lab6sol

October 9, 2014

1 Virtual Lab 6 Solution: Public Key Cryptography and Lagrange Interpolation

1.0.1 EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, October 13th, 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 or code blocks you are required to fill out for each question. Always make sure to double check before you submit.


```
In [10]: %pylab inline
```

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

```
In [40]: import sys
         import math
         import random
         import copy
```

Introduction

In this week's Virtual Lab, we will implement a toy RSA cryptosystem. We will start by generating the public and private keys for RSA using functions that we have implemented in the past two weeks (`mod_exp`,

`egcd`, etc.). Then, we will encrypt and decrypt messages with the RSA function – you will decrypt a secret and tell us in your written homework what that secret is.

Finally, we will learn to carry out Lagrange Interpolation for a degree 2 polynomial. Let's dive in!

Once again, you will find below sample implementations of the functions we asked you to implement in Virtual Lab 4. Feel free to use them when appropriate.

In [41]:
```
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 [42]:
```
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 [43]:
```
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)
```

In [44]:
```
def inverse(x, m):
    """
    Returns the positive multiplicative inverse of x mod m
    """

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

In [45]:
```
def sieve(n):
    """
    Returns 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
```

Question 1: Pairwise coprime

Warmup: Implement the function `pairwise_coprime`, which returns True if every pair of numbers in `lst` is coprime (relatively prime) and False otherwise. You can retrieve every pair in `lst` with the function `combinations(lst, 2)` from the `itertools` module.

```
In [46]: def pairwise_coprime(lst):
             """
             Returns True if every pair of numbers in lst is
             coprime (relatively prime) and False otherwise

             YOUR CODE HERE
             """

             from itertools import combinations
             for i, j in combinations(lst, 2):
                 if gcd(i, j) != 1:
                     return False
             return True
```

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

```
In [47]: pairwise_coprime([2, 3, 5, 7, 11]) == True

Out[47]: True

In [48]: pairwise_coprime([2, 3, 5, 7, 12]) == False  # 2 and 12, 3 and 12 are not coprime

Out[48]: True

In [49]: pairwise_coprime([2]) == True  # no pair, so True by default

Out[49]: True
```

Question 2: Generate RSA keypair

Recall part (c) of last week lab, where we implement the function `is_prime_fermat` to test if a positive integer is a probable prime or is definitely composite. Below you will find an implementation of the function `is_probable_prime`, which uses the Miller-Rabin algorithm to carry out the same task with a very small error rate.

The implementation is quite complex, and do not worry if you don't understand the code below.

```
In [50]: def is_probable_prime(n, k=7):
             """
             Uses the Miller-Rabin algorithm to return True (n is probably prime)
             or False (n is definitely composite)
             """

             if n < 6:  # shortcut the first few cases here...
                 return [False, False, True, True, False, True][n]
             elif n & 1 == 0:  # should be faster than n % 2
                 return False
```

```
    else:
        s, d = 0, n - 1
        while d & 1 == 0:
            s, d = s + 1, d >> 1
            for a in random.sample(xrange(2, min(n - 2, sys.maxint)), min(n - 4, k)):
                x = pow(a, d, n)
                if x != 1 and x + 1 != n:
                    for r in xrange(1, s):
                        x = pow(x, 2, n)
                        if x == 1:
                            return False  # composite for sure
                        elif x == n - 1:
                            a = 0  # so we know loop didn't continue to end
                            break  # could be strong liar, try another a
                    if a:
                        return False  # composite if we reached end of this loop
                return True  # probably prime if reached end of outer loop
```

Here's how the function `is_probable_prime` could be used. Notice how 561 doesn't pass the Miller-Rabin test. :-)

```
In [51]: is_probable_prime(7)
```

```
Out[51]: True
```

```
In [52]: is_probable_prime(561)
```

```
Out[52]: False
```

Below is a simple implementation of a function that generates a pseudorandom prime. This function will help you generate your public and private key pair for RSA.

```
In [53]: def gen_prime(a, b, k=7):
             """
             Generates a pseudo prime number roughly between a and b.

             Raises ValueError if we still fail to find a prime number after
             10 * math.log(x) + 3 tries.
             """

             x = random.randint(a, b)
             for i in range(0, int(10 * math.log(x) + 3)):
                 if is_probable_prime(x, k):
                     return x
                 else:
                     x += 1
             raise ValueError
```

Runs the code cell below multiple times! You should see that it generates a pseudorandom prime roughly between 3 and 100.

```
In [54]: gen_prime(3, 100)
```

```
Out[54]: 11
```

Now it's your turn! Implement the function `gen_key`, which generates a pair of public and private keys for RSA. For simplicity, return a triple `(N, e, d)` in your implementation, where $N = pq$ (`p` and `q` are two large primes), e is relatively prime to $(p-1)(q-1)$, and d is the inverse of e mod $(p-1)(q-1)$. Refer to Page 2 of Lecture Note 6 for more details on how to generate the RSA key pair.

```
In [55]: def gen_key(a, b, k=7):
             """
             Generates public and private keys for RSA encryption.
             Raises ValueError if the function fails to find one.

             Please fill in the three code blocks marked with "YOUR CODE HERE"
             below. Each block requires no more than five lines.
             """

             try:
                 # Finds two large primes p and q, where q needs to be
                 # different than p
                 # YOUR CODE HERE
                 p = gen_prime(a, b, k)
                 q = gen_prime(a, b, k)
                 while q == p:
                     q = gen_prime(a, b, k)
             except:
                 raise ValueError

             # Computes N = p * q, and m = (p - 1) * (q - 1)
             # YOUR CODE HERE
             N = p * q
             m = (p - 1) * (q - 1)

             # Finds e coprime to m. You can use the function
             # `pairwise_coprime` from part (a), or just good ol' `gcd`
             # Then find d, the inverse of e mod (p-1)(q-1)
             # YOUR CODE HERE
             while True:
                 e = random.randint(1, m)
                 if pairwise_coprime([e, m]):
                     break
             d = inverse(e, m)

             # Returns our keypair as a triple
             return (N, e, d)
```

Unfortunately, it's not quite easy to test if your implementation above is correct. We'll know by the time we reach the next question, where you get to actually encrypt/decrypt messages!

Question 3: Encrypt/Decrypt simple messages

Implement the functions `encrypt_integer` and `decrypt_integer`, which encrypts and decrypts a positive integer x. We will generate our RSA keypair using the function `gen_key` you implemented above.

```
In [56]: N, e, d = gen_key(3, 100)  # we will work with small numbers for this question
         print N, e, d
```

```
6499 6253 229
```

```
In [57]: def encrypt_integer(x, N, e):
             """
             Encrypts the message x, where x is a positive integer

             YOUR CODE HERE
             """
```

```
            return mod_exp(x, e, N)
```

```
In [58]: def decrypt_integer(x, N, d):
            """
            Decrypts the cipher x to get a positive integer
            encrypted with `encrypt_integer`

            YOUR CODE HERE
            """

            return mod_exp(x, d, N)
```

Time to encrypt/decrypt something! If your implementations for Q2 and Q3 are both correct, the first two tests below should print True.

```
In [59]: decrypt_integer(encrypt_integer(28, N, e), N, d) == 28
```

```
Out[59]: True
```

```
In [60]: decrypt_integer(encrypt_integer(199, N, e), N, d) == 199
```

```
Out[60]: True
```

```
In [61]: decrypt_integer(encrypt_integer(290000, N, e), N, d) == 290000
```

```
Out[61]: False
```

Wait what? Why does 290000 not work?

The answer can be found on the bottom of Page 2 of Note 6. We need to map our message x into a number between 2 and $N-1$, which we didn't. The largest prime below 100 is 97, and even if we allow p and q to be the same, $N = p \cdot q$ would not be larger than 290000. :-(

Question 4: Encrypt/Decrypt text messages

Q3 was a bit boring, wasn't it? In this part, we will give you the code to convert text messages into lists of integers based on ASCII values, and you will actually get to encrypt or decrypt actual messages.

Belows are some functions that you might find helpful for completing this question. You do not need to understand the implementation, but please read the docstring carefully to make sure you understand when to use each function.

```
In [62]: def string_to_num_list(s):
            """
            Converts a string to a list of integers based on ASCII values
            """

            return [ord(char) for char in s]
```

```
In [63]: def num_list_to_string(l):
            """
            Converts a list of integers to a string based on ASCII values
            """

            return ''.join(map(chr, l))
```

```
In [64]: def num_list_to_blocks(l, n):
            """
            Takes a list of integers (each between 0 and 127), and combines them
            into list of blocks, each of size n, using base 256. If len(L) % n != 0,
```

```
        use some random junk to fill L.
        """

        blocks = []
        to_process = copy.copy(l)
        if len(to_process) % n != 0:
            for i in range(0, n - len(to_process) % n):
                to_process.append(random.randint(32, 126))
        for i in range(0, len(to_process), n):
            block = 0
            for j in range(0, n):
                block += to_process[i + j] << (8 * (n - j - 1))
            blocks.append(block)
        return blocks
```

```
In [65]: def blocks_to_num_list(blocks, n):
        """
        Takes a list of blocks and converts them back into
        a number list. This function is the inverse of num_list_to_blocks
        """

        to_process = copy.copy(blocks)
        num_list = []
        for num_block in to_process:
            inner = []
            for i in range(0, n):
                inner.append(num_block % 256)
                num_block >>= 8
            inner.reverse()
            num_list.extend(inner)
        return num_list
```

Now it's your turn! Once again, implement the `encrypt` and `decrypt` functions, this time for text messages. Both functions should be relatively simple. For `encrypt`, you should convert the text message to a number list, then convert the number list to blocks, then encrypt each block with `mod_exp`, and finally return a list of numbers, where each number is the secret for a corresponding block.

For `decrypt`, do the reverse. The four functions given above should do the trick if you use them correctly.

```
In [66]: N, e, d = gen_key(10 ** 100, 10 ** 101)  # we will work with super large numbers like actual R
         print N
         print e
         print d
```

```
21979458361210240375081691016239511447517524881598754493018204390362360813594942992010588391346532338140
20249251673998884770982196923295403276215867367564486321580389699975479408084177577270644295822879225749
57536837187952065346227738269172409691453843352269260104005942017149601564883985846984769304998447757280
```

```
In [67]: def encrypt(message, N, e, block_size):
        """
        Takes a text message, the public keys (N, e), and block's size,
        and encrypts the text message using RSA

        YOUR CODE HERE
        """
```

```
    num_list = string_to_num_list(message)
    num_blocks = num_list_to_blocks(num_list, block_size)
    return [mod_exp(block, e, N) for block in num_blocks]
```

```
In [68]: def decrypt(secret, N, d, block_size):
    """
    Takes a secret, the modulus N, the private key d, and block's size,
    decrypts the secret back to a text message

    YOUR CODE HERE
    """

    num_blocks = [mod_exp(block, d, N) for block in secret]
    num_list = blocks_to_num_list(num_blocks, block_size)
    return num_list_to_string(num_list)
```

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

```
In [69]: block_size = 14
         message = "70 is awesome!"
         print message

         secret = encrypt(message, N, e, block_size)
         decrypted_message = decrypt(secret, N, d, block_size)
         print decrypted_message
         message == decrypted_message
```

```
70 is awesome!
70 is awesome!
```

```
Out[69]: True
```

70 is indeed awesome, isn't it? :-)

However, this implementation requires your message length to be a multiple of `block_size`, otherwise the decrypted message will have some garbage at the end. Try the following code cell.

```
In [70]: block_size = 8
         message = "70 is awesome!"
         print message

         secret = encrypt(message, N, e, block_size)
         decrypted_message = decrypt(secret, N, d, block_size)
         print decrypted_message
```

```
70 is awesome!
70 is awesome!&U
```

Oops! :-(But that's okay – we have just implemented a toy RSA cryptosystem in this Virtual Lab, which is already a huge accomplishment!

In your written homework, answer the following two questions:

(1) What's the secret of your class log-in (cs70-XYZ) using a block size of 4? If your login only has two letters, treat the last letter as a whitespace. Everyone's answer to this question should be quite different from one another – we will check the answer!

(Since the secret will be quite long, make sure you copy and paste the answer instead of writing it down. If you handwrite your homework, use a software like PDF Escape to edit the PDF and paste the answer in.)

(2) What's the original message of the following secret? You will need to figure out the block size yourself. *Hint*: the length of the original message is 126. Check the numbers from `range(1, len(message))` that divides the message's length.

For both parts, please use the following N, e, d. Make sure you run this code cell below (and don't run `gen_key` again), otherwise your answers will not match up with ours.

Also, don't worry if the pdf conversion command cuts off the numbers since they are too big – that's okay, we'll look at your ipynb file.

```
In [71]: N = 359122472872824396743178033002932297030101609912324913715269158696549777769331867875320354
         e = 22653132660083357264560320661837145188112338982140715301318927512235173339063201503434503
         d = 28283740770418684421439495338658018373533454980350053565645105730752643657694690384791879
```

```
In [72]: super_secret = \
         [63383844792187768156728821640290198101927731287417765268665521466091323877702285007340628012
```

```
In [73]: # YOUR CODE HERE for subpart 1
         block_size = 4
         message = "cs70-ta "
         print message

         secret = encrypt(message, N, e, block_size)
         print secret
```

```
cs70-ta
[24478109325507273539876918387004670482782211487127207860649107783223167710880134807628484669759489837
```

```
In [74]: # YOUR CODE HERE for subpart 2
         for i in range(2, 126/2+1):
             if 126 % i == 0:  # found a number that divides 126
                 decrypted_message = decrypt(super_secret, N, d, i)
                 if len(decrypted_message) == 126:
                     print i, decrypted_message
```

```
18 One of the powerful things about mathematics is thatit provides you with an alternative to your naiv
```

For subpart 2, there are other block sizes that "prints" the correct message, although the message length is incorrect.

Question 5: Degree 2 polynomial interpolation

Let's switch gear to something quite different! This question aims to introduce you to Lagrange interpolation for a simple degree 2 polynomial.

Below is the example data from Note 7.

```
In [75]: x = np.linspace(0, 4, 1001)  # generates a fine grid of random points
         xj = np.array([1., 2., 3.])
         yj = np.array([1., 2., 4.])
```

Recall that the three polynomials Δ_i can be found as follows:

$$\Delta_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}, \tag{1}$$

(2)

$$\Delta_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}, \tag{3}$$

(4)

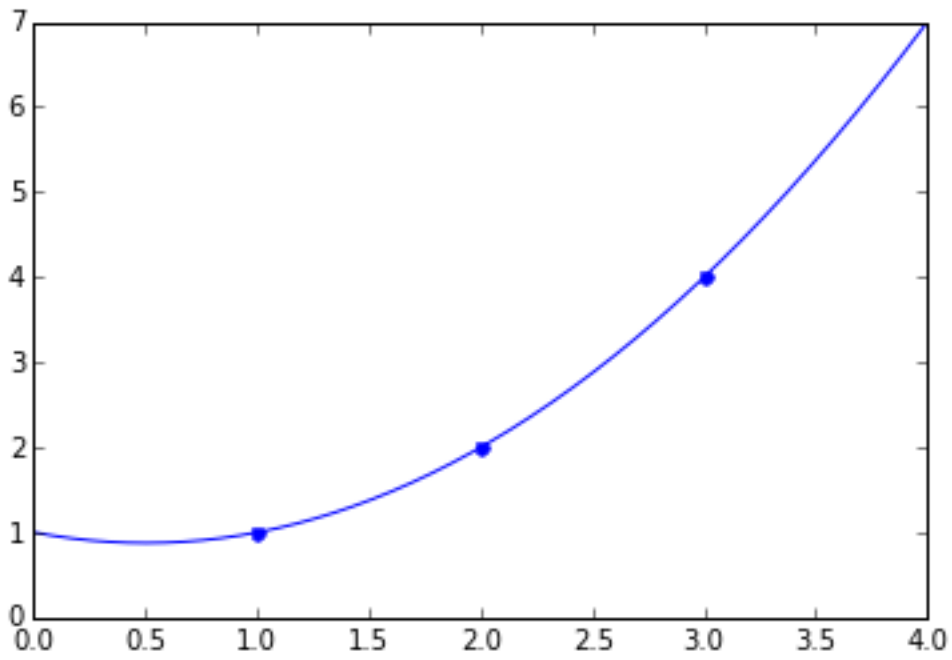
$$\Delta_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \tag{5}$$

To represent Δ in Python, we're going to use D. Implement the function `interpolate_2D` to calculate D_1, D_2, D_3, and return the final polynomial $p(x)$.

```
In [76]: def interpolate_2D(xj, yj):
             """
             Carries out Lagrange interpolation for a degree 2 polynomial.

             xj and yj are list of data points.

             YOUR CODE HERE
             """

             D1 = (x-xj[1])*(x-xj[2]) / ((xj[0]-xj[1])*(xj[0]-xj[2]))
             D2 = (x-xj[0])*(x-xj[2]) / ((xj[1]-xj[0])*(xj[1]-xj[2]))
             D3 = (x-xj[0])*(x-xj[1]) / ((xj[2]-xj[0])*(xj[2]-xj[1]))
             p = yj[0]*D1 + yj[1]*D2 + yj[2]*D3
             return p
```

Let's see how we did with a plot.

```
In [77]: p = interpolate_2D(xj, yj)
         plt.plot(x, p)
         plt.plot(xj, yj, 'b.', markersize=10)

Out[77]: [<matplotlib.lines.Line2D at 0xb09637ec>]
```



Not bad at all. The polynomial does indeed pass through the given three points!

Question 6: Plotting Bar Charts

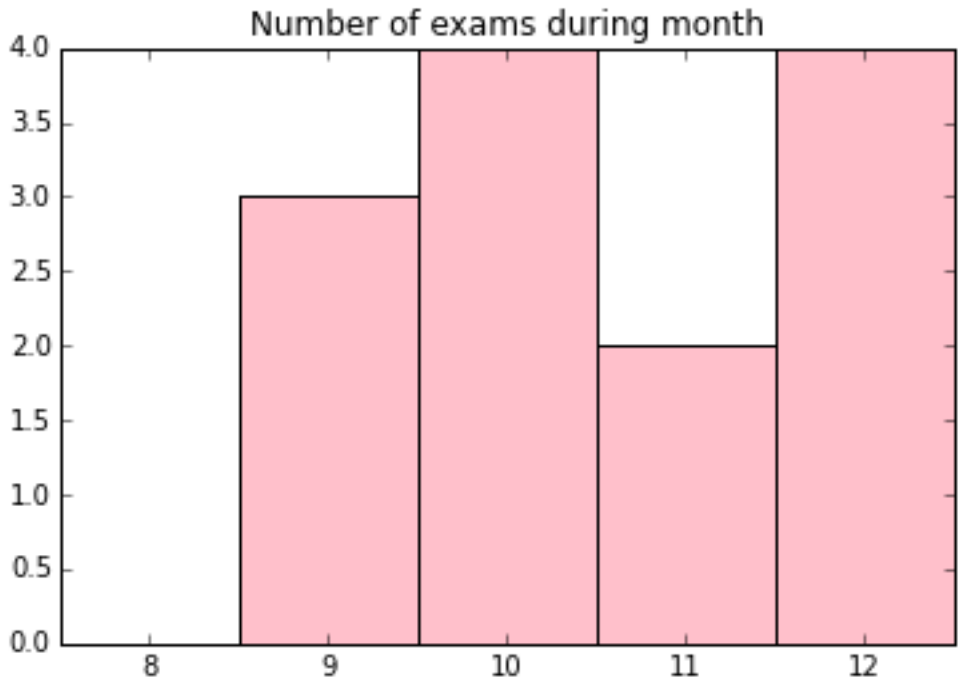
Last week, we learned how to plot a simple curve using Matplotlib. This week, we will learn how to plot a bar chart. By the end of this question, you will plot a bar chart grouping the EECS 70 lecture notes by their last modified months!

But first, let's look at an example. Months are represented by the x-axis, whereas the y-axis represents the number of exams a student typically have during that month. For simplicity, we will only be working with 5 months from August to December.

```
In []: # Take a look at the documentation for np.arange
       help(np.arange)
```

```
In [50]: width = 1
         y = [0, 3, 4, 2, 4]
         x = np.arange(8, 13)
         plt.bar(x, y, width=width, color="pink")
         plt.xticks(x + width*0.5, x)  # center the xticks
         plt.title('Number of exams during month')
```

```
Out[50]: <matplotlib.text.Text at 0xb03b6d6c>
```



Wow, that's pretty straightforward! Just like last time, we define a (NumPy) list that contains the x values, another one that contains the y values, and then call the appropriate Matplotlib function to get the plot we want. Last week it was `plt.plot()`. This week, it's just `plt.bar()`.

Believe it or not, you now know more than enough to complete this question! We will give you the code to recursively fetch the lecture note pdfs on the course website, as well as the Python commands to extract the last modified dates. **You can tackle this problem in any creative way you want**, but here's our recommended approach.

- Group the lecture notes by month (*Hint: use a counter dictionary, where the keys are 1 to 12 (the month), and the values are the number of notes that were last modified during that month*)
- Convert the dictionary into two lists and make sure to preserve the order. For example, there are three lecture notes that were last modified in January, so 1 should be the first value of the list of months, and 3 should be the first value of the list of counts.

- Plot the list values as a bar chart.

First, you will need to download the dataset (if you think you can get the last modified dates straight from the course website, go for it!). Execute the command below, which will save all the lecture note pdfs under the directory `notes`.

```
In []: !wget -r -nH --cut-dirs=2 -A .pdf --no-parent http://www-inst.eecs.berkeley.edu/~cs70/fa14/notes/
```

```
In [13]: !ls notes
```

```
n0.pdf             n12.pdf  n16.pdf  n1.pdf        n4.pdf        n8.pdf
n0.sp13.pdf  n13.pdf  n17.pdf  n20.pdf        n5.pdf        n9.pdf
n10.pdf      n14.pdf  n18.pdf  n2.pdf        n6.pdf        noteShannon.pdf
n11.pdf      n15.pdf  n19.pdf  n3.pdf        n7.pdf
```

```
In [14]: !ls notes | wc -l
```

```
23
```

The command above should return either 23 or 24. At the time of this writing, it's 23, but the Chinese Remainder Theorem note might be added to the course website during the week. :-)

We will now make a dictionary, where the keys are the lecture notes' names, and the values are the last modified months.

```
In [23]: import os, os.path, datetime

         notes = {}
         for f in os.listdir('notes'):
             notes[f] = datetime.datetime.fromtimestamp(os.path.getmtime('notes/' + f)).month

         notes
```

```
Out[23]: {'n0.pdf': 4,
          'n0.sp13.pdf': 5,
          'n1.pdf': 1,
          'n10.pdf': 4,
          'n11.pdf': 4,
          'n12.pdf': 3,
          'n13.pdf': 4,
          'n14.pdf': 4,
          'n15.pdf': 4,
          'n16.pdf': 4,
          'n17.pdf': 4,
          'n18.pdf': 4,
          'n19.pdf': 4,
          'n2.pdf': 1,
          'n20.pdf': 4,
          'n3.pdf': 1,
          'n4.pdf': 9,
          'n5.pdf': 2,
          'n6.pdf': 2,
          'n7.pdf': 2,
          'n8.pdf': 3,
          'n9.pdf': 3,
          'noteShannon.pdf': 4}
```

The initial dataset is now complete! The rest of the work will be yours. Be as creative as you want, or you can follow our recommended approach above. Feel free to add axes' labels, the count above each bar, etc. to your heart's content.

To get an idea of what we expect, please take a look at the homework pdf to see what the final plot should look like.

In [30]:
```
# Step 1 -- Convert the `notes` dictionary into a counter
# Feel free to use advanced Python data structures like Counter, defaultdict, etc.
# YOUR CODE HERE

from collections import defaultdict

counter = defaultdict(int)
for month in notes.values():
    counter[month] += 1

counter
```

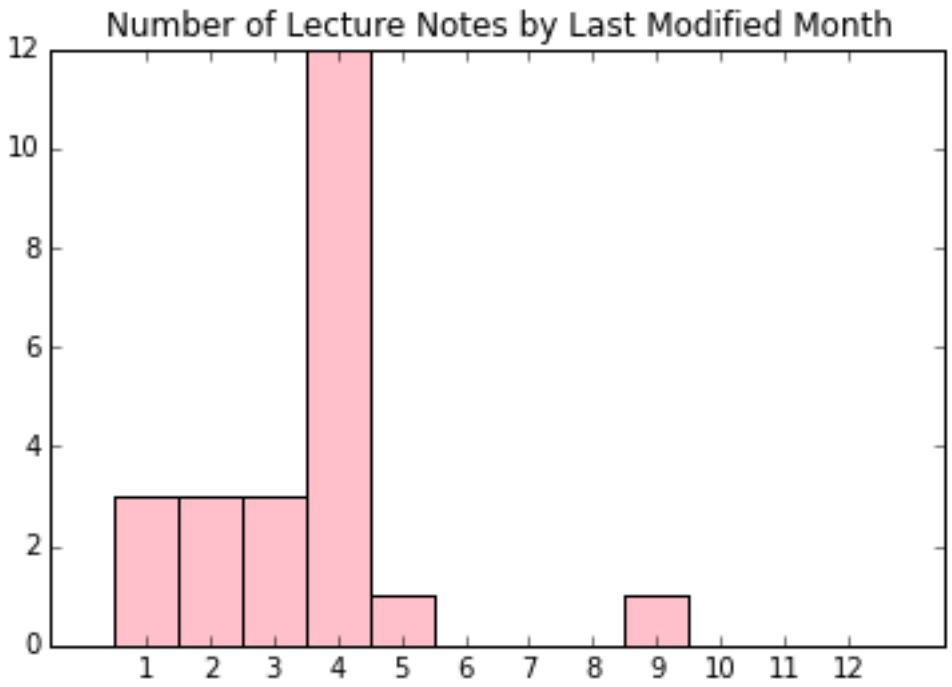
Out[30]:
```
defaultdict(<type 'int'>, {1: 3, 2: 3, 3: 3, 4: 12, 5: 1, 9: 1})
```

In [46]:
```
# Step 2 -- Convert the counter into two lists while preserving the order
# Hint: you can use the `sorted()` function in Python
# https://docs.python.org/2/library/functions.html#sorted
# YOUR CODE HERE

labels = np.arange(1, 13)
values = [counter[label] for label in labels]

values
```

Out[46]:
```
[3, 3, 3, 12, 1, 0, 0, 0, 1, 0, 0, 0]
```

In [52]:
```
# Step 3 -- Plot the bar chart!
# YOUR CODE HERE

width = 1
plt.bar(labels, values, width=width, color='pink')
plt.xticks(labels + width*0.5, labels)
plt.title('Number of Lecture Notes by Last Modified Month')
```

Out[52]:
```
<matplotlib.text.Text at 0xb0302d4c>
```



Congratulations! You are done with Virtual Lab 6.

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.