Part 1
rsa.py

rsa_key_gen(p_txt, q_txt)

This function uses the PrimGenerator class to generate two primes p and q. It also checks the conditions that they are not equal, the gcd of (p, e) is not 1 and the gcd of (q, e) is not 1. If any of these checks are false, it generates new primes. This function also writes to the input files p_txt and q_txt the values of p and q once they are finished.

rsa_encrypt(message_txt, p_txt, q_txt, encrypted_txt)

This function first reads in p and q from their respective text files, as well as reading in the message from its file into a BitVector. It then reads from this bit vector 128 bits at a time. If this read bit vector is less than 128 bits, it pads from the right 0 bits until the bit vector is 128 bits. It then pads from the left 128 0 bits to make it 256 bits long. This function then uses the RSA formula, C = M ^ e % n on the 256 bit block. Where M is the 256 bit block, e is 65537 integer defined by the homework, n = p * q, and C is the cipher text output. It does this for every 128 bit block in the message file then writes the output cipher text to the encrypted_txt file in hex format.

rsa_decrypt(encrypted_txt, p_txt, q_txt, decrypted_txt)

This function first reads in p and q from their respective text files, as well as reading in the encrypted text from its file into a BitVector. It then reads 256 bits from the file at a time. Using this 256 block, we use the formula described in Lecture 12.5 mainly on page 36 which describes the Chinese Remainder Theorem. We first calculate vp and vq which are C ^ d % p and C ^ d % q. We then calculate xp and xq which are q * 1/q % p and p * 1/p % q. 1/q and 1/p are calculated using the multiplicative inverse function that I wrote in HW3. We then use the final formula, (vp * xp + vq * xq) % n and write this into a bit vector with a size of 256. Since we need to remove the padding to get the original plaintext, we use list indexing; [128:], to grab all the bits after the padding. I then write the plaintext output to the decrypted_txt file as text.

main():

Main parses the inputs and calls the respective function.

Part 2
breakRSA.py

This file uses modified functions from rsa.py which have different inputs or different outputs. The two functions needed are rsa_key_gen() and rsa_encrypt() from part 1.

breakRSA_encrypt(message_txt, enc1_txt, enc2_txt, enc3_txt, n_1_2_3_txt)

This function follows what is described in the hw pdf. Simply, it generates three encryptions of the same message file using randomly generated p and q pair values.

This function first creates three randomly generated p and q values. It then uses the rsa_encrypt function from part 1 to encrypt these messages into their respective output files. It also generates three n values based on n = p * q and writes them into its own file for the break function.

breakRSA_crack(enc1_txt, enc2_txt, enc3_txt, n_1_2_3_txt, cracked_txt)

The formulas used are from Lecture 11.7 pg 57 and Lecture 12.3.2 pg 25. We figure out from lecture 11.7 that we can reconstruct the message if the message “can be expressed as a product of n integers that are pairwise coprime.” Following down the proof we see that the final formula is

$$A \quad = \quad \left(\sum_{i=1}^{k} a_i \times c_i\right) \bmod M$$

where a_i is the respective bit vector blocks and c_i = Mi * Mi ^ -1 mod mi, where Mi is Ni and mi is ni. We define N to be the product of each n found in n_1_2_3_txt and we can define each Ni to be the product of each n that is not the same i. So N1 = n2 * n3 and so on. (We do this instead of doing N / ni because the division in python wasn’t working. We know that N / ni = ni2 * ni3 where ni2 and ni3 is not the original ni based on basic algebra) We then can use the formula for c_i = Ni * (Ni ^ -1 % ni), to solve for the coefficients.

This function first reads all the bit vectors from each file. It also reads the 3 N values from the n_1_2_3_txt file. It then does the c_i calculations to calculate c1, c2, and c3. We then read 256 bits from each bitvector. Using the formula above, we solve A using the c_i calculated before and where a_i is each block read from each file. a = (c1 * bitvec1.int_val() + c2 * bitvec2.int_val() + c3 * bitvec3.int_val()) % N. We know from lecture 12 that a = M^3 and that we need to take the cube root to finally solve for M. we use the solve_pRoot function for this. We do this until all files are done being read then we write the output to the cracked_txt file.

main():

Main parses the inputs and calls the respective function.