CS 105 - Lab 11

Today's Objectives

✓ Prime Number Generation
✓ Implement Diffie-Hellman Key Exchange
✓ Implement RSA Encryption

Part 1: Diffie-Hellman Key Exchange

In class you learned about the Diffie-Hellman-Merkle key exchange protocol, and how it can be used to let 2 people come up with a shared secret key without actually sending the key itself over the network. The process can be summarized with paint:

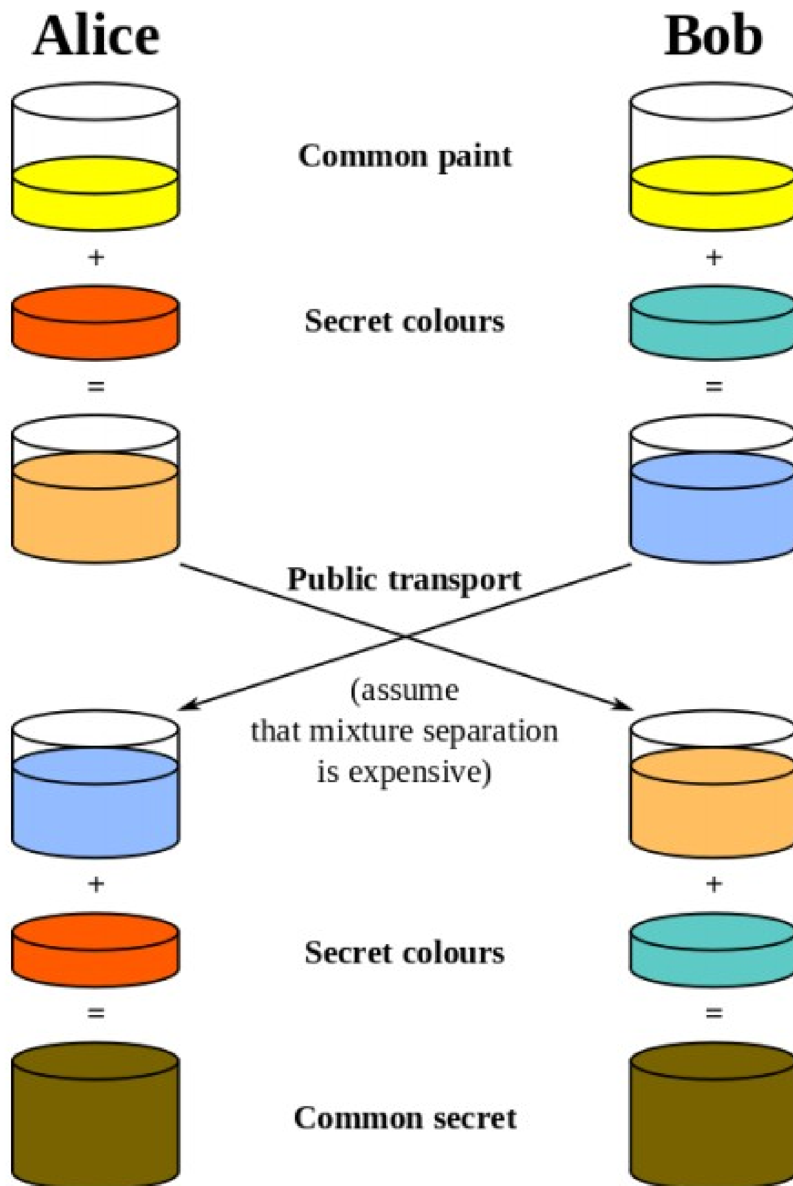

Figure 11. The Diffie-Hellman key exchange imagined as colors.

Image credit: Dr. Allen, Furman University Computer Science

What you're going to be doing is creating a Python version of the key exchange following the same principles outlined in your class handout. We'll be following the following algorithm:

- Alice part 1
  - Choose prime `p`
  - Choose prime `q`
  - Choose secret value `a`
  - Mix the "common paint" `p` and `q` with "secret color" `a` to get mixture `A = (q ** a) % p`
  - Send `p`, `q`, and `A` to Bob.
- Bob's part
  - With `A`, `p`, and `q`:
  - Choose a secret value `b`
  - Compute `B = (q ** b) % p`
  - Computer Bob's K value K_B as `(A ** b) % p`
  - Send `B` back to Alice.
- Alice Part 2
  - Compute Alice's K value K_A `= (B ** a) % p`

The numbers a, b, and K are never transmitted, Alice and Bob have just enough information to independently compute K, and an eavesdropper doesn't know a or b.

We'll start out with *prime number generation* so that we can choose `p` and `q`. Remember that prime numbers only have two factors: *1 and themselves*. Suppose we want to find out if a number, `n`, is prime. If `n` doesn't divide evenly into anything less than it besides 1, it should be prime.

Start up Spyder and create a new function, `is_prime(n)`, which returns `True` if the given number `n` is prime and returns `False` if `n` isn't prime. Test your function with the code below.

```
def is_prime(n):
    # finish this function!

primes = []

for i in range(1, 100):
    if is_prime(i):
        primes.append(i)

print(primes)
```

To make sure you're doing it right, change the `print(primes)` statement to `print(primes[8:12])` and write down the 4 numbers you get. We'll check 'em when we check you off!

Now that we have a prime number generator, let's extend it. Write a function, `get_random_prime(limit)`, which will generate a list of primes from 1 to `limit` and return a random prime from the list. Look up the `random.choice()` function to learn about how to get a random item from a list, or look at an example below:

```
>>> import random
>>> seq = [1,2,3,4,5]
>>> random.choice(seq)
1
```

```
>>> random.choice(seq)
1
>>> random.choice(seq)
3
>>> random.choice(seq)
5
```

Hint: the code provided in the first exercise is great at generating the list of primes. You might want to adapt it to work for you.

```
import  random

def is_prime(n):
    # what you wrote last  time

def  get_random_prime(limit):
    # finish this function!

print(get_random_prime(100))
print(get_random_prime(100))
```

Run your program and write down the two numbers that it gives you. Of course, they'll be random numbers but as long as they're both primes from 1 to 100 you're good.

Alright, you're ready to finish the program!

```
import  random

def is_prime(n):
    # same as last time

def  get_random_prime(limit):
    # same as last time

# Generate 2 random  primes


print('Primes: p:', p,  'q:',  q)

# Input a secret integer a
a =

# Compute A per the algorithm
A =
print('A is', A)

# Input a secret integer  b
```

```
b = 

# Compute B per the algorithm
B = 
print('B is', B)

# Compute Bob's K
KB = 

# Compute Alice's K
KA = 

print('K-Values:')
print('Alice:', KA)
print('Bob:   ', KB)

if KA == KB:
    print('The shared secrets  match!')
else:
    print("The shared secrets don't  match!")
```

Part 2: the RSA cryptosystem

While symmetric key cryptography (everyone has the same key) is fun and easy. It proposes a lot of issues when it comes with dealing with confidentiality when dealing with people you have never met or are in able to safely and easily transfer a symmetric key to. For instance, if I want to buy a book from Amazon.com. All my private information including credit card information *needs* to be secure or I will not do business with you. Also, I do not have time to go met someone to transfer a symmetric key. Finally, It's very very hard for Amazon to come up with unique, non-reusable symmetric keys for every transaction.

So what do we do? Asymmetric cryptography.

Simply put, asymmetric cryptography allows two different parties to create private and public numbers and transfer information confidentially over the Internet securely. Have you ever noticed the padlock symbol on browsers when you are on certain web sites or how some website URLs start with HTTPS rather than HTTP? That indicates that your website connection is using SSL which implements asymmetric cryptography and even that algorithm you are about to attempt to code in Python.

The most famous and popular asymmetric cryptographic scheme is RSA.

What follows is a straightforward mathematical description of the mechanics of RSA encryption and decryption.

1. Alice picks two giant prime numbers, p and q. The primes should be enormous, but for simplicity we assume that Alice chooses p = 17, q = 11. She must keep these numbers secret.

2. Alice multiplies them together to get another number, N. In this case N = 187. She now picks another number e, and in this case she chooses e = 7.

(e and (p - 1) x (q - 1) should be relatively prime, but this is a technicality.)

3. Alice can now publish e and N in something akin to a telephone directory. Since these two numbers are necessary for encryption, they must be available to anybody who might want to encrypt a message to Alice. Together these numbers are called the public key. (As well as being part of Alice's public key, e could also be part of everybody else's public key. However, everybody must have a different value of N, which depends on their choice of p and q.)

4. To encrypt a message, the message must first be converted into a number, M. For example, a word is changed into ASCII binary digits, and the binary digits can be considered as a decimal number. M is then encrypted to give

the ciphertext, C, according to the formula

$C = M^e \pmod{N}$

5. Imagine that Bob wants to send Alice a simple kiss: just the letter X. In ASCII this is represented by 1011000, which is equivalent to 88 in decimal. So, M = 88.

6. To encrypt this message, Bob begins by looking up Alice's public key, and discovers that N = 187 and e = 7. This provides him with the encryption formula required to encrypt messages to Alice. With M = 88, the formula gives

$C = 88^7 \pmod{187}$

7. Working this out directly on a calculator is not straightforward, because the display cannot cope with such large numbers. However, there is a neat trick for calculating exponentials in modular arithmetic. We know that, since 7 = 4 + 2 + 1,

$$88^7 \pmod{187} = [88^4 \pmod{187} \times 88^2 \pmod{187} \times 88^1 \pmod{187}] \pmod{187}$$

$$88^1 = 88 = 88 \pmod{187}$$

$$88^2 = 7{,}744 = 77 \pmod{187}$$

$$88^4 = 59{,}969{,}536 = 132 \pmod{187}$$

$$88^7 = 88^4 \times 88^2 \times 88^1 = 88 \times 77 \times 132 = 894{,}432 = 11 \pmod{187}$$

Bob now sends the ciphertext, C = 11, to Alice.

8. We know that exponentials in modular arithmetic are one-way functions, so it is very difficult to work backward from C = 11 and recover the original message, M. Hence, Eve cannot decipher the message.

9. However, Alice can decipher the message because she has some special information: she knows the values of p and q. She calculates a special number, d, the decryption key, otherwise known as her private key. The number d is calculated according to the following formula

$$e \times d = 1 \pmod{(p-1) \times (q-1)}$$

$$7 \times d = 1 \pmod{16 \times 10}$$

$$7 \times d = 1 \pmod{160}$$

$$d = 23$$

(Deducing the value of d is not straightforward, but a technique known as Euclid's algorithm allows Alice to find d quickly and easily.)

10. To decrypt the message, Alice must simply use the following formula,

$$M = C^d \bmod (N)$$

$$M = C^d \bmod (187)$$

$$M = 11^{23} \bmod (187)$$

$$M = [11^1 \bmod (187) \times 11^2 \bmod (187) \times 11^4 \bmod (187) \times 11^{16} \bmod (187)] \pmod{187}$$

$$M = 11 \times 121 \times 55 \times 154 \pmod{187}$$

```
M = 88 = X in ASCII
```

Rivest, **S**hamir, and **A**dleman had created a special one-way function, one that could be reversed only by someone with access to privileged information, namely the values of p and q. Each function can be personalized by choosing p and q, which multiply together to give N. The function allows everybody to encrypt messages to a particular person by using that person's choice of N, but only the intended recipient can decrypt the message because the recipient is the only person who knows p and q, and hence the only person who knows the decryption key, d.

Finish the Python representation of the algorithm below!

```
# rsa.py - Let's practice the RSA encryption algorithm.

import random

# We need to write a helper function that allows us to determine if two
# integers are relatively prime.  In other words, the only  positive
# integer that divides both a and b is 1.
def relatively_prime(a,  b):
    # First, determine the lesser of a and b and put it in a variable
    if
        lesser =
    else:
        lesser =

    # For all i from 2 up to the lesser, see if this number i divides
    # both a and b.  If so, we have found that the numbers a and b  are
    # not relatively prime.
    found_common_divisor = False
    i = 2
    while i <= lesser:
        if
            found_common_divisor = True
            break
        i += 1

    if found_common_divisor  == True:
        return False
    else:
        return True

# 1.  Choose secret and distinct primes p  and q.  We assume they  are prime!
def  is_prime(n):
    # paste this in from  before

def  get_random_prime(limit):
    # paste this in from  before

p = get_random_prime(100)
q = get_random_prime(100)

# 2.  Let N = pq and let M =  (p-1)(q-1).
N =
M =

# 3.  Choose public encryption key e, which is a number < M  and
```

```
#     relatively prime to M.
#     Let's show the user a list of valid candidates from which to choose.

print("Here is a list of possible values you can choose for e:\n")
i = 1
while i < M:
    if
        print(i)
    i += 1

e = int(input("Which value would you like for public encryption  key e?  "))

# 4.  Message is x.  Sender needs to transmit y = x^e  mod N.

x = int(input("Enter numerical  message x:  "))
y =

print("The message you send is y = ",  y)

# 5.  Choose private decryption key d, where ed mod M =  1.
#     Let's show the user a list of possible choices for  d.

print("Here is a list of possible values you can choose for d:\n")
L = []
i = 1
while i < M:
    if
        print(i)
    i += 1

d = int(input("Which value would you like for private decryption  key d?  "))

# 6.  Recipient computes z = y^d mod N, which should equal original  message x.

z =

print("Recipient computes message z = ",  z)

if x == z:
    print("Good news!  As expected, x and z  match.")
else:
    print("Uh-oh!  Something is wrong with  the implementation.  x != z.")
```

Run the program. You'll be prompted for an encryption key and a number to encrypt. Notice that if x is too large, then the final result (z) is incorrect. Why? What is the largest value of x we should input?