CSC222 Fall 2012
Data Structures & Algorithms II
Programming Project # 1

RSA Encryption

Implement RSA encryption. For this project, you will need large-integer arithmetic. Fortunately, folks in the open source community have saved us a lot of work by writing GMP, the open source GNU Multi-Precision library.

If you are running Ubuntu, you can install the package with:

```
sudo apt-get install libgmp3-dev
```

If you want to use the C++ bindings and overloaded operators, you should also install:

```
sudo apt-get install libgmpxx4ldbl
```

Your implementation will need to:

- Generate two large prime numbers.
  - For simplicity, let us agree to use 32-bit prime numbers, giving a 64-bit modulus
- Compute public and private keys
  - Let us agree to use a small prime for the exponent e.
- Perform encryption and decryption
  - Process messages larger than the modulus

For this project you will need write three programs:

- **gen_key**
- **rsa_encrypt**
- **rsa_decrypt**

We describe the input and output requirements for each program:

gen_key:

Input: None

Output: The program will create two files named `public_key` and `private_key`. The file `public_key` contains one line with two base 10 numbers: The modulus N and the exponent e. The modulus must be greater than 9223372036854775808. For example:

```
13288485248123405329 7
```

The file `private_key` contains one line with the exponent d in base 10. For example:

```
    1898355034404673463
```

rsa_encrypt:

Input: The input is a sequence of characters read from standard input, ending at EOF. We will run our programs using UNIX re-direction. For example:

```
    rsa_encrypt < clear_text
```

Since we are using 64 bit encryption, we must process the input characters in chunks of 8 characters. Each input character should be treated as a single base 256 digit. Use Horner's algorithm to compute a single number representing the chunk of 8 characters.

Output: The program `rsa_encrypt` will write a file named "cipher_text" The output should represent the encrypted form of each 8 byte chunk as a string of 16 hexadecimal digits. Write only four strings of hexadecimal digits per line. See the documentation for function `mpz_out_str()` for an easy way to output to a file in base 16. For example:

```
66f11684a3d9cdb1 27eddc090bb990f3 5a3d65cd1fcaec2 908f14466e578340
923475dd8915611d 5f76ef60e032b29d 81fd70d4023e41f7 a64c933882aa2d07
2c8e61d7ecf143a3 4e6e0a0c6cacbe33 4ddd0cac8782bc5e 4d6a37c11806505d
1ebd5a725e886a5 b3d652869bba5bf5 802462089af5d444
```

rsa_decrypt:

The function `rsa_decrypt` should read the file "cipher_text", decrypt the cipher text (using the private key), and write the result to standard output. Your implementation of `rsa_decrypt` must be compatible with your implementation of `rsa_encrypt`.

Using your implementation of `rsa_decrypt` and the private key given above to decode the example cipher text. Turn in your result (the message in clear text) with your program.

Bonus challenge: Assuming you did not know the private key, try to break the code. I.e., factor $N = 13288485248123405329$ into the product two primes.