Dossier : Code CLE

Nicolas Poulain

18 mars 2012

Table des matières

1 Introduction

La gestion des nombres de très grande taille peut poser des problèmes en informatique puisque généralement le type utilisé pour stocker ces entiers est de taille finie.

Pour pallier cet inconvénient, on a cherché d'autres façons d'écrire les nombres. L'une d'entre elles s'appelle le code CLE (Code à Large Echelle), elle se base sur le repérage de la position des 1 dans l'écriture binaire des nombres.

2 Code CLE et entiers naturels non nuls

2.1 Représentation informatique du code CLE d'un entier naturel non nul

Les écritures en base deux ne comportent que des 1 et des 0. Par exemple, avec un octet (8 bits) on ne peut représenter les entiers naturels que de 0 (00000000 en binaire) à 255 (11111111 en binaire). Il suffit donc de connaître la liste des positions de chiffres 1 pour connaître le nombre. La liste contient alors les exposants des puissances de deux dans l'écriture binaire, c'est-à-dire la position des bits. Ainsi, la nécessité du bit positionnel 0 disparaît. Le code CLE peut être considéré comme une formulation de l'écriture binaire. Voici deux exemples

– Le nombre 67 s'écrit 100011 en base 2. En effet on a la décomposition suivante :

$$67 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1$$
$$67 = 2^6 + 2^1 + 2^0$$

  67 s'écrit alors en code CLE sous la forme $(6; 1; 0)_{\text{CLE}}$.
– 1548 s'écrit 11000001100 en binaire et donc $(10; 9; 3; 2)_{\text{CLE}}$.

L'intérêt du code CLE est évident pour les "grands nombres" dont l'écriture binaire contient peu de 1. On pense en particulier aux puissances de deux comme 2^{64} dont l'écriture en code CLE est $(64)_{\text{CLE}}$ quand son écriture décimale est 18446744073709551616.

2.2 Remarques pédagogiques

Dans cette première partie, on traite uniquement le cas des nombres entiers naturels non nuls afin de simplifier le problème. Par ailleurs, on se limite à la programmation de fonctions itératives afin de travailler les notions de boucles.

Ce sujet présente de nombreux avantages du point de vue mathématique et algorithmique, en effet le sujet ne nécessite que des connaissances de base d'arithmétique, cependant certains calculs ne sont pas triviaux même pour des élèves de lycée. Par ailleurs le projet est très modulaire : chaque problème se résoud avec un algorithme de petite taille. De plus les difficultés sont très variées, on pourra ainsi facilement faire travailler les élèves par groupe. Les différentes méthodes et fonctions doivent cependant être cohérentes en ce qui concerne leurs entrées et sorties, ce qui oblige les groupes à bien fixer les règles du travail en commun.

Du point de vue informatique,

– Types de données. Taille des variables, de stockage et de traitement d'un nouveau type de données sont bien sûr à la base du projet. Un entier naturel en code CLE est une suite finie de nombres entiers naturels.

La première question à se poser concerne le type données le plus adapté à la représentation informatique de ces nombres en code CLE. La structure de tableau est intéressante mais dans de nombreux langages, la taille d'un tableau est fixée à la création et les opérations d'ajout ou de supression de colonnes peuvent être difficiles.

– Choix du langage. Il s'est porté sur Python car en plus d'être libre, on trouve facilement de la documentation et que le type `list` qui est proche de la notion de coe CLE est facile à manipuler puisqu'il s'agit d'une liste de longueur variable pour laquelle l'accès aux éléments peut se faire facilement par l'intermédiaire des indices.
– Programmation orientée objet. Comme le projet commence par traiter le cas des nombres entiers naturels non nuls avant de généraliser aux nombres décimaux relatifs, on fait le choix de la prgrammation orientée objet en commençant par définir une classe `cle` avec ses méthodes pour ensuite créer une classe dérivée `cle_etendu` qui héritera de la première et pour laquelle on pourra au besoin surcharger les méthodes.
– Commentaires et tests. Un programme Python est généralement documenté au sein même de son code source par un texte entouré de triple guillemets au début de chaque fonction afin de résumer ce qu'elle réalise et détailler les variables. Il est souvent utile de présenter des cas d'utilisation de la fonction et ce qui sera retourné, c'est pourquoi python propose un module appelé `doctest` dont le but est de permettre de réaliser des tests unitaires à l'intérieur même du commentaire. Un utilisateur pourra ainsi en regardant la documentation d'une fonction avoir directement accès à des exemples d'utilisation, exemples nécessairement à jour, puisque capables de passer les tests.

2.3 Premières fonctions

1. Écrire une méthode `cle_to_deci` capable de donner l'écriture décimale d'un nombre en code CLE.
2. Écrire une méthode `est_plus_grand_que` qui teste si un nombre en code CLE est plus grand qu'un autre.

2.4 Conversion d'un nombre en code CLE

On cherche à écrire une méthode `entier_to_cle` capable de donner l'écriture en code CLE d'un nombre entier naurel non nul. On peut procéder par divisions successives par 2. Voyons sur un exemple où on cherche l'écriture

en code CLE du nombre 13

$$\begin{aligned} 13 &= 2\times\underline{6}+1 \quad \ell_0 \\ \underline{6} &= 2\times\underline{3}+0 \quad \ell_1 \\ \underline{3} &= 2\times\underline{1}+1 \quad \ell_2 \\ \underline{1} &= 2\times\underline{0}+1 \quad \ell_3 \end{aligned} \qquad \longrightarrow 13 = 2^3+2^2+2^0$$

2.5 Somme de nombres en code CLE

Avant de programmer, quelques questions :

1. Que vaut la somme $(15)_{\text{CLE}} + (15)_{\text{CLE}}$, puis pour $n \in \mathbb{N}$, la somme $(n)_{\text{CLE}} + (n)_{\text{CLE}}$?
2. Que vaut la somme $(11;5;0)_{\text{CLE}} + (34;11;5)_{\text{CLE}}$?

L'algorithme d'addition peut être construit de plusieurs façons, voici deux pistes

– **Somme avec tri préalable** : `somme_tri` — Concaténer puis ordonner les deux listes correspondant aux deux nombres à additionner. Ensuite, parcourir de droite à gauche à la recherche de couples $\{n,n\}$ qu'on remplacera dans la liste par le singleton $\{n+1\}$. On s'arrête quand, ayant parcouru la liste, on n'a trouvé aucune paire. Exemple

$$\begin{aligned} (6;5;2;0)_{\text{CLE}} + (5;2)_{\text{CLE}} &= (6;5;5;\underline{2;2};0)_{\text{CLE}} && = 2^6+2^5+2^5+(2^2+2^2)+2^0 \\ &= (6;\underline{5;5};3;0)_{\text{CLE}} && = 2^6+(2^5+2^5)+2^3+2^0 \\ &= (\underline{6;6};3;0)_{\text{CLE}} && = (2^6+2^6)+2^3+2^0 \\ &= (7;3;0)_{\text{CLE}} && = 2^7+2^3+2^0 \end{aligned}$$

– **Somme avec tri à bulle** : `somme_bulle` — Concaténer sans ordonner les deux listes en une liste s et parcourir de droite à gauche en comparant les termes consécutifs $s(i-1)$ et $s(i)$; s'ils sont égaux, on remplace $\{s(i-1),s(i)\}$ par $\{s(i-1)+1\}$ et si $s(i-1)$ est strictement inférieur à $s(i)$, on les permute. On s'arrête quand, ayant parcouru la liste, on n'a trouvé aucune paire ni éléments à intervertir. Exemple

$$\begin{aligned} (6;5;2;0)_{\text{CLE}} + (5;2)_{\text{CLE}} &= (6;5;2;\underline{0;5};2)_{\text{CLE}} && = 2^6+2^5+2^2+2^0+2^5+2^2 \\ &= (6;5;2;5;\underline{0;2})_{\text{CLE}} && = 2^6+2^5+2^2+2^5+2^0+2^2 \\ &= (6;5;\underline{2;5};2;0)_{\text{CLE}} && = 2^6+2^5+2^2+2^5+2^2+2^0 \\ &= (6;5;5;\underline{2;2};0)_{\text{CLE}} && = 2^6+2^5+2^5+2^2+2^2+2^0 \\ &= (6;\underline{5;5};3;0)_{\text{CLE}} && = 2^6+2^5+2^5+2^3+2^0 \\ &= (\underline{6;6};3;0)_{\text{CLE}} && = 2^6+2^6+2^3+2^0 \\ &= (7;3;0)_{\text{CLE}} && = 2^7+2^3+2^0 \end{aligned}$$

2.6 Différence (positive) de nombres en code CLE

La méthode `est_plus_grand_que` permet de vérifier que la différence donnera un résultat positif, on s'attache donc à construire un algorithme de soustraction du type suivant :

$$(a_1;\dots;a_n)_{\text{CLE}} - (b_1;\dots;b_m)_{\text{CLE}} \quad \text{avec} \begin{cases} n\in\mathbb{N},\ a_1,\dots,a_n\in\mathbb{N} \\ m\in\mathbb{N},\ b_1,\dots,b_m\in\mathbb{N} \\ (a_1;\dots;a_n)_{\text{CLE}} \geqslant (b_1;\dots;b_m)_{\text{CLE}} \end{cases}$$

L'algorithme de différence peut être construit de plusieurs façons, voici une piste.

Remarque préliminaire : pour $n,m\in\mathbb{N},\ n>m$, on a $2^n-2^m = 2^{n-1}+\cdots+2^m$. Aussi

$$\text{pour } a_k \geq b_1 \text{ on a } (a_k)-(b_1) = (a_k-1, a_k-2, \dots, b_1)$$

On souhaite calculer $D = (a_1, \ldots, a_i) - (b_1, \ldots, b_j)$. Notons a_k le plus petit élément de $\{a_1, \ldots, a_i\}$ tel que $a_k \geq b_1$. Alors

$$\begin{aligned} D &= (a_1, a_2 \ldots, a_{k-1}, \underline{a_k}, a_{k+1}, a_{k+2} \ldots, a_i) - (\underline{b_1}, b_2, \ldots, b_j) \\ &= (a_1, a_2 \ldots, a_{k-1}, \underline{a_k - 1, a_k - 2, \ldots, b_1}, a_{k+1}, a_{k+2} \ldots, a_i) - (b_2, \ldots, b_j) \end{aligned}$$

Sur un exemple

$$\begin{aligned} D &= (\underline{7}, 3, 0) - (\underline{4}, 3, 1) \\ &= (\underline{6, 5, 4}, 3, 0) - (3, 1) \\ &= (6, 5, 4, \underline{3}, 0) - (\underline{3}, 1) \\ &= (6, 5, 4, 0) - (1) \\ &= (6, 5, \underline{4}, 0) - (\underline{1}) \\ &= (6, 5, \underline{3, 2, 1}, 0) \end{aligned}$$

2.7 Produit de nombres en code CLE

Avant de programmer, quelques questions :

1. Ecrire en code CLE le produit $(n)_{\text{CLE}} \times (m)_{\text{CLE}}$ où n et m sont deux entiers naturels.
2. Ecrire en code CLE :
3. le produit $(5; 2; 0)_{\text{CLE}} \times (4)_{\text{CLE}}$
4. le carré : $(12; 4)^2_{\text{CLE}}$
5. le produit $(5; 3)_{\text{CLE}} \times (7; 2; 1)_{\text{CLE}}$.
6. Énoncer une règle générale.

Construire un algorithme de multiplication et le programmer.

2.8 Division euclidienne de nombres en code CLE

Construire un algorithme de division euclidienne pour des nombres en code CLE. Écrire les fonctions `quotient` et `reste` capables de donner le quotient et le reste dans la divsion de deux nombres en code CLE.

2.9 Programme Python

```
#!/usr/bin/python
import math   # This will import math module

class cle():
  """La classe cle : nombres positifs non nuls en code CLE"""
  pass

  def __init__(self, lst):
    """Initialisation de la classe cle (constructeur)"""
    try:
      lst = list(lst)
    except TypeError:
        print("l'argument␣doit␣etre␣de␣type␣list.")
    self.lst = lst

  def affiche(self):
    """Methode d'affichage d'un code CLE
    - Exemples :
    >>> cle([7, 3, 0]).affiche()
    [7, 3, 0]
    >>> p = cle(7)
```

```
    l'argument doit etre de type list.
    """
    return self.lst

  ## Trois methodes redefinies... pour gagner du temps
  def __str__(self):
    """Permet d'afficher les messages d'erreur correctement"""
    return self.lst
  def __len__(self):
    """Permet de demander len(cle) plutot que len(cle.lst)"""
    return len(self.lst)
  def __getitem__(self,key):
    """Permet de demander cle[i] plutot que cle.lst[i]"""
    return self.lst[key]

  def cle_to_deci(self):
    """Methode de calcul de l'ecriture decimale d'un nombre en code CLE.
    - Exemples :
    >>> p = cle([6, 1])
    >>> p.cle_to_deci()
    66.0
    >>> cle([7, 3, 0]).cle_to_deci()
    137.0
    """
    n = 0
    for i in range(len(self)):
      n = n + math.pow(2, self[i])
    return n

  def est_plus_grand_que(self,other):
    """Methode de test du classement de deux nombres en code CLE.
    - Algorithme : Comparer les premiers elements de chaque
    liste jusqu'a obtenir une comparaison.
    En cas d'egalite de tous les elements communs aux deux listes,
    on departage les deux listes par leur longueur.
    - Exemples :
    >>> p = cle([7, 3, 0])
    >>> q = cle([7, 2])
    >>> p.est_plus_grand_que(q)
    True
    >>> q.est_plus_grand_que(cle([7, 2, 1, 0]))
    False
    """
    i = 0
    while self[i] == other[i] and i < min(len(self),len(other))-1:
      i = i+1
    if self[i] > other[i] or ( self[i] == other[i] and len(self) > len(other) ):
      return True
    return False

  def __add__(self,other):
    """Redefinition pour la classe cle de la methode __add__
    qui implemente l'operateur d'addition +.
    L'operation c + d sera resolue pour les codes cle.
    - Exemples :
    >>> ( cle([7, 3, 0]) + cle([7, 1]) ).affiche()
    [8, 3, 1, 0]
    >>> ( cle([11, 1, 0]) + cle([11, 7, 3, 0]) ).affiche()
    [12, 7, 3, 2]
    """
    return cle(somme_bulle(self.lst,other.lst))
    #return cle(somme_tri(self.lst,other.lst))

  def __sub__(self,other):
    """Redefinition pour la classe cle de la methode __sub__
    qui implemente l'operateur de soustraction -.
    L'operation c - d sera resolue pour les codes cle.
    - Exemples :
    >>> ( cle([7, 3, 0]) - cle([7, 1]) ).affiche()
    [2, 1, 0]
    >>> cle([7, 3, 0]) - cle([8])
    'Erreur : la difference doit etre positive'
    >>> ( cle([12]) - cle([11, 7, 3, 0]) ).affiche()
    [10, 9, 8, 6, 5, 4, 2, 1, 0]
    """
    c = diff_cle(self.lst,other.lst)
```

```
    if not isinstance(c, list):
      return("Erreur : la difference doit etre positive")
    return cle(c)

  def __mul__(self,other): # implemete l'operateur *
    """Redefinition pour la classe cle de la methode __mul__
    qui implemente l'operateur de multiplication *.
    L'operation c * d sera resolue pour les codes cle.
    - Exemples :
    >>> ( cle([5, 2, 0]) * cle([4]) ).affiche()
    [9, 6, 4]
    >>> ( cle([12, 4]) * cle([12, 4]) ).affiche()
    [24, 17, 8]
    >>> ( cle([5, 3]) * cle([7, 2, 0]) ).affiche()
    [12, 10, 7, 6, 3]
    """
    c = []
    for i in range(len(self)):
      for j in range(len(other)):
        c = c + [ self[i] + other[j] ]
    return cle(c) + cle([])

  #def __div__(self,other): # implemente l'operateur /

def entier_to_cle(n):
  """Fonction de calcul de la liste des nombres apparaissant
  dans le code CLE d'un nombre entier en base 10 donne.
  - Algorithme : Etude du reste dans les divisions
  successives par deux
  - Exemples :
  >>> entier_to_cle(137).affiche()
  [7, 3, 0]
  >>> entier_to_cle(66).affiche()
  [6, 1]
  """
  x = float(n) # pour premier le test de parite dans la boucle while
  c = cle([])
  i = 0
  while x>0:
    if math.floor(x/2) != x/2 :
      c.lst.insert(0,i)
    x = math.floor(x/2)
    i = i+1
  return c

def somme_tri(c,d):
  """Fonction de calcul de la liste des nombres apparaissant
  dans le code cle de la somme de deux nombres en
  code CLE (deux listes non vides)
  - Algorithme : Tri prealable d'une liste formee des
  deux listes fusionnees puis sommation des jumeaux
  """
  s = c+d
  s.sort(reverse=True)
  for i in range(len(s)-1,0,-1):
    if s[i-1]==s[i]:   # addition
      s[i-1] = s[i]+1
      s.pop(i)
  return s

def somme_bulle(c,d):
  """Fonction de calcul de la liste des nombres apparaissant
  dans le code cle de la somme de deux nombres en
  code CLE (deux listes non vides)
  - Algorithme : Tri a bulle et puis sommation
  des jumeaux
  """
  s = c+d
  flag = 0
  while flag==0 and len(s)>1:
    flag = 1
    for i in range(len(s)-1,0,-1):
      if s[i-1]==s[i]:   # addition
        s[i-1] = s[i]+1
        s.pop(i)
        flag = 0
```

```
            elif s[i-1]<s[i]:  # permutation
              tmp = s[i]
              s[i] = s[i-1]
              s[i-1] = tmp
              flag = 0
        return s

    def diff_cle(c,d):
        """Fonction de calcul de la liste des nombres apparaissant
        dans le code cle de la difference de deux nombres positifs
        c et d (c>d) en code CLE (deux listes non vides)
        - Algorithme : voir la documentation
        """
        if cle(d).est_plus_grand_que(cle(c)):
            return "Erreur : la difference doit etre positive"
        u = c
        for i in range(len(d)):
            n = d[i]
            j = len(u)-1
            while ( u[j]<n and j>0 ):
                j = j-1
            u = u[:j] + [k for k in range(u[j]-1,n-1,-1)] + u[j+1:]
        return u

    if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

voir http ://code.google.com/p/code-cle/ pour le code à jour.

3 Code CLE et nombres décimaux relatifs

Afin d'aller plus loin, et de pouvoir traiter les nombres décimaux relatifs, il faut redéfinir le cadre, c'est à dire repenser la représentation du nombre. Avant cela, voyons comment est généralement défini un nombre réel dans une machine.

3.1 Représentation binaire en virgule flottante

La représentation en virgule flottante d'un nombre binaire sur 32 bits ou 64 bits (double précision) est définie par trois composantes :

- le signe S est représenté par un seul bit : le bit de poids fort (le plus à gauche)
- l'exposant E est codé sur les 8 bits consécutifs au signe en excès de 127 (simple précision) ou 1023 (double précision)
- la mantisse M (bits situés après la virgule) est représentée sur les 23 (simple précision) ou 55 (double précision) bits restants.

Ainsi le codage sur un mot de 32 bits se fait sous la forme suivante :

$$s \ \underbrace{eeeeeeee}_{E} \ \underbrace{mmmmmmmmmmmmmmmmmmmmmmm}_{M}$$

Dans cette représentation la valeur d'un nombre N sur 32 bit est donné par l'expression :

$$N = (-1)^s \times (1 + M \times 2^{-23}) \times 2^{(E-127)}$$

Exemple : Soit à coder la valeur 525,5.

525,5 s'écrit en base 2 de la façon suivante : 1000001101,1 on veut l'écrire sous la forme $1,0000011011 \times 2^9$. Par conséquent :

- le bit s vaut 1
- l'exposant $E = 9 + 127 = 136$, soit 10010000
- la mantisse est $M = 0000011011$

La représentation du nombre 525,5 en binaire est :

$$1\ 10010000\ 00000110110000000000000$$

Remarque : Le zéro est le seul nombre qui ne peut pas se représenter dans le système à virgule flottante à cause de la contrainte de non nullité du premier digit $(1, M \times 2^E)$. Le zéro est donc représenté par une convention de représentation. La norme IEEE754 précise cette convention de représentation $(E = M = 0)$, c'est à dire 1 0 0.

3.2 Représentation informatique du code CLE d'un nombre réel

La méthode décrite ci-dessus ne permet pas d'écrire des nombres supérieurs à $1,11111111111111111111111 \times 2^{127}$ de plus la précision des nombres réels non entiers est limitée. Nous allons donc étendre le système de codage CLE

1. en autorisant les élémnts de la liste à prendre des valeurs négatives
2. en plaçant au premier rang un coefficient $s \in \{-1; 0; 1\}$ qui permettra de représenter les nombres négatifs anisi que le nombre zéro.

Ainsi, le code CLE $(-1; 9; 3; 2; 0; -2)_{\text{CLE}}$ représente le nombre

$$-1 \times (2^9 + 2^3 + 2^2 + 2^0 + 2^{-2}) = -525,25$$

3.3 Remarques pédagogiques

Dans cette seconde partie, on construit la classe `cle_etendu` dérivée de la classe `cle` et on s'appuie sur le polymorphisme afin de rendre transparente la réutilisation des méthodes de la classe parent.

Dans cette partie, le typage fort du langage python ne pardonne pas les approximations. Les élèves devront veiller à toujours avoir en tête le type ou la classe des variables objets utilisés.

Des algorithmes récursifs seront mis en œuvre chaque fois que la situation s'y prêtera.

3.4 Programmation

Adapter (si l'héritage ne suffit pas) toutes les méthodes de la classe `cle` à la classe `cle_etendu`.

```
#!/usr/bin/python
import math   # This will import math module
from classCle import *

class cle_etendu(cle):
  """La classe cle : nombres decimaux relatifs en code CLE"""
  pass

  def cle_to_deci(self):
    """Surcharge de la methode de la classe cle pour la prise
    en compte du bit de signe
    - Exemples :
    >>> p = cle_etendu([-1, 9, 3, 2, 0, -2])
    >>> p.cle_to_deci()
    -525.25
    >>> cle_etendu([0, 0]).cle_to_deci()
    0.0
    >>> cle_etendu([1, 19, -7]).cle_to_deci()
    524288.0078125
    """
    return self[0] * cle(self[1:]).cle_to_deci()

  def est_plus_grand_que(self,other):
    """Surcharge de la methode de la classe cle pour la prise
    en compte du bit de signe
    - Exemples :
    >>> cle_etendu([1, 5]).est_plus_grand_que(cle_etendu([1, 4]))
    True
    >>> cle_etendu([-1, 5]).est_plus_grand_que(cle_etendu([-1, 4]))
```

```
    False
    >>> cle_etendu([1, 5]).est_plus_grand_que(cle_etendu([-1, 4]))
    True
    >>> cle_etendu([1, 4]).est_plus_grand_que(cle_etendu([-1, 5]))
    True
    """
    if self[0] > other[0]:
      return True     # cas signe(a) > signe(b)
    elif self[0] < other[0]:
      return False    # cas signe(a) < signe(b)
    elif self[0]==1:
      if cle(self[1:]).est_plus_grand_que(cle(other[1:])):
        return True   # cas a >  b > 0
      return False    # cas b >= a > 0
    elif self[0]==-1:
      if cle(self[1:]).est_plus_grand_que(cle(other[1:])):
        return False  # cas a <  b < 0
      return True     # cas b <= a < 0
    return True       # cas a =  b = 0

  def __add__(self,other):
    """Redefinition pour la classe cle_etendu de la methode __add__
    qui implemente l'operateur d'addition +.
    L'operation c + d sera resolue pour les codes cle_etendu.
    - Exemples :
    >>> p = cle_etendu([-1, 7, 3, 0, -3])
    >>> q = cle_etendu([1, 7, 1])
    >>> print p.cle_to_deci(),"+",q.cle_to_deci(),"=",(p+q).affiche(),"=",(p+q).cle_to_deci()
    -137.125 + 130.0 = [-1, 2, 1, 0, -3] = -7.125
    >>> r = cle_etendu([1, 9, 1])
    >>> print p.cle_to_deci(),"+",r.cle_to_deci(),"=",(p+r).affiche(),"=",(p+r).cle_to_deci()
    -137.125 + 514.0 = [1, 8, 6, 5, 4, 3, -1, -2, -3] = 376.875
    """
    if self[0] * other[0] > 0:
      c = [self[0]] + ( cle(self[1:]) + cle(other[1:]) ).lst
    elif self[0] * other[0] <0:
      if cle(self[1:]).est_plus_grand_que(cle(other[1:])):
        c = [self[0]] + ( cle(self[1:]) - cle(other[1:]) ).lst
      else:
        c = [other[0]] + ( cle(other[1:]) - cle(self[1:]) ).lst
    elif self[0]==0:
        c = other.lst
    else:
        c = self.lst
    return cle_etendu(c)

  def __sub__(self,other):
    """Redefinition pour la classe cle_etendu de la methode __sub__
    qui implemente l'operateur d'addition -.
    L'operation c - d sera resolue pour les codes cle_etendu.
    Algorithme : implementation du fait que c - d = c + (-d)
    """
    return self + cle_etendu( -1*other[0] + cle(other[1:]).lst )

  def __mul__(self,other):
    """Redefinition pour la classe cle_etendu de la methode __mul__
    qui implemente l'operateur de multiplication *.
    L'operation c * d sera resolue pour les codes cle_etendu.
    - Exemples :
    >>> ( cle_etendu([-1, 5, 2, 0]) * cle_etendu([-1, 4]) ).affiche()
    [1, 9, 6, 4]
    >>> ( cle_etendu([1, 12, 4]) * cle_etendu([-1, 12, 4]) ).affiche()
    [-1, 24, 17, 8]
    >>> ( cle_etendu([-1, 5, 3]) * cle_etendu([1, 7, 2, 0]) ).affiche()
    [-1, 12, 10, 7, 6, 3]
    """
    signe = self[0]*other[0]
    val_abs = ( cle(self[1:]) * cle(other[1:]) ).lst
    return cle_etendu( [signe] + val_abs )

def deci_to_cle(n,prof_max):
  """Fonction de calcul de la liste des nombres apparaissant
  dans le code CLE d'un nombre decimal relatif en base 10 donne.
  Une profondeur maximale est a prciser puisque l'ecriture CLE
  d'un nombre decimal n'est pas necessairement finie.
  - Algorithme : on reutilise la fonction programmee pour la partie entiere
```

```
    on cree un fonction recursive pour la partie fractionnaire
    - Exemples :
    >>> deci_to_cle(-525.25, 5).affiche()
    [-1, 9, 3, 2, 0, -2]
    >>> deci_to_cle(524288.0078125, 10).affiche()
    [1, 19, -7]
    >>>
    """
    if n==0:
      return [0, 0]
    # Partie entiere
    p_ent = math.floor(abs(n))
    p_ent_lst = entier_to_cle(p_ent).lst
    # Partie fractionnaire
    p_frac = float(abs(n)-math.floor(abs(n)))
    p_frac_lst = frac_to_cle(p_frac,prof_max,0,[])
    # Conacatenation du signe et des deux parties
    return cle_etendu([int(abs(n)/n)] + p_ent_lst + p_frac_lst)

def frac_to_cle(p_frac, prof_max, idx, cle_p_frac):
    """Fonction a appel recursif pour le calcul de
    la representation CLE de la partie decimale d'un nombre.
    """
    if p_frac==0 or idx==prof_max:
      return list(cle_p_frac)
    if 2*p_frac>=1:
      return frac_to_cle(2*p_frac-1, prof_max, idx+1, cle_p_frac+[-idx-1])
    else:
      return frac_to_cle(2*p_frac, prof_max, idx+1, cle_p_frac)


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```