RSA 暗号の実装から見る暗号化技術

作成日：2015 年 7 月 2 日

1.はじめに

最近、セキュリティに関する問題が取り沙汰されており、その根本となる暗号化技術について深く知りたくなった。オペレーティングシステムそのものの機能としての技術ではないが、コンピュータを使用する上で重要なものと考えられるため選択した。

2. 調査結果

2.1 調査結果の概要

暗号化技術について、RSA 暗号を中心として、暗号化と読解の可能性について実験的に調査した。

2.2 暗号化技術の背景

暗号術(cryptographia)の由来は、ギリシャ語の crypta(洞窟)と graphia(文字)とされている。古代の人々が洞窟に描いた壁画が暗号として見えたことによるものだろう。古代暗号として最も有名なのはロゼッタストーンに書かれていたヒエログリフである。しかし、これは元々意思伝達の手段として書かれたものであり、本来暗号として書かれたものではない。

本来の意味での暗号として使われたものとして有名なのが、古代ローマ帝国皇帝のジュリアス・シーザーが使ったとされるシーザー暗号である。

時代は進み、現代の暗号として実用的に使われたものとして、ドイツのエニグマ暗号と、日本の紫暗号が存在する。これは、現在使われている暗号技術の基礎とも考えられるものである。

現在使われている暗号化技術は 1976 年の公開鍵暗号方式の発明によって進歩した。はじめの 10 年ほどで RSA 暗号やエルガマル暗号など基本的な暗号方式が発明される黎明期であった。その後、10 年ほどかけて実用的なレベルに到達した。現在では数学に基づいて厳密な安全性の証明が行われている。しかし、近年のコンピュータ性能の向上によって安全性が揺るいでいる暗号方式も存在する。

2.3 暗号化技術の目的

エニグマ暗号や紫暗号は戦争時に使われたことで有名である。目的は敵に知られたくない情報（例えば奇襲に関する作戦）を遠く離れた味方に伝えることである。そのまま電波に乗せて伝えたのでは敵に傍受されてしまい、作戦は失敗に終わるだろう。そこで、味方同士しか知らない法則に従って、伝えたい情報（主に文字情報）を変換して送信する。そうすることで、敵に傍受されたとしても法則を知らない限り（結果的にはエニグマ暗号や紫暗号は解読されてしまうのだが）情報が漏れることはない。

現在のコンピュータ社会においては、個人間での伝言から個人情報やクレジットカード情報などの機密情報までインターネットを通じてやりとりされている。これらの情報を第三者に盗み取られ、不正に使用されることを防ぐことを目的として、暗号化技術が使われている。

2.4 暗号化技術の仕組み

暗号化技術に関して、いくつかに分類することができる。

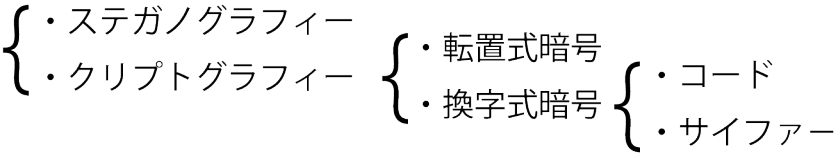

図 1 暗号の種類

大きくステガノグラフィとクリプトグラフィの 2 つに分けられる。ステガノグラフィは古典的な方法で、通信文自体を互いに決めた場所に隠しておくというものである。もう 1 つのクリプトグラフィは通信文の内容を変換することで、第三者に読むことができないようにする方法である。

クリプトグラフィは転置式暗号と換字式暗号の 2 つに分けられる。転置式暗号は、文字の位置を入れ替えることで読めなくするものである。換字式暗号は、現在広く使われている暗号の基礎となるもので、文字の位置は変えることなく、別のものに変換するものである。

換字式暗号は、コード暗号とサイファーに分類され、戦時中日本で使われていた紫暗号はコード暗号である。例えば、

00001:第一艦隊
01234:撤退
55555:せよ
⋮

という表を作成しておき、
「第一艦隊 撤退 せよ」という情報を伝達したいときに、
「00001 01234 55555」という数字の羅列を送れば、表の内容を知らない敵は解読することは困難であるという性質を利用した暗号である。
シーザー暗号はサイファーに分類され、文字（数字）を文字（数字）に置き換えるものである。具体的には、例えば、
「abc」という文字を送りたいときに、1 つずつアルファベットをずらし、
「bcd」という風に置き換えて送る。1 文字ではなく、2 文字、3 文字……とずらすものも考えられる。
現在使用されている暗号はこのサイファーと呼ばれるものが多数を占めている。

・秘密鍵暗号方式と公開鍵暗号方式

秘密鍵暗号方式は送信者と受信者しか知らない鍵（秘密鍵）を所有している。先の例のシーザー暗号の場合、ずらす文字数が鍵となり得る。送信者はお互いが持っている鍵の数値分だけずらしてから送信する。受信者は、何文字ずらして送られているのか知っているため元に戻す（復号する）ことができる。
しかし、この方法では秘密鍵をどのようにして知らせるのかが大きな問題となっていた。ときには、物理的に人の手によって運ばれることもあった。そこで、新たに考え出されたのが、公開鍵暗号方式と呼ばれるものである。これは、暗号化に使用する鍵と、復号に使用する鍵が異なるもので、暗号化に使用する鍵は公開されており、誰でも暗号化して送信することができる。しかし、復号に使用する鍵は、受信者しか所持しておらず、通信内容が傍受されたとしても、解読される心配は少ない。公開鍵暗号方式の有名なものとして、RSA 暗号が存在する。

以下に RSA 暗号の暗号化と復号の手順を説明する。

1. 準備

受信者は以下の手順で公開鍵・秘密鍵を生成する。
素数 p,q を選び、N=pq とする。また、L=lcm(p-1,q-1)とする。lcm(a,b)は a と b の最小公倍数を意味する。次に、gcd(E,L)=1 となる E を選ぶ。L と互いに素な E を選ぶことを意味する。
このときの E と N を公開鍵として公開する。
次に、(E×D) mod L = 1 となる D を探す。
このときの D と N を秘密鍵とする。この鍵があると復号ができてしまうため、絶対に公開してはならない。

2. 暗号化

送信者は
暗号文 = (平文)^{E} mod N
とする。A mod N は A を N で割った余りを意味する。

3. 復号

受信者は
平文 = (暗号文)^{D} mod N
とすることで復号できる。

ただし、平文は N より大きな値であってはならない。もし平文が N より大きい数字であったと場合、N で割った余りが結果として復号されてしまうため正しく伝えることができない。一般には、鍵は 1024 ビット以上の長さであり、平文は文字ごとに区切って文字コードを元にして暗号化されるためこの問題は生じない。

実際に RSA 暗号の手順で暗号化・復号を行うプログラムを作成して、検証してみた。検証に使用したプログラム類は最後に添付してある。

p=13,q=19 の場合

```
$cat open.key
5 247
$ cat secret.key
29 247
```

```
-----plain text-----
hoge piyo 1234
ID:kwansei
password:ksc2015

-----plain number----
104 111 103 101 32 112 105 121 111 32 49 50 51 52 10 73 68 58 107 119 97
110 115 101 105 10 112 97 115 115 119 111 114 100 58 107 115 99 50 48 49
53 10 10 10 10

-----encrypt number------
130 232 12 43 223 177 79 49 232 223 121 46 90 143 212 99 178 210 217 123
184 2 20 43 79 212 177 184 20 20 123 232 95 237 210 217 20 112 46 3 121 40
212 212 212 212

-----decode text-----
hoge piyo 1234
ID:kwansei
password:ksc2015
```

プログラムでは、p,q を元にして鍵をファイルに書き出している。open.key の値で暗号化をおこない、secret.key の値で復号をおこなう。
文字ごとに、改行や空白文字も含めて文字コードに変換して、RSA 暗号を適用している。

次に、暗号文と平文の計算に繰り返し二乗法を用いることで、比較的大きな桁数のキーであっても計算できるようにした。

p=6481,q=8087 の場合

```
$ cat open.key
7 52411847
$ cat secret.key
3742663 52411847
```

```
-----plain text-----
hoge piyo 1234
ID:kwansei
password:ksc2015

-----plain number-----
104 111 103 101 32 112 105 121 111 32 49 50 51 52 10 73 68 58 107 119 97
110 115 101 105 10 112 97 115 115 119 111 114 100 58 107 115 99 50 48 49
53 10 10 10 10

-----encrypt number------
28244640 5655897 81708 18223042 29978583 1631920 9036572 5792129
5655897 29978583 13772669 51420465 15033517 13323623 10000000
29408437 16380048 30288983 41105418 26059840 23907372 7847005
19444453 18223042 9036572 10000000 1631920 23907372 19444453
19444453 26059840 5655897 37883730 30338645 30288983 41105418
19444453 31501943 51420465 3244025 13772669 4413026 10000000
10000000 10000000 10000000

-----decode text-----
hoge piyo 1234
ID:kwansei
password:ksc2015
```

復号のときに 10^8 のオーダの暗号化された数値の 3742663 乗の計算をすることになるが、剰余を求めさえすれば良いので、比較的早く計算できた。

p,q が素数で無い場合にも検証してみた。

p=13,q=40

-----plain text-----
hoge piyo 1234
ID:kwansei
password:ksc2015

-----plain number-----
104 111 103 101 32 112 105 121 111 32 49 50 51 52 10 73 68 58 107 119 97 110 115 101 105 10 112 97 115 115 119 111 114 100 58 107 115 99 50 48 49 53 10 10 10 10

-----encrypt number------
104 271 103 381 392 112 105 361 271 392 329 280 51 312 160 73 48 288 347 279 457 80 475 381 105 160 112 457 475 475 279 271 264 120 288 347 475 99 280 328 329 53 160 160 160 160

-----decode text-----
hoge piyo 1?38?IH?kwa?sei?passwo?h?ksc?015????1,6c1
< hoge piyo 1234
< ID:kwansei
< password:ksc2015
<
<
<

> hoge piyo 1?38?IH?kwa?sei?passwo?h?ksc?015????
¥ No newline at end of file

p=12,q=40

```
-----plain text-----
hoge piyo 1234
ID:kwansei
password:ksc2015




-----plain number-----
104 111 103 101 32 112 105 121 111 32 49 50 51 52 10 73 68 58 107 119 97
110 115 101 105 10 112 97 115 115 119 111 114 100 58 107 115 99 50 48 49
53 10 10 10 10

-----encrypt number------
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1

-----decode text-----
1,6c1
< hoge piyo 1234
< ID:kwansei
< password:ksc2015
<
<
<
---
>
¥ No newline at end of file
```

以上のように、うまく復号できないことが分かった。他にも試してみたところ、p と q が互いに素でない場合、暗号文が全て 1 となり、うまく復号されないということがわかった。ただし、p と q が互いに素な数の場合は一部はうまく復号されている。

・実行時間の比較

公開鍵・秘密鍵が求まっている状態での繰り返し二乗法を用いた暗号化と復号に要した合計時間を計測した。

p=13.q=19 の場合(N=pq=247)

```
$ time ./check.sh
real    0m0.040s
user    0m0.012s
sys     0m0.020s
```

p=6481,q=8087 の場合(N=pq=52411847)

```
$ time ./check.sh
real    0m0.062s
user    0m0.034s
sys     0m0.020s
```

赤で示している部分が、プログラム自体の処理時間である。これを比較すると、N=pq の値が 20 万倍程度違うにもかかわらずそれほど違いがない。
復号の際に、愚直に計算していった場合には、O(D)の計算時間が必要になるが、このアルゴリズムでは、O(logD)で計算が済むことになる。
前者は D=29 で、後者が D=3742663 であることを考えると、
log29=4.85798
log3742663=21.8356
となり、4 倍程度の差となることが予想される。実際には 3 倍程度であるが、これは、この計算時間には暗号化に要した時間も含まれている。暗号化も同じく O(logE)となるが、E=3,E=7 とほとんど値が変わらないことから、暗号化については所要時間に大きな違いが見られないことが要因と考えられる。

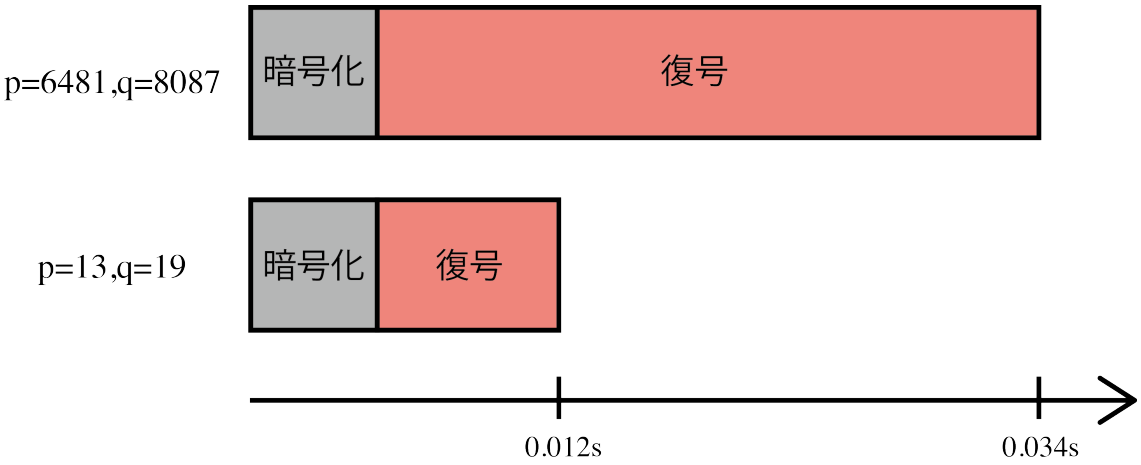

図 2 暗号化・復号に要した時間の内訳の予測図

2.5 暗号化技術の利点

RSA 暗号の暗号化は公開鍵を用いて誰でもおこなうことができるが、復号には秘密鍵が必要であり、受信者のみが復号することができる。また、公開鍵から秘密鍵が推定できないことが重要である。RSA 暗号では、大きな桁数の素因数分解が困難であることを利用して鍵を生成している。

2.6 暗号化技術の限界

現在、RSA 暗号は大きな 2 つの素数の積は素因数分解が困難であるという信念に基づいて使われている。2 つの素数 p,q の積である N は公開鍵として公開されている。これをもとにして p と q を求めることができれば解読されてしまう。将来、効率的な素因数分解の手法の発見、コンピュータの性能の向上、並列化技術の発展によって、現実的な時間で RSA 暗号に使われているレベルの数であっても素因数分解されてしまう可能性がある。実際、RSA 社が開催した、「RSA Factoring Challenge」において、2009 年段階で 768 ビットの素因数分解が実現できている。ただし、300 台の PC を並列処理することによって 3 年間かかっているため実用的では無いだろう。ムーアの法則に陰りが見えるとはいえどもコンピュータの性能は向上しているため、今後 1024 ビットなどの素因数分解も実現されてしまうかもしれない。

・素因数分解の所要時間計測

ここでは、素因数分解にかかる所要時間について実際の計測をもとに考察してみる。

まずは愚直に、入力 n に対して、2 から$\sqrt{n}$ まで順番に割ることによって素因数分解を試みる。

```
52411847
6481 8087

user	0m0.001s
```

先ほどの RSA 暗号で使用した値が一瞬にして素因数分解されてしまった。公開鍵の情報をもとにして簡単に解読されてしまうことがわかる。

```
99942028404137
9997093 9997109

user	0m0.161s
```

およそ 10^7 の素数同士の積、$N=10^{14}$ 程度であっても 1 秒とかからずに計算されてしまう。

型の最大値の問題を解消するために、python を使用して計測する。
先ほどと同じ数値に対しては、

```
99942028404137
9997093
9997109

user	0m1.643s
```

以上のように、C++に比べて、およそ 10 倍の計算時間がかかることが分かった。

・$10^8 \times 10^8$ の場合

```
10000004400000259
100000007
100000037

user	0m16.364s
```

先ほどより桁数をそれぞれ 1 桁、積では 2 桁増やしたところ、およそ 10 倍となった。

・$10^9 \times 10^9$ の場合

```
1000000016000000063
1000000007
1000000009

user	3m2.060s
```

エラトステネスのふるいやポラード・ロー素因数分解法などの効率の良いアルゴリズムが考え出されているが、根本的な手法は変わらない。27 ビット×27 ビット＝54 ビットの計算でこれだけの時間がかかっていることから考えると、RSA 暗号で使用されている 1024 ビットの素因数分解を現実的な時間で行うことは現時点では難しいのでは無いかと考える。しかも、現在 1024 ビットの RSA 暗号は米国標準規格から外されており、2048 ビットや 4096 ビットのものが用いられている。

解読防止のためにビット数を増やすということは秘密鍵を知っている自身にとっても復号に時間がかかることになる。全ての通信内容に対して RSA 暗号を適用するのではなく、最初に触れた秘密鍵暗号方式に使う秘密鍵の配送に RSA 暗号を使用することで、復号による計算量を減らすことができる。

また、RSA 暗号は素因数分解を行うこと以外に解読の方法が無いことが証明されていない。素因数分解を行うことは不可能だと仮定しても他の方法で解読されてしまうかもしれないという問題点が存在する。NTT が開発した EPOC という暗号は素因数分解の効率的な解法が見つからない限り解読されないことが証明されている（http://www.ntt.co.jp/news/news98/9804/980416.html）。しかし、

数十年使用されている RSA 暗号の効率的な解読方法が見つかっていない（少なくとも公にはなっていない）ことから RSA 暗号も素因数分解の効率的な解法が存在しない限り解読されることはないと思われる。

3 感想

実際にプログラムを作成することで、現在使われている暗号化技術の安全性を確かめることができた。また、良いアルゴリズムを使えば計算時間が大幅に短縮されるということを実感することができた。レポートでは触れることができなかったが、エルガマル暗号や楕円曲線暗号やハッシュ関数などのさらに奥が深い暗号の分野を垣間見ることができ、興味をもつことができた。また、整数論など、数学の理論に基づいて暗号が作られていることを知ることができ、コンピュータの分野で実際に数学が役に立っていることを実感できた。これからコンピュータを専門的に扱っていく分野の人として、現在使われているセキュリティ技術を知っておくことは大事なことだと考える。

また、数学だけではなく、物理に基づいた量子暗号なるものも存在しており、暗号化技術とひとくちに言っても様々な学問領域の知識が用いられているのだと感じた。

C の組み込み型だけでは、大きな桁数の演算を行うことができなかったことがくやまれる。多倍長演算のライブラリを使用するなどして、さらに長い鍵についても暗号化・復号の検証したいと考える。

4 文献一覧

1. 黒澤馨『現代暗号への招待』(サイエンス社・2010 年 9 月 10 日)
2. 中村次男・笠原宏『パソコンで実習しながら学べる 暗号のしくみと実装』(日本理工出版会・2009 年 6 月 30 日)
3. 福井幸男『情報システム入門-社会を守る暗号セキュリティ編-』(日科技連出版社・2010 年 12 月 1 日)
4. 石井茂『量子暗号 絶対に盗聴されない暗号を作る』

5 添付(プログラム)

5.1 シェルスクリプト

- make.sh

  C++ファイルをまとめてコンパイル

- check.sh

  p,q から鍵を求め、平文を暗号化し、復号し diff を取る処理を連続して実行

- check_with_error.sh

  check.sh の内容を実際の暗号文などを表示しながら確認できる

- en.sh

  公開鍵と平文から暗号化する処理をまとめて実行

- de.sh

  秘密鍵と暗号文から平文に復号する処理をまとめて実行

make.sh

```
#!/bin/bash
g++ -o key_generate key_generate.cpp
g++ -o encoding encoding.cpp
g++ -o decoding decoding.cpp
```

check.sh

```
#!/bin/bash
./key_generate <pq.txt
./en.sh
./de.sh
diff plain.txt decode.txt
```

check_with_error.sh

```
#!/bin/bash
./key_generate <pq.txt
echo -e "¥n-----plain text-----"
cat plain.txt
./en.sh
echo -e "¥n-----encrypt number-------"
cat encript.txt
./de.sh
echo -e "¥n¥n-----decode text-----"
cat decode.txt
diff plain.txt decode.txt
```

en.sh

```
#!/bin/bash
cat open.key plain.txt | ./encoding   >encript.txt
```

de.sh

```
#!/bin/bash
cat secret.key encript.txt | ./decoding >decode.txt
```

5.2 プログラムリスト

- key_generate.cpp
  p,q から公開鍵と秘密鍵を生成
- encoding.cpp
  公開鍵と平文から暗号文を生成
- decodeing.cpp
  秘密鍵と暗号文から平文を生成
- prime_factor.cpp
  入力された整数に対して素因数分解をする
- prime_factor.py
  入力された整数に対して素因数分解をする

key_generate.cpp

```
#include<iostream>
#include<string>
#include<cstdio>
#include<cmath>
#include<fstream>
#include<string>
using namespace std;

int gcd(int a,int b){
   while(true){
      if(a<b){
         int t=a;
         a=b;
         b=t;
      }
      if(a%b==0){return b;}
      else{
         a=a%b;
      }
   }
}

int lcm(int a,int b){
   return a*b/gcd(a,b);
}

int main(){
   int p,q;
   cin>>p>>q;

   int n=p*q;
```

```
    int l=lcm(p-1,q-1);

    int e;
    for(e=2;e<l;e++){
        if(gcd(e,l)==1){break;}
    }

    int d;
    for(d=1;d<n;d++){
        if((e*d)%l==1){break;}
    }

    ofstream ofs("open.key");
    ofs<<e<<' '<<n<<endl;

    ofstream ofs2("secret.key");
    ofs2<<d<<' '<<n<<endl;
}
```

encodeing.cpp

```
#include<iostream>
#include<string>
#include<cstdio>
#include<cmath>
using namespace std;

long long mypowtwo(long long y){
    long long ret=1;
    if(y==0)return 1;
    for(long long i=0;i<y;i++){
        ret*=2;
    }
    return ret;
}

long long modulo(long long x,long long e,long long n){
    long long binary[100]={};
    long long mod[100]={};
    long long s=1,j;
    for(j=0;s<e;s*=2,j++){}

    long long ss=s,jj=j,ee=e;
    for(;ee>0;jj--){
        if(ee>=mypowtwo(jj)){ee-=mypowtwo(jj);binary[jj]=1;}
    }

    for(long long i=0;i<j;i++){
        if(i==0){mod[i]=x;}
        else{
            mod[i]=(mod[i-1]*mod[i-1])%n;
        }
    }

    long long sum=1;
    for(long long i=0;i<j;i++){
        if(binary[i]==1){
            sum=(sum*mod[i])%n;
        }
    }
    return sum%n;
}
```

```
long long gcd(long long a,long long b){
   while(true){
      if(a<b){
         long long t=a;
         a=b;
         b=t;
      }
      if(a%b==0){return b;}
      else{
         a=a%b;
      }
   }
}

long long lcm(long long a,long long b){
   return a*b/gcd(a,b);
}

int main(){

   //encryption
   long long e,n;
   cin>>e>>n;
   getchar();
    cerr<<"-----plain number-----"<<endl;

   char s;
   long long enc;

   while(true){

      s=getchar();

      if(s==EOF){break;}
      else{
         cerr<<(int)s<<' ';
         enc=modulo((long long)s,e,n);
         cout<<enc<<' ';
      }
   }
   cerr<<endl;
}
```

decoding.cpp

```
#include<iostream>
#include<string>
#include<cstdio>
#include<cmath>
using namespace std;

long long mypowtwo(long long y){
   long long ret=1;
   if(y==0)return 1;
   for(long long i=0;i<y;i++){
      ret*=2;
   }
   return ret;
}

long long modulo(long long x,long long e,long long n){
   long long binary[100]={};
   long long mod[100]={};
   long long s=1,j;
   for(j=0;s<e;s*=2,j++){}
```

```
    long long ss=s,jj=j,ee=e;
    for(;ee>0;jj--){
        if(ee>=mypowtwo(jj)){ee-=mypowtwo(jj);binary[jj]=1;}
    }

    for(long long i=0;i<j;i++){
        if(i==0){mod[i]=x;}
        else{
            mod[i]=(mod[i-1]*mod[i-1])%n;
        }
    }

    long long sum=1;
    for(long long i=0;i<j;i++){
        if(binary[i]==1){
            sum=(sum*mod[i])%n;
        }
    }
    return sum%n;
}

long long gcd(long long a,long long b){
    while(true){
        if(a<b){
            long long t=a;
            a=b;
            b=t;
        }
        if(a%b==0){return b;}
        else{
            a=a%b;
        }
    }
}

long long lcm(long long a,long long b){
    return a*b/gcd(a,b);
}

int main(){
    long long d,n;
    cin>>d>>n;

    long long num;
    long long dec;

    while(cin>>num){
        dec=modulo(num,d,n);
        cout<<(char)dec;
    }
}
```

prime_factor.cpp

```
#include<iostream>
#include<cmath>
using namespace std;
int main(){

    long long n;
    cin>>n;
    for(long long i=2;i<sqrt(n)+1;i++){
        if(n%i==0){cout<<i<<' '<<n/i<<endl;return 0;}
    }
```

```
    cerr<<n<<" is a prime number."<<endl;
}
```

prime_factor.py

```
# -*- coding: utf-8 -*-
import math
import sys

n = int(raw_input())
for i in xrange(int(math.sqrt(n)+1)):
        if n%(i+2)==0:
                print (i+2)
                print n/(i+2)
                sys.exit()
```