Math 480 -- April 11, 2008

```
def last_digits1(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: First naive version.
    """
    return str(n^e)[-d:]
```

```
time last_digits1(3,10^6,4)
```

```
'0001'
CPU time: 0.43 s,  Wall time: 0.43 s
```

```
def last_digits2(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: Rewritten to break up statements which makes timing
easier.
    """
    time m = n^e
    time s = str(m)
    time t = s[-d:]
    return t
```

```
last_digits2(3,10^6,4)
```

```
Time: CPU 0.04 s, Wall: 0.04 s
Time: CPU 0.39 s, Wall: 0.40 s
Time: CPU 0.00 s, Wall: 0.00 s
'0001'
```

```
# From the above, we see that the string conversion takes by far
the most time
```

```
def last_digits2b(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: Rewritten to use the cputime() command, which works
    even in .py files, Sage library code, and Cython code.
    """
    tm=cputime(); m = n^e; print cputime(tm)
    tm=cputime(); s = str(m); print cputime(tm)
    tm=cputime(); t = s[-d:]; print cputime(tm)
    return t
```

```
time last_digits2(3,10^6,4)
```

```
    Time: CPU 0.04 s, Wall: 0.04 s
    Time: CPU 0.39 s, Wall: 0.39 s
    Time: CPU 0.00 s, Wall: 0.00 s
    '0001'
    CPU time: 0.43 s,  Wall time: 0.43 s
```

```
def last_digits3(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: Rewritten to use an algorithm that avoids big string
conversion.
    """
    time m = n^e
    time s = m%(10^d)
    time t = str(s)
    time t = '0'*(d-len(t)) + t
    return t
```

```
time last_digits3(3,10^6,4)
```

```
    Time: CPU 0.04 s, Wall: 0.04 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    '0001'
    CPU time: 0.04 s,  Wall time: 0.04 s
```

```
# We switch to a bigger benchmark
```

```
time last_digits3(3,10^7,4)
```

```
    Time: CPU 0.53 s, Wall: 0.54 s
    Time: CPU 0.01 s, Wall: 0.01 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    '0001'
    CPU time: 0.53 s,  Wall time: 0.54 s
```

```
# From the above we see that the initial powering operation m = n^e
takes
# most of the time ("dominates")
```

```
def last_digits4(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: Rewritten to use an algorithm/data type that
    speeds up the first powering.
    """
    time m = Integers(10^d)(n)
    time s = m ^ e
    time t = str(s)
    time t = '0'*(d-len(t)) + t
    return t
```

```
time last_digits4(3,10^7,4)
```

```
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    '0001'
    CPU time: 0.00 s,  Wall time: 0.00 s
```

```
# We need a bigger example now.
```

```
time last_digits4(389,10^50,10)
```

```
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    Time: CPU 0.00 s, Wall: 0.00 s
    '0000000001'
    CPU time: 0.00 s,  Wall time: 0.00 s
```

```
# OK, instead we time doing the computation many times.
# For this, we better get rid of the print statements.

def last_digits5(n, e, d):
    """
    Return the last d digits of n^e.
    """
    m = Integers(10^d)(n)
    s = m ^ e
    t = str(s)
    t = '0'*(d-len(t)) + t
    return t
```

```
# Using timeit we get the result of running the above function
# on given input hundreds of times, and taking the best wall time.

a = 389; b = 10^50; d = 10
timeit('last_digits5(a,b,d)')         # input must be in quotes
```

```
625 loops, best of 3: 69.7 Âµs per loop
```

```
# That seems really really good.  Can we do better? ...
```

```
%cython
def last_digits6(n, e, d):
    """
    Return the last d digits of n^e.
    """
    m = Integers(10**d)(n)   # IMPORTANT: no preparsing in Cython,
so use ** instead of ^!
    s = m ** e
    t = str(s)
    t = '0'*(d-len(t)) + t
    return t
```

__Users_wa...5_code_sage90_spyx.c __Users_wa...sage90_spyx.pyx.ht

```
time last_digits6(389,10^50,10)
```

```
'0000000001'
CPU time: 0.01 s,  Wall time: 0.08 s
```

```
a = 389; b = 10^50; d = 10
timeit('last_digits6(a,b,d)')
```

```
625 loops, best of 3: 66.4 Âµs per loop
```

```
# No speedup yet; now we're getting desperate...
# Let's say we really really care about the speed of this code.
# We are willing to spend hours looking at manuals
# and source code.  We are willing to put up with the
# possibility of segfaults, etc.  We are willing to
# learn the basics of the GMP C library.  We are also
# willing to sacrifice readability.
```

```
%cython

from sage.rings.integer cimport Integer
def last_digits7(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)     # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)   # tenpow = 10^d

    cdef Integer s = Integer()
    mpz_powm(s.value, n.value, e.value, tenpow)  # m = n^d % tenpow

    t = str(s)
    t = '0'*(d-len(t)) + t
    return t
```

__Users_wa..._code_sage110_spyx.c __Users_wa...age110_spyx.pyx.ht

```
# does it work?
last_digits7(389,10^50,10)
```

```
'0000000001'
```

```
a = 389; b = 10^50; d = 10
timeit('last_digits7(a,b,d)')
```

```
625 loops, best of 3: 30.4 Âµs per loop
```

```
# Wow, it's over twice as fast!
# Is the time in the str stuff at the end? ...
```

```
%cython

from sage.rings.integer cimport Integer
def last_digits8(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)     # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)   # tenpow = 10^d

    cdef Integer s = Integer()
    mpz_powm(s.value, n.value, e.value, tenpow)  # m = n^d % tenpow
    mpz_clear(ten); mpz_clear(tenpow)  # avoid memory leaks!

    return s
```

__Users_wa..._code_sage156_spyx.c __Users_wa...age156_spyx.pyx.ht

```
a = 389; b = 10^50; d = 10
timeit('last_digits8(a,b,d)')
```

625 loops, best of 3: 25 Âµs per loop

```
# No, the time is not dominated by the string stuff at the end.
# Is it Python function call overhead? ...
```

```
%cython
from sage.rings.integer cimport Integer
def last_digits9(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """

    return n
```

__Users_wa..._code_sage125_spyx.c __Users_wa...age125_spyx.pyx.ht

```
a = 389; b = 10^50; d = 10
timeit('last_digits9(a,b,d)')
```

625 loops, best of 3: 306 ns per loop

```
# No, it is not that.  Is it the Integer creation code? ...
```

```
%cython

from sage.rings.integer cimport Integer
def last_digits10(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)     # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)   # tenpow = 10^d
    cdef mpz_t s
    mpz_init(s)
    mpz_powm(s, n.value, e.value, tenpow)  # m = n^d % tenpow
    mpz_clear(ten); mpz_clear(tenpow)  # avoid memory leaks!
    return 0
```

__Users_wa..._code_sage158_spyx.c __Users_wa...age158_spyx.pyx.ht

```
a = 389; b = 10^50; d = 10
timeit('last_digits10(a,b,d)')
```

625 loops, best of 3: 25.2 Âµs per loop

```
# No.  And if one removes the mpz_powm statement above the timing
# goes down to almost nothing, so *that one call* to the
# GMP C Library dominates the runtime.  Thus our function
# is probably fully optimized.
```

```
%cython

from sage.rings.integer cimport Integer
def last_digits11(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)     # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)   # tenpow = 10^d
    cdef mpz_t s
    #mpz_init(s)
    #mpz_powm(s, n.value, e.value, tenpow)  # m = n^d % tenpow

    mpz_clear(ten); mpz_clear(tenpow)  # avoid memory leaks!
    return 0
```

__Users_wa..._code_sage161_spyx.c __Users_wa...age161_spyx.pyx.ht

```
a = 389; b = 10^50; d = 10
timeit('last_digits11(a,b,d)')
```

625 loops, best of 3: 821 ns per loop

```
# How did we do? .
```

```
a = 389; b = 10^5; d = 10
timeit('last_digits7(a,b,d)')
```

625 loops, best of 3: 8.69 Âµs per loop

```
a = 389; b = 10^5; d = 10
timeit('last_digits1(a,b,d)')
```

5 loops, best of 3: 183 ms per loop

```
# The speedup from start to finish is by a factor of over 20,000:
183/(8.69/1000)
```

21058.6881472957

```
last_digits7(a,b,d)
```

'6806000001'

```
last_digits1(a,b,d)
```

'6806000001'

```

```

```
# PROFILING CODE
```

```
def last_digits1(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: First naive version.
    """
    return str(n^e)[-d:]
```

```
import profile
profile.run(preparse('last_digits1(3,10^6,4)'))
```

```
         4 function calls in 0.429 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.429    0.429    0.429    0.429 174.py:6(last_digits1
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.429    0.429
<string>:1(<module>)
        1    0.000    0.000    0.429    0.429
profile:0(last_digits1(Integer(3),Integer(10)**Integer(6),Integer(4
))
        0    0.000             0.000          profile:0(profiler)
```

```
profile.help()
```

```
Documentation for the profile module can be found
in the Python Library Reference, section 'The Python Profiler'.
```

```
html('<h2>Profile the given input</h2>')
import cProfile; import profile
@interact
def _(cmd = ("Statement", '2 + 2'), do_preparse=("Preparse?",
True), cprof =("cProfile?", False)):
    if do_preparse: cmd = preparse(cmd)
    print "<html>"  # trick to avoid word wrap
    if cprof:
        cProfile.run(cmd)
    else:
        profile.run(cmd)
    print "</html>"
```

Profile the given input

Statement `2 + 2`

Preparse? ☑

cProfile? ☐

```
       563 function calls (530 primitive calls) in 0.009 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(IntegerRing)
      147    0.001    0.000    0.001    0.000 :0(__getitem__)
      123    0.001    0.000    0.001    0.000 :0(append)
        1    0.000    0.000    0.000    0.000 :0(degree)
        1    0.000    0.000    0.000    0.000 :0(get_debug_level)
        1    0.000    0.000    0.000    0.000 :0(is_commutative)
        8    0.000    0.000    0.000    0.000 :0(isinstance)
    73/40    0.001    0.000    0.001    0.000 :0(len)
        1    0.000    0.000    0.000    0.000 :0(parent)
        1    0.000    0.000    0.000    0.000 :0(range)
        1    0.000    0.000    0.000    0.000 :0(set_debug_level)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        2    0.000    0.000    0.000    0.000 :0(sort)
        2    0.000    0.000    0.000    0.000 :0(sum)
        1    0.000    0.000    0.009    0.009 :1()
        1    0.000    0.000    0.000    0.000 arith.py:1556(__factor_using_pari)
        1    0.000    0.000    0.001    0.001 arith.py:1580(factor)
        1    0.004    0.004    0.009    0.009 arith.py:917(divisors)
        1    0.000    0.000    0.001    0.001 factorization.py:161(__init__)
      147    0.002    0.000    0.003    0.000 factorization.py:257(__getitem__)
       34    0.000    0.000    0.001    0.000 factorization.py:297(__len__)
        1    0.000    0.000    0.000    0.000 factorization.py:420(base_ring)
        1    0.000    0.000    0.000    0.000 factorization.py:439(is_commutative)
        1    0.000    0.000    0.000    0.000 factorization.py:484(simplify)
        2    0.000    0.000    0.000    0.000 factorization.py:500()
        1    0.000    0.000    0.000    0.000 factorization.py:511(sort)
        1    0.000    0.000    0.009    0.009 profile:0(divisors(Integer(10)**Integer(10)))
        0    0.000             0.000          profile:0(profiler)
        1    0.000    0.000    0.000    0.000 proof.py:151(get_flag)
        4    0.000    0.000    0.000    0.000 rational_field.py:135(__hash__)
        1    0.000    0.000    0.000    0.000 rational_field.py:147(__call__)
        1    0.000    0.000    0.000    0.000 rational_field.py:233(_coerce_impl)
```