You just have two questions to complete this week!

Get Assignment link: https://classroom.github.com/a/PKruQTpV

(8) 1. Many people in English-speaking countries have played the Pig Latin game at some point in their lives. There are other invented "languages" in which words are created using some simple transformation of English. One such language is called *Obenglobish*, in which words are created by adding the letters *ob* before the vowels (*a,e,i,o* and *u*) in an English word. For example, under this rule, the word *english* gets the letters *ob* added before the *e* and the *i*, to form *obenglobish*, which is how the language got its name.

In official Obenglobish, the `ob` characters are only add before vowels that are pronounced, which means that a word like *game* would become *gobame* rather than *gobamobe* since the final *e* is silent. While it is impossible to implement this rule perfectly, you can do a pretty good job by adopting the rule that *ob* should be added before every vowel in the word **except when**:

- the vowel follows another vowel
- the vowel is an *e* occurring at the end of a word

Write a function `to_obenglobish` that takes an English word as an argument and returns its Obenglobish equivalent, using the translation rule given above. For example, your function should be able to output something similar to below:

```
>>> print(to_obenglobish("english"))
obenglobish
>>> print(to_obenglobish("gooiest"))
gobooiest
>>> print(to_obenglobish("amaze"))
obamobaze
>>> print(to_obenglobish("rot"))
robot
```

Don't forget about decomposition! I found it very useful in this problem to write predicate functions which take care of checking the above special conditions.

2. A common form of tabulation is that of creating a histogram, wherein the different elements of the histogram array represent the counts of how many times a particular value (or range of values) showed up. For example, the first 16 digits of π look like:

```
PI_DIGITS = [3,1,4,1,5,9,2,6,5,3,5,5,8,9,7,9]
```

If we are dealing with single digits, then there are 10 possible values: 0 through 9. This maps nicely to index values of an array! So we could store how many times a 1 appears in `PI_DIGITS` (2 times) in the index 1 element of our histogram. The entire histogram array might then look like:

`hist_array`

0	2	1	2	1	4	1	1	1	3
0	1	2	3	4	5	6	7	8	9

showing that there are no 0s, two 1s, one 2, two 3s, and so forth, in the first sixteen digits of π.

(4) (a) Write a function called `create_histogram_array(imax, data)` which takes two values as arguments:

- A maximum index, which represents the total number of possible things you'd like to count
- A list of integer data, which represents the items which you would like to tabulate a histogram of.

This function should return a new list with the histogram counts as each element. For example, your function should be able to mimic:

```
>>> create_histogram_array(5, [1,0,3,2,4,2,2,1,3,0])
>>> [2,2,3,2,1]
```

Note that if your value of `imax` is smaller than the variety of digits in `data`, the later digits will not be counted. Only the first `imax` will be counted. If `imax` is larger than the number of different digits in `data`, 0's will just be counted for those extra digits. So, for instance:

```
>>> create_histogram_array(3, [1,0,3,2,4,2,2,1,3,0])
>>> [2,2,3]
>>> create_histogram_array(8, [1,0,3,2,4,2,2,1,3,0])
>>> [2,2,3,2,1,0,0,0]
```

(3) (b) Write a new function called `print_histogram(array)` that takes a histogram counts list (of the sort returned by `create_histogram_array`!) as an argument. This function should print to the console a rotated histogram using asterisks to indicate the count of values in each index. For example, if you call `print_histogram` with the histogram formed by the first 16 digits of π, you should see the following output:

```
0:
1: **
2: *
3: **
4: *
5: ****
6: *
7: *
8: *
9: ***
```

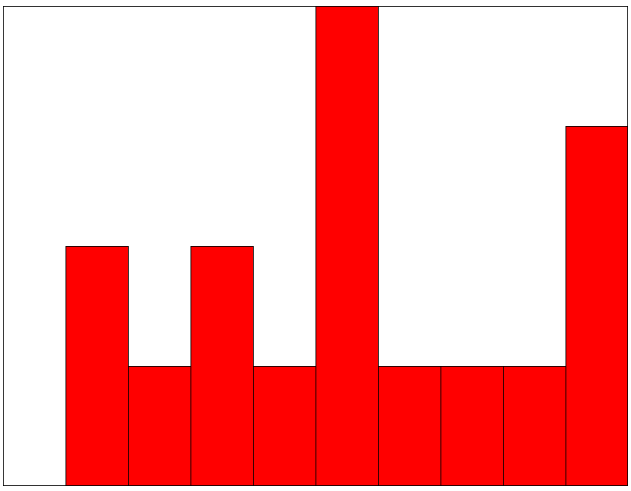
Your function should mimic this printing *exactly*. This should not be particularly complicated, as you have all the tools you need now with string formatting and the repetition operator for strings.

(5) (c) Write a function called `create_histogram_graph(array, width, height)`, which will use PGL to create a histogram `GCompound`. The `array` parameter represents the same histogram array as created in Part (a), and the width and the height parameters represent the desired size of the `GCompound`. You will need to use these values to figure out how wide and tall each of your histogram rectangles should be. The function should return a `GCompound` object that contains filled `GRect` objects arranged to produce a traditional bar graph histogram, where the height of a particular bar reflects a proportional number of items in that index of the histogram array. The scaling of the bars should be chosen so that the largest bar in the graph completely fills the desired height, and all the bars arranged next to one onether completely fill the desired width. The reference point of the `GCompound` should be the upper left corner.

To illustrate how this part should work, it may help to consider the following simple test program, which is included in the code template:

```
def test_create_histogram_graph():
    WIDTH, HEIGHT = 800, 600
    gw = GWindow(WIDTH, HEIGHT)
    PI_DIGITS = [3,1,4,1,5,9,2,6,5,3,5,5,8,9,7,9]
    array = create_histogram_array(10, PI_DIGITS)
    graph = create_histogram_graph(array, WIDTH, HEIGHT)
    gw.add(graph)
```

which should display the following graph so that it fills the graphics window:



Note that because `create_histogram_graph` just gives you a `GCompound`, if you made 4 of them with half the width and height, you could also display multiple across the screen by adding them at the correct coordinates: