**University of Pennsylvania**

**CIT 592: Fall 2011**

# Python Project: RSA Encryption

**Instructor: Donna Dietz**

You should first read in your text/texts and/or on Wikipedia for relevant background material. Your classnotes should also prove useful. For this project, you should start by cutting and pasting the code from your previous project into your new RSA.py file. Not all of the previous code will be used, but much of it will. The new tester retests the old project, so you should start off feeling like you're already partway into this project. (Well, you are!) Now, for the bad news. The new code will be roughly three times as long as the old code. But, you can do it! Let's start! Your project file will be named **RSA.py**. The name of the tester file is **testRSA.py**. You will probably want to import random and math.

For this project create these functions:

**def extractTwos(m):** m is a positive integer. A tuple (s,d) of integers is returned such that $m = 2^s d$ and d is odd. Bit operations make this job a breeze.

**def int2baseTwo(x):** x is a positive integer. Convert it to base two as a list of integers in reverse order. For example, int2baseTwo(6) = [0, 1, 1]

**def modExp(a,d,n):** returns $a^d \mod n$. Use the fast algorithm as explained in class. Do not just create a loop and keep multiplying. This will cause the program to take too long to run. (I mean *waaaaaaaay* too long!)

**def millerRabin(n, k):** returns True or False. If it returns True, n is probably a prime. If it returns False, n is definately composite. The value of k can be increased to make the test more accurate. This is probably the hardest method in the project. Read about this algorithm on Wikipedia. My code is about 20 lines long. Don't forget to test for even numbers greater than 2! Those return false immediately!!! (Two returns true immediately as well.)

**def primeSieve(k):** This is just for fun, really. You don't need to call it from any other function. This returns a list of length k+1 which begins $[-1, -1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1....]$. If $p$ is a prime, then the element with index $p$ should be a 1. If it is composite, it should be a 0. Indices zero and one are just set to -1 because 0 and 1 are neither prime nor composite.

**def findAPrime(a,b,k):** uses millerRabin to locate and return a prime roughly between a and b (although it can actually be larger than b). First, find a random number between a and b, then keep incrementing that value until you run into a prime. Remember, the prime number theorem says that the next prime is roughly $\ln(x)$ away, so you should have a good idea of how far you have to look. In my code, if I fail to find a prime after $10\ln(x)+3$ tries, I give up and return -1. (I don't think I've ever seen this happen yet, though!) The k input is just passed along to the millerRabin function.

**def newKey(a,b,k):** generates an RSA encryption/decryption key set. The values a, b, and k are passed directly to findAPrime, and two separate primes are generated. (You need to assure than you did not fail to find a prime by not looking far enough!) This function returns a 3-tuple of integers, containing the public modulus, the public encryption exponent, and the private decryption exponent respectively.

**def string2numList(strn):** converts a string to a list of integers, one integer for each character in the string. The values are the corresponding ASCII values for the characters. Only ASCII characters will be used in this project. Note: there are roughly 256 ASCII characters.

**def numList2string(L):** This is exactly the inverse function of string2numList.

**def numList2blocks(L,n):** takes a list of integers (each between 0 and 127), and combines them into blocks of size "n" using base 256. Note that you may need to add "spares" to L first in order to make this come out even. For example: [13,25,100,23] could become a block of size four: $13(256)^3+25(256)^2+100(256)+23$. This can be quickly done using bit operations.

**def blocks2numList(blocks,n):** This function is the inverse function of numList2blocks, with the natural caveat that there may be extra "trash" tacked onto the end of the resulting list which had been added to make the blocks come out even previously. And, of course, it *should* be trash; nothing predictable! The Germans lost WWII in part due to their consistent encrypted messages!

**def encrypt(message, modN, e, blockSize):** message is a string, the rest are integers. The message turns into a list of ascii numbers, then is grouped into blocks, and is then encrypted. A list of integers is returned.

**def decrypt(secret, modN, d, blockSize):** a plaintext message is returned. The input, secret, is the output of encrypt. d is the decryption exponent.

HAVE FUN!!!