RSA explanation:

For encrypt I read in the p and q vars from files and made n = q*p. then I read from the bitvector formed from the plaintext file 128 bits at a time. I padded 128 bits to the right, and padded more if the chunk size after that was less than 256 to account for the end block. Then I raised that chunk to the power of self.e, modulus n, and the output was the result of that operation, written to the output file 128 bits at a time.

```
def encrypt(self, plaintext:str, ciphertext:str) -> None:
    # always pad from right, don't pad from left because it doesn't wo
    with open(sys.argv[4], 'r') as f:
        self.q = f.read()
    with open (sys.argv[3], 'r') as f:
        self.p = f.read()
    n = int(self.q) * int(self.p) # this is the mod
    plainbv = BitVector(filename = plaintext)
    with open(ciphertext, 'w') as f:
        while (plainbv.more_to_read):
            plainchunk = plainbv.read_bits_from_file(128)
            plainchunk.pad_from_left(128) # this is prepending right?
            if (plainchunk.size < 256):
                plainchunk.pad_from_right(256 - plainchunk.size) # for
            plainchunk = pow(plainchunk.intValue(), self.e, n)
            outchunk = BitVector(intVal = plainchunk, size = 256)
            f.write(outchunk.get_bitvector_in_hex())
```

Decrypt was very similar. I got the var d = e^-1 * (p-1)*(q-1), and recovered the plaintext by taking 256 bit chunks at a time, raising them to d, modulus n. then I cut it back down to 128 bits for the output BitVector, and wrote that to the output file to re-form the plaintext.

```
def decrypt(self, ciphertext:str, recovered_plaintext:str) -> None:
    with open(sys.argv[4], 'r') as f:
        self.q=f.read()
    with open(sys.argv[3], 'r') as f:
        self.p=f.read()
    n = int(self.p) * int(self.q)
    d = pow(self.e, -1, ((int(self.p)-1)*(int(self.q)-1))) # d = e^-1
    with open(ciphertext, 'r') as f:
        tempstring = f.read()
    bv = BitVector(hexstring = tempstring)
    with open("temp.txt", 'wb') as f:
        bv.write_to_file(f)
    cipherbv = BitVector(filename='temp.txt')
    with open(recovered_plaintext, 'w') as f:
        while (cipherbv.more_to_read):
            cipherchunk = cipherbv.read_bits_from_file(256)
            # no padding
            cipherchunk = pow(cipherchunk.intValue(), d, n) # M = C^d
            outchunk = BitVector(intVal = cipherchunk, size = 128)
            f.write(outchunk.get_bitvector_in_ascii())
```

CRT to break RSA:

I first read in the moduli from the n_1_2_3 file and multiplied them together to get the final large modulus N. Then I made an array N1N2N3 to hold N // Ni where Ni was one of the individual input moduli. Then I made array Ni_inv where I got the mult inverse of N1N2N3[i] in mod n. This prepared me for the loop where I iterated through the 256 bit input blocks. I iterated through all the input files at the same time using start and end block variables and splicing the bitvectors. Then I would make each chunk, for example: chunk1 = chunk1.int_val() * N1N2N3[0] * Ni_inv[0]. The other 2 were made likewise, then I added together the 3 chunks and modded by N. Then I took the cube root, and casted to a bitvector of size 128. I would then write that bitvector to output file in ascii, so the output was written 128 bits at a time.

Loop contents:

```
with open(cracked, 'w') as FILEOUT:
    while (currBlock < lastBlock): # enc1-3 should all be same ler
        chunk1 = text1[startblock:endblock]
        chunk2 = text2[startblock:endblock]
        chunk3 = text3[startblock:endblock]
        chunk1 = chunk1.int_val() * N1N2N3[0] * Ni_inv[0]
        chunk2 = chunk2.int_val() * N1N2N3[1] * Ni_inv[1]
        chunk3 = chunk3.int_val() * N1N2N3[2] * Ni_inv[2]
        M3 = (chunk1 + chunk2 + chunk3) % N
        M3 = self.solve_pRoot(3, M3)
        final = BitVector(intVal = M3,size =128)
        # print(final.get_bitvector_in_ascii())
        FILEOUT.write(final.get_bitvector_in_ascii())
        startblock += 256
        endblock += 256
        currBlock += 1
```