Alex Asch

Assignment 6 Design

Description of Program

Deliverables

- encode.c
  - This contains the implementation of the Huffman encoder
- decode.c
  - This contains the implementation of the Huffman decoder.
- defines.h
  - This declares the macros necessary for completing the assignment. This is given in the resources repository by Professor Long
- header.h
  - This contains the struct definition for a file header
  - This is given by Professor Long in the resources repository.
- node.h
  - This file contains the interface for the node ADT.
  - This is given by Professor Long in the resources repository
- pq.h
  - This contains the interface to the ADT queue.
  - This is given by Professor Long in the resources repository
- code.h
  - This file will contain the code ADT interface.
  - This is given by Professor Long in the resources repository
- io.h
  - This file will contain the I/O module interface.
  - Given in the resources repository by Professor Long.
- stack.h
  - This file will contain the stack ADT interface.
  - Given in the resources repository by Professor Long.
- huffman.h
  - This file will contain the Huffman coding module interface. This file will be provided.
  - This code is given in the resources repository by Professor Long.
- node.c
  - This file will contain the implementation of the node ADT/
- pq.c
  - This file will contain the implementation of the priority queue ADT.
- code.c
  - This file will contain the implementation of the code ADT.
- io.c

    - This file will contain the implementation of the input and output module.
- huffman.c
    - This file will have the implementation of the Huffman coding module interface.
- Makefile
    - Make the above program.
    - Links the above files
- README.md
    - A document in markdown syntax on the usage of the program.
- DESIGN.pdf
    - The pdf being viewed at the moment (this document).

Notes

- Public key cryptography by RSA method.
    - First choose two large primes p and q.
    - Then calculate product, n = pq
    - Very very computationally hard to do largest prime factorization
    - Computationally infeasible for n factor p and q
    - Calculate the totient of n, which is (p-1)*(q-1)
    - coprime aka relative prime mean gcd is 1
    - Then choose the public exponent
        - This e is some exponent
        - Calculate inverse of e which is e-1 modulo totient n
        - This means d*e is equal to 1 modulo totient(n)
        - 5 mod 13, then inverse of 5 is 8, 5 inverse mod 13 congruent 8 mod 13
        - This is because 5*8 = 40 mod 13, which is 1 mod 13.
    - Using this RSA is computed as
        - d is private key
        - n and e are public key
        - This is RSA
    - The encryption of a message m is m^e mod(n)
    - decryption of said encryption is encryption raised to d mod n.
- Keygen makes the keys
- Encryptor encrypts files
- decryptor decrypts files
- The stuff should be gigantic
- c numbers small we need big so we use gmp
- We need to make a random state
- This is step one
- **randstate.c**
    - Using a seed we make a random number using mersenne twister algo
    - Then set seed using passed in seed

- then we can clear the random state
- use gmp lib funcs
- **Modular exponentiation**
  - pow mod
  - compute a base raised to exponent power modulo modulus, store computed result in out
- **Testing primality**
  - don't do it the normal deterministic way
  - Use a randomized algorithm with high probability of something being prime
  - Miller Rabin primality test
  - first find an s and r such that n-1 = 2^s*r and r is odd
    - Basically, while not found, divide n-1 by 2^s until r is odd, then when it is like that use s and 3.
  - take a k number of iterations and do it and return if.
  - is prime? take a number, check if it is prime using miller rabin
  - make_prime, generates a prime that is at least bits number of bits long
- **GCD**
  - Euclidian algorithm of calculating gxd
  - We also need a function to compute modulo inverse of something and mod n
  - If no inverse found set i to 0
- **RSA lib**
  - Creates parts of a new RSA pub key, two large primes p and q, their product n and the public exponent e
    - First make primes p and q using make_prime such that log2(n)) >= nbits
    - Let the number of bits for p be a random num in the range nbits/4 to ¾ nbits, the remaining bits will go to q, the number of miller rabin iterations is specified by iters.
    - next to the totient of p and q
    - We need a suitable public exponent e, generate random nums of around nbits using mpz_urandomb() compute the gcd of each rand num aand the computed torrent stop the loop when found a coprime with totient
  - Write public key
    - Write the key to a pbfile for public file
  - Read pub key
    - Read key from file
  - make a private key and also wr the priv key
  - **RSA encrypt**
    - Take m to the e mod n and put that into c,
    - encrypt file
    - read in bytes of a file

        - Read a block of bytes and turn the bytes into a number
        - We need to guarantee that m is less than n
        - Calculate block size, that is log2(n) -⅛
        - Dynamically allocate an array of that block size
        - Pad a 0xFF to the beginning of the block
    - Decrypt
        - Computing message m by decrypting ciphertext using priv key and public modulus n
        - rsa_decrypt file
            - Decrypts the contents of infile writing the contents to the outfile
            - Decrypt in blocks, array of block size
            - scan in hexstring saving it as a new mpz
            - use mpz_export to make c back into bytes and store them in allocated block, let j be num bytes converted, you want to set the order param for mpz_export() to 1 for most significant word for, 1 for endian and 0 for nails
            - write out j-1 bytes starting from index 1 of the block to outfile
            - This is because index 0 is prepended 0XFF don't print the 0xFF.
    - RSA_sign
        - produce signature x by signing message with private key and public modulus, signing is s(m) = s = m^d(mod n)
        - proves sender.
    - s is going to be x to the d mod n
    - v = y to the e mod n
    - signing s = x to the d mod n
    - verify r is y to the e mod n
    - inverse operations of encrypt decrypt sign verify

Psuedo

- randstate_init
    - Take the extern variable and initialize it with gmp_randinit_mt()
    - call gmp_randseed_ui() to set the seed
- randstate_clear
    - gmp_randclear the extern var.
- pow_mod
    - Does fast modular exponentiation, base raised to the exponent power module modulus and storing the computed result in out.
    - make a temp variable of 1
    - make a placeholder copy of a
    - while d is greater than 0
        - IF 2 mod 2 is 1

            - v is v times p mod n
        - Then, p is p^2 modulo n
        - Half d
    - then return v
    - Because gmp, instead of return, assign the value to the given output variable. This convention will remain true for the rest of the num_theory functions
- is_prime
    - find if n is prime in k numbers of iters.
    - First, find and r and s such that n(being the number to test) - 1 = 2^s*r and r is odd.
        - make a temp var for n-1
        - To do this, make a while loop such that until an odd r is found, divide (n - 1) by 2, starting at division by one.
        - Have a counter variable starting at zero count each iteration.
        - This counter variable is s.
        - Once odd n is found keep s as s and r as r.
    - For i in range of 1 to k(inclusive)
        - Generate a random number, modulo (n-4) +2, this makes a random number between 2 and n -2
        - y equals the power_mod of a, r and n
        - if y is not 1 and y is not n - 1
            - new variable j is 1
            - while j is less than s minus one, and y is not equal to n -1
                - y is the power modulo of y^2 mod n
                - if y ==1
                    - return false;
            - if y does not equal n -1 return false
    - return true
- make_prime
    - generates a new prime number that is at least bits number of bits, using mpz_librandomb
    - check if it is prime using inters number of generations
    - if not prime, generate and check again.
- gcd
    - take the greatest common denominator of a and b, store it in d and return it.
    - while b is not zero
        - store b into t
        - b is a modulo b
        - store t into a
    - return a

- mod_inverse
  - take i and a and n, compute the modular inverse of of a mod n,
  - Pseudo will be written assuming parallel assignment is allowed, every time we do so, in reality a temp variable is used.
  - Set r to n, r' to a
  - set t to 0 t' to 1
  - while r' is not 0
    - q is r/r'
    - parallel assign r to r', r, to r - q times r'
    - t = t' t' = t- q x t'
  - if r is greater than 1
    - Set i to zero for no inverse
  - if t is less than 0
    - t is t +n
  - return t through i
- rsa_make_pub
  - Creates two new primes, p and q, the bits for p will be a random number in the range nbits/4 and 3nbits/4.
  - Next compute totient of pq
  - While (no prime exp)
    - generate a new random number
    - is the gcd of totient and the random number 1?
      - Then we found the prime exponent
- rsa_write_pub
  - Write the public key values, then the username to the outfile
  - Write each with newline after
- rsa_make_priv
  - compute the inverse modulo of the totient and e
- rsa_write_priv
  - Write n then d, both with trailing newline
- rsa_read_priv
  - Reads a private key from pvfile, the format of a private key should be n then d, read both of these values
- rsa_encrypt
  - Takes a message as a number m, then compute the ciphertext as c = m^e (mod n)
- rsa_encrypt file
  - Encrypts the contents of input file to the output file with mod n
  - Step one, set a block size var k = floor of log2(n-1)/8
    - do n - 1, find the then the size in base 2 aka the size in bits of the number
    - then divide that number by 8 and set it to k

            - Dynamically allocate an array that can hold k bytes
                - array of type uint8_t
            - Set byte one of the array to 0xFF
            - fread() from k -1 bytes from the infile and set j to the number of bytes returned
            - for i in range 1 to block put the bytes into the buffer
            - using mpz_import, convert the buffer to a number
            - pass this number to rsa encrypt with n and e
            - Take the encrypted number, write it, then a newline
- RSA decrypt
    - compute message from ciphertext, d, and n
    - powermod of m to the d mod n
    - return the result
- rsa_decryptor
    - Calculate block size, same as above
    - Make an array that can hold k bytes to hold the block
    - While the number returned by mpz_inp_str >0
        - use mpz_inp_str to read the hexstring
        - decrypts the given hexstring with RSA decrypt
        - mpz_export the hexstring into the allocated array
        - use fwrite to write from the array (minus the first element) to the number of bytes read from mpz_inp_str to outfile
- rsa_sign
    - Take message s and sign it by doing powermod of the message to the dth mod n
    - return that in s
- rsa_verify
    - if t is equal to s to the e mod n, return true, else return false

KEYGEN

- parse command line opts, same as prev assignment
- fopen() the given output files for pub and priv key
- fchmod() the private key file permissions to 0600 also know as rw for use only
- using seed from cmd line initialize a random state with randstate_init()
- make the public and private keys using respective functions
- getenv() the current environment
- Coonvert the username with mpz_set_str() as a base 62
- Then use rsa_sign to compute the signature of the username
- Write the computed public and private keys to their respective files
- If verbose is enabled
    - print username, signature, p ,q, n, e, and d
    - Print these in decimal with number of bitse

- Close the files and randstate_clear

ENCRYPTOR

- open the public key file using fopen()
- Read the entire public key file using mpz_inp_str() to read each field n, e, s, and username
- Convert the username that was read in to an mpz_t
- Verify the signature with rsa_verify
- Encrypt the file using rsa_encrypt file

DECRYPTOR

- Open the private key file, then read the private key
- Decrypt the file with a call to rsa_decrypt