Data compression

In order to send information as fast as possible across the Internet, web servers first represent the data in a compact format (**compression**) before it is actually sent over the network. A web browser that receives this compressed data therefore needs to unpack it first (**decompression**) before the information can be used. To understand how data compression can be achieved, you need to know first that all data (text, images, sound, video, …) are represented inside a computer as **bit strings** — strings of consecutive zeros and ones. Textual data, for example, is traditionally encoded by representing each symbol (character) with a fix-length bit string. If we represent all 8 symbols of the word internet by a single **byte** (8 bits), the computer representation of this word takes 64 bits in total.

While he was a PhD student at MIT, David A. Huffman developed an algorithm that represents each symbol as a unique variable-length bit string. The details of this algorithm are not relevant for this assignment, but the gain for data **compression** lies in the fact that more common symbols are generally represented using fewer bits than less common symbols. The table below shows an example on how a list of symbols is converted to their corresponding bit string representation. From this you can derive that the letter i is represented by the bit string 1000, the letter e by the bit string 000, the letter x by the bit string 10010 and a space by the bit string 111. A full text is compressed by concatenating the bit string representations of the consecutive symbols in the text. According to the scheme of the table below, the word internet is compressed as the bit string 10000010011000011000001000000110 (31 bits).

symbol	bit string	symbol	bit string
space	111	s	1011
a	010	t	0110
e	000	l	11001
f	1101	o	00110
h	1010	p	10011
i	1000	r	11000
m	0111	u	00111
n	0010	x	10010

Decompressing a fragment of compressed data relies on an ingenious feature of the bit strings that are assigned by Huffman's algorithm. Of all possible symbols in the coding scheme, there is always exactly one symbol whose corresponding bit string is a prefix (a string at the start of another string) of the remaining portion of the compressed data.

Suppose that we wish to decompress the compressed bit string 10000010011000011000001000000110. We observe in the table above that the symbol i is the only one whose corresponding bit string (1000) is a prefix of the compressed bit string. This gives us the first symbol, and we can reduce the compressed bit string to 00100110000110000010000110 by removing the corresponding prefix. Again, we find that the symbol n is the only one whose corresponding bit string (0010) is a prefix of the remaining compressed bit string, so we found the second symbol. By repeatedly finding prefixes and reducing the compressed bit string, we finally end up decompressing the entire bit string.

Assignment

For this assignment you are asked to compress strings into bit strings, and decompress compressed bit string into the original strings according to the principle of Huffman coding. Bit strings are represented as strings that only consist of the characters `0` and `1`. The bit strings that are used to compress the individual symbols are given in a text file, whose lines consist of a symbol (a single character), followed by a tab and the bit string used to represent this symbol. Below, you find the text file containing the bit strings that correspond to the symbols in the table above. Note that the symbol on the first line of this text file is a space.

```
  111
a 010
e 000
f 1101
h 1010
i 1000
m 0111
n 0010
s 1011
t 0110
l 11001
o 00110
p 10011
r 11000
u 00111
x 10010
```

Define a class `ZIP` that can be used to compress strings as bit strings, and decompress bit strings into the original strings according to the principle of Huffman coding. Upon initialization of an object of this class, the location of a text file must be passed. This file must contain the encoding of the individual symbols in the format as outlined above. In addition, the objects of the class `ZIP` must support at least the following methods:

- A method `symbol2bitstring` that takes a symbol (a string containing a single character). The method must return the compressed bit string that corresponds to the given symbol. In case the given symbol does not occur in the file that was passed upon initialization, the method must raise an `AssertionError` with the message `unknown symbol "`*x*`"`, with *x* being the given symbol.
- A method `bitstring2symbol` that takes a bit string. The method must return the symbol that corresponds to the given bit string. In case the given bit string does not occur in the file that was passed upon initialization, the method must raise an `AssertionError` with the message `invalid bitstring`.
- A method `compress` that takes a string. The method must return the compressed bit string that corresponds to the given string. In case the given string contains characters whose bit string encoding is not contained in the file that was passed upon initialization, the method must raise an `AssertionError` with the message `unknown symbol "`*x*`"`, with *x* being the leftmost symbol that does not occur in the file.
- A method `decompress` that takes a compressed bit string. The method must return the original string that corresponds to the given bit string according to the Huffman coding scheme. In case the given bit string is not valid according to the Huffman coding scheme, the method must raise an `AssertionError` with the message `invalid bitstring`.

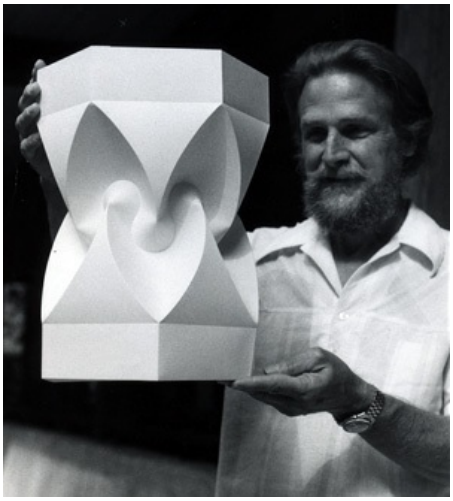
Example

In the following interactive session we assume that the text file codes.txt is located in the current directory.

```
>>> zip = ZIP('codes.txt')

>>> zip.symbol2bitstring('i')
'1000'
>>> zip.symbol2bitstring('e')
'000'
>>> zip.symbol2bitstring('T')
Traceback (most recent call last):
AssertionError: unknown symbol "T"

>>> zip.bitstring2symbol('1000')
'i'
>>> zip.bitstring2symbol('000')
'e'
>>> zip.bitstring2symbol('01')
Traceback (most recent call last):
AssertionError: invalid bitstring

>>> zip.compress('internet')
'10000010011000011000001000001l0'
>>> len(zip.compress('internet'))
31
>>> zip.compress('internet explorer')
'10000010011000011000001000001101110001001010011110010011011000000011000'
>>> zip.compress('mozilla firefox')
Traceback (most recent call last):
AssertionError: unknown symbol "z"

>>> zip.decompress('10000010011000011000001000001l0')
'internet'
>>> zip.decompress('10000010011000011000001000001101110001001010011110010011011000000011000')
'internet explorer'
>>> zip.decompress('1000001001100001100000l000001101')
Traceback (most recent call last):
AssertionError: invalid bitstring
>>> zip.decompress('10000010011000011000000000110')
Traceback (most recent call last):
AssertionError: invalid bitstring
```

Epilogue

When David Huffman died in 1999, the world lost a talented computer scientist — Huffman was best known for discovering the Huffman coding technique used in data compression. But it also lost a pioneer in **mathematical origami**, an extension of the traditional art of paper folding that applies computational geometry, number theory, coding theory, and linear algebra. The field today is finding wide application, helping researchers to fold everything from proteins to automobile airbags and space-based telescopes.



David Huffman, with one of his paper foldings, in 1978.

Huffman was drawn to the work through his investigations into the mathematical properties of "zero curvature" surfaces, studying how paper behaves near creases and apices of cones. During the last two decades of his life he created hundreds of beautiful, perplexing paper models in which the creases were curved rather than straight.

But he kept his folding research largely to himself. He published only one paper on the subject, and much of what he discovered was lost at his death. "He anticipated a great deal of what other people have since rediscovered or are only now discovering," laser physicist Robert Lang told the New York Times in 2004. "At least half of what he did is unlike anything I've seen." MIT computer scientist Erik Demaine is working now with Huffman's family to recover and document his discoveries.

"I don't claim to be an artist. I'm not even sure how to define art," Huffman told an audience in 1979. "But I find it natural that the elegant mathematical theorems associated with paper surfaces should lead to visual elegance as well."

Om gegevens zo snel mogelijk over het Internet te verzenden, worden ze door een webserver eerst zo compact mogelijk voorgesteld (**compressie**) voordat ze over het netwerk verstuurd worden. Een browser die de gecomprimeerde gegevens ontvangt, moet ze daardoor eerst uitpakken (**decompressie**) voordat ze kunnen gebruikt worden. Om te begrijpen hoe men tot een dergelijke datacompressie kan komen, moet je eerst weten dat alle gegevens (tekst, afbeeldingen, geluid, video, …) binnen een computer worden voorgesteld als **bitstrings** — strings van opeenvolgende nullen en enen. Bijvoorbeeld, voor tekstuele data wordt elk symbool (karakter) traditioneel voorgesteld door een bitstring met een vast aantal bits. Als we op die manier elk van de 8 symbolen van het woord internet voorstellen door één enkele **byte** (8 bits), dan gebruikt de computervoorstelling van dit woord in totaal 64 bits.

Tijdens zijn doctoraat aan MIT ontwikkelde David A. Huffman een algoritme dat elk symbool voorstelt door een unieke bitstring, waarbij het aantal bits kan verschillen van symbool tot symbool. De details van dit algoritme zijn voor deze opgave niet belangrijk, maar de winst bij **compressie** zit hem in het feit dat symbolen die vaker voorkomen, worden voorgesteld door kortere bitstrings dan symbolen die minder vaak voorkomen. Hieronder zie je bijvoorbeeld een tabel die aangeeft hoe symbolen kunnen omgezet worden naar hun corresponderende bitstringvoorstelling. Daaruit kan je afleiden dat de letter i wordt voorgesteld door de bitstring 1000, de letter e door de bitstring 000, de letter x door de bitstring 10010 en een spatie door de bitstring 111. Een volledige tekst wordt gecomprimeerd door de bitstrings van de opeenvolgende

symbolen achter elkaar te zetten. Het woord internet wordt volgens de codering uit onderstaande tabel dus voorgesteld door de bitstring 10000010011000011000001000001110 (31 bits).

symbool	bitstring	symbool	bitstring
spatie	111	s	1011
a	010	t	0110
e	000	l	11001
f	1101	o	00110
h	1010	p	10011
i	1000	r	11000
m	0111	u	00111
n	0010	x	10010

Voor het **decomprimeren** van gecomprimeerde gegevens moeten we rekenen op een ingenieuze eigenschap van de bitstrings die door het algoritme van Huffman worden vastgelegd. Het is namelijk zo dat van alle mogelijke symbolen in het codeerschema, er altijd **juist één** symbool is waarvan de corresponderende bitstring een prefix (string aan het begin van een andere string) is van het resterende deel van de gecomprimeerde gegevens.

Stel bijvoorbeeld dat we de gecomprimeerde bitstring 10000010011000011000001000001110 willen decomprimeren. Dan stellen we vast dat het symbool i het enige is uit bovenstaande tabel waarvan de corresponderende bitstring (1000) een prefix vormt van de gecomprimeerde bitstring. Daarmee kennen we dus het eerste symbool, en kunnen we de gecomprimeerde bitstring herleiden tot 00100110000110000010000001110 door de corresponderende prefix te verwijderen. Opnieuw kunnen we vaststellen dat het symbool n het enige symbool is waarvan de corresponderende bitstring (0010) een prefix vormt van de resterende gecomprimeerde bitstring, waardoor we het tweede symbool gevonden hebben. Door op deze manier prefixen te zoeken en de gecomprimeerde bitstring te reduceren, kunnen we uiteindelijk de volledige bitstring decomprimeren.

Opgave

In deze opgave vragen we je om strings te comprimeren naar bitstrings, en gecomprimeerde bitstrings te decomprimeren naar de oorspronkelijke string volgens het principe van de Huffmancodering. Hierbij worden bitstrings voorgesteld als strings die enkel bestaan uit de karakters 0 en 1. We geven je de bitstrings die gebruikt worden om de verschillende symbolen te comprimeren onder de vorm van een tekstbestand, waarvan elke regel bestaat uit een symbool (één enkel karakter), gevolgd door een tab en de bitstring die gebruikt wordt om dit symbool voor te stellen. Hieronder zie je bijvoorbeeld de inhoud van het tekstbestand met de bitstrings die corresponderen met de symbolen uit bovenstaande tabel. Merk hierbij op dat het symbool op de eerste regel in dit voorbeeld een spatie is.

```
 111
a 010
e 000
f 1101
h 1010
i 1000
m 0111
n 0010
```

s 1011
t 0110
l 11001
o 00110
p 10011
r 11000
u 00111
x 10010

Definieer een klasse ZIP die kan gebruikt worden om strings te comprimeren als bitstrings, en gecomprimeerde bitstrings te decomprimeren naar de oorspronkelijke strings volgens het principe van de Huffmancodering. Bij initialisatie van een object van deze klasse moet de locatie van een tekstbestand opgegeven worden. Dit bestand moet de codering van de verschillende symbolen bevatten, in het formaat zoals hierboven omschreven. Voorts moeten de objecten van de klasse ZIP minstens de volgende methoden ondersteunen:

- Een methode symbool2bitstring waaraan een symbool (een string met één enkel karakter) moet doorgegeven worden. De methode moet de gecomprimeerde bitstring teruggeven die correspondeert met het gegeven symbool. Indien het gegeven symbool niet voorkomt in het bestand dat bij initialisatie werd opgegeven, dan moet de methode een AssertionError opwerpen met de boodschap onbekend symbool "*x*", waarbij *x* moet ingevuld worden met het gegeven symbool.
- Een methode bitstring2symbool waaraan een bitstring moet doorgegeven worden. De methode moet het symbool teruggeven dat correspondeert met de gegeven bitstring. Indien de gegeven bitstring niet voorkomt in het bestand dat bij initialisatie werd opgegeven, dan moet de methode een AssertionError opwerpen met de boodschap ongeldige bitstring.
- Een methode comprimeer waaraan een string moet doorgegeven worden. De methode moet de gecomprimeerde bitstring teruggeven die correspondeert met de gegeven string. Indien de gegeven string karakters bevat waarvan de bitstringcodering niet vervat zit in het bestand dat bij initialisatie werd opgegeven, dan moet de methode een AssertionError opwerpen met de boodschap onbekend symbool "*x*", waarbij *x* moet ingevuld worden met het eerste (meest linkse) karakter dat niet voorkomt in het bestand.
- Een methode decomprimeer waaraan een gecomprimeerde bitstring moet doorgegeven worden. De methode moet de oorspronkelijke string teruggeven die correspondeert met de gegeven string volgens het schema van de Huffmancodering. Indien de gegeven bitstring geen geldige stringvoorstelling heeft volgens het schema van de Huffmancodering, dan moet de methode een AssertionError opwerpen met de boodschap ongeldige bitstring.

Voorbeeld

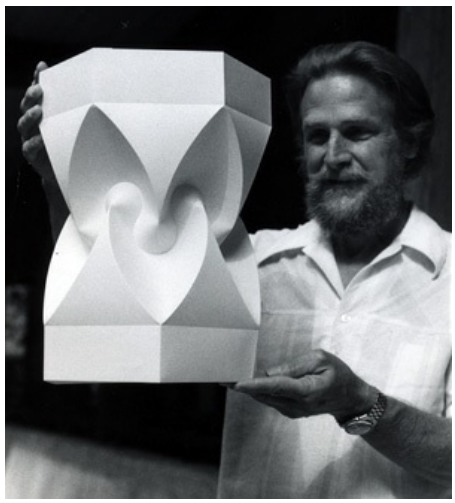
Bij onderstaande voorbeeldsessie gaan we ervan uit dat het tekstbestand codes.txt zich in de huidige directory bevindt.

```
>>> zip = ZIP('codes.txt')

>>> zip.symbool2bitstring('i')
'1000'
>>> zip.symbool2bitstring('e')
'000'
>>> zip.symbool2bitstring('T')
Traceback (most recent call last):
AssertionError: onbekend symbool "T"
```

```
>>> zip.bitstring2symbool('1000')
'i'
>>> zip.bitstring2symbool('000')
'e'
>>> zip.bitstring2symbool('01')
Traceback (most recent call last):
AssertionError: ongeldige bitstring

>>> zip.comprimeer('internet')
'10000010011000011000001000001110'
>>> len(zip.comprimeer('internet'))
31
>>> zip.comprimeer('internet explorer')
'1000001001100001100000100000110111000100101001111001001101100000011000'
>>> zip.comprimeer('mozilla firefox')
Traceback (most recent call last):
AssertionError: onbekend symbool "z"

>>> zip.decomprimeer('10000010011000011000001000001110')
'internet'
>>> zip.decomprimeer('1000001001100001100000100000110111000100101001111001001101100000011000')
'internet explorer'
>>> zip.decomprimeer('10000010011000011000001000001101')
Traceback (most recent call last):
AssertionError: ongeldige bitstring
>>> zip.decomprimeer('10000010011000011000000000110')
Traceback (most recent call last):
AssertionError: ongeldige bitstring
```

Epiloog

Toen David Huffman in 1999 stierf, verloor de wereld één van zijn meest getalenteerde informatici — Huffman was het best gekend voor zijn ontdekking van de Huffmancodering, een techniek die gebruikt wordt in datacompressie. Maar het verloor ook een pionier op het vlak van **wiskundige origami**, een uitbreiding van de traditionele kunst van het papiervouwen als toepassing van computationele meetkunde, getaltheorie, codeertheorie en lineaire algebra. Dit onderzoeksdomein vindt vandaag de dag steeds meer toepassingen, en helpt onderzoekers bij het opvouwen van allerlei zaken zoals eiwitten, airbags voor auto's en ruimtetelescopen.



David Huffman met één van zijn papiervouwsels in 1978.

Huffman werd sterk aangetrokken tot dit werk door zijn onderzoek naar de wiskundige

eigenschappen van oppervlakken zonder kromming, waarbij hij bestudeerde hoe papier zich gedraagt in de buurt van plooien en toppen van kegels. Tijdens de laatste twee decennia van zijn leven maakte hij honderden prachtig ingewikkelde papiermodellen waarvan de plooien gebogen waren in plaats van recht.

Maar hij hield zijn onderzoek over papiervouwen grotendeels voor zichzelf. Hij publiceerde slechts één artikel over dit onderwerp, en veel van zijn ontdekkingen gingen verloren bij zijn dood. "Hij voorzag heel wat zaken die andere mensen pas later hebben ontdekt of nu pas ontdekken," vertelde laserfysicus Robert Lang in 2004 aan de New York Times. "Minstens de helft van wat hij gedaan heeft wijkt af van alles wat ik tot nu toe reeds gezien heb." Computerwetenschapper Erik Demaine van MIT werkt momenteel met Huffman's familie om zijn ontdekkingen bijeen te sprokkelen en te documenteren.

"Ik beweer niet dat ik een kunstenaar ben. Ik weet zelfs niet zeker hoe ik kunst zou moeten definiëren," vertelde Huffman in 1979 tijdens een publieke voordracht. "Maar ik vind het logisch dat de elegante wiskundige stellingen in verband met papieren oppervlakken ook moeten leiden tot visuele elegantie."