Question 5)

The purpose of this question is to show how to generalize the Square and Multiply Exponentiation method to different radixes besides 2. This approach to modular exponentiation is known as a “fixed window” exponentiation.

1. Write a Sage function that, given an integer *x*, a modulus *N*, and a base *b*, computes a list of length b, where the *i*th element of the list is *xi*mod *N*. You may use the ModExp function or any other method to compute the exponentiation (but you don’t have to.)
2. Write a Sage function that takes an integer x, an exponent e, a base b, and a modulus N. This function should compute a power table using the function you wrote in part (a) and then use it by using the base b expansion of e to determine where to index into the table. You may use modular exponentiation, but only to calculate *yb* mod *N*, for any integer *y*.
3. Given that you use ModExp as a routine in the function you wrote in part (b) what can you conclude about the optimal.

Solution to Question 5)

The purpose of this question is to show how to generalize the Square and Multiply Exponentiation method to different radixes besides 2. This approach to modular exponentiation is known as a “fixed window” exponentiation.

1. Write a Sage function that, given an integer *x*, a modulus *N*, and a base *b*, computes a list of length b, where the *i*th element of the list is *xi*mod *N*. You may use the ModExp function or any other method to compute the exponentiation (but you don’t have to.)  
     
   Here’s an example of how you can do it with modular exponentiation:  
     
     
     
     
     
     
     
     
     
     
     
     
     
     
   Here is an example of how you can do it without relying on modular exponentiation:

def GeneratePowerTable(x, N, b):

r"""

Given an integer x, modulus N, and base b

this creates a list of length b

where the the element at index i (for i=0,1,2,...,b-1)

is x^i mod N

"""

power\_table = [1 for j in xrange(b)]

for j in xrange(1,b):

power\_table[j] = (x\*power\_table[j-1]) % N

return power\_table

def GeneratePowerTable(x, N, b):

r"""

Given an integer x, modulus N, and base b

this creates a list of length b

where the the element at index i (for i=0,1,2,...,b-1)

is x^i mod N

"""

power\_table = [0 for j in xrange(b)]

for j in xrange(b):

j = Integer(j)

power\_table[j] = ModExp(x, j, N)

return power\_table

1. Write a Sage function that takes an integer x, an exponent e, a base b, and a modulus N. This function should compute a power table using the function you wrote in part (a) and then use it by using the base b expansion of e to determine where to index into the table. You may use modular exponentiation, but only to calculate *yb* mod *N*, for any integer *y*.

def FixedWindowExp(x, e, N, b):

r"""

Given an integer x, exponent e,

modulus N, and base b

this function performs a fixed window exponentiation with base b

"""

power\_table = GeneratePowerTable(x, N, b)

exp\_t = e

exp\_list = []

# preprocess the exponent into a list with base b

while (0 != exp\_t):

(exp\_t, r) = exp\_t.quo\_rem(b)

exp\_list.append(r)

exp\_list\_c = len(exp\_list)

ret = 1

for j in xrange(exp\_list\_c):

ret = ModExp(ret, b, N)

offset = exp\_list[exp\_list\_c - j - 1]

ret = (ret\*power\_table[offset]) % N

return ret

1. Given that you use ModExp as a routine in the function you wrote in part (b) what can you conclude about the optimal.  
     
   Because you use modular exponentiation, you will still be doing square and multiply. Albeit on smaller numbers. So, the cheapest usage, is when b has very low hamming weight (number of one bits in the binary expansion.) The low hamming weight case is when b is a power of 2 (and hence only has a single bit set.)