# 信息安全概论实验

RSA算法实现

姓名：赵浩宁

学号：1140310226

# 一 实验目的

1. 了解非对称加密机制
2. 理解RSA算法的加解密原理
3. 实现RSA算法的加解密过程

# 二 实验背景

在公开密钥密码体制中，加密密钥(即公开密钥)PK是公开信息，而解密密钥(即秘密密钥)SK是需要保密的。加密算法E和解密算法D也都是公开的。虽然秘密密钥SK是由公开密钥PK决定的，但却不能根据PK计算出SK。正是基于这种理论，1978年出现了著名的RSA算法，它通常是先生成一对RSA 密钥，其中之一是保密密钥，由用户保存；另一个为公开密钥，可对外公开，甚至可在[网络服务器](http://baike.baidu.com/view/813.htm)中注册。为提高保密强度，RSA密钥至少为500位长，一般推荐使用1024位。这就使加密的计算量很大。为减少计算量，在传送信息时，常采用传统加密方法与公开密钥加密方法相结合的方式，即信息采用改进的DES或IDEA对话密钥加密，然后使用RSA密钥加密对话密钥和信息摘要。对方收到信息后，用不同的密钥解密并可核对信息摘要。

RSA算法是第一个能同时用于加密和数字签名的算法，也易于理解和操作。RSA是被研究得最广泛的[公钥](http://baike.baidu.com/view/355291.htm)算法，从提出到现在的这么多年里，经历了各种攻击的考验，逐渐为人们接受，普遍认为是目前最优秀的公钥方案之一。

# 三 RSA算法

RSA算法是一种非对称密码算法，所谓非对称，就是指该算法需要一对密钥，使用其中一个加密，则需要用另一个才能解密。

RSA的算法涉及三个参数，n、e1、e2。

其中，n是两个大质数p、q的积，n的二进制表示时所占用的位数，就是所谓的密钥长度。

e1和e2是一对相关的值，e1可以任意取，但要求e1与(p-1)\*(q-1)互质；再选择e2，要求(e2\*e1)mod((p-1)\*(q-1))=1。

（n，e1),(n，e2)就是密钥对。其中(n，e1)为公钥，(n，e2)为私钥。[1]

RSA加解密的算法完全相同，设A为明文，B为密文，则：A=B^e2 mod n；B=A^e1 mod n；（公钥加密体制中，一般用公钥加密，私钥解密）

e1和e2可以互换使用，即：

A=B^e1 mod n；B=A^e2 mod n;

# 四 程序接口

1. 算法设计
2. pvoid bignum\_iadd(bignum\* source, bignum\* add);

void bignum\_add(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_isubtract(bignum\* source, bignum\* add);

void bignum\_subtract(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_imultiply(bignum\* source, bignum\* add);

void bignum\_multiply(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_idivide(bignum\* source, bignum\* div);

void bignum\_idivider(bignum\* source, bignum\* div, bignum\* remainder);

void bignum\_remainder(bignum\* source, bignum \*div, bignum\* remainder);

说明：随机产生一个高精度大整数，定义大整数的相关操作。

1. void bignum\_imodulate(bignum\* source, bignum\* modulus);
2. void bignum\_divide(bignum\* quotient, bignum\* remainder, bignum\* b1, bignum\* b2);

说明：利用Java语言的中的java.math.BigInteger类的方法中随机产生大数。

1. pvoid randPrime(int numDigits, bignum\* result)

int probablePrime(bignum\* n, int k)

int solovayPrime(int a, bignum\* n)

说明：着三个函数是，费马素性检验是一种随机化算法，判断一个数是合数还是可能是素数

根据费马小定理：如果p是素数，<math>1 \le a \le p</math>，那么

<math>a^ \equiv 1 \pmod</math>。

如果我们想知道n是否是素数，我们在中间选取a，看看上面等式是否成立。如果对于数值a等式不成立，那么n是合数。如果有很多的a能够使等式成立，那么我们可以说n 可能是素数，或者伪素数。

在我们检验过程中，有可能我们选取的a都能让等式成立，然而n却是合数。这时等式

<math>a^ \equiv 1 \pmod</math>

被称为Fermat liar。如果我们选取满足下面等式的a

<math>a^ \not\equiv 1 \pmod</math>

那么a也就是对于n的合数判定的Fermat witness。

整个算法可以写成是下面两大部：

输入：n需要检验的数；k：参数之一来决定检验需要进行的次数。

输出：当n是合数时，否则可能是素数：

重复k次：

在[1, n − 1]范围内随机选取a

如果an − 1 mod n ≠ 1 那么返回合数

返回可能是素数

若使用模指数运算的快速算法，这个算法的运行时间是O(k × log3n)，这里k是一个随机的a需要检验的次数，n是我们想要检验的数。

众所周知，对于卡米歇尔数n，全部的a都会令gcd(a,n)=1，我们称之为fermat liars。尽管卡米歇尔数很是稀有，但是却足够令费马素性检验无法像如米勒-拉宾和Solovay-Strassen的素性检验般，成为被经常实际应用的素性检验。

一般的，如果n 不是卡米歇尔数，那么至少一半的

<math>a\in(\mathbb/n\mathbb)^\*</math>

是 Fermat witnesses。在这里，令 a 为Fermat witness 、 a1, a2, ..., as 为 Fermat liars。那么

<math>(a\cdot a\_i)^ \equiv a^\cdot a\_i^ \equiv a^ \not\equiv 1\pmod</math>

所有的 a × ai for i = 1, 2, ..., s 都是 Fermat witnesses

1. void randExponent(bignum\* phi, int n, bignum\* result)

说明：随机产生一个与φ(n)互为素数的密钥

1. int readFile(FILE\* fd, char\*\* buffer, int bytes)

说明：对text.txt文件里的明文进行读取

Fd是文件指针，buffer是申请的内存块，bytes是文件字节数

1. int \*decodeMessage(int len, int bytes, bignum \*cryptogram, bignum \*exponent, bignum \*modulus)

方法名称：解密算法。

参数说明：

Len是密文长度

Bytes是密文字节数

Cryptogram是私钥

Exponent是公钥

Modulus是

PrimeP和PrimeQ是由GetPrime方法产生的两个大素数。

n是由p,q大素数得到的乘积

算法：

解密 (用d,p,q)

密文：C，明文：M=Cd(mod n).

注意：加密和解密是一对逆运算

1. bignum \*encodeMessage(int len, int bytes, char \*message, bignum \*exponent, bignum \*modulus)

函数名称：加密算法。

参数说明：

len是明文长度

byte是明文字节数

message是明文字符串

exponent是公钥

modulus是由p,q大素数得到的乘积

算法：

加密 (用e,n)

明文：M<n，密文：C=Me(mod n).

在对称加密中：n，d两个数构成公钥，可以告诉别人；n，e两个数构成私钥，e自己保留，不让任何人知道。给别人发送的信息使用e加密，只要别人能用d解开就证明信息是由你发送的，构成了签名机制。别人给你发送信息时使用d加密，这样只有拥有e的你能够对其解密。

RSA的安全性在于对于一个大数n，没有有效的方法能够将其分解从而在已知n，d的情况下无法获得e；同样在已知n，e的情况下无法求得d。

# 五 实验源代码

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <limits.h>

/\* Accuracy with which we test for prime numbers using Solovay-Strassen algorithm.

\* 20 Tests should be sufficient for most largish primes \*/

#define ACCURACY 20

#define FACTOR\_DIGITS 100

#define EXPONENT\_MAX RAND\_MAX

#define BUF\_SIZE 1024

/\* Initial capacity for a bignum structure. They will flexibly expand but this

\* should be reasonably high to avoid frequent early reallocs \*/

#define BIGNUM\_CAPACITY 20

/\* Radix and halfradix. These should be changed if the limb/word type changes \*/

#define RADIX 4294967296UL

#define HALFRADIX 2147483648UL

#define MAX(a,b) ((a) > (b) ? (a) : (b))

/\*\*

\* Basic limb type. Note that some calculations rely on unsigned overflow wrap-around of this type.

\* As a result, only unsigned types should be used here, and the RADIX, HALFRADIX above should be

\* changed as necessary. Unsigned integer should probably be the most efficient word type, and this

\* is used by GMP for example.

\*/

typedef unsigned int word;

/\*\*

\* Structure for representing multiple precision integers. This is a base "word" LSB

\* representation. In this case the base, word, is 2^32. Length is the number of words

\* in the current representation. Length should not allow for trailing zeros (Things like

\* 000124). The capacity is the number of words allocated for the limb data.

\*/

typedef struct \_bignum {

int length;

int capacity;

word\* data;

} bignum;

/\*\*

\* Some forward delcarations as this was requested to be a single file.

\* See specific functions for explanations.

\*/

void bignum\_iadd(bignum\* source, bignum\* add);

void bignum\_add(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_isubtract(bignum\* source, bignum\* add);

void bignum\_subtract(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_imultiply(bignum\* source, bignum\* add);

void bignum\_multiply(bignum\* result, bignum\* b1, bignum\* b2);

void bignum\_idivide(bignum\* source, bignum\* div);

void bignum\_idivider(bignum\* source, bignum\* div, bignum\* remainder);

void bignum\_remainder(bignum\* source, bignum \*div, bignum\* remainder);

void bignum\_imodulate(bignum\* source, bignum\* modulus);

void bignum\_divide(bignum\* quotient, bignum\* remainder, bignum\* b1, bignum\* b2);

/\*\*

\* Save some frequently used bigintegers (0 - 10) so they do not need to be repeatedly

\* created. Used as, NUMS[5] = bignum("5"), etc..

\*/

word DATA0[1] = { 0 }; word DATA1[1] = { 1 }; word DATA2[1] = { 2 };

word DATA3[1] = { 3 }; word DATA4[1] = { 4 }; word DATA5[1] = { 5 };

word DATA6[1] = { 6 }; word DATA7[1] = { 7 }; word DATA8[1] = { 8 };

word DATA9[1] = { 9 }; word DATA10[1] = { 10 };

bignum NUMS[11] = { { 1, 1, DATA0 },{ 1, 1, DATA1 },{ 1, 1, DATA2 },

{ 1, 1, DATA3 },{ 1, 1, DATA4 },{ 1, 1, DATA5 },

{ 1, 1, DATA6 },{ 1, 1, DATA7 },{ 1, 1, DATA8 },

{ 1, 1, DATA9 },{ 1, 1, DATA10 } };

/\*\*

\* Initialize a bignum structure. This is the only way to safely create a bignum

\* and should be called where-ever one is declared. (We realloc the memory in all

\* other cases which is technically safe but may cause problems when we go to free

\* it.)

\*/

bignum\* bignum\_init() {

bignum\* b = (bignum \*)malloc(sizeof(bignum));

b->length = 0;

b->capacity = BIGNUM\_CAPACITY;

b->data = (unsigned int \*)calloc(BIGNUM\_CAPACITY, sizeof(word));

return b;

}

/\*\*

\* Free resources used by a bignum. Use judiciously to avoid memory leaks.

\*/

void bignum\_deinit(bignum\* b) {

free(b->data);

free(b);

}

/\*\*

\* Check if the given bignum is zero

\*/

int bignum\_iszero(bignum\* b) {

return b->length == 0 || (b->length == 1 && b->data[0] == 0);

}

/\*\*

\* Check if the given bignum is nonzero.

\*/

int bignum\_isnonzero(bignum\* b) {

return !bignum\_iszero(b);

}

/\*\*

\* Copy from source bignum into destination bignum.

\*/

void bignum\_copy(bignum\* source, bignum\* dest) {

dest->length = source->length;

if (source->capacity > dest->capacity) {

dest->capacity = source->capacity;

dest->data = (unsigned int \*)realloc(dest->data, dest->capacity \* sizeof(word));

}

memcpy(dest->data, source->data, dest->length \* sizeof(word));

}

/\*\*

\* Load a bignum from a base 10 string. Only pure numeric strings will work.

\*/

void bignum\_fromstring(bignum\* b, char\* string) {

int i, len = 0;

while (string[len] != '\0') len++; /\* Find string length \*/

for (i = 0; i < len; i++) {

if (i != 0) bignum\_imultiply(b, &NUMS[10]); /\* Base 10 multiply \*/

bignum\_iadd(b, &NUMS[string[i] - '0']); /\* Add \*/

}

}

/\*\*

\* Load a bignum from an unsigned integer.

\*/

void bignum\_fromint(bignum\* b, unsigned int num) {

b->length = 1;

if (b->capacity < b->length) {

b->capacity = b->length;

b->data = (unsigned int\*)realloc(b->data, b->capacity \* sizeof(word));

}

b->data[0] = num;

}

/\*\*

\* Print a bignum to stdout as base 10 integer. This is done by

\* repeated division by 10. We can make it more efficient by dividing by

\* 10^9 for example, then doing single precision arithmetic to retrieve the

\* 9 remainders

\*/

void bignum\_print(bignum\* b) {

int cap = 100, len = 0, i;

char\* buffer = (char\*)malloc(cap \* sizeof(char));

bignum \*copy = bignum\_init(), \*remainder = bignum\_init();

if (b->length == 0 || bignum\_iszero(b)) printf("0");

else {

bignum\_copy(b, copy);

while (bignum\_isnonzero(copy)) {

bignum\_idivider(copy, &NUMS[10], remainder);

buffer[len++] = remainder->data[0];

if (len >= cap) {

cap \*= 2;

buffer = (char\*)realloc(buffer, cap \* sizeof(char));

}

}

for (i = len - 1; i >= 0; i--) printf("%d", buffer[i]);

}

bignum\_deinit(copy);

bignum\_deinit(remainder);

free(buffer);

}

/\*\*

\* Check if two bignums are equal.

\*/

int bignum\_equal(bignum\* b1, bignum\* b2) {

int i;

if (bignum\_iszero(b1) && bignum\_iszero(b2)) return 1;

else if (bignum\_iszero(b1)) return 0;

else if (bignum\_iszero(b2)) return 0;

else if (b1->length != b2->length) return 0;

for (i = b1->length - 1; i >= 0; i--) {

if (b1->data[i] != b2->data[i]) return 0;

}

return 1;

}

/\*\*

\* Check if bignum b1 is greater than b2

\*/

int bignum\_greater(bignum\* b1, bignum\* b2) {

int i;

if (bignum\_iszero(b1) && bignum\_iszero(b2)) return 0;

else if (bignum\_iszero(b1)) return 0;

else if (bignum\_iszero(b2)) return 1;

else if (b1->length != b2->length) return b1->length > b2->length;

for (i = b1->length - 1; i >= 0; i--) {

if (b1->data[i] != b2->data[i]) return b1->data[i] > b2->data[i];

}

return 0;

}

/\*\*

\* Check if bignum b1 is less than b2

\*/

int bignum\_less(bignum\* b1, bignum\* b2) {

int i;

if (bignum\_iszero(b1) && bignum\_iszero(b2)) return 0;

else if (bignum\_iszero(b1)) return 1;

else if (bignum\_iszero(b2)) return 0;

else if (b1->length != b2->length) return b1->length < b2->length;

for (i = b1->length - 1; i >= 0; i--) {

if (b1->data[i] != b2->data[i]) return b1->data[i] < b2->data[i];

}

return 0;

}

/\*\*

\* Check if bignum b1 is greater than or equal to b2

\*/

int bignum\_geq(bignum\* b1, bignum\* b2) {

return !bignum\_less(b1, b2);

}

/\*\*

\* Check if bignum b1 is less than or equal to b2

\*/

int bignum\_leq(bignum\* b1, bignum\* b2) {

return !bignum\_greater(b1, b2);

}

/\*\*

\* Perform an in place add into the source bignum. That is source += add

\*/

void bignum\_iadd(bignum\* source, bignum\* add) {

bignum\* temp = bignum\_init();

bignum\_add(temp, source, add);

bignum\_copy(temp, source);

bignum\_deinit(temp);

}

/\*\*

\* Add two bignums by the add with carry method. result = b1 + b2

\*/

void bignum\_add(bignum\* result, bignum\* b1, bignum\* b2) {

word sum, carry = 0;

int i, n = MAX(b1->length, b2->length);

if (n + 1 > result->capacity) {

result->capacity = n + 1;

result->data = (unsigned int\*)realloc(result->data, result->capacity \* sizeof(word));

}

for (i = 0; i < n; i++) {

sum = carry;

if (i < b1->length) sum += b1->data[i];

if (i < b2->length) sum += b2->data[i];

result->data[i] = sum; /\* Already taken mod 2^32 by unsigned wrap around \*/

if (i < b1->length) {

if (sum < b1->data[i]) carry = 1; /\* Result must have wrapped 2^32 so carry bit is 1 \*/

else carry = 0;

}

else {

if (sum < b2->data[i]) carry = 1; /\* Result must have wrapped 2^32 so carry bit is 1 \*/

else carry = 0;

}

}

if (carry == 1) {

result->length = n + 1;

result->data[n] = 1;

}

else {

result->length = n;

}

}

/\*\*

\* Perform an in place subtract from the source bignum. That is, source -= sub

\*/

void bignum\_isubtract(bignum\* source, bignum\* sub) {

bignum\* temp = bignum\_init();

bignum\_subtract(temp, source, sub);

bignum\_copy(temp, source);

bignum\_deinit(temp);

}

/\*\*

\* Subtract bignum b2 from b1. result = b1 - b2. The result is undefined if b2 > b1.

\* This uses the basic subtract with carry method

\*/

void bignum\_subtract(bignum\* result, bignum\* b1, bignum\* b2) {

int length = 0, i;

word carry = 0, diff, temp;

if (b1->length > result->capacity) {

result->capacity = b1->length;

result->data = (word\*)realloc(result->data, result->capacity \* sizeof(word));

}

for (i = 0; i < b1->length; i++) {

temp = carry;

if (i < b2->length) temp = temp + b2->data[i]; /\* Auto wrapped mod RADIX \*/

diff = b1->data[i] - temp;

if (temp > b1->data[i]) carry = 1;

else carry = 0;

result->data[i] = diff;

if (result->data[i] != 0) length = i + 1;

}

result->length = length;

}

/\*\*

\* Perform an in place multiplication into the source bignum. That is source \*= mult

\*/

void bignum\_imultiply(bignum\* source, bignum\* mult) {

bignum\* temp = bignum\_init();

bignum\_multiply(temp, source, mult);

bignum\_copy(temp, source);

bignum\_deinit(temp);

}

/\*\*

\* Multiply two bignums by the naive school method. result = b1 \* b2. I have experimented

\* with FFT mult and Karatsuba but neither was looking to be more efficient than the school

\* method for reasonable number of digits. There are some improvments to be made here,

\* especially for squaring which can cut out half of the operations.

\*/

void bignum\_multiply(bignum\* result, bignum\* b1, bignum\* b2) {

int i, j, k;

word carry, temp;

unsigned long long int prod; /\* Long for intermediate product... this is not portable and should probably be changed \*/

if (b1->length + b2->length > result->capacity) {

result->capacity = b1->length + b2->length;

result->data = (unsigned int\*)realloc(result->data, result->capacity \* sizeof(word));

}

for (i = 0; i < b1->length + b2->length; i++) result->data[i] = 0;

for (i = 0; i < b1->length; i++) {

for (j = 0; j < b2->length; j++) {

prod = (b1->data[i] \* (unsigned long long int)b2->data[j]) + (unsigned long long int)(result->data[i + j]); /\* This should not overflow... \*/

carry = (word)(prod / RADIX);

/\* Add carry to the next word over, but this may cause further overflow.. propogate \*/

k = 1;

while (carry > 0) {

temp = result->data[i + j + k] + carry;

if (temp < result->data[i + j + k]) carry = 1;

else carry = 0;

result->data[i + j + k] = temp; /\* Already wrapped in unsigned arithmetic \*/

k++;

}

prod = (result->data[i + j] + b1->data[i] \* (unsigned long long int)b2->data[j]) % RADIX; /\* Again, should not overflow... \*/

result->data[i + j] = prod; /\* Add \*/

}

}

if (b1->length + b2->length > 0 && result->data[b1->length + b2->length - 1] == 0) result->length = b1->length + b2->length - 1;

else result->length = b1->length + b2->length;

}

/\*\*

\* Perform an in place divide of source. source = source/div.

\*/

void bignum\_idivide(bignum \*source, bignum \*div) {

bignum \*q = bignum\_init(), \*r = bignum\_init();

bignum\_divide(q, r, source, div);

bignum\_copy(q, source);

bignum\_deinit(q);

bignum\_deinit(r);

}

/\*\*

\* Perform an in place divide of source, also producing a remainder.

\* source = source/div and remainder = source - source/div.

\*/

void bignum\_idivider(bignum\* source, bignum\* div, bignum\* remainder) {

bignum \*q = bignum\_init(), \*r = bignum\_init();

bignum\_divide(q, r, source, div);

bignum\_copy(q, source);

bignum\_copy(r, remainder);

bignum\_deinit(q);

bignum\_deinit(r);

}

/\*\*

\* Calculate the remainder when source is divided by div.

\*/

void bignum\_remainder(bignum\* source, bignum \*div, bignum\* remainder) {

bignum \*q = bignum\_init();

bignum\_divide(q, remainder, source, div);

bignum\_deinit(q);

}

/\*\*

\* Modulate the source by the modulus. source = source % modulus

\*/

void bignum\_imodulate(bignum\* source, bignum\* modulus) {

bignum \*q = bignum\_init(), \*r = bignum\_init();

bignum\_divide(q, r, source, modulus);

bignum\_copy(r, source);

bignum\_deinit(q);

bignum\_deinit(r);

}

/\*\*

\* Divide two bignums by naive long division, producing both a quotient and remainder.

\* quotient = floor(b1/b2), remainder = b1 - quotient \* b2. If b1 < b2 the quotient is

\* trivially 0 and remainder is b2.

\*/

void bignum\_divide(bignum\* quotient, bignum\* remainder, bignum\* b1, bignum\* b2) {

bignum \*b2copy = bignum\_init(), \*b1copy = bignum\_init();

bignum \*temp = bignum\_init(), \*temp2 = bignum\_init(), \*temp3 = bignum\_init();

bignum\* quottemp = bignum\_init();

word carry = 0;

int n, m, i, j, length = 0;

unsigned long long factor = 1;

unsigned long long gquot, gtemp, grem;

if (bignum\_less(b1, b2)) { /\* Trivial case, b1/b2 = 0 iff b1 < b2. \*/

quotient->length = 0;

bignum\_copy(b1, remainder);

}

else if (bignum\_iszero(b1)) { /\* 0/x = 0.. assuming b2 is nonzero \*/

quotient->length = 0;

bignum\_fromint(remainder, 0);

}

else if (b2->length == 1) { /\* Division by a single limb means we can do simple division \*/

if (quotient->capacity < b1->length) {

quotient->capacity = b1->length;

quotient->data = (unsigned int\*)realloc(quotient->data, quotient->capacity \* sizeof(word));

}

for (i = b1->length - 1; i >= 0; i--) {

gtemp = carry \* RADIX + b1->data[i];

gquot = gtemp / b2->data[0];

quotient->data[i] = gquot;

if (quotient->data[i] != 0 && length == 0) length = i + 1;

carry = gtemp % b2->data[0];

}

bignum\_fromint(remainder, carry);

quotient->length = length;

}

else { /\* Long division is neccessary \*/

n = b1->length + 1;

m = b2->length;

if (quotient->capacity < n - m) {

quotient->capacity = n - m;

quotient->data = (unsigned int\*)realloc(quotient->data, (n - m) \* sizeof(word));

}

bignum\_copy(b1, b1copy);

bignum\_copy(b2, b2copy);

/\* Normalize.. multiply by the divisor by 2 until MSB >= HALFRADIX. This ensures fast

\* convergence when guessing the quotient below. We also multiply the dividend by the

\* same amount to ensure the result does not change. \*/

while (b2copy->data[b2copy->length - 1] < HALFRADIX) {

factor \*= 2;

bignum\_imultiply(b2copy, &NUMS[2]);

}

if (factor > 1) {

bignum\_fromint(temp, factor);

bignum\_imultiply(b1copy, temp);

}

/\* Ensure the dividend is longer than the original (pre-normalized) divisor. If it is not

\* we introduce a dummy zero word to artificially inflate it. \*/

if (b1copy->length != n) {

b1copy->length++;

if (b1copy->length > b1copy->capacity) {

b1copy->capacity = b1copy->length;

b1copy->data = (unsigned int\*)realloc(b1copy->data, b1copy->capacity \* sizeof(word));

}

b1copy->data[n - 1] = 0;

}

/\* Process quotient by long division \*/

for (i = n - m - 1; i >= 0; i--) {

gtemp = RADIX \* b1copy->data[i + m] + b1copy->data[i + m - 1];

gquot = gtemp / b2copy->data[m - 1];

if (gquot >= RADIX) gquot = UINT\_MAX;

grem = gtemp % b2copy->data[m - 1];

while (grem < RADIX && gquot \* b2copy->data[m - 2] > RADIX \* grem + b1copy->data[i + m - 2]) { /\* Should not overflow... ? \*/

gquot--;

grem += b2copy->data[m - 1];

}

quottemp->data[0] = gquot % RADIX;

quottemp->data[1] = (gquot / RADIX);

if (quottemp->data[1] != 0) quottemp->length = 2;

else quottemp->length = 1;

bignum\_multiply(temp2, b2copy, quottemp);

if (m + 1 > temp3->capacity) {

temp3->capacity = m + 1;

temp3->data = (unsigned int\*)realloc(temp3->data, temp3->capacity \* sizeof(word));

}

temp3->length = 0;

for (j = 0; j <= m; j++) {

temp3->data[j] = b1copy->data[i + j];

if (temp3->data[j] != 0) temp3->length = j + 1;

}

if (bignum\_less(temp3, temp2)) {

bignum\_iadd(temp3, b2copy);

gquot--;

}

bignum\_isubtract(temp3, temp2);

for (j = 0; j < temp3->length; j++) b1copy->data[i + j] = temp3->data[j];

for (j = temp3->length; j <= m; j++) b1copy->data[i + j] = 0;

quotient->data[i] = gquot;

if (quotient->data[i] != 0) quotient->length = i;

}

if (quotient->data[b1->length - b2->length] == 0) quotient->length = b1->length - b2->length;

else quotient->length = b1->length - b2->length + 1;

/\* Divide by factor now to find final remainder \*/

carry = 0;

for (i = b1copy->length - 1; i >= 0; i--) {

gtemp = carry \* RADIX + b1copy->data[i];

b1copy->data[i] = gtemp / factor;

if (b1copy->data[i] != 0 && length == 0) length = i + 1;

carry = gtemp % factor;

}

b1copy->length = length;

bignum\_copy(b1copy, remainder);

}

bignum\_deinit(temp);

bignum\_deinit(temp2);

bignum\_deinit(temp3);

bignum\_deinit(b1copy);

bignum\_deinit(b2copy);

bignum\_deinit(quottemp);

}

/\*\*

\* Perform modular exponentiation by repeated squaring. This will compute

\* result = base^exponent mod modulus

\*/

void bignum\_modpow(bignum\* base, bignum\* exponent, bignum\* modulus, bignum\* result) {

bignum \*a = bignum\_init(), \*b = bignum\_init(), \*c = bignum\_init();

bignum \*discard = bignum\_init(), \*remainder = bignum\_init();

bignum\_copy(base, a);

bignum\_copy(exponent, b);

bignum\_copy(modulus, c);

bignum\_fromint(result, 1);

while (bignum\_greater(b, &NUMS[0])) {

if (b->data[0] & 1) {

bignum\_imultiply(result, a);

bignum\_imodulate(result, c);

}

bignum\_idivide(b, &NUMS[2]);

bignum\_copy(a, discard);

bignum\_imultiply(a, discard);

bignum\_imodulate(a, c);

}

bignum\_deinit(a);

bignum\_deinit(b);

bignum\_deinit(c);

bignum\_deinit(discard);

bignum\_deinit(remainder);

}

/\*\*

\* Compute the gcd of two bignums. result = gcd(b1, b2)

\*/

void bignum\_gcd(bignum\* b1, bignum\* b2, bignum\* result) {

bignum \*a = bignum\_init(), \*b = bignum\_init(), \*remainder = bignum\_init();

bignum \*temp = bignum\_init(), \*discard = bignum\_init();

bignum\_copy(b1, a);

bignum\_copy(b2, b);

while (!bignum\_equal(b, &NUMS[0])) {

bignum\_copy(b, temp);

bignum\_imodulate(a, b);

bignum\_copy(a, b);

bignum\_copy(temp, a);

}

bignum\_copy(a, result);

bignum\_deinit(a);

bignum\_deinit(b);

bignum\_deinit(remainder);

bignum\_deinit(temp);

bignum\_deinit(discard);

}

/\*\*

\* Compute the inverse of a mod m. Or, result = a^-1 mod m.

\*/

void bignum\_inverse(bignum\* a, bignum\* m, bignum\* result) {

bignum \*remprev = bignum\_init(), \*rem = bignum\_init();

bignum \*auxprev = bignum\_init(), \*aux = bignum\_init();

bignum \*rcur = bignum\_init(), \*qcur = bignum\_init(), \*acur = bignum\_init();

bignum\_copy(m, remprev);

bignum\_copy(a, rem);

bignum\_fromint(auxprev, 0);

bignum\_fromint(aux, 1);

while (bignum\_greater(rem, &NUMS[1])) {

bignum\_divide(qcur, rcur, remprev, rem);

/\* Observe we are finding the inverse in a finite field so we can use

\* a modified algorithm that avoids negative numbers here \*/

bignum\_subtract(acur, m, qcur);

bignum\_imultiply(acur, aux);

bignum\_iadd(acur, auxprev);

bignum\_imodulate(acur, m);

bignum\_copy(rem, remprev);

bignum\_copy(aux, auxprev);

bignum\_copy(rcur, rem);

bignum\_copy(acur, aux);

}

bignum\_copy(acur, result);

bignum\_deinit(remprev);

bignum\_deinit(rem);

bignum\_deinit(auxprev);

bignum\_deinit(aux);

bignum\_deinit(rcur);

bignum\_deinit(qcur);

bignum\_deinit(acur);

}

/\*\*

\* Compute the jacobi symbol, J(ac, nc).

\*/

int bignum\_jacobi(bignum\* ac, bignum\* nc) {

bignum \*remainder = bignum\_init(), \*twos = bignum\_init();

bignum \*temp = bignum\_init(), \*a = bignum\_init(), \*n = bignum\_init();

int mult = 1, result = 0;

bignum\_copy(ac, a);

bignum\_copy(nc, n);

while (bignum\_greater(a, &NUMS[1]) && !bignum\_equal(a, n)) {

bignum\_imodulate(a, n);

if (bignum\_leq(a, &NUMS[1]) || bignum\_equal(a, n)) break;

bignum\_fromint(twos, 0);

/\* Factor out multiples of two \*/

while (a->data[0] % 2 == 0) {

bignum\_iadd(twos, &NUMS[1]);

bignum\_idivide(a, &NUMS[2]);

}

/\* Coefficient for flipping \*/

if (bignum\_greater(twos, &NUMS[0]) && twos->data[0] % 2 == 1) {

bignum\_remainder(n, &NUMS[8], remainder);

if (!bignum\_equal(remainder, &NUMS[1]) && !bignum\_equal(remainder, &NUMS[7])) {

mult \*= -1;

}

}

if (bignum\_leq(a, &NUMS[1]) || bignum\_equal(a, n)) break;

bignum\_remainder(n, &NUMS[4], remainder);

bignum\_remainder(a, &NUMS[4], temp);

if (!bignum\_equal(remainder, &NUMS[1]) && !bignum\_equal(temp, &NUMS[1])) mult \*= -1;

bignum\_copy(a, temp);

bignum\_copy(n, a);

bignum\_copy(temp, n);

}

if (bignum\_equal(a, &NUMS[1])) result = mult;

else result = 0;

bignum\_deinit(remainder);

bignum\_deinit(twos);

bignum\_deinit(temp);

bignum\_deinit(a);

bignum\_deinit(n);

return result;

}

/\*\*

\* Check whether a is a Euler witness for n. That is, if a^(n - 1)/2 != Ja(a, n) mod n

\*/

int solovayPrime(int a, bignum\* n) {

bignum \*ab = bignum\_init(), \*res = bignum\_init(), \*pow = bignum\_init();

bignum \*modpow = bignum\_init();

int x, result;

bignum\_fromint(ab, a);

x = bignum\_jacobi(ab, n);

if (x == -1) bignum\_subtract(res, n, &NUMS[1]);

else bignum\_fromint(res, x);

bignum\_copy(n, pow);

bignum\_isubtract(pow, &NUMS[1]);

bignum\_idivide(pow, &NUMS[2]);

bignum\_modpow(ab, pow, n, modpow);

result = !bignum\_equal(res, &NUMS[0]) && bignum\_equal(modpow, res);

bignum\_deinit(ab);

bignum\_deinit(res);

bignum\_deinit(pow);

bignum\_deinit(modpow);

return result;

}

/\*\*

\* Test if n is probably prime, by repeatedly using the Solovay-Strassen primality test.

\*/

int probablePrime(bignum\* n, int k) {

if (bignum\_equal(n, &NUMS[2])) return 1;

else if (n->data[0] % 2 == 0 || bignum\_equal(n, &NUMS[1])) return 0;

while (k-- > 0) {

if (n->length <= 1) { /\* Prevent a > n \*/

if (!solovayPrime(rand() % (n->data[0] - 2) + 2, n)) return 0;

}

else {

int wit = rand() % (RAND\_MAX - 2) + 2;

if (!solovayPrime(wit, n)) return 0;

}

}

return 1;

}

/\*\*

\* Generate a random prime number, with a specified number of digits.

\* This will generate a base 10 digit string of given length, convert it

\* to a bignum and then do an increasing search for the first probable prime.

\*/

void randPrime(int numDigits, bignum\* result) {

char \*string = (char\*)malloc((numDigits + 1) \* sizeof(char));

int i;

string[0] = (rand() % 9) + '1'; /\* No leading zeros \*/

string[numDigits - 1] = (rand() % 5) \* 2 + '1'; /\* Last digit is odd \*/

for (i = 1; i < numDigits - 1; i++) string[i] = (rand() % 10) + '0';

string[numDigits] = '\0';

bignum\_fromstring(result, string);

while (1) {

if (probablePrime(result, ACCURACY)) {

free(string);

return;

}

bignum\_iadd(result, &NUMS[2]); /\* result += 2 \*/

}

}

/\*\*

\* Choose a random public key exponent for the RSA algorithm. The exponent will

\* be less than the modulus, n, and coprime to phi.

\*/

void randExponent(bignum\* phi, int n, bignum\* result) {

bignum\* gcd = bignum\_init();

int e = rand() % n;

while (1) {

bignum\_fromint(result, e);

bignum\_gcd(result, phi, gcd);

if (bignum\_equal(gcd, &NUMS[1])) {

bignum\_deinit(gcd);

return;

}

e = (e + 1) % n;

if (e <= 2) e = 3;

}

}

/\*\*

\* Read the file fd into an array of bytes ready for encryption.

\* The array will be padded with zeros until it divides the number of

\* bytes encrypted per block. Returns the number of bytes read.

\*/

int readFile(FILE\* fd, char\*\* buffer, int bytes) {

int len = 0, cap = BUF\_SIZE, r;

char buf[BUF\_SIZE];

\*buffer = (char\*)malloc(BUF\_SIZE \* sizeof(char));

while ((r = fread(buf, sizeof(char), BUF\_SIZE, fd)) > 0) {

if (len + r >= cap) {

cap \*= 2;

\*buffer = (char\*)realloc(\*buffer, cap);

}

memcpy(&(\*buffer)[len], buf, r);

len += r;

}

/\* Pad the last block with zeros to signal end of cryptogram. An additional block is added if there is no room \*/

if (len + bytes - len % bytes > cap) \*buffer = (char\*)realloc(\*buffer, len + bytes - len % bytes);

do {

(\*buffer)[len] = '\0';

len++;

} while (len % bytes != 0);

return len;

}

/\*\*

\* Encode the message m using public exponent and modulus, result = m^e mod n

\*/

void encode(bignum\* m, bignum\* e, bignum\* n, bignum\* result) {

bignum\_modpow(m, e, n, result);

}

/\*\*

\* Decode cryptogram c using private exponent and public modulus, result = c^d mod n

\*/

void decode(bignum\* c, bignum\* d, bignum\* n, bignum\* result) {

bignum\_modpow(c, d, n, result);

}

/\*\*

\* Encode the message of given length, using the public key (exponent, modulus)

\* The resulting array will be of size len/bytes, each index being the encryption

\* of "bytes" consecutive characters, given by m = (m1 + m2\*128 + m3\*128^2 + ..),

\* encoded = m^exponent mod modulus

\*/

bignum \*encodeMessage(int len, int bytes, char \*message, bignum \*exponent, bignum \*modulus) {

/\* Calloc works here because capacity = 0 forces a realloc by callees but we should really

\* bignum\_init() all of these \*/

int i, j;

bignum \*encoded = (bignum \*)calloc(len / bytes, sizeof(bignum));

bignum \*num128 = bignum\_init(), \*num128pow = bignum\_init();

bignum \*x = bignum\_init(), \*current = bignum\_init();

bignum\_fromint(num128, 128);

bignum\_fromint(num128pow, 1);

for (i = 0; i < len; i += bytes) {

bignum\_fromint(x, 0);

bignum\_fromint(num128pow, 1);

/\* Compute buffer[0] + buffer[1]\*128 + buffer[2]\*128^2 etc (base 128 representation for characters->int encoding)\*/

for (j = 0; j < bytes; j++) {

bignum\_fromint(current, message[i + j]);

bignum\_imultiply(current, num128pow);

bignum\_iadd(x, current); /\*x += buffer[i + j] \* (1 << (7 \* j)) \*/

bignum\_imultiply(num128pow, num128);

}

encode(x, exponent, modulus, &encoded[i / bytes]);

bignum\_print(&encoded[i / bytes]);

printf(" ");

}

return encoded;

}

/\*\*

\* Decode the cryptogram of given length, using the private key (exponent, modulus)

\* Each encrypted packet should represent "bytes" characters as per encodeMessage.

\* The returned message will be of size len \* bytes.

\*/

int \*decodeMessage(int len, int bytes, bignum \*cryptogram, bignum \*exponent, bignum \*modulus) {

int \*decoded = (int \*)malloc(len \* bytes \* sizeof(int));

int i, j;

bignum \*x = bignum\_init(), \*remainder = bignum\_init();

bignum \*num128 = bignum\_init();

bignum\_fromint(num128, 128);

for (i = 0; i < len; i++) {

decode(&cryptogram[i], exponent, modulus, x);

for (j = 0; j < bytes; j++) {

bignum\_idivider(x, num128, remainder);

if (remainder->length == 0) decoded[i\*bytes + j] = (char)0;

else decoded[i\*bytes + j] = (char)(remainder->data[0]);

printf("%c", (char)(decoded[i\*bytes + j]));

}

}

return decoded;

}

/\*\*

\* Main method to demostrate the system. Sets up primes p, q, and proceeds to encode and

\* decode the message given in "text.txt"

\*/

int main(void) {

int i, bytes, len;

bignum \*p = bignum\_init(), \*q = bignum\_init(), \*n = bignum\_init();

bignum \*phi = bignum\_init(), \*e = bignum\_init(), \*d = bignum\_init();

bignum \*bbytes = bignum\_init(), \*shift = bignum\_init();

bignum \*temp1 = bignum\_init(), \*temp2 = bignum\_init();

char filedata[BUFSIZ];

bignum \*encoded;

int \*decoded;

char \*buffer;

FILE\* f = NULL;

srand(time(NULL));

randPrime(FACTOR\_DIGITS, p);

printf("p = ");

bignum\_print(p);

printf("\n");

//getchar();

randPrime(FACTOR\_DIGITS, q);

printf("q = ");

bignum\_print(q);

printf("\n");

//getchar();

bignum\_multiply(n, p, q);

printf("n = p\*q = ");

bignum\_print(n);

printf("\n");

//getchar();

bignum\_subtract(temp1, p, &NUMS[1]);

bignum\_subtract(temp2, q, &NUMS[1]);

bignum\_multiply(phi, temp1, temp2); /\* phi = (p - 1) \* (q - 1) \*/

printf("f =(p-1)\*(q-1)= ");

bignum\_print(phi);

printf("\n");

//getchar();

randExponent(phi, EXPONENT\_MAX, e);

printf("Chose public exponent, e = ");

bignum\_print(e);

printf("\nPublic key is (");

//bignum\_print(e);

//printf(", ");

bignum\_print(n);

printf(")\n");

printf(")\n");

//getchar();

bignum\_inverse(e, phi, d);

printf("Calculated private exponent, d = ");

bignum\_print(d);

printf("\nPrivate key is (");

//bignum\_print(d);

//printf(", ");

bignum\_print(n);

printf(")\n");

printf(")\n");

//getchar();

/\* Compute maximum number of bytes that can be encoded in one encryption \*/

bytes = -1;

bignum\_fromint(shift, 1 << 7); /\* 7 bits per char \*/

bignum\_fromint(bbytes, 1);

while (bignum\_less(bbytes, n)) {

bignum\_imultiply(bbytes, shift); /\* Shift by one byte, NB: we use bitmask representative so this can actually be a shift... \*/

bytes++;

}

printf("Opening file \"text.txt\" for reading\n");

f = fopen("text.txt", "r");

if (f == NULL) {

printf("fail to read text.txt\n");

return EXIT\_FAILURE;

}

len = readFile(f, &buffer, bytes); /\* len will be a multiple of bytes, to send whole chunks \*/

printf("File \"text.txt\" read successfully");

//getchar();

printf("\n");

printf("\nEncoding information is:\n\n");

encoded = encodeMessage(len, bytes, buffer, e, n);

//getchar();

printf("\n\nDecoding encoded message is:\n");

//getchar();

printf("\n");

decoded = decodeMessage(len / bytes, bytes, encoded, d, n);

/\* Eek! This is why we shouldn't of calloc'd those! \*/

for (i = 0; i < len / bytes; i++) free(encoded[i].data);

free(encoded);

free(decoded);

free(buffer);

bignum\_deinit(p);

bignum\_deinit(q);

bignum\_deinit(n);

bignum\_deinit(phi);

bignum\_deinit(e);

bignum\_deinit(d);

bignum\_deinit(bbytes);

bignum\_deinit(shift);

bignum\_deinit(temp1);

bignum\_deinit(temp2);

fclose(f);

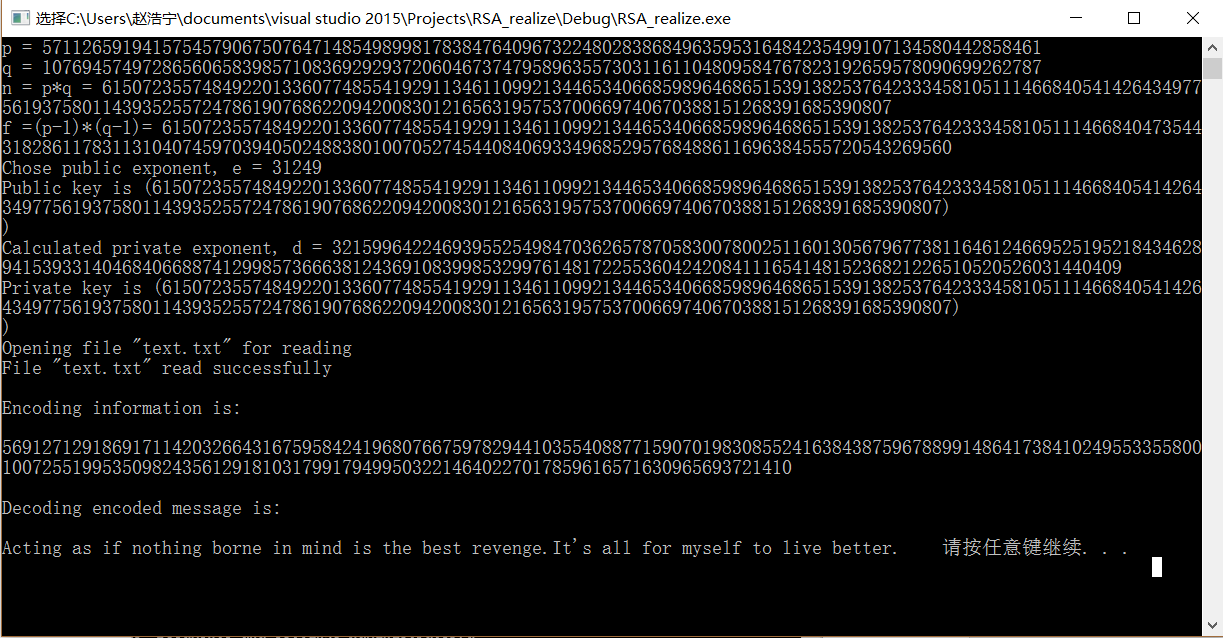
system("pause");

return EXIT\_SUCCESS;

}

# 六 实验结果及分析

实验结果如图所示



运行结果界面

# 八 程序使用

对text.txt文件进行修改，保存要加密的密文。打开RSA\_realize.exe，程序会自动运行，产生相关参数，并进行加密和解密

# 七 实验小结

本次实验对输入的任意一段明文字母，实现了输出对应密钥的密文字母，密钥，公钥，和欧拉函数，大素数等相关参数。亲手实际编写RSA密码算法代码，更好的了解和掌握了RSA的相关内容。通过用C++对RSA密码体制进行编程，对RSA密码体制的加解密过程有了更深入的理解。通过这个实验更是让我获得了很多实际工作中所要具备的能力。