Strings and the Burrows-Wheeler Transform

Author: Franco Saliola

Sage/Python includes a builtin datastructure for strings. There are several ways to input strings. You can input a string using single quotes `'` or double quotes `"`:

```
sage: s = "This is a string!"
sage: s
```

```
sage: t = 'So is this!'
sage: print(t)
```

You can also input a string using three quotes (`"""` or `'''`). This is useful if you want to use both `"` and `'` in your string, or you want your string to span multiple lines:

```
sage: s = """
sage: This is a multi-line
....:         string
....: that includes 'single quotes'
....:           and "double quotes".
sage: """
sage: print(s)
```

Exercise: Create and print the following string.

```
sage: #   \ | ( | ) / /
....: # _____________________
....: # |                   |
....: # |                   |
....: # |  I <3 Coffee! /--\
....: # |                   |  |
....: #  \              /\--/
....: #   \____________/
```

Exercise: Without using cut-and-paste(!) *replace* the substring **I <3 Coffee!** with the substring **I <3 Tea!**.

Exercise: Print a copy of your string with all the letters capitalized (uppercase).

Strings behave very much like lists. For example,

Operation	**Syntax for strings**	**Syntax for lists**
Accessing a letter	`string[3]`	`list[3]`
Slicing	`string[3:17:2]`	`list[3:17:2]`
Concatenation	`string1 \+ sting2`	`list1 \+ list2`
A copy	`string[:]`	`list[:]`
A reversed copy	`string[::\-1]`	`list[::\-1]`
Length	`len(string)`	`len(list)`

Exercise: The *factors* of length 2 of 'rhubarb' are

- hu
- ub
- ba
- ar
- rb

Write a function called `factors(s, l)` that returns a list of the factors of length `l` of `s`.

Exercise: Using your `factors` function list all the factors of length 3 of `'rhubarb'`.

Exercise: What happens if you apply your function `factors` to the list `[0,1,1,0,1,0,0,1]`? If it doesn't work for both lists and strings, go back and modify your function so that it does work for both.

Run-length encoding

The string

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWW

begins with **W** 12 times, then **B** once, then **W** 12 times, then **B** 3 times, then **W** 24 times, then **B** once and then **W** 14 times. Thus, it can be encoded as

(W, 12), (B, 1), (W, 12), (B, 3), (W, 24), (B, 1), (W, 14).

This is called the *run-length encoding* of the string.

Exercise: Write a function `run_length_encoding(s)` that returns the run-length encoding of a string `s`. Does your function work for lists as well as strings? If not, then can you make it so that it works for both strings and lists? Use your function to compute the run-length encoding of the following list.

```
sage: l = [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,
....: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Rotations

The *rotations* of the string 'bananas' are:

- ananasb
- nanasba
- anasban
- nasbana
- asbanan
- sbanana

and if we sort these alphabetically, then we get:

- anasban
- asbanan
- bananas
- nanasba

- nasbana
- sbanana

Exercise: Define a function `print_sorted_rotations(s)` that sorts all the rotations of a string and prints them in an array as above. Print the sorted rotations of the strings `'ananas'` and `'cocomero'`.

The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* (BWT) of a string **s** sorts all the rotations of **s** and then returns the last column.

For example, if we sort the rotations of 'bananas':

- ananasb
- anasban
- asbanan
- bananas
- nanasba
- nasbana
- sbanana

then the last column is *bnnsaaa* , so the BWT of *bananas* is *bnnsaaa* .

Exercise: Write a function `bwt(s)` that returns the BWT of the string `s`. Compute the BWT of *bananas* , *ananas* and *cocomero* . (*Hint:* You can return your answer as a list, but if you want to return a string, then you might want to use the `.join()` method for strings.)

Exercise: Combine the functions you defined above to create an `@interact` object that takes a string `s` and prints:

- the sorted rotations of `s`
- the run-length encoding of `s`
- the BWT of `s`
- the run-length encoding of the BWT of `s`

(*hint:* String formatting can be done using the `%` operator. Here are some examples:

```
sage: print('The sum of %s and %s is %s.' % (3,2,3+2))
```

If you are familiar with the *C* language then you will notice that string formating is very similar to *C* 's *printf* statement.)

Use your interact object to explore this transformation, and then to answer the following questions below.

Exercise: What is the BWT of the three following strings?

```
sage: s1 = 'xxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy'
sage: s2 = '01101001100101101001011001101001100101100110100101'
sage: s3 = 'cdccdcdccdccdcdccdcdccdccdcdccdccdcdccdcdccdccdcdc'
```

Do you notice any patterns in the BWT of a string?

Can you think of an application for this transformation?

Find 3 other strings that have a 'nice' image under the BWT.

Exercise: Is the Burrows-Wheeler transformation invertible? (That is, can you find two strings that have the same BWT?)

Exercise: By comparing the BWT of a string with the first column of the array of sorted rotations of a string `s`, devise and implement an algorithm that reconstructs the first column of the array from the BWT of `s`.

Exercise: By examining the first *two* columns of the array, devise and implement an algorithm that reconstructs the first *two* columns of the array from the BWT of a string.

(*hint:* compare the last and first column with the first two columns.)

Exercise: By examining the first *three* columns of the array, devise and implement an algorithm that reconstructs the first *three* columns of the array from the BWT of a string.

Exercise: Write a function that reconstructs the entire array of sorted rotations of a string from the BWT of the string.

Exercise: A *Lyndon word* is a word w that comes first in alphabetical order among all its rotations. Is the BWT invertible on Lyndon words?

Exercise: Explain how one can modify the BWT to make it invertible on arbitrary words. Implement your modified transformation and the inverse transformation.