CS:1210 (22C:16) Homework 2

Due via ICON on Thursday, Feb 27th, 4:59 pm

What to submit: Your submission for this homework will consist of three text files, named `hw2a.py`, `hw2b.py`, and `hw2c.py`. These should contain Python programs for Problems (a), (b), and (c) respectively. These files should each start with with a comment block containing your name, section number, and student ID. You will get no credit for this homework if your files are named differently, have a different format (e.g., docx), and if your files are missing your information. For this homework (and all future homeworks), make sure that your program is carefully documented and variables names are chosen with care.

(a) One way to speedup our implementation of primality testing is the following. Suppose that n (a positive integer) is the input that we want to check for being a prime. Let us assume that we have already tested if 2 and 3 are factors of n and suppose that neither of these are factors of n. It turns out that the rest of the candidate factors the program needs to consider have the form $6k \pm 1$ for $k = 2, 3, \ldots$ (i.e., one less than a multiple of 6 and one more than a multiple of 6). In other words, the program need not consider any factors besides 5, 7, 11, 13, 17, 19, 23, 25, etc. Thus a more efficient algorithm would only consider these candidate factors, skipping over the rest. The reason for why this works is quite straightforward and I'll let you read about this at Wikipedia's page on primality testing (see `http://en.wikipedia.org/wiki/Primality_test`). In fact this Wikipedia page also provides a nice Python implementation of this faster algorithm.

For this problem, you are asked to implement a program that compares the running time of the "naive" algorithm that we developed in class (see `primalityTesting3.py` on the course page) with the faster primality testing algorithm, mentioned above. Your program should start by reading a positive integer N, specifying the number of integers the program is expected to test for primality. Then the program reads N positive integers that it is going to test for primality using both algorithms (i.e., the "naive" algorithm and the faster algorithm). The program then runs the "naive" algorithm on each of the N input integers and outputs the average running time. Following this, the program runs the faster algorithm on each of the N integers and outputs the average running time.

(b) Generating random input data is a problem that computer scientists (and many others) think about because it provides an easily repeatable way of testing programs and algorithms against data that might model real world features. Here is a simple programming problem that asks you to generate a random sequence according to a prescribed probability distribution.

Write a program that reads a positive integer N and outputs a random sequence of N integers in the range $[1, 100]$ such that each integer x in the sequence is generated according to the following probability distribution. First, a sub-range is chosen for x, with sub-range $[1, 25]$ chosen with probability 1/8, the sub-range $[26, 50]$ with probability 1/2, the sub-range $[51, 75]$ with probability 1/4, and the sub-range $[76, 100]$ with probability 1/8. Once a sub-range for x has been chosen (e.g., $[1, 25]$) then a value for x is picked uniformly at random from that sub-range (i.e., each value in the sub-range is equally likely to be chosen).

Take a look at the `random` module to find functions useful for this task. The interaction between the user and your program should look like:

```
10
4
78
```

```
37
44
6
91
30
84
90
57
```

In this example, the user starts by inputting 10, the size of the sequence she wants. In response your program generates the sequence $4, 78, 37$, etc. Of course, since this is a random sequence, your program will almost surely generate some other output and furthermore each time it is run, it will generate a different output.

(c) The input is a sequence of non-empty strings terminated by an empty string. After reading this sequence, your program should output the longest string and the second-longest string. For example, the interaction between your program and the user might look like this:

```
hello
ok
secondary
mammoth
estimate
density

LONGEST: secondary
NEXT LONGEST: estimate
```

In this example, the user inputs the strings `"hello"`, `"ok"`, etc., followed by the empty string. Your program outputs the longest string and the second-longest string. In general, if there are two or more longest strings, it does not matter what your program outputs. Similarly, if there are several second-longest strings, it does matter what your program outputs.

Here is a simple algorithm that can perform this task. Suppose that after processing some number of strings you have figured out (i) the longest string and its length and (ii) the second-longest string and its length. Now your program reads the next string and there are three possibilities:

(a) The new string is longer than the longest string. In this case, the new string becomes the longest string and the longest string gets demoted to being the second-longest string.

(b) The new string is not longer than the longest string, but is longer than the second-longest string. In this case, the new string becomes the second-longest string.

(c) The new string is not longer than the second-longest string. In this case, your program need not make any updates.

You should implement this algorithm for your program.
