Virtual Lab 5 Solution: Chinese Remainder Theorem and Euler's Theorem

EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, October 6th, 2014 at 12pm

Instructions:

- Complete this lab by filling in all of the required functions, marked with `"YOUR CODE HERE"`
- **This week's Virtual Lab builds heavily on last week's lab**. Make sure you understand how to do last week's lab and/or read the solution carefully, or you will be very confused.
- If you plan to use Python, make sure to go over **Tutorial 1A: Introduction to Python and IPython** before attempting the lab
- Make sure you run every code cell one after another, i.e. don't skip any cell. A shortcut for doing this in the notebook is Shift+Enter. When you finish, choose 'Cell > Run All' to test your code one last time all at once.
- Most of the solution requires no more than a few lines each
- Please do not hardcode the result or change any function without the `"YOUR CODE HERE"` mark
- Questions? Bring them to our Office Hour and/or ask on Piazza
- Good luck, and have fun!

Table of Contents

The number inside parentheses is the number of functions or code cells you are required to fill out for each question. Always make sure to double check before you submit.

In [1]:
```
# We'll do some basic plotting in this week's lab.
# Please do not remove the line below.
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

## Introduction

Oystein Ore described a puzzle with a dramatic element from Brahma-Sphuta-Siddhanta (Brahma's Correct System) by Brahmagupta (born 598 AD) as follows.

> An old woman goes to market and a horse steps on her basket and crushes the eggs. The rider offers to pay for the damages and asks her how many eggs she had brought. She does not remember the exact number, but when she had taken them out two at a time, there was one egg left. The same happened when she picked them out three, four, five, and six at a time, but when she took them seven at a time they came out even. What is the smallest number of eggs she could have had?

In the first part of this week's Virtual Lab, we will implement the functions to solve the above puzzle based on the Chinese Remainder Theorem. In the second part, we will explore two very useful theorems in modular arithmetic: Fermat's Little Theorem and Euler's Theorem.

## Question 1: Chinese remainder theorem

Below, you will find an implementation of the function `egcd` that we asked you to implement in last week's lab.

In [2]:
```
def egcd(x, y):
    """
    Extended Euclid's Algorithm. It takes a pair of natural numbers x >= y,
    and returns a triple of integers (d, a, b) such that d = gcd(x, y) = ax +
by.
    """

    if y == 0:
        return (x, 1, 0)
    (d, a, b) = egcd(y, x % y)
    return (d, b, a - (x // y) * b)
```

Based on the above implementation, it's easy to find the positive multiplicative inverse of \(x\) mod \(m\) as follows.

In [3]:
```
def inverse(x, m):
    """
    Returns the positive multiplicative inverse of x mod m
    """

    res = egcd(x, m)[1]
    if res < 0:
        res += m
    return res
```

For example, the inverse of \(117\) mod \(103\) from last week's lab is \(81\).

In [4]:
```
inverse(117, 103)
```

Out[4]:
```
81
```

Implement the `chinese_remainder` function using the `inverse` function above. Refer to the lectures, lecture notes, or Discussion 5M on how the Chinese Remainder Theorem works.

In [5]:
```python
def chinese_remainder(items):
    """
    Solves the Chinese Remainder Theorem

    Given a list of tuples (a_i, n_i), this function solves for x
    such that x \equiv a_i (mod n_i) and 0 <= x < product(n_i)

    Assumes that n_i are pairwise co-prime.

    YOUR CODE HERE
    """

    # Determines N, the product of all n_i
    n_list = [i[1] for i in items]
    N = reduce(lambda x, y: x * y, n_list, 1)

    # Finds the solution (mod N)
    result = 0
    for a, n in items:
        m = N // n
        result += a * inverse(m, n) * m

    return result % N
```

Check you answer to the system of congruences below. The test should print True if your implementation is correct.

\(x \equiv 2\) mod \(3\)

\(x \equiv 3\) mod \(5\)

\(x \equiv 2\) mod \(7\)

In [6]:
```python
chinese_remainder([(2, 3), (3, 5), (2, 7)]) == 23
```

Out[6]:
```
True
```

## Question 2: Solve the puzzle

Now that we have implemented `chinese_remainder`, it's time to solve the puzzle posed in the Introduction section.

In your written homework, write the system of congruences that describes the puzzle (see the example above). Then solve it with the `chinese_remainder` function below. Report the answer in your written homework.

Hint: be very careful, remember that we're assuming the \(n_i\) are pairwise co-prime. Do you really need all the congruences to solve the problem? Recall in Discussion 5M how we eliminate redundant information.

Make sure to double check that this is indeed the smallest number of eggs as well.

```
In [7]: chinese_remainder([(1, 3), (1, 4), (1, 5), (0, 7)])
```

```
Out[7]: 301
```

See the homework solution for the system of congruences, as well as the reasoning behind the above code block.

## Question 3: Fermat's primality test

Last week, you implemented a naive way of testing whether a positive integer p is prime. Here's a sample implementation.

```
In [8]: def is_prime(x):
            """
            Checks if the positive integer x is a prime number
            """
            
            for i in range(2, x-1):
                if x % i == 0:
                    return False
            else:
                return True
```

```
In [9]: def mod_exp(x, y, m):
            """
            Returns the result of (x^y) mod m using repeated squaring
            """
            
            if y == 0:
                return 1
            z = mod_exp(x, y // 2, m)
            if y % 2 == 0:
                return z * z % m
            return x * z *z % m
```

In lecture, we studied Fermat's Little Theorem, which states that if p is a prime number, then for any integer a, the number \(a^p − a\) is an integer multiple of p.

In the notation of modular arithmetic, this is expressed as: \(a^p \equiv a\) (mod \(p\)).

Based on Fermat's Little Theorem, we can come up with a new primality test as follows.

For a randomly-chosen \(a\), where \(1 \leq a \leq p-1\), test if \(a^{(p-1)} \equiv 1\) (mod \(p\)). If the equality does not hold, then \(p\) is composite. If the equality does hold, then we can say that p is a *probable prime*.

Why "probable prime"? If you are really curious, you can read more about the term on Wikipedia (https://en.wikipedia.org/wiki/Probable_prime). We'll get more into this once we hit the probability part of the class

Your task is to implement the function `is_prime_fermat`, which basically conveys the idea described above. You can choose a randomly -- we will give you the code to do that.

```
In [10]: def is_prime_fermat(p):
    """
    Tests if a positive integer p is prime
    using Fermat's Little Theorem
    """

    import random
    a = random.randint(2, p-1)

    # YOUR CODE HERE
    # For modular exponentiation, make sure to use
    # the mod_exp function above
    return mod_exp(a, p-1, p) == 1
```

Test your implementation below. Both tests should print True if your implementation is correct.

In [11]:
```
is_prime_fermat(17) == True
```

Out[11]:
```
True
```

In [12]:
```
is_prime_fermat(24) == False
```

Out[12]:
```
True
```

How about this example? Is \(561\) a prime number? Try it out! If `is_prime_fermat` prints "False" for \(561\), run the code cell again!

In [13]:
```
is_prime_fermat(561)
```

Out[13]:
```
False
```

In [14]:
```
is_prime(561)
```

Out[14]:
```
False
```

Wait a minute! \(561 = 3 \cdot 11 \cdot 17\), so how could it be prime? Is our implementation of `is_prime_fermat` broken?

The answer is no, \(561\) is an example of Carmichael number (https://en.wikipedia.org/wiki/Carmichael_number), which we encourage you to read more on if you are interested. In simple terms, Carmichael number is a composite number \(n\) which satisfies: \(a^n \equiv a\) (mod \(n\)) for all integers \(1<a<n\) for which \(a\) and \(n\) are relatively prime.

This is the reason why, even when the algorithm returns True, we can only conclude that our positive integer \(p\) is only a "probable prime". In EECS 170, you will learn a way to improve this function's chance of returning whether a number is prime or not correctly. (*Hint*: simply do more iterations with different, random \(a\)'s. Can you see why doing so would increase the probability of testing whether a number is prime or not using Fermat's Little Theorem?)

## Question 4: Euler's totient function

Below are sample implementations of the functions `gcd` and `sieve` that you were asked to implement in last week's lab.

In [15]:
```
def gcd(x, y):
    """
    Computes the greatest common divisor between two numbers x and y
    """

    if y == 0:
        return x
    return gcd(y, x % y)
```

In [16]:
```
def sieve(n):
    """
    Return a list of all primes <= n, where n is a positive integer
    """

    lst = range(n+1)
    lst[1] = 0  # 1 is not prime, so we mark it with '0' for future removal
    sqrtn = int(round(n**0.5))
    for i in range(2, sqrtn + 1):  # why do we only need to go up to sqrt(n)
+ 1?
        if lst[i]:
            lst[i*i: n+1: i] = [0] * len(range(i*i, n+1, i))
    return filter(None, lst)  # filter out all 0s from lst
```

Consider the problem of trying to find the value of a tower of iterated exponents modulo some prime number \(p\). We know from the running time of the modular exponentiation algorithm that this can be done quickly for expressions of the form

\[x^x \mathrm{\ mod\ } p \]

Moreover, because of Fermat's Little Theorem (FLT), we know that we can even quickly calculate expressions of the form

\[x^{x^x} \mathrm{\ mod\ } p \]

Here we use FLT to change the exponent to \(x^x \mathrm{\ mod\ } (p - 1)\), which we can then calculate quickly using the modular exponentiation algorithm.

Great! But now what if we want to take this even higher? If our exponent was instead \(x^{x^x} \mathrm{\ mod\ } (p-1)\), what can we do? We can't use Fermat's Little Theorem here because there is no guarantee that \(p-1\) is prime (in fact, unless \(p=3\), it's not). We have to calculate the exponent \(x^x\) without a modulus. If \(x\) is sufficiently large, we'll be working with numbers so big they'll fry your computer (try asking Python for \(\texttt{1000000**1000000}\)).

In order for calculations of this form to be tractable, we need some way to bubble the modulus up through the exponents so we can continue using the modular exponentiation algorithm.

In 1763, Euler generalized Fermat's Little Theorem to not require the modulus to be prime. Before Euler's theorem though, we need to know about Euler's Totient Function.

Euler’s totient function is defined as follows:

\[\phi(n)=\left|\{i:1 \leq i \leq n,\texttt{gcd}(n,i)=1\}\right|\]

In other words, \(\phi(n)\) is the total number of positive integers less than \(n\) which are relatively prime to it, where \(1\) is counted as being relatively prime to all numbers. Since a number less than or equal to and relatively prime to a given number is called a totative, the totient function \(\phi(n)\) can be simply defined as the number of totatives of \(n\). For example, there are eight totatives of \(24\) \((1, 5, 7, 11, 13, 17, 19\), and \(23)\), so \(\phi(24)=8\).

Implement the function `totient`, which returns the number of positive integers less than \(n\) which are relatively prime to it. Use the `gcd` function given above.

In [17]:
```
def totient(n):
    """
    Returns the number of positive integers less than $n$ which are relativel
y prime to it

    YOUR CODE HERE
    """

    return len(filter(lambda i: gcd(i, n) == 1, xrange(1, n)))
```

Test your implementation below. The test should print True if your implementation is correct

In [18]:
```
totient(4) == 2  # 1 and 3 are relatively prime to 4
```

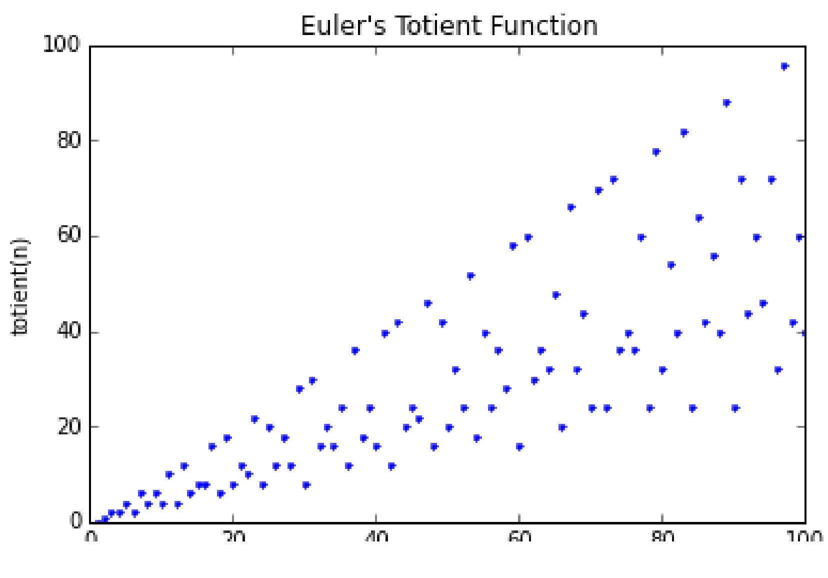
Out[18]:
```
True
```

In [19]:
```
totient(24) == 8
```

Out[19]:
```
True
```

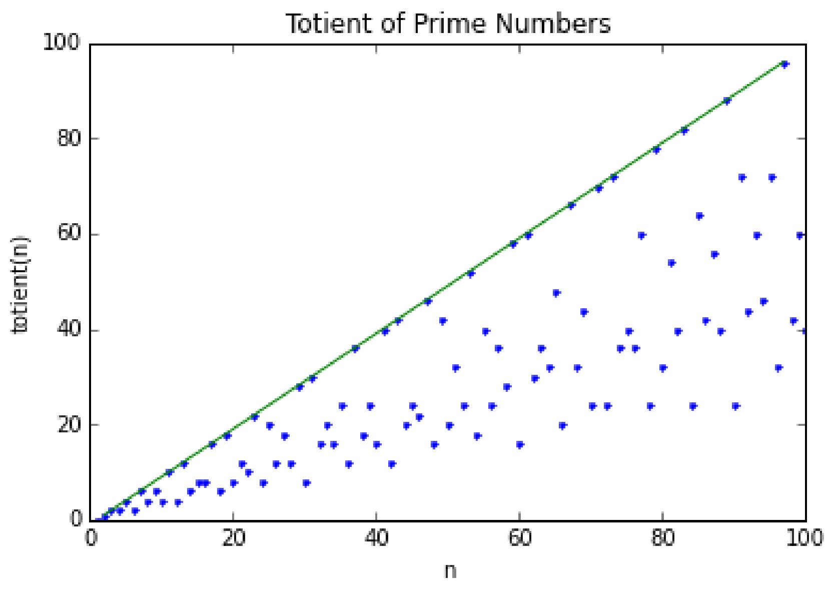
Let's examine some of the behavior of the totient function. Run the cell below to plot some values.

In [20]:
```
plt.plot(xrange(1, 101), map(totient, xrange(1, 101)), '.')
plt.xlabel('n')
plt.ylabel('totient(n)')
plt.title("Euler's Totient Function")
plt.show()
```



It's all over the place! But we notice a few patterns. In particular, all the values of \(\phi(n)\) seem to be bounded by a line. Run the following cell, which overlays on the plot the value of the totient at all primes less than 100.

```
In [21]: primes = sieve(100)
         plt.plot(xrange(1, 101), map(totient, xrange(1, 101)), '.')
         plt.plot(primes, map(totient, primes))
         plt.xlabel('n')
         plt.ylabel('totient(n)')
         plt.title("Totient of Prime Numbers")
         plt.show()
```



Interesting. What can you conclude about the value of \(\phi(n)\) when \(n\) is prime? Why does this make sense?

## Question 5: Euler's theorem

Euler's theorem states that for coprime \(a,n\)

\[a^{\phi(n)} \equiv 1 \ (\mathrm{mod\ } n)\]

This is very similar to Fermat's Little Theorem, but instead of requiring \(n\) to be prime, we only require that it is coprime to \(a\), a looser condition. As a sanity check, verify that this works for prime \(n\).

As with Fermat's Little Theorem, this theorem can be massaged into a more useful form with some algebra.

\[a^b \mathrm{\ mod\ } n \equiv a ^ {b \mathrm{\ mod\ } \phi(n)} \mathrm{\ mod\ } n\].

How did we get to the above form? Try to work out the algebra yourself before continue reading.

Let \(b' = b\) mod \(\phi(n)\). Hence, \(b = m \phi(n) + b'\) for some integer \(m\). Then \(a^b = a^{m \phi(n) + b'} = (a^{\phi(n)})^m \cdot a^{b'} = a^{b'} = a^{b \mathrm{\ mod\ } \phi(n)}\), using Euler's theorem in the next-to-last step.

This is sometimes called blindingly fast exponentiation (http://www.cs.berkeley.edu/~kamil/teaching/fa03/101003.pdf).

Armed with this knowledge, write the function `tower` which, given \(x\) and some prime \(p\), calculates the value of

\[x^{x^{x^x}} \mathrm{\ mod\ } p\]

Hint: break the chain of exponents into three parts, and calculate each part in a bottom-up approach using the formula above. Use the functions `mod_exp` and `totient` defined in previous parts.

In [22]:
```
def tower(x, p):
    """
    Returns the value of x^(x^(x^x)) mod p

    YOUR CODE HERE
    """


    first = mod_exp(x, x, totient(p-1))
    second = mod_exp(x, first, p-1)
    return mod_exp(x, second, p)
```

Alternatively, if you went one step further and applied the formula to the innermost layer, it's perfectly fine. The code above would be modified as follows.

In [23]:
```
def tower2(x, p):
    """
    Returns the value of x^(x^(x^x)) mod p

    YOUR CODE HERE
    """


    first = x % totient(totient(p-1))
    second = mod_exp(x, first, totient(p-1))
    third = mod_exp(x, second, p-1)
    return mod_exp(x, third, p)
```

Test your implementation below. The test should print True if your implementation is correct. Calculate \(21^{21^{21^{21}}} \mathrm{\ mod\ } 101\).

In [24]:
```
tower(21, 101) == 9
```

Out[24]:
```
True
```

In [25]:
```
tower2(21, 101) == 9
```

Out[25]:
```
True
```

## Question 6: Plot polynomials

This last question will introduce you to basic polynomial plotting in Python using Matplotlib.

In [26]:
```
# First, we'll generate some fake data to use
x = np.linspace(0, 10, 50) # 50 evenly-spaced points from 0 to 10
```

```
# Remember, you can always look at the help for linspace
# help(np.linspace)
```

Now let's say we want to plot the line \(y = 2x + 1\). What should we do?

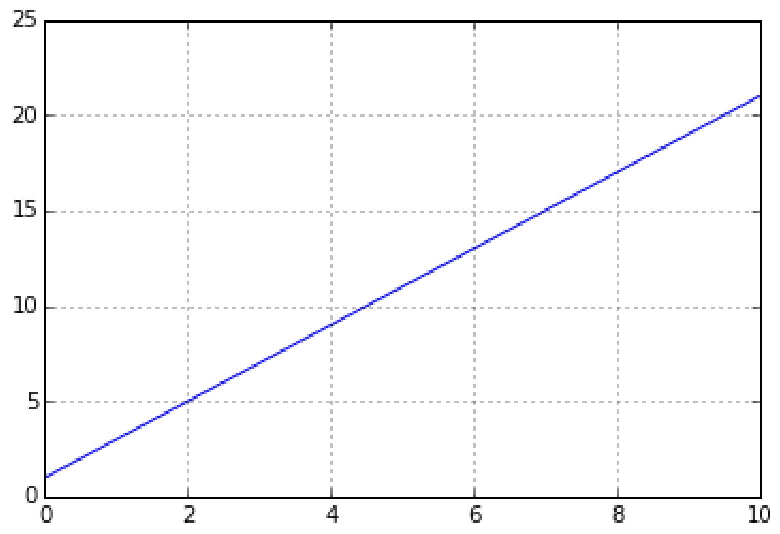
It's exactly how you would guess it...

In [27]:
```
y = 2*x + 1
```

Let's see the output!

In [28]:
```
plt.grid(True)
plt.plot(x, y)
```

Out[28]:
```
[<matplotlib.lines.Line2D at 0xb084daac>]
```



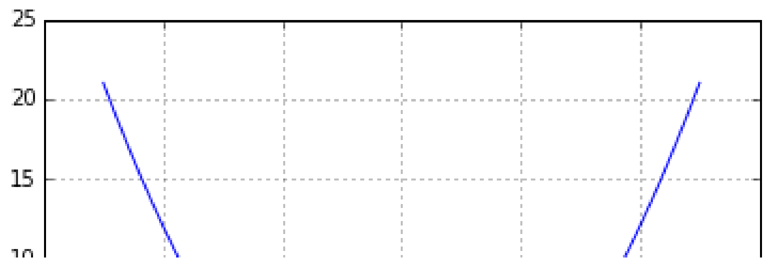
Woo! Isn't that pretty straightforward?

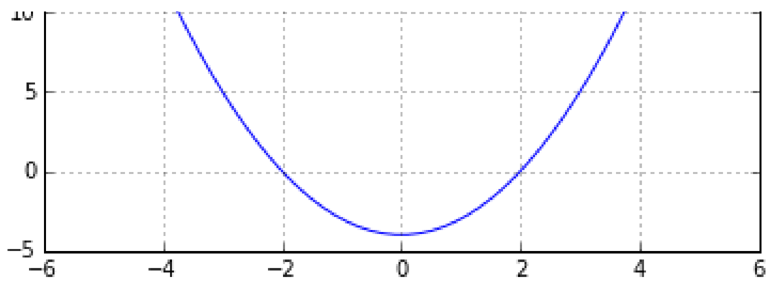
Now it's your turn. Generate \(50\) points between \(-5\) and \(5\), then plot the curve \(y = x^2 - 4\) as seen in Note 7.

In [29]:
```
# YOUR CODE HERE

x = np.linspace(-5, 5, 50)
y = x**2 - 4
plt.grid(True)
plt.plot(x, y)
```

Out[29]:
```
[<matplotlib.lines.Line2D at 0xb090570c>]
```





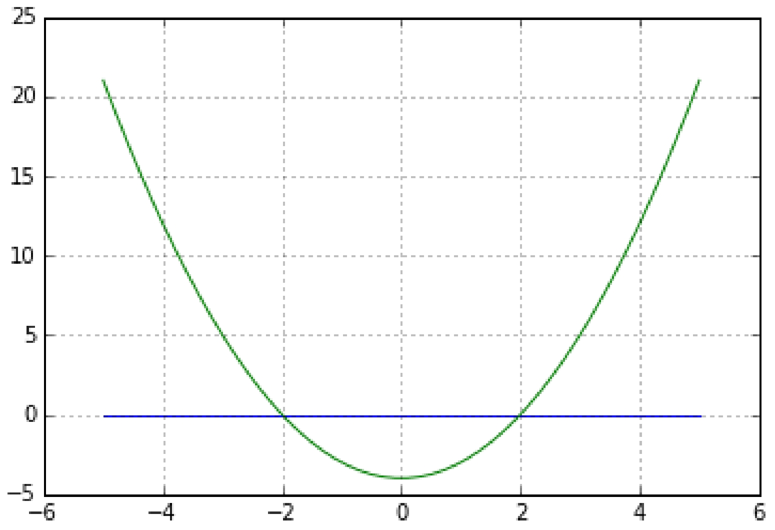
Now we're going to add the x-axis (y = 0) to the plot above. Can you spot the two roots of \(y = x^2 - 4\)?

Hint: all the points on the x-axis has a y-coordinate of 0.

In [30]:
```
# YOUR CODE HERE

x_axis = x * 0
plt.grid(True)
plt.plot(x, x_axis)
plt.plot(x, y)
```

Out[30]:
```
[<matplotlib.lines.Line2D at 0xb0aa2fec>]
```



In future labs, we will explore more powerful features of `matplotlib`, as well as do more fun plotting with polynomials. Stay tuned!

Congratulations! You are done with Virtual Lab 5.

Don't forget to convert this notebook to a pdf document, merge it with your written homework, and submit both the pdf and the code (as a zip file) on glookup.

Reminder: late submissions are NOT accepted. If you have any technical difficulty, resolve it early on or use the provided VM.

Acknowledgments:

- Davis Foote, one of our Readers this semester, creates Questions 4 and 5.