Make 3 Programs:
Keygen: Makes public and private Key
Encrypt: Takes public key and encrypts the message using the public key and creates the ciphertext
Decrypt: Decrypts with the private key

1. Chose 2 large prime numbers p and q
2. Compute the product of them n = pq. n is public modulus
3. Compute de is congruent to (≅) 1 modulus φ(n), where φ is Euler's quotient functions
    - φ(n) = φ(p) * φ(q) = > (p - 1) * (q - 1)
    - φ(p): How many numbers smaller than p are coprime with p, the gcd(x,p) = 1. It would be p - 1 because if it is prime the only common factor would be 1.
    - de ≅ mod ((p - 1) * (q - 1))

    - d,e are modular inverses
      ex: 5 * 8 ≅ 1 mod 13, 40%13 = 1
      Modular inverses can only exist if the number you are trying to get the modular inverse of is coprime with the number taking the mod by.

    - gcd(e, φ(n)) = 1 => Guarantees and inverse d
4. Encyrtion: c = m^e mod n
   Decryption: m = c^d mod n

   m is just a number.
   "Abc" => Ascii [97, 98, 99] => bits 1100010… taken as a single number m, then do m^n mod n = c

This assignment uses Carmicheal's which is λ
- λ(n) = lcm(λ(p),λ(q))
    lcm(p - 1, q - 1)
    λ(n)= Abs((p - 1)(q - 1))/gcd((p - 1)(q - 1)) //used for sra key generation
- p,q are large primes
- de ≅ mod (λ(n))
- gcd(e, λ(n)) = 1
- Encyrtion: c = m^e mod n
  Decryption: m = c^d mod n

M will be a very large number, in order to do so use Gnu Multiple Precision Library
p, q, e are pseudo-randomly generated

Files:
randstate.h - interface for modules goes here?
- Makes a struct for global random state named 'state' with Mersenne Twister Algorithm
- Mersenne Twister Algorithm: ???
- For gmp, use randstate_init(uint64_t seed) then call srandom(), cleared with randstate_clear()

randstate.c - implementation

Modular Exponentiation:
POWER-MOD(a,d,n): computes a^d mod n

- Used in both Encryption and Decryption

void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)

    // out = base^exponent mod modulus

```
int pow_mod(int out, int base, int exponent, int modulus) {
    int v = 1;
    while (exponent > 0) {
        if (exponent & 1) {
            v = (v*base) % modulus;
        }
        base = (base*base) % modulus;
        exponent = (int)(exponent/2);
    }
    return v;
}
```

void make_prime(mpz_t p, uint64_t bits, uint64_t iters):

    // generates prime number stored in p

- Urandomb: create a number of some specific bit length + offset
- If the number needs to be b bits long, value n has to be n >= 2^b + random number
- random number range[0, 2^(b-1)]
- Then check if it is prime

```c
int make_prime (int p, int bits, int iters) {
    bool has_prime = false; //Do not have a prime YET (growth mindset)
    int offset = pow (2, bits - 1); //Ex 4 bits: add 1000 or 2^3 or 8
    // Ex 4 bits: rand % 2^1 * 2 = rand % 4 => range [0, 3] * 2 + 1
    // Ex 4 bits: [1 to 7] odd numbers
    int random;         //will have the additional random
    int n;          //will have the possible values to check if prime

    while (!has_prime) {
        random = (rand () % (int) (pow (2, bits - 3))) * 2 + 1;

        n = offset + random;
        has_prime = is_prime (n, iters);
        printf ("%d prime %s \n", n, has_prime ? "true" : "false");
    }
    return n;

}
```

Primality Testing: Check if the big number p,q created are prime

MILLER-RABIN(n,k): Checks with a high probability that a number is prime

n == odd or n == 2
n-1 is even then = (2^s)r // r is odd

Ex:
N = 17, 17 - 1 = 16, 16 = (2^s)r
Repeatedly divide 16 by 2, s = 0
8, s = 1
4, s = 2
2, s = 3
1, s = 4, r = 1
16 = (2^4) * 1

Generate odd ([0, range/2] * 2) + 1: Maybe can be used to skip the r is odd checking

- Any number * 2 = even. Even + odd = odd

```
bool is_prime(int n, int iters) {

    int s = 0;
    int n_temp = n - 1;

    while (!(n_temp & 1)) {
        s++;
        n_temp = n_temp >> 1; //a number is only divisible by 2 for the number consecitive trailing 0's
    }

    int r = n_temp;
    int j; //will be array pointer? gmp
    int y; //will be array pointer? gmp
    for (int i = 1; i < iters; i++) {
        int a = rand() % (n - 5) + 2; //[0, n-4] + 2 = [2, n-2]
        y = pow_mod(0, a, r, n);

        while (j <= s - 1 && y != 1) {
            j = 1; //j <= 1
            y = pow_mod(0, y, 2, n);
            if (y == 1) return false;
            j = j + 1;
        }
        if (y != n - 1) return false;
    }
    return true;
}
```

MOD-INVERSE(a,n)
void mod_inverse(mpz_t i, mpz_t a, mpz_t n) // i = a mod n

GCD(a, b)
    while(b != 0):
        t = b
        b = a%b
        a = t
    return a

LCM(a, b)
    return abs(a * b)/gcd(a, b)

void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)
    // create public key
- Specify number of bits n // keygen program has parameter to specify nbits
    - find num bits of q and q, n = pq
    - pbits = random [nbits/4, 3*nbits/4] => [0, nibits/2] + nbits/4
      qbits = nibits - pbits
      // pbits and qbits is the bits long for make_prime
      // Call make prime for p and q

```c
int n_bits = 11; //User input
int iters = 6; //User input iterations

int p_bits = rand()%(n_bits/2) + (n_bits/4);
int q_bits = n_bits - p_bits;

int p = make_prime(0, p_bits, iters);
int q = make_prime (0, q_bits, iters);

printf("p_bits %d, p: %d\n", p_bits, p);
printf("q_bits %d, q: %d\n", q_bits, q);

int n = p*q;
printf("n: %d, %d * %d\n", n, p, q);
return 0;
```

- Compute lcm (p - 1, q - 1), find suitable public exponent e. gcd(e, lambda(n)) == 1
    - Make a suitable_e function, 65537 (prof uses)

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)

- Got p, e, q from make_pub
- Find d such that, de ≅ 1 mod (λ(n)) //call modular inverse
  From that find d

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)

- Format n, e, s, username, each with trailing newline n,e,s should be in hexstrings
  fprintf("%x\n %x\n %x\n", n, e, s)
  fprintf("%c", username)

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)

- Get n,d?

void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)

- Encyrtion: c = m^e (mod n)

void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)

- Data in infile must be encrypted in blocks, because %n

- Value of a block cannot be 0 or 1, solve this by prepending 0xFF
- Block Size k = floor((LogBase2 (n) - 1)/8) // -1 is room for the 0xFF

  j = 0; //counts the number of bytes read
  while(!EOF):
  read (k-1) bypes from infile
  j+= bytes read this iteration

  mpz_import() //converts to the large number m
  rsa_encypt(m)
  fprintf(message + “\n”, outfile);

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)

- Decryption: m = c^d mod n

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)

- Block Size k = floor((LogBase2 (n) - 1)/8)
  Allocate k bytes type (uint8_t *)
- Make hextring into a number
- Decrypt
- mpz_export(), convert c back into bytes
- fprintf(k - 1 bytes, outfile)

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)

- signature s = (m^d) mod n
  - d private key d //only you should have the private key
  - Sign your username: akbalakr => m

bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)

- s^e mod n
  - m = m’?