This is to solve the Sage portion of Chapter 7 of

"A journey through the realm of numbers:

from quadratic equations to quadratic reciprocity."

In [1]:
```
# Looking for generators modulo p.
```

In [2]:
```
def is_generator(a, p):
    a = mod(a, p)
    if not(is_prime(p)):
        return "stick to primes"
    if a == 0:
        return false
    b = a
    k = 1
    while not(b == 1):
        k = k+1
        b = b*a
    if k == p-1:
        return true
    else:
        return false
```

In [3]:
```
is_generator(0, 1)
```

Out[3]: 'stick to primes'

In [4]:
```
is_generator(2, 13)
```

Out[4]: True

In [5]:
```
def has_generator(p):
    if not(is_prime(p)):
        return "not a prime"
    for a in Integers(p):
        if is_generator(a, p):
            return true
    return false
```

In [6]:
```
has_generator(13)
```

Out[6]: True

In [7]:
```
has_generator(29)
```

Out[7]: True

In [8]:
```
all([has_generator(p) for p in range(10000) if is_prime(p)])
```
Out[8]: True

In [9]:
```
# Which elements are squares modulo p for odd primes?
```

In [10]:
```
list2square = [p for p in range(3,1000)
               if is_prime(p) and is_square(mod(2,p))]
```

In [11]:
```
Set([mod(p, 4) for p in list2square]) # modulo 4 seems useless
```
Out[11]: {1, 3}

In [12]:
```
Set([mod(p, 8) for p in list2square]) # modulo 8 might help
```
Out[12]: {1, 7}

In [13]:
```
# This checks whether (at least for p<1000) the condition
# whether 2 is a square in F_p relates to the above
# congruence condition modulo 8.
all([p in list2square for p in range(3, 1000)
                    if is_prime(p) and (mod(p, 8) in [1, 7])])
```
Out[13]: True

In [14]:
```
list3square = [p for p in range(5, 1000)
               if is_prime(p) and is_square(mod(3, p))]
```

In [15]:
```
Set([mod(p, 9) for p in list3square]) # modulo 9 seems useless
```
Out[15]: {1, 2, 4, 5, 7, 8}

In [16]:
```
list5square = [p for p in range(7, 1000)
               if is_prime(p) and is_square(mod(5, p))]
```

In [17]:
```
Set([mod(p, 5) for p in list5square]) # modulo 5 might help
```
Out[17]: {1, 4}

In [18]:
```
# This checks whether (at least for p<1000) the condition
# whether 5 is a square in F_p relates to the above
# congruence condition modulo 5.
all([p in list5square for p in range(7, 1000)
                    if is_prime(p) and (mod(p, 5) in [1, 4])])
```
Out[18]: True

In [19]:
```
# This raises the questions (see chapter 8):
# For which primes q is the condition
#      "q is a square in F_p"
# characterized by a congruence condition modulo q?
# How can one characterize the condition for q=3?
```

In [20]:
```
# To define the symmetric group with 3 elements we
# simply need to write
G = SymmetricGroup(3)
```

In [21]:
```
for g in G:
    print(g)
```

```
()
(1,3,2)
(1,2,3)
(2,3)
(1,3)
(1,2)
```

In [22]:
```
G.is_abelian()
```

Out[22]:
```
False
```

In [23]:
```
G.order()
```

Out[23]:
```
6
```

In [24]:
```
table([[g*h for g in G] for h in G])
```

Out[24]:

	(1, 3, 2)	(1, 2, 3)	(2, 3)	(1, 3)	(1, 2)
(1, 3, 2)	(1, 2, 3)		(1, 3)	(1, 2)	(2, 3)
(1, 2, 3)		(1, 3, 2)	(1, 2)	(2, 3)	(1, 3)
(2, 3)	(1, 2)	(1, 3)		(1, 2, 3)	(1, 3, 2)
(1, 3)	(2, 3)	(1, 2)	(1, 3, 2)		(1, 2, 3)
(1, 2)	(1, 3)	(2, 3)	(1, 2, 3)	(1, 3, 2)	

In [25]:
```
# in the second row and third column we see the composition
# of (1,2) and (1,2,3):
#    the latter sends 1 to 2, the former 2 to 1
#    the latter sends 2 to 3, the former 3 to 3
#    the latter sends 3 to 1, the former 1 to 2
# which in total amounts to the permutation (2,3)
```

In [26]:
```
def max_order(n):
    G = SymmetricGroup(n)
    orders = [g.order() for g in G]
    return max(orders)
```

In [27]:
```
# (1,2,3) has order 3
max_order(3)
```

Out[27]:
```
3
```

In [28]:
```python
# (1,2,3,4) has order 4
max_order(4)
```
Out[28]: 4

In [29]:
```python
# (1,2,3,4,5)(6,7,8)(9,10) has order 30
max_order(10)
```
Out[29]: 30

In [30]:
```python
# We implement a crude version of the Diffie-Hellman key exchange.
```

In [31]:
```python
# The following 'dictionary' allows us to translate letters,
# the space, and fullstops to unique numerical codes.
numdict={" ":0,"a":1,"b":2,"c":3,"d":4,"e":5,"f":6,"g":7,
         "h":8,"i":9,"j":10,"k":11,"l":12,"m":13,"n":14,"o":15,
         "p":16,"q":17,"r":18,"s":19,"t":20,"u":21,"v":22,"w":23,
         "x":24,"y":25,"z":26,".":27}
```

In [32]:
```python
numdict["c"] # dictionaries work similarly to a list
```
Out[32]: 3

In [33]:
```python
# Using base 28 and the dictionary we can encode a text string
# into a number.
def encode(text):
    numcode = 0
    for ch in text:
        numcode = numcode*28 + numdict[ch]
    return numcode
```

In [34]:
```python
messagenumber = encode("remember the milk.")
messagenumber
```
Out[34]: 72719048976766574891132623

In [35]:
```python
# This number encodes the message (in a simple unsecure manner).
```

In [36]:
```python
chardict={0:" ",1:"a",2:"b",3:"c",4:"d",5:"e",6:"f",7:"g",8:"h",
         9:"i",10:"j",11:"k",12:"l",13:"m",14:"n",15:"o",16:"p",
         17:"q",18:"r",19:"s",20:"t",21:"u",22:"v",23:"w",24:"x",
         25:"y",26:"z",27:"."}
```

In [37]:
```python
def decode(number):
    text = ""
    while number > 0:
        text = chardict[number%28] + text
        number = number // 28
    return text
```

In [38]:
```
decode(messagenumber)
```
Out[38]: 'remember the milk.'

In [39]:
```
# To securely transmit the message we need a prime larger
# than our message.
myprime = random_prime(10*messagenumber, True, 2*messagenumber)
myfield = FiniteField(myprime)
myprime
```
Out[39]: 3280614565536642581402749

In [40]:
```
# We also need to use a generator for the multiplicative group.
mygen = myfield.multiplicative_generator()
mygen
# The above randomly chosen prime and the generator (often but not
# always 2) are public information.
```
Out[40]: 3

In [41]:
```
# Alice chooses a secret exponent a and transmits the
# a-th power of the generator.
alice_number_a = floor((1/4 + 1/2*random())*myprime)
alice_sends = mygen ^ alice_number_a
```

In [42]:
```
# Bob chooses a secret exponent b and transmits the
# b-th power of the generator.
bob_number_b = floor((1/4 + 1/2*random())*myprime)
bob_sends = mygen ^ bob_number_b
```

In [43]:
```
# Alice receives bob_sends and knows her number. Hence can
# calculate:
alice_key = bob_sends ^ alice_number_a
```

In [44]:
```
# Bob received alice_sends and knows his number. Hence he can
# calculate:
bob_key = alice_sends ^ bob_number_b
```

In [45]:
```
# Did they get the same number?
alice_key == bob_key
```
Out[45]: True

In [46]:
```
alice_key
```
Out[46]: 1205287098868661999304432

In [47]:
```
def encrypt(message, key):
    return message * key

def decrypt(coded_message, key):
    return Integer(coded_message / key)
# The command Integer() transforms the element in the finite
# field back to an integer that can be used in the decode
# routine.
```

In [48]:
```
coded_message = encrypt(messagenumber, alice_key)
coded_message
```

Out[48]:
```
2338259226670710587403154363
```

In [49]:
```
# The point of this new number is that the letters stored
# in the various digits with respect to base 28 are now scrambled.
decode(decrypt(coded_message, bob_key))
```

Out[49]:
```
'remember the milk.'
```

In [50]:
```
# It is near impossible to recover the message without
# the correct key.
print(decode(decrypt(coded_message, bob_key-2)))
print(decode(decrypt(coded_message, bob_key-1)))
print(decode(decrypt(coded_message, bob_key+1)))
print(decode(decrypt(coded_message, bob_key+2)))
```

```
aiimeyqr.jmwtkxqibm
aicqdydzztquq y jot
awis.lkdw yygelr pg
ajpbnfxcpsh.itwvylb
```

In [51]:
```
# However, since myfield and mygen are public the evesdropper Eve
# could calculate the numbers alice_number_a, bob_number_b,
# and in particular the key.
eves_guess_for_a = alice_sends.log(mygen)
```

In [52]:
```
eves_guess_for_a == alice_number_a
```

Out[52]:
```
True
```

In [53]:
```
# With this Eve can also get the message.
decode(decrypt(coded_message, bob_sends ^ eves_guess_for_a))
```

Out[53]:
```
'remember the milk.'
```

In [54]:
```
timeit('alice_sends.log(mygen)', number=1, repeat=1)
```

Out[54]:
```
1 loops, best of 1: 1.85 s per loop
```

In [55]:
```
# We suggest to try encoding a (slightly) longer message. In this
# case the above code will pick larger primes and everything will
# work smoothly except for Eve.
```

```
In [ ]:
```