Exercise: Encrypting and Decrypting with RSA

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
In this exercise, you will encrypt and decrypt numbers using a simple version of the RSA algorithm. Each team should have at least two members. ***One*** team-member should complete the instructions for Bob, then pass along some information (but not the whole sheet!) to **a different** team-member, who will complete the exercise as Alice. In this way, both team-members will play both roles in the exercise. You should conceal your actual numbers from your team-members. Try to perform the steps by hand if you can. For larger calculations feel free to write a short loop of code. You can do this in the Python interactive prompt or in a source file.

# Instructions for Bob:

We will be doing *B*=16-bit RSA.

1. Select the encryption exponent *e*=17. (In practice, e=65537 would often be used for larger *p* and *q*.)
2. Calculate *p* like this: (Write results in the table below)
   1. Create an 8-bit binary number by flipping a coin (or using some other random number generator) 5 times to select 5 random bits, and then append 11\_\_\_\_1 around your number. This forms our tentative *p.*  
      [To create the same sort of number in Python, first import random. Next, select an (*B*/2=) 8-bit random number using random.randint(i,j) to select an integer *x* satisfying i<=*x*<=j. Use bits() to view the binary form. Then, set the two highest bits and the lowest bit (to 1) using p = p | 0b11000001. (| is bitwise or)]
   2. Check if *p* is prime. If *p* is not prime, add 2 to *p* and try again. (You can check that a number is prime by hand, or use a program, e.g., check if all numbers smaller than p do not divide p. Avoid googling.).
   3. Check if the number (*p-1*) is co-prime with *e*=17, i.e., gcd(*p*-1, *e*)=1. If not, add 2 to *p* and try again. This step is necessary to ensure that we can find a d such that ed = 1 (mod z) Note: since e=17 is prime, you can simply check that (*p*-1) mod *e*≠0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial random number (decimal) | Initial random number (binary) | *p (decimal)* | *p (binary)* | Is *p* prime? | Is  (p-1)%e≠0 ? |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | Final: |  |  |  |  |

(Instructions for Bob, continued.)

1. Repeat step 2 to select *q*. (Note: *q* must be different from *p*. Start over if *q* will equal *p*.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial random number (decimal) | Initial random number (binary) | *q (decimal)* | *q (binary)* | Is *q* prime? | Is  (q-1)%e≠0 ? |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | Final: |  |  |  |  |

1. Calculate the modulus *n* = *p q*  
    *n* =
2. Calculate the totient z = (*p*-1) \*(*q*-1)  
    z =
3. Select the decryption exponent *d* such that (e \* d) mod z = 1. You can “guess and check” all values of 1 < d < z. Only one value of d will work if z is selected as in step 5, and p and q are selected as in steps 2 and 3. This is your private key. Do not reveal d, p, q, or z to Alice or Trudy!

d =

1. Provide your public key (e, n), in other words the numbers e and n, to Alice and Trudy. This simulates posting your public key for the world to see.
2. Wait for Alice to send you a secret message.
3. Once you receive the secret message from Alice, you can decrypt it using your private key. Suppose *c* is the ciphertext. Compute the original message *m* as *m* = *cd* mod *n*.

c = m =

Don’t reveal the secret message to Trudy!

# Instructions for Alice:

1. Record the name of the team member playing Bob here:
2. You will receive the public key (e, n), in other words the numbers e and n, from Bob. Write it here:

e = n =

1. Select any number 0 <= m < n for your plaintext secret message. If you like, you can encrypt a sequence of ASCII characters as separate messages *m* (that is, using block encryption.)

m =

1. Compute the ciphertext c as c = me mod n.

c =

1. Give the ciphertext message *c* to Bob and Trudy. This simulates Trudy eavesdropping on the wire.

**Additional Tips**

To computing me mod n and cd mod n can overflow your calculator especially for large values of d. There is a trick you can use that takes advantage of the fact that the modulus is taken of the result of the exponentiation. You can find out more here: <https://en.wikipedia.org/wiki/Modular_exponentiation>  
  
This pseudocode might be helpful:  
  
function modular\_pow(base, exponent, modulus) is  
 if modulus = 1 then  
 return 0  
 Assert :: (modulus - 1) \* (modulus - 1) does not overflow base  
 result := 1  
 base := base mod modulus  
 while exponent > 0 do  
 if (exponent mod 2 == 1) then  
 result := (result \* base) mod modulus  
 exponent := exponent >> 1  
 base := (base \* base) mod modulus  
 return result

# Instructions for Trudy: (optional)

(If you have extra time, you may want to play this role

1. Wait to receive the public key (e, n), in other words the numbers e and n from Bob.

e =

n =

1. Factor n to find p and q. Use a brute-force Python loop. This is the hard step that makes RSA secure for large numbers.

p =

q =

1. Compute z = (p-1) \* (q-1)

z =

1. Now compute *d* the same way as Bob did: Select the decryption exponent *d* such that (*e \* d*) mod z = 1. You can simply “guess and check” all values of d < z. Only one value of d will work for a given e and z.
2. Wait to receive (eavesdrop) on the ciphertext message *c* from Alice to Bob.

c =

1. Decrypt the *c*, ciphertext message: Compute the original message *m* as *m* = *cd* mod *n*.

m =

Acknowledgement: The simple form of the RSA encryption/decryption used in this exercise is based on Avi Kak’s lecture notes on cryptography, available at <https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf>

and from the text, Kurose & Ross, Computer Networking: A Top-Down Approach, 7th Edition, Section 8.2.2, pp. 606-610 (6th ed: same section, pp. 684-688).