# RSA加密算法

## 1120152035 王奥博

## rsa基本原理

RSA算法最核心的有5个参数:

```
n, e, d, q, p
```

其中:

- n = p * q, p和q为**大素数**
- e为(1, **φ(n)**)间的整数,且gcd(e, **φ(n)**) = 1

  > **φ(n)**为n的欧拉函数,即**φ(n)** = (p - 1) * (q - 1)

- d为e对于**φ(n)**的模反元素

  > ed ≡ 1 (mod φ(n)), 可以用扩展欧几里得算法解出d

- (n, e)为公钥,可以公开
- (n, d)为私钥,需要保密

设c为密文,m为明文,则RSA的加密解密过程可以表示为:

> $c = m^e \bmod n$
>
> $m = c^d \bmod n$

## 对程序中的函数进行解释

> 因为RSA的计算中经常设计到大整数运算,用C语言处理大整数比较麻烦,因此这里以小素数为例演示RSA的加解密过程.
>
> 同时完成了python版的RSA算法,支持大整数运算

该程序使用了下列函数:

---

函数定义:***void Init()***

函数功能:完成加密之间的准备工作,生成100个不大于1000的整数用于RSA的加密

```
//算法初始化，生成100个不大于1000的整数
void Init()
{
    srand((unsigned)time(NULL));
    cout<<"Generate 100 random numbers:" <<'\n';
    for(int i = 0; i < 100; i++)
    {
        Plaintext[i] = rand()%1000;
        cout<<Plaintext[i]<<" ";
    }

    cout << "\n\n";
}
```

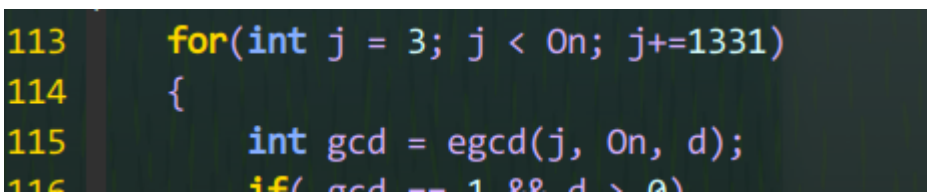
---

函数定义:***void RSA_Init()***

函数功能:初始化p,q,e等参数(因为q,p,e需要均为素数,用户较难控制输入,因此改用程序随机生成)

```
void RSA_Init()
{
    //取出1000内素数保存在prime[]数组中
    int prime[200];
    int cntPrime = genPrime(prime);
    // cout << "\n\n\n";
    // for(int i = 0; i < 5000; i++)
        // cout << prime[i] << " ";
    // cout << "\n\n\n";

    //随机取两个素数p,q
    srand((unsigned)time(NULL));
    int ranNum1 = rand() % cntPrime;
    int ranNum2 = rand() % cntPrime;
    int p = prime[ranNum1], q = prime[ranNum2];

    n = p * q;

    int On = (p - 1) * (q - 1);


    //用欧几里德扩展算法求e,d
    //保证e足够大
    for(int j = 3; j < On; j+=1331)
    {
        int gcd = egcd(j, On, d);
        if( gcd == 1 && d > 0)
        {
            e = j;
            break;
        }

    }

}
```

值得一提的是,如果较小,很容易用爆破的方法爆破出明文,需要保证e足够大,在本份代码中,使用了每次使e自加一个较大质数的方法保证该点:




---

函数定义:***int egcd(int m, int n, int &x)***

函数功能:数学中的拓展欧几里得算法,RSA中用于计算d

```
//欧几里得扩展算法
int egcd(int m,int n,int &x)
{
    int x1,y1,x0,y0, y;
    y0 = x1 = x = 0;
    x0 = y1 = y = 1;
    int r=m%n;
    int q=(m-r)/n;
    while(r)
    {
        x=x0-q*x1; y=y0-q*y1;
        x0=x1; y0=y1;
        x1=x; y1=y;
        m=n; n=r; r=m%n;
        q=(m-r)/n;
    }
    return n;
}
```

函数定义:***ll powMod(ll a, int b, int n)***

参数功能:该函数返回值为$a^b$ % n

```
ll powMod(ll a, int b, int n)
{
    int c = 0, bin[1000];
    ll d = 1;
    int k = binTransform(b, bin)-1;

    for(int i = k; i >= 0; i--)
    {
        c = 2*c;
        d = (d*d)%n;
        if(bin[i] == 1)
        {
            c = c + 1;
            d = (d*a)%n;
        }
    }
    return d;
}
```

该函数中为了实现快速幂,调用了另一个二进制转换的函数:

```
//二进制转换
int binTransform(int num, int bin[])
{

    int i = 0,  mod = 0;

    //转换为二进制，逆向暂存 tmp[]数组中
    while(num != 0)
    {
        mod = num%2;
        bin[i] = mod;
        num = num/2;
        i++;
    }

    //返回二进制数的位数
    return i;
}
```

快速幂的具体算法细节不再展开叙述

---

函数定义:***void Encrypt()***

函数功能:实现RSA的加密过成,对Init()生成的100个素数进行加密,同时打印出公钥(n, e)与私钥(n, d)

```
//RSA加密
void Encrypt()
{
    cout<<"Public Key (e, n) : e = " <<e<<" n = "<<n<<'\n';
    cout<<"Private Key (d, n) : d = " <<d<<" n = "<<n<<'\n'<<'\n';

    int i = 0;
    for(i = 0; i < 100; i++)
        Ciphertext[i] = powMod(Plaintext[i], e, n);

    cout<<"Use the public key (e, n) to encrypt:" <<'\n';
    for(i = 0; i < 100; i++)
        cout<<Ciphertext[i]<<" ";
    cout<<'\n'<<'\n';
}
```

---

函数定义:***void Decrypt()***

函数功能:实现RSA的解密过成,使用之前过程中的(n, d)进行解密

```
//RSA解密
void Decrypt()
{
    int i = 0;
    for(i = 0; i < 100; i++)
        Ciphertext[i] = powMod(Ciphertext[i], d, n);

    cout<<"Use private key (d, n) to decrypt:" <<'\n';
    for(i = 0; i < 100; i++)
        cout<<Ciphertext[i]<<" ";
    cout<<"\n\n";
}
```

## 完整代码：

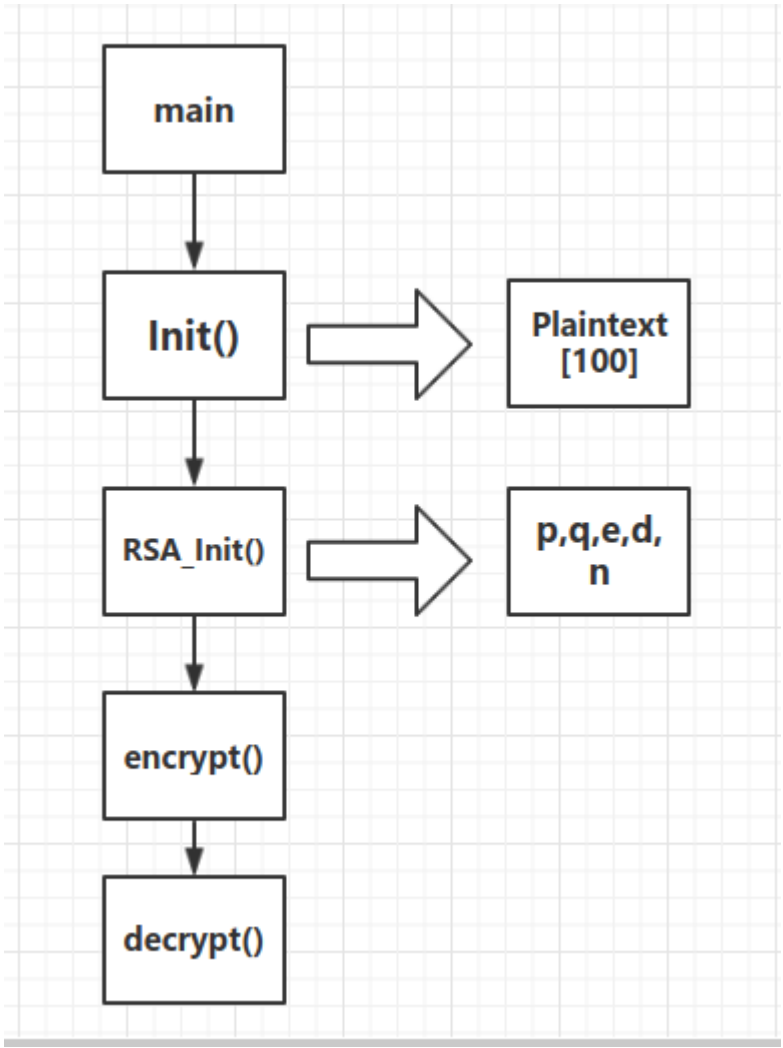
```cpp
#include <iostream>
#include <cmath>
#include <cstring>
#include <ctime>
#include <cstdlib>
#define ll long long
using namespace std;

int Plaintext[100];//明文
ll Ciphertext[100];//密文
int n, e = 0, d;

//二进制转换
int binTransform(int num, int bin[])
{

    int i = 0,  mod = 0;

    //转换为二进制，逆向暂存 tmp[]数组中
    while(num != 0)
    {
        mod = num%2;
        bin[i] = mod;
        num = num/2;
        i++;
    }

    //返回二进制数的位数
    return i;
}

//反复平方求幂
ll powMod(ll a, int b, int n)
{
    int c = 0, bin[1000];
    ll d = 1;
    int k = binTransform(b, bin)-1;

    for(int i = k; i >= 0; i--)
    {
        c = 2*c;
        d = (d*d)%n;
        if(bin[i] == 1)
        {
            c = c + 1;
            d = (d*a)%n;
        }
    }
    return d;
}

//生成1000以内素数
int genPrime(int prime[])
```

```
{
    int c = 0;
    bool vis[1001];
    memset(vis, 0, sizeof(vis));
    for(int i = 2; i <= 1000; i++)
        if(!vis[i])
        {
            prime[c++] = i;
            for(int j = i*i; j <= 1000; j+=i)
                vis[j] = true;
        }

    return c;
}


//欧几里得扩展算法
int egcd(int m,int n,int &x)
{
    int x1,y1,x0,y0, y;
    y0 = x1 = x = 0;
    x0 = y1 = y = 1;
    int r=m%n;
    int q=(m-r)/n;
    while(r)
    {
        x=x0-q*x1; y=y0-q*y1;
        x0=x1; y0=y1;
        x1=x; y1=y;
        m=n; n=r; r=m%n;
        q=(m-r)/n;
    }
    return n;
}

//RSA初始化
void RSA_Init()
{
    //取出1000内素数保存在prime[]数组中
    int prime[200];
    int cntPrime = genPrime(prime);
    // cout << "\n\n\n";
    // for(int i = 0; i < 5000; i++)
        // cout << prime[i] << " ";
    // cout << "\n\n\n";

    //随机取两个素数p,q
    srand((unsigned)time(NULL));
    int ranNum1 = rand() % cntPrime;
    int ranNum2 = rand() % cntPrime;
    int p = prime[ranNum1], q = prime[ranNum2];


    n = p * q;
```

```
    int On = (p - 1) * (q - 1);


    //用欧几里德扩展算法求 e,d
     //保证e足够大
    for(int j = 3; j < On; j+=1331)
    {
        int gcd = egcd(j, On, d);
        if( gcd == 1 && d > 0)
        {
            e = j;
            break;
        }

    }

}

//RSA加密
void Encrypt()
{
    cout<<"Public Key (e, n) : e = " <<e<<" n = "<<n<<'\n';
    cout<<"Private Key (d, n) : d = " <<d<<" n = "<<n<<'\n'<<'\n';

    int i = 0;
    for(i = 0; i < 100; i++)
        Ciphertext[i] = powMod(Plaintext[i], e, n);

    cout<<"Use the public key (e, n) to encrypt:" <<'\n';
    for(i = 0; i < 100; i++)
        cout<<Ciphertext[i]<<" ";
    cout<<'\n'<<'\n';
}

//RSA解密
void Decrypt()
{
    int i = 0;
    for(i = 0; i < 100; i++)
        Ciphertext[i] = powMod(Ciphertext[i], d, n);

    cout<<"Use private key (d, n) to decrypt:" <<'\n';
    for(i = 0; i < 100; i++)
        cout<<Ciphertext[i]<<" ";
    cout<<"\n\n";
}


//算法初始化，生成100个不大于1000的整数
void Init()
{

    srand((unsigned)time(NULL));
```

```
    cout<<"Generate 100 random numbers:" <<'\n';
    for(int i = 0; i < 100; i++)
    {
        Plaintext[i] = rand()%1000;
        cout<<Plaintext[i]<<" ";
    }

    cout << "\n\n";
}

int main()
{
    Init();
    while(!e)
        RSA_Init();

    Encrypt();
    Decrypt();

    return 0;
}
```

## 程序流程：




## 测试:

完成了在linux和windows下两个平台上的测试:

Linux下:

```
RSA [master●●] g++ rsa.cpp -o rsa
RSA [master●●] ./rsa
Generate 100 random numbers:
935 314 860 558 346 551 88 156 803 333 475 890 79 687 510 110 42 620 306 161 793 444 26 528 128 404 162 617 644 890 279 579 205 491 489 903 394
 578 411 198 263 238 440 342 926 951 804 320 923 110 482 68 907 860 596 35 616 110 5 612 0 636 543 557 128 33 460 522 611 871 72 226 110 865 92
0 388 168 724 60 91 187 542 159 94 754 107 481 371 217 838 335 569 475 879 127 603 912 939 477 875

Public Key (e, n) : e = 2665 n = 422053
Private Key (d, n) : d = 160721 n = 422053

Use the public key (e, n) to encrypt:
276062 120028 159971 373343 216580 204665 2687 260660 29207 167761 261928 74915 124669 131096 13794 107164 6951 60250 206375 206171 83855 28566
0 268305 128406 411174 231003 190167 326861 153921 74915 384513 156164 252616 218020 306009 204590 165933 38473 154730 396997 24809 269717 3718
75 263243 242322 386883 244806 310725 123555 107164 103342 414254 346532 159971 405886 70824 353950 107164 253810 254178 0 98046 212645 56450 4
11174 269222 67784 408878 287540 364161 421418 380639 107164 402845 393756 121505 302230 86283 153462 356958 249242 332496 85230 173198 21213 2
01668 266657 180939 335791 58415 185327 288887 261928 167532 120643 98230 257392 148311 344826 403022

Use private key (d, n) to decrypt:
935 314 860 558 346 551 88 156 803 333 475 890 79 687 510 110 42 620 306 161 793 444 26 528 128 404 162 617 644 890 279 579 205 491 489 903 394
 578 411 198 263 238 440 342 926 951 804 320 923 110 482 68 907 860 596 35 616 110 5 612 0 636 543 557 128 33 460 522 611 871 72 226 110 865 92
0 388 168 724 60 91 187 542 159 94 754 107 481 371 217 838 335 569 475 879 127 603 912 939 477 875

RSA [master●●]
```

Windows下:

```
Generate 100 random numbers:
14 461 770 530 424 140 960 819 474 603 552 623 116 324 994 104 944 16 487 311 942 422 274 324 65 182 827 633 570 423 90 295 644 798 529 692 784 528 501 627 881 626 822 931 522 1
74 341 157 310 832 46 960 104 378 558 327 293 894 182 390 535 578 801 113 844 192 186 478 614 6 562 355 852 812 873 923 968 724 882 581 866 559 740 5 822 117 381 832 137 789 122
 506 682 190 765 602 430 780 991 800

Public Key (e, n) : e = 10651 n = 76327
Private Key (d, n) : d = 28051 n = 76327

Use the public key (e, n) to encrypt:
2351 25313 45244 19372 44362 10146 74165 59294 52536 68516 10867 66456 48605 57419 23046 23936 52684 54106 66885 12431 32795 59320 55900 57419 46342 71607 70091 63738 9284 69538
 75035 37476 3640 71923 11347 75169 51503 24531 61202 27019 38322 71544 31273 65728 35117 47966 41209 9773 16513 35829 74478 74165 23936 52659 38430 8407 17136 45984 71607 5799
38027 36083 64507 12508 35101 46469 26633 24390 25229 67919 59538 9114 4931 36271 42414 1820 52521 21286 46597 15646 45145 10376 23300 46873 31273 2888 28702 35829 38768 69904 2
9327 67167 49363 45966 29383 27046 31513 53668 71909 23037

Use private key (d, n) to decrypt:
14 461 770 530 424 140 960 819 474 603 552 623 116 324 994 104 944 16 487 311 942 422 274 324 65 182 827 633 570 423 90 295 644 798 529 692 784 528 501 627 881 626 822 931 522 1
74 341 157 310 832 46 960 104 378 558 327 293 894 182 390 535 578 801 113 844 192 186 478 614 6 562 355 852 812 873 923 968 724 882 581 866 559 740 5 822 117 381 832 137 789 122
 506 682 190 765 602 430 780 991 800


--------------------------------
Process exited after 0.1092 seconds with return value 0
请按任意键继续. . .
```

另附python版RSA:

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__Auther__ = 'M4x'

from libnum import prime_test as isPrime
from libnum import generate_prime as genPrime
from libnum import gcd, xgcd
from libnum import n2s, s2n
from gmpy2 import invert

def encrypt():
    m = raw_input("请输入明文: ")
    p = genPrime(1024)
    q = genPrime(1024)
    n = p * q
    fi = (p- 1) * (q - 1)
    i = (p - 1) * (q - 1) - 1
    while True:
        if gcd(i, (p - 1) * (q - 1)) == 1:
            e = i
            break
        i -= 1

    c = pow(s2n(m), e, n)
    print "密文: ",  c
    return p, q, n, e, c

def decrypt(p, q, e, c):
    d = invert(e, (p - 1) * (q - 1))
    m = pow(c, d, n)
    print "解密: ", n2s(m)


if __name__ == "__main__":
    p, q, n, e, c = encrypt()
    decrypt(p, q, e, c)
```

python版测试结果: