# 22C:16 (CS:1210) Homework 2

## Due via ICON on Friday, Feb 22nd, 4:59 pm

**What to submit:** Your submission for this homework will consist of three files. One of them will be a pdf file called `homework2.pdf`. This will contain partial answers to Problems 1 and 2. For Problem 1, the pdf file should contain a classification of the given numbers as prime/composite and for Problem 2, the pdf file should contain the output produced by your program and the answers to the questions posed. This pdf file should start with your name, section number, and student ID. The remaining files should be called `FLTPTest1.py`, `FLTPTest2.py`. These should contain Python programs for Problems 1 and 2 respectively. These files should also start with your name, section number, and student ID appearing at the top of the file as Python comments. 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.

**Fast Primality testing.** Hopefully, you have all been scratching your heads about how it is possible to test integers with 100s of digits for primality in any reasonable amount of time. You might recall the discussion we had in class on how the program we wrote for primality testing would take about $10^{33}$ years (which is much more than the lifetime of the Universe!) even on fast computers, for input with 300 digits. This homework will introduce you to a method for designing extremely fast primality testing algorithms.

One of the oldest *fast* primality testing algorithm, called the *Miller-Rabin Algorithm* dates back to the late 70s. If implemented correctly, the Miller-Rabin Algorithm can easily test integers with 100s of digits, for primality. The Miller-Rabin Algorithm belongs to a class of algorithms called *randomized* algorithms. What makes an algorithm randomized is that one or more of the algorithm's steps are performed by rolling a die (or flipping a coin). In other words, the algorithm rolls a die and uses the outcome of the die roll to determine what to do next. Of course, algorithms don't have access to "real" dice and randomized algorithms call random number generators to simulate the roll of a die. The Miller-Rabin algorithm is extremely fast, not just because it is randomized, but also because it is allowed to produce an incorrect answer! If the input is a prime, the Miller-Rabin algorithm will correctly figure this out; however, if the input is a composite, then the algorithm may, with a very tiny probability, make an error and report the number as a prime.

In this homework, you are asked to implement a randomized primality testing algorithm that is simpler than the Miller-Rabin Algorithm and is called the *Fermat's Little Theorem Primality Test (FLTP Test)*. The well-known encryption program PGP uses the FLTP Test in its algorithms. As the name suggests, the FLTP Test depends on *Fermat's Little Theorem.* This is an old mathematical result, first stated by Pierre de Fermat in 1640 and it says this:

If $p$ is a prime then for all integers $a$, $1 \leq a < p$, $a^{p-1}$ mod $p$ equals 1.

In other words, if $p$ is a prime then you can pick any integer $a$ between 1 and $p-1$ (inclusive of 1 and $p-1$) and compute $a^{p-1}$, divide this by $p$ and the remainder will be 1.

**Example.** Suppose $p = 7$. Then Fermat's Little Theorem is saying that for $a = 1, 2, \ldots, 6$, $a^6$ mod 7 equals 1. This is easy to check.

| | |
|---|---|
| $1^6 = 1$ | $1 \bmod 7 = 1$ |
| $2^6 = 64$ | $64 \bmod 7 = 1$ |
| $3^6 = 729$ | $729 \bmod 7 = 1$ |
| $4^6 = 4096$ | $4096 \bmod 7 = 1$ |
| $5^6 = 15625$ | $15625 \bmod 7 = 1$ |
| $6^6 = 46656$ | $46656 \bmod 7 = 1$ |

Fermat's Little Theorem suggests the following simple algorithm for primality testing:

> Given an integer $n > 1$, compute $a^{n-1} \bmod n$ for each $a = 1, 2, \ldots n-1$. If for any of the $a$'s that were considered, $a^{n-1} \bmod n \neq 1$ then output `composite`; otherwise output `prime`.

While this algorithm is correct, it is not any faster than the naive primality testing algorithms we have already implemented. Notice that the above algorithm simply runs through all $a$'s between 1 and $n-1$.

To speed up up this algorithm we use a different mathematical fact. Before we can state this fact, we need to define two pieces of terminology.

(i) For a composite $n$, we call an integer $a$, $1 \leq a < n$, a *Fermat witness* if $a^{n-1} \bmod n \neq 1$. Thus a Fermat witness is a "witness" to the compositeness of $n$.

(ii) A positive integer $n$ is a *Carmichael number* if $a^{n-1} \bmod n = 1$ for all $a$'s in the range $[1, n-1]$ *that are relatively prime to n.* Recall that two numbers are relatively prime if they have no common factors other than 1. For example, 4 and 9 are relatively prime.

**Example.** The smallest Carmichael number is 561. 561 is a composite because $3 \times 11 \times 17 = 561$. 561 is a Carmichael number because for every $a$ in the range $[1, 560]$ that is relatively prime to 561, $a^{560} \bmod 561 = 1$. So what are some values of $a$ that are relatively prime to 561? 1, 2, 4, 5, and 7 are the first 5 integers in the range $[1, 560]$ that are relatively prime to 561. For each value of $a = 1, 2, 4, 5, 7$, it is the case that $a^{560} \bmod 561 = 1$. For values of $a$ not relatively prime to 561, $a^{560} \bmod 561$ may or may not be equal to 1.

Now the mathematical fact that helps us speed up primality testing is this:

> Every composite integer $n$ is either a Carmichael number or at least 1/2 of the integers in $[1, n-1]$ are Fermat witnesses.

Thus the above fact is telling us that with the exception of Carmichael numbers, every composite $n$ has lots of Fermat witnesses – at least 50% of the numbers in the range $[1, n-1]$ are Fermat witnesses.

**Example.** Let $n = 6$. The following table shows values of $a^5 \bmod 6$ for $a = 1, 2, 3, 4, 5$.

| | |
|---|---|
| $1^5 = 1$ | $1 \bmod 6 = 1$ |
| $2^5 = 32$ | $32 \bmod 6 = 2$ |
| $3^5 = 243$ | $243 \bmod 6 = 3$ |
| $4^5 = 1024$ | $1024 \bmod 6 = 4$ |
| $5^5 = 3125$ | $3125 \bmod 6 = 5$ |

From the table it is clear that 6 has 4 Fermat witnesses - thus more than 50% of the 5 possible values of $a$ are Fermat witnesses.
This means that if the input is a non-Carmichael composite $n$, then we can pick an integer $a$ at *random* from the range $[1, n-1]$ and expect that $a$ will be a Fermat witness for the compositeness of $n$ with probability at least 1/2. Thus we would have an at least 50% chance of correctly identifying $n$ as a composite, just by performing one test. To improve the chances of getting the test right, we could just repeat the random choice of $a$ a few times. Suppose we repeat the above process 10 times, independently picking $a$ at random (from the range $[1, n-1]$) each time, then the probability of not finding a Fermat witness all 10 times would be under $1/2^{10} = 1/1024$. Thus the probability of incorrectly declaring that a non-Carmichael composite is a prime is less than 1/1000, even if we repeat the test only 10 times. This leads to the following simple *randomized* algorithm for primality testing:

```
**Input:** a positive integer $n$
**Algorithm:** FLTP TEST
**repeat** 10 times
    pick an integer $a$ at random from $[1, n-1]$
    if $a^{n-1} \bmod n \neq 1$, output composite and exit the program.
output prime
```

This leaves just the vexing issue of Carmichael numbers. These may have only few Fermat witnesses and so the above algorithm may declare them as primes (even though they are composite) with a fairly high probability. For example, 561 has 240 Fermat witnesses, somewhat less than half of 560. In fact, this is the main reason that the Miller-Rabin Algorithm and not the FLTP test is used for primality testing in general. However, Carmichael numbers are not that common. The smallest Carmichael number is 561 and these numbers get rarer as we start considering larger numbers.

1. Implement the FLTP TEST. Your program should prompt the user for a positive integer, larger than 1 and then output a message indicating the primality of the input. Notice that FLTP TEST involves computing $a^{n-1} \bmod n$, where $n$ could be quite large. What Python provides via the `pow` and the `math.pow` functions or the `**` operator are just not fast enough for our purposes So I have written a function called `fastPowerMod(a, d, n)` that computes $a^d \bmod n$ efficiently and is quite suitable even when $n$ has 100s of digits. A link to this function is posted on the course page.

   Use your implementation to test the primality of the following numbers:

   - 5991810554633396517767024967580894321153
   - 19822271254366240129112696248055903545291688310293
   - 27175146095341224357465037532218133092930145221379
   - 470287785858076441566723507866751092927015824834881906763507
   - 693711969678975263512873427191894879124339838606362751311911118403883

2. As you know from the above discussion, the FLTP TEST can incorrectly classify composites as primes. Write a program that runs the FLTP TEST on all integers in the range $[500, 100000]$ and reports all integers in this range that are *incorrectly* classified as primes.

   To complete this task, your program would have to be able to correctly identify primes/-composites and the easiest way to do this is to simply use the naive primality testing algorithm we have already implemented. Recall that this algorithm always returns the correct answer, just that it is terribly slow. So take Version 3 of our primality testing program (`primalityTesting3.py`) and use this to create a boolean function called `exactPrimalityTest(n)` that returns `True` if $n$ is prime and `False` otherwise. Once the function `exactPrimalityTest` has been implemented, you can solve this problem by simply running through each number $n = 500, 501, \ldots, 100000$, first using FLTP TEST on $n$ and then calling `exactPrimalityTest` on $n$.

   Examine the output of your program and compare the output with the list of Carmichael numbers less than 100000 (there are not that many Carmichael numbers under 10,000). Are you seeing any non-Carmichael composites classified as primes? In general, what would be a simple way of improving the accuracy of the program with respect to non-Carmichael composites?

**Extra Credit: 10 points.** The 2013 University of Iowa Computing Conference is being held on Friday, Feb 22nd (starting at 6 pm) and on Feb 23rd (see `http://acm.uiowa.edu/uicc/` for details). Since you will be busy with Exam 1, you will have to skip the Feb 22nd talk. However, there are 4 talks on Saturday. Go to the UICC website for details on times and locations.

To get extra credit, attend any one talk and write a 1 paragraph (5-7 sentences) summary of the talk. These summaries are due in your discussion section on Tuesday, 2/26.

---