ASSIGNMENT 2: PRIMES, DUE THURSDAY, OCTOBER 27 AT 11:59PM

Contents

1. The basic computational algorithms for primes

Problem 1 (10 points)**.** This problem asks you to write two very closely related functions.

First, write a function, isprime(n), which accepts a positive integer n and determines whether or not n is prime using trial division. Your program should be able to test numbers as large as 10 or 11 digits more or less instantaneously, even in the worse case (namely, when n is prime). The function should return the boolean value 'True' if n is prime, and 'False' if n is composite.

Second, write a function, divisor(n), which accepts a positive integer $n \geq 2$ and returns the smallest positive divisor of n greater than 1. Again, your program should run more or less instantaneously for numbers as large as 10 or 11 digits.

Problem 2 (10 points)**.** Write a function, eratosthenes(n), which accepts a positive integer $n \geq 2$ and returns a list consisting of all the prime numbers from 2 to n, in increasing order. The function should actually implement the sieve of Eratosthenes to find a list of all the prime numbers between 2 and n instead of calling isprime repeatedly on integers from 1 to n. Your function should run in under a second for n up to 10^6. (If your computer is a bit old, it's possible that your program will take slightly longer than 1 second to run if $n = 10^6$. That's okay.)

An interesting experiment to try on your own is to implement a function which finds all the primes from 2 to n by repeatedly running isprime (which uses trial division) on every integer from 2 to n. Compare the run time of this function against the eratosthenes function you wrote. You should find that the eratosthenes function is faster, at least when n is reasonably large (I noticed a difference in speed around $n = 10^6$.) If you want to be really precise, you can learn how to use the *timeitmodule* in Python, which lets you time short pieces of code.

The next problem will ask you to find a factorization of an arbitrary positive integer $n \geq 2$. To do so, we need a reasonably compact way of describing a factorization. Recall that the factorization of n has the form

$$n = p_1^{e_1} \dots p_r^{e_r},$$

where the e_i are various positive integers, and the p_i distinct primes. We could represent this information several ways. Perhaps the most natural is some sort of data structure which pairs each p_i with its exponent e_i. One possibility is to return a list or tuple of pairs (p_i, e_i), but we will take this opportunity to introduce a new and very useful data structure known as the *dictionary*.

A dictionary is like a list, except that you can use a much wider variety of objects as indices. A list always requires you to use consecutive integers, starting at 0, as the indices for that list, but a dictionary lets you use numbers or strings, or tuples of numbers or strings, as indices. The indices of a dictionary are known as *keys*. (You are allowed to use any *immutable* object; *mutable* objects like lists or other dictionaries are not allowed as keys. The distinction between the two would take a bit long for us to discuss here, and the reasons why only immutable objects are allowed as keys are a bit too technical for us to explain right now.)

Recall that a tuple is delimited by $(,)$ and a list by $[,]$. A dictionary is delimited by $\{,\}$. Despite the similarity to set notation, dictionaries should not be thought of as sets! On the other hand, you can think of a dictionary as an implementation of a mathematical function.

You can create and initialize a dictionary as follows:

```
>>> d = {``yes": 4, ``no": 2, 5: 7, 2: "blah"}
>>> d
{`yes': 4, 2: `blah', 5: 7, `no': 2}
>>> d["yes"]
4
>>> d["not here"]

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    d["not here"]
KeyError: 'not here'
```

Objects which serve as indices to a dictionary are called *keys* of that dictionary. The object that is associated to a particular key is called a *value*, and if k is a key of d, then k, d[k] is called a *key-value* pair of d.

Notice that if you try to use objects which are not keys of d, you get an error. On the other hand, assuming that you initialized d as above, you can create new key-value pairs by simply assigning something to a new key, just like how variables are defined by being assigned to them:

```
>>> d["new"] = "this is new"
>>> d
{'new': 'this is new', 'yes': 4, 2: 'blah', 5: 7, 'no': 2}
```

If you want a list of keys for the dictionary d, then d.keys() will return a list containing the keys, in no particular order. Similarly, d.values() returns a list of values of d, and d.items() returns a list of tuples, where each tuple contains a key and its associated value.

```
>>> d.keys()
['new', 'yes', 2, 5, 'no']
>>> d.values()
['this is new', 4, 'blah', 7, 2]
```

```
>>> d.items()
[('new', 'this is new'), ('yes', 4), (2, 'blah'), (5, 7), ('no', 2)]
```

Problem 3 (10 points)**.** Write a function, factor(n), which given a positive integer $n \geq 1$, returns a dictionary containing the prime factorization of n, where the key-value pairs are the primes appearing in the factorization as keys and the exponents as values. For example, factor(15) should return $\{3 : 1, 5 : 1\}$. If factor is called with $n = 1$, factor should return the empty dictionary.

2. Statistical experiments with primes

In this section, we will use the functions we wrote in the previous section to help us gather some basic numerical data on distribution of primes. These are the sorts of 'numerical experiments' that mathematicians like Euler or Gauss might have done in their preliminary studies on counting prime numbers, except that they had to do all their calculations by hand!

Problem 4 (5 points)**.** Write a function, primepi(x), which given a positive number $x \geq 2$ returns the number of primes $\leq x$. For example, primepi(2) would return 1, while primepi(3.5) would return 2. Your function should work not just with integers, but also with floating point numbers. You can use any of the functions you wrote earlier in this assignment; your function should run under 1s for $x \leq 10^6$.

Recall that the prime number theorem says that

$$\pi(x) \sim \frac{x}{\log x} \sim \int_2^x \frac{1}{\log t}\, dt.$$

We will investigate whether $x/\log x$ or $\mathrm{li}(x) = \int_2^x \frac{1}{\log t}\, dt$ is a better estimate for $\pi(x)$, at least for 'small' x.

It is relatively easy to get Python to calculate the value of $x/\log x$. The function $\log x$ is not loaded into the Python interpreter when you initially boot it up, but it is available in the math module. A module is a collection of prewritten code, either in Python or another language (most frequently C), which provides Python functions (and possibly other Python objects) for use. For example, the math module contains definitions for several common mathematical functions besides $\log x$, such as the various trigonometric functions. The math module also contains approximations to π and e which are quite accurate.

To access the functions in the math module, you need to *import* it. You do so as follows:

```
>>> import math
```

At this point, the functions in the math module are available for you to use! To use, say, the log function, you would use the function math.log:

```
>>> math.log(5)
1.6094379124341003
>>> math.log(math.e**2)
2.0
```

The reason you need to prefix each function with 'math.' is because each function is part of the math module, and if you use the 'import math' command to import the math module, Python requires that you keep track of the fact that these functions are part of the math module. However, there is a way to circumvent this extra typing. If you use the command 'from math import log', then you can use the log function by typing in log instead of math.log:

```
>>> from math import log
>>> log(5)
1.6094379124341003
```

You use 'from math import log' without using 'import math' earlier. As a matter of fact, if you want to have access to every function in the math module without typing 'math.', you can use the command 'from math import *'. However, use of this in code is frowned upon, because it can lead to unexpected collisions with user-defined functions. For example, maybe you define a function in some code you write, and then use 'from [some module] import *'. If [some module] contains a function with the same name as yours, that function will be used instead of your function, which could lead to all sorts of confusion and hard-to-trace bugs. You should avoid using 'from [module] import *' in code you write, but it is fine to use this command in interactive sessions.

The Python standard library contains many, many modules which are prepackaged with installations of Python, which can be used to do many, many different things. A lot of the power of Python comes from the fact that the standard library is so vast and (for the most part) easy to use.

In summary, once we know of the existence of the math module, we can easily calculate $x/\log x$. However, how do we estimate the logarithmic integral? We can't find an antiderivative of $1/\log t$ in terms of any simple combination of elementary functions, so instead we will numerically estimate the logarithmic integral. To do so, we will use Simpson's rule, from calculus.

Simpson's rule tells us that if we want to numerically estimate $\int_a^b f(t)\,dt$, we can subdivide the interval $[a, b]$ into n equal subintervals, where n is an even number. Let $x_0, x_1, \ldots, x_n$ be the endpoints of these intervals, arranged in increasing order. Then Simpson's rule estimates $\int_a^b f(t)\,dt$ by the following expression:

$$\int_a^b f(t)\,dt \approx \frac{b-a}{3n}(f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\ldots+4f(x_{n-1})+f(x_n)).$$

The first and last terms are $f(x_0), f(x_n)$ multiplied by 1, whereas the rest of the terms are multiplied by coefficients which alternate between 4 and 2. The fact that the second-to-last term is $4f(x_{n-1})$ instead of $2f(x_{n-1})$ is due to the fact that n is even.

Problem 5 (10 points)**.** Write a function, simpsons_li(n), which accepts an even integer $n \geq 4$ and returns an estimate for $\mathrm{li}(n)$ using Simpson's rule, where we divide the interval $[2, n]$ into $n-2$ equal subintervals of length 1. Your function should return a floating point number.

A fun numerical experiment to try is to compare how accurate $\mathrm{li}(x)$ and $x/\log x$ are when used as estimates for $\pi(x)$. One way to do this is to compute the ratios of $\pi(x)$ with $\mathrm{li}(x)$ and $x/\log x$, and see how rapidly they converge to 1.

The next problem asks you to numerically investigate the prime number race 1 mod 4 vs 3 mod 4. Recall that $\pi(x; q, a)$ is the number of primes p with $p \leq x, p \equiv a \bmod q$.

Problem 6 (5 points)**.** Write a function, primerace_3vs1_mod4(n), which accepts an odd integer $n \geq 3$ and returns $\pi(n; 4, 3) - \pi(n; 4, 1)$. This function is positive exactly when there are more primes of the form $4k + 3$ than $4k + 1$ when looking at primes $\leq n$.

Using the above function (or perhaps a slight modification), what is the first odd integer n for which $\pi(n; 4, 1) > \pi(n; 4, 3)$? What is the second? You may need to modify the above function slightly to answer these questions in a reasonable amount of time (although you do not need to turn the answers to these questions in.)

3. Useful information

3.1. **Problems to hand in.** In your file which contains the code you submit, make sure that you can run the file in IDLE without error and that all functions are loaded appropriately. In other words, if you open the .py file you submit and run it in IDLE (using the Run Module command under the Run submenu), there should be no error messages in the interactive interpreter, and you should be able to use the functions you wrote. Also make sure that you spell the names of the required functions correctly! (On the other hand, the names of the arguments passed into the function can be arbitrary, as long as the function accepts the correct number of arguments.) Finally, please put your name near the top of the file in a comment (use the # symbol to write comments; Python will ignore everything on that line which comes after the #).

- Turn in problems 1, 2, 3, 4, 5, and 6 in a single file named [lastname]2.py (without brackets around your last name.)

3.2. **Test cases.** Starting with this assignment we will use a better method for providing test cases. The Python standard library contains a handy module called doctest, which provides a lightweight method for testing test cases. In the assignment page, the template .py file will contain a lengthy *docstring* at the beginning of each function, which contains the test cases we provide for each function (if there are any).

Also, at the end of the file, there will be a short snippet of code, which will only run if you ask IDLE to execute the entire file with the 'Run Module' command. The template we provide will include a 'verbose=True' switch, which will cause your program to output extensive information on the results of each test case. If you would like to suppress the output, and only be informed if cases fail, remove 'verbose=True' from within the parentheses of the doctest.testmod function.

For questions that use floating point numbers, we only test whether your answer is within some degree of accuracy. It is conceivable for your function to fail those testcases even if it is correct because of the way floating point arithmetic is handled on the machine being used.

Usage of the test cases is optional. If you want, you can delete the docstrings, delete the code at the end which calls doctest.testmod, or just not use the template file provided.