This is to solve the Sage portion of Chapter 3 of

"A journey through the realm of numbers:

from quadratic equations to quadratic reciprocity."

In [1]:
```
table([[m^2 + n^2 for m in range(16)] for n in range(16)])
        # sums of two squares in a table
```

Out[1]:

0	1	4	9	16	25	36	49	64	81	100	121	144	169	196	225
1	2	5	10	17	26	37	50	65	82	101	122	145	170	197	226
4	5	8	13	20	29	40	53	68	85	104	125	148	173	200	229
9	10	13	18	25	34	45	58	73	90	109	130	153	178	205	234
16	17	20	25	32	41	52	65	80	97	116	137	160	185	212	241
25	26	29	34	41	50	61	74	89	106	125	146	169	194	221	250
36	37	40	45	52	61	72	85	100	117	136	157	180	205	232	261
49	50	53	58	65	74	85	98	113	130	149	170	193	218	245	274
64	65	68	73	80	89	100	113	128	145	164	185	208	233	260	289
81	82	85	90	97	106	117	130	145	162	181	202	225	250	277	306
100	101	104	109	116	125	136	149	164	181	200	221	244	269	296	325
121	122	125	130	137	146	157	170	185	202	221	242	265	290	317	346
144	145	148	153	160	169	180	193	208	225	244	265	288	313	340	369
169	170	173	178	185	194	205	218	233	250	269	290	313	338	365	394
196	197	200	205	212	221	232	245	260	277	296	317	340	365	392	421
225	226	229	234	241	250	261	274	289	306	325	346	369	394	421	450

In [2]:
```
def sumsq(k=2, t=100):          # recursive definition
    N = floor(sqrt(t))+1        # of sums of k squares
    if k == 1:
        return Set([m^2 for m in range(N)])
    else:
        oldsums = sumsq(k-1,t)
        return Set([s + n^2 for s in oldsums
                            for n in range(N)
                            if s+n^2<=t])
```

In [3]:
```
sumsq(2, 30)              # testing the case of 2 squares
```

Out[3]:
```
{0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 25, 26, 29}
```

In [4]:
```
sumsq(3, 30)              # of 3 squares
```

Out[4]:
```
{0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20
, 21, 22, 24, 25, 26, 27, 29, 30}
```

In [5]:
```
sumsq(4, 30)              # of 4 squares
```

Out[5]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

In [6]:
```
sumsq(4, 10^5) == Set(range(10^5+1))
            # checks whether Lagrange's theorem holds below 10^5
```

Out[6]:
```
True
```

In [7]:
```
def legendre(n):          # checks whether the condition
    if n == 0:              # in the Legendre-Gauss theorem holds
        return True
    elif n%4 == 0:
        return legendre(n / 4)
    else:
        return n%8 != 7
```

In [8]:
```
legendre(28)              # 4*7 is not a sum of 3 squares
```

Out[8]:
```
False
```

In [9]:
```
sumsq(3, 10^5) == Set([n for n in range(10^5+1) if legendre(n)])
            # checks whether Legendre-Gauss holds below 10^5
```

Out[9]:
```
True
```

In [10]:
```
def pyth_triple(t=100):   # exhaustive search is easy to implement
    return Set([(a,b,c) for a in range(1,t+1)
                        for b in range(1,t+1)
                        for c in range(1,t+1)
                        if a^2+b^2==c^2])
```

In [11]:
```
pyth_triple(10)             # to find all triples with c<=10
```

Out[11]:
```
{(3, 4, 5), (8, 6, 10), (6, 8, 10), (4, 3, 5)}
```

In [12]:
```
def euclid_pyth(t=100):   # Euclid's formula is more involved
    N = floor(sqrt(t))
    primitive = [(m^2-n^2, 2*m*n, m^2+n^2)
                 for m in range(1, N+1)
                 for n in range(1, N+1)
                 if n < m and gcd(m,n) == 1 and
                    m^2+n^2 <= t and (m%2 == 0 or n%2 == 0)]
                # gcd calculates the greatest common divisor
    multiples=[(k*a, k*b, k*c) for k in range(1, t)
                               for a, b, c in primitive
                               if k*c<=t]+\
                [(k*b, k*a, k*c) for k in range(1,t)
                               for a, b, c in primitive
                               if k*c <= t]
    return Set(multiples)
```

In [13]:
```
euclid_pyth(10)           # to find all triples with c<=10
```

Out[13]:
```
{(4, 3, 5), (3, 4, 5), (8, 6, 10), (6, 8, 10)}
```

In [14]:
```
pyth_triple(200) == euclid_pyth(200)  # same outcome
```

Out[14]:
```
True
```

In [15]:
```
timeit('pyth_triple(400)', number=1, repeat=1) # exhaustive search
                    # timeit checks how long the calculation takes
```

Out[15]:
```
1 loops, best of 1: 84.5 s per loop
```

In [16]:
```
timeit('euclid_pyth(400)', number=1, repeat=1) # Euclid's formula
                    # which gives a clear winner
```

Out[16]:
```
1 loops, best of 1: 12.1 ms per loop
```

In [17]:
```
def sums_pm(k=2, t=100):        # again sums of k squares but this
    N = floor(sqrt(t))+1        # time using integer inputs and
    if k == 1:                  # using lists to be able to count
        return ([m^2 for m in range(-N+1, N)])
    else:
        oldsums = sums_pm(k-1, t)
        return ([s+n^2 for s in oldsums\
                       for n in range(-N+1, N)\
                       if s+n^2 <= t])
```

In [18]:
```
sums_pm(1, 25)                  # squares below 25
```

Out[18]:
```
[25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25]
```

In [19]:
```
def r_list(k=2, t=100):         # calculating multiplicity r_k(n)
    allsums = sums_pm(k,t)
    return [(n, allsums.count(n)) for n in range(t+1)]
```

In [20]:
```
r_list(2,10)
```

Out[20]:
```
[(0, 1),
 (1, 4),
 (2, 4),
 (3, 0),
 (4, 4),
 (5, 8),
 (6, 0),
 (7, 0),
 (8, 4),
 (9, 4),
 (10, 8)]
```

In [21]:
```
mylist = r_list(4,10^3)
# This calculates the multiplicity for 4 squares up to 10^3
# and takes some time to create due to our quite crude routine.
```

In [22]:
```
all([not(is_prime(p)) or rp == 8*(p+1) for p, rp in mylist])
# "p is not prime or the multiplicity is 8(p+1)"
#           amounts to the same thing as
# "p prime implies that the multiplicity is 8(p+1)"
```

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

In [23]:
```
all([m == 0 or n == 0 or gcd(m,n) != 1 or m*n > 10^3
     or 8*mylist[m*n][1] == rm*rn
     for m, rm in mylist
     for n, rn in mylist])
    # "m==0 or n==0" are there to avoid talking about these cases
    # "gcd(m,n)!=1" is there to restrict talking about coprime m,n
    # "m*n>10^3" is there to avoid talking about to large cases
    # finally "8*mylist[m*n][1]==rm*rn" is the desired formula
```

Out[23]:
```
True
```

In [24]:
```
# Practicing congruences and powers.
def find_loop(a=7, d=10):
    remlist = [1]                    # corresponds to a^0
    k = 1                            # next power
    while not(a^k%d in remlist):     # until repetition is found
        remlist.append(a^k % d)      # new entry is added
        k = k+1                      # power is increased
    start = remlist.index(a^k % d) # finds first repetition
    print("powers of", a, "mod", str(d)+":")
    looptext = ""
    for j in range(start):
        looptext = looptext + str(remlist[j]) + "->"
    looptext = looptext + "loop->"
    for j in range(start, len(remlist)):
        looptext = looptext + str(remlist[j]) + "->"
    looptext = looptext + "loop"
    print(looptext)
    return (remlist, start, k-start)
            # returns full list of residues
            # together with index of where the loop starts
            # and the length of the loop
```

In [25]:
```
find_loop()
```

```
powers of 7 mod 10:
loop->1->7->9->3->loop
```

Out[25]:
```
([1, 7, 9, 3], 0, 4)
```

In [26]:
```
find_loop(5,100)
```

```
powers of 5 mod 100:
1->5->loop->25->loop
```

Out[26]:
```
([1, 5, 25], 2, 1)
```

In [27]:
```
find_loop(2,14)
```

```
powers of 2 mod 14:
1->loop->2->4->8->loop
```

Out[27]:
```
([1, 2, 4, 8], 1, 3)
```

In [28]:
```
# Finding the last two digits of a=9^(9^(9^ ...)) is now easier.
find_loop(9,100)
```

```
powers of 9 mod 100:
loop->1->9->81->29->61->49->41->69->21->89->loop
```

Out[28]:
```
([1, 9, 81, 29, 61, 49, 41, 69, 21, 89], 0, 10)
```

In [29]:
```
# This tell us that for 9^b modulo 100
# we have to find b=9^(9^...) modulo 10.
find_loop(9,10)
```

```
powers of 9 mod 10:
loop->1->9->loop
```

Out[29]:
```
([1, 9], 0, 2)
```

In [30]:
```
# Since b is an odd power of 9, b is congruent to 9 modulo 10.
# Taking this to the loop modulo 100, we obtain that the number
# a=9^(9^(9^....)) has 89 as the last two digits.
```

In [31]:
```
# We now automate the above procedure.
# For this bases=[a,b,c,...] and we consider N=a^(b^(c^..)),
# where we assume that a,b,c,.. are natural numbers.
# In the general case we will have to know whether N is larger
# than the start of the loop found by find_loop -- without
# actually calculating N as this would very quickly defeat sage.

def power_large(bases, start):  # is N larger than start?
    if start <= 1:
       return True              # N is an integer, clearly true
    elif len(bases) == 1:
       return bases[0] >= start   # N=bases[0] in this case
    else:
       return power_large(bases[1:], log(start, bases[0]).n())
       # bases[1:] picks everything except the entry at 0
       # using the logarithm to base bases[0]
```

In [32]:
```
# Is 2^(3^2) larger than 10?
power_large([2, 3, 2], 10)
```

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

In [33]:
```
# If we don't get into the loop, we suspect that N is not too
# large and simply calculate it.
def power_calc(bases):
    if len(bases) == 1:
        return bases[0]
    else:
        return bases[0]^(power_calc(bases[1:]))
```

In [34]:
```
power_calc([2, 3, 2])     # a small example to test it
```

Out[34]:
```
512
```

In [ ]:
```
# This is a large example to test it (filling 185 pages).
power_calc([7, 7, 7])
```

In [36]:
```
# This is way too large and creates an error message.
power_calc([7, 7, 7, 7])
```

```
---------------------------------------------------------------------
---------
OverflowError                                Traceback (most recent c
all last)
<ipython-input-36-7861af3e1762> in <module>()
      1 # This is way too large and creates an error message.
----> 2 power_calc([Integer(7), Integer(7), Integer(7), Integer(7)
])

<ipython-input-33-198844605601> in power_calc(bases)
      5         return bases[Integer(0)]
      6     else:
----> 7         return bases[Integer(0)]**(power_calc(bases[Intege
r(1):]))

/Users/manfred/Documents/SageMath9/local/lib/python3.7/site-packag
es/sage/rings/integer.pyx in sage.rings.integer.Integer.__pow__ (b
uild/cythonized/sage/rings/integer.c:14870)()
   2208
   2209         if type(left) is type(right):
-> 2210             return (<Integer>left)._pow_(right)
   2211         elif isinstance(left, Element):
   2212             return coercion_model.bin_op(left, right,
operator.pow)

/Users/manfred/Documents/SageMath9/local/lib/python3.7/site-packag
es/sage/rings/integer.pyx in sage.rings.integer.Integer._pow_ (bui
ld/cythonized/sage/rings/integer.c:15220)()
   2288                 r = smallInteger(1)
   2289         else:
-> 2290             raise OverflowError(f"exponent must be at most
{LONG_MAX}")
   2291         if mpz_sgn(exp) >= 0:
   2292             return r

OverflowError: exponent must be at most 9223372036854775807
```

In [37]:
```
def power_mod(bases, d):
    if len(bases) == 1 or d == 1:
        return bases[0] % d
    else:    # the next command defines three variables at once
        looplist, start, newd = find_loop(bases[0], d)
        if power_large(bases, bases[0]^start):   # in the loop?
            newp_mod = power_mod(bases[1:], newd)
            index = start + (newp_mod-start)%newd
            return looplist[index]
        else:                          # if not, simply calculate
            return power_calc(bases) % d
```

In [38]:
```
power_mod([6, 7, 8, 9, 10], 100)
```

```
powers of 6 mod 100:
1->6->loop->36->16->96->76->56->loop
powers of 7 mod 5:
loop->1->2->4->3->loop
powers of 8 mod 4:
1->loop->0->loop
```

Out[38]: 56

In [39]:
```
# powers of 8 are congruent to 0 modulo 4
# hence the power 7^(8^...) is congruent to 1 modulo 5
# hence the power 6^(7^(8^...)) is in the loop for powers of 6
# modulo 5 at a position congruent to 1, but needs to be
# in the loop, this shows that the outcome is 56.
```

In [40]:
```
# testing another set of examples
power_mod([3], 100)
```

Out[40]: 3

In [41]:
```
power_mod([3, 3], 100)
```

```
powers of 3 mod 100:
loop->1->3->9->27->81->43->29->87->61->83->49->47->41->23->69->7->
21->63->89->67->loop
```

Out[41]: 27

In [42]:
```
power_mod([3, 3, 3], 100)
```

```
powers of 3 mod 100:
loop->1->3->9->27->81->43->29->87->61->83->49->47->41->23->69->7->
21->63->89->67->loop
powers of 3 mod 20:
loop->1->3->9->7->loop
```

Out[42]: 87

In [43]:
```
power_mod([3, 3, 3, 3], 100)
```

```
powers of 3 mod 100:
loop->1->3->9->27->81->43->29->87->61->83->49->47->41->23->69->7->
21->63->89->67->loop
powers of 3 mod 20:
loop->1->3->9->7->loop
powers of 3 mod 4:
loop->1->3->loop
```

Out[43]: 87

In [44]:
```
(3^(3^3)) % 100              # just to verify our previous answer
```

Out[44]: 87

In [45]:
```
power_mod([7, 7, 7, 7], 100) # also works for the too large one
```

```
powers of 7 mod 100:
loop->1->7->49->43->loop
powers of 7 mod 4:
loop->1->3->loop
powers of 7 mod 2:
loop->1->loop
```

Out[45]: 43

In [ ]: