Word Order

Many problems require you to keep data for a number of different identifiers. In this case, we need a count associated with each particular word. Python's dictionary structure is a good fit for such problems. In this case, the keys will be the words themselves, and the associated values will be the counts.

This is enough to simply give us counts for each word, but dictionaries do not traditionally maintain order. (That has changed: In CPython 3.6, dictionaries keep insertion order as an implementation detail, and in Python 3.7, this is guaranteed by the language. Note that HackerRank is on CPython 3.7.1, but we can't necessarily count on everyone having the latest version of Python.) There are a couple of ways to guarantee the order we want (check out `collections.OrderedDict`), and we'll handle it manually. Our values will be a tuple of (first appearance, count), so we can reconstruct the order after the dictionary is assembled.

```
1   import sys
2   _ = sys.stdin.readline()
3   words = sys.stdin.read().split()
4
5   d = dict()
6   order = 0
7   for word in words:
8       if word in d:
9           d[word] = (d[word][0], d[word][1] + 1)
10      else:
11          d[word] = (order, 1)
12          order += 1
13
14  print(len(d))
15  ordered_counts = sorted(d.values())
16  for order, count in ordered_counts:
17      print(count, end=' ')
```

The first line of the input is the number of words in the input. In languages with manual memory management, knowing this is very helpful. But in Python, we can just `.read()` the whole input and then `.split()` the words apart (line 3). Therefore, on line 2 we read in this value and assign it to the variable _. This is a Python convention that indicates that the return value of the function call isn't being used for anything. We could have called it anything else, but other Pythonistas will recognize what you mean by this.

We actually don't need to assign the return value to anything; the interpreter will just discard this value. But this may worry the next person to read the code. (Who may just happen to be future you.) Why are we reading input but not doing anything with it? Is it a mistake? Assigning the output to `_` emphasizes that this was intentional.

On line 15, `d.values()` will return a list of the values in the dictionary. We can't guarantee the order in which we get the items (depending upon the version of Python we are using), so we use `sorted` to order them by the first appearance (the first element of each value).

The print function usually appends a new line to the end of the output. However, it takes an optional keyword argument, `end` to override that character. By setting `end=' '` (line 17), we make all of the counts appear separated by spaces on the same line.

Reading from standard input

A common task in HackerRank challenges is reading data from standard input. Sometimes there will be boilerplate code to help you out, but in cases like this you have to manage it yourself. There are several ways to do this.

sys.stdin

This object is a stream containing standard input. You may be familiar with file streams in Python (you get one by using `open()`, for example), but the stream interface is used in a number of places. There are many methods you can call on a stream, including:

`.read()`
: Returns a string containing everything up to the end of stream.

`.readline()`
: Read and return a single line from the stream. Note that the line will contain a trailing newline character; use `line.strip()`, for example, to remove this. The stream position is advanced to the next line.

`.readlines()`
: Returns a list of lines. Again, each will include trailing new lines characters.

iteration
: A stream is iterable, yielding a single line (with new line character) each time. This means you can do

```
for line in sys.stdin:
    process(line)
```

Using the `sys.stdin` stream can be very powerful, but you will need to think about the best way to use it. Remember that you need to `import sys` before using it.

input()

The built-in function `input()` is designed for prompting the user for input. But since user input goes to standard in, you can (ab)use it to read whatever is on standard input.

`input()` will read a single line from standard input, and return it without a trailing new line. It is essentially identical to `sys.stdin.readline()[:-1]`. You can’t do all of the things with it that you can do with a stream, but if you just want a single line, it might be a little easier.