This is to solve the Sage portion of Chapter 8 of

"A journey through the realm of numbers:

from quadratic equations to quadratic reciprocity."

In [1]:
```
# We first try to find the congruence condition for
# primes of the form a^2-2b^2.
```

In [2]:
```
listminus2 = Set(a^2-2*b^2 for a in range(100)
                           for b in range(100))
primesminus2 = Set(p for p in listminus2
                     if p in range(1000) and is_prime(p))
# unlike earlier cases, it is not clear whether
# p around 2000 can be written with a and b
# below 100, restricting to p<1000 is trying
# to eliminate this issue but it remains a gamble
```

In [3]:
```
# We see that modulo 4 seems useless.
Set(n%4 for n in primesminus2)
```

Out[3]:
```
{1, 2, 3}
```

In [4]:
```
# However, modulo 8 might help.
Set(n%8 for n in primesminus2)
```

Out[4]:
```
{1, 2, 7}
```

In [5]:
```
primesminus2 == Set(p for p in range(1000)
                    if is_prime(p)
                       and (p==2 or (p%8 in [1, 7])))
```

Out[5]:
```
True
```

In [6]:
```
# Next we study numbers of the form a^2+5b^2.
```

In [7]:
```
valuesofmain = Set(a^2 + 5*b^2 for a in range(100)
                               for b in range(100)
                               if a^2 + 5*b^2 < 10000)
```

In [8]:
```
valuesofflipped = Set(2*a^2 + 2*a*b + 3*b^2
                      for a in range(-300, 300)
                      for b in range(-300, 300)
                      if 2*a^2 + 2*a*b + 3*b^2 < 10000)
```

In [9]:
```
def pos5test(n):
    return all((p%20 in [1, 2, 3, 5, 7, 9]) or is_even(e)
               for (p, e) in factor(n))
possible = Set(n for n in range(10000)
               if n == 0 or pos5test(n))
```

In [10]:
```
possible == valuesofmain.union(valuesofflipped)
```

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

In [11]:
```
valuesofmain.intersection(valuesofflipped)
```

Out[11]:
```
{0}
```

In [12]:
```
# Next we study a^2+11b^2 and a^2+ab+3b^2.
```

In [13]:
```
full = Set(a^2 + a*b + 3*b^2 for a in range(-300, 300)
                             for b in range(-300, 300)
                             if a^2 + a*b + 3*b^2 < 50000)
```

In [14]:
```
main = Set(a^2 + 11*b^2 for a in range(-300, 300)
                        for b in range(-300, 300)
                        if a^2 + 11*b^2 < 50000)
```

In [15]:
```
main.issubset(full)
```

Out[15]:
```
True
```

In [16]:
```
# Counting actual primes below 50000.
Set(p for p in main if is_prime(p)).cardinality()
```

Out[16]:
```
840
```

In [17]:
```
# Counting possible primes for comparison, suggests
# that there are infinitely many shadow primes.
Set(p for p in full if is_prime(p)).cardinality()
```

Out[17]:
```
2544
```

In [18]:
```
# Which d below 100 can be used to distinguish actual
# numbers? (Note that above 100 one may have to increase the
# lists beyond 50000.)
[d for d in range(1,100)
   if Set(n%d for n in main) != Set(n%d for n in full)]
```

Out[18]:
```
[]
```

In [19]:
```
# Checking test for possible list.
cong = Set(p%11 for p in full if is_prime(p))
print(cong)
(Set(p for p in full if is_prime(p)) ==
 Set(p for p in range(1, 50000)
       if is_prime(p) and p%11 in cong))
```

```
{0, 1, 3, 4, 5, 9}
```

Out[19]:
```
True
```

In [20]:
```
shadowprimes = Set(p for p in full
                   if is_prime(p) and not(p in main))
```

In [21]:
```
def shadowtest(n):
    if n == 0 or n == 1:
        return true  # these work
    factlist = factor(n)
    if any(not(p%11 in cong) and is_odd(e)
           for (p,e) in factlist):
        return false  # n is not possible
    if factlist[0][0]==2:
        return true   # a factor 4 helps
    sumshexp = sum([e for (p,e) in factlist
                    if p in shadowprimes])
    if sumshexp == 0 or sumshexp > 1:
        return true  # no or at least two
    else:            # shadowprimes
        return false
```

In [22]:
```
main == Set(n for n in range(50000)
              if shadowtest(n))
```

Out[22]:
```
True
```

In [23]:
```
# Finally we study a^2+14b^2.
```

In [24]:
```
def q0(x,y):
    return x^2+14*y^2
def q1(x,y):
    return 2*x^2+7*y^2
def q2(x,y):
    return 3*x^2+2*x*y+5*y^2
def q3(x,y):
    return 3*x^2-2*x*y+5*y^2
```

In [25]:
```
q0val = Set([q0(a, b) for a in range(250)
                      for b in range(250)
                      if q0(a,b) < 50000])
```

In [26]:
```
q1val = Set(q1(a, b) for a in range(250)
                     for b in range(250)
                     if q1(a, b)<50000)
```

In [27]:
```
q2val = Set(q2(a, b) for a in range(-600,600)
                     for b in range(600)
                     if q2(a,b)<50000)
# as q2 contains the term 2xy, we need to allow
# both signs for x (but may restrict to positive y)
```

In [28]:
```
q3val = q2val # because the sign change does
              # not change the set of values
```

In [29]:
```
print(q0val.cardinality())  # how many actual values
print((q0val.intersection(q1val)).cardinality())
                            # huge overlap
print(q0val.intersection(q2val))
                            # small overlap
print(q1val.intersection(q2val))
                            # small overlap
```

```
7515
5603
{0}
{0}
```

In [30]:
```
# The congruence condition for -14 being a
# square modulo p, can be phrased in terms
# of a congruence modulo 8*7=56:
# -7 is a square if p is congruent to 1,2,4 mod 7
# 2 is a square if p is congruent to 1,7 mod 8
def squaretest(p):
    return ((p%7 in [1,2,4] and p%8 in [1,7])
            or (p%7 in [3,5,6] and p%8 in [3,5])
            or p==2 or p==7)
```

In [31]:
```
squaretest(3)  #-14=-2=1 mod 3
```

Out[31]:
```
True
```

In [32]:
```
squaretest(5)  #-14=-4=1 mod 5
```

Out[32]:
```
True
```

In [33]:
```
squaretest(11) #-14=-3=8 mod 11 is not a square
```

Out[33]:
```
False
```

In [34]:
```
# We define possible values using the standard pattern!
def posstest(n):
    if n == 0:
        return true
    return all(is_even(e) or squaretest(p)
               for (p, e) in factor(n))
```

In [35]:
```
# Does this test work for the actual values?
q0val == Set(n for n in range(10000) if posstest(n))
```

Out[35]:
```
False
```

In [36]:
```
# So we instead consider the union:
allvalues = q0val.union(q1val).union(q2val)
```

In [37]:
```
# This in turn is indeed characterized by the criterion.
allvalues == Set(n for n in range(50000) if posstest(n))
```

Out[37]:
```
True
```

In [38]:
```
# Which d below 100 can be used to distinguish the
# values of q0 and of q1?
[d for d in range(1, 100)
   if Set((n%d) for n in q0val) != Set((n%d) for n in q1val)]
```

Out[38]:
```
[]
```

In [39]:
```
# Which d below 20 can be used to distinguish the
# values of q0 and of q2?
[d for d in range(1, 20)
   if Set((n%d) for n in q0val) != Set((n%d) for n in q2val)]
```

Out[39]:
```
[7, 8, 14, 16]
```

In [40]:
```
print(Set((n%7) for n in q0val))
print(Set((n%7) for n in q2val))
```

```
{0, 1, 2, 4}
{0, 3, 5, 6}
```

In [41]:
```
# For primes p the value p%7 allows for a precise criterion
# whether p is a value of either q0 or of q1.
(Set(p for p in q0val.union(q1val) if is_prime(p)) ==
 Set(p for p in range(50000)
       if is_prime(p) and posstest(p)
          and p%7 in [0,1,2,4]))
```

Out[41]:
```
True
```

In [42]:
```
# This programs a mysterious class field theory test
# whether a possible prime is a value of q0.
def classtest(p):
    return any((n^2+1)^2 == 8 for n in Integers(p))
```

In [43]:
```
# Avoiding the small primes 2 and 7, this property
# now characterizes the prime values of q0:
(Set(p for p in q0val
       if is_prime(p) and p!=2 and p!=7)
 == Set(p for p in range(50000)
          if is_prime(p) and p!=2 and p!=7
             and posstest(p) and p%7 in [0,1,2,4]
             and classtest(p)))
```

Out[43]: True

In [ ]: