lab4sol

EECS 70

September 25, 2014

1 Virtual Lab 4 Solution: Modular Arithmetic and Primality Testing

EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, September 29th, 2014 at 12pm

Instructions:

- Complete this lab by filling in all of the required functions, marked with `"YOUR CODE HERE"`
- 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!

1.1 Table of Contents

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

IntroductionIn Python, you can perform a lot of powerful modular arithmetic operations. The basic arithmetic operations (+, -, *, /) are all supported. Note that when used with integers, the division operator returns the greatest integer less than the exact result.

The modular reduction operator is represented by the % operator (e.g. `7 % 2` returns 1). This differs from many other languages in that, if the modulus is positive the result is always positive.

Exponentiation can be carried out with the ** symbol.

The `pow()` function can be used for exponentiation if called with two parameters, or efficient modular exponentiation if called with three parameters. `pow(a, b, c)` returns the same result as `(a**b) % c`, but is much more efficient.

Below you will find a few examples using the operators described above.

In [1]:
```
(4 + 7) % 13
```

Out [1]:
```
11
```

In [2]:
```
(5 * 4) % 13
```

Out [2]:
```
7
```

In [3]:
```
7 // 2
```

Out [3]:
```
3
```

In [4]:
```
3 ** 2
```

Out [4]:
```
9
```

In [5]:
```
pow(2, 6, 11)
```

Out [5]:
```
9
```

Question 1: Modular exponentiationImplement the function `mod_exp`, which computes `(x^y) mod m` using **repeated squaring**. **Do NOT use the `pow` function**.

Hint: Try the pseudocode on page 3 of Note 5 if you get stuck. Make sure you understand why this implementation is better than the naive approach.

In [6]:
```
def mod_exp(x, y, m):
    """
    Returns the result of (x^y) mod m using repeated squaring

    YOUR CODE HERE
    """

    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
```

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

In [7]:
```
mod_exp(2, 6, 11) == pow(2, 6, 11)
```

Out [7]:
```
True
```

In [8]:
```
mod_exp(3, 6, 7) == pow(3, 6, 7)
```

Out [8]:
```
True
```

Question 2: GCDImplement the function `gcd`, which computes the greatest common divisor between two numbers.

In [9]:
```
def gcd(x, y):
    """
    Computes the greatest common divisor between two numbers x and y

    YOUR CODE HERE
    """

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

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

In [10]:
```
gcd(6, 4) == 2
```

Out [10]:
```
True
```

In [11]:
```
gcd(10500, 725) == 25
```

```
True
```

Out [11]:

Question 3: Extended GCDImplement the `egcd` function, which 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`.

In [12]:
```
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.

    YOUR CODE HERE
    """

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

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

In [13]:
```
egcd(6, 4) == (2, 1, -1)
```

```
True
```

Out [13]:

In [14]:
```
egcd(16, 10) == (2, 2, -3)
```

```
True
```

Out [14]:

Use `egcd(x, y)` to find the **positive** inverse of `117 mod 103`, of `17947 mod 222`, and of `1812647 mod 1234567`. Report the answers in your homework writeup.

In [15]:
```
egcd(117, 103)[1] + 103
```

```
81
```

Out [15]:

In [16]:
```
egcd(17947, 222)[1]
```

```
19
```

Out [16]:

In [17]:
```
egcd(1812647, 1234567)[1] + 1234567
```

```
710348
```

Out [17]:

Question 4: Naive primality testImplement the function `is_prime`, which simply checks if a number `x` is a prime number. You can assume that `x` is a positive integer. It's okay to go with a naive implementation for this question, we'll look at some more efficient implementations in later questions and/or virtual labs.

In [18]:
```
def is_prime(x):
    """
    Checks if the positive integer x is a prime number

    YOUR CODE HERE
    """

    for i in range(2, x-1):
        if x % i == 0:
            return False
    else:
        return True
```

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

In [19]:
```
is_prime(17) == True
```

```
True
```

Out [19]:

In [20]:
```
is_prime(6) == False
```

```
True
```

Out [20]:

Question 5: Sieve of EratosthenesThe Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2. For more information, check out the relevant Wikipedia article.

Here's a sample execution of the algorithm for primes below 121, taken from Wikipedia.

In [21]:
```
from IPython.display import Image
Image(url='https://upload.wikimedia.org/wikipedia/commons/b/b9/Sieve_of_Eratosthenes_a
```

Out [21]:
```
<IPython.core.display.Image at 0x9442e8c>
```

Implements the function `sieve`, which takes a positive integer `n`, and returns a list of all primes less than or equal to `n`.

Hint: create a list containing all numbers less than `n`, then iteratively remove the multiples of each number remaining in the list. You should not need to use the `is_prime` function implemented above.

In [22]:
```
def sieve(n):
    """
    Return a list of all primes <= n, where n is a positive integer

    YOUR CODE HERE
    """

    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
```

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

In [23]:
```
sieve(3) == [2, 3]
```

Out [23]:
```
True
```

In [24]:
```
sieve(20) == [2, 3, 5, 7, 11, 13, 17, 19]
```

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

Congratulations! You are done with Virtual Lab 4.

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.