SSH `keygen`

The purposes of this project are to (1) explore cryptography, and (2) implement the algorithms that back semi-numerical operations.

Project Details

In this project, you will write the fundamental pieces of modern authentication protocols. In particular, you will build a multi-precision number library, and implement RSA key generation.

Project Restrictions

You *must* implement this project using the *C Programming Language*. You may use the provided *stripped down* version of `mbedtls` in addition to `libc`. We will use C to match the low level of the algorithms we will be working with. Please note that we are implementing a stripped down version of an open source library; this means a solution to the project is readily available on the internet. It would, clearly, be an honor code violation to look at or use this (or any other) implementation.

Deliverables

- A multi-precision integer library (`libbigint`)
- An RSA Key Generator (`keygen`)

Part 1: `libbigint`

In this part, you will implement a multi-precision integer library which will get used as the backing for the key generation utility. Your library must implement the following public interface in accordance with the specifications given with each function. Note that many functions return ints as error values; possible values are specified in `bigint.h`.

`bigint` Representation

We represent a big integer as an array of 64-bit "digits", a sign bit, and a number of "digits". Conceptually, we are storing the number in base 2^{64} which is convenient because it matches the instruction width of the machines we will use. Because "digits" usually refer to base 10, we will use the (somewhat) standard terminology "limb" to refer to the equivalent idea in a multi-precision integer. We call the limb type `big_uint`.

Task 1.1: Lifecycle Functions

```
void big_init(bigint *X)
```

Initializes `X` to be ready for other bigint operations.

```
void big_free(bigint *X)
```

Frees all memory associated with the bigint `X`.

```
int big_copy(bigint *X, const bigint *Y)
```

Replaces the value of the bigint `X` with the value of the bigint `Y`. `Y` should remain unchanged.

```
size_t big_bitlen(const bigint *X)
```

Returns the smallest number of bits necessary to represent `X`. That is, returns the number of bits in the value of `X` up to and including the most-significant 1.

```
size_t big_size(const bigint *X)
```

Returns the total size of `X` in bytes. Leading 0 bytes should not be included.

```
int big_set_nonzero(bigint *X, big_uint limb)
```

Set `X` to a single nonzero limb.

Task 1.2: I/O Functions

```
int big_read_string(bigint *X, const char *s)
```

Sets the value of `X` to the value of the hexadecimal string, `s`. Returns 0 on success and an appropriate error code on failure. Must support lowercase hexadecimal. May also support uppercase or a mix.

Since *exactly 16* hexadecimal digits make up a 64-bit limb, one approach might use the standard base conversion algorithm on groups of 16 hex digits. For example, we convert the hexadecimal number `1234567890abcdef1234` to two limbs by

(1) padding it to a multiple of 16 digits: `00000000000001234567890abcdef1234`

(2) chunking it into groups of 16

(3) converting each group to base 10 using the standard hex-to-decimal conversion.

Your algorithm should correctly handle leading zeroes.

```
int big_write_string(const bigint *X, char *buf, size_t buflen,
                     size_t *olen)
```

Writes the value of `X`, as a *lowercase* hexadecimal string (including null terminator), to the buffer `buf`. Returns 0 on success and an appropriate error code on failure. Regardless of result, sets `*olen` to the number of bytes that *should* have been written (including null terminator).

Repeatedly mod and divide each limb by 16, 16 times for each limb. Your algorithm should make sure to remove all leading zeroes.

Most tests will depend on **fully correct** implementations of read and write. Be sure to write your own extensive tests for them!

```
int big_read_binary(bigint *X, const unsigned char *buf, size_t buflen)
```

Sets the value of `X` to the value of the buffer `buf`, where `buf` is a big endian (most-significant-byte first) representation of `X`. Returns 0 on success and an appropriate error code on failure.

The representation should match the base-256 representation expected in SSH keys, as discussed in class.

`int` **`big_write_binary`**`(`**`const`** `bigint *X, unsigned char *buf, size_t buflen)`

Writes the value of `X`, as a big endian (most-significant-byte first) string, to the buffer `buf`. Returns an appropriate error code if the binary representation of `X` does not fit in `buflen` bytes.

The representation should match the base-256 representation expected in SSH keys, as discussed in class.

Task 1.3: Core Operations

`int` **`big_add`**`(bigint *X,` **`const`** `bigint *A,` **`const`** `bigint *B)`

Sets the value of `X` to the sum of the values of `A` and `B`. Note that `X`, `A`, and `B` might refer to the same location in memory. Returns 0 on success and an appropriate error code on failure.

You should use the grade-school addition algorithm described in the algorithms document.

`int` **`big_sub`**`(bigint *X,` **`const`** `bigint *A,` **`const`** `bigint *B)`

Sets the value of `X` to the difference of the values of `A` and `B`. Note that `X`, `A`, and `B` might refer to the same location in memory. Returns 0 on success and an appropriate error code on failure.

You should use the grade-school subtraction algorithm analogous to the addition algorithm.

`int` **`big_cmp`**`(`**`const`** `bigint *X,` **`const`** `bigint *Y)`

Returns the following:

- 0 iff $\texttt{X} = \texttt{Y}$
- -1 iff $\texttt{X} < \texttt{Y}$
- 1 iff $\texttt{X} > \texttt{Y}$

`int` **`big_mul`**`(bigint *X,` **`const`** `bigint *A,` **`const`** `bigint *B)`

Sets the value of `X` to the product of the values of `A` and `B`. Note that `X`, `A`, and `B` might refer to the same location in memory. Returns 0 on success and an appropriate error code on failure.

You should use the grade-school multiplication algorithm discussed in the algorithms document. A possible above-and-beyond extension would be to implement a switch to Karatsuba multiplication, Toom-Cook multiplication, and/or FFT multiplication at reasonable thresholds.

`int` **`big_div`**`(bigint *Q, bigint *R,` **`const`** `bigint *A,` **`const`** `bigint *B)`

Sets the value of `Q` to the quotient of the values of `A` and `B`. Sets the value of `R` to the remainder of the values of `A` and `B`. If `Q` or `R` is `NULL`, this function does not compute the corresponding value. Note that `Q`, `A`, and `B` might refer to the same location in memory. Returns 0 on success and an appropriate error code on failure or if the value of `B` is 0.

You should use grade-school division algorithm described in the Handbook of Applied Cryptography. A possible above-and-beyond extension would be to implement Burnikel-Ziegler division.

Task 1.4: Modular Operations

`int` **`big_gcd`**`(bigint *G, const bigint *A, const bigint *B)`
Sets the value of `G` to the greatest common divisor of the values of `A` and `B`.
We recommend that you implement this function iteratively (i.e., not recursively).

`int` **`big_inv_mod`**`(bigint *X, const bigint *A, const bigint *N)`
Sets the value of `X` such that, for some $k \in \mathbb{Z}$, $\texttt{AX} + k\texttt{N} = 1$.
You should use the extended euclidean algorithm to implement this. Again, you should *not* do it recursively.

`int` **`big_exp_mod`**`(bigint *X, const bigint *A, const bigint *E,` `const bigint *N, bigint *_RR)`
Sets the value of `X` to the value of $\texttt{A}^{\texttt{E}}$ mod `N`
Your implementation must use repeated squaring and Montgomery Multiplication as discussed in the algorithms document.

Task 1.5: Primality Operations

`int` **`big_is_prime`**`(const bigint *X)`
Returns 1 if the value of `X` is "probably prime".
Your implementation should use the Miller-Rabin algorithm with a pass over small potential divisors.

`int` **`big_gen_prime`**`(bigint *X, size_t nbits)`
Sets the value of `X` to a *random* prime with approximately `nbits` bits.
Make sure to set the lowest bit to 1!

Part 2: `keygen`

In this part, you will implement RSA key generation which is compatible with the standard `OpenSSH` implementation of `ssh`.

Your `bigint` library has almost everything you need to do key generation. We have provided a private key implementation (`rsa_private_key.c`), a base64 implementation (`base64.c`), and a main (`keygen.c`) The only remaining things to implement are the actual key generation, and writing the public key.

`void` **`rsa_init`**`(rsa_context *ctx)`
Initializes `ctx` to be ready for other RSA operations.

`void` **`rsa_free`**`(rsa_context *ctx)`

Frees all memory associated with `ctx`.

`int` **`rsa_gen_key`**`(rsa_context *ctx,` **`size_t`** `nbits, uint64_t exponent)`

Generate an RSA private key with the requested number of bits, and $e =$ exponent. Return 0 on success and an appropriate bigint error code otherwise.

Follow the restrictions from the notes, reproduced here:

- $gcd(e, \phi(n)) = 1$ (otherwise, we wont be able to find d)
- d should be calculated as e^{-1} mod lcm$(p-1, q-1)$ (this leads to a *slightly* better d than ϕ does)
- We insist $d > 2^{\text{bits}/2}$ to avoid attacks abusing small d
- We insist $|p - q| > 2^{\text{bits}/2-100}$ to avoid Fermats factorization attack

Additionally, the RSA context contains some intermediate computations for the private key, which should be filled in upon key generation.

`int` **`rsa_write_public_key`**`(`**`const`** `rsa_context *ctx, FILE *file)`

Write the public key to the given file, following the public key format described in the RSA keygen document.

The cool part is actually trying your SSH key. To do this, copy your public key to a remote server using the `ssh-copy-id` command, and then use the `-i` option of `ssh` to use your key. If it doesn't ask you for a password, then it worked!