthode de Horner manquent d'efficacité : on calcule beaucoup de puissances de $x$ sans en avoir l'usage.
On peut toujours représenter le polynôme

$$\sum_{0 \leqslant k < n} a_k x^{d_k}$$

par une liste de couples :

$$L = \big((a_0, d_0), (a_1, d_1), \ldots, (a_{n-1}, d_{n-1})\big)$$

en supposant que les scalaires $a_k$ sont tous différents de 0 et que les degrés $d_k$ sont deux à deux distincts.
Si le nombre $n$ de termes non nuls est petit et que le degré maximal $d_{n-1}$ est grand, il est plus efficace de calculer les $x^{d_k}$ avec l'algorithme d'exponentiation rapide qu'avec la formule de récurrence habituelle.

```
def evaluation(L, x):
  s = 0
  for (a,d) in L:
    s += a*puissance(x, d) # Exponentiation rapide [23]
  return s
```

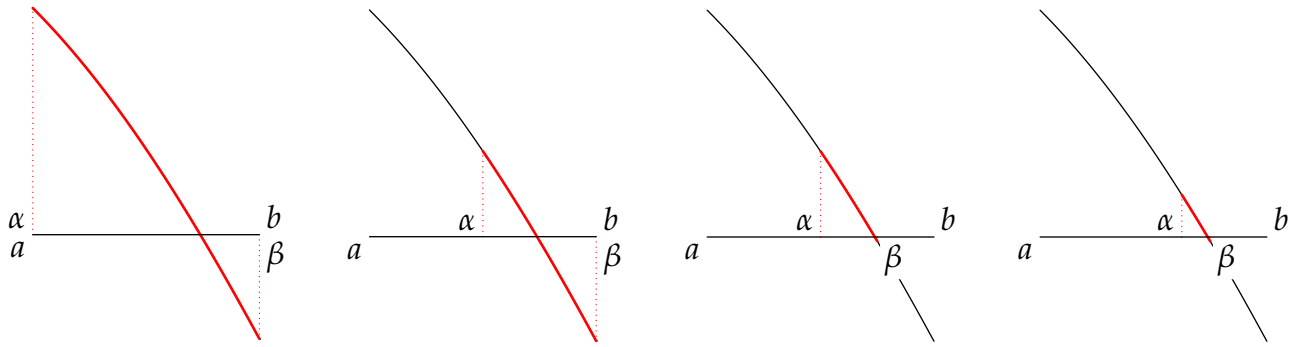
# IV

# Algorithmes dichotomiques

## IV.1 Zéro d'une fonction monotone

**27.** On considère une fonction continue et strictement monotone sur un intervalle $I$. On suppose connu un segment $[a,b] \subset I$ tel que $f(a)$ et $f(b)$ soient de signes opposés : l'équation $f(x) = 0$ possède alors une, et une seule, solution $x_0 \in [a,b]$.
Étant donnée une tolérance $\varepsilon > 0$ (raisonnablement petite), on cherche un encadrement $\alpha < x_0 \leqslant \beta$ d'incertitude $(\beta - \alpha) \leqslant \varepsilon$.

**28.** La recherche dichotomique consiste à considérer le segment $[a,b]$ comme un premier encadrement de $x_0$ puis à remplacer chaque encadrement de $x_0$ par un encadrement d'incertitude moitié.
De la sorte, au bout de $n$ itérations, l'incertitude n'est plus que $2^{-n}(b-a)$. Le nombre d'itérations nécessaires pour que l'incertitude devienne inférieure à la tolérance est donc $\mathcal{O}(\lg \varepsilon)$.




**29.** On suppose connu un encadrement $\alpha_n < x_0 \leqslant \beta_n$, de telle sorte que $f(\alpha_n) \neq 0$. En notant $\gamma_n$, le milieu du segment $[\alpha_n, \beta_n]$, on distingue deux cas.
– Si $f(\alpha_n)f(\gamma_n) > 0$, alors $f(\gamma_n)$ est du signe de $f(\alpha_n)$, donc la solution $x_0$ n'appartient pas au segment $[\alpha_n, \gamma_n]$. Elle appartient donc à l'intervalle $]\alpha_{n+1}, \beta_{n+1}] = ]\gamma_n, \beta_n]$.
– Dans le cas contraire, $f(\gamma_n)$ peut être nul (et donc $\gamma_n = x_0$) ou du signe opposé au signe de $f(\alpha_n)$. Dans les deux cas, la solution $x_0$ appartient à l'intervalle $]\alpha_n, \gamma_n]$.

```
def resoudre(f, a, b, epsilon=0.01):
  incertitude = b-a
  c = (a+b)/2
  fa, fc = f(a), f(c)
  while (incertitude>epsilon):
    if (fa*fc>0):
      a, fa = c, fc     # b non modifié
    else:
      b = c             # a et fa non modifiés
    c = (a+b)/2
    fc = f(c)
    incertitude /= 2
  return a, b
```

**30.** La version récursive repose sur la même discussion. Si elle est plus simple à formuler, elle doit calculer deux valeurs de la fonction $f$ à chaque itération (au lieu d'une seule valeur).

```
def resoudre(f, a, b, epsilon=0.01):
  if (b-a < epsilon):
    return (a,b)
  else:
    c = (b+a)/2
    if (f(a)*f(c) > 0):
      return resoudre(f, c, b, epsilon)
    else:
      return resoudre(f, a, c, epsilon)
```

La profondeur de récursion étant limitée, la tolérance choisie $\varepsilon$ doit être supérieure à $2^{-1\,000}(b-a)$. Mais comme $2^{-1\,000} \approx 10^{-301}$, ce n'est pas une vraie limitation !

## IV.2 Listes triées

**31.** Soit une liste réelle triée $L = (L_k)_{0 \leqslant k < n}$, par exemple par ordre croissant.

$$L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$$

**31.1** Pour un réel quelconque $x$, on distingue trois cas :
- ou bien $x < L_0$ ;
- ou bien $L_{n-1} \leqslant x$ ;
- ou bien il existe un indice $0 \leqslant r < n-2$ tel que $L_r \leqslant x < L_{r+1}$. Dans ce dernier cas, l'indice $r$ est unique : quel est-il ?

**31.2** On suppose donc que $L_0 \leqslant x < L_{n-1}$ et on construit de proche en proche une famille croissante d'indices $(i_k)$ et une famille décroissante d'indices $(j_k)$ telles que

$$\forall\, k, \qquad L_{i_k} \leqslant x < L_{j_k}.$$

À chaque étape, on compare $x$ à la valeur $L_{m_k}$ où l'indice $m_k$ est à peu près le milieu de $[i_k, j_k]$. Comme on impose ainsi que l'écart $(j_k - i_k)$ soit *strictement* décroissant et strictement positif, cet entier finit par prendre la valeur 1 au bout d'un certain nombre d'itérations et dans ce cas, l'indice $i_k$ est égal à l'indice $r$ cherché.

```
def recherche_dicho(x, L):
  i, j = 0, len(L)-1
  if (x<L[0]):
    return (None, 0)
  elif (x>=L[-1]):
    return (len(L)-1, None)
  else:
    while (j-i>1):
      m = (i+j)//2
      if (x<L[m]):
        j = m     # i reste inchangé
      else:
        i = m     # j reste inchangé
    return (i,j)
```

**31.3** Il faut faire attention à utiliser la division euclidienne par 2 (notée `//2`) et non pas la division usuelle par 2 (notée `/2`) pour que l'indice $m$ soit toujours un entier.

**32.** En style récursif, on conçoit deux fonctions : la première compare la donnée $x$ aux valeurs extrêmes de la liste $L$ ; la seconde met en œuvre la recherche récursive.

```
def recherche_dicho(x, L):
  n = len(L)
  if (x<L[0]):
    return (None,0)
  elif (x>L[-1]):
    return (n-1,None)
  else:
    return rech_rec(x, L, 0, n-1)

def rech_rec(x, L, i, j):
    if (j-i==1):
        return (i,j)
    else:
        m = (j+i)//2
        if (x<L[m]):
            return rech_rec(x, L, i, m)
        else:
            return rech_rec(x, L, m, j)
```

**Q 7.** **Insertion dans une liste triée**
*On considère un réel x et une liste L triée par ordre croissant.*

**Q 7.a** *Écrire une fonction d'arguments x et L qui retourne une liste triée par ordre croissant, obtenue en insérant x dans la liste L.*

**Q 7.b** *Écrire une fonction d'arguments x et L qui modifie la liste L en y insérant l'élément x tout en conservant le fait que cette liste soit triée.*
*On utilisera la méthode* `insert`.

# V

# Recherche d'un motif

**33.** On s'intéresse ici à l'une des principales fonctions des traitements de texte, des visionneuses de documents et des logiciels de séquençage du génôme : comment retrouver un **motif** donné $m$ (une chaîne de caractères particulière) dans une **source** $s$ (une chaîne de caractères) ?

**34.** L'algorithme récursif est ingénieux et remarquablement simple.
Cependant, la profondeur de récursion restreinte rend cet algorithme inutile en pratique, puisqu'il ne s'applique qu'à une source $s$ de moins de 1 000 caractères !

**34.1** La première étape consiste à vérifier si le motif $m$ ne constitue pas le début de la source $s$. Si tel est le cas, on dit que le motif est un **préfixe** de la source.
On vérifie si le premier élément du motif et le premier élément de la source concordent. Si tel est le cas, on vérifie si le reste du motif est un préfixe du reste de la source.
La descente récursive s'arrête lorsqu'on est arrivé au bout du motif ou au bout de la source.

```
def est_prefixe(m, s):
  if (len(m)==0):
    return True    # Le motif vide est contenu dans toutes les sources
  elif (len(s)==0):
    return False   # La source vide ne peut contenir que le motif vide
  else:
    return (m[0]==s[0]) and est_prefixe(m[1:], s[1:])
```

**34.2** Pour la recherche générale du motif, il faut comprendre que le motif est une sous-chaîne de caractères de la source si, et seulement si, c'est un préfixe d'une sous-chaîne de la source.
On distingue donc trois cas.
– Ou bien le motif $m$ est un préfixe de la source $s$ et donc une sous-chaîne de $s$.
– Ou bien le motif $m$ n'est pas un préfixe de la source $s$ :
  – si la source $s$ est vide, on ne peut pas y trouver le motif $m$ ;
  – sinon, on continue à chercher le motif $m$ dans le reste de la source.

```
def rechercher(m, s):
  if est_prefixe(m, s):
    return True
  elif (len(s)==0):
    return False
  else:
    return rechercher(m, s[1:])
```

**Q 8.** *Écrire la fonction* `est_prefixe(m, s)` *en style impératif.*

**35.** On considère qu'on cherche un court motif $m$ dans un long texte $s$ et donc que la longueur de la chaîne $m$ est (très) inférieure à la longueur de la chaîne $s$.
**35.1** En style impératif, l'algorithme est plus délicat à mettre au point : on s'imagine comme une tête de lecture, capable de lire les caractères de $s$ en avançant et en reculant.
**35.2** Sur un caractère de la source $s$, on a posé une *marque*. À partir de cette marque, on compare les caractères du motif à ceux de la source : le nombre de caractères communs à partir de la marque est le *nombre de coïncidences*.
**35.3** Tant que la tête de lecture n'a pas atteint la fin de la source $s$ et que le nombre de coïncidences n'atteint pas la longueur du motif, on continue les comparaisons.
– Si les caractères suivants sont les mêmes, on conserve la marque, on note la coïncidence supplémentaire et on se déplace pour comparer les deux caractères suivants. Peut-être allons-nous découvrir le motif $m$ en entier ?
– Dans le cas contraire, c'est un échec : le motif $m$ n'est pas sous nos yeux. Dans ces conditions, on déplace la marque et on recommence les comparaisons du début en mettant à 0 le nombre de coïncidences.
**35.4** Lorsqu'on sort de la boucle `while`, on a trouvé le motif si, et seulement si, le nombre de coïncidences est égal à la longueur du motif. (Si tel n'est pas le cas, on est sorti de la boucle en atteignant la fin du motif.)

```
def rechercher(m, s):
  lnm, lns = len(m), len(s)
  marque = 0
  nb_coincidences = 0
  position = marque + nb_coincidences
  while ((position<lns) and (nb_coincidences<lnm)):
    if (s[position] == m[nb_coincidences]):
      nb_coincidences += 1
      position += 1
    else:
      nb_coincidences = 0
      marque += 1
      position = marque
  return (nb_coincidences == lnm), marque
```

# Réponses aux questions

## I Listes de nombres

**R 1. Variantes**

Principe général : utiliser ce qui existe déjà, ne pas toujours recommencer la même chose !

**R 1.a** Calcul de la moyenne

```
def moyenne(L):
    return somme(L)/len(L)
```

**R 1.b** Calcul de la variance

```
def variance(L):
  return moyenne([x**2 for x in L])-(moyenne(L))**2
```

**R 1.c** Calcul d'une moyenne pondérée

```
def moyenne_ponderee(L, C):
  LC = [L[i]*C[i] for i in range(len(L))]
  return somme(LC)/somme(C)
```

Cet algorithme suppose d'une part que les deux listes $L$ et $C$ sont de même taille et d'autre part que la somme des coefficients de $C$ est non nulle. Dans le cadre *normal* d'un calcul de moyenne pondérée, ces deux conditions sont toujours satisfaites !

**R 2. Variantes**

Pour conserver le plus petit indice, il faut enregistrer la première fois où le maximum est atteint.

```
def max_pos(L):
  M, position = L[0], 0
  for i in range(len(L)):
    if (L[i]>M):
      M, position = L[i], i
  return M, position
```

Pour conserver le plus grand indice, il faut noter l'indice à chaque fois que le maximum est atteint : la dernière fois sera la bonne.

```
def max_pos(L):
  M, position = L[0], 0
  for i in range(len(L)):
    if (L[i]>M):        # Cas d'un nouveau maximum
      M, position = L[i], i
    elif (L[i]==M):     # Nouvelle occurrence du maximum connu
      position = i
  return M, position
```

**R 3.** Le principe est le suivant :
- Tant qu'on n'a pas trouvé $x$, on continue de parcourir la liste $L$.
- En fin de parcours,
  - ou bien on a trouvé $x$ et on connaît la valeur de l'indice $i$ à ce moment (cette variable n'a pas été incrémentée depuis la découverte de $x$) ;
  - ou bien on n'a pas trouvé $x$ et le booléen `absent` nous le dit.

```
def position(x, L):
  absent = True
  i = 0
  while (absent and (i<len(L))):
    absent = (L[i]!=x)
    if absent:
      i += 1
  if not(absent):
    return i
  else:
    return None
```

**R 4.**

```
def positions(x, L, p, i=0):
  if (L!=[]):
    if (L[0]==x):
      p.append(i)
    positions(x, L[1:], p, i+1)
```

## II Arithmétique

**R 5.**
Voici la liste des appels récursifs.

```
>>> pgcd(L[0],pgcd_liste([L[1],L[2],L[3],L[4],L[5]]))
>>> pgcd(L[0],pgcd(L[1],pgcd_liste([L[2],L[3],L[4],L[5]])))
>>> pgcd(L[0],pgcd(L[1],pgcd(L[2],pgcd_liste([L[3],L[4],L[5]]))))
>>> pgcd(L[0],pgcd(L[1],pgcd(L[2],pgcd(L[3],pgcd_liste([L[4],L[5]])))))
>>> pgcd(L[0],pgcd(L[1],pgcd(L[2],pgcd(L[3],pgcd(L[4],L[5])))))
```

Les valeurs calculées sont donc, dans l'ordre chronologique, les suivantes.

$$\begin{aligned}
&\text{pgcd}\,(L_4, L_5)\\
&\text{pgcd}\,(L_3, L_4, L_5)\\
&\text{pgcd}\,(L_2, L_3, L_4, L_5)\\
&\text{pgcd}\,(L_1, L_2, L_3, L_4, L_5)\\
&\text{pgcd}\,(L_0, L_1, L_2, L_3, L_4, L_5)
\end{aligned}$$

**R 6.**
C'est très simple en style récursif : on place le bit de poids faible (ou **Least Significant Bit**) $\varepsilon_0$ *après* la décomposition du quotient.

```
def binaire(n):
  if (n<2):
    return [n]
  else:
    return binaire(n//2)+[n%2]
```

C'est moins direct en style impératif, car l'insertion d'un élément en tête de liste n'est pas une opération aussi courante que l'insertion en queue de liste.

```
def binaire(n):
  L = []
  while (n>0):
    L.insert(0,n%2)
    n = n//2
  return L
```

## IV Algorithmes dichotomiques

### Insertion dans une liste triée

**R 7.a** On reprend la discussion du [**31.1**]. Dans les deux cas particuliers, on insère $x$ en tête ou en queue de liste. Dans le cas général, on détermine d'abord l'indice $r$ tel que $L_r \leqslant x < L_{r+1}$ avec `recherche_dicho`. Dans tous les cas, la liste retournée est construire par concaténation de listes.
Il faut veiller à placer $x$ au bon endroit dans la liste $L$, c'est-à-dire *après* $L_r$ (ou encore à la place de $L_{r+1}$).

```
def insertion(x, L):
  if (x<L[0]):
    return [x]+L
  elif (x>=L[-1]):
    return L+[x]
  else:
    i, j = recherche_dicho(x, L)
    return L[:j]+[x]+L[j:]
```

**R 7.b** Même principe qu'à la question précédente.

```
def insertion(x, L):
  if (x<L[0]):
    L.insert(x, 0)
  elif (x>=L[-1]):
    L.append(x)
  else:
    i, j = recherche_dicho(x, L)
    L.insert(j, x)
```

## V Recherche d'un motif

**R 8.** Dans l'écriture récursive, le cas où la longueur de la source est inférieure à la longueur du motif est envisagé comme un cas de terminaison de l'algorithme et ce cas se présente à chaque fois que le motif n'est pas contenu dans la source.
En style impératif, on se contente de parcourir le motif en comparant chaque caractère au caractère correspondant dans la source et il n'y a aucune raison valable d'imaginer que la longueur du motif soit supérieure à la longueur de la source. (On cherche un *mot* dans un *texte* !)

```
def est_prefixe(m, s):
  prfx = True
  for i in range(len(m)):
    prfx = prfx and (m[i]==s[i])
  return prfx
```

À la fin de la $i$-ème itération, le booléen `prfx` vaut `True` si, et seulement si,

$$\forall\ 0 \leqslant k < i, \qquad m_i = s_i.$$