Codage de Huffman

Marc Lorenzi

29 décembre 2018

In [1]:
```
import math
import heapq
```

Voici trois chaînes de caractères que nous utiliserons dans tout le notebook. La chaîne `abra` est très courte et a des propriétés intéressantes. La chaîne `ovide` est un peu plus longue.

In [2]:
```
abra = 'abracadabra'
ovide = 'Ante mare et terras et quod tegit omnis caelum unus erat toto
```

1 Notion de code

1.1 Alphabets, lettres, mots

Un **alphabet** A est un ensemble fini d'objets. Les éléments d'un alphabet sont appelés des **lettres**. Jusque là tout le monde me suit ? L'alphabet auquel nous pensons tous est évidemment $A = \{a, b, \dots, z\}$, auquel nous pourrions rajouter les lettres majuscules, les chiffres, les symboles de ponctuation, etc. Mais n'importe quel ensemble fini fera l'affaire.

Un **mot** sur l'alphabet A est une suite finie, éventuellement vide, de lettres.

Notation : On note A^* l'ensemble des mots sur l'alphabet A. Un mot de **longueur** (ou **taille**) $n \geq 1$, $u = (a_0, \dots, a_{n-1})$ est noté plus simplement $u = a_0 a_1 \dots a_{n-1}$ par la juxtaposition de ses lettres. Le mot vide, l'unique mot de longueur 0, qui n'a pas de lettre, est noté ε si besoin. Nous noterons $|u|$ la longueur du mot u.

Nous disposons donc de la fonction $|\bullet| : A^* \to \mathbb{N}$ définie par $u \mapsto |u|$.

Convention : Nous identifierons un mot de longueur 1 avec son unique lettre. Avec cette convention, $A \subset A^*$.

Dans toute la suite, A désigne un alphabet. Lorsque nous écrivons $u = a_0 \dots a_{n-1}$ et que u est un mot, les a_i sont les lettres de u et n est sa longueur. Le mot vide ne sera pas systématiquement traité comme il conviendrait à son rang, le lecteur est censé s'assurer que ce que nous raconterons s'applique aussi dans ce cas.

1.2 Produit de mots

L'ensemble A^* est muni de l'opération de **concaténation** (ou **produit**) des mots, que nous noterons multiplicativement :

Définition :

- Soient $u = a_0 a_1 \dots a_{m-1}$ et $v = b_0 b_1 \dots b_{m-1}$ deux mots non vides. Le produit de u et v est le mot

$$uv = a_0 a_1 \dots a_{m-1} b_0 b_1 \dots b_{n-1}$$

- Si $u = \epsilon$ est le mot vide, on pose $uv = v$.
- Si $v = \epsilon$ est le mot vide, on pose $uv = u$.

Propriété : A^*, muni du produit des mots, est un monoïde en général non commutatif. Précisément :

- Pour tous $u, v, w \in A^*$, $(uv)w = u(vw)$.
- Pour tout $u \in A^*$, $u\varepsilon = \varepsilon u = u$.
- L'opération de concaténation est non commutative dès que A possède au moins 2 lettres.

Démonstration : Sans difficulté particulière. Noter que si a et b sont deux lettres différentes de l'alphabet A, alors $a.b \neq b.a$. Le produit n'est donc pas commutatif si A possède au moins deux lettres.

Propriété : Tout mot est le produit de ses lettres.

Propriété : Pour tous mots $u, v \in A^*$, $|uv| = |u| + |v|$. La fonction $|\bullet|$ est donc un morphisme de monoïdes.

Démonstrations : faciles.

Remarque : Dans tout ce notebook, sauf tout à la fin dans la partie théorique, nous prendrons pour A un ensemble contenant uniquement des lettres "usuelles" a, b, etc, des chiffres, des espaces, des caractères de ponctuation. Cela nous permet de représenter en Python un mot sur l'alphabet A par une **chaîne de caractères** (`string`).

1.3 Coder

Une étape essentielle dans le stockage ou la transmission de l'information est le **codage** des mots. Pour être stocké ou transmis, un mot doit être codé. Et ce que nous obtenons après codage doit pouvoir être décodé (c'est préférable). Pour ceux d'entre-vous qui se disent qu'ils ne stockent jamais de mots, dites vous qu'un fichier n'est qu'un long mot dont les lettres sont, par exemple, les octets du fichier. Nous allons travailler dans ce notebook sur des chaînes de caractères mais *tout ce que je vais raconter peut s'adapter assez facilement à des fichiers concrets*.

Définition incomplète : Un code est une fonction $c : A \to \mathbb{B}^*$ où $\mathbb{B} = \{0, 1\}$.

Proposition : Soit c un code. Il existe un unique morphisme de monoïdes $c^* : A^* \to \mathbb{B}^*$ tel que pour tout $u \in A$, $c^*(u) = c(u)$.

Preuve : Soit $u = a_0 \dots a_{n-1} \in A^*$. Si un tel morphisme existe, on a $c^*(u) = c(a_0) \dots c(a_{n-1})$ puisque l'image d'un produit est le produit des images. Inversement, cette fonction convient.

Bref, si on sait coder les lettres alors on sait coder les mots.

Exercice : Pensez-vous au mot vide ? Que vaut $c^*(\varepsilon)$? Indication : Un morphisme conserve le neutre.

1.4 Le code ASCII

ASCII signifie "American Standard Code for Information Interchange". Le code ASCII a été développé au paléolithique, plus précisément dans les années 1960. C'est une norme de codage des caractères. Le code ASCII "pur" code 128 caractères par des mots de 7 bits sur l'alphabet $\mathbb{B}$. Par exemple, la lettre a est codée `01100001`, la lettre b est codée `01100010`, etc. En interprétant ces mots de $\mathbb{B}^*$ comme des entiers en base 2, on retient plutôt que le code ASCII de a est 97, celui de b est 98, etc.

Il existe de nombreuses extensions du code ASCII permettant de coder $2^8 = 256$ caractères. La plus utilisée par les français est le code ISO 8859-1, appelé aussi ISO Latin-1. Il permet de coder, en particulier, les lettres accentuées, par des codes supérieurs ou égaux à 128.

Remarque : Le code ASCII s'est avéré au fil du temps insuffisant pour coder les nombreux caractères des langues de la Galaxie. Son évolution est le code **Unicode** qui peut coder $65536 = 2^{16}$ caractères, dont je ne parlerai pas plus avant. Les idées sous-tendant Unicode et ses implémentations concrètes sont intéressantes et je ne saurais trop vous conseiller de vous documenter sur le sujet.

La fonction Python `ord` permet d'obtenir le code ASCII d'un caractère.

In [3]:
```
for c in 'Je vais à la pêche le dimanche.':
    print(c, ord(c))
```

```
J 74
e 101
  32
v 118
a 97
i 105
s 115
  32
à 224
  32
l 108
a 97
  32
p 112
ê 234
c 99
h 104
e 101
  32
l 108
e 101
  32
d 100
i 105
m 109
a 97
n 110
c 99
h 104
e 101
. 46
```

Sa "réciproque" est la fonction `chr` .

In [4]:
```
print([(k, chr(k)) for k in range(256)])
```

```
[(0, '\x00'), (1, '\x01'), (2, '\x02'), (3, '\x03'), (4, '\x04'), (5
, '\x05'), (6, '\x06'), (7, '\x07'), (8, '\x08'), (9, '\t'), (10, '\
n'), (11, '\x0b'), (12, '\x0c'), (13, '\r'), (14, '\x0e'), (15, '\x0
f'), (16, '\x10'), (17, '\x11'), (18, '\x12'), (19, '\x13'), (20, '\
x14'), (21, '\x15'), (22, '\x16'), (23, '\x17'), (24, '\x18'), (25,
'\x19'), (26, '\x1a'), (27, '\x1b'), (28, '\x1c'), (29, '\x1d'), (30
, '\x1e'), (31, '\x1f'), (32, ' '), (33, '!'), (34, '"'), (35, '#'),
(36, '$'), (37, '%'), (38, '&'), (39, "'"), (40, '('), (41, ')'), (4
2, '*'), (43, '+'), (44, ','), (45, '-'), (46, '.'), (47, '/'), (48,
'0'), (49, '1'), (50, '2'), (51, '3'), (52, '4'), (53, '5'), (54, '6
'), (55, '7'), (56, '8'), (57, '9'), (58, ':'), (59, ';'), (60, '<')
, (61, '='), (62, '>'), (63, '?'), (64, '@'), (65, 'A'), (66, 'B'),
(67, 'C'), (68, 'D'), (69, 'E'), (70, 'F'), (71, 'G'), (72, 'H'), (7
3, 'I'), (74, 'J'), (75, 'K'), (76, 'L'), (77, 'M'), (78, 'N'), (79,
'O'), (80, 'P'), (81, 'Q'), (82, 'R'), (83, 'S'), (84, 'T'), (85, 'U
'), (86, 'V'), (87, 'W'), (88, 'X'), (89, 'Y'), (90, 'Z'), (91, '['
), (92, '\\'), (93, ']'), (94, '^'), (95, '_'), (96, '`'), (97, 'a'),
(98, 'b'), (99, 'c'), (100, 'd'), (101, 'e'), (102, 'f'), (103, 'g')
, (104, 'h'), (105, 'i'), (106, 'j'), (107, 'k'), (108, 'l'), (109,
'm'), (110, 'n'), (111, 'o'), (112, 'p'), (113, 'q'), (114, 'r'), (1
15, 's'), (116, 't'), (117, 'u'), (118, 'v'), (119, 'w'), (120, 'x')
, (121, 'y'), (122, 'z'), (123, '{'), (124, '|'), (125, '}'), (126,
'~'), (127, '\x7f'), (128, '\x80'), (129, '\x81'), (130, '\x82'), (1
31, '\x83'), (132, '\x84'), (133, '\x85'), (134, '\x86'), (135, '\x8
7'), (136, '\x88'), (137, '\x89'), (138, '\x8a'), (139, '\x8b'), (14
0, '\x8c'), (141, '\x8d'), (142, '\x8e'), (143, '\x8f'), (144, '\x90
'), (145, '\x91'), (146, '\x92'), (147, '\x93'), (148, '\x94'), (149
, '\x95'), (150, '\x96'), (151, '\x97'), (152, '\x98'), (153, '\x99'
), (154, '\x9a'), (155, '\x9b'), (156, '\x9c'), (157, '\x9d'), (158,
'\x9e'), (159, '\x9f'), (160, '\xa0'), (161, '¡'), (162, '¢'), (163,
'£'), (164, '¤'), (165, '¥'), (166, '¦'), (167, '§'), (168, '¨'), (1
69, '©'), (170, 'ª'), (171, '«'), (172, '¬'), (173, '\xad'), (174, '
®'), (175, '¯'), (176, '°'), (177, '±'), (178, '²'), (179, '³'), (18
0, '´'), (181, 'µ'), (182, '¶'), (183, '·'), (184, '¸'), (185, '¹'),
(186, 'º'), (187, '»'), (188, '¼'), (189, '½'), (190, '¾'), (191, '¿
'), (192, 'À'), (193, 'Á'), (194, 'Â'), (195, 'Ã'), (196, 'Ä'), (197
, 'Å'), (198, 'Æ'), (199, 'Ç'), (200, 'È'), (201, 'É'), (202, 'Ê'),
(203, 'Ë'), (204, 'Ì'), (205, 'Í'), (206, 'Î'), (207, 'Ï'), (208, 'Ð
'), (209, 'Ñ'), (210, 'Ò'), (211, 'Ó'), (212, 'Ô'), (213, 'Õ'), (214
, 'Ö'), (215, '×'), (216, 'Ø'), (217, 'Ù'), (218, 'Ú'), (219, 'Û'),
(220, 'Ü'), (221, 'Ý'), (222, 'Þ'), (223, 'ß'), (224, 'à'), (225, 'á
'), (226, 'â'), (227, 'ã'), (228, 'ä'), (229, 'å'), (230, 'æ'), (231
, 'ç'), (232, 'è'), (233, 'é'), (234, 'ê'), (235, 'ë'), (236, 'ì'),
(237, 'í'), (238, 'î'), (239, 'ï'), (240, 'ð'), (241, 'ñ'), (242, 'ò
'), (243, 'ó'), (244, 'ô'), (245, 'õ'), (246, 'ö'), (247, '÷'), (248
, 'ø'), (249, 'ù'), (250, 'ú'), (251, 'û'), (252, 'ü'), (253, 'ý'),
(254, 'þ'), (255, 'ÿ')]
```

Histoire de nous échauffer, écrivons une fonction Python qui code un mot au moyen du code ASCII.

Voici tout d'abord une fonction qui prend en entier un entier n et renvoie sa représentation binaire sous forme d'une chaîne de 8 caractères.

In [5]:
```
def entier_vers_binaire(n):
    s = ''
    for k in range(8):
        s = str(n % 2) + s
        n = n // 2
    return s
```

In [6]:
```
entier_vers_binaire(97)
```

Out[6]:
```
'01100001'
```

Le codage d'un mot u est alors immédiat. Il suffit de concaténer les codes de ses lettres.

In [7]:
```
def coder_ascii(u):
    s = ''
    for a in u:
        s = s + entier_vers_binaire(ord(a))
    return s
```

Le mot `ovide` , par exemple, est codé en ASCII sur 816 bits.

In [8]:
```
v = coder_ascii(ovide)
print(v)
print(len(v))
```

```
0100000101101110011101000110010100100000011011010110000101110010011
0101001000000110010101110100001000000111010001100101011100100111001
0110000101110011001000000110010101110100001000000111000101110101011
1111011001000010000001110100011001010110011101101001011101000010000
0110111101101101011011100110100101110011001000000110001101100001011
0101011011000111010101101101001000000111010101101110011101010111011
0010000001100101011100100110000101110100001000000111010001101111011
0100011011110010000001101110011000010111010001110101011100100110001
0110010100100000011101010111010101101100011101000111010101110011001
0000011010010110111000100000011011110111001001100010011001010010000
0111000101110101011001010110111010010000001100100011010010111100001
1
0101011100100110010100100000010000110110100001100001011011110111001
816
```

1.5 Représenter un code en Python. Coder

Décidons de représenter un code par un dictionnaire. Si c est un code, nous accédons donc à $c(a)$ par l'expression `c[a]` .

In [9]:
```
code = {'a': '00', 'b': '01', 'c': '10', 'd' : '11'}
```

In [10]:
```
def coder(s, code):
    s1 = ''
    for a in s:
        s1 = s1 + code[a]
    return s1
```

In [11]:
```
coder('babaca', code)
```

Out[11]:
```
'010001001000'
```

Voici un sous-code du code ASCII.

In [12]:
```
def sous_ascii():
    c = {}
    s = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz 01234567
    for a in s:
        c[a] = entier_vers_binaire(ord(a))
    return c
```

In [13]:
```
mini_ascii = sous_ascii()
print(mini_ascii)
```

```
{'A': '01000001', 'B': '01000010', 'C': '01000011', 'D': '01000100',
'E': '01000101', 'F': '01000110', 'G': '01000111', 'H': '01001000',
'I': '01001001', 'J': '01001010', 'K': '01001011', 'L': '01001100',
'M': '01001101', 'N': '01001110', 'O': '01001111', 'P': '01010000',
'Q': '01010001', 'R': '01010010', 'S': '01010011', 'T': '01010100',
'U': '01010101', 'V': '01010110', 'W': '01010111', 'X': '01011000',
'Y': '01011001', 'Z': '01011010', 'a': '01100001', 'b': '01100010',
'c': '01100011', 'd': '01100100', 'e': '01100101', 'f': '01100110',
'g': '01100111', 'h': '01101000', 'i': '01101001', 'j': '01101010',
'k': '01101011', 'l': '01101100', 'm': '01101101', 'n': '01101110',
'o': '01101111', 'p': '01110000', 'q': '01110001', 'r': '01110010',
's': '01110011', 't': '01110100', 'u': '01110101', 'v': '01110110',
'w': '01110111', 'x': '01111000', 'y': '01111001', 'z': '01111010',
' ': '00100000', '0': '00110000', '1': '00110001', '2': '00110010',
'3': '00110011', '4': '00110100', '5': '00110101', '6': '00110110',
'7': '00110111', '8': '00111000', '9': '00111001', ',': '00101100',
';': '00111011', ':': '00111010', '.': '00101110'}
```

In [14]:
```
def coder_ascii(u):
    return coder(u, mini_ascii)
```

In [15]:
```
coder_ascii(ovide)
```

Out[15]:
```
'0100000101101110011101000110010100100000011011010110000101110010011
0010100100000011001010111010000100000011101000110010101110010011100
1
0011000010111001100100000011001010111010000100000011100010111010101
1
0111101100100001000000111010001100101011001110110100101110100001000
0
0011011110110110101101110011010010111001100100000011000110110000101
1
0010101101100011101010110110100100000011101010110111001110101011100
1
1001000000110010101110010011000010111010000100000011101000110111101
1
1010001101111001000000110111001100001011101000111010101110010011000
0
1011001010010000001110101011101010110110001110100011101010111001100
1
0000001101001011011100010000001101111011100100110001001100101001000
0
0011100010111010101100101011011010010000001100100011010010111100001
1
0010101110010011001010010000001000011011010000110000101101111011100
1
1'
```

1.6 Décoder

Comment décode-t-on ??? Nous pouvons commencer par fabriquer un dictionnaire "inverse".

In [16]:
```
def inverser(code):
    d = {}
    for a in code:
        d[code[a]] = a
    return d
```

In [17]:
```
inv_mini_ascii = inverser(mini_ascii)
print(inv_mini_ascii)
```

```
{'01000001': 'A', '01000010': 'B', '01000011': 'C', '01000100': 'D',
'01000101': 'E', '01000110': 'F', '01000111': 'G', '01001000': 'H',
'01001001': 'I', '01001010': 'J', '01001011': 'K', '01001100': 'L',
'01001101': 'M', '01001110': 'N', '01001111': 'O', '01010000': 'P',
'01010001': 'Q', '01010010': 'R', '01010011': 'S', '01010100': 'T',
'01010101': 'U', '01010110': 'V', '01010111': 'W', '01011000': 'X',
'01011001': 'Y', '01011010': 'Z', '01100001': 'a', '01100010': 'b',
'01100011': 'c', '01100100': 'd', '01100101': 'e', '01100110': 'f',
'01100111': 'g', '01101000': 'h', '01101001': 'i', '01101010': 'j',
'01101011': 'k', '01101100': 'l', '01101101': 'm', '01101110': 'n',
'01101111': 'o', '01110000': 'p', '01110001': 'q', '01110010': 'r',
'01110011': 's', '01110100': 't', '01110101': 'u', '01110110': 'v',
'01110111': 'w', '01111000': 'x', '01111001': 'y', '01111010': 'z',
'00100000': ' ', '00110000': '0', '00110001': '1', '00110010': '2',
'00110011': '3', '00110100': '4', '00110101': '5', '00110110': '6',
'00110111': '7', '00111000': '8', '00111001': '9', '00101100': ',',
'00111011': ';', '00111010': ':', '00101110': '.'}
```

Décoder le code ASCII, c'est facile. Il suffit de regrouper les 0 et les 1 par paquets de 8.

In [18]:
```
def decoder_ascii(s):
    s1 = ''
    for k in range(len(s) // 8):
        a = s[8 * k: 8 * (k + 1)]
        s1 = s1 + inv_mini_ascii[a]
    return s1
```

In [19]:
```
v = coder_ascii(ovide)
print(v)
print(decoder_ascii(v))
```

```
0100000101101110011101000110010100100000011011010110000101110010011
0101001000000110010101110100001000000111010001100101011100100111001
0110000101110011001000000110010101110100001000000111000101110101011
1111011001000010000001110100011001010110011101101001011101000010000
0110111101101101011011100110100101110011001000000110001101100001011
0101011011000111010101101101001000000111010101101110011101010111011
0010000001100101011100100110000101110100001000000111010001101111011
0100011011110010000001101110011000010111010001110101011100100110001
0110010100100000011101010111010101101100011101000111010101110011001
0000011010010110111000100000011011110111001001100010011001010010000
0111000101110101011001010110111010010000001100100011010010111100001
0101011100100110010100100000010000110110100001100001011011110111001
Ante mare et terras et quod tegit omnis caelum unus erat toto natura
e uultus in orbe quem dixere Chaos
```

Tentons un autre exemple, plus subtil.

In [20]:
```
code = {'a':'1', 'b':'110', 'c' : '10', 'd':'111'}
inv_code = inverser(code)
print(inv_code)
```

```
{'1': 'a', '110': 'b', '10': 'c', '111': 'd'}
```

Comment décoder un tel code ? Les codes des lettres n'ont pas tous la même longueur ! Tentons une approche par force brute. Pour décoder une chaîne de 0 et de 1 :

- On cherche tous les codes qui sont des préfixes de cette chaîne.
- Pour chacun de ces codes, on décode récursivement ce qui reste lorsqu'on a ôté le code du début de la chaîne.
- On recolle le tout.

La fonction `est_prefixe` prend en paramètres deux chaînes s_1 et s_2. Elle renvoie `True` lorsque s_1 est un **préfixe** de s_2, c'est à dire lorsque les premières lettres de s_2 sont celles de s_1.

In [21]:
```
def est_prefixe(s1, s2):
    n = len(s1)
    return n <= len(s2) and s2[:n] == s1
```

In [22]:
```
est_prefixe('ab', 'abc')
```

Out[22]:
```
True
```

Et voici la fonction `decoder`. Elle prend en paramètres une chaîne de 0 et de 1 et un "code inverse". Elle renvoie la liste des mots dont le code est la chaîne s. Euh, comment ça **les** mots ? Eh, oui, regardez ...

In [23]:
```
def decoder(s, inv_code):
    if s == '': return ['']
    else:
        cs = [c for c in inv_code if est_prefixe(c, s)]
        sols = []
        for c in cs:
            m = len(c)
            sols1 = decoder(s[m:], inv_code, dbg, pad + 4)
            sols = sols + [inv_code[c] + s1 for s1 in sols1]
        return sols
```

En voici une version qui affiche des informations au cours de son exécution.

In [24]:
```
def decoder(s, inv_code, dbg=False, pad=0):
    if dbg: print(pad * '.' + 'mot', s)
    if s == '':
        if dbg: print(pad * '.' + 'mot trouvé', s)
        return ['']
    else:
        cs = [c for c in inv_code if est_prefixe(c, s)]
        if dbg: print(pad * '.' + 'prefixes', *cs)
        sols = []
        for c in cs:
            m = len(c)
            sols1 = decoder(s[m:], inv_code, dbg, pad + 4)
            sols = sols + [inv_code[c] + s1 for s1 in sols1]
        return sols
```

La fonction `decoder` a une complexité terrible. Mais pas d'inquiétude, nous ne l'utiliserons que dans ce paragraphe, juste pour illustrer un gros problème dans notre définition d'un code.

In [25]:
```
faux_code = {'a':'1', 'b':'110', 'c' : '10', 'd':'111'}
inv_code = inverser(faux_code)
print(faux_code)
print(inv_code)
```

```
{'a': '1', 'b': '110', 'c': '10', 'd': '111'}
{'1': 'a', '110': 'b', '10': 'c', '111': 'd'}
```

In [26]:
```
s = 'baba'
s1 = coder(s, faux_code)
print(s1)
s2 = decoder(s1, inv_code, dbg=True)
print(s2)
```

```
11011101
mot 11011101
prefixes 1 110
....mot 1011101
....prefixes 1 10
........mot 011101
........prefixes
........mot 11101
........prefixes 1 111
............mot 1101
............prefixes 1 110
..............mot 101
..............prefixes 1 10
..................mot 01
..................prefixes
..................mot 1
..................prefixes 1
......................mot
......................mot trouvé
..............mot 1
..............prefixes 1
..................mot
..................mot trouvé
............mot 01
............prefixes
....mot 11101
....prefixes 1 111
........mot 1101
........prefixes 1 110
............mot 101
............prefixes 1 10
..............mot 01
..............prefixes
..............mot 1
..............prefixes 1
..................mot
..................mot trouvé
............mot 1
............prefixes 1
..............mot
..............mot trouvé
........mot 01
........prefixes
['acaaca', 'acaba', 'baaca', 'baba']
```

Nous avons clairement un souci. Quel est le mot d'origine ? C'est l'un des mots renvoyés par `decoder`, mais lequel ? Il est évident que nous devons changer notre définition.

Définition : Un code est une fonction $c : A \to \mathbb{B}^*$ tel que $c^* : A^* \to \mathbb{B}^*$ soit **injectif**.

Ainsi, étant donnés deux mots $u, v \in A^*$, si $c^*(u) = c^*(v)$ alors $u = v$. Maintenant, notre fonction `decoder` utilisée avec un "vrai" code renvoie une liste ayant un seul mot, le mot d'origine.

Le "code" sur lequel nous venons de travailler n'en est pas un, puisque des mots différents de A^* ont le même code. Un exemple de vrai code ? Le code ASCII, évidemment (?). Un exemple un peu moins évident ?

In [27]:
```
code = {'a':'0', 'b':'110', 'c':'10', 'd':'111'}
inv_code = inverser(code)
```

In [28]:
```
s1 = coder('babacadaa', code)
print(s1)
s2 = decoder(s1, inv_code)
print(s2)
```

```
1100110010011100
['babacadaa']
```

Exercice : Prouvez que le code ASCII est un code. Ou alors attendez un peu. Lorsque nous aurons parlé de codes préfixes ce sera évident.

2 Codes préfixes

2.1 C'est quoi ?

Soit $c : A \to \mathbb{B}^*$. Comment être certains que c est bien un code ? C'est loin d'être un problème trivial puisqu'il s'agit de vérifier que son **prolongement** $c^* : A^* \to \mathbb{B}^*$ est injectif. Mais A^* est un ensemble infini. En fait, il est très difficile de donner une condition nécessaire et suffisante "utilisable" pour que c soit un code. Il existe en revanche une condition **suffisante** intéressante. Les codes qui la vérifient sont très utilisés en pratique.

Définition : Une fonction $c : A \to \mathbb{B}^*$ est dite **préfixe** lorsque pour tous $a, a' \in A$ distincts, le mot $c(a)$ n'est pas un préfixe du mot $c(a')$.

Proposition : Une fonction préfixe est un code.

Démonstration : Supposons $c^*(a_0 \dots a_{m-1}) = c^*(b_0 \dots b_{n-1})$. Cela signifie que $c(a_0) \dots c(a_{m-1}) = c(b_0) \dots c(b_{n-1})$. Que voit-on comme lettres au début de ce mot ? Il y a les lettres du mot $c(a_0)$, mais aussi celles du mot $c(b_0)$. On en déduit que $c(a_0)$ est un préfixe de $c(b_0)$, ou le contraire (le mot le plus court est préfixe du mot le plus long). Par la propriété **préfixe**, on en déduit que $a_0 = b_0$.

Le lecteur l'aura compris, le preuve se termine par une récurrence sur la longueur des mots que je ne ferai pas. Bref, c est un code.

Exercice : Prouvez que le code ASCII est une fonction préfixe (facile, tous les codes ont la même longueur). Donc, le code ASCII est un code :-).

Le code donné un tout petit peu plus haut était un code préfixe. Mais tous les codes ne sont-ils pas forcément préfixes ? Non. Voici un exemple de code non préfixe. $c(a)$ est un préfixe de $c(b)$, et pourtant il y a bien injectivité de la fonction de codage des mots.

Exercice : Prouvez-le. En exécutant les cellules ci-dessous vous comprendrez pourquoi on a bien un code.

In [29]:
```
code = {'a':'10','b':'100','c':'110'}
inv_code = inverser(code)
```

In [30]:
```
s = 'aba'
s1 = coder(s, code)
print(s1)
s2 = decoder(s1, inv_code, True)
print(s2)
```

```
1010010
mot 1010010
prefixes 10
....mot 10010
....prefixes 10 100
........mot 010
........prefixes
........mot 10
........prefixes 10
............mot
............mot trouvé
['aba']
```

2.2 Représenter un code préfixe par un arbre

Soit $c : A \to \mathbb{B}^*$ un code préfixe. On peut coder c (mais oui, codons les codes) par un arbre binaire. Les feuilles de cet arbre sont étiquetées par les lettres de l'alphabet A. Le code de chaque lettre est obtenu comme suit : il existe un unique chemin qui va de la racine à une feuille donnée. Chaque fois que l'on descend d'un niveau, on emprunte le fils gauche ou bien le fils droit du noeud où l'on se trouve. On peut ainsi coder le chemin suivi par une liste de 0 et de 1, où 0 signifie "fils gauche" et 1 signifie "fils droit". Le code de la feuille est la concaténation de ces 0 et 1.

La fonction `faire_arbre` prend en paramètre un code préfixe. Elle renvoie l'arbre du code.

In [31]:
```
def faire_arbre(code):
    t = []
    for a in code:
        t = inserer_arbre(t, a, code[a])
    return t
```

Il est temps de nous demander comment nous allons représenter un arbre en Python.

- L'arbre vide est codé par la liste vide `[]`.
- Si l'arbre est réduit à une simple feuille contenant la lettre a, nous le représentons par la liste `['F', a]` (F comme "feuille").
- Sinon, il possède un fils gauche t_1 et un fils droit t_2. Soient L et R leurs représentations : nous représentons notre arbre par la liste `['N', L, R]` (N comme "noeud").

Les fonctions `gauche` et `droit` renvoient les fils de l'arbre t. Elles renvoient l'arbre vide si t est vide : cela nous sera pratique plus tard.

In [32]:
```
def gauche(t):
    if t == []: return []
    elif t[0] == 'F':
        raise Exception('Feuille inattendue')
    else:
        return t[1]
```

In [33]:
```
def droit(t):
    if t == []: return []
    elif t[0] == 'F':
        raise Exception('Feuille inattendue')
    else:
        return t[2]
```

La fonction `inserer_arbre` insère un caractère a de code c dans l'arbre t. Elle renvoie un nouvel arbre, qui contient une nouvelle feuille d'étiquette a.

In [34]:
```
def inserer_arbre(t, a, c):
    if c == '':
        return ['F', a]
    elif c[0] == '0':
        t1 = inserer_arbre(gauche(t), a, c[1:])
        return ['N', t1, droit(t)]
    else:
        t1 = inserer_arbre(droit(t), a, c[1:])
        return ['N', gauche(t), t1]
```

In [35]:
```
code = {'a':'0', 'b':'110', 'c':'10', 'd':'111'}
```

In [36]:
```
t = faire_arbre(code)
print(t)
```

```
['N', ['F', 'a'], ['N', ['F', 'c'], ['N', ['F', 'b'], ['F', 'd']]]]
```

Rappelez-vous le mini-code ASCII défini un peu plus_haut ... Voici son arbre.

In [37]:
```
t = faire_arbre(mini_ascii)
print(t)
```

```
['N', ['N', ['N', [], ['N', ['N', ['N', ['N', ['N', ['F', ' ']], []],
[]], []], ['N', [], ['N', ['N', ['F', ','], []], ['N', ['F', '.'], [
]]]]], ['N', ['N', ['N', ['N', ['F', '0'], ['F', '1']], ['N', ['F',
'2'], ['F', '3']]], ['N', ['N', ['F', '4'], ['F', '5']], ['N', ['F',
'6'], ['F', '7']]]], ['N', ['N', ['N', ['F', '8'], ['F', '9']], ['N'
, ['F', ':'], ['F', ';']]], []]]]], ['N', ['N', ['N', ['N', ['N', ['
N', [], ['F', 'A']], ['N', ['F', 'B'], ['F', 'C']]], ['N', ['N', ['F
', 'D'], ['F', 'E']], ['N', ['F', 'F'], ['F', 'G']]]], ['N', ['N', [
'N', ['F', 'H'], ['F', 'I']], ['N', ['F', 'J'], ['F', 'K']]], ['N',
['N', ['F', 'L'], ['F', 'M']], ['N', ['F', 'N'], ['F', 'O']]]]], ['N
', ['N', ['N', ['N', ['F', 'P'], ['F', 'Q']], ['N', ['F', 'R'], ['F'
, 'S']]], ['N', ['N', ['F', 'T'], ['F', 'U']], ['N', ['F', 'V'], ['F
', 'W']]]], ['N', ['N', ['N', ['F', 'X'], ['F', 'Y']], ['N', ['F', '
Z'], []]], []]]], ['N', ['N', ['N', ['N', ['N', [], ['F', 'a']], ['N
', ['F', 'b'], ['F', 'c']]], ['N', ['N', ['F', 'd'], ['F', 'e']], ['
N', ['F', 'f'], ['F', 'g']]]], ['N', ['N', ['N', ['F', 'h'], ['F', '
i']], ['N', ['F', 'j'], ['F', 'k']]], ['N', ['N', ['F', 'l'], ['F',
'm']], ['N', ['F', 'n'], ['F', 'o']]]]], ['N', ['N', ['N', ['N', ['F
', 'p'], ['F', 'q']], ['N', ['F', 'r'], ['F', 's']]], ['N', ['N', ['
F', 't'], ['F', 'u']], ['N', ['F', 'v'], ['F', 'w']]]], ['N', ['N',
['N', ['F', 'x'], ['F', 'y']], ['N', ['F', 'z'], []]], []]]]]], []]
```

2.3 Décoder un code préfixe

Si un code c est un code préfixe et que l'on dispose de son arbre, il devient alors très facile de décoder ! Pour décoder une chaîne s de 0 et de 1, on prend les caractères de s l'un après l'autre. Si le caractère est un 0, on descend à gauche dans l'arbre. Si c'est un 1, on descend à droite. Lorsqu'on atteint une feuille on stocke la lettre qui est à cette feuille et on repart à la racine de l'arbre.

La fonction `decoder_prefixe` a une complexité **linéaire** en la longueur de la chaîne à décoder. On peut difficilement faire mieux.

In [38]:
```
def decoder_prefixe(s, t):
    t1 = t
    s1 = ''
    k = 0
    while k < len(s):
        if t1[0] == 'F':
            s1 = s1 + t1[1]
            t1 = t
        elif s[k] == '0':
            t1 = gauche(t1)
            k = k + 1
        else:
            t1 = droit(t1)
            k = k + 1
    return s1 + t1[1]
```

Exercice : Pourquoi la fonction renvoie-t-elle `s1+t1[1]` et pas tout bêtement `s1` ?

In [39]:
```
print(code)
t = faire_arbre(code)
s = coder('ababaacabd', code)
print(s)
s1 = decoder_prefixe(s, t)
print(s1)
```

```
{'a': '0', 'b': '110', 'c': '10', 'd': '111'}
0110011000100110111
ababaacabd
```

Maintenant que comprenons ce qu'est un code préfixe et que nous savons coder et décoder, nous pouvons entrer dans le vif du sujet.

3 Compression

3.1 De quoi s'agit-il ?

Pourquoi se compliquer la vie avec des codes bizarres ? Le code ASCII n'est-il pas parfait ? Je vais expliquer où nous voulons en venir.

Soit c un code. Pour tout mot $u = a_0 \dots a_{m-1}$, le **nombre de bits** du codage de u est $\sum_{k=0}^{m-1} |c(a_k)|$ où la valeur absolue, rapppelons-le, désigne la longueur du mot.

Reformulons différemment : soient $\alpha_1, \dots, \alpha_n$ les lettres **distinctes** du mot u. L'ensemble $A = \{\alpha_1, \dots, \alpha_n\}$ est le plus petit alphabet tel que $u \in A^*$. Pour tout $\alpha \in A$, notons $\pi(\alpha)$ le **poids** de la lettre α dans le mot u, c'est à dire son nombre d'occurences. Pour tout code c sur l'alphabet A nous disposons de la quantité :

$$\mathcal{B}(c) = \sum_{k=1}^{n} \pi(\alpha_k)|c(\alpha_k)|$$

que nous exprimerons en **bits**.

Vocabulaire : La quantité $\mathcal{B}(c)$ est appelée la longueur du code c.

Problème : Trouver un code *préfixe* c tel que $\mathcal{B}(c)$ soit **minimal**.

Un tel code existe forcément. En effet, si nous notons $\mathcal{C}$ l'ensemble de tous les codes préfixes sur l'alphabet A, l'ensemble $\{\mathcal{B}(c), c \in \mathcal{C}\}$ est une partie non vide de $\mathbb{N}$, qui possède donc un plus petit élément.

Nous allons passer le reste de ce notebook à étudier un tel code : il s'agit du **code de Huffman**. Mais avant tout, un peu de Python.

3.2 Un exemple

Voici tout d'abord une fonction qui calcule **l'histogramme** d'un mot. Elle prend en paramètre un mot u et renvoie un dictionnaire qui associe à chaque lettre de u le nombre d'occurences de celle-ci dans u.

In [40]:
```
def histogramme(u):
    h = {}
    for a in u:
        if a in h: h[a] += 1
        else: h[a] = 1
    return h
```

In [41]:
```
h = histogramme(ovide)
print(h)
```

```
{'A': 1, 'n': 5, 't': 11, 'e': 13, ' ': 18, 'm': 4, 'a': 7, 'r': 7,
's': 5, 'q': 2, 'u': 9, 'o': 6, 'd': 2, 'g': 1, 'i': 4, 'c': 1, 'l':
2, 'b': 1, 'x': 1, 'C': 1, 'h': 1}
```

On y perd un peu son latin. Prenons un exemple plus simple. Abracadabra ...

In [42]:
```
h = histogramme(abra)
print(h)
```

```
{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

La fonction `longueur` prend en paramètres un histogramme h et un code c. Elle renvoie l'entier $\mathcal{B}(c)$.

In [43]:
```
def longueur(h, c):
    s = 0
    for a in h:
        s += h[a] * len(c[a])
    return s
```

Testons sur un micro-code ASCII.

In [44]:
```
c = {'a':'000', 'b':'001', 'c':'010', 'd':'011', 'r':'100'}
```

In [45]:
```
longueur(h, c)
```

Out[45]:
```
33
```

Testons maintenant sur un code préfixe c un peu plus subtil.

In [46]:
```
c = {'a':'0', 'b':'10', 'c':'1101', 'd':'1100', 'r':'111'}
```

Exercice : Vérifiez que c est un code préfixe.

In [47]:
```
longueur(h, c)
```

Out[47]:
```
23
```

La longueur du second code est plus faible. Nous venons d'effectuer une **COMPRESSION**. Si nous codons le mot "abracadabra" avec le second code, 23 bits suffisent alors qu'avec le code ASCII nous avons besoin de 33 bits. Notre but est donc d'effectuer une compression **optimale** : quel est le code préfixe associé à "abracadabra" dont la longueur est la plus petite possible ? Il s'avère que ce sera 23 ...

4 Le codage de Huffman

4.1 L'arbre de Huffman

Nous allons tout d'abord décrire l'algorithme de Huffman. Cet algorithme prend en paramètre un histogramme et crée l'arbre associé à un certain code préfixe. Comment ?

Pour fabriquer l'arbre associé à un histogramme on utilise une **file de priorité** Q. Une file de priorité est une structure de données permettant les opérations suivantes :

- Mettre dans la file un objet avec une priorité donnée (une priorité est, par exemple, un entier).
- Retirer de la file un objet de priorité minimale.

Voici notre algorithme :

1. On met dans la file Q les futures feuilles de notre arbre avec leurs poids comme priorités. Pour cela il suffit de lire l'histogramme dont les clés sont les lettres et les valeurs sont les poids.
2. Pour k allant de 0 au nombre de feuilles -2 :
   A. On retire de la file les deux arbres x et y de priorités minimales.
   B. On crée l'arbre z dont les deux fils sont x et y.
   C. On met z dans la file avec la priorité adéquate.
3. Maintenant, la file contient un unique arbre. On renvoie cet arbre.

In [48]:
```
def faire_arbre(h, dbg=False):
    Q = []
    for a in h:
        heapq.heappush(Q, (h[a], ['F', a]))
    if dbg: print('File initiale :', Q)
    for k in range(len(h) - 1):
        px, x = heapq.heappop(Q)
        py, y = heapq.heappop(Q)
        z = ['N', x, y]
        heapq.heappush(Q, (px + py, z))
        if dbg: print('Étape %d : %s' % (k, Q))
    return heapq.heappop(Q)[1]
```

In [49]:
```
t = faire_arbre(histogramme(abra), dbg=True)
print(t)
```

```
File initiale : [(1, ['F', 'c']), (1, ['F', 'd']), (2, ['F', 'r']),
(5, ['F', 'a']), (2, ['F', 'b'])]
Étape 0 : [(2, ['F', 'b']), (2, ['N', ['F', 'c'], ['F', 'd']]), (2,
['F', 'r']), (5, ['F', 'a'])]
Étape 1 : [(2, ['N', ['F', 'c'], ['F', 'd']]), (5, ['F', 'a']), (4,
['N', ['F', 'b'], ['F', 'r']])]
Étape 2 : [(5, ['F', 'a']), (6, ['N', ['N', ['F', 'c'], ['F', 'd']],
['N', ['F', 'b'], ['F', 'r']]])]
Étape 3 : [(11, ['N', ['F', 'a'], ['N', ['N', ['F', 'c'], ['F', 'd']
], ['N', ['F', 'b'], ['F', 'r']]]])]
['N', ['F', 'a'], ['N', ['N', ['F', 'c'], ['F', 'd']], ['N', ['F', '
b'], ['F', 'r']]]]
```

4.2 Créer les codes de Huffman

Une fois l'arbre construit, créer le code est une formalité. Il s'agit d'un parcours d'arbre. La fonction `faire_codes` prend en paramètre un arbre t et renvoie le code associé.

In [50]:
```
def faire_codes_aux(t, s):
    if t[0] == 'F':
        return {t[1]: ''}
    else:
        c1 = faire_codes_aux(t[1], '0')
        c2 = faire_codes_aux(t[2], '1')
        for k in c1: c1[k] = '0' + c1[k]
        for k in c2: c2[k] = '1' + c2[k]
        c1.update(c2)
        return c1

def faire_codes(t):
    return faire_codes_aux(t, None)
```

In [51]:
```
c = faire_codes(t)
print(c)
```

```
{'a': '0', 'c': '100', 'd': '101', 'b': '110', 'r': '111'}
```

Résumé : On part d'un mot u.

1. On crée un histogramme h à partir de u.
2. On crée un arbre t à partir de h.
3. On crée un code c à partir de t.

4.3 Coder un mot

Nous avons déjà écrit la fonction `coder` .

In [52]:
```
s1 = coder('abracadabra', c)
print(s1)
```

```
01101110100010101101110
```

4.4 Décoder

Nous avons aussi déjà écrit la fonction `decoder_prefixe` .

In [53]:
```
decoder_prefixe(s1, t)
```

Out[53]:
```
'abracadabra'
```

Tentons le tout avec `ovide` .

In [54]:
```
print('Mot :', ovide)
h = histogramme(ovide)
t = faire_arbre(h)
c = faire_codes(t)
s1 = coder(ovide, c)
print('Mot codé :', s1)
s2 = decoder_prefixe(s1, t)
print('Mot décodé :', s2)
```

```
Mot : Ante mare et terras et quod tegit omnis caelum unus erat toto
naturae uultus in orbe quem dixere Chaos
Mot codé : 1101100001001010011111001101010111001111000101110101001011
1011101000111111000101111101010000111011011111010100011000110000101
1011111001001011000001111111011111010100110100000110011110000010000
0011111100101110100101110100111010011111100101010010000101110101001 1
1000000110100010000001111111000001011101111011110111011010011111010100 0
1001100111101101111000011010100101110011111011010110011010011100 11
Mot décodé : Ante mare et terras et quod tegit omnis caelum unus era
t toto naturae uultus in orbe quem dixere Chaos
```

5 Optimalité du code de Huffman

5.1 Pensons arbres

Soit $u \in A^*$ où $A = \{\alpha_1, \dots, \alpha_n\}$ est l'ensemble des lettres distinctes de u, poids $\pi(\alpha_1), \dots, \pi(\alpha_n)$. À chaque code préfixe c sur l'alphabet A correspond, nous l'avons vu, un arbre T dont les feuilles sont étiquetées par les lettres de A et leurs poids. Remarquons que la longueur $|c(\alpha)|$ du code d'une lettre α est en fait la **profondeur** $d(\alpha)$ de la feuille correspondante, c'est à dire la distance de la feuille à la racine. Nous avons donc

$$\mathcal{B}(c) = \sum_{\alpha \in A} \pi(\alpha) d(\alpha)$$

et nous voyons que nous pouvons oublier le code c pour nous concentrer sur l'arbre T. Définissons donc

$$\mathcal{B}(T) = \sum_{\alpha \in A} \pi(\alpha) d(\alpha)$$

Notre problème sur des codes se ramène donc à un problème sur des arbres. On peut oublier les codes et se concentrer sur des **alphabets pondérés**, c'est à dire des alphabets dont chaque lettre est associée à un entier, son poids. Dans toute la suite, nous considérerons que nos alphabets sont pondérés.

Définition : L'alphabet A et les poids des lettres étant fixés, l'arbre T est dit **optimal** lorsque la quantité $\mathcal{B}(T)$ est minimale.

5.3 Un arbre optimal est complet

Lemme : Soit T un arbre optimal. L'arbre T est **complet** : aucun noeud de l'arbre n'a un fils vide.

Démonstration : Supposons qu'un noeud de l'arbre T a un fils vide, son fils gauche par exemple. En supprimant ce noeud et en recollant à la place le fils droit du noeud, on obtient un arbre T' dont la longueur est strictement inférieure. Impossible puisque T est optimal.

5.4 Lettres de plus petit poids

Lemme : Soient x et y les deux lettres de A de plus petit poids. Il y a un un arbre (de code) optimal pour lequel ces deux lettres sont dans des feuilles **soeurs** (i.e. qui ont le même père) de profondeur maximale.

Démonstration : Soit T un arbre optimal. Soient b et c deux feuilles soeurs de profondeur maximale (deux telles feuilles existent car notre arbre est complet). Supposons par exemple $\pi(b) \leq \pi(c)$ et $\pi(x) \leq \pi(y)$. Comme x et y ont les deux plus petits poids, on a $\pi(x) \leq \pi(b)$ et $\pi(y) \leq \pi(c)$. De plus, b et c sont à une profondeur maximale dans l'arbre T, donc $d(x) \leq d(b)$ et $d(y) \leq d(c)$. Échangeons x et b. On obtient un nouvel arbre T' tel que

$$\mathcal{B}(T') = \mathcal{B}(T) - \pi(x)d(x) - \pi(b)d(b) + \pi(x)d(b) + \pi(b)d(x) = \mathcal{B}(T) - (\pi(b) - \pi(x))(d(b$$

Mais l'arbre T est optimal, donc $\mathcal{B}(T') = \mathcal{B}(T)$. Ainsi, T' est aussi optimal. On fait de même en échangeant c et y et on obtient un nouvel arbre T'', toujours optimal, qui vérifie la condition voulue.

5.5 Optimalité du code de Huffman

Théorème : L'algorithme de Huffman produit un arbre optimal.

Démonstration : Faisons une récurrence sur le nombre n de lettres de l'alphabet A.

Si $n = 2$, l'arbre renvoyé par l'algorithme de Huffman est clairement optimal. Il a deux feuilles à la profondeur 1 et on peut difficilement faire mieux :-).

Soit maintenant $n \geq 3$. Supposons la propriété d'optimalité vraie pour un alphabet A ayant $n - 1$ lettres. Regardez le code de la fonction `faire_arbre`. À l'itération $k = 0$, la fonction prend les lettres x et y de plus petits poids et les accole pour former un arbre z. Puis elle met z dans la file de priorité Q qui contient maintenant 1 arbre de moins. Les $n - 1$ dernières itérations ne sont autres que l'algorithme de Huffman appliqué à l'alphabet $A' = A \cup \{z\} \setminus \{x, y\}$, où $\pi(z) = \pi(x) + \pi(y)$ et les poids des autres lettres sont les mêmes pour A et A'. Par l'hypothèse de récurrence, l'algorithme renvoie donc un arbre H optimal pour l'alphabet A'. Il nous reste à voir que cet arbre est optimal pour l'alphabet A.

Alourdissons un peu les notations pour préciser à chaque fois sur quel alphabet on fait les calculs.

On a $\mathcal{B}_{A'}(H) = \mathcal{B}_A(H) - \pi(x)d(x) - \pi(y)d(y) + \pi(z)d(z)$. Mais $d(z) = d(x) - 1 = d(y) - 1$. De là, $\mathcal{B}_{A'}(H) = \mathcal{B}_A(H) - (\pi(x) + \pi(y))$.

Soit maintenant T un arbre optimal pour l'alphabet A avec les lettres x et y sur des feuilles soeurs de profondeurs maximales (ceci est possible par le lemme vu plus haut). T peut également être vu comme un arbre sur l'alphabet A'. Or H est optimal pour A'. Donc $\mathcal{B}_{A'}(T) \geq \mathcal{B}_{A'}(H)$. Mais $\mathcal{B}_{A'}(T) = \mathcal{B}_A(T) - (\pi(x) + \pi(y))$ (mêmes calculs que ceux faits pour H), donc $\mathcal{B}_A(T) - (\pi(x) + \pi(y)) \geq \mathcal{B}_{A'}(H) = \mathcal{B}_A(H) - (\pi(x) + \pi(y))$. On en déduit que $\mathcal{B}_A(T) \geq \mathcal{B}_A(H)$. Comme T est optimal pour A, on a en fait égalité : $\mathcal{B}_A(H) = \mathcal{B}_A(T)$ et H est optimal pour l'alphabet A.

Exercice : Qu'arrive-t-il si l'alphabet A n'a qu'une lettre ?

6 Un exemple grandeur nature

Coder des mots de 10 lettres c'est rigolo, mais comment se comporte l'algorithme de Huffman dans la vie réelle ? J'ai choisi au hasard de coder l'Énéide de Virgile. Le texte intégral se trouve dans le fichier `eneide.txt`, qui fait environ 400 kilo-octets. Ce n'est pas gigantesque, mais bon, nous programmons en Python alors soyons raisonnables.

6.1 Lire le texte de Virgile

In [55]:
```
f = open('eneide.txt')
virgile = f.read()
f.close()
print(len(virgile))
```

```
465773
```

La chaîne `virgile` contient maintenant tout le texte de Virgile. Affichons les 1000 premiers caractères.

In [56]:
```
print(virgile[:1000])
```

```
PUBLI VERGILI MARONIS

AENEIDOS


  LIBER I


  ARMA virumque cano, Troiae qui primus ab oris
  Italiam, fato profugus, Laviniaque venit
  litora, multum ille et terris iactatus et alto
  vi superum saevae memorem Iunonis ob iram;
  multa quoque et bello passus, dum conderet urbem,
  inferretque deos Latio, genus unde Latinum,
  Albanique patres, atque altae moenia Romae.

  Musa, mihi causas memora, quo numine laeso,
  quidve dolens, regina deum tot volvere casus
  insignem pietate virum, tot adire labores
  impulerit.  Tantaene animis caelestibus irae?

  Urbs antiqua fuit, Tyrii tenuere coloni,
  Karthago, Italiam contra Tiberinaque longe
  ostia, dives opum studiisque asperrima belli;
  quam Iuno fertur terris magis omnibus unam
  posthabita coluisse Samo; hic illius arma,
  hic currus fuit; hoc regnum dea gentibus esse,
  si qua fata sinant, iam tum tenditque fovetque.
  Progeniem sed enim Troiano a sanguine duci
  audierat, Tyrias olim quae verteret arces;
  hinc populum late reg
```

6.2 L'histogramme

Calculons l'histogramme de `virgile` .

In [57]:
```
h = histogramme(virgile)
print(h)
```

```
{'P': 611, 'U': 108, 'B': 92, 'L': 405, 'I': 907, ' ': 74631, 'V': 2
16, 'E': 417, 'R': 208, 'G': 113, 'M': 339, 'A': 1245, 'O': 258, 'N'
: 461, 'S': 538, '\n': 10371, 'D': 432, 'v': 5414, 'i': 34878, 'r':
25631, 'u': 29655, 'm': 18584, 'q': 6395, 'e': 44060, 'c': 14142, 'a
': 34569, 'n': 21830, 'o': 18270, ',': 6847, 'T': 1079, 'p': 9124, '
s': 27233, 'b': 5065, 't': 29474, 'l': 11690, 'f': 3812, 'g': 4727,
';': 937, 'd': 8996, '.': 2818, 'h': 2715, '?': 367, 'y': 706, 'K':
10, 'x': 1758, ':': 871, '-': 264, 'H': 520, '!': 98, "'": 835, 'Q':
278, 'F': 138, 'z': 26, 'Z': 9, 'C': 431, '(': 48, ')': 46, '[': 29,
']': 29, 'X': 13}
```

6.3 L'arbre de Huffman

Fabriquons l'arbre de Huffman.

In [58]:
```
t = faire_arbre(h)
print(t)
```

```
['N', ['N', ['N', ['N', ['N', ['F', '\n'], ['N', ['F', 'b'], ['F', '
v']]], ['F', 'n']], ['F', 'e']], ['N', ['N', ['N', ['F', 'l'], ['N',
['N', ['F', 'h'], ['F', '.']], ['F', 'q']]], ['F', 'r']], ['N', ['F'
, 's'], ['N', ['N', ['N', ['N', ['N', ['F', 'y'], ['N', ['F', 'M'],
['F', '?']]], ['N', ['N', ['F', 'L'], ['N', ['N', ['F', '!'], ['F',
'U']], ['F', 'R']]], ['F', "'"]]], ['N', ['N', ['N', ['F', 'E'], ['F
', 'C']], ['F', ':']], ['F', 'x']]], ['F', ',']], ['F', 'c']]]]], ['
N', ['N', ['N', ['F', 't'], ['F', 'u']], ['N', ['F', 'a'], ['N', ['N
', ['N', ['N', ['N', ['N', ['F', 'D'], ['N', ['F', 'V'], ['N', ['F',
'G'], ['N', ['N', ['F', 'z'], ['F', '[']], ['N', ['F', ']'], ['N', [
'F', 'X'], ['N', ['F', 'Z'], ['F', 'K']]]]]]], ['F', 'I']], ['N', [
'F', ';'], ['N', ['F', 'N'], ['F', 'H']]]], ['F', 'f']], ['F', 'd']]
, ['F', 'o']]]], ['N', ['N', ['F', 'i'], ['N', ['N', ['F', 'p'], ['N
', ['N', ['N', ['N', ['N', ['F', 'O'], ['F', '-']], ['F', 'S']], ['F
', 'T']], ['N', ['N', ['N', ['F', 'Q'], ['N', ['F', 'F'], ['N', ['F'
, 'B'], ['N', ['F', ')'], ['F', '(']]]]], ['F', 'P']], ['F', 'A']]],
['F', 'g']]], ['F', 'm']]], ['F', ' ']]]]
```

6.4 Le code de Huffman

Puis les codes de Huffman.

In [59]:
```
c = faire_codes(t)
print(c)
```

```
{'\n': '00000', 'b': '000010', 'v': '000011', 'n': '0001', 'e': '001
', 'l': '01000', 'h': '0100100', '.': '0100101', 'q': '010011', 'r':
'0101', 's': '0110', 'y': '011100000', 'M': '0111000010', '?': '0111
000011', 'L': '0111000100', '!': '011100010100', 'U': '011100010101'
, 'R': '01110001011', "'": '011100011', 'E': '0111001000', 'C': '011
1001001', ':': '011100101', 'x': '01110011', ',': '011101', 'c': '01
111', 't': '1000', 'u': '1001', 'a': '1010', 'D': '1011000000', 'V':
'10110000010', 'G': '101100000110', 'z': '10110000011100', '[': '101
10000011101', ']': '10110000011110', 'X': '101100000111110', 'Z': '1
0110000001111110', 'K': '10110000011111111', 'I': '101100001', ';': '1
01100010', 'N': '1011000110', 'H': '1011000111', 'f': '1011001', 'd'
: '101101', 'o': '10111', 'i': '1100', 'p': '110100', 'O': '11010100
000', '-': '11010100001', 'S': '1101010001', 'T': '110101001', 'Q':
'11010101000', 'F': '110101010010', 'B': '1101010100110', ')': '1101
0101001110', '(': '11010101001111', 'P': '1101010101', 'A': '1101010
11', 'g': '1101011', 'm': '11011', ' ': '111'}
```

6.5 Codons

In [60]:
```
s1 = coder(virgile, c)
print(s1[:2000])
```

```
11010101010111000101011101010100110011100010010110000111110110000010
01110010000111000101110110000011010110000101110001001011000011110111
00001011010101101110001011110101000001011000110101100001110101000100
00000000110101011011100100010110001100111001000101100001101100000011
01010000011010100010000000000000001111110111000100101100001110101010
01100111001000011100010111111011000010000000000000001111111101010110
11100010110111000010110101010111110000111100010110011101101001110010 01
11101111101000011011101110111111101010010101101111100101000111101001 1
10011100111110100010111001101110010110111101000001011110111010111000
11000000111111101100001100010100100011001010110111011110111111011001101
01000101111111101000101101111011001100111010111001011001110111101110
00100101000001111000001110010100100111001001111000011001000111001000
00000111111010001100100010111010110100111011111101110010100010001001
11011111110001000010000011110011000111100000101010101110001101111100
10100111110001010100010010110111001100011110100100010001011100000111
11100001111001110110100111010000101011001110111111011010100010000111 0
10001111110110011101110111010100111011111101100001100100011011100011
10001101111011100001011111000101101011011101100010000001111111101110
01010001000101011101001110011011101001110010011110011000111000010001
01000010001011111111010010100110011010010110011101111101101100111011
11101111101110001101101001010100110001111001010100001000111011011101
00000111111110000011011001001010101010011000010011100100111110110100
11011101101110111000100101010001100101110111011111101011001000110010
11011110010001101101001111011100010010101000110000011001110110111010
00001111111110101011010000000101010000111000100111001001111110100101 0
10000101001011001110111110101000010011100100111110100100010001010001
11111011101110010001110010101110111000101110111111011101000101001010 0
00000000111111011100001010010110101001110111111011110001001001100111
01111010100101101010011011111011001110111011101011010011101111010 01
11001101111110001100111011111
```

Évidemment, Virgile est beaucoup moins lisible comme ça.

6.6 Taux de compression

Définition : Soit c un code. Soit u un mot. Le taux de compression de u par le code c est

$$\tau(u, c) = 1 - \frac{|c(u)|}{|u|}$$

où toutes les longueurs sont exprimées en bits. Plus le taux de compression est élevé, plus le code est efficace. Certains auteurs appellent taux de compression la quantité $1 - \tau$ ...

Quel est le taux de compression obtenu dans notre exemple ? Divisons la longueur de s_1 (qui contient des bits) par 8 fois la longueur de s. Pourquoi 8 fois ? Parce que chaque lettre de la chaîne s serait codée avec le code ASCII sur 8 bits.

In [61]:
```
print(1 - len(s1) / (8 * len(virgile)))
```

```
0.45638299128545445
```

C'est d'après la théorie, le mieux que l'on peut faire avec un code préfixe. Et avec un code quelconque ? Ceci est une tout autre histoire ...

6.7 Décodons

La chaîne codée par l'algorithme de Huffman a une longueur 43% plus courte qu'en codant par le code ASCII. Mais peut-être nous sommes nous trompés ? Essayons de décoder !

In [62]:
```
s2 = decoder_prefixe(s1, t)
print(s2[10000:11000])
```

```
 accingunt, dapibusque futuris;
  tergora deripiunt costis et viscera nudant;
  pars in frusta secant veribusque trementia figunt;
  litore aena locant alii, flammasque ministrant.
  Tum victu revocant vires, fusique per herbam
  implentur veteris Bacchi pinguisque ferinae.
  Postquam exempta fames epulis mensaeque remotae,
  amissos longo socios sermone requirunt,
  spemque metumque inter dubii, seu vivere credant,
  sive extrema pati nec iam exaudire vocatos.
  Praecipue pius Aeneas nunc acris Oronti,
  nunc Amyci casum gemit et crudelia secum
  fata Lyci, fortemque Gyan, fortemque Cloanthum.

  Et iam finis erat, cum Iuppiter aethere summo
  despiciens mare velivolum terrasque iacentis
  litoraque et latos populos, sic vertice caeli
  constitit, et Libyae defixit lumina regnis.
  Atque illum talis iactantem pectore curas
  tristior et lacrimis oculos suffusa nitentis
  adloquitur Venus:  'O qui res hominumque deumque
  aeternis regis imperiis, et fulmine terres,
  quid meus Aeneas i
```

Petit temps d'attente pour avoir le résultat. Pour tout dire, notre fonction de décodage n'est pas très efficace. Elle fait beaucoup de recopies de données (exemple : `t1 = gauche(t1)`) et ces données peuvent être de grande taille. Une implémentation propre utiliserait des pointeurs sur les données. J'imagine (un peu en l'air) qu'un code proprement écrit en C pourrait être mille fois plus rapide.

6.8 Sommes-nous bien certains ???

Bon, histoire d'être bien certains :

In [63]:
```
print(virgile == s2)
```

```
True
```

À moins que Huffman ne fonctionne que pour Virgile, nous pouvons être raisonnablement confiants dans nos fonctions Python :-).

Exercice :

- Quel est le taux de compression obtenu pour Voyage au Centre de la Terre, de Jules Verne ? Je vous conseillle ce site (http://www.gutenberg.org).
- Quel est le taux de compression obtenu pour le notebook `Huffman.ipynb` ?
- Quel est le taux de compression obtenu pour une image `JPG` de la Tour Eiffel (soyez raisonnables sur la taille de l'image, évitez une image de 6 Mo)?

7 Compléments

7.1 Où est le piège ?

Au lieu de coder lettre par lettre, regroupons les lettres k par k, où $k \geq 1$. Voici notre nouvelle fonction `histogramme2` .

In [64]:
```
def histogramme2(s, k):
    h = {}
    n = len(s)
    for i in range(n // k):
        a = s[k * i : k * (i + 1)]
        if a in h: h[a] += 1
        else: h[a] = 1
    if n % k != 0:
        a = s[k * (n // k):]
        h[a] = 1
    return h
```

In [65]:
```
h = histogramme2(abra, 2)
print(h)
```

```
{'ab': 1, 'ra': 1, 'ca': 1, 'da': 1, 'br': 1, 'a': 1}
```

Il nous faut également réécrire la fonction `coder` .

In [66]:
```
def coder2(s, code, k):
    s1 = ''
    n = len(s)
    for i in range(n // k):
        a = s[k * i: k * (i + 1)]
        s1 = s1 + code[a]
    if n % k != 0:
        a = s[k * (n // k):]
        s1 = s1 + code[a]
    return s1
```

Reprenons notre épopée et lançons un compression avec $k = 2$.

In [67]:
```
K = 2
h = histogramme2(virgile, K)
print(len(h))
t = faire_arbre(h)
#print(t)
c = faire_codes(t)
s1 = coder2(virgile, c, K)
s2 = decoder_prefixe(s1, t)
print(virgile == s2)
```

```
851
True
```

Quel est le taux de compression ?

In [68]:
```
print(1 - len(s1) / (8 * len(virgile)))
```

```
0.5097343555766436
```

Nous avons donc amélioré le taux de compression. Euh, mais Huffman n'est-il pas optimal ? Oui, **pour un alphabet donné**. Mais en groupant les lettres deux par deux nous travaillons avec un mot sur un alphabet qui est l'ensemble des mots de deux lettres. Une idée nous vient alors à l'esprit : et si nous prenions $k = 3$? $k = 4$? $k = 16$? Vous pouvez essayer en changeant ci-dessus la valeur de K. On constate que le taux de compression augmente, augmente ...

Allez, soyons fous. Et si nous prenions $k = n - 1$ où n est la longueur du mot à coder ? Alors l'arbre de Huffman a deux feuilles, étiquetées par

- Le mot u privé de sa dernière lettre, de poids 1
- La dernière lettre de u, de poids 1.

Et le mot u est codé par '01', c'est à dire avec 2 bits. QUEL QUE SOIT u.

Alors où est le piège ?

Si je veux stocker ou transmettre le mot u, je le code avec l'algorithme de Huffman ce qui aura pour effet de le compresser. Mais si je veux pouvoir le décoder un jour ... il va me falloir aussi stocker l'arbre de Huffman ! Ainsi, si c est le code de Huffman associé à u, je stockerai (par exemple) le couple $(c^*(u), T)$ où T est l'arbre associé au code T. Le taux de compression n'est donc pas $\tau = 1 - \frac{|c^*(u)|}{|u|}$ mais $\tau' = 1 - \frac{|c^*(u)|+|T|}{|u|} = \tau - \frac{|T|}{|u|}$ où $|T|$ est la "taille de T" qui resterait à préciser. Le taux de compression est donc plus faible que prévu.

Bon, et alors ? Nous avons menti tout au long du notebook ? Oui et non. Lorsque la taille de u tend vers l'infini, τ' tend vers τ car pour un alphabet fixé la taille de T, elle, reste bornée. Je n'entre pas plus avant dans les détails mais il vaudrait mieux dire que l'algorithme de Huffman est asymptotiquement optimal. Et, toujours sans détailler, oui, il est vrai qu'en regroupant les lettres par paquets de k on obtient des taux de compression asymptotiquement meilleurs en augmentant la valeur de k.

7.2 Huffman et Shannon

Soit A un alphabet pondéré. Pour $\alpha \in A$, soit $P(\alpha) = \frac{\pi(\alpha)}{N}$ où N est la somme de tous les poids des lettres. P définit une probabilité sur A. Appelons **entropie** de A la quantité :

$$\mathcal{H}(A) = -\sum_{\alpha \in A} P(\alpha) \lg P(\alpha)$$

où $\lg$ désigne le logarithme en base 2. Pour tout mot u, l'entropie de u est alors l'entropie de l'alphabet pondéré associé à u.

Le nombre $\mathcal{H}(u)$ est la **quantité d'information** contenue dans u (ou A). Il est mesuré en **bits**. La justification de cette appellation nécessiterait un exposé à part entière et nous admettrons qu'elle est cohérente.

In [69]:
```
def entropie(u):
    n = len(u)
    h = histogramme(u)
    s = 0
    for a in h:
        P = h[a] / n
        s = s - P * math.log(P, 2)
    return s
```

Prenons l'exemple de notre épopée :

In [70]:
```
entropie(virgile)
```

Out[70]:
```
4.318174908401346
```

L'énéide a une entropie de 4.318. Virgile s'en doutait-il ?

Théorème du codage sans bruit (Shannon) : Soit u un mot. Soit A l'alphabet pondéré associé à u. Soit enfin c un code préfixe optimal pour u. Alors :

$$\mathcal{H}(u) \leq \sum_{\alpha \in A} P(\alpha)|c(\alpha)| < \mathcal{H}(u) + 1$$

Ce notebook étant déjà d'une longueur terrible, je ne prouverai pas ce théorème. Mais c'est quoi $\sum_{\alpha \in A} P(\alpha)|c(\alpha)|$? Ce n'est rien d'autre que $\frac{1}{|u|} \sum_{\alpha \in A} \pi(\alpha)|c(\alpha)| = \frac{|c^*(u)|}{|u|} = \frac{\mathcal{B}(c)}{|u|}$, quantité fortement liée à ce que nous avons appelé le taux de compression.. Que nous raconte le théorème du codage sans bruit ? Il nous dit que

- $\mathcal{H}(u)$ est une borne inférieure infranchissable pour tout codage préfixe de u.
- Un codage préfixe optimal est égal à cette borne inférieure, à 1 près.

Exercice : Soit A un alphabet ayant n lettres de poids tous égaux. Montrer que $\mathcal{H}(A) = \lg n$. Que vaut $\mathcal{H}(A)$ si $n = 256$? Nous venons de montrer que la quantité d'information d'un octet aléatoire est égale à 8 bits :-).

Exercice : Que vaut $\sum_{\alpha \in A} P(\alpha)|c(\alpha)|$ si c est le code ASCII ? En déduire en utilisant le théorème de Shannon que pour tout alphabet A ayant 256 lettres (de poids quelconques), $\mathcal{H}(A) \leq 8$. Le code ASCII possède donc le pire taux imaginable. Mais ne lui jetons pas la pierre parce qu'il peut coder TOUS les mots !

Exercice : Seriez-vous capables de généraliser le résultat précédent, sans utiliser le théorème de Shannon, pour un alphabet à n lettres avec $n \geq 2$ quelconque ? Indication : la fonction $\lg$ est une fonction concave. Elle vérifie donc des inégalités bien connues.

In [71]:
```
def taux(u, code):
    n = len(u)
    h = histogramme(u)
    s = 0
    for a in h:
        P = h[a] / n
        s = s + P * len(code[a])
    return s
```

In [72]:
```
h = histogramme(virgile)
t = faire_arbre(h)
c = faire_codes(t)
print(taux(virgile, c))
#s1 = coder(s, c)
#print(len(s1) / len(s))
```

```
4.3489360697163635
```

Exercice :

- Quelle est l'entropie de Voyage au Centre de la Terre ?
- Quelle est l'entropie du notebook `Huffman.ipynb` ?
- Quelle est l'entropie d'une image `JPG` de la Tour Eiffel ?

7.3 On n'est pas à 1 bit près ...

Le 1 dans le majorant du théorème de Shannon n'a pas l'air bien méchant. Pour l'Énéide l'entropie est d'environ 4.32 alors que le taux est 4.35. Maintenant, imaginons une entropie très faible, 0.01 par exemple. Et un taux égal à 1. Le taux serait alors égal à 100 fois l'entropie ! Quand cela arrive-t-il ? Prenons un alphabet $A = \{a, b\}$ de deux lettres, avec $\pi(a) = N \in \mathbb{N}^*$ et $\pi(b) = 1$. On a

$$\mathcal{H}(A) = -\frac{N}{N+1}\lg(\frac{N}{N+1}) - \frac{1}{N+1}\lg(\frac{1}{N+1}) = \frac{N}{N+1}\lg(1+\frac{1}{N}) + \frac{1}{N+1}\lg(N+1)$$

Le code de Huffman associé à cet alphabet code la lettre a par '1' et la lettre b par '0' (ou le contraire si $N = 1$). Le taux associé est

$$\tau = 1$$

On a $\mathcal{H}(A) \to 0$ lorsque N tend vers l'infini, alors que τ reste constant égal à 1. Ainsi, $\frac{\tau}{\mathcal{H}(A)} \to +\infty$ lorsque N tend vers l'infini, ce qui est très mauvais.

Ainsi 1 bit fait **TOUTE** la différence ...

7.4 Et après ?

Si vous compressez le fichier `eneide.txt` au format `zip` vous obtiendrez un taux de compression d'environ 0.6. C'est en tout cas ce que j'obtiens sur ma machine. C'est mieux que Huffman. Alors, Huffman n'est pas optimal ?

S'il faut retenir **UNE** chose de ce notebook c'est que "être optimal" est une expression qui n'a aucun sens en soi. On est toujours optimal **dans un certain contexte**. Nous avons prouvé des résultats très précis sur le code de Huffman, en l'occurence que pour un alphabet **fixé** avec des poids de lettres **fixés** le code de Huffman est optimal parmi les codes **préfixes**. Cela fait beaucoup de conditions ... Je n'irai pas plus avnt, la théorie de la compression est vaste et riche, nous n'avons fait que lever un coin du voile.

Exercice : Lisez l'Énéide.

In [ ]: