Part I

Labs

Lab 1

Public Key Cryptography

Lab Objective: *Implement the RSA cryptosystem as an example of public key cryptography.*

A *public key cryptosystem* uses separate keys for encryption and decryption. If Alice wishes to send Bob a message, she encrypts it using Bob's *public key*, which is available to everyone. Then Bob decrypts it with his *private key*, which only he knows. As long as it is difficult to find the private key from the public key, this is a secure system.

As an analogy, consider a locked box with two kinds of keys. One kind of key, called the public key, is available to anyone and can lock the box but not unlock it. The other kind of key, called the private key, is only available to one person and can unlock the box. To send a message to someone with the private key, the sender will put the message in the box and lock it. Since the only person that can unlock the box is the recipient, the message is safe until it arrives. This is how public key cryptography works.

One of the oldest and most popular public key cryptosystems is RSA cryptography.

The RSA system

Suppose Alice wants to receive secret messages using RSA. To do so, she first needs to generate a public key (for encryption) and a private key (for decryption). Alice does this by choosing two prime numbers, p and q, and setting $n = pq$. Then she sets $\phi(n) = (p-1)(q-1)$.[1] Alice chooses her encryption exponent to be some integer e that is relatively prime to $\phi(n)$. Then she uses the Extended Euclidean Algorithm to find d' such that $ed' + \phi(n)x' = 1$, and adds or subtracts multiples of $\phi(n)$ until $d = d' + k\phi(n)$ is between 0 and $\phi(n)$. Alice publishes her public key (e, n) for the world to see and keeps her private key (d, n) a secret.

[1] The function $\phi : \mathbb{Z} \to \mathbb{N}$ is called the *Euler phi function*. In general $\phi(n)$ is the number of positive integers less than n that are relatively prime to n.

Now imagine Bob wants to send Alice a message, which he represents as an integer $m < n$ (say, using the A=01, B=02 scheme). Bob computes

$$c \equiv m^e \pmod{n}$$

and sends the ciphertext c to Alice.

To decrypt the message, Alice computes

$$m' \equiv c^d \pmod{n}.$$

At this point, she uses the following theorem, which is easily proved with a combination of ring and group theory.

Theorem 1.1. *For any integers m and n such that n does not divide m, the following equality holds:*

$$m^{\phi(n)} \equiv 1 \pmod{n}.$$

Then Alice concludes that

$$m' = c^d \equiv m^{ed} = m^{\phi(n)(ek-x')+1} \equiv m \pmod{n}.$$

Now Alice can read Bob's message.

As an example, let us encrypt and decrypt the message SECRET=190503180520. First we define p, q, n, and $\phi(n)$.

```
>>> p = 1000003
>>> q = 608609
>>> n = p*q
>>> phi_n = (p-1)*(q-1)
```

Now we choose an encryption exponent $e = 1234567891$ and compute $d = 589492582555$ using the Extended Euclidean Algorithm.

```
>>> e = 1234567891
>>> d = 589492582555
```

Finally we are ready to encrypt the message. Note that $m < n$. If this were not the case, we would need to break up m into shorter pieces. Also, we force m to be a `long` integer so that the exponentiation operation does not overflow. The function `pow(a, b, n)` computes $a^b \pmod{n}$.

```
>>> m = long(190503180520)
>>> c = pow(m, e, n)
```

We decrypt the message by raising c to the d^{th} power, modulo n.

```
>>> m == pow(c, d, n)
True
```

Logistical considerations

The cryptosystem described is not particularly easy to use, since the message must be converted to an integer and back again by hand. The module `rsa_tools` contains some functions to fix this problem. The function `string_to_int()` turns any string into an integer (using a mapping more complicated than $A = 01, B = 02$), and the function `int_to_string()` changes it back again.

```
>>> import rsa_tools as r
>>> r.string_to_int('SECRET')
91556947314004
>>> r.int_to_string(91556947314004)
'SECRET'
```

At this point we have a problem, because the message 91556947314004 is larger than $n = 608610825827$. We can only use RSA to encrypt messages that are smaller than n. In fact, the function `string_size()` provided in the `rsa_tools` tells us the maximum number of characters we can encrypt with this choice of n.

```
>>> r.string_size(608610825827)
4
```

The function `partition()`, also in the `rsa_tools` will break our message into pieces of length 4. We specify the "fill value" `x` that will be used to make all pieces the same length.

```
>>> r.partition('SECRET', 4, 'x')
['SECR', 'ETxx']
```

Now we can proceed to encrypt and decrypt the strings `'SECR'` and `'ETxx'` as before.

Problem 1. Write a class called `myRSA` that can generate keys, encrypt messages, and decrypt messages.

Write a method called `generate_keys` that accepts a pair of primes and an encryption exponent and sets the `public_key` and `_private_key` attributes. (Starting `_private_key` with an underscore hides the attribute from the user.)

Also include an `encrypt` method that accepts a string and encrypts it, using `public_key` as the default encryption key. If a different public key is provided, then the message should be encrypted with the provided key.

Finally, write a `decrypt` method that decrypts a message with `_private_key`.

Use methods from the provided `rsa_tools.py` module to convert strings to ints and vice-versa.

Security of RSA

Suppose an enemy Eve wants to read the message that Bob sent to Alice. Like Bob, she has access to Alice's public key (e, n). Let's assume she has also intercepted the ciphertext c.

One way for Eve to read Bob's message is to directly find m such that $m^e \equiv c \pmod{n}$. Such a computation is known as taking a *discrete logarithm*. When n is very large, this computation is essentially impossible.

Another option is for Eve to compute (d, n) and then find c^d. However, computing d means computing $\phi(n)$. Computing $\phi(n)$ from n directly requires factoring n. When n has hundreds of digits, finding its factors with known algorithms can take years.

Thus, the security of RSA depends on selecting large primes so that n has many digits.

Making RSA Secure

Since the security of RSA depends on the use of large prime numbers for p and q, we need a fast way to find such numbers for RSA to be a practical cryptosystem. It is very slow to check that a large number is prime by finding its factors. In fact, it is the difficulty of factoring that makes RSA secure.

However, there exist fast algorithms that determine when a number is "probably prime." One such algorithm comes from a special case of Theorem **??** known as *Fermat's Little Theorem*:

$$a^{p-1} \equiv 1 \pmod{p}$$

for all primes p and integers a that are not divisible by p. Therefore, to check if a fixed number p is likely a prime, we can compute $a^{p-1} \pmod{p}$ for several values of a. If we ever get something different than 1, then we know that p is composite. The value of a that proved p was composite is called a *witness number*. In fact, the converse is true: if $a^{p-1} \equiv 1 \pmod{p}$ for every $a < p$ then p is prime.

With a few exceptions, if p is composite then more than half the integers in $[2, p-1]$ are witness numbers. Thus, if we run the above test on p just a few times and don't find a witness number, there is a high probability that p is prime. This test is called Fermat's test for primality.

In practice, a probabilistic algorithm like Fermat's test is used to identify numbers that have a high chance of being prime. Then, deterministic algorithms are used to verify the primality of a candidate.

Problem 2. Write a function called `is_prime` that implements Fermat's test for primality. Run the test at most five times, using integers randomly chosen from $[2, n-1]$ as possible witnesses. If a witness number is found, return the number of tries it took to find it. If no witness number is found after five tries, return 0.

For most composite values of n, if you call `is_prime(n)` one hundred times, you should expect to get 0 at most five times. However, some composite numbers do not follow this rule. For example, how many times do you have to call `is_prime()` on 340561 to get an answer of 0?

PyCrypto: RSA in Python (Optional)

PyCrypto is a professional implementation of RSA in python. The package is called `Crypto`, and can be downloaded here. This library contains many random number generators and encryption classes. Many programs use PyCrypto for their security needs.

Warning

Make certain that you are using the latest version of PyCrypto. Security software is updated often to fix security vulnerablilities and bugs. The current version of PyCrypto at the time of writing is version 2.6.1.

The RSA module in PyCrypto is located in the PublicKey module. The library allows us to explicity construct a key, or generate a key automatically.

```
>>> from Crypto.PublicKey import RSA

# generate a 2048-bit RSA key
>>> keypair = RSA.generate(2048)

# Save the public key as a string for distribution.
>>> publickey = keypair.publickey()
>>> share_this = publickey.exportkey()
```

The `share_this` variable may be distributed as a public key. Another user may then import it and use it to encrypt a message.

```
>>> encrypter = RSA.importKey(share_this)
>>> encoded_message = encrypter.encrypt('abcde', 2048)
>>> encoded_message
('\xb1\t\xc6L\xd1u\x80.C@\x07$#\x8e\xca\x8a\x05*\xdf\x1f.N\xa9\x80
\x08\xcb*8~7\x1d\x87&&Ke\xd5\xed_H\xb9\xd0x\xac!\xf3\xa9\xdc\xbfy5
s\x92\x8d\x15\xf7vY\x99G\xb6\x03j[\xa3\xc6\x92a\n\x91\x08N\xbc\xe4
\xcd\xe2\x9b\xeb\x1eT\xe5\xef\x96\x83\x10\xb7\x0c\xd1\x9bK]z\xa5!\
xcc\xe0/\xd3L\xd4\xa9xx?*\xfb\xf6\xcaM8\xe6\x9d\xd4u\xbd\xda\xa8tf
X\x02\xfa\xff\x99\xff\xbb"\xd1\x87\'\xb9d\x1c\x1b\x9fcWd\x83\xea}\
x1f\xff\xd3\x9b0\x8e\x0f\x91\n\x16\r\r\xa5\xa5S\xafw\x07N`$\x9c]\x
ac\x96\xe3\x80l\xc9\xe5\xe4Od\xa5\t>\x16j\xa1\xb9\x9c5\xc0\xfe\xe3
\xe5i\xcd\xaf\xdc\xcad\x82\x10u\x91\xa0"\xcf\xe3A\x11\x82\x87\xb9\
xdf\xd7\x86\x87\xff\x11#-\xb5!Q)\xf7\xf1a\x94\xb3e?\xd0\x96W4\xb4\
xca\xcf\x18\xd1I\xcd\x1dl\xd7\xe0Y\xdf}F\x93\x92\xe1(d\xcc\xdc\xa7
 x\xc5`',)

>>> keypair.decrypt(encoded_message)
'abcde'
```

The RSA encryption and decryption methods on these keys are textbook approaches.

Problem 3. Write a new RSA class called `PyCrypto` that acts like the `myRSA` class, but implement it with methods from PyCrypto's `RSA` package.

Store both of your RSA keys in a single attribute called `_keypair`, and store a sharable string representation of the public key in the `public_key` attribute. Initialize `_keypair` and `public_key` in the constructor.

The `encrypt` method accepts a string and encrypts it, using `_keypair` as the default encryption key. If the string representation of a different public key is provided, then the message should be encrypted with the provided key.

The `decrypt` method decrypts a message using `_keypair`.

With a partner, test your classes by exchanging public keys and sending secret messages.

Warning

PyCrypto is a very powerful tool and should not be used casually. Something about being imprisoned and/or exiled if you leak NSA secrets.

Lab 2

Data Structures I

Lab Objective: *Learn to implement basic data structures.*

Introduction

Analyzing and manipulating data are essential skills in scientific computing. Storing, retrieving, and manipulating data take time. As a dataset grows, so does the amount of time it takes to access and manipulate it. The structure of how the data are stored determines how efficiently the data may be processed.

In this lab we will begin to study data structures. Data structures are objects for organizing data. There are diverse data structures, each with specific strengths and weaknesses. For example, some data structures take a long time to build, but once built their data are quickly accessible. Others are built quickly, but are not as efficiently accessible. Different applications will require different structures.

Nodes

Recall that some built-in data types for Python are booleans, strings, floats, and integers. Most data in applications will take one of these forms. However, as the size of a dataset increases, these types prove inefficient. Data structures will use *nodes* to overcome these inefficiencies.

Think of data as several types of objects that need to be stored in a warehouse. Then a node is a standard size box that can hold all the different types of objects. For example, suppose the warehouse needs to store lamps of various sizes. Rather than trying to stack lamps of different shapes on top of each other efficiently, it is preferable to put them in the boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier. This analogy extends to data structures.

A `Node` class is usually simple. In Python, the data in the Node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure. The data structure links the nodes together in a way that is efficient for its particular application.

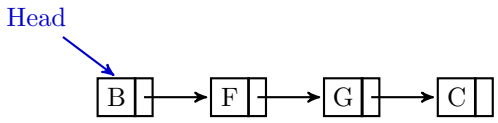

Figure 2.1: Singly-linked List

```
# Location: Node.py

class Node(object):
    """A Node class for storing data."""
    def __init__(self, data):
        """Construct a new node that stores some data."""
        self.data = data
```

```
# Import the Node class from Node.py
>>> from Node import Node

# Create some nodes.  Note that any data type may be stored.
>>> int_node = Node(1)
>>> str_node = Node('abc')
>>> lis_node = Node([1,'abc'])

# Access a nodes data.
>>> lis_node.data
[1, 'abc']
```

Problem 1. Add a new class to `Node.py` called `StrNode` that only accepts strings as data. Use inheritance and call the `Node` class constructor in the constructor for `StrNode` (Hint: Use the `type` built-in function).

Implement the `__str__` magic method in the `Node` class so that it returns a string representation of its data.

Linked Lists

A linked list is a simple data type that chains nodes together. Each node instance in a linked list stores a reference to the next link in the chain. A linked list class also stores a reference to the first node in the chain, called the head. See Figure 2.1.

```
# Location: Node.py

class LinkedListNode(Node):
    """A Node class for linked lists. Inherits from the 'Node' class.
    Contains a reference to the next node in the list.
    """
    def __init__(self, data):
        """Construct a Node and add an attribute for
```

Figure 2.2: Include a picture of the `add_node` method and describe it.

```
        the next node in the list.
        """
        Node.__init__(self, data)
        self.next = None
```

A basic implementation of a linked list will have a constructor and a method for adding new nodes to the end of the list. To get to the end of the list, start at the `head` of the list. Then traverse the list by going from node to node until the end is reached. Then, set the `next` attribute of the last node to be the new node. This is done in the following class. See Figure 2.2 for an illustration.

```
# Location: Lab4_Spec.py
from Node import LinkedListNode as Node

class LinkedList(object):
    """A class for creating linked list objects."""

    def __init__(self):
        """Creates a new linked list.
        Create the head attribute and set it
        to None since the list is empty
        """
        self.head = None

    def add_node(self, data):
        """Create a new Node containing the data
        and add it to the end of the list
        """

        new_node = Node(data)
        if self.head is None:
            # If the list is empty, point the head attribute to the new node.
            self.head = new_node
        else:
            # If the list is not empty, traverse the list and place the new_node ↩
                 at the end.
            current_node = self.head
            while current_node.next is not None:
                # This moves the current node to the next node if it is not empty
                current_node = current_node.next

            current_node.next = new_node
```

Problem 2. Implement the `__str__` method for the `LinkedList` class so that when a `LinkedList` instance is printed, its output matches that of a Python list.

```
# Example of what printing a LinkedList object should like.
>>> from Lab4_spec import LinkedList
>>> my_list = LinkedList
```

Figure 2.3: Include a picture of the `remove_node` method and describe it.

Figure 2.4: Include a picture of the `insert_node` method and describe it.

```
>>> my_list.add_node(1)
>>> my_list.add_node(2)
>>> my_list.add_node(3)
>>> print(my_list)
[1, 2, 3]
```

In addition to adding new nodes to the end of a list, it is also useful to remove nodes and insert new nodes at specified locations. To delete a node, all references to the node must be removed. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to none.

```
# A naive node removal - Does not work.

def remove_node(self, data):

    # Find the node whose next node contains data
    current_node = self.head
    while current_node.next.data != data:
        current_node = current_node.next

    # Remove the next reference to the node that
    # Is to be deleted.

    current_node.next = None
```

Since the only reference to the node that is deleted is the previous node's next attribute, this will delete the node. However, since the only reference to the next node came from the deleted node, it also will deleted. This will continue to the end of the list. Thus, deleting one node in this manner deletes the remainder of the list. This can be remedied by pointing the previous node's next attribute to the node after the deleted node. Then there will be no reference to the removed node and it will be deleted. See Figure 2.3 for an illustration.

```
# A node removal that works

def remove_node(self, data):

    # First, check if the head is the node to be removed.
    # If so, then set the head to be the node after head.
    # This will remove the only reference to head, so it
    # will be deleted.
    if self.head.data == data:
        self.head = self.head.next
    else:
        current_node = self.head
        # Move current_node through the list until it points
        # To the node that precedes that target node.
```

```
        while current_node.next.data != data:
            current_node = current_node.next

        # Point the current node to the node after the
        # node that is to be deleted.

        new_next_node = current_node.next.next
        current_node.next = new_next_node
```

Problem 3. Though the above code works to remove specified nodes, it is not quite complete. Modify the `remove_node` method so that if the user tries to remove a node that is not in the list the method prints "Node not in list." The method should function as follows:

```
>>> my_list = LinkedList()
>>> my_list.add_node(1)
>>> my_list.add_node(2)
>>> print(my_list)
[1, 2]
>>> my_list.remove_node(2)
>>> print(my_list)
[1]
>>> my_list.remove_node(2)
Node not in list.
```

Problem 4. Inserting a node will use similar techniques to removing a node. Add an `insert_node` method to the `LinkedList` class that inserts a new node before the first node that contains the data specified by the user. This function will require two arguments: the data for the new node, and the data of the node before which the new node will be inserted. For example, the function should work as follows.

```
>>> my_list = LinkedList()
>>> my_list.add_node(1)
>>> my_list.add_node(3)
>>> print my_list
[1, 3]
>>> my_list.insert_node(2, 3)
>>> print my_list
[1, 2, 3]
```

See Figure 2.4 for an illustration of the `insert_node` method. Note that since `insert_node` inserts a node before a specified node, it is not possible to `insert_node` at the end of the list.

Doubly-Linked Lists

A doubly-linked list is a linked list where each node keeps track the node that precedes it as well as the node that follows. See Figure **??** for an illustration.

```
# location: Node.py

class DoublyLinkedListNode(Node):
    """A node for a doubly-linked list"""
    def __init__(self, data):
        """Set prev and next attributes"""
        Node.__init__(self, data)
        self.prev = None
        self.next = None
```

All of the methods for linked lists can be implemented for doubly-linked lists. See Figures A and B for illustrations of the insert and remove methods.

Problem 5. Implement a doubly-linked list class called `DoublyLinkedList` that builds a doubly-linked list of `DoublyLinkedListNode` instances. Add an attribute called `tail` that keeps track of the node at the end of the list. Inherit from the `LinkedList` class and overwrite the `add_node`, `remove_node`, and `insert_node` methods. Use the `tail` attribute to make the `add_node` method more efficient.

Problem 6. Implement a new data structure called a sorted linked list. A sorted linked list adds new nodes so that their data is in order. Inherit this class from `DoublyLinkedList`. Inherit the node class for this data structure from `StrNode`, and modify it so that it is compatible with doubly-linked lists. The only method that needs to be changed from `DoublyLinkedList` is `add_node`. When a new node is added, traverse the list until the data in the next node is greater than or equal to the data for the new node. Then insert the new node so that order is maintained. For example, the class should function as follows:

```
>>> from Lab4_spec import SortedLinkedList
>>> sorted_list = SortedLinkedList()
>>> sorted_list.add_node('aardvark')
>>> sorted_list.add_node('backgammon')
>>> print sorted_list
['aardvark', 'backgammon']

>>> sorted_list.add_node('zebra')
>>> sorted_list.add_node('year')
>>> print sorted_list
['aardvark', 'backgammon', 'year', 'zebra']
```

Import the `Lab4_data` module. This includes a method called `create_word_list` that returns a list of words that are out of order. Add these words to a sorted

linked list.