CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #1-3

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 PM

Getting Started

To help you get started on this assignment, we have given you a skeleton Python file named `counting.py`. This file contains several *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You must fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `1-3.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

This assignment is worth a total of 20 points. The automated grading system will execute your solution several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. You must earn at least 15 points (75%) in order to pass (receive credit for) this assignment.

- At the top of the `counting.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Assignment #1-3)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters, or that use different parameter names) can't be graded by our automated grading system, and may receive a grading penalty.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Counting Problems

Part I: Smart TV Blues (10 points)

My smart TV has an onscreen keyboard feature that allows me to use the remote control to enter passwords for online services (e.g., Netflix), enter the names of movies to search for, etc. This keyboard resembles the following:

a	b	c	d	e	f	g
h	i	j	k	l	m	n
o	p	q	r	s	t	u
v	w	x	y	z	0	1
2	3	4	5	6	7	8
9	Delete, Enter, etc.					

Unfortunately, my remote control is severely broken due to a coffee spill; the up and down buttons don't work at all, so I can only move between characters on the keyboard using the left and right buttons. Fortunately, there's a workaround (sort of): moving off the left edge of a row takes you to the row above, and moving off the right edge of a row takes you to the row below. This is a nuisance. Being an engineer, I want to measure the "annoyance factor" for each password or movie title that I need to enter using this keyboard (and broken remote). My *annoyance factor* is the total number of rows that I have to move between in order to enter a particular string.

For example, consider the password `"avalon31"`:

- I begin at 'a'. The first character doesn't count towards the annoyance factor, which is currently 0.
- Moving to 'v' requires me to move three rows down, adding 3 to my annoyance factor.
- Moving back to 'a' adds another 3 to my annoyance factor, since I have to move three rows back up.
- The letter 'l' is one row down from 'a', so my annoyance factor only increases by 1 this time.
- 'o' is one row down from 'l', adding another 1 to my annoyance factor.
- 'n' is one row back up, so my annoyance factor increases by 1.
- '3' is three rows below 'n', adding another 3 to my annoyance factor.
- Finally, '1' is one row above '3', adding a final 1 to my annoyance factor.

Thus, my final annoyance factor is 3 + 3 + 1 + 1 + 1 + 3 + 1, or 13.

Complete the `annoyanceFactor()` function, which takes a single string of lowercase letters and digits representing the string that I want to enter. The function returns the total number of row changes I need to make.

As a general solution strategy, consider the following:

1. Create a list that holds the letters and digits on the keyboard shown above (**Hint:** Use Python's `list()` function on the concatenation of `string.ascii_lowercase` and `string.digits`).
2. Identify the first character's row number. A character's row number is equal to its list index divided by 7 (using floor division). For example, 'c' (index 2) is in row 0 (2 // 7 is 0), and 'y' (index 24) is in row 3 (24 // 7 is 3). This value will be our starting row. (**Hint:** Python lists have an `index()` method that is extremely useful here)

3. For each remaining character in the string:
   (a) Calculate the current character's row number using the formula from Step 2.
   (b) Use Python's `abs()` function to calculate the absolute difference between the new character's row number and our current row number. Add that value to the "annoyance factor" variable.
   (c) Set the current row to the new character's row number.

Examples:

Method Call	Expected Output
`annoyanceFactor("avalon31")`	`13`
`annoyanceFactor("23fish")`	`7`
`annoyanceFactor("waterbed")`	`11`
`annoyanceFactor("ncc1701a")`	`9`
`annoyanceFactor("excelsior")`	`10`

Part II: Primes Embedded in Primes (10 points)

Complete the `primeContainer()` function, which takes a single (non-negative) integer argument *n*. This function returns another integer value: the smallest prime number that contains exactly *n* distinct smaller prime numbers as substrings.

As an example, suppose that *n* is 0. 2 is the smallest prime number that has 0 smaller primes as substrings (the only substring of 2 is 2, which is *not* smaller than itself). If *n* is 1, the smallest prime number that contains exactly 1 smaller prime number within itself is 13 (since 3 is prime). If *n* is 2, the smallest prime number that contains 2 distinct smaller prime numbers as substrings is 23, which contains both 2 and 3.

We have provided a helper function, `sieve()`, that uses the Sieve of Eratosthenes algorithm (from the beginning of the semester) to generate a list of prime numbers for your use.

Your general solution strategy should be as follows:

1. Use the `sieve()` function to generate a list of prime numbers (start with the set of all primes up to 50000).
2. For each prime number `p` in that list:
   (a) Set `count` to 0
   (b) For each prime number `x`, from 2 up to (but not including) `p`:
      - If `x` is a substring of `p` (note: convert both `x` and `p` to strings in order to do this), add 1 to `count`.
      - As soon as `count` equals `n`, return `p` to end the function.

Hint: For Step 2(b), use a `while` loop. Set your starting index to 0, and loop while the prime number at the current index is less than `p`.

WARNING: This process may take a noticeable amount of time (as in minutes) for values of `n` greater than 11. If you test large values of `n`, be prepared to wait for a short while as your computer performs the check.

Examples:

Method Call	Expected Output	Prime Substrings (for reference)
`primeContainer(3)`	`113`	3, 11, 13
`primeContainer(5)`	`1237`	2, 3, 7, 23, 37
`primeContainer(6)`	`1373`	3, 7, 13, 37, 73, 137
`primeContainer(8)`	`11317`	3, 7, 11, 13, 17, 31, 113, 131